KLEBER DE OLIVEIRA ANDRADE

MASMORRA ASCII

APRENDA DART CONSTRUINDO UM ROGUELIKE NO TERMINAL

Masmorra ASCII

Aprenda Dart construindo um roguelike no terminal

Kleber de Oliveira Andrade

Clube de Autores
2026

Edição e texto: Kleber de Oliveira Andrade
Revisão: Kleber de Oliveira Andrade
Capa | Ilustração: Nano Banana AI
Capa | Fechamento: Nano Banana AI
Diagramação: Kleber de Oliveira Andrade
Ilustrações Internas: Nano Banana AI

ISBN: 978-65-00-XXXXX-X

Clube de Autores

Licença

Uso dos exemplos de código

Os trechos de código-fonte Dart reproduzidos ao longo desta obra são disponibilizados sob a licença **MIT** e podem ser livremente estudados, modificados e incorporados em projetos pessoais, educacionais e comerciais, respeitando os termos dessa licença. A citação da obra e do autor é apreciada, mas não obrigatória.

O repositório oficial com o código completo da *Masmorra ASCII em Dart*, incluindo todas as versões incrementais de cada capítulo, está disponível publicamente e segue a mesma licença MIT para o código. Os textos, capítulos, ilustrações e material didático do livro permanecem sob direitos autorais reservados.

Marcas registradas

Os nomes de produtos, tecnologias e empresas mencionados ao longo do livro (Dart, Flutter, Google, GitHub, entre outros) são marcas registradas de seus respectivos proprietários e são citados apenas com finalidade didática, sem qualquer intenção de infringir seus direitos.

Contato

Correções, sugestões e relatos de erratas podem ser enviados pelos canais indicados na seção *Sobre o autor*.

Dedicatória

Para quem aprende melhor quando há um dragão no fim do corredor, mesmo que ele seja feito apenas de pontos e traços, símbolos e linhas de código que respiram na tela escura.

Sumário

Prefácio

No princípio não havia mapa. Havia só o cursor, aquele ponto de luz insolente que pisca num retângulo negro como quem pergunta “e agora?” sem qualquer piedade. O terminal não perdoa vírgulas mal postas nem parênteses que nunca fecham; é o fundo da cova antes de existir masmorra, o vazio absoluto antes de haver sequer uma sala. E no entanto (e aqui está o milagre) é justamente aqui que a história começa a dobrar para o impossível. Esse vazio é fértil. Linha a linha, caractere a caractere, nasce um mundo inteiro: corredores que não estavam no papel, ouro que não brilha mas pesa na conta, inimigos que não têm rosto mas têm letra própria, masmorras que nunca se repetem porque o algoritmo recusa ser fotografia, recusa ser padrão repetido.

Esta é a promessa de *Masmorra ASCII*: não um cemitério de lições numeradas e esquecidas, não um manual que encosta na prateleira após a última página. É uma descida em espiral ao coração de uma linguagem moderna, com um jogo de texto como seu mapa, como sua bússola, como troféu brilhando no fim do corredor mais profundo. Cada capítulo é um degrau que desce; cada degrau, um patamar seguro onde você pode parar, rodar o programa e constatar que algo mudou, não na imaginação ou na teoria, mas no executável que suas mãos acabam de compilar.

Há uma tradição entre programadores que os manuais acadêmicos raramente nomeiam em voz alta: aprender fazendo algo que realmente importa. Não o “exemplo 7.3” que ilustra uma sintaxe isolada e morre esquecido na página seguinte, mas um sistema vivo que reage às suas escolhas, que se quebra quando você o quebra de propósito, que devolve um erro que parece um insulto pessoal e que, quando finalmente você compreende o porquê, levanta-se outra vez, mais forte, mais sábio, mais capaz do que antes. *Masmorra ASCII* segue essa tradição ao pé da letra: o jogo não é um cenário de fundo opcional; é o coração que bate em cada `if`, o propósito de cada classe, a razão de cada teste.

Se alguma vez você se sentiu perdido entre tutoriais que prometem tudo e chegam ao fim deixando você suspenso no vazio, este livro propõe o oposto radical: um círculo que fecha a cada marco conquistado, um artefato que você pode mostrar a alguém e dizer “eu fiz isso”; código que deixa rastro no Git e símbolos no terminal. A jornada é longa (longa de verdade) e cansativa

em seus momentos. A recompensa, porém, não é uma medalha digital, é o conhecimento profundo de que você não apenas construiu o labirinto; você aprendeu a navegá-lo.

Desça, então. O primeiro `print` já é a primeira tocha acesa contra o escuro de tudo que ainda não sabe.

Notas do autor

O que te espera no fundo da masmorra

Você vai aprender Dart construindo um roguelike funcional no terminal: exploração com campo de visão, combate tático por turnos, geração procedural de mapas, economia dinâmica, progressão através de múltiplos níveis, e uma interface completa feita exclusivamente com caracteres ASCII. Sem motor gráfico de terceiros, sem framework de jogo pronto, sem atalhos opacos que escondem como as coisas realmente funcionam. Tudo aquilo que aparece na tela está lá porque *você escreveu o código* que o coloca, linha após linha. E quando algo falha (e vai falhar) o stack trace aponta precisamente para você, para o seu código, que é exatamente onde um programador em formação precisa estar para crescer.

Para quem é este livro

Foi escrito para quem já deu os primeiros passos em alguma linguagem de programação: alguém que entende variáveis, condicionais e funções, mas ainda não domina Dart ou nunca montou um jogo do zero ao teclado. Se você já escreveu um "Hello World" em Python, JavaScript, C# ou Ruby, está no lugar certo. Se veio do Flutter e só conhece Dart através do padrão dos widgets e da reatividade, vai descobrir a linguagem em seu habitat natural: rápida, tipada com rigor, exigente com o `null`, e terrivelmente satisfatória quando o `dart analyze` fica verde de saúde.

Você não precisa (absolutamente não precisa) de formação em design de jogos, algoritmos avançados, ou padrões de projeto memorizados em livros. Tudo isso emerge quando o jogo o exige, não como teoria isolada ou exercício escolar, mas como ferramenta concreta para construir o próximo marco jogável, o próximo degrau descido.

Como o livro está organizado

O livro divide-se em seis partes, cada uma com um marco funcional que você pode executar, testar e mostrar a alguém com genuíno orgulho:

A Parte I (capítulos 1 a 7) assenta os fundamentos sólidos de Dart enquanto você constrói uma aventura textual viva: salas conectadas, itens que você manipula, comandos que o computador compreende. No fim desta jornada inicial, você navega entre locais reais, manipula estado que persiste, e o loop de jogo está fechado e estável.

A Parte II (capítulos 8 a 14) introduz orientação a objetos em profundidade e transforma o projeto num MUD-lite genuíno: combate tático por turnos, ouro que você acumula, armas que você equipa e desequipa, vários tipos de inimigo cada um com comportamentos e estratégias distintas.

A Parte III (capítulos 15 a 21) leva todo o mundo para uma grade bidimensional em puro ASCII: geração procedural de mapas que nunca são iguais, campo de visão realista, névoa de guerra que envolve o desconhecido, e um dungeon crawl completo descendo através de múltiplos andares sucessivos.

A Parte IV (capítulos 22 a 27) adiciona os sistemas sofisticados que transformam o jogo em experiência completa e coerente: economia funcional, loja interativa, progressão de nível com benefícios tangíveis, um boss final que assusta, e uma run inteira jogável do primeiro movimento até a vitória ou morte.

A Parte V (capítulos 28 a 32) é engenharia de verdade, refatoração profissional: organização modular, testes unitários que você escreve e que protegem seu código, async/await para operações não bloqueantes, e persistência com JSON para salvar a progressão entre sessões (sim, você pode sair e voltar).

A Parte VI (capítulos 33 a 37) aplica padrões de projeto clássicos à IA inimiga e à arquitetura do código: Strategy para comportamentos variáveis, Command para fila de ações, Factory para criação de entidades, Observer para sistemas de eventos, e State para máquinas de estados. Termina com uma síntese profunda que deixa você não apenas pronto para explorar o ecossistema oficial em `dart.dev`, mas também preparado para o salto natural rumo ao Flutter, se esse for o seu próximo andar na torre.

Configuração do terminal

A Masmorra ASCII depende do terminal para sua apresentação visual: caracteres especiais, barras coloridas (com ANSI), arte ASCII. Para uma experiência ideal, você precisa de um terminal bem configurado.

Windows: Recomendamos **Windows Terminal** (versão moderna e rápida, disponível na Microsoft Store). O `cmd.exe` antigo e o PowerShell le-

gado não suportam completamente ANSI ou caracteres box-drawing. Para habilitar suporte a ANSI, nenhuma configuração adicional é necessária no Windows Terminal moderno. Se quiser usar PowerShell, prefira a versão 7+.

macOS: O **Terminal.app** padrão funciona bem, mas **iTerm2** oferece renderização superior e melhor suporte a cores e caracteres especiais. Qualquer um dos dois renderizará a arte ASCII corretamente.

Linux: Praticamente todos os terminais funcionam (GNOME Terminal, Konsole, Alacritty). Nenhuma configuração especial é necessária. Se quiser máxima performance e renderização limpa, use Alacritty.

Fonte: Use uma **fonte monoespaçada** de qualidade. Recomendações: JetBrains Mono, Fira Code, Consolas ou DejaVu Sans Mono. Evite fontes proporcionais, pois elas quebram o alinhamento de arte ASCII.

Teste rápido: Execute `dart run` no projeto. Se ver saída colorida (se suportada) e caracteres box-drawing alinhados (╔═╗║), sua configuração está correta. Se os caracteres ficarem desalinhados ou ANSI não aparecer, verifique a fonte e o suporte ANSI do seu terminal.

O repositório de código

Todo o código-fonte acompanha este livro com etiquetas Git (`step-01` até `step-37`), uma por capítulo, tornando fácil comparar estados em diferentes marcos da jornada. Você pode explorar diferenças entre etapas, voltar quando algo der inesperadamente errado, ou clonar um ponto específico e rodar antes de escrever uma linha sequer, apenas para entender. O pacote vive em `code/masmorra_ascii/`, estruturado com testes automatizados, análise estática rigorosa, e linting configurado conforme o livro prega.

Como usar este livro

A experiência ideal é com o terminal aberto ao lado da leitura, lado a lado, sem distrações. Cada capítulo segue um ritmo cuidadosamente planejado: uma cena narrativa breve que motiva o conceito a vir; a explicação clara em Dart com exemplos concretos; a integração desse conceito no jogo vivo; desafios da masmorra para fixar e aprofundar o aprendizado; um boss final no fim de cada capítulo, um desafio maior que testa tudo que você aprendeu.

Não pule os desafios. Quebre-se neles. Não copie sem ler. Leia o código antes de rodar. Não tenha medo de quebrar o build: o Git e o compilador são seus aliados mais chatos e, paradoxalmente, mais úteis para aprender.

Soluções dos desafios

As soluções completas de todos os desafios e boss finals propostos neste livro, juntamente com o código-fonte íntegro de todos os 37 passos da jornada de desenvolvimento da Masmorra ASCII, estão disponíveis online. Acesse o site abaixo para acompanhar sua própria evolução, comparar suas implementações com as soluções de referência, aprofundar seu entendimento através de múltiplas abordagens, e continuar aprendendo na criação de jogos com Dart.

Site: https://masmorra.io **Código-fonte no GitHub:** https://github.com/kleberandrade/masmorra-ascii-dart

No site você encontrará muito mais: errata viva e correções sugeridas pela comunidade que estuda conosco, artigos profundos sobre tópicos que o livro toca em superfície, desafios bônus para expandir suas habilidades além dos capítulos, e comunidades onde se conectar com outros desenvolvedores Dart, pessoas que estão na mesma descida que você, construindo seus próprios roguelikes e compartilhando criações.

A aventura na Masmorra ASCII não termina quando você fecha este livro. Visite https://masmorra.io, explore as soluções, discuta com a comunidade, e continue evoluindo como programador e construtor de mundos Dart.

Boa descida. Que o seu `dart run` nunca lhe falte coragem, curiosidade e a teimosia necessária para depurar o impossível até que ele funcione.

Agradecimentos

Antes de mais nada: a você que está virando esta página, deslizando a tela ou abrindo o arquivo onde quer que este livro o tenha encontrado. Sem leitores curiosos, um manual técnico é só tinta e bits; com você, torna-se conversa viva. Obrigado por arriscar tempo e atenção numa descida que não promete atalhos, apenas degraus que descem de verdade e conhecimento suficiente para aprender no caminho.

À minha família, que aguentou horas em que o "só mais um parágrafo" virava madrugada, que dividiu a mesa com commits e dúvidas, e que me lembrou, com paciência, que há vida para além do terminal, vida que vale a pena voltar a viver depois de fechar o notebook.

Aos amigos que ouviram ideias pela metade, riram das analogias exageradas, perguntaram "mas isso funciona mesmo?" com ceticismo genuíno, e alguns até testaram o jogo quando ainda era um rascunho de letras na tela. A paciência de vocês, os "continua" nos momentos certos e a honestidade das críticas fizeram toda a diferença.

À comunidade Dart e Flutter, aos mantedores de bibliotecas, aos autores de tutoriais pacientes, às respostas cuidadosas nos fóruns, e aos desenvolvedores de ferramentas abertas (Pandoc, XeLaTeX, Python e toda essa companhia generosa), sem a qual este livro jamais ganharia asas em PDF, EPUB ou DOCX.

Um gesto de profundo respeito à tradição dos roguelikes e dos MUDs em texto puro: mundos que provaram, décadas atrás, que uma tela escura pode brilhar mais intensamente do que muitos polígonos coloridos jamais conseguirão.

E a quem leu rascunhos cuidadosamente, apontou incoerências sem piedade, ou simplesmente disse "e depois?" no momento exato em que eu precisava escutar: obrigado por transformarem um projeto solitário em diálogo genuíno. Essa conversa está em cada página.

Prólogo

A tela pisca. Preta. Está escura há séculos, ou talvez desde esta manhã, quando o computador foi desligado pela última vez. O tempo não passa no monitor quando ninguém está olhando. Você se senta. A cadeira range sob o seu peso. Os dedos encontram as letras familiares do teclado, aquelas teclas que seus músculos já decoraram há anos de prática, e você digita como tantas vezes antes, o comando que torna tudo real, que desperta o silício do repouso:

```
dart run
```

Só isso. Duas palavras. Uma linha. O compilador logo começaria seu trabalho.

Mas desta vez, desta vez, algo é diferente. O cursor não para no ponto de espera habitual, naquele piscar monótono que significa "tudo bem, estou aqui, esperando". A tela não mostra a mensagem tranquilizadora de sucesso que você conhece. Em vez disso, caracteres começam a cair pela janela do terminal como chuva de texto, gotas de código tornando-se visíveis. Símbolos que você não solicitou explicitamente. Paredes feitas de cerquilhas (`#`) despontam na tela. Chão que respira em pontos (`.`), vivo, pulsando. As linhas se organizam em padrão, e quando você pisca, quando permite que seus olhos se focalizem, você percebe que não está olhando para um log de execução. Está olhando para um mapa. Não apenas seu código. Um mundo. Uma masmorra inteira.

No centro daquela grade ASCII, ali no coração da tela, onde o cursor do editor deveria estar piscando, há um símbolo que não deveria estar lá: um arroba, simples e arrogante, um `@`. Você move a mão para o mouse, para sair, para fechar a janela e voltar à segurança da ignorância. Mas os dedos hesitam. Ou talvez não queiram obedecer. A masmorra tem peso. A masmorra é real.

Você toma fôlego profundo. Naquele momento, você compreende algo fundamental: aquilo que você construiu em código (linha por linha, função após função, estrutura sobre estrutura) está agora diante de você em sua forma viva. E, num estalo, programar não significa mais abstrair o mundo

em símbolos textuais. Significa estar *dentro* dele. Você não está escrevendo um jogo de um lado de fora. Não está observando. Você *é* o jogo. Cada variável que você declarar será um poder que possuirá. Cada função que você chamar será um passo literal que dará. Cada `if` será uma encruzilhada que o salva da morte ou o enterra no chão úmido da masmorra.

A tela mostra pouco: as paredes ao seu redor desenhadas em cerquilhas, o vazio e a segurança das salas em pontos, e você, tão pequeno, tão simples, feito de apenas uma letra, no meio de tudo. Nenhuma instrução. Nenhuma tela de tutorial condescendente. Nenhuma certeza, nenhuma promessa de que há saída. Há apenas a descida que começou, e a lembrança vaga de que, se você aprender rápido, muito rápido, quem sabe chegará ao fundo vivo.

Ou construirá algo que valha a pena conquistar na volta.

Você respira fundo. Segura ar nos pulmões. Os dedos tocam as setas do teclado com a delicadeza de quem toca um instrumento pela primeira vez. O `@` se mexe. Você se mexe. A masmorra responde (responde!) como se estivesse viva, como se o houvesse estado esperando. A tela se redesenha. Inimigos aparecem nas adjacências. O ouro brilha (em amarelo ASCII). Pela primeira vez, você compreende na profundidade dos ossos: não é o roguelike que foi criado para você o observar e aprender. É você que está sendo *criado* pela masmorra, moldado por ela, um símbolo por vez, um comando por vez, um nível descendente de cada vez.

A descida começa. E desta vez, você não pode voltar atrás.

Não quer voltar atrás.

Rode o primeiro comando. A primeira tocha acesa contra a escuridão.

PARTE I

A PRIMEIRA TOCHA

O terminal é o calabouço das máquinas. O cursor, a sua única tocha. Mas neste escuro, você descobrirá que toda jornada começa com um primeiro passo — uma atribuição, uma função, uma verdade fundamental. Acenda-a.

Capítulo 1 - Seu primeiro programa Dart

O terminal pisca. Você digita o primeiro comando e algo acontece: letras surgem na tela, obedecendo à sua vontade. Não é mágica, é Dart. Mas a sensação é parecida. Nesta primeira descida, você vai aprender a falar a língua da masmorra. Variáveis guardam informações como baús guardam ouro. Tipos definem o que cabe em cada baú. Funções são feitiços que você invoca com um nome e parênteses. Tudo aqui é novo, e a masmorra sabe disso, então os corredores são largos e os inimigos, poucos.

Ao final desta parte, você terá um jogo de texto rodando no terminal. Simples, sim. Mas funcional. E o mais importante: escrito por você, linha por linha, entendendo cada palavra. A descida começa devagar, mas cada passo conta.

> *Você empurra a porta de pedra. Ela range, revelando um corredor iluminado por tochas. No chão, alguém gravou uma inscrição:* `void main() {}`*. É o começo de tudo.*

Antes de construir masmorras, criar monstros ou gerar mapas procedurais, precisamos de uma base sólida. Um projeto Dart que compila e executa. Este capítulo não é sobre o jogo, é sobre garantir que a ferramenta obedece a você. Quando terminar, terá um programa rodando, saberá o que cada arquivo faz, e terá os hábitos que vão acompanhar todo o resto do livro.

O que é Dart e por que usá-lo num roguelike

Dart é uma linguagem criada pelo Google, conhecida principalmente por ser a base do Flutter. Mas Dart não é apenas Flutter. É uma linguagem completa, com tipagem estática, **null safety** nativo, compilação ahead-of-time (AOT) e just-in-time (JIT), e um ecossistema maduro de pacotes.

Para um *roguelike* no terminal, Dart oferece vantagens reais. A tipagem estática captura erros antes de o programa rodar, e num jogo com dezenas de classes interagindo (jogador, inimigos, itens, salas), isso evita bugs que só apareceriam em runtime. O null safety, que vamos explorar no Capítulo 4, impede aquele crash clássico de tentar acessar algo que não existe. E a sintaxe é limpa o suficiente para que o código seja legível mesmo durante o aprendizado.

Um *roguelike* clássico combina morte permanente (*permadeath*), exploração de dungeon, combate por turnos, e **geração procedural** para criar infinitos mundos únicos.

Além disso, tudo que aprender aqui se aplica diretamente ao Flutter. Quando terminar o livro, a transição para interfaces gráficas será natural, a lógica do jogo e os padrões já estarão prontos.

Instalando o Dart SDK

O primeiro passo é garantir que o Dart SDK está instalado. Acesse dart.dev/get-dart e siga as instruções para o seu sistema operacional.

No Windows, a forma mais simples é usar o Chocolatey:

```
choco install dart-sdk
```

No macOS, com Homebrew:

```
brew tap dart-lang/dart
brew install dart
```

No Linux (Debian/Ubuntu):

```
sudo apt-get update
sudo apt-get install dart
```

Se você já tem o Flutter instalado, o Dart vem junto. O comando `dart` provavelmente já está disponível no seu terminal.

Depois de instalar, abra um terminal e confirme que tudo está funcionando:

```
dart --version
```

Você deve ver algo como:

```
Dart SDK version: 3.11.3 (stable) on "linux_x64"
```

Este livro assume Dart 3.11.3 ou superior. Se sua versão for mais antiga, atualize antes de continuar. Vamos usar recursos como *pattern matching* e *sealed classes* que só existem a partir do Dart 3. Para experimentar sem instalar, use **DartPad** (dartpad.dev), um IDE online para Dart.

Criando o projeto

Dart tem uma ferramenta de linha de comando que gera a estrutura inicial. Para uma aplicação de console, usamos o template `console`:

```
dart create -t console masmorra_ascii
```

Esse comando cria uma pasta `masmorra_ascii` com a seguinte estrutura:

```
masmorra_ascii/
├── analysis_options.yaml
├── bin/
│   └── masmorra_ascii.dart
├── lib/
│   └── masmorra_ascii.dart
├── pubspec.yaml
├── README.md
└── test/
    └── masmorra_ascii_test.dart
```

Vamos entender cada peça que vai acompanhar todo o desenvolvimento.

O arquivo **pubspec.yaml** é o coração administrativo do projeto. Ele declara o nome do pacote, a versão do SDK, e as dependências. Por enquanto está quase vazio, vamos adicionar coisas à medida que precisarmos.

A pasta `bin/` contém um ponto de entrada gerado automaticamente pelo `dart create`. Em projetos maiores, esse arquivo serve como inicializador fino que delega para o código em `lib/`. No nosso caso, vamos simplificar: todo o código do jogo, incluindo a função `main()`, ficará em `lib/main.dart`. Isso

mantém tudo num lugar só enquanto aprendemos, e é o padrão que vamos seguir em todo o livro.

A pasta `lib/` é onde ficará todo o código do jogo: a função `main()`, classes, funções utilitárias, modelos de dados. À medida que o projeto crescer, vamos criar mais arquivos dentro de `lib/` para organizar melhor, mas o ponto de entrada será sempre `lib/main.dart`.

A pasta `test/` é para testes automatizados. Vamos usá-la bastante mais adiante, mas é bom saber que ela existe desde o início.

O arquivo `analysis_options.yaml` configura a análise estática, as regras que o Dart usa para apontar problemas no código antes mesmo de executá-lo.

Agora entre na pasta do projeto:

```
cd masmorra_ascii
```

A função `main`, onde tudo começa

Crie o arquivo `lib/main.dart`. Este será o programa principal do nosso jogo durante todo o livro. Escreva:

```
// main.dart
void main() {
  print('Bem-vindo à Masmorra ASCII!');
  print('Prepare-se para explorar o desconhecido.');
}
```

Vamos analisar linha por linha.

`void main()` é a função principal do programa. Toda aplicação Dart começa aqui. A palavra `void` indica que a função não retorna nenhum valor. O nome `main` é especial, é o ponto de entrada que o Dart procura quando você executa o programa. Os parênteses vazios indicam que, por enquanto, não precisamos receber argumentos da linha de comando.

`print()` é a função que escreve texto no terminal. Tudo que você passar entre os parênteses aparece como uma linha na saída. A string (texto) fica entre aspas simples, essa é a convenção em Dart, embora aspas duplas também funcionem.

Cada instrução termina com ponto e vírgula (`;`). Dart é uma linguagem onde o ponto e vírgula é obrigatório. Se esquecer, o analisador vai reclamar.

Executando pela primeira vez

Antes de executar, precisamos resolver as dependências do projeto. Mesmo que não tenhamos nenhuma dependência externa por enquanto, esse passo é necessário:

```
dart pub get
```

Você verá algo como:

```
Resolving dependencies...
Got dependencies!
```

Agora, execute o programa:

```
dart lib/main.dart
```

Se tudo correu bem, o terminal mostra:

```
Bem-vindo à Masmorra ASCII!
Prepare-se para explorar o desconhecido.
```

Esse é o seu primeiro programa Dart compilado e executado com sucesso. Pode parecer pouco, duas linhas de texto, mas o mecanismo por trás é poderoso. O Dart leu o código-fonte, verificou se havia erros de tipo, compilou para uma representação intermediária, e a máquina virtual Dart executou as instruções. Todo esse pipeline vai funcionar silenciosamente a cada `dart lib/main.dart` que você fizer daqui para frente.

Expandindo o programa, variáveis e interpolação

Vamos dar um passo além. Altere o `main` para usar uma variável:

```
// main.dart
void main() {
  var nomeJogo = 'Masmorra ASCII';
  var versao = 1;

  print('');
  print('$nomeJogo, versão $versao');
  print('');
  print('Prepare-se para explorar o desconhecido.');
  print('');
}
```

Aqui aparecem três conceitos novos.

Variáveis com `var`. `var nomeJogo = 'Masmorra ASCII'` cria uma **variável** chamada `nomeJogo` e armazena o texto `'Masmorra ASCII'` nela. O Dart infere automaticamente que o **tipo** é `String`. Depois da atribuição, `nomeJogo` só pode guardar textos; a tipagem é estática, mesmo usando `var`. Da mesma forma, `versao` é inferido como `int` (número inteiro).

Interpolação de strings. O cifrão (`$`) dentro de uma string permite inserir o valor de uma variável diretamente no texto. `'$nomeJogo, versão $versao'` produz `'Masmorra ASCII, versão 1'`. Para expressões mais complexas, usamos chaves: `'${2 + 3}'` produz `'5'`.

Caracteres especiais. Os caracteres = são caracteres Unicode que formam linhas. Dart suporta Unicode nativamente, o que é ótimo para arte ASCII.

Execute novamente com `dart lib/main.dart` e veja a saída formatada:

```


Masmorra ASCII, versão 1

Prepare-se para explorar o desconhecido.
```

Está começando a parecer um jogo.

Analisar e formatar, os dois hábitos essenciais

Dois comandos vão acompanhar você pelo resto do livro. Internalize-os agora, porque quanto mais cedo virarem hábito, menos tempo você vai perder caçando bugs.

dart analyze examina o código sem executá-lo. Ele procura erros de tipo, variáveis não usadas, imports desnecessários e dezenas de outros problemas. Execute agora:

```
dart analyze
```

Se o código estiver correto, você verá:

```
Analyzing masmorra_ascii...
No issues found!
```

Vamos provocar um erro de propósito para ver o que acontece. Adicione esta linha no `main`:

```
int x = 'texto';  // tipo errado!
```

Execute `dart analyze` novamente:

```
Analyzing masmorra_ascii...

  error - A value of type 'String' can't be assigned to a variable
          of type 'int' - lib/main.dart:3:11

1 issue found.
```

O analisador encontrou o problema antes mesmo de você tentar executar. Ele diz exatamente qual é o erro, em qual arquivo e em qual linha. Remova a linha com erro e siga em frente.

`dart format .` reformata todo o código do projeto segundo as convenções oficiais de Dart. Indentação, espaçamento, quebras de linha, tudo fica padronizado:

```
dart format .
```

Não existe discussão sobre tabs vs espaços em Dart. O formatter decide, e todo mundo segue. Isso é libertador, você escreve do jeito que quiser, roda o formatter, e o resultado é sempre limpo.

O ciclo que vai se repetir a cada capítulo é: escrever, `dart analyze`, `dart format`, `dart lib/main.dart`. Decore isso.

O arquivo pubspec.yaml em detalhe

Abra o arquivo `pubspec.yaml`. Ele deve estar assim:

```
name: masmorra_ascii
description: A sample command-line application.
version: 1.0.0

environment:
  sdk: ^3.11.0

dev_dependencies:
  lints: ^5.0.0
  test: ^1.24.0
```

`name` é o identificador do pacote. `description` é uma descrição curta. `version` segue o versionamento semântico (major.minor.patch).

`environment.sdk: ^3.11.0` significa este projeto precisa do Dart SDK versão 3.11.0 ou superior, mas inferior a 4.0.0. O acento circunflexo (`^`) indica compatibilidade com versões futuras dentro da mesma major version.

`dev_dependencies` são pacotes usados apenas durante o desenvolvimento, não vão junto quando alguém usa o seu pacote como biblioteca. `lints` fornece regras de análise estática e `test` é o framework de testes.

Vamos personalizar um pouco:

```
name: masmorra_ascii
description: Um *roguelike* em ASCII construído com Dart puro.
version: 0.1.0

environment:
  sdk: ^3.11.0

dev_dependencies:
  lints: ^5.0.0
  test: ^1.24.0
```

Mudamos a descrição para algo que faz sentido para o nosso projeto e a versão para `0.1.0`, ainda estamos na fase inicial.

Configurando a análise estática

Abra o arquivo `analysis_options.yaml`. Vamos configurá-lo para ser rigoroso desde o início:

```
include: package:lints/recommended.yaml

analyzer:
  language:
    strict-casts: true
    strict-inference: true
    strict-raw-types: true
```

Essas três flags ativam o modo mais exigente do analisador Dart. `strict-casts` proíbe conversões implícitas de `dynamic` para tipos concretos. `strict-inference` exige que o Dart consiga inferir o tipo de toda expressão. `strict-`

`raw-types` proíbe usar tipos genéricos sem especificar o parâmetro (como `List` em vez de `List<String>`).

Pode parecer excessivo para um projeto simples, mas essas regras vão nos salvar de bugs quando o jogo ficar complexo. Melhor acostumar desde o primeiro capítulo.

Polindo o banner

Nosso programa funciona, mas o banner está nu demais. Vamos usar os caracteres box-drawing do Unicode para dar um ar de jogo ao terminal. Substitua o conteúdo de `lib/main.dart`:

```
// main.dart
void main() {
  var nomeJogo = 'Masmorra ASCII';
  var versao = '0.1.0';

  print('');
  print('═══════════════════════════════════════');
  print('       $nomeJogo v$versao');
  print('═══════════════════════════════════════');
  print('');
  print('  Prepare-se para explorar');
  print('  o desconhecido.');
  print('');
  print('═══════════════════════════════════════');
  print('');
}
```

Execute `dart lib/main.dart` e veja o banner:

```

═══════════════════════════════════
       Masmorra ASCII v0.1.0
═══════════════════════════════════

```

```
  Prepare-se para explorar
  o desconhecido.

═══════════════════════════════════
```

O jogo ainda não é um jogo, é uma placa na porta da masmorra. Mas a fundação está sólida: o projeto existe, compila sem erros, e você já sabe como analisar e formatar o código.

Observe que todo nosso código vive em `lib/main.dart`. À medida que o jogo crescer, vamos criar mais arquivos em `lib/` para organizar classes e funções, mas o ponto de entrada será sempre `lib/main.dart`. O arquivo `lib/masmorra_ascii.dart`, gerado pelo `dart create`, serve como declaração da biblioteca:

```dart
/// Masmorra ASCII, um *roguelike* em Dart puro.
///
/// Biblioteca principal do jogo.
library;
```

Ele ficará útil quando começarmos a importar módulos entre arquivos.

Desafios da Masmorra

Desafio 1.1. Personalize o banner. Modifique o banner para incluir o seu nome como autor. Use uma nova variável `autor` e interpolação de string para exibi-la. Dica: use múltiplas linhas de `print()` para deixar o banner legível.

Desafio 1.2. Explore o dart analyze. Introduza três erros diferentes no código (tipo errado, variável não usada, ponto e vírgula faltando) e veja como o `dart analyze` os reporta. Depois corrija todos. Observe como o analisador ajuda você a encontrar problemas sem executar o programa.

Desafio 1.3. Moldura com caracteres box-drawing. Reescreva o banner usando os caracteres ╔, ╗, ╚, ╝, ═ e ═ para criar uma moldura completa. A moldura deve ter largura fixa de 40 caracteres. Execute e veja o resultado alinhado.

Desafio 1.4. Múltiplas linhas com quebra de linha. Em vez de usar vários `print()`, tente criar uma única string multilinha usando `\n` ou aspas triplas (`'''`). Execute e compare qual abordagem você acha mais legível no código.

Boss Final 1.5. ASCII art** de portal mágico.** Crie uma arte ASCII de um portal mágico ou de uma inscrição antiga na parede da masmorra, usando apenas `print()`. Comece simples (5-10 linhas) e incremente. Teste caracteres especiais: ◆, ◊, ✦, ✧ para efeitos visuais. O objetivo é dominar a saída no terminal e entender como texto visual funciona num *roguelike*.

Dica do Mestre: Comece com caracteres simples como `*`, `#` e `-` para desenhar a forma geral, depois refine com box-drawing como ╔, ═, ═. Linhas de symmetry ajudam: desenha metade, depois copia e inverte.

Pergaminho do Capítulo

Neste capítulo você instalou o Dart SDK, criou um projeto com `dart create`, entendeu a estrutura de pastas e definiu `lib/main.dart` como o ponto de entrada do jogo. Escreveu e executou seu primeiro programa, aprendeu os dois hábitos essenciais (`dart analyze` e `dart format`), e viu como variáveis e interpolação de strings funcionam em Dart.

No Capítulo 2, vamos ligar o programa ao teclado: o jogador vai poder digitar comandos, e o programa vai responder. É o primeiro passo para transformar um programa estático num jogo interativo.

Dica do Mestre: Use um editor com suporte a Dart. VS Code com a extensão Dart, IntelliJ com o plugin Dart, ou até o Android Studio. Esses editores rodam `dart analyze` automaticamente enquanto você digita, mostram erros sublinhados em vermelho, oferecem auto-complete e permitem navegar entre definições. Escrever Dart num editor sem suporte é como explorar uma masmorra sem tocha: possível, mas desnecessariamente difícil.

Capítulo 2 - Conversando com o terminal

O aventureiro se aproxima da inscrição na parede. Letras brilham: "Diga seu nome e a porta se abrirá." Ele digita. A masmorra escuta.

No capítulo anterior, o programa falava mas não ouvia. Mostrava um banner bonito e saía. Isso é um pôster, não um jogo. A partir de agora, o programa vai ler o que o jogador digita, processar a entrada e responder. Essa ida e volta, input, processamento, output, é o alicerce de qualquer programa interativo, e de todo jogo que já existiu.

Saída, o que já sabemos e um pouco mais

Você já conhece a função `print()`, que escreve uma linha no terminal e pula para a próxima. Dart oferece outra função de saída que é útil em jogos:

```
import 'dart:io';

void main() {
  stdout.write('Qual é o seu nome? ');
}
```

A diferença entre `print()` e `stdout.write()` é sutil mas importante. A função `print()` sempre adiciona uma quebra de linha no final; o cursor vai para a linha seguinte. `stdout.write()` escreve o texto e mantém o cursor na mesma linha. Isso é ideal para mensagens de entrada (prompts) em que queremos que o jogador digite na mesma linha da pergunta:

```
Qual é o seu nome? _
```

Em vez de:

```
Qual é o seu nome?
_
```

Para usar `stdout.write()`, precisamos importar `dart:io`, a biblioteca de entrada e saída do Dart. A linha `import 'dart:io';` deve ficar no topo do arquivo, antes de tudo. **stdout** e **stdin** são objetos do módulo `dart:io` para escrever e ler do terminal.

Para ler o que o jogador digita, usamos `stdin.readLineSync()`:

Entrada, ouvindo o jogador

```dart
import 'dart:io';

void main() {
  stdout.write('Qual é o seu nome? ');
  var nome = stdin.readLineSync();
  print('Bem-vindo, $nome!');
}
```

Execute com o comando `dart lib/main.dart`:

```
Qual é o seu nome? Aldric
Bem-vindo, Aldric!
```

O programa parou, esperou você digitar algo e pressionar Enter, capturou o texto, guardou na variável `nome`, e usou interpolação para incluí-lo na mensagem de boas-vindas.

Mas há um detalhe importante. O tipo retornado por `stdin.readLineSync()` não é `String`, é `String?` (**nullable**). Isso significa que o valor pode ser uma string ou `null` (nada). O Dart nos avisa disso porque existem situações em que `readLineSync()` pode não conseguir ler nada; por exemplo, se a entrada padrão for redirecionada de um arquivo vazio.

Para lidar com esse risco de `null`, usamos o operador `??`:

```
import 'dart:io';

void main() {
  stdout.write('Qual é o seu nome? ');
  var nome = stdin.readLineSync() ?? 'Aventureiro';
  print('Bem-vindo, $nome!');
}
```

O operador `??` é o "operador de coalescência nula": se o valor à esquerda for `null`, usa o valor à direita. Então se `readLineSync()` retornar `null`, `nome` será `'Aventureiro'`. Vamos explorar null safety em profundidade no Capítulo 4. Por enquanto, pense no operador `??` como uma rede de segurança contra o nada.

Funções, organizando o código

Conforme o programa cresce, colocar tudo dentro do `main` vira uma bagunça rápido. **Funções** são a primeira ferramenta de organização, blocos de código com nome, que fazem uma coisa específica e podem ser chamados de qualquer lugar.

Funções que não retornam nada (void)

Vamos criar uma função que exibe o banner do jogo:

```
void exibirBanner() {
  print('');
  print('╔══════════════════════════════════════╗');
  print('║       MASMORRA ASCII v0.1            ║');
  print('╚══════════════════════════════════════╝');
  print('');
}
```

`void` indica que a função não retorna nada, ela faz algo (imprime no terminal) mas não produz um valor. `exibirBanner` é o nome que escolhemos. Os parênteses vazios indicam que a função não recebe parâmetros.

Para chamar a função, basta escrever o nome com parênteses:

```
void main() {
  exibirBanner();
}
```

Funções que retornam um valor

Agora uma função que retorna um valor:

```
String pedirNome() {
  stdout.write('Como devo chamá-lo? ');
  var nome = stdin.readLineSync() ?? 'Aventureiro';
  return nome.trim();
}
```

Aqui o tipo de retorno é `String`. A função recebe input do jogador e devolve o nome como texto. A instrução `return` diz qual é o valor que a função produz. O método `.trim()` remove espaços em branco no início e no final do texto, um cuidado que evita problemas se o jogador digitar `"  Aldric "` com espaços acidentais.

Quando chamamos essa função, podemos guardar o resultado numa variável:

```
var nome = pedirNome();
```

Funções com parâmetros

Funções também podem receber valores de entrada, chamados parâmetros:

```
void saudar(String nome) {
  print('Bem-vindo à Masmorra, $nome!');
  print('Que os dados estejam ao seu favor.');
```

```
}
```

O parâmetro `nome` é declarado com seu tipo (`String`) dentro dos parênteses. Quando chamamos a função `saudar('Aldric')`, o valor `'Aldric'` é passado para `nome` dentro da função.

Funções podem ter múltiplos parâmetros:

```
void exibirStatus(String nome, int nivel, int ouro) {
  print('$nome, Nível $nivel, $ouro moedas de ouro');
}
```

Juntando tudo

Vamos usar todas essas funções juntas:

```
// main.dart
import 'dart:io';

void exibirBanner() {
  print('');
  print('MASMORRA ASCII v0.1.0');
  print('');
}

String pedirNome() {
  stdout.write('Como devo chamá-lo? ');
  var nome = stdin.readLineSync() ?? 'Aventureiro';
  return nome.trim();
}

void saudar(String nome) {
  print('');
  print('Bem-vindo à Masmorra, $nome!');
  print('Que os dados estejam ao seu favor.');
```

```
  print('');
}

void main() {
  exibirBanner();
  var nome = pedirNome();
  saudar(nome);
}
```

Execute:

```
MASMORRA ASCII v0.1.0

Como devo chamá-lo? Aldric

Bem-vindo à Masmorra, Aldric!
Que os dados estejam ao seu favor.
```

Repare como o `main` agora é limpo e legível. Três linhas que contam uma história: mostrar banner, pedir nome, saudar. Os detalhes ficam dentro de cada função. Essa clareza é fundamental num projeto que vai crescer para milhares de linhas de código.

Strings em detalhe

Strings são o tipo de dado mais usado num jogo de texto, então vale a pena conhecê-las bem.

Criação com aspas simples e duplas. Ambas são equivalentes em Dart. A convenção em Dart é usar aspas simples:

```
var a = 'texto com aspas simples';
var b = "texto com aspas duplas";
```

Interpolação com cifrão ($). O cifrão insere variáveis. Chaves (`${}`) são necessárias para expressões:

```
var nome = 'Aldric';
var nivel = 3;
print('$nome está no nível $nivel');
print('Próximo nível: ${nivel + 1}');
print('Nome em maiúsculas: ${nome.toUpperCase()}');
```

Strings multilinha com aspas triplas. Permitem texto que ocupa várias linhas:

```
var descricao = '''
Você está numa sala escura.
O ar é úmido e cheira a mofo.
Uma tocha fraca ilumina a parede norte.''';
print(descricao);
```

Métodos úteis de String. Esses métodos vão aparecer repetidamente no jogo:

```
var texto = '  Masmorra ASCII  ';

texto.trim()           // 'Masmorra ASCII', remove espaços nas pontas
texto.toUpperCase()    // '  MASMORRA ASCII  '
texto.toLowerCase()    // '  masmorra ascii  '
texto.contains('ASCII') // true
texto.length           // 19
texto.isEmpty          // false
''.isEmpty             // true

print('=' * 30);       // ==============================
```

O `.trim()` limpa input do jogador. `.toLowerCase()` permite comparar comandos sem se preocupar com maiúsculas/minúsculas. `.contains()` procura

palavras-chave. `.isEmpty` verifica se o jogador pressionou Enter sem digitar nada. A repetição com `*` é perfeita para desenhar molduras ASCII.

Caracteres de escape. Alguns caracteres especiais precisam de barra invertida:

```
print('Linha 1\nLinha 2');
print('Coluna1\tColuna2');
print('Ele disse: \'olá\'');
```

Tipos de dados, a primeira visão

Até agora usamos `var` para declarar variáveis e deixamos o Dart inferir o tipo. Mas é importante saber quais tipos básicos existem em Dart:

```
String nome = 'Aldric';
int nivel = 1;
double vida = 100.0;
bool estaVivo = true;
```

Quando usamos `var`, o Dart determina o tipo automaticamente a partir do valor inicial. O tipo `bool` é o booleano, pode ser `true` ou `false`:

```
var nome = 'Aldric';
var nivel = 1;
var vida = 100.0;
var estaVivo = true;
```

Depois da atribuição inicial, o tipo é fixo. Você não pode fazer `nome = 42`, o Dart vai reclamar porque `nome` é `String`, não `int`. Essa é a tipagem estática em ação: erros de tipo são encontrados antes de o programa rodar.

`final` e `const`: variáveis imutáveis. Quando um valor não deveria mudar, declare com `final`:

```
final nomeJogo = 'Masmorra ASCII';
```

E `const` para valores que são conhecidos em tempo de compilação:

```
const versao = '0.1.0';
const maxVida = 100;
```

A diferença prática: `final` aceita valores calculados em runtime (como o resultado de `pedirNome()`), enquanto `const` exige valores literais. Usaremos `final` na maioria dos casos e `const` para constantes globais do jogo como dano base e HP máximo.

Convertendo entre tipos. Quando o jogador digita um número, ele chega como `String`. Para usá-lo como **double** ou `int`:

```
var texto = '42';
var numero = int.parse(texto);
```

Mas se o texto não for um número válido, `int.parse` causa um erro. A versão segura é `int.tryParse`:

```
var resultado = int.tryParse('abc');
var numero = int.tryParse('42');
```

Vamos usar `tryParse` sempre que o jogador puder digitar algo que não é um número, o que em jogos de texto é o tempo todo.

Aplicação no jogo, o primeiro diálogo interativo

Vamos expandir o programa para algo que já começa a parecer um jogo. O programa vai pedir o nome, apresentar uma descrição de sala, e oferecer opções ao jogador:

```
// main.dart
import 'dart:io';
```

```
void exibirBanner() {
  print('');
  print('MASMORRA ASCII v0.1.0');
  print('Aprenda Dart, conquiste a masmorra');
  print('');
}

String pedirNome() {
  stdout.write('Como devo chamá-lo, aventureiro? ');
  var entrada = stdin.readLineSync() ?? '';
  var nome = entrada.trim();
  if (nome.isEmpty) {
    nome = 'Aventureiro';
    print('Sem nome? Tudo bem, chamarei você de $nome.');
  }
  return nome;
}

void descreverSala(String nome) {
  print('');
  print('═══════════════════════════════════');
  print(' $nome, você está na Praça Central.');
  print('');
  print(' Uma fonte de pedra murmura ao centro');
  print(' da praça. Tochas iluminam três saídas.');
  print('');
  print(' Ao norte: um corredor escuro.');
  print(' A leste: uma porta de madeira.');
  print(' Ao sul: a saída da masmorra.');
  print('═══════════════════════════════════');
  print('');
}

String pedirComando() {
  stdout.write('O que deseja fazer? ');
  var comando = stdin.readLineSync() ?? '';
```

```
  return comando.trim().toLowerCase();
}

void responderComando(String comando) {
  if (comando == 'norte' || comando == 'n') {
    print('Você caminha para o norte...');
    print('O corredor é frio e úmido.');
  } else if (comando == 'leste' || comando == 'l') {
    print('Você empurra a porta de madeira...');
    print('Rangidos ecoam pelo corredor.');
  } else if (comando == 'sul' || comando == 's') {
    print('Você recua para a saída.');
    print('A luz do sol aquece seu rosto.');
  } else if (comando == 'sair') {
    print('Até a próxima aventura!');
  } else {
    print('Não entendi "$comando".');
    print('Tente: norte, leste, sul ou sair.');
  }
  print('');
}

void main() {
  exibirBanner();
  var nome = pedirNome();
  descreverSala(nome);
  var comando = pedirComando();
  responderComando(comando);
}
```

Execute e interaja:

```
MASMORRA ASCII v0.1.0
Aprenda Dart, conquiste a masmorra
```

```
Como devo chamá-lo, aventureiro? Aldric

Aldric, você está na Praça Central.

Uma fonte de pedra murmura ao centro
da praça. Tochas iluminam três saídas.

Ao norte: um corredor escuro.
A leste: uma porta de madeira.
Ao sul: a saída da masmorra.

O que deseja fazer? norte
Você caminha para o norte...
O corredor é frio e úmido.
```

O programa ainda aceita apenas um comando e depois termina. No Capítulo 3, vamos adicionar repetição para que o jogador possa dar vários comandos seguidos.

Repare em dois detalhes importantes no código acima. Primeiro, `pedirComando()` usa `.trim().toLowerCase()`. Isso significa que `"Norte"`, `"NORTE"`, `" norte "` e `"norte"` funcionam da mesma forma. Sempre normalize a entrada do jogador antes de compará-la. Segundo, a função `responderComando()` trata o caso "não entendi" com uma mensagem clara que lista os comandos válidos. Nunca ignore input inválido. Um bom jogo sempre responde ao jogador, mesmo que seja para dizer que não entendeu.

Desafios da Masmorra

Desafio 2.1. Pergunta extra. Depois de pedir o nome, pergunte a classe do personagem (Guerreiro, Mago ou Ladrão) e inclua essa informação na saudação. Crie uma função `pedirClasse()` que retorna `String`. Valide que a entrada é uma das três opções válidas.

Desafio 2.2. Moldura dinâmica. Escreva uma função `exibirEmMoldura(String texto)` que recebe qualquer texto e o exibe dentro de uma moldura com bordas box-drawing. A moldura deve se ajustar dinamicamente

ao tamanho do texto. Dica: use `texto.length` para saber o tamanho e `'=' * n` para repetir o caractere.

Desafio 2.3. Validação de entrada robusta. Modifique `pedirNome()` para recusar nomes com menos de 2 caracteres ou mais de 20. Se o jogador digitar algo fora desse intervalo, mostre uma mensagem clara de erro, sugira um intervalo válido, e peça novamente em vez de usar nome padrão. Dica: use `nome.length` na condição e considere um loop.

Desafio 2.4. Múltiplas salas com navegação. Crie três funções: `descreverPraca()`, `descreverCorredor()`, `descreverPorta()`. Dependendo do comando que o jogador digitar na praça, chame a função correspondente. O programa ainda aceita um único comando e termina, mas a ideia de navegar entre descrições já começa a surgir. Observe como a lógica começa a ficar mais complexa.

Boss Final 2.5. Diálogo com NPC (Velho Sábio). Adicione um comando especial `"falar"` que inicia uma conversa com um NPC chamado Velho Sábio na Praça Central. O Velho Sábio faz uma pergunta ao jogador (por exemplo: "Qual é a sua maior virtude?" com opções "coragem", "sabedoria", "justiça") e responde com uma observação diferente para cada escolha. Integre isso ao fluxo principal: depois de responder, o programa termina com uma mensagem final do Velho Sábio.

Dica do Mestre: Crie uma função `iniciarDialogoVelhoSabio()` que encapsula toda a conversa: lê input do jogador, valida a opção, mostra a resposta. A função `main()` fica limpa, apenas chama essa função quando apropriado.

Pergaminho do Capítulo

Você aprendeu a ler entrada do jogador com `stdin.readLineSync()`, importar bibliotecas com `import 'dart:io'`, criar funções com parâmetros e retornos, manipular strings com métodos como `.trim()` e `.toLowerCase()`, e usar o operador `??` para fornecer valores padrão. O programa agora conversa com o jogador: pede nome, descreve uma sala, e responde a comandos.

No Capítulo 3, você adicionará loops e decisões mais complexas para manter o jogo rodando enquanto o jogador quiser.

Dica do Mestre: Em jogos de texto, a regra de ouro é nunca ignore o input do jogador. Mesmo que o comando não faça sentido, responda. Um simples "Não entendi, tente novamente" é infinitamente melhor que silêncio. O jogador precisa saber que o jogo está ouvindo, caso contrário, vai achar que travou.

Próximo Capítulo

No próximo capítulo, suas variáveis ganham poder de decisão. Com `if`, `else` e `switch`, o jogo começará a reagir às escolhas do jogador. Prepare-se para loops que dão vida ao combate.

Capítulo 3 - Decisões e repetições

> *O menu da taverna está pregado na parede com uma adaga. "1, Cerveja. 2, Hidromel. 0, Sair." O aventureiro escolhe. O taberneiro reage. E o menu continua ali, esperando a próxima escolha.*

Até agora o programa aceita um comando e morre. Isso não é um jogo, é uma máquina de venda automática com uma única moeda. Um jogo de verdade precisa de duas coisas: decisões (reagir de forma diferente a cada input) e repetição (continuar rodando até o jogador decidir parar). Neste capítulo vamos construir as duas, e o resultado será o primeiro loop interativo com **turnos** do nosso *roguelike*.

Decisões com if, `else if`, `else`

Você já viu um `if` rápido no capítulo anterior. Agora vamos entendê-lo por completo.

A estrutura básica é:

```
if (condicao) {
  // código executado se a condição for verdadeira
}
```

A condição é uma expressão que resulta em um `bool`: `true` ou `false`. Se for `true`, o bloco entre chaves executa. Se for `false`, o programa pula para o próximo bloco.

Para múltiplas alternativas, usamos `else if` e `else`:

```
var opcao = 1;

if (opcao == 1) {
  print('Você escolheu explorar.');
```

```
} else if (opcao == 2) {
  print('Você escolheu o inventário.');
} else if (opcao == 0) {
  print('Até a próxima!');
} else {
  print('Opção inválida.');
}
```

O Dart avalia as condições de cima para baixo e executa apenas o primeiro bloco cuja condição for verdadeira. O `else` final é a rede de segurança: pega tudo que não combinou com nenhuma condição anterior.

Operadores de comparação:

```
a == b    // igual a
a != b    // diferente de
a > b     // maior que
a < b     // menor que
a >= b    // maior ou igual
a <= b    // menor ou igual
```

Operadores lógicos para combinar condições:

```
a && b    // E lógico, ambos precisam ser true
a || b    // OU lógico, pelo menos um precisa ser true
!a        // NÃO, inverte o valor
```

Um exemplo prático para o jogo, aceitar sinônimos de comandos:

```
var cmd = 'n';

if (cmd == 'norte' || cmd == 'n') {
  print('Você vai para o norte.');
} else if (cmd == 'sul' || cmd == 's') {
```

```
  print('Você vai para o sul.');
}
```

O switch, alternativa elegante para múltiplas opções

Quando você está comparando uma variável com vários valores fixos, o `switch` pode ser mais claro que uma cadeia de `if`/`else if`:

```
var opcao = 2;

switch (opcao) {
  case 1:
    print('Explorar a masmorra.');
  case 2:
    print('Ver inventário.');
  case 0:
    print('Sair do jogo.');
  default:
    print('Opção inválida.');
}
```

No Dart 3, o `switch` não precisa de `break`: cada `case` termina automaticamente. O `default` funciona como o `else`, captura qualquer valor não listado.

O `switch` também funciona com strings, o que será útil quando construirmos o parser de comandos:

```
var comando = 'norte';

switch (comando) {
  case 'norte' || 'n':
    print('Indo para o norte...');
  case 'sul' || 's':
    print('Indo para o sul...');
  case 'sair':
```

```
    print('Até logo!');
  default:
    print('Comando desconhecido: $comando');
}
```

Repare na sintaxe `case 'norte' || 'n':`. Isso é um pattern do Dart 3 que permite agrupar valores no mesmo case. Muito prático para sinônimos de comandos.

Repetição com while

O `while` executa um bloco de código enquanto a condição for verdadeira:

```
var contador = 1;

while (contador <= 5) {
  print('Contagem: $contador');
  contador++;
}
```

Saída:

```
Contagem: 1
Contagem: 2
Contagem: 3
Contagem: 4
Contagem: 5
```

O `contador++` é uma abreviação de `contador = contador + 1`. Sem ele, o loop nunca terminaria, um loop infinito, que trava o programa.

O loop infinito controlado com break

Para jogos, o padrão mais comum é um `while (true)` com `break`:

```
while (true) {
  // mostrar opções
  // ler input
  // se for "sair", break
  // senão, processar o comando
}
```

O `while (true)` roda para sempre até encontrar um `break`, que sai imediatamente do loop. Esse padrão é limpo e legível: o loop continua rodando, e a condição de saída fica explícita dentro do bloco.

O `continue` é o irmão do `break`: em vez de sair do loop, ele pula o resto do bloco e volta ao início da próxima iteração:

continue, pule para a próxima volta

```
while (true) {
  stdout.write('> ');
  var cmd = stdin.readLineSync() ?? '';
  cmd = cmd.trim().toLowerCase();

  if (cmd.isEmpty) {
    continue;
  }

  if (cmd == 'sair') {
    break;
  }

  print('Você digitou: $cmd');
}
```

Se o jogador pressionar Enter sem digitar nada, o `continue` pula o `print` e volta a pedir input.

Repetição com for

O `for` é ideal quando você sabe quantas vezes quer repetir:

```
for (var i = 1; i <= 5; i++) {
  print('Item $i');
}
```

Lê-se: comece com `i` igual a 1, enquanto `i` for menor ou igual a 5, depois de cada volta, incremente `i` em 1.

O `for` vai ser essencial quando formos desenhar o mapa em grade, percorrer linhas e colunas. Mas por enquanto o `while` é a estrela do capítulo.

De texto para número, int.tryParse

Quando o jogador digita "2" no terminal, o programa recebe o texto `"2"`, não o número `2`. Para usar em comparações numéricas, precisamos converter:

```
var texto = '42';
var numero = int.tryParse(texto);
```

`int.tryParse` é a versão segura: se o texto não for um número válido, retorna `null` em vez de causar um erro:

```
int.tryParse('42')     // 42
int.tryParse('abc')    // null
int.tryParse('')       // null
int.tryParse('3.14')   // null (não é inteiro)
```

O padrão que usaremos sempre:

```
var linha = stdin.readLineSync() ?? '';
var opcao = int.tryParse(linha.trim()) ?? -1;
```

Se o jogador digitar algo que não é número, `opcao` será `-1`, um valor que sabemos que não é nenhuma opção válida.

Aplicação no jogo, o menu interativo

Agora vamos juntar tudo num menu que roda em loop:

```
// main.dart
import 'dart:io';

void exibirBanner() {
  print('');
  print('╔═══════════════════════════════════════╗');
  print('║        MASMORRA ASCII v0.1            ║');
  print('╚═══════════════════════════════════════╝');
  print('');
}

void exibirMenu() {
  print('┌───────────────────────────────────────┐');
  print('│           O QUE DESEJA FAZER?         │');
  print('├───────────────────────────────────────┤');
  print('│  1, Explorar a masmorra               │');
  print('│  2, Ver status do herói               │');
  print('│  3, Ajuda                             │');
  print('│  0, Sair do jogo                      │');
  print('└───────────────────────────────────────┘');
}

void explorar(String nome) {
  print('');
  print('$nome adentra o corredor escuro...');
  print('Tochas fracas iluminam paredes de pedra.');
  print('Você ouve algo se movendo na escuridão.');
  print('(Exploração completa virá nos próximos capítulos.)');
  print('');
```

```
}

void mostrarStatus(String nome) {
  print('');
  print('╔══════════════════════════════════════╗');
  print('║  HERÓI: $nome');
  print('║  HP: 100/100');
  print('║  Ouro: 0');
  print('║  Arma: Nenhuma');
  print('╚══════════════════════════════════════╝');
  print('');
}

void mostrarAjuda() {
  print('');
  print('Masmorra ASCII é um *roguelike* em texto.');
  print('Use os números do menu para navegar.');
  print('Em breve você poderá explorar masmorras,');
  print('lutar contra monstros e coletar tesouros.');
  print('');
}

void main() {
  exibirBanner();

  stdout.write('Como devo chamá-lo? ');
  var nome = (stdin.readLineSync() ?? '').trim();
  if (nome.isEmpty) nome = 'Aventureiro';

  print('');
  print('Bem-vindo, $nome! Sua jornada começa agora.');

  var jogando = true;

  while (jogando) {
    print('');
```

```
    exibirMenu();
    stdout.write('> ');

    var linha = (stdin.readLineSync() ?? '').trim().toLowerCase();

    if (linha.isEmpty) {
      print('Digite uma opção do menu.');
      continue;
    }

    var opcao = int.tryParse(linha);
    if (opcao == null) {
      switch (linha) {
        case 'explorar' || 'jogar':
          opcao = 1;
        case 'status':
          opcao = 2;
        case 'ajuda' || 'help':
          opcao = 3;
        case 'sair' || 'quit':
          opcao = 0;
        default:
          print('Não entendi "$linha". Use os números do menu.');
          continue;
      }
    }

    switch (opcao) {
      case 1:
        explorar(nome);
      case 2:
        mostrarStatus(nome);
      case 3:
        mostrarAjuda();
      case 0:
        jogando = false;
```

```
        print('');
        print('Até a próxima aventura, $nome!');
      default:
        print('Opção $opcao não existe. Escolha entre 0 e 3.');
    }
  }
}
```

Execute e experimente:

```
╔═══════════════════════════════════════╗
║        MASMORRA ASCII v0.1            ║
╚═══════════════════════════════════════╝

Como devo chamá-lo? Aldric

Bem-vindo, Aldric! Sua jornada começa agora.

┌───────────────────────────────────────┐
│           O QUE DESEJA FAZER?         │
├───────────────────────────────────────┤
│  1, Explorar a masmorra               │
│  2, Ver status do herói               │
│  3, Ajuda                             │
│  0, Sair do jogo                      │
└───────────────────────────────────────┘

> 2

╔═══════════════════════════════════════╗
║  HERÓI: Aldric
║  HP: 100/100
║  Ouro: 0
║  Arma: Nenhuma
```

```
╚═══════════════════════════════════════╝

> explorar

Aldric adentra o corredor escuro...
Tochas fracas iluminam paredes de pedra.
Você ouve algo se movendo na escuridão.
(Exploração completa virá nos próximos capítulos.)

> 0

Até a próxima aventura, Aldric!
```

Vamos destacar os pontos mais importantes desse código.

A variável `jogando` é um `bool` que controla o loop. Quando o jogador escolhe sair, `jogando = false` faz com que o `while (jogando)` termine na próxima verificação. Essa é uma alternativa ao `while (true)` com `break`: ambas funcionam bem.

O programa aceita tanto números (`1`, `2`, `0`) quanto palavras (`explorar`, `status`, `ajuda`, `sair`). Primeiro tenta usar `int.tryParse()`; se falhar (resultado `null`), tenta casar a palavra com um `switch`. Essa flexibilidade faz o jogo parecer mais inteligente.

O `continue` aparece em dois lugares: quando o input é vazio e quando a palavra não é reconhecida. Em ambos os casos, o programa pula o processamento e volta ao início do loop.

Variáveis e escopo

Uma sutileza importante: variáveis declaradas dentro de um bloco (entre `{` `}`) existem apenas dentro dele:

```
void main() {
  var nome = 'Aldric';

  if (nome.isNotEmpty) {
    var saudacao = 'Olá, $nome!';
```

```
    print(saudacao);
  }
}
```

Isso se chama escopo. A variável `nome` tem escopo de `main`, é acessível em qualquer lugar dentro da função. A `saudacao` tem escopo do `if`, existe apenas dentro daquelas chaves.

No jogo, as variáveis importantes (nome do jogador, flag de jogo ativo) são declaradas no escopo do `main` para que possam ser usadas em todo o loop. Variáveis temporárias podem ser declaradas dentro do loop, elas são recriadas a cada iteração.

Operador ternário, if compacto

Para condições simples que produzem um valor, existe o operador ternário:

```
var mensagem = vida > 0 ? 'Vivo' : 'Morto';
```

Lê-se: se `vida > 0`, o resultado é `'Vivo'`; caso contrário, é `'Morto'`. É um `if/else` condensado numa expressão. Útil para escolher valores, não para executar lógica complexa:

```
print('HP: ${hp}/${maxHp} ${hp < 20 ? "(PERIGO!)" : ""}');
```

Desafios da Masmorra

Desafio 3.1. Contador de turnos. Adicione uma variável `turno` que começa em 1 e incrementa a cada vez que o jogador escolhe uma opção válida (não sair, não ajuda). Mostre o turno atual no prompt: `"[Turno 5] > "`. Isso simula a passagem de tempo na masmorra.

Desafio 3.2. Menu com mais opções (Ver mapa). Adicione a opção "4, Ver mapa" que imprime um mapa ASCII simples fixo da masmorra. Por

enquanto, o mapa pode ser um retângulo com `#` (paredes) e `.` (chão vazio). Use o mesmo padrão de switch/case para processar a opção.

Desafio 3.3. HP que diminui (Simulando perigo). Declare `var hp = 100;` no início. Cada vez que o jogador escolher explorar, reduza o HP em um valor fixo (por exemplo, 10-20 pontos de dano simulado). Mostre o HP atualizado no status. Se HP chegar a 0 ou menos, encerre o jogo com uma mensagem de derrota. Observação: isso torna o jogo com tempo limitado.

Desafio 3.4. Confirmação ao sair (Dupla verificação). Quando o jogador escolher sair (opção 0), pergunte "Tem certeza (s/n)?" de forma clara. Se digitar s, sim, ou y (yes), saia de verdade. Caso contrário, volte ao menu. Use um loop interno ou um if para capturar essa confirmação.

Boss Final 3.5. Painel de estatísticas finais. Ao final do jogo, após sair confirmado ou morte (HP = 0), exiba um painel com uma tabela mostrando: nome do herói, total de turnos sobrevividos, HP final, dano total sofrido (100 − HP final), e uma nota final (S, A, B, C) baseada em turnos/HP. Use operadores ternários para determinar a nota e box-drawing para tornar o painel visual. Exemplo: se sobreviveu mais de 20 turnos, nota A; entre 10 e 20 turnos, nota B, etc.

Dica do Mestre: Declare uma função `exibirEstatisticasFinais(String nome, int turnos, int hpFinal)` que calcula a nota com operadores ternários aninhados: `var nota = turnos > 20 ? 'A' : turnos > 10 ? 'B' : 'C';`. Depois monta a tabela com `StringBuffer` e box-drawing.

Pergaminho do Capítulo

Você aprendeu `if/else if/else`, `switch/case`, `while` com o padrão `while (true)` e `break`, `for` para repetições contadas, `int.tryParse` para conversão segura, e o operador ternário para condições compactas. O programa agora é um *game loop* funcional: roda continuamente, aceita múltiplos comandos, e trata input inválido.

No Capítulo 4, vamos entender null safety de Dart: por que `readLineSync()` retorna `String?` e como Dart protege você de crashes.

Dica do Mestre: O `while (true)` com `break` e o `while (variavel)` são dois padrões igualmente válidos para *game loops*. Use o que fizer mais sentido para cada situação.

Capítulo 4 - Null safety, o escudo contra crashes

> *O aventureiro abre um baú. Dentro, encontra... nada. Não uma espada, não uma poção, não um mapa, literalmente nada. Em muitas linguagens, tentar usar esse nada causaria um crash. Em Dart, o compilador já teria avisado: "esse baú pode estar vazio. Trate isso antes de enfiar a mão."*

Se você veio de linguagens como JavaScript, Python ou Java, provavelmente já encontrou a infame null pointer exception: um erro que acontece quando o código tenta usar um valor que não existe. É o bug mais comum do mundo, responsável por bilhões de dólares em prejuízo.

Dart resolveu esse problema de forma elegante com o **null safety**: o sistema de tipos distingue entre valores que sempre existem e valores que podem ser nulos. O compilador força você a tratar a possibilidade de nulo antes de o programa rodar. Este capítulo explica como isso funciona e por que vai salvar seu jogo de crashes misteriosos.

Tipos normais vs tipos nullable

Em Dart, por padrão, uma variável não pode ser nula:

```
String nome = 'Aldric';
String nome2 = null;       // ERRO de compilação!
```

Se você quiser que uma variável possa conter `null`, precisa declarar isso explicitamente com `?` no tipo:

```
String? nome = null;
String? nome2 = 'Aldric';
```

Pense assim: `String` é um contrato que diz “aqui sempre haverá texto”. `String?` diz “aqui pode haver texto ou não haverá nada”. O ponto de interrogação é o sinal visual de “cuidado, pode estar vazio”.

Essa distinção existe para todos os tipos:

```
int vida = 100;
int? dano = null;

bool vivo = true;
bool? fugiu = null;

List<String> itens = [];
List<String>? inventario = null;
```

Por que readLineSync retorna String?

Agora faz sentido por que usamos `??` desde o Capítulo 2:

```
var nome = stdin.readLineSync() ?? 'Aventureiro';
```

A função `stdin.readLineSync()` retorna `String?` porque existem situações em que ela genuinamente não consegue ler nada, por exemplo, se o programa receber input redirecionado de um arquivo que acabou. Nesses casos, retornar `null` é mais honesto que retornar uma string vazia.

Da mesma forma, `int.tryParse()` retorna `null` porque uma entrada como `"abc"` não é um número. Retornar `null` é mais informativo que retornar `0` ou lançar uma exceção.

Os quatro operadores de null safety

Dart oferece quatro operadores para trabalhar com valores nullable. Cada um resolve um problema diferente: acessar propriedades com segurança, fornecer valores padrão, atribuir sob condição, ou garantir explicitamente que algo não é nulo. Você vai usar esses operadores o tempo todo daqui para frente.

1. O operador ?., acesso condicional

O `?.` chama um método ou acessa uma propriedade somente se o valor não for `null`. Se for `null`, o resultado inteiro é `null`. Pense nele como uma forma de dizer "se isso existir, me dá aquilo; se não existir, me dá `null` em vez de crashar":

```
String? texto = obterTexto();
var tamanho = texto?.length;
```

Se `texto` for `'Masmorra'`, `tamanho` será `8`. Se `texto` for `null`, `tamanho` será `null`, sem crash.

Você pode encadear vários operadores `?.`:

```
String? descricao = sala?.inimigo?.nome;
```

Se `sala` for null ou `inimigo` for null, o resultado também é null.

2. O operador ??, valor padrão

O `??` fornece um valor substituto quando algo é null. É um operador muito prático: se você tem um valor que pode ser null mas precisa de uma alternativa garantida, o `??` resolve de forma elegante:

```
var nome = entrada ?? 'Aventureiro';
```

Se `entrada` não for `null`, `nome` recebe `entrada`. Se for `null`, recebe `'Aventureiro'`.

O operador `??` pode ser encadeado:

```
var local = salaAtual ?? salaAnterior ?? 'Praça Central';
```

3. O operador ??=, atribuição se nulo

O `??=` é um atalho que combina verificação e atribuição: atribui um valor somente se a variável for `null`, fazendo tudo em uma linha. Você vai vê-lo muito em inicializações padrão:

```
String? apelido;
apelido ??= 'Herói Sem Nome';
apelido ??= 'Outro Nome';
print(apelido);
```

A primeira atribuição funciona porque `apelido` é `null`. A segunda não atribui porque a variável já possui um valor.

4. O operador !, asserção de não-nulo

O `!` é o operador mais perigoso dos quatro. Ele diz ao compilador: "eu tenho certeza de que isso não é `null` neste momento, confie em mim". Se você estiver errado, o programa crasha. Use com extrema cautela:

```
String? texto = obterTexto();
var tamanho = texto!.length;
```

Se `texto` for `null`, o programa lança uma exceção em tempo de execução. Use `!` com extrema cautela: ele desativa exatamente a proteção que null safety oferece. Neste livro, evitaremos `!` sempre que possível.

Promoção de tipo (type promotion)

Uma das funcionalidades mais inteligentes do Dart é a promoção de tipo. Quando você faz uma verificação de null, o Dart automaticamente promove o tipo dentro desse bloco:

```
String? entrada = stdin.readLineSync();
```

```
if (entrada != null) {
  // Aqui dentro, o Dart garante: não há null.
  var tamanho = entrada.length;
  print('Você digitou: ${entrada.trim()}');
}
```

Fora do `if`, `entrada` continua sendo `String?`. Mas dentro do bloco onde verificamos `!= null`, Dart promove o tipo para `String`, e todos os métodos ficam disponíveis sem operadores especiais.

A promoção funciona com vários tipos de verificação:

```
void processar(String? entrada) {
  if (entrada == null) {
    print('Nada digitado.');
    return;
  }
  print('Você disse: ${entrada.toUpperCase()}');
}

void mostrar(Object? valor) {
  if (valor is String) {
    print('Texto com ${valor.length} caracteres');
  } else if (valor is int) {
    print('Número: ${valor * 2}');
  }
}
```

Aplicação no jogo, input robusto

Vamos criar funções de leitura de input que tratam todos os casos de forma limpa:

```
import 'dart:io';
```

```
/// Lê uma linha do terminal, limpa espaços e converte para
↪  minúsculas.
String lerComando() {
  stdout.write('> ');
  var entrada = stdin.readLineSync();

  if (entrada == null) {
    return 'sair';
  }

  return entrada.trim().toLowerCase();
}

/// Tenta interpretar o input como número do menu.
int? interpretarComoNumero(String input) {
  if (input.isEmpty) return null;
  return int.tryParse(input);
}

/// Converte uma palavra em número de menu, se for sinônimo conhecido.
int? interpretarComoPalavra(String input) {
  return switch (input) {
    'explorar' || 'jogar' || 'entrar' => 1,
    'status' || 'heroi' => 2,
    'ajuda' || 'help' => 3,
    'sair' || 'quit' || 'fim' => 0,
    _ => null,
  };
}

/// Interpreta o input do jogador como opção de menu.
int? interpretarInput(String input) {
  return interpretarComoNumero(input) ??
  ↪  interpretarComoPalavra(input);
}
```

Repare como o `null` flui naturalmente pelo código. A função `interpretarComoNumero()` retorna `null` se não for um número. A função `interpretarComoPalavra()` retorna `null` se não for uma palavra conhecida. O operador `??` encadeia as duas tentativas.

```
// main.dart
void main() {
  while (true) {
    var cmd = lerComando();

    if (cmd.isEmpty) {
      print('Digite algo.');
      continue;
    }

    var opcao = interpretarInput(cmd);

    if (opcao == null) {
      print('Não entendi "$cmd".');
      continue;
    }

    switch (opcao) {
      case 1:
        print('Explorando...');
      case 2:
        print('Status...');
      case 3:
        print('Ajuda...');
      case 0:
        print('Até logo!');
        return;
      default:
        print('Opção inválida.');
    }
  }
}
```

Esse código nunca crasha por causa de `null`. Toda possibilidade de valor ausente é tratada explicitamente. Esse é o poder do null safety.

Padrão de leitura segura com validação

Aqui está um padrão que vamos reutilizar ao longo de todo o livro:

```
/// Pede um número ao jogador dentro de um intervalo.
int pedirOpcao(String prompt, int min, int max) {
  while (true) {
    stdout.write(prompt);
    var linha = stdin.readLineSync()?.trim() ?? '';

    if (linha.isEmpty) {
      print('Digite um número entre $min e $max.');
      continue;
    }

    var numero = int.tryParse(linha);
    if (numero == null) {
      print('"$linha" não é um número.');
      continue;
    }

    if (numero < min || numero > max) {
      print('Escolha entre $min e $max.');
      continue;
    }

    return numero;
  }
}
```

Essa função é um mini-loop que só sai quando recebe um número válido dentro do intervalo. Cada tipo de erro tem sua própria mensagem.

Exemplo de uso:

```
var opcao = pedirOpcao('Escolha (0-3): ', 0, 3);
```

O resultado é um `int` garantido entre 0 e 3, sem null, sem surpresas.

O tipo late, inicialização tardia

Há situações em que você sabe que uma variável vai ser inicializada antes de ser usada, mas não consegue dar um valor no momento da declaração:

```
late String nomeDoJogador;

void inicializarJogo() {
  nomeDoJogador = pedirNome();
}

void mostrarHUD() {
  print('Jogador: $nomeDoJogador');
}
```

O `late` diz ao Dart: essa variável será inicializada antes de ser acessada, confie em mim. Use `late` quando a inicialização depende de algo que acontece depois da declaração.

Desafios da Masmorra

Desafio 4.1. Validação de nome robusta. Reescreva `pedirNome()` para recusar nomes com menos de 2 caracteres ou mais de 20. Se inválido, mostre exatamente o motivo (“Muito curto”, “Muito longo”) e peça novamente em vez de usar um padrão. Use promoção de tipo dentro de um `if (entrada != null)` para garantir segurança.

Desafio 4.2. Menu com confirmação bilateral. Crie uma função `confirmar(String mensagem) -> bool` que mostra a mensagem, aceita s/sim/y/yes para verdadeiro e n/não/no para falso. Se o jogador digitar algo inválido, repita a pergunta. Use `??` para proteger `readLineSync()`.

Desafio 4.3. Interpretação de comandos em três níveis. Implemente um parser que reconheça três formas do mesmo comando: numeração (1), palavra completa (explorar) e abreviação (e). Use `int.tryParse()` para tentar número primeiro, depois `??` para tentar palavra, depois `??` para tentar abreviação. Demonstre com explorar/e/1.

Desafio 4.4. Função parametrizada pedirTexto. Escreva `String pedirTexto(String prompt, {int minLength = 1, int maxLength = 50})` com parâmetros nomeados e valores padrão. A função repete até receber um texto com tamanho válido. Use `texto.length` e lance exceção (ou retorne padrão) se sair do intervalo.

Boss Final 4.5. Cadeia de null safety (Sala inicial). Crie um mapa representando três salas: `salaPraca`, `salaCorredo`, `salaTesouraria`, cada uma como `String?`. Algumas salas podem ser `null` (não existem). Implemente um getter `salaAtual() -> String` que usa encadeamento `??` para sempre garantir que o jogador está em uma sala válida, caindo para "Praça Central" se tudo mais for `null`. Demonstre que o encadeamento funciona mesmo com múltiplos níveis de `null`.

Dica do Mestre: A chave é encadeamento: `var atual = salaPraca ?? salaCorredo ?? salaTesouraria ?? 'Praça Central';`. Teste atribuindo `null` a cada uma e observe como a cadeia "cai" para a próxima opção.

Pergaminho do Capítulo

Neste capítulo você aprendeu o que é null safety e por que Dart o implementa, a diferença entre `String` e `String?`, os quatro operadores de null: `?.`, `??`, `??=` e `!`, promoção de tipo em blocos `if`, `late` para inicialização tardia, e padrões reutilizáveis de validação de input.

O código do jogo agora é robusto contra qualquer input do jogador. Nenhuma combinação de Enter vazio, texto aleatório ou números fora do intervalo causa crash. No Capítulo 5, vamos dar memória real ao jogo com coleções: listas para o inventário, mapas para as salas, e conjuntos para itens únicos.

Dica do Mestre: Em Dart, prefira `??` e promoção de tipo ao operador `!`. O `!` é a última opção, não a primeira. Sempre que você escreve `!`, está dizendo "se eu estiver errado, o programa pode crashar". Com `??`, você está dizendo "se estiver vazio, use isso"; o programa nunca crasha. Código de jogo que não crasha é código de jogo que os jogadores respeitam.

Próximo Capítulo

No próximo capítulo, seu herói ganha um inventário de verdade. Listas, mapas e sets organizam itens, inimigos e salas. A masmorra fica maior — e mais perigosa.

Capítulo 5 - Coleções, o inventário do herói

> *O aventureiro abre a mochila. Dentro, uma tocha, uma chave enferrujada e três moedas de ouro. Ele precisa de algo para guardar tudo isso, algo com ordem, algo que associe nomes a coisas, algo que responda rápido: já tenho essa chave?*

Até agora, cada dado do jogo vivia numa variável separada: `nome`, `hp`, `opcao`. Isso funciona para três ou quatro valores, mas um jogo de verdade precisa de listas de itens, mapas de salas, conjuntos de comandos válidos. Neste capítulo, vamos conhecer as três **coleções** fundamentais de Dart: **List**, **Map** e **Set**, e usá-las para dar estrutura real ao jogo.

List, quando a ordem importa

Uma **lista** é uma sequência ordenada de valores. Pense numa fila: o primeiro elemento tem índice 0, o segundo tem índice 1, e assim sucessivamente. No nosso jogo, a lista será essencial para guardar o inventário do jogador, os itens numa sala e a sequência de ações registradas. Quando a ordem importa ou você precisa acessar elementos por posição, use `List`.

```
var inventario = ['Tocha', 'Chave Enferrujada', 'Poção de Vida'];
```

O Dart infere o tipo como `List<String>`, uma lista onde cada elemento é uma `String`.

Acessar elementos por índice:

```
print(inventario[0]);
print(inventario[2]);
print(inventario.length);
```

Cuidado: acessar um índice que não existe causa erro em runtime. Por exemplo, `inventario[5]` crasha se a lista possui apenas 3 elementos.

Adicionar e remover:

```
inventario.add('Espada Curta');
inventario.insert(0, 'Mapa Antigo');
inventario.remove('Tocha');
inventario.removeAt(1);
```

Percorrer a lista:

```
for (var item in inventario) {
  print('- $item');
}

for (var i = 0; i < inventario.length; i++) {
  print('${i + 1}. ${inventario[i]}');
}
```

Verificar conteúdo:

```
inventario.contains('Tocha');
inventario.isEmpty;
inventario.isNotEmpty;
```

Métodos funcionais (muito úteis no jogo):

```
var numeros = [3, 1, 4, 1, 5, 9];

numeros.where((n) => n > 3).toList();
numeros.map((n) => n * 2).toList();
numeros.any((n) => n > 8);
numeros.every((n) => n > 0);

var armas = ['Adaga', 'Espada', 'Machado'];
```

```
var lista = armas.map((a) => '  - $a').join('\n');
print(lista);
```

A sintaxe `(n) => n > 3` é uma arrow function, uma função anônima compacta. Lê-se: para cada `n`, retorne `n > 3`.

Map, quando cada valor tem um nome

Um **mapa** associa chaves a valores, como um dicionário: você procura por uma chave e recebe o valor correspondente. Mapas serão fundamentais no nosso jogo para associar IDs de sala a descrições, nomes de itens a seus preços em ouro ou abreviações a comandos completos. Se você precisa procurar algo por nome e depois recuperar um dado associado, use `Map`.

```
var salas = <String, String>{
  'praca': 'Uma praça com uma fonte de pedra murmurante.',
  'taverna': 'Uma taverna barulhenta.',
  'corredor': 'Um corredor estreito e escuro.'
};
```

O tipo é `Map<String, String>`, chaves `String`, valores `String`.

Acessar valores:

```
var descricao = salas['praca'];
var nada = salas['floresta'];
```

Repare: acessar uma chave que não existe retorna `null`, não um erro. O tipo de retorno é `String?`. O null safety força você a tratar isso:

```
var descricao = salas['praca'] ?? 'Local desconhecido.';
```

Adicionar e remover:

```
salas['portao'] = 'Um portão de ferro oxidado.';
salas.remove('corredor');
```

Percorrer:

```
for (var entrada in salas.entries) {
  print('${entrada.key}: ${entrada.value}');
}

for (var nome in salas.keys) {
  print(nome);
}

salas.containsKey('taverna');
```

Mapa de sinônimos, muito útil para o parser de comandos (interpretador de comandos):

```
var sinonimos = <String, String>{
  'n': 'norte', 's': 'sul', 'l': 'leste', 'o': 'oeste',
  'e': 'leste',
  'i': 'inventario',
};

var input = 'n';
var comando = sinonimos[input] ?? input;
```

Set, quando só importa se existe

Um **conjunto** é como uma lista que não permite duplicatas e não garante ordem. Sua vantagem principal é a velocidade: verificar se um elemento existe num `Set` é muito mais rápido que em uma `List`, pois usa uma tabela de hash internamente. No jogo, usaremos `Set` para rastrear salas visitadas, inimigos derrotados ou conquistas desbloqueadas, quando só importa se algo foi feito ou não, não quantas vezes ou em que ordem.

```
var chavesPossuidas = <String>{'chave_prata', 'chave_ouro'};

chavesPossuidas.contains('chave_prata');
chavesPossuidas.contains('chave_ferro');

chavesPossuidas.add('chave_ferro');
chavesPossuidas.add('chave_prata');
print(chavesPossuidas.length);
```

No jogo, conjuntos são perfeitos para salas já visitadas, inimigos derrotados, conquistas desbloqueadas. Estruturas como **fila** (Queue) e **pilha** (Stack) são especializações: fila é FIFO (primeiro a entrar, primeiro a sair), pilha é LIFO (último a entrar, primeiro a sair). Não as usaremos diretamente neste jogo, mas são fundamentais em algoritmos como BFS ou backtracking.

```
var salasVisitadas = <String>{};

void visitarSala(String id) {
  if (salasVisitadas.contains(id)) {
    print('Você já esteve aqui antes.');
  } else {
    print('Um lugar novo!');
    salasVisitadas.add(id);
  }
}
```

Listas tipadas, o tipo importa

Dart permite especificar o tipo dos elementos:

```
List<String> itens = ['Espada', 'Escudo'];
List<int> danos = [10, 5, 15];
Map<String, int> precos = {'Espada': 100, 'Poção': 30};
Set<String> visitadas = {'praca', 'taverna'};
```

Isso evita misturar tipos por acidente. `itens.add(42)` causa erro de compilação, pois a lista é de `String`, não `int`. Essa segurança de tipos é especialmente valiosa num jogo com centenas de itens. Estruturas de dados são frequentemente visualizadas como **árvores** (hierarquias) ou **grafos** (redes): o mapa da masmorra é, tecnicamente, um grafo, com salas como nós e corredores como arestas.

O operador spread (...)

O **operador spread** (`...`) é uma forma compacta e legível de desempacotar uma coleção dentro de outra. Em vez de manualmente copiar cada elemento de uma lista, você coloca três pontos e deixa o Dart fazer o trabalho. Vamos usar spread o tempo todo para combinar inventários, juntar listas de saídas possíveis ou montar listas de resultado.

```
var basicos = ['Tocha', 'Corda'];
var extras = ['Poção', 'Mapa'];
var todosItens = [...basicos, ...extras];
```

Útil para combinar inventários ou montar listas de opções.

Aplicação no jogo, salas com dados estruturados

Vamos reconstruir o jogo usando coleções para representar o mundo. Cada sala é uma entrada num mapa:

```
// main.dart
import 'dart:io';

final salas = <String, Map<String, dynamic>>{
  'praca': {
    'descricao': 'Você está na Praça Central.\n'
        'Uma fonte de pedra murmura ao centro.\n'
        'Tochas iluminam três passagens.',
    'saidas': {
```

```
      'norte': 'corredor',
      'leste': 'taverna',
      'sul': 'portao'
    },
    'itens': <String>['Chave Enferrujada'],
  },
  'corredor': {
    'descricao': 'Um corredor estreito e frio.\n'
        'As paredes são úmidas. Algo se move na escuridão.',
    'saidas': {'sul': 'praca'},
    'itens': <String>[],
  },
  'taverna': {
    'descricao': 'Uma taverna aconchegante.\n'
        'O cheiro de cerveja e pão fresco preenche o ar.\n'
        'Um velho sábio está sentado no canto.',
    'saidas': {'oeste': 'praca'},
    'itens': <String>['Poção de Vida'],
  },
  'portao': {
    'descricao': 'Um portão de ferro enorme.\n'
        'Além dele, a escuridão absoluta.\n'
        'Você ainda não está pronto para entrar.',
    'saidas': {'norte': 'praca'},
    'itens': <String>[],
  },
};

final sinonimos = <String, String>{
  'n': 'norte', 's': 'sul', 'l': 'leste', 'o': 'oeste',
  'e': 'leste', 'w': 'oeste',
  'i': 'inventario', 'inv': 'inventario',
};

var salaAtual = 'praca';
var inventario = <String>[];
```

```
var salasVisitadas = <String>{};

void exibirSala() {
  var sala = salas[salaAtual] ?? {};
  var descricao = sala['descricao'] as String?;
  var saidasMap = sala['saidas'] as Map<String, String>?;
  var itensNaSala = sala['itens'] as List<String>?;

  var primeira = !salasVisitadas.contains(salaAtual);
  if (primeira) salasVisitadas.add(salaAtual);

  print('');
  if (primeira) {
    print('** Lugar novo! **');
  }
  for (var linha in (descricao ?? '').split('\n')) {
    print(linha);
  }

  var saidasTexto =
      saidasMap?.keys.map((d) => '[$d]').join(' ') ??
      'Sem saídas';
  print('Saídas: $saidasTexto');

  if (itensNaSala != null && itensNaSala.isNotEmpty) {
    var itensTexto = itensNaSala.join(', ');
    print('No chão: $itensTexto');
  }

  print('');
}

void exibirInventario() {
  print('');
  if (inventario.isEmpty) {
    print('Sua mochila está vazia.');
```

```
  } else {
    print('Inventário:');
    for (var i = 0; i < inventario.length; i++) {
      print('  ${i + 1}. ${inventario[i]}');
    }
  }
  print('');
}

void mover(String direcao) {
  var sala = salas[salaAtual] ?? {};
  var saidasMap = sala['saidas'] as Map<String, String>?;

  if (saidasMap != null && saidasMap.containsKey(direcao)) {
    salaAtual = saidasMap[direcao] ?? salaAtual;
    print('Você vai para $direcao...');
    exibirSala();
  } else {
    print('Não há saída para $direcao.');
  }
}

void pegarItem(String nomeItem) {
  var sala = salas[salaAtual] ?? {};
  var itens = sala['itens'] as List<String>?;

  if (itens == null || itens.isEmpty) {
    print('Não há "$nomeItem" aqui.');
    return;
  }

  var encontrado = itens.where(
    (item) => item.toLowerCase() == nomeItem.toLowerCase()
  ).toList();

  if (encontrado.isEmpty) {
```

```
    print('Não há "$nomeItem" aqui.');
  } else {
    var item = encontrado.first;
    itens.remove(item);
    inventario.add(item);
    print('Você pegou: $item.');
  }
}

void main() {
  print('');
  print('MASMORRA ASCII v0.2');
  print('');

  stdout.write('Como devo chamá-lo? ');
  var nome = (stdin.readLineSync() ?? '').trim();
  if (nome.isEmpty) nome = 'Aventureiro';

  print('\nBem-vindo, $nome!');
  exibirSala();

  while (true) {
    stdout.write('\n> ');
    var input = (stdin.readLineSync() ?? '').trim().toLowerCase();

    if (input.isEmpty) continue;

    var partes = input.split(' ');
    var cmd = sinonimos[partes[0]] ?? partes[0];
    var argumento = partes.length > 1
        ? partes.sublist(1).join(' ')
        : '';

    switch (cmd) {
      case 'norte' || 'sul' || 'leste' || 'oeste':
        mover(cmd);
```

```
      case 'inventario':
        exibirInventario();
      case 'pegar':
        if (argumento.isEmpty) {
          print('Pegar o quê? Use: pegar <item>');
        } else {
          pegarItem(argumento);
        }
      case 'olhar':
        exibirSala();
      case 'sair' || 'quit':
        print('\nAté a próxima aventura, $nome!');
        return;
      case 'ajuda':
        print('Comandos: norte, sul, leste, oeste, pegar <item>,');
        print('          inventario, olhar, ajuda, sair');
      default:
        print('Não entendi "$input". '
            'Digite "ajuda" para ver os comandos.');
    }
  }
}
```

Execute e explore:

```
MASMORRA ASCII v0.2

Como devo chamá-lo? Aldric

Bem-vindo, Aldric!

** Lugar novo! **
Você está na Praça Central.
Uma fonte de pedra murmura ao centro.
```

```
Tochas iluminam três passagens.

Saídas: [norte] [leste] [sul]
No chão: Chave Enferrujada

> pegar chave enferrujada
Você pegou: Chave Enferrujada.

> i
Inventário:
  1. Chave Enferrujada

> n
Você vai para norte...

** Lugar novo! **
Um corredor estreito e frio.
As paredes são úmidas. Algo se move na escuridão.

Saídas: [sul]
```

Repare como os mapas e listas tornam o código flexível: adicionar uma nova sala é acrescentar uma entrada em `salas`, não reescrever a lógica. Adicionar um sinônimo é uma linha em `sinonimos`. O mapa de saídas define automaticamente a navegação, sem `if/else` para cada direção.

O uso de `dynamic` no mapa de salas não é ideal. No Capítulo 8, classes resolvem isso de forma tipada. Mas por enquanto, funciona e mostra o poder das coleções.

Desafios da Masmorra

Desafio 5.1. Expandir o mundo (Mais salas). Adicione pelo menos duas salas novas ao mapa: por exemplo, “Câmara do Tesouro” e “Biblioteca Antiga”. Conecte-as ao mundo existente com saídas apropriadas e descrições atmosféricas. Teste navegando até elas e verificando se as saídas funcionam nos dois sentidos.

Desafio 5.2. Largar itens. Implemente o comando `"largar <item>"` que remove um item do inventário e o coloca na sala atual (adicionando à lista de `itens` da sala). Valide que o jogador realmente possui o item antes de largá-lo. Dica: é exatamente o inverso de `pegarItem`.

Desafio 5.3. Limite de inventário com feedback. Adicione um limite de 5 itens máximo no inventário. Se estiver cheio, mostre uma mensagem clara: "Sua mochila está cheia! Você tem 5/5 itens. Largue algo antes de pegar novo item." Use `.length` para verificar.

Desafio 5.4. Sala de tesouro (Múltiplos itens). Adicione uma sala especial "Câmara do Tesouro" com 5 itens valiosos (Moeda de Ouro, Anel de Prata, Diamante, Corrente, Gema). Implemente o comando `"pegar tudo"` que pega todos os itens da sala de uma vez, respeitando o limite do inventário. Se a mochila ficar cheia no meio, mostre quantos pôde pegar.

Boss Final 5.5. Visualizar o mundo (Mapa de adjacência). Crie uma função `exibirMapaMundi()` que imprime um diagrama ASCII mostrando todas as salas conectadas. Por exemplo, usando um formato de árvore ou de grafo simples. Use nomes de salas e setas para mostrar as conexões (Praça →[norte] Corredor, etc). Isso ajuda o jogador a visualizar a topologia do mundo.

Dica do Mestre: Comece simples: imprima cada sala e suas saídas diretas. Próxima evolução: use indentação ou ASCII arrows para mostrar hierarquia. Use `.keys` para iterar sobre as salas e `.entries` para pegar chaves e valores (conexões).

Pergaminho do Capítulo

Neste capítulo você aprendeu `List` para sequências ordenadas, `Map` para associar chaves a valores, `Set` para conjuntos de valores únicos, métodos essenciais como `.add`, `.remove`, `.contains`, `.where`, `.map` e `.join`, a sintaxe de arrow functions, e o operador spread `...`.

O jogo agora tem um mundo real com salas conectadas, itens coletáveis e navegação por comandos de texto. No Capítulo 6, vamos dar um salto visual: construir molduras, barras e arte ASCII usando `StringBuffer`.

Dica do Mestre: Resista à tentação de usar `Map<String, dynamic>` para tudo. É flexível mas perde toda a segurança de tipos. No Capítulo 8, vamos substituir esses mapas por classes tipadas, e muitos bugs potenciais vão simplesmente desaparecer.

Próximo Capítulo

No próximo capítulo, transformamos dados em visual. Com `StringBuffer` e arte ASCII, a masmorra ganha forma na tela do terminal.

Capítulo 6 - Arte ASCII e StringBuffer

O artesão da masmorra não usa pincéis, usa caracteres. Um ╔ no canto, um ═ na lateral, um ═ no topo. Linha por linha, o texto bruto vira interface. Neste capítulo, você se torna o artesão.

O jogo já tem salas, inventário e navegação. Mas a apresentação visual ainda é primitiva: prints soltos, sem moldura consistente, sem alinhamento. Neste capítulo vamos aprender a construir telas ASCII de forma programática com **sprites ASCII** (molduras, barras de HP, caixas de texto e banners), tudo usando **StringBuffer** para montar o desenho eficientemente antes de exibi-lo.

Por que não usar print diretamente?

Até agora usamos `print()` para cada linha de saída. Funciona, mas tem dois problemas conforme o jogo cresce.

O primeiro é performance. Quando seu mapa tiver 20 linhas por 40 colunas e precisar ser redesenhado a cada turno, chamar `print()` 20 vezes causa cintilação visível no terminal. É mais eficiente montar toda a saída numa única string e imprimir de uma vez.

O segundo é organização. Quando você quer construir uma moldura cujo tamanho depende do texto dentro dela, precisa calcular larguras, preencher espaços e alinhar colunas. Fazer isso com múltiplos `print()` é confuso. Com `StringBuffer`, você monta o desenho inteiro em uma única variável e só depois exibe.

StringBuffer, o bloco de construção

A classe `StringBuffer` acumula texto de forma eficiente. Em vez de criar strings intermediárias com `+` (o que é lento porque cria novas strings a cada operação), você vai escrevendo pedaços e no final extrai o resultado como uma única string. Para telas complexas, `StringBuffer` é essencial: você constrói tudo em memória e imprime de uma vez, evitando cintilação no terminal.

```
var buffer = StringBuffer();
buffer.writeln('╔═══════════════════╗');
buffer.writeln('║  Masmorra ASCII   ║');
buffer.writeln('╚═══════════════════╝');
print(buffer.toString());
```

Os dois métodos principais são `write()` (adiciona texto sem quebra de linha) e `writeln()` (adiciona texto com quebra de linha).

```
var buffer = StringBuffer();
buffer.write('HP: ');
buffer.write('██████████');
buffer.writeln('░░ 80%');
print(buffer.toString());
```

String manipulation, as ferramentas do artesão

Para construir arte ASCII programática, precisamos dominar os métodos de manipulação de strings.

Repetição com `*`:

```
print('=' * 30);
print('#' * 10);
```

Preenchimento com `padLeft()` e `padRight()`:

```
var texto = 'Aldric';
print(texto.padRight(20));
print(texto.padLeft(20));
print(texto.padRight(20, '.'));
```

A função `padRight(20)` garante que a string tenha exatamente 20 caracteres, preenchendo com espaços à direita se necessário. Isso é essencial para alinhar colunas em tabelas.

Centralizar texto (Dart não tem método nativo para isso, mas podemos criar):

```
String centralizar(String texto, int largura) {
  if (texto.length >= largura) return texto;
  var espacos = largura - texto.length;
  var esquerda = espacos ~/ 2;
  return texto.padLeft(texto.length + esquerda).padRight(largura);
}

print(centralizar('MENU', 30));
```

O operador `~/` é a divisão inteira, retorna um `int` em vez de `double`. Por exemplo, `7 ~/ 2` retorna `3`, não `3.5`.

Construindo uma moldura dinâmica

Agora que dominamos `StringBuffer`, vamos criar uma função mais sofisticada: uma moldura que se ajusta dinamicamente ao conteúdo. Esse é um excelente exercício de manipulação de strings. Calcular tamanhos programaticamente, preencher espaços e construir strings complexas são habilidades fundamentais para qualquer desenvolvedor de jogos de texto.

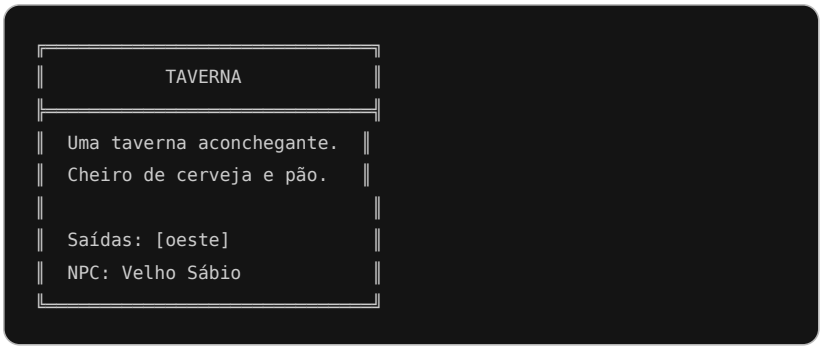
```
String moldura(
  String titulo,
  List<String> linhas, {
  int minLargura = 30,
}) {
  var maxTexto = titulo.length;
  for (var linha in linhas) {
    if (linha.length > maxTexto) maxTexto = linha.length;
  }
  var larguraInterna = maxTexto + 4;
  if (larguraInterna < minLargura) larguraInterna = minLargura;
```

```
  var buffer = StringBuffer();

  buffer.write('╔');
  buffer.write('═' * larguraInterna);
  buffer.writeln('╗');

  var tituloFormatado = centralizar(titulo, larguraInterna);
  buffer.writeln('║$tituloFormatado║');

  buffer.write('╠');
  buffer.write('═' * larguraInterna);
  buffer.writeln('╣');

  for (var linha in linhas) {
    var conteudo = '  $linha'.padRight(larguraInterna);
    buffer.writeln('║$conteudo║');
  }

  buffer.write('╚');
  buffer.write('═' * larguraInterna);
  buffer.writeln('╝');

  return buffer.toString();
}
```

Exemplo de uso:

```
print(moldura('TAVERNA', [
  'Uma taverna aconchegante.',
  'Cheiro de cerveja e pão.',
  '',
  'Saídas: [oeste]',
  'NPC: Velho Sábio'
]));
```

Resultado:



A função `moldura()` é um ótimo exercício de manipulação de strings. No nosso jogo, porém, vamos usar saídas mais simples no terminal para manter o código limpo e focado na lógica do *roguelike*.

Barras de HP e XP

Barras visuais são o tipo mais icônico de interface em jogos de texto. Uma barra bem desenhada comunica informações instantaneamente: num relance na proporção de blocos preenchidos, o jogador sabe exatamente em que estado seu personagem está. Vamos construir uma barra de HP reutilizável que usa caracteres Unicode para representar a vida de forma visual.

```
String barraHP(int atual, int maximo, {int largura = 20}) {
  var proporcao = atual / maximo;
  var preenchidos = (proporcao * largura).round();
  var vazios = largura - preenchidos;

  var barra = '\u2588' * preenchidos + '\u2591' * vazios;
  var porcentagem = (proporcao * 100).toInt();

  return '$barra $atual/$maximo ($porcentagem%)';
}

print('HP: ${barraHP(75, 100)}');
print('HP: ${barraHP(20, 100)}');
```

```
print('HP: ${barraHP(100, 100)}');
```

Podemos adicionar rótulos semânticos usando palavras:

```
String barraComStatus(String rotulo, int atual, int maximo) {
  var barra = barraHP(atual, maximo, largura: 15);
  var status = atual < maximo * 0.25
      ? '** PERIGO! **'
      : atual < maximo * 0.5
          ? '(cuidado)'
          : '';
  return '$rotulo: $barra $status';
}
```

Tabelas ASCII

Para listas de itens, lojas ou comparar estatísticas entre armas, tabelas ASCII estruturadas são essenciais. Uma tabela bem construída permite que o jogador absorva informações complexas num relance. Vamos montar uma função que cria uma tabela com bordas alinhadas automaticamente, sem que você precise calcular os espaços manualmente.

```
String tabela(List<String> cabecalhos, List<List<String>> linhas) {
  var larguras = List<int>.filled(cabecalhos.length, 0);
  for (var i = 0; i < cabecalhos.length; i++) {
    larguras[i] = cabecalhos[i].length;
  }
  for (var linha in linhas) {
    for (var i = 0; i < linha.length && i < larguras.length; i++) {
      if (linha[i].length > larguras[i]) {
        larguras[i] = linha[i].length;
      }
    }
  }
```

```
  var buffer = StringBuffer();
  var separador = '+${larguras.map((l) => '-' * (l + 2)).join('+')}+';

  buffer.writeln(separador);
  var cab = cabecalhos
      .asMap()
      .entries
      .map((e) => ' ${e.value.padRight(larguras[e.key])} ')
      .join('|');
  buffer.writeln('|$cab|');
  buffer.writeln(separador);

  for (var linha in linhas) {
    var row = linha
        .asMap()
        .entries
        .map((e) => ' ${e.value.padRight(larguras[e.key])} ')
        .join('|');
    buffer.writeln('|$row|');
  }
  buffer.writeln(separador);

  return buffer.toString();
}
```

Exemplo de uso:

```
print(tabela(
  ['Item', 'Preço', 'Dano'],
  [
    ['Adaga', '30g', '+5'],
    ['Espada Curta', '80g', '+8'],
    ['Machado', '120g', '+12']
  ]
```

```
));
```

Resultado:

```
+--------------+-------+------+
| Item         | Preço | Dano |
+--------------+-------+------+
| Adaga        | 30g   | +5   |
| Espada Curta | 80g   | +8   |
| Machado      | 120g  | +12  |
+--------------+-------+------+
```

Aplicação no jogo: HUD composto

Agora vamos integrar tudo: molduras, barras e alinhamento. Um bom **HUD** (Head-Up Display) comunica a saúde do jogador, recursos disponíveis e equipamento atual, tudo num pequeno espaço. Vamos montar um HUD que combina as técnicas de moldura, preenchimento e barra para criar uma tela profissional.

```
String montarHUD(
  String nome,
  int hp,
  int maxHp,
  int ouro,
  String? arma,
) {
  var buffer = StringBuffer();
  var largura = 38;

  buffer.write('╔');
  buffer.write('=' * largura);
  buffer.writeln('╗');
```

```
  var nomeFormatado = '  $nome'.padRight(largura);
  buffer.writeln('║$nomeFormatado║');

  buffer.write('╠');
  buffer.write('═' * largura);
  buffer.writeln('╣');

  var hpBarra = barraHP(hp, maxHp, largura: 15);
  var hpLinha = '  HP: $hpBarra'.padRight(largura);
  buffer.writeln('║$hpLinha║');

  var ouroLinha = '  Ouro: ${ouro}g'.padRight(largura);
  buffer.writeln('║$ouroLinha║');

  var armaTexto = arma ?? 'Nenhuma';
  var armaLinha = '  Arma: $armaTexto'.padRight(largura);
  buffer.writeln('║$armaLinha║');

  buffer.write('╚');
  buffer.write('═' * largura);
  buffer.writeln('╝');

  return buffer.toString();
}

print(montarHUD('Aldric', 75, 100, 42, 'Espada Curta'));
```

Resultado:

```
╔══════════════════════════════════╗
║  Aldric                          ║
╠══════════════════════════════════╣
║  HP: ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▒▒▒▒▒ 75/100 (75%)   ║
║  Ouro: 42g                       ║
║  Arma: Espada Curta              ║
```

Desafios da Masmorra

Desafio 6.1. Moldura com título e rodapé. Modifique a função `moldura()` para aceitar um parâmetro opcional `rodape`. Se fornecido, adicione uma linha separadora (com –) entre o conteúdo e o rodapé, depois exiba o rodapé com alinhamento. Por exemplo: uma caixa de inventário com "Mochila Vazia" como rodapé.

Desafio 6.2. Barra de XP customizada. Crie uma função `barraXP(int xpAtual, int xpProximoNivel, int nivel)` que mostra uma barra diferente da de HP: use ▓ (preenchido) e ⁘ (vazio), similar à de HP mas com caracteres diferentes. Ao lado, mostre "Nível X" e a percentagem de progresso.

Desafio 6.3. Caixa de diálogo de NPC (Com bordas especiais). Crie uma função `dialogoNPC(String nomeNPC, String fala)` que exibe uma caixa estilizada: o nome do NPC em negrito (ou destacado com cores se em suporte a ANSI) no topo, a fala envolvida com uma borda especial diferente da HUD (use caracteres como ┌, └, |).

Desafio 6.4. Mini-mapa do mundo. Usando `StringBuffer`, desenhe um mini-mapa 5x5 onde `@` é o jogador, `#` são paredes (limites da masmorra), `.` é chão livre e `?` são salas não visitadas. Use a sala atual e salas vizinhas para popular o mapa.

Boss Final 6.5. Tela de morte épica (Game Over). Crie uma função `telaGameOver(String nome, int turnos, int ouro)` que monta uma tela de game over elaborada. Inclua: arte ASCII de um túmulo ou caveira, nome do herói caído, quantos turnos sobreviveu, ouro acumulado, e uma última mensagem do tipo "Descansa em paz, herói." Use box-drawing para tornar impressionante.

Dica do Mestre: Monte tudo com `StringBuffer`. Comece com uma moldura externa grande usando ╔═╗ ║ ╚═╝. Dentro, desenhe um túmulo simples com `/\` e `|_|`. Centralize o nome e os números. Use `'=' * largura` para linhas separadoras.

O próximo passo: organizando o caos com classes

Você agora domina `StringBuffer`, strings interpoladas, alinhamento e arte ASCII. São ferramentas sólidas para desenhar qualquer tela. Mas há um problema que vai aparecer conforme o jogo cresce: o código fica espalhado. Você tem funções `moldura()`, `barraHP()`, `tabela()` e `montarHUD()`. Depois virão mais 20 funções para combate, inventário, equipamento e magia. Tudo solto, sem relação clara.

No Capítulo 5 aprendemos que coleções (List, Map, Set) agrupam *dados*. Mas dados sozinhos não bastam. Você precisa agrupar *dados e comportamento*. Seu jogador tem HP, nome, ouro e inventário. Seu inventário tem itens. Cada item tem dano, preço e descrição. Hoje isso é feito com Map e variáveis globais. Amanhã, com classes, você agrupa tudo: dados mais métodos que operam naqueles dados.

Parte II começa a essa jornada. Suas salas soltas em `mundoSalas` viram objetos `Sala`. Seus itens em listas viram objetos `Item`. Seu personagem vira `Jogador`. E cada classe organiza seus dados e suas funções de forma clara e reutilizável.

Pergaminho do Capítulo

Neste capítulo você aprendeu a usar `StringBuffer` para montar texto complexo de forma eficiente, os métodos `padLeft`, `padRight` e `*` para alinhamento e repetição, a construir molduras dinâmicas que se ajustam ao conteúdo, barras de HP visuais com caracteres de bloco, tabelas ASCII com colunas alinhadas, e a divisão inteira `~/` para cálculos de posicionamento.

Essas são as ferramentas visuais que vamos usar pelo resto do livro. Toda interface do jogo será construída com essas mesmas técnicas. No Capítulo 7, vamos unificar tudo num *game loop* completo e bem organizado.

Dica do Mestre: Ao desenhar interfaces ASCII, escolha uma largura padrão (como 40 ou 50 caracteres) e mantenha-a consistente em todas as telas. Nada quebra mais a imersão visual que molduras desalinhadas em sequência. Defina uma constante global `const larguraTela = 40;` e use-a em todas as funções de desenho.

Capítulo 7 - O game loop, o coração do jogo

> *O loop é o coração que bate no peito de todo jogo. Enquanto ele pulsa, o mundo respira; o jogador age, o jogo reage, a tela redesenha. Quando ele para, o jogo morre. Neste capítulo, construímos o coração.*

Nos seis capítulos anteriores, construímos peças separadas: input, output, funções, null safety, coleções e arte ASCII. Agora é hora de juntar tudo num programa coeso - uma aventura textual completa com salas conectadas, itens coletáveis, HUD visual e um loop principal bem organizado. Este é o marco da Parte I: o primeiro *roguelike* jogável.

O que é um *game loop*

Todo jogo, de Pac-Man a Elden Ring, roda sobre o mesmo conceito: um *game loop* que se repete indefinidamente, executando três passos a cada iteração.

O primeiro passo é mostrar o estado atual: desenhar o mundo na tela (ou, no nosso caso, no terminal).

O segundo passo é receber input: ler o que o jogador quer fazer.

O terceiro passo é atualizar o estado: mover o jogador, resolver combate, coletar itens, mudar de sala.

E então o loop volta ao primeiro passo, mostrando o novo estado. Esse ciclo se repete até o jogo acabar. Em jogos gráficos, esse loop roda dezenas de vezes por segundo. Num jogo de texto por turnos como o nosso, o loop espera o jogador digitar algo antes de avançar. A filosofia é a mesma; o ritmo é que muda.

Organizando o código, separar dados de lógica

Antes de montar o loop, precisamos organizar o código em seções bem definidas. Vamos separar o que é constante (variáveis que nunca mudam), o que é estado do mundo (dados das salas), o que é estado do jogador (HP, ouro, inventário) e o que são funções de processamento. Essa separação torna o código muito mais fácil de entender, manter e estender.

```
import 'dart:io';

// ═══════════════════════════════════════
// CONSTANTES
// ═══════════════════════════════════════

const larguraTela = 40;
const versao = '0.3.0';

// ═══════════════════════════════════════
// DADOS DO MUNDO
// ═══════════════════════════════════════

final mundoSalas = <String, Map<String, dynamic>>{
  'praca': {
    'nome': 'Praça Central',
    'descricao': 'Uma fonte de pedra murmura ao centro da praça.\n'
        'Tochas iluminam três passagens que se abrem\n'
        'nas paredes de pedra.',
    'saidas': {
      'norte': 'corredor',
      'leste': 'taverna',
      'sul': 'portao'
    },
    'itens': <String>['Tocha', 'Chave Enferrujada']
  },
  'corredor': {
    'nome': 'Corredor Escuro',
    'descricao': 'Um corredor estreito e frio. As paredes são\n'
        'cobertas de musgo. Água pinga do teto. Algo\n'
        'se move na escuridão à frente.',
    'saidas': {'sul': 'praca', 'norte': 'armaria'},
    'itens': <String>[]
  },
  'taverna': {
    'nome': 'Taverna do Javali',
```

```
    'descricao': 'Uma taverna aconchegante. O cheiro de cerveja\n'
        'e pão fresco preenche o ar. Um velho sábio\n'
        'cochila no canto junto à lareira.',
    'saidas': {'oeste': 'praca'},
    'itens': <String>['Poção de Vida']
  },
  'portao': {
    'nome': 'Portão da Masmorra',
    'descricao': 'Um portão de ferro enorme. Além dele, escuridão\n'
        'absoluta. Correntes de ar frio sopram de dentro.\n'
        'Você sente que não está pronto... ainda.',
    'saidas': {'norte': 'praca'},
    'itens': <String>['Moeda de Ouro']
  },
  'armaria': {
    'nome': 'Armaria Abandonada',
    'descricao': 'Armas enferrujadas penduradas nas paredes.\n'
        'A maioria está inútil, mas algo brilha\n'
        'debaixo de um pano rasgado.',
    'saidas': {'sul': 'corredor'},
    'itens': <String>['Adaga', 'Escudo de Madeira']
  }
};

final sinonimos = <String, String>{
  'n': 'norte', 's': 'sul', 'l': 'leste', 'o': 'oeste',
  'e': 'leste', 'w': 'oeste',
  'i': 'inventario', 'inv': 'inventario',
  'p': 'pegar', 'olhar': 'olhar', 'ver': 'olhar',
  'largar': 'largar', 'drop': 'largar',
  'h': 'ajuda', 'help': 'ajuda', '?': 'ajuda',
  'q': 'sair', 'quit': 'sair',
};

// ═══════════════════════════════════
// ESTADO DO JOGADOR
```

```
// ═══════════════════════════════════════

var nomeJogador = 'Aventureiro';
var salaAtual = 'praca';
var inventario = <String>[];
// Rastreia salas visitadas para indicar novos lugares
var salasVisitadas = <String>{};
var ouro = 0;
var hp = 100;
var maxHp = 100;
var turno = 0;
```

Separamos claramente os dados do jogador. No Capítulo 8, essas variáveis soltas serão substituídas por uma classe `Jogador`. Por enquanto, essa organização em seções claras já é um grande avanço.

Funções de renderização

Agora adicionamos as funções que constroem a interface visual do jogo. Essas funções recebem valores (nome do jogador, HP atual) e retornam strings formatadas prontas para imprimir. Reutilizaremos essas funções repetidamente: `exibirHUD()` sempre que precisamos mostrar o painel, `exibirSala()` quando o jogador entra numa sala nova.

```
// ═══════════════════════════════════════
// RENDERIZAÇÃO
// ═══════════════════════════════════════

String centralizar(String texto, int largura) {
  if (texto.length >= largura) return texto;
  var espacos = largura - texto.length;
  var esquerda = espacos ~/ 2;
  return texto.padLeft(texto.length + esquerda).padRight(largura);
}
```

```
String barraHP(int atual, int maximo, {int largura = 15}) {
  var prop = atual / maximo;
  var cheios = (prop * largura).round();
  var vazios = largura - cheios;
  return '${'\u2588' * cheios}${'\u2591' * vazios} $atual/$maximo';
}

void exibirHUD() {
  print('');
  print('  $nomeJogador');
  print('  HP: ${barraHP(hp, maxHp)}');
  print('  Ouro: ${ouro}g');
  print('');
}

void exibirSala() {
  var sala = mundoSalas[salaAtual]!;
  var nome = sala['nome'] as String;
  var descricao = sala['descricao'] as String;
  var saidas = sala['saidas'] as Map<String, String>;
  var itens = sala['itens'] as List<String>;

  var primeira = !salasVisitadas.contains(salaAtual);
  if (primeira) {
    salasVisitadas.add(salaAtual);
    print('');
    print('\u2605 Lugar novo! \u2605');
  } else {
    print('');
    print('(Você já visitou este lugar)');
  }
  print(nome.toUpperCase());

  for (var linha in descricao.split('\n')) {
    print('  $linha');
  }
```

```
  var saidasTexto = saidas.keys.map((d) => '[$d]').join(' ');
  print('Saídas: $saidasTexto');

  if (itens.isNotEmpty) {
    print('No chão: ${itens.join(', ')}');
  }

  print('');
}

void exibirInventario() {
  print('');
  if (inventario.isEmpty) {
    print('Sua mochila está vazia.');
  } else {
    print('INVENTÁRIO');
    for (var i = 0; i < inventario.length; i++) {
      print('  ${i + 1}. ${inventario[i]}');
    }
  }
  print('');
}
```

Funções de comando

Agora as funções que processam as ações do jogador. Cada uma delas representa um comando válido: `mover()` altera a sala atual, `pegarItem()` modifica o inventário e a sala, `largarItem()` faz o oposto. Essas funções encapsulam a lógica do jogo, tornando o loop principal limpo e legível.

```
// ═══════════════════════════════════════
// COMANDOS
// ═══════════════════════════════════════
```

```
void mover(String direcao) {
  var sala = mundoSalas[salaAtual]!;
  var saidas = sala['saidas'] as Map<String, String>;

  if (!saidas.containsKey(direcao)) {
    print('Não há saída para $direcao.');
    return;
  }

  salaAtual = saidas[direcao]!;
  turno++;
  print('Você vai para $direcao...');
  exibirSala();
}

void pegarItem(String nomeItem) {
  var sala = mundoSalas[salaAtual]!;
  var itens = sala['itens'] as List<String>;

  var encontrado = itens.where(
    (item) => item.toLowerCase().contains(nomeItem.toLowerCase())
  ).toList();

  if (encontrado.isEmpty) {
    print('Não há "$nomeItem" aqui.');
    return;
  }

  if (inventario.length >= 10) {
    print('Mochila cheia! Largue algo primeiro.');
    return;
  }

  var item = encontrado.first;
  itens.remove(item);
```

```
  // Moedas de Ouro vão direto para o contador de
  // ouro, não para o inventário
  if (item == 'Moeda de Ouro') {
    ouro += 10;
    print('Você pegou $item e ganhou 10g! (Total: ${ouro}g)');
  } else {
    inventario.add(item);
    print('Você pegou: $item.');
  }
  turno++;
}

void largarItem(String nomeItem) {
  var encontrado = inventario.where(
    (item) => item.toLowerCase().contains(nomeItem.toLowerCase())
  ).toList();

  if (encontrado.isEmpty) {
    print('Você não tem "$nomeItem".');
    return;
  }

  var item = encontrado.first;
  inventario.remove(item);
  var sala = mundoSalas[salaAtual]!;
  (sala['itens'] as List<String>).add(item);
  print('Você largou: $item.');
  turno++;
}
```

O *game loop* principal

Finalmente, o loop que une tudo em `main()`. O `main()` é surpreendentemente simples agora: inicializa o jogo, mostra a cena inicial e depois entra num `while` que continua até o jogador sair. Em cada iteração, imprime um prompt, lê o comando, processa e renderiza. Esse é o padrão que vai funcionar para todos os nossos jogos por turnos.

```dart
import 'dart:io';

// ═══════════════════════════════════════
// GAME LOOP
// ═══════════════════════════════════════

void main() {
  print('');
  print('╔${'═' * larguraTela}╗');
  print('║${centralizar('M A S M O R R A   A S C I I',
↪  larguraTela)}║');
  print('║${centralizar('v$versao', larguraTela)}║');
  print('╚${'═' * larguraTela}╝');
  print('');

  stdout.write('Como devo chamá-lo? ');
  nomeJogador = (stdin.readLineSync() ?? '').trim();
  if (nomeJogador.isEmpty) nomeJogador = 'Aventureiro';

  print('\nBem-vindo, $nomeJogador! Sua aventura começa agora.\n');

  exibirHUD();
  exibirSala();

  while (true) {
    print('');
    stdout.write('Turno $turno > ');
    var input = (stdin.readLineSync() ?? '').trim().toLowerCase();

    if (input.isEmpty) continue;

    var partes = input.split(' ');
    var cmd = sinonimos[partes[0]] ?? partes[0];
    var argumento = partes.length > 1
        ? partes.sublist(1).join(' ')
        : '';
```

```
    switch (cmd) {
      case 'norte' || 'sul' || 'leste' || 'oeste':
        mover(cmd);

      case 'pegar':
        if (argumento.isEmpty) {
          print('Pegar o quê? Use: pegar <item>');
        } else {
          pegarItem(argumento);
        }

      case 'largar':
        if (argumento.isEmpty) {
          print('Largar o quê? Use: largar <item>');
        } else {
          largarItem(argumento);
        }

      case 'inventario':
        exibirInventario();

      case 'olhar':
        exibirSala();

      case 'status':
        exibirHUD();

      case 'ajuda':
        print('');
        print('Comandos disponíveis:');
        print('  norte/sul/leste/oeste (n/s/l/o), mover');
        print('  pegar <item> (p), pegar item do chão');
        print('  largar <item>, largar item no chão');
        print('  inventario (i), ver mochila');
        print('  olhar, ver sala atual');
```

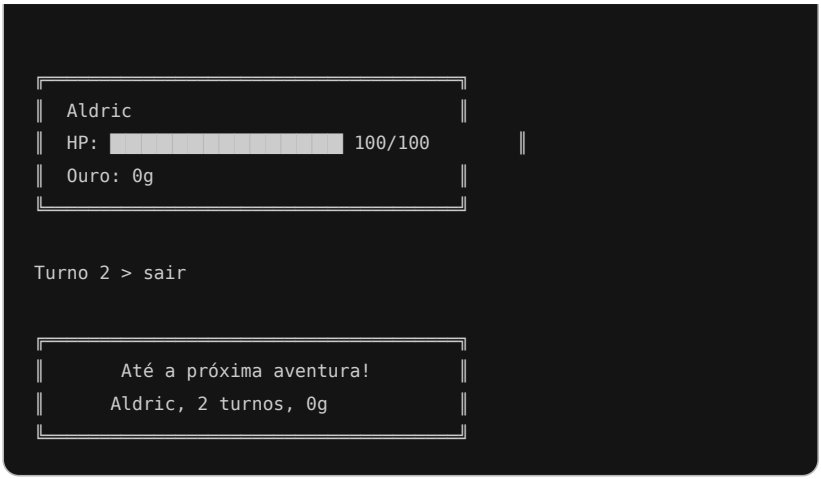
```
        print('  status, ver HP e ouro');
        print('  ajuda (h/?), esta mensagem');
        print('  sair (q), encerrar o jogo');

      case 'sair':
        print('');
        print('╔${'=' * larguraTela}╗');
        var msgFinal = centralizar(
          'Até a próxima aventura!',
          larguraTela,
        );
        print('║$msgFinal║');
        final resumo = '$nomeJogador, $turno turnos, ${ouro}g';
        var resumoFormatado = centralizar(resumo, larguraTela);
        print('║$resumoFormatado║');
        print('╚${'=' * larguraTela}╝');
        return;

      default:
        print('Não entendi "$input". '
            'Digite "ajuda" para ver os comandos.');
    }
  }
}
```

Uma sessão completa

Execute o programa e interaja como o exemplo abaixo:

```
╔═══════════════════════════╗
║      M A S M O R R A  A S C I I  ║
║            v0.3.0             ║
╚═══════════════════════════╝
```

```
Como devo chamá-lo? Aldric

Bem-vindo, Aldric! Sua aventura começa agora.

╔══════════════════════════════════╗
║  Aldric                          ║
║  HP: ████████████████████ 100/100        ║
║  Ouro: 0g                        ║
╚══════════════════════════════════╝

★ Lugar novo! ★
PRAÇA CENTRAL

Uma fonte de pedra murmura ao centro
da praça. Tochas iluminam três
passagens que se abrem nas paredes.

Saídas: [norte] [leste] [sul]
No chão: Tocha, Chave Enferrujada

Turno 0 > p tocha
Você pegou: Tocha.

Turno 1 > n
Você vai para norte...

★ Lugar novo! ★
CORREDOR ESCURO

Um corredor estreito e frio. As
paredes são cobertas de musgo. Água
pinga do teto.

Saídas: [sul] [norte]

Turno 2 > status
```



Esse é o marco da Parte I. Você tem um jogo funcional: salas conectadas, itens que podem ser pegos e largados, um HUD com barra de HP, ouro que pode ser coletado, contador de turnos e um loop que roda até o jogador decidir sair. É simples, mas é completo, e tudo foi feito com Dart puro no terminal.

Ponte para a Parte II: classes chegam

O jogo roda, mas repare num problema: os dados estão espalhados. O jogador é um monte de variáveis soltas (`nomeJogador`, `hp`, `maxHp`, `ouro`, `inventario`, `turno`). As salas são `Map<String, dynamic>`, sem segurança de tipo. Se você digitar `hpAtual` em vez de `hp`, o compilador não reclama até a execução falhar.

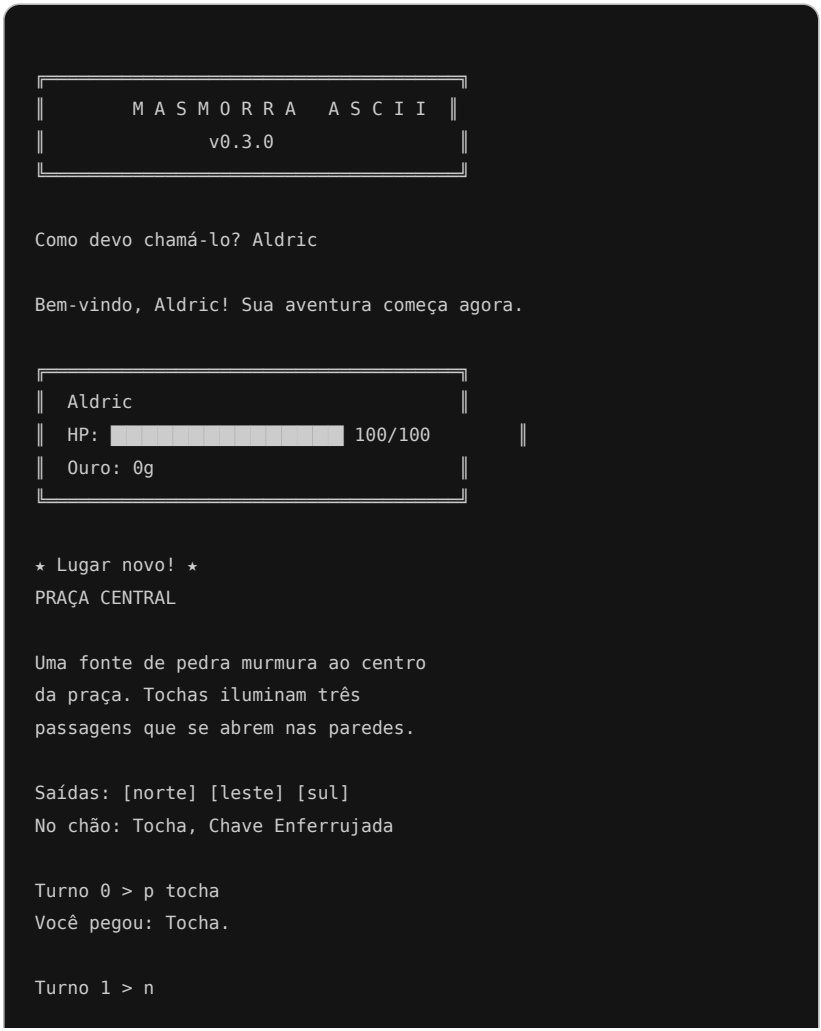
E há pior: o comportamento está separado dos dados. Renderizar o HUD é uma função `exibirHUD()` que lê variáveis globais. Coletar item é `pegarItem()` que manipula listas. Não há coesão. Imagine daqui a 10 capítulos com 100 funções, 50 variáveis globais e 20 classes de inimigos diferentes; isto seria um caos.

Na Parte II, vamos organizar tudo com classes. Seus dados soltos viram objetos tipados: `Jogador`, `Sala`, `Item`, `Inimigo`. Cada classe agrupa seus dados com os métodos que operam neles. O jogador *sabe* como levar dano, equipar uma arma, ganhar XP. Uma sala *sabe* como renderizar a si mesma. Um item *sabe* seu peso e preço. O código fica limpo, reutilizável e pronto para crescer.

Comece o Capítulo 8. Está na hora de aprender orientação a objetos de verdade.

O Jogo Até Aqui

Ao final desta parte, seu jogo no terminal se parece com isto:



Cada parte adiciona novas camadas ao jogo. Compare com o início e veja o quanto você evoluiu!

Desafios da Masmorra

Desafio 7.1. Eventos aleatórios (Suspense). Adicione um evento aleatório a cada turno com 20% de probabilidade. Use `import 'dart:math'` e `Random().nextInt(100) < 20` para decidir. Exemplos: "Você ouve passos distantes...", "Um sopro frio passa por você", "Algo se move na sombra". Mostre apenas quando o evento ocorrer.

Desafio 7.2. Comando examinar (Pistas escondidas). Adicione um campo `detalhes` (texto longo) a cada sala além da descrição breve. O comando `"examinar"` ou `"x"` mostra esses detalhes. Serve para esconder pistas e informações extras para jogadores curiosos investigarem.

Desafio 7.3. Ambiente hostil (HP dinâmico). Cada vez que o jogador entrar numa sala com descrição contendo "escuro", "frio", "úmido" ou "perigoso", perca 5 HP automaticamente. Use `.contains()`. Se HP chegar a 0, exiba a tela de game over. Isso torna algumas salas mais perigosas que outras: ambiente vs jogador.

Desafio 7.4. Tela de estatísticas finais. Ao sair do jogo, exiba uma tabela formatada com: turnos jogados, salas visitadas (conte as únicas), itens coletados, ouro final e HP sobrevivido. Use box-drawing e formatação visual.

Boss Final 7.5. Sistema de diálogo com NPC (Velho Sábio). Adicione um NPC chamado "Velho Sábio" numa sala especial "Taverna". O comando `"falar"` inicia um diálogo com 3 opções de respostas (use número ou letra). Uma das respostas revela uma dica sobre uma sala secreta. Se o jogador tiver a "Chave Enferrujada" no inventário quando resolver voltar, uma nova saída aparece na "Câmara Secreta" com ouro ou uma arma valiosa.

Dica do Mestre: Guarde uma flag `conversouComVelhoSabio = false` para saber se já falou com ele. O diálogo apresenta 3 opções; uma delas ('sabedoria') desbloqueia a dica. Depois, quando voltar à Taverna, a lógica de saídas verifica se tem a chave E já conversou: se sim, adiciona saída para "Câmara Secreta".

Pergaminho do Capítulo

Neste capítulo você construiu o *game loop* completo: ler input, processar comando, atualizar estado, redesenhar a tela. Organizou o código em seções claras (constantes, dados, estado, renderização, comandos, loop principal). Integrou todas as técnicas dos capítulos anteriores numa aventura textual jogável com 5 salas, inventário, ouro e HUD visual.

Este é o fim da Parte I. Você partiu de `print('Olá')` e chegou a um jogo funcional no terminal.

Dica do Mestre: O *game loop* que construímos é síncrono: ele para e espera o jogador digitar. Isso é perfeito para um jogo por turnos. Mas quando adicionarmos persistência nos capítulos futuros, vamos precisar de `async`/`await` para operações de disco. Não se preocupe com isso agora. Quando chegar a hora, a transição será natural. O importante é que a estrutura do loop já está correta.

PARTE II

SANGUE, OURO E AÇO

Agora os objetos ganham vida própria — não são mais dados brutos, mas seres com responsabilidade e poder. Inimigos ganham nome, classe, vontade. O combate não é apenas lógica; é estratégia e hierarquia. E nesta arena programada, aprenderá que organização vence o caos.

Capítulo 8 - Classes: dando vida ao jogador

Você deixou de ser observador da masmorra; agora é arquiteto. Classes são a forja onde você molda jogadores, inimigos e itens com precisão. Herança conecta criaturas em linhagens, como famílias de monstros que compartilham traços mas guardam suas próprias surpresas. Mixins são habilidades que qualquer criatura pode aprender, como um pergaminho de poder colado nas costas de quem precisar.

Nesta parte, o código deixa de ser uma sequência de instruções e vira um conjunto de objetos que conversam entre si. Você vai criar seu primeiro sistema de combate por turnos, com inventário, equipamentos e inimigos que morrem de verdade. Quando o último Zumbi cair, você vai perceber que não está mais apenas programando. Está construindo um mundo.

> *Na mesa do mestre do jogo, a ficha do personagem é mais que números, é identidade. Nome, vida, força, inventário, tudo organizado numa folha. Em Dart, essa ficha chama-se classe. E o personagem escrito a lápis é o objeto.*

No Capítulo 7, o jogador era um punhado de variáveis soltas: `nomeJogador`, `hp`, `ouro`, `inventario`, `salaAtual`. Funcionava, mas era frágil. Se quiséssemos dois jogadores (multiplayer?), teríamos que duplicar tudo manualmente. Se uma função precisasse de todos os dados do jogador, teríamos que passar seis parâmetros; e se alguém alterasse `hp` num canto obscuro do código, não havia como rastrear.

Classes resolvem tudo isso. Uma **classe** agrupa dados relacionados e as operações que atuam sobre eles num único lugar. É a ferramenta mais importante de Dart (e de qualquer linguagem orientada a objetos), e a partir deste capítulo ela estará em cada linha do jogo.

O conceito: classe vs objeto

Uma `class` é um molde. Ela descreve o que um tipo de coisa tem e o que pode fazer. A `class Jogador` diz: "um jogador tem nome, HP, ouro e inventário. Pode sofrer dano, coletar itens e equipar armas."

Um **objeto** (ou instância) é uma coisa concreta criada a partir desse molde. Quando você escreve `var heroi = Jogador('Aldric')`, está criando

um objeto específico: o jogador Aldric, com seus próprios valores de HP, ouro e inventário.

A analogia clássica: a classe é a planta de uma casa; o objeto é a casa construída. Você pode construir várias casas a partir da mesma planta, e cada uma tem sua própria cor de parede e mobília.

Criando a classe Jogador

A forma mais direta de transformar dados soltos em um objeto é criar uma classe que agrupa tudo. Uma classe Dart começa com `class` seguido do nome em PascalCase, depois declara os campos (dados que cada instância carrega) e o construtor (a função especial que cria novas instâncias). Para o Jogador, precisamos de nome, HP, ouro, arma, localização atual e inventário. Todos esses dados vivem juntos, são modificados em conjunto e fazem parte da mesma entidade.

```
class Jogador {
  String nome;
  int hp;
  int maxHp;
  int ouro;
  int ataque;
  String salaAtual;
  List<String> inventario;

  Jogador(this.nome, {
    this.hp = 100,
    this.maxHp = 100,
    this.ouro = 0,
    this.ataque = 5,
    this.salaAtual = 'praca',
    List<String>? inventario,
  }) : inventario = inventario ?? [];
}
```

Vamos decompor isso linha por linha.

`class Jogador {` declara o início da `class`. O nome `Jogador` segue a convenção Dart de PascalCase para classes (primeira letra de cada palavra em maiúscula).

Os campos (`nome`, `hp`, `maxHp`, etc.) são as propriedades do jogador, os dados que cada instância carrega consigo. São declarados com tipo e nome, como variáveis comuns.

`Jogador(this.nome, { ... })` é o construtor, a função especial que cria um novo objeto. O `this.nome` é um atalho de Dart que significa "o primeiro parâmetro se chama `nome` e vai direto para o campo `this.nome`". É equivalente a escrever:

```
Jogador(String nome) {
  this.nome = nome;
}
```

Os parâmetros entre `{ }` são parâmetros nomeados com valores padrão. Isso permite criar um jogador com apenas o nome, e tudo mais ganha valores automáticos:

```
var heroi = Jogador('Aldric');
// hp = 100, maxHp = 100, ouro = 0, ataque = 5, salaAtual = 'praca'

var veterano = Jogador(
  'Kael',
  hp: 150,
  maxHp: 150,
  ouro: 50,
  ataque: 10,
);
// valores customizados
```

A parte `: inventario = inventario ?? []` é uma lista de inicialização, código que roda antes do corpo do construtor. Aqui garantimos que se ninguém passar um inventário, ele começa como lista vazia.

Métodos: o que o jogador sabe fazer

Campos guardam dados. **Métodos** definem comportamentos, ações que o objeto pode executar (usando `void`, `bool`, etc.). Um método é uma função dentro da classe que pode acessar e modificar os campos do objeto. Para o Jogador, precisamos de métodos para sofrer dano, curar, gastar/receber ouro, pegar e largar itens. Cada método encapsula a lógica, garantindo que as regras do jogo sejam respeitadas.

```
class Jogador {
  String nome;
  int hp;
  int maxHp;
  int ouro;
  int ataque;
  String salaAtual;
  List<String> inventario;

  Jogador(this.nome, {
    this.hp = 100,
    this.maxHp = 100,
    this.ouro = 0,
    this.ataque = 5,
    this.salaAtual = 'praca',
    List<String>? inventario,
  }) : inventario = inventario ?? [];

  void sofrerDano(int quantidade) {
    hp -= quantidade;
    if (hp < 0) hp = 0;
  }

  void curar(int quantidade) {
    hp += quantidade;
    if (hp > maxHp) hp = maxHp;
  }
```

```
  bool gastarOuro(int quantidade) {
    if (ouro < quantidade) return false;
    ouro -= quantidade;
    return true;
  }

  void receberOuro(int quantidade) {
    ouro += quantidade;
  }

  bool get estaVivo => hp > 0;
  bool get inventarioCheio => inventario.length >= 10;

  bool pegarItem(String item) {
    if (inventarioCheio) return false;
    inventario.add(item);
    return true;
  }

  bool largarItem(String item) {
    return inventario.remove(item);
  }
}
```

Repare em vários padrões importantes aqui.

Validação interna: o método `sofrerDano` garante que HP nunca fica negativo. Antes, essa verificação teria que estar em cada lugar do código que modifica HP. Agora está num único lugar. Se amanhã a regra mudar (por exemplo, armadura reduz dano), você muda apenas aqui.

Retorno booleano: `gastarOuro` retorna `true` ou `false` indicando sucesso. Quem chama pode reagir:

```
if (heroi.gastarOuro(50)) {
  print('Compra realizada!');
} else {
```

```
  print('Ouro insuficiente.');
}
```

Getters: `estaVivo` e `inventarioCheio` usam a sintaxe `get:` são propriedades computadas que parecem campos mas na verdade calculam um valor (como `bool get`):

```
if (heroi.estaVivo) {
  print('Ainda de pé!');
}
```

A **arrow syntax** (`=>`) é um atalho para funções de uma linha. `bool get estaVivo => hp > 0;` é idêntico a usar `{ return ... }`:

```
bool get estaVivo {
  return hp > 0;
}
```

A classe Sala

O mesmo princípio se aplica às salas. Uma sala tem dados (ID, nome, descrição, saídas, itens no chão) e comportamentos (verificar se tem uma saída em determinada direção, saber se há um inimigo, saber se contém itens). Ao encapsular essa lógica em métodos, reutilizamos código e mantemos as regras num único lugar.

Repare no uso de `final` nos campos. `id`, `nome`, `descricao` e `temLoja` são imutáveis: definidos na criação e nunca mudam. Isso faz sentido, uma sala não muda de nome no meio do jogo. Mas `itens` é uma lista `final` cujo conteúdo pode mudar (itens são pegos ou largados), e `saidas` pode ser modificado dinamicamente (uma porta secreta que se revela).

> **Nota sobre evolução do modelo:** No Capítulo 10, vamos expandir a classe `Sala` substituindo `inimigoId: String?` por `inimigoPresente: Inimigo?`, armazenando a instância do inimigo diretamente em vez de apenas um

identificador de texto. Isso torna o modelo mais poderoso e tipado.

O parâmetro **required** obriga quem cria uma sala a fornecer `id`, `nome` e `descricao`. Sem eles, a sala não faz sentido. Você verá a implementação completa da classe `Sala` mais abaixo neste capítulo.

Referências: mesmo objeto, vários nomes

Um conceito crucial em Dart (e em qualquer linguagem orientada a objetos): quando você passa um objeto para uma função, está passando uma referência, não uma cópia. Isso significa que qualquer modificação feita na função afeta o objeto original. É importante entender isso porque torna o código eficiente (nenhuma cópia desperdiçada) mas exige cuidado (qualquer função pode modificar o jogador).

```
void danificarJogador(Jogador p) {
  p.sofrerDano(10);
}

var heroi = Jogador('Aldric');
print(heroi.hp);
danificarJogador(heroi);
print(heroi.hp);
```

`p` e `heroi` apontam para o mesmo objeto na memória. Modificar um modifica o outro. Isso é poderoso (evita cópias desnecessárias) mas requer atenção: qualquer função que receba o jogador pode alterá-lo.

Aplicação no jogo: refatorando com classes

Vamos refatorar o jogo do Capítulo 7 usando classes. Primeiro, criamos as classes num arquivo separado. A convenção em Dart é colocar cada classe no seu próprio arquivo em `lib/`. Essa separação torna o projeto escalável: adicionar um novo tipo de inimigo é adicionar um novo arquivo, não editar um megaarquivo. Por enquanto, mantemos tudo direto em `lib/`, sem subpastas. Mais adiante, quando o projeto crescer, vamos reorganizar em pastas por domínio.

Note que cada arquivo (como `lib/jogador.dart` e `lib/sala.dart`) pode ser importado noutros arquivos usando `import 'jogador.dart';` ou `import 'sala.dart';` quando estiverem na mesma pasta, ou com caminhos completos se em subpastas.

```
// lib/jogador.dart

class Jogador {
  String nome;
  int hp;
  int maxHp;
  int ouro;
  int ataque;
  String salaAtual;
  List<String> inventario;

  Jogador(this.nome, {
    this.hp = 100,
    this.maxHp = 100,
    this.ouro = 0,
    this.ataque = 5,
    this.salaAtual = 'praca',
    List<String>? inventario,
  }) : inventario = inventario ?? [];

  void sofrerDano(int quantidade) {
    hp -= quantidade;
    if (hp < 0) hp = 0;
  }

  void curar(int quantidade) {
    hp += quantidade;
    if (hp > maxHp) hp = maxHp;
  }

  bool gastarOuro(int quantidade) {
```

```
    if (ouro < quantidade) return false;
    ouro -= quantidade;
    return true;
  }

  void receberOuro(int quantidade) {
    ouro += quantidade;
  }

  bool get estaVivo => hp > 0;
  bool get inventarioCheio => inventario.length >= 10;

  bool pegarItem(String item) {
    if (inventarioCheio) return false;
    inventario.add(item);
    return true;
  }

  bool largarItem(String item) {
    return inventario.remove(item);
  }
}
```

```
// lib/sala.dart

class Sala {
  final String id;
  final String nome;
  final String descricao;
  final Map<String, String> saidas;
  final List<String> itens;
  final bool temLoja;
  final String? inimigoId;

  Sala({
```

```
    required this.id,
    required this.nome,
    required this.descricao,
    Map<String, String>? saidas,
    List<String>? itens,
    this.temLoja = false,
    this.inimigoId,
  }) : saidas = saidas ?? {},
       itens = itens ?? [];

  bool temSaida(String direcao) => saidas.containsKey(direcao);
  String? saidaPara(String direcao) => saidas[direcao];
  bool get temInimigo => inimigoId != null;
  bool get temItens => itens.isNotEmpty;
}
```

Agora o mundo do jogo usa objetos tipados em vez de `Map<String, dynamic>`:

```
// Antes (Capítulo 5-7): dados soltos em mapa genérico
var salas = <String, Map<String, dynamic>>{
  'praca': {
    'nome': 'Praça Central',
    'descricao': '...',
    'saidas': {'norte': 'corredor'},
    'itens': ['Tocha'],
  }
};

// Depois (Capítulo 8): objetos tipados
var salas = <String, Sala>{
  'praca': Sala(
    id: 'praca',
    nome: 'Praça Central',
    descricao: 'Uma fonte de pedra murmura ao centro...',
```

```
    saidas: {'norte': 'corredor', 'leste': 'taverna', 'sul':
↪   'portao'},
    itens: ['Tocha', 'Chave Enferrujada'],
  )
};
```

A diferença é enorme. Com o mapa genérico, `sala['descricao']` podia ser qualquer coisa; o compilador não reclamava se você escrevesse `sala['desc']` por engano. Com a `class Sala`, `sala.descricao` é garantido como `String` pelo compilador. Erros de digitação viram erros de compilação, não bugs em tempo de execução.

E o jogador:

```
// Antes: variáveis soltas
var nomeJogador = 'Aventureiro';
var hp = 100;
var ouro = 0;

// Depois: um único objeto
var jogador = Jogador('Aventureiro');
```

No `loop` principal, em vez de `hp -= 10`, fazemos `jogador.sofrerDano(10)`. Em vez de verificar `hp > 0`, fazemos `jogador.estaVivo`. O código fica mais legível e as regras, centralizadas.

O método toString

Todo objeto Dart pode ter um método `toString()` que define como ele é representado como texto. Isso é extraordinariamente útil para depuração: quando você imprime um objeto ou vê um erro, quer saber exatamente em que estado ele estava. Um bom `toString()` mostra os dados mais importantes num formato legível, sem ser tão longo a ponto de poluir o console.

```
class Jogador {
  // ... campos e métodos anteriores ...

  @override
  String toString() {
    return 'Jogador($nome, HP: $hp/$maxHp, Ouro: ${ouro}g, '
        'Sala: $salaAtual, Itens: ${inventario.length})';
  }
}
```

Agora `print(jogador)` mostra algo como:

```
Jogador(Aldric, HP: 85/100, Ouro: 42g, Sala: corredor, Itens: 3)
```

O `@override` indica que estamos substituindo o `toString` padrão (que mostra apenas `Instance of 'Jogador'`). Vamos usar `@override` muito mais nos próximos capítulos com `extends` e herança.

Desafios da Masmorra

Desafio 8.1. Classe Item (Objeto com peso e descrição). Crie uma classe `Item` com campos `nome`, `descricao` e `peso` (em gramas). Substitua as strings no inventário do jogador por objetos `Item`. Atualize `pegarItem` e `largarItem` para usar `Item` em vez de `String`. Implemente `toString()` para exibir o item de forma legível (exemplo: "Espada Curta (500g)").

Desafio 8.2. Peso e limite de carga. Adicione um campo `pesoMaximo` ao `Jogador` (padrão: 5000 gramas). O método `pegarItem` deve verificar se adicionar o novo item ultrapassaria o limite. Se ultrapassar, recuse com mensagem clara. Crie um getter `pesoAtual` que calcula o peso total do inventário em tempo real.

Desafio 8.3. Método toString robusto para Sala. Implemente `toString()` em `Sala` que mostra: nome, saídas disponíveis e quantidade de itens. Para debug, ao mudar de sala, imprima `print(novaSala)` para validar que o estado está correto. Formato exemplo: `"Sala(Praça Central, saídas: [n, l, s], itens: 2)"`.

Desafio 8.4. Método descrever para renderização. Adicione um método `String descrever()` em `Sala` que retorna uma descrição completa e formatada (usando `StringBuffer`): nome com moldura, descrição longa, saídas listadas, itens no chão com seus pesos. Substitua a função `exibirSala()` do Capítulo 7 por uma chamada a `sala.descrever()`.

Boss Final 8.5. Classe MundoTexto (Gerenciador de mundo). Crie uma classe `MundoTexto` que encapsula o `Map<String, Sala>` e fornece métodos: `Sala? obterSala(String id)`, `void adicionarSala(Sala sala)`, `List<String> salasConectadas(String id)` que retorna as salas alcançáveis. Substitua o mapa global `mundoSalas` por uma instância `var mundo = MundoTexto();` e use-a para todas as operações do jogo.

Pergaminho do Capítulo

Neste capítulo você aprendeu a criar classes com campos e métodos, construtores com parâmetros posicionais e nomeados, valores padrão e `required`, getters computados, que objetos são passados por referência, e o método `toString()` para representação textual.

O jogo deu um salto de organização: de variáveis soltas e mapas genéricos para objetos tipados com validação interna. No Capítulo 9, vamos refinar essas classes com encapsulamento (campos privados com _), construtores nomeados e factory constructors, as ferramentas que transformam uma classe básica numa API bem desenhada.

Dica do Mestre: Quando não souber se algo deveria ser um campo ou um getter, use esta regra: se o valor é armazenado e pode ser diferente entre instâncias, é um campo. Se o valor é calculado a partir de outros campos, é um getter. `nome` é campo. `estaVivo` é getter (calculado a partir de `hp`). Essa distinção mantém o modelo limpo e evita dados duplicados que ficam inconsistentes.

Próximo Capítulo

No próximo capítulo, protegemos o herói. Construtores controlam a criação, e encapsulamento garante que ninguém mexa no HP sem permissão.

Capítulo 9 - Construtores e encapsulamento

> *O ferreiro não deixa qualquer um enfiar a mão na forja. Há uma porta para pedidos e uma janela para entregas. O que acontece lá dentro (o martelar, o temperar, o polir) é problema dele. Em código, chamamos isso de encapsulamento.*

Trataremos de **construtores** e **encapsulamento**: no capítulo anterior, criamos classes com campos públicos. Qualquer parte do código pode ler e modificar `jogador.hp` diretamente. Isso funciona, mas quando o jogo cresce o acesso sem restrições causa bugs: alguém pode setar `hp = -50` sem querer, ou mudar `salaAtual` para uma sala que não existe. Neste capítulo, vamos aprender a proteger o estado interno das classes e a criar múltiplas formas de construir objetos.

O sublinhado _: privacidade em Dart

Em Dart, a privacidade funciona no nível da biblioteca (ou seja, do arquivo). Qualquer identificador que comece com _ é invisível fora daquele arquivo. Isso permite que você mantenha detalhes internos da classe privados, forçando o código externo a usar a API pública (getters e métodos) que você expôs. Use `_nomeVariavel` para campos privados e crie getters públicos apenas para o que realmente precisa ser lido de fora.

```
// lib/jogador.dart

class Jogador {
  final String nome;
  int _hp;
  int _maxHp;
  int _ouro;
  int _ataque;
  String _salaAtual;
  final List<String> _inventario;
```

```
  Jogador(this.nome, {
    int hp = 100,
    int maxHp = 100,
    int ouro = 0,
    int ataque = 5,
    String salaAtual = 'praca',
    List<String>? inventario,
  }) : _hp = hp,
       _maxHp = maxHp,
       _ouro = ouro,
       _ataque = ataque,
       _salaAtual = salaAtual,
       _inventario = inventario ?? [];

  int get hp => _hp;
  int get maxHp => _maxHp;
  int get ouro => _ouro;
  int get ataque => _ataque;
  String get salaAtual => _salaAtual;
  List<String> get inventario => List.unmodifiable(_inventario);
}
```

Agora, de fora do arquivo, `jogador._hp` causa erro de compilação. O único jeito de mudar o HP é através dos métodos que a `class` oferece (como `sofrerDano()` e `curar()`):

```
  void sofrerDano(int quantidade) {
    if (quantidade < 0) return;
    _hp -= quantidade;
    if (_hp < 0) _hp = 0;
  }

  void curar(int quantidade) {
    if (quantidade < 0) return;
```

```
  _hp += quantidade;
  if (_hp > _maxHp) _hp = _maxHp;
}
```

Repare que agora temos validação dupla: dano negativo é ignorado e HP nunca fica abaixo de zero. Essas garantias são impossíveis com campos públicos, porque qualquer trecho de código pode escrever `jogador.hp = -999`.

O getter `inventario` retorna `List.unmodifiable(_inventario)`, uma visão da lista que não permite `.add()` ou `.remove()` de fora. Quem quiser modificar o inventário precisa usar os métodos da classe:

```
bool pegarItem(String item) {
  if (_inventario.length >= 10) return false;
  _inventario.add(item);
  return true;
}

bool largarItem(String item) {
  return _inventario.remove(item);
}

bool temItem(String item) {
  return _inventario.any(
    (i) => i.toLowerCase() == item.toLowerCase()
  );
}
```

O campo `final`: imutável após construção

O `nome` do jogador é `final`: definido no construtor e nunca mais alterado. Isso faz sentido, o aventureiro não muda de nome no meio da partida. Marcar um campo como `final` diz ao compilador (e aos futuros leitores do código) que esse valor é um atributo permanente do objeto. Use `final` para campos imutáveis, tanto por clareza quanto por segurança:

```
final String nome;
```

Já `_hp` é mutável (sem `final`) porque o HP muda durante o jogo. A regra prática: se um campo não deve mudar após a criação do objeto, marque como `final`.

Para a `class Sala`, quase tudo é `final`:

```
class Sala {
  final String id;
  final String nome;
  final String descricao;
  final Map<String, String> saidas;
  final List<String> itens;
  final bool temLoja;
  final String? inimigoId;
}
```

Uma distinção sutil: `final List<String> itens` significa que a variável `itens` sempre aponta para a mesma lista, mas o conteúdo da lista pode mudar (itens adicionados ou removidos). Se quiséssemos impedir até isso, usaríamos uma lista imutável usando `const` ou `List.unmodifiable()` no construtor de forma mais robusta.

Construtores nomeados

Dart permite ter múltiplos construtores com nomes diferentes. Enquanto o construtor principal faz inicialização genérica, construtores nomeados podem oferecer formas especializadas de criar objetos. No jogo, queremos criar um recruta fraco para modo fácil, um veterano forte para modo difícil ou carregar um jogador salvo de um arquivo. Cada situação é um construtor nomeado, tornando o código que cria o jogador legível e expressivo.

```
class Jogador {
  // Construtor principal
```

```
  Jogador(this.nome, { /* ... */ });

  // Construtor nomeado: novo recruta com stats fracos
  Jogador.recruta(String nome)
      : this(nome, hp: 80, maxHp: 80, ouro: 10, ataque: 3);

  // Construtor nomeado: veterano com stats fortes
  Jogador.veterano(String nome)
      : this(nome, hp: 150, maxHp: 150, ouro: 100, ataque: 12);

  // Construtor nomeado: carregar de um mapa (para save/load)
  Jogador.deArquivo(Map<String, dynamic> dados)
      : this(
          dados['nome'] as String,
          hp: (dados['hp'] as int?) ?? 100,
          maxHp: (dados['maxHp'] as int?) ?? 100,
          ouro: (dados['ouro'] as int?) ?? 0,
          ataque: (dados['ataque'] as int?) ?? 5,
          salaAtual: (dados['salaAtual'] as String?) ?? 'praca',
          inventario: List<String>.from(
            dados['inventario'] as List? ?? [],
          ),
        );
}
```

Uso:

```
var noob = Jogador.recruta('Timmy');
var lenda = Jogador.veterano('Kael');

// Carregar do arquivo com tratamento de erro
Jogador? salvo;
try {
  salvo = Jogador.deArquivo(dadosSalvos);
} catch (e) {
```

```
    print('Erro ao carregar jogador: $e');
    salvo = null;
  }
```

O construtor `Jogador.deArquivo` é uma prévia do sistema de save/load que construiremos mais adiante. A ideia é simples: salvar o jogador como um mapa JSON e reconstruí-lo de volta.

Factory constructors

Um **factory constructor** é um construtor que tem poderes especiais: pode retornar uma instância já existente (útil para cache), pode fazer lógica complexa antes de criar o objeto e pode retornar uma subclasse em vez do tipo original. Diferente de um construtor normal, um factory não tem acesso a `this` porque pode não estar criando um novo objeto. No nosso jogo, usaremos factory constructors para construir inimigos a partir de dados, aplicando regras e validações antes de criar a instância final.

```
class Sala {
  static final Map<String, Sala> _cache = {};

  factory Sala.cacheado({
    required String id,
    required String nome,
    required String descricao,
    Map<String, String>? saidas,
    List<String>? itens,
    bool temLoja = false,
    String? inimigoId,
  }) {
    return _cache.putIfAbsent(id, () => Sala(
      id: id,
      nome: nome,
      descricao: descricao,
      saidas: saidas,
```

```
      itens: itens,
      temLoja: temLoja,
      inimigoId: inimigoId,
    ));
  }
}
```

O `factory` é diferente de um construtor normal porque pode retornar um objeto já existente (do cache), pode retornar uma instância de uma subclasse e não tem acesso a `this` no corpo.

No nosso jogo, factory constructors serão muito úteis quando criarmos inimigos a partir de dados (JSON/tabelas).

O método paraMap: preparando para persistência

O inverso de `deArquivo` é `paraMap()`, que converte o objeto para um mapa que pode ser salvo como JSON ou convertido em string. Esse par de métodos é fundamental para save/load: você salva o objeto convertendo-o para um mapa (facilmente serializado em JSON) e o carrega criando um novo objeto a partir de um mapa.

```
class Jogador {
  Map<String, dynamic> paraMap() {
    return {
      'nome': nome,
      'hp': _hp,
      'maxHp': _maxHp,
      'ouro': _ouro,
      'ataque': _ataque,
      'salaAtual': _salaAtual,
      'inventario': List<String>.from(_inventario),
    };
  }

  @override
```

```
  String toString() {
    return 'Jogador($nome, HP: $_hp/$_maxHp, '
        'Ouro: ${_ouro}g, Sala: $_salaAtual)';
  }
}
```

O par `paraMap()`/`deArquivo()` é um padrão essencial em Dart, é assim que objetos viajam para JSON e voltam. Vamos usá-lo extensivamente mais adiante.

Movimentação encapsulada

Agora que temos campos privados, podemos adicionar métodos que modificam o estado interno de forma controlada. O método `moverPara()` permite que o jogador se mova, mas apenas atualizando a sala interna. Ninguém de fora pode setar `_salaAtual = 'invalida'`: podem apenas chamar `moverPara()` e confiar que a lógica interna está correta.

```
  void moverPara(String novaSalaId) {
    _salaAtual = novaSalaId;
  }
```

E no jogo, o código de navegação fica mais limpo:

```
// Antes (Capítulo 7):
if (saidas.containsKey(direcao)) {
  salaAtual = saidas[direcao]!;
}

// Depois (Capítulo 9):
var destino = sala.saidaPara(direcao);
if (destino != null) {
  jogador.moverPara(destino);
}
```

Cada objeto cuida do que é seu. A sala sabe quais saídas tem (via `saidaPara()`). O jogador sabe como mudar de sala (via `moverPara()`). Ninguém acessa campos internos diretamente.

Desafios da Masmorra

Desafio 9.1. Sala com API protegida. Torne os campos de `Sala` que são listas (`itens`) verdadeiramente protegidos com `_`: `_itens`. Adicione métodos públicos `adicionarItem(String)` e `removerItem(String)` em vez de expor a lista diretamente. Crie um getter `List<String> get itens => List.unmodifiable(_itens)` para leitura segura.

Desafio 9.2. Construtores nomeados de dificuldade. Crie `Jogador.facil(nome)`, `Jogador.normal(nome)` e `Jogador.dificil(nome)` com stats progressivamente mais altos (HP: 50/100/150, ataque: 3/5/10, ouro inicial: 0/50/200). Teste cada um imprimindo `toString()` e verificando se os stats fazem sentido.

Desafio 9.3. Validação de movimentação. Refatore o método `moverPara(String novaSalaId)` para aceitar também o `MundoTexto` (ou `Map<String, Sala>`) do jogo. Valide se a sala destino realmente existe antes de permitir o movimento. Se não existir, lance uma `Exception` ou retorne `bool false`.

Desafio 9.4. Construtor deArquivo resiliente (Carregamento seguro). Aperfeiçoe o construtor `Jogador.deArquivo(Map<String, dynamic> dados)` com tratamento de erros: se uma chave estiver faltando ou for do tipo errado, use valores padrão em vez de crashar. Use casting seguro: `(dados['hp'] as int?) ?? 100`.

Boss Final 9.5. Padrão Copy-With (Imutabilidade). Crie ou refatore `Sala` para ser completamente imutável com `final` em todos os campos. Implemente um método `Sala copyWith({List<String>? itens, bool? temLoja})` que retorna uma nova `Sala` com as mudanças aplicadas. Demonstre com uma sequência: sala1 → adiciona item (cria sala2) → remove item (cria sala3).

Pergaminho do Capítulo

Neste capítulo você aprendeu privacidade com _ (no nível do arquivo), getters como interface pública controlada, `final` para campos imutáveis, construtores nomeados para múltiplas formas de criação, factory constructors para lógica antes da instanciação e o par `paraMap`/`deArquivo` para serialização.

O modelo do jogo agora é robusto: campos protegidos, validação interna, e uma API clara. No Capítulo 10, vamos usar herança para criar uma família de inimigos, `Zumbi`, `Esqueleto`, `Lobo`, cada um com stats e comportamentos diferentes, todos compartilhando uma base comum `Inimigo`.

Dica do Mestre: Em Dart, a regra é: torne privado por padrão, exponha por necessidade. Se um campo não precisa ser lido de fora, não crie getter. Se precisa ser lido mas não escrito, crie getter sem setter. Só exponha o mínimo necessário. Quanto menos superfície de API, menos formas o código externo tem de criar bugs no seu objeto.

Próximo Capítulo

No próximo capítulo, a masmorra ganha inimigos variados. Herança permite criar zumbis, esqueletos e goblins a partir de uma base comum.

Capítulo 10 - Herança: a família dos inimigos

Toda a masmorra é feita de famílias de criaturas. Um zumbi é um zumbi porque tem aquele espírito errante e faminto. Um esqueleto é resistente mas lento. Se você modelar cada um separadamente, o código fica repleto de cópias. Mas se você criar uma antepassada comum, uma classe `Inimigo`, você define uma vez o que qualquer criatura faz e deixa cada descendente escolher seu próprio caminho. Assim cresce a masmorra: através de herança e extends.

A família de inimigos: onde a herança brilha

Quando você começou a desenhar a `Jogador`, copiou muito código. Linhas iguais: `int hp`, `int maxHp`, `String nome`. Agora vai criar inimigos: `Zumbi`, `Esqueleto`, `Lobo`, e pode parecer que, se copiar a mesma estrutura várias vezes, em seis meses quando precisar mudar "calcular dano", vai ter de editar em múltiplos lugares. Isso se chama duplicação de código, e é o sintoma clássico de que você precisa de herança.

Herança em Dart significa: uma `class` "herda" de outra. A classe-mãe (ou superclasse) define o que é comum; a classe-filha (ou subclasse) especifica o que é diferente.

O primeiro conceito: extends

```
// lib/inimigo.dart

abstract class Inimigo {
  final String nome;
  final String simbolo;
  int hp;
  final int maxHp;
  final int ataque;
```

```
  final String descricao;

  Inimigo({
    required this.nome,
    required this.simbolo,
    required this.hp,
    required this.maxHp,
    required this.ataque,
    required this.descricao,
  });

  void sofrerDano(int d) {
    hp -= d;
    if (hp < 0) {
      hp = 0;
    }
  }

  bool get estaVivo => hp > 0;

  String descreverAcao();

  @override
  String toString() => '$nome (HP: $hp/$maxHp), $descricao';
}
```

Nota bem a palavra-chave `abstract`. Uma **classe abstrata** é um contrato de abstração: define o que toda subclasse deve fazer, mas não é uma entidade que você pode criar diretamente com `Inimigo(...)`. Isso força os criadores de zumbis, esqueletos etc. a respeitar a interface.

As três famílias: Zumbi, Esqueleto, Lobo

Agora vêm os filhos. Cada um `extends Inimigo` (herda da classe-mãe). Nem todo baú é o que parece—alguns inimigos têm natureza enganadora, como aqueles que fingem ser simples cofres de tesouro. Mas comecemos com os mais óbvios:

```dart
// lib/zumbi.dart

import 'inimigo.dart';

class Zumbi extends Inimigo {
  Zumbi()
      : super(
          nome: 'Zumbi',
          simbolo: 'Z',
          hp: 8,
          maxHp: 8,
          ataque: 3,
          descricao: 'Uma criatura de decomposição e vontade de
↪  carne.',
        );

  @override
  String descreverAcao() {
    return 'O Zumbi grunhe e avança, despedaçando o ar!';
  }
}
```

```dart
// lib/esqueleto.dart

import 'inimigo.dart';

class Esqueleto extends Inimigo {
  Esqueleto()
      : super(
          nome: 'Esqueleto',
          simbolo: 'E',
          hp: 15,
          maxHp: 15,
          ataque: 4,
          descricao: 'Ossos antigos, alma presa. '
```

```
              'Rangem com cada passo.',
        );

  @override
  String descreverAcao() {
    return 'O Esqueleto levanta o braço ósseo, '
        'você sente o frio da morte.';
  }
}
```

```
// lib/lobo.dart

import 'inimigo.dart';

class Lobo extends Inimigo {
  Lobo()
      : super(
          nome: 'Lobo',
          simbolo: 'L',
          hp: 5,
          maxHp: 5,
          ataque: 2,
          descricao: 'Uma criatura selvagem de garras afiadas.',
        );

  @override
  String descreverAcao() {
    return 'O Lobo rosna ameaçadoramente, dentes à mostra.';
  }
}
```

```
// lib/mimico.dart
// Classe em ASCII (`Mimico`); nome exibido no jogo continua
```

```
// acentuado.

import 'inimigo.dart';

class Mimico extends Inimigo {
  Mimico()
      : super(
          nome: 'Mímico',
          simbolo: 'M',
          hp: 12,
          maxHp: 12,
          ataque: 5,
          descricao: 'Um baú vivo. Nem todo tesouro é o que parece.',
        );

  @override
  String descreverAcao() {
    return 'O baú se abre de repente! Garras saem de suas laterais!';
  }
}
```

A palavra-chave @override

Quando você redefine um método da classe-mãe (como `descreverAcao()`), marca-o com `@override`. Isso diz ao analisador Dart: "Sei que estou redefinindo isto propositalmente". Se você escrever o nome errado, o Dart avisa antes de você rodar o programa:

```
@override
String descreverAcao() {
  return '...';
}
```

Como chama-se a relação IS-A

Quando você diz `class Zumbi extends Inimigo`, está dizendo: um Zumbi IS-A (é um) Inimigo. Isso significa:

- Um `Zumbi` é um `Inimigo` (pode ser usado onde se espera um `Inimigo`).
- Um `Zumbi` herda todos os campos e métodos de `Inimigo`.
- Um `Zumbi` pode redefinir (`@override`) métodos para ter comportamento específico.

Se você quiser tratar todos os inimigos de forma igual (no combate, por exemplo), você pode armazenar qualquer inimigo numa variável do tipo `Inimigo`:

```
Inimigo ini = Zumbi();
print(ini.estaVivo);
ini.sofrerDano(3);
print(ini.descreverAcao());
```

MundoTexto: o mapa de salas como um grafo

Agora você precisa de um lugar para guardar os inimigos: as salas. Uma `Sala` pode conter um inimigo. O mapa de salas é um grafo dirigido onde os nós são `Sala` e as arestas são as direções.

> **Nota sobre inimigoPresente:** No Capítulo 8, usávamos `inimigoId: String?` como simples texto. Agora, armazenamos a instância do inimigo diretamente com `inimigoPresente: Inimigo?`. Isso é mais poderoso e totalmente tipado: podemos chamar métodos do inimigo (como `inimigoPresente.sofrerDano()`) sem conversões.

```
// lib/sala.dart

import 'inimigo.dart';

class Sala {
  final String id;
```

```
  final String nome;
  final String descricao;
  final Map<String, String> saidas;
  final bool temLoja;
  Inimigo? inimigoPresente;

  Sala({
    required this.id,
    required this.nome,
    required this.descricao,
    required this.saidas,
    this.temLoja = false,
    this.inimigoPresente,
  });

  @override
  String toString() => '$nome ($id)';
}
```

```
// lib/mundo_texto.dart

class MundoTexto {
  final Map<String, Sala> salas;

  MundoTexto({required this.salas});

  Sala? obterSala(String id) => salas[id];

  bool temSaida(String salaId, String direcao) {
    final sala = obterSala(salaId);
    return sala?.saidas.containsKey(direcao) ?? false;
  }

  String? irParaDirecao(String salaId, String direcao) {
    final sala = obterSala(salaId);
```

```
    return sala?.saidas[direcao];
  }
}
```

Populando o mundo com inimigos

Aqui está como você integra tudo numa criação do mundo:

```
// lib/mundo_dados.dart

import 'inimigo.dart';
import 'zumbi.dart';
import 'esqueleto.dart';
import 'lobo.dart';
import 'sala.dart';
import 'mundo_texto.dart';

MundoTexto criarMundoVila() {
  final salas = {
    'praca': Sala(
      id: 'praca',
      nome: 'Praça da Vila',
      descricao: 'O coração da vila. Uma fonte antiga no centro.',
      saidas: {
        'norte': 'taverna',
        'leste': 'mercado',
      },
      inimigoPresente: null,
    ),
    'taverna': Sala(
      id: 'taverna',
      nome: 'Taverna do Galo Bravo',
      descricao: 'Fumo, som de risadas, cheiro a cerveja.',
      saidas: {
        'sul': 'praca',
```

```
      'norte': 'floresta',
    },
    inimigoPresente: Zumbi(),
  ),
  'mercado': Sala(
    id: 'mercado',
    nome: 'Mercado da Vila',
    descricao: 'Bancas de comida, armas, e poções.',
    saidas: {
      'oeste': 'praca',
      'norte': 'cripta',
    },
    temLoja: true,
    inimigoPresente: null,
  ),
  'floresta': Sala(
    id: 'floresta',
    nome: 'Floresta Escura',
    descricao: 'Árvores altas. Sons estranhos na escuridão.',
    saidas: {
      'sul': 'taverna',
      'norte': 'caverna',
    },
    inimigoPresente: Lobo(),
  ),
  'cripta': Sala(
    id: 'cripta',
    nome: 'Cripta Antiga',
    descricao: 'Lápides rotas. Silêncio assustador.',
    saidas: {
      'sul': 'mercado',
    },
    inimigoPresente: Esqueleto(),
  ),
  'caverna': Sala(
    id: 'caverna',
```

```
      nome: 'Caverna do Dragão',
      descricao: 'Escura demais. Você sente respiração quente.',
      saidas: {
        'sul': 'floresta',
      },
      inimigoPresente: null,
    ),
  };

  return MundoTexto(salas: salas);
}
```

Desafios da Masmorra

Desafio 10.1. Novo tipo de inimigo (Orc). Crie uma classe `Orc` que estende `Inimigo`. Dê-lhe: HP=12, maxHp=12, ataque=5, símbolo='O', e uma descrição agressiva ("Um orc musculoso com fome de batalha"). Sobrescreva `descreverAcao()` para retornar algo temível como "O orc rosna e levanta sua clava!". Teste criando uma instância e imprimindo.

Desafio 10.2. Popule o mundo com Orcs. Mude a `cripta` no `MundoTexto` para ter um `Orc` em vez de `Esqueleto`. Verifique se o símbolo `'O'` aparece corretamente. Adicione também um `Orc` em outra sala, por exemplo a `caverna`.

Desafio 10.3. Método em MundoTexto (Listar todos). Escreva um método `List<Inimigo> todosOsInimigos()` em `MundoTexto` que devolve uma lista com todos os inimigos das salas (filtrando nulos). Teste imprimindo um relatório de todos os inimigos encontrados, mostrando nome, tipo e HP.

Desafio 10.4. Sala de Combate obrigatório. Crie uma classe `SalaCombate extends Sala` que força a derrota do inimigo antes de permitir sair. Adicione um método `bool podeSair()` que verifica se o `inimigoPresente` está vivo. O jogador pode executar ações normais, mas `"sair"` retorna erro se o inimigo não estiver derrotado.

Desafio 10.5. Hierarquia de três níveis (Avó e netos). Crie uma classe `BipedeInteligente extends Inimigo` (sem `abstract`) que adiciona um campo `inteligencia: int` e um método `String insulto()`. Depois crie `Zumbi`

e `Orc` estendendo `BipedeInteligente` e sobrescrevendo `insulto()` com mensagens diferentes. Teste a hierarquia: Zumbi → BipedeInteligente → Inimigo.

Boss Final 10.6. Integrar combate ao game loop. Refatore o game loop do Capítulo 7 para usar a classe `Jogador` em vez de variáveis soltas. Depois, modifique as salas para conter inimigos (use `Sala.inimigoPresente`). Quando o jogador entrar numa sala com inimigo, mostre: "Um [Zumbi] está aqui! [Z] HP: 5/8". Adicione o comando `"atacar"` que reduz o HP do inimigo e toca um turno do inimigo atacando de volta. Sem a classe `Combate` ainda; apenas lógica simples de turnos. Quando o inimigo morre, a sala fica segura.

Pergaminho do Capítulo

Neste capítulo você aprendeu:

- Herança (`extends`) permite que uma `class` herde campos e métodos de outra.
- `abstract class` definem um contrato que as subclasses devem cumprir.
- `@override` marca explicitamente que você está redefinindo um método da classe-mãe.
- IS-A: um `Zumbi` IS-A `Inimigo`, pode ser usado onde se espera um `Inimigo`.
- `MundoTexto` encapsula um `Map<String, Sala>`, modelando o mapa como um grafo dirigido.
- Salas podem conter inimigos, criando o cenário para combate no próximo capítulo.

A herança é a ferramenta clássica para eliminar duplicação quando há uma relação clara "tipo de". No próximo capítulo, veremos mixins, que servem para compartilhar comportamento sem forçar uma árvore de herança profunda.

Dica do Mestre: Evite hierarquias profundas. A cada vez que você adiciona um nível de herança, aumenta a complexidade. Depois de três níveis (`Inimigo > BipedeInteligente > Zumbi`), fica difícil compreender onde cada comportamento vem. Use herança quando há uma razão clara para IS-A; caso contrário, prefira composição (guardar um objeto dentro de outro) ou `mixin`. Dart favorece composição e `mixin` para código mais limpo.

Próximo Capítulo

No próximo capítulo, descobrimos poderes compartilhados. Mixins permitem que qualquer criatura ganhe habilidades sem herança múltipla.

Capítulo 11 - Mixins: poderes compartilhados

Toda criatura que respira sente dano e sangra. O dragão, o mago, o zumbi—todos sofrem golpes da mesma forma. Em vez de copiar essa verdade a cada classe, Dart oferece uma verdade compartilhada: um mixin é como um poder que você injeta em qualquer criatura sem reescrever sua árvore familiar.

O problema que mixins resolvem

No capítulo anterior, você criou `Inimigo` com o método `sofrerDano()`. Agora você tem um `Jogador` que também precisa de `sofrerDano()`, e mais tarde talvez tenha `Besta` ou `Golem` com a mesma lógica. Se você copiar a implementação várias vezes, quando tiver de corrigir um bug ou mudar as regras, tem de editar em muitos lugares.

Mixins são a solução: é um pacote reutilizável de comportamento que você pode aplicar a várias classes sem necessidade de herança.

Entender `mixin` e with

Um `mixin` é definido de forma parecida a uma `class`, mas usa a palavra-chave `mixin`:

> **Nota sobre encapsulamento:** Você vai notar que os campos `hp` e `maxHp` aparecem aqui sem sublinhado (_), ao contrário do que aprendemos no capítulo 9. Há uma razão técnica: mixins em Dart não podem declarar campos privados que sejam acessados pelas classes que os usam — a privacidade por sublinhado funciona no nível de arquivo. Para manter o código simples enquanto aprendemos o conceito, deixaremos os campos expostos aqui. Nos capítulos seguintes, quando consolidarmos `Jogador` e `Inimigo` como classes concretas, voltaremos a encapsular esses campos atrás de getters públicos e setters controlados. É uma exposição temporária, consciente, não um recuo.

```dart
// lib/combatente.dart

mixin Combatente {
  int hp = 0;
  int maxHp = 0;

  void sofrerDano(int d) {
    hp -= d;
    if (hp < 0) {
      hp = 0;
    }
    print('Sofri $d de dano! HP agora é $hp');
  }

  void curar(int q) {
    hp += q;
    if (hp > maxHp) {
      hp = maxHp;
    }
    print('Curado por $q. HP agora é $hp');
  }

  bool get estaVivo => hp > 0;

  String mostrarBarraVida() {
    final preenchimento = '' * (hp ~/ (maxHp ~/ 10 + 1));
    final vazio = '' * (10 - preenchimento.length);
    return '[$preenchimento$vazio] $hp/$maxHp';
  }
}
```

Agora, quando você criar um `Jogador`, aplique este mixin com a palavra-chave `with`:

```dart
// lib/jogador.dart

import 'combatente.dart';

class Jogador with Combatente {
  String nome;
  String classe;
  int nivel = 1;
  List<Item> inventario = [];
  Item? armaEquipada;

  Jogador({
    required this.nome,
    required this.classe,
    required int hpInicial,
  }) {
    hp = hpInicial;
    maxHp = hpInicial;
  }

  @override
  String toString() =>
      '$nome [$classe, nível $nivel], '
      '${mostrarBarraVida()}';

  void adicionarItem(Item item) {
    inventario.add(item);
    print('Você obteve ${item.nome}!');
  }

  void equiparArma(Item item) {
    if (item is Arma) {
      armaEquipada = item;
      print('Você equipou ${item.nome}');
    } else {
      print('Você não pode equipar isto.');
```

```
    }
  }
}
```

E também aplica a `Inimigo`:

```
// lib/inimigo.dart

import 'combatente.dart';

abstract class Inimigo with Combatente {
  final String nome;
  final String simbolo;
  final int ataque;
  final String descricao;

  Inimigo({
    required this.nome,
    required this.simbolo,
    required int hp,
    required int maxHp,
    required this.ataque,
    required this.descricao,
  }) {
    this.hp = hp;
    this.maxHp = maxHp;
  }

  String descreverAcao();

  @override
  String toString() => '$nome ${mostrarBarraVida()}, $descricao';
}
```

A diferença: extends (IS-A) vs with (HAS-A-BEHAVIOR)

Isso é crucial para entender quando usar cada um:

Ferramenta	Significado	Exemplo
`extends`	IS-A	`class Zumbi extends Inimigo`: "Um Zumbi IS-A (é um) tipo de Inimigo"
`with`	HAS-A-BEHAVIOR	`class Jogador with Combatente`: "Um Jogador HAS-A comportamento Combatente"

Quando você diz `extends`, você está dizendo "isto é um tipo especializado daquilo". Quando você diz `with`, você está dizendo "isto compartilha este conjunto de comportamentos, mas a sua natureza é diferente".

Um exemplo prático

```
final jogador = Jogador(
  nome: 'Herói',
  classe: 'Guerreiro',
  hpInicial: 30,
);
final zumbi = Zumbi();

jogador.sofrerDano(5);
zumbi.sofrerDano(3);

if (jogador is Zumbi) {
  print('Isto é falso!');
}

if (jogador is Combatente && zumbi is Combatente) {
```

```
  print('Ambos sabem combater! Usam o mixin Combatente');
}
```

Restrições de mixins: on keyword

Por vezes, um `mixin` depende de propriedades específicas. Por exemplo, um `mixin Envenenavel` precisa de acesso ao campo `hp` e ao método `sofrerDano()`. Use a palavra-chave `on` para declarar isso:

```
// lib/envenenavel.dart

import 'combatente.dart';

mixin Envenenavel on Combatente {
  int veneno = 0;

  void envenenar(int quantidade) {
    veneno += quantidade;
    print('Veneno acumulado: $veneno!');
  }

  void aplicarDanoVeneno() {
    if (veneno > 0) {
      sofrerDano(veneno);
      veneno = 0;
    }
  }
}
```

Agora você pode fazer (uma `class` com múltiplos `mixin`):

```
abstract class Inimigo with Combatente, Envenenavel {
  // Inimigo tem acesso a:
  // - sofrerDano(), curar(), estaVivo (de Combatente)
```

```
  // - envenenar(), aplicarDanoVeneno() (de Envenenavel)
}
```

Vários mixins na mesma classe

Você pode aplicar múltiplos `mixin`. Isso é poderoso em Dart (ao contrário de linguagens que só permitem uma classe-mãe):

```
// lib/descritivel.dart

mixin Descritivel {
  String get descricaoCompleta => 'Uma criatura indescritível.';

  void apresentar() {
    print('Sou: $descricaoCompleta');
  }
}
```

```
class Zumbi extends Inimigo with Combatente, Envenenavel, Descritivel
↪  {
  // Agora tem todas as capacidades dos mixins
}
```

Integração no combate

Vê como tudo se junta quando há combate:

```
// lib/turno_combate.dart

class TurnoCombate {
  final Jogador jogador;
  final Inimigo inimigo;
```

```
  TurnoCombate(this.jogador, this.inimigo);

  void atacarInimigo(int dano) {
    print('${jogador.nome} ataca!');
    inimigo.sofrerDano(dano);
    print(inimigo.mostrarBarraVida());

    if (!inimigo.estaVivo) {
      print('${inimigo.nome} foi derrotado!');
      return;
    }

    print('${inimigo.nome} contra-ataca!');
    jogador.sofrerDano(inimigo.ataque);
    print('${jogador.nome}: ${jogador.mostrarBarraVida()}');
  }

  void executarCombate() {
    while (jogador.estaVivo && inimigo.estaVivo) {
      print('\n--- Turno ---');
      print('Ataca o ${jogador.nome}?');
      atacarInimigo(5);
    }

    if (jogador.estaVivo) {
      print('Vitória! ${inimigo.nome} foi derrotado!');
    } else {
      print('Derrota... ${jogador.nome} morreu.');
    }
  }
}
```

Desafios da Masmorra

Desafio 11.1. Mixin Herbívoro. Crie um mixin `Herbivoro` com um método `comer(String planta)` que imprime "Comi uma $planta! Recuperei 3 HP." e chama `curar(3)`. Aplique-o a uma classe concreta `Coelho` que também herda de `Inimigo with Combatente`. Teste comendo uma maçã.

Desafio 11.2. Aplicar Combatente ao Jogador (Integração). Certifique-se de que a sua classe `Jogador` usa `with Combatente`. Teste `sofrerDano()` e `mostrarBarraVida()` no main. Verifique se a barra de vida funciona corretamente durante combate.

Desafio 11.3. Mixin Voador. Crie um mixin `Voador` com `bool estaNoAr = false` e métodos `voar()` (coloca `estaNoAr = true`), `pousar()` (coloca `false`). Crie uma classe `Dragao extends Inimigo with Combatente, Voador`. O dragão pode voar enquanto está em combate (aumentando sua defesa?).

Desafio 11.4. Mixin restrito com on. Crie um mixin `Regenerador on Combatente` que tem um método `regenerar()` que cura 2 HP por turno. Aplique-o a `Inimigo with Combatente, Regenerador`. O inimigo deve regenerar 2 HP ao final de cada turno de combate.

Boss Final 11.5. Múltiplos mixins e resolução de conflito. Crie dois mixins `Lutador` e `Mago`, ambos com métodos `atacar()` que retornam `String`. Depois crie uma classe `Paladim extends Inimigo with Combatente, Lutador, Mago`. Como Dart resolve o conflito? (O último mixin, `Mago`, ganha.) Teste implementando `String atacar()` em ambos e veja qual é chamado. Demonstre a ordem de resolução.

Pergaminho do Capítulo

Neste capítulo você aprendeu:

- `mixin` são "pacotes reutilizáveis de comportamento" definidos com `mixin`.
- `with` aplica um `mixin` a uma `class`. Uma `class` pode ter múltiplos `mixin`.
- `extends` (IS-A) vs `with` (HAS-A-BEHAVIOR): `extends` é para hierarquias, `with` é para compartilhar comportamento.
- `on` keyword restringe um `mixin` a apenas funcionar com `class` que já têm outro `mixin`.
- Múltiplos `mixin` são poderosos: `class Zumbi with Combatente, Envenenavel, Descritivel`.
- Quando há conflito de nomes (dois `mixin` com `atacar()`), o último `mixin` ganha. Melhor: usar nomes distintos.

Mixins são particularmente úteis em jogos, onde muitos tipos diferentes de entidades (jogador, inimigos, objetos) compartilham capacidades (receber dano, mover, descrição).

Dica do Mestre: Mixins resolvem o *Diamond Problem* melhor que herança múltipla. Em linguagens como C++, herança múltipla pode criar confusão sobre qual classe-mãe fornece qual método. Dart evita isso: é explícito (`with` te diz que é um `mixin`). Se uma `class Zumbi` usa `with Combatente, Envenenavel, Descritivel`, toda gente sabe que tem esses comportamentos. Quando há dúvida sobre compartilhar código, mixins são geralmente a resposta mais limpa do que herança profunda.

Próximo Capítulo

No próximo capítulo, organizamos os comandos do jogo. Enums e um parser transformam texto digitado em ações do herói.

Capítulo 12 - Enums e o parser de comandos

A masmorra entende apenas algumas palavras: "norte", "ataca", "inventário". Se o jogador sussurra algo estranho, o jogo não compreende. Os enums são como um dicionário fechado—só existem as palavras que você definiu, nada mais, nada menos.

Enums: tipos fechados e pequenos

Um **enum** (enumeração) é um tipo que só pode ter um conjunto predefinido de valores. É perfeito para coisas que não mudam: direções cardeais, dias da semana, fases da lua. Use `enum` para valores imutáveis:

```
// lib/direcao.dart

enum Direcao {
  norte,
  sul,
  leste,
  oeste,
}
```

Simples e poderoso:

```
void main() {
  final direcao = Direcao.norte;
  print(direcao);
}
```

Dart garante que `direcao` é uma das quatro opções. Nada de erros estranhos como `direcao = 'septentriao'` (typo). O compilador não permite valores inválidos.

Enums com membros (Dart 3+)

Em Dart 3, `enum` podem ter propriedades, construtores e métodos (use `const` no construtor):

```
// lib/direcao.dart

enum Direcao {
  norte(simbolo: '↑', id: 'n'),
  sul(simbolo: '↓', id: 's'),
  leste(simbolo: '→', id: 'e'),
  oeste(simbolo: '←', id: 'o');

  final String simbolo;
  final String id;

  const Direcao({required this.simbolo, required this.id});

  Direcao get oposta {
    switch (this) {
      case Direcao.norte:
        return Direcao.sul;
      case Direcao.sul:
        return Direcao.norte;
      case Direcao.leste:
        return Direcao.oeste;
      case Direcao.oeste:
        return Direcao.leste;
    }
  }

  static Direcao? deString(String s) {
    switch (s.toLowerCase()) {
      case 'n':
      case 'norte':
        return Direcao.norte;
      case 's':
```

```
      case 'sul':
        return Direcao.sul;
      case 'e':
      case 'leste':
        return Direcao.leste;
      case 'o':
      case 'oeste':
        return Direcao.oeste;
      default:
        return null;
    }
  }
}
```

Uso:

```
void main() {
  final dir = Direcao.norte;
  print('Vou para ${dir.simbolo}');

  final dirOposta = dir.oposta;
  print('Oposta: ${dirOposta.simbolo}');

  final dirDoInput = Direcao.deString('n');
  print(dirDoInput);
}
```

Sealed classes: comandos com tipagem estrita

Agora vem o prato forte. `sealed class` são uma forma de Dart de dizer "esta hierarquia está fechada, apenas estas subclasses existem". É perfeito para command (padrão de design), porque cada comando é diferente e tem argumentos diferentes. Dart também oferece extensions para adicionar métodos a tipos existentes sem herança e typedefs para nomear assinaturas de função complexas. Use `sealed class` para hierarquias fechadas:

```dart
// lib/comando_jogo.dart

import 'direcao.dart';

sealed class ComandoJogo {
  const ComandoJogo();

  String executar(); // Método abstrato
}

class ComandoMover extends ComandoJogo {
  final Direcao direcao;

  const ComandoMover(this.direcao);

  @override
  String executar() => 'Movendo para $direcao';
}

class ComandoAtacar extends ComandoJogo {
  final String alvo;

  const ComandoAtacar(this.alvo);

  @override
  String executar() => 'Atacando $alvo!';
}

class ComandoPegar extends ComandoJogo {
  final String item;

  const ComandoPegar(this.item);

  @override
  String executar() => 'Você pegou $item';
}
```

```
class ComandoInventario extends ComandoJogo {
  const ComandoInventario();

  @override
  String executar() => 'Abrindo inventário...';
}

class ComandoOlhar extends ComandoJogo {
  const ComandoOlhar();

  @override
  String executar() => 'Você observa ao seu redor...';
}

class ComandoStatus extends ComandoJogo {
  const ComandoStatus();

  @override
  String executar() => 'Mostrando status...';
}

class ComandoAjuda extends ComandoJogo {
  const ComandoAjuda();

  @override
  String executar() =>
      'Comandos: norte/sul/leste/oeste, atacar, pegar, inv, '
      'status, olhar, ajuda, sair';
}

class ComandoSair extends ComandoJogo {
  const ComandoSair();

  @override
  String executar() => 'Até logo!';
```

```
}

class ComandoDesconhecido extends ComandoJogo {
  final String entrada;

  const ComandoDesconhecido(this.entrada);

  @override
  String executar() => 'Não entendo "$entrada". Tenta "ajuda".';
}
```

O parser: transformar texto em comandos

Agora o coração da magia: uma função que lê uma linha de texto e devolve um `ComandoJogo` tipado.

```
// lib/parser.dart

import 'comando_jogo.dart';
import 'direcao.dart';

ComandoJogo analisarLinha(String entrada) {
  final linha = entrada.trim().toLowerCase();

  if (linha.isEmpty) {
    return const ComandoDesconhecido('(vazio)');
  }

  final palavras = linha.split(RegExp(r'\s+'));
  final verbo = palavras[0];
  final args = palavras.length > 1 ? palavras.sublist(1) : [];

  switch (verbo) {
    case 'n':
    case 'norte':
```

```
      return const ComandoMover(Direcao.norte);

    case 's':
    case 'sul':
      return const ComandoMover(Direcao.sul);

    case 'e':
    case 'leste':
      return const ComandoMover(Direcao.leste);

    case 'o':
    case 'oeste':
      return const ComandoMover(Direcao.oeste);

    case 'atacar':
    case 'a':
      if (args.isEmpty) {
        return const ComandoDesconhecido('atacar o quê?');
      }
      final alvo = args.join(' ');
      return ComandoAtacar(alvo);

    case 'inv':
    case 'inventario':
    case 'i':
      return const ComandoInventario();

    case 'pegar':
    case 'p':
      if (args.isEmpty) {
        return const ComandoDesconhecido('pegar o quê?');
      }
      final item = args.join(' ');
      return ComandoPegar(item);

    case 'status':
```

```
      return const ComandoStatus();

    case 'olhar':
    case 'ver':
    case 'l':
      return const ComandoOlhar();

    case 'ajuda':
    case 'help':
    case '?':
      return const ComandoAjuda();

    case 'sair':
    case 'quit':
    case 'exit':
      return const ComandoSair();

    default:
      return ComandoDesconhecido(entrada);
  }
}
```

Switch exaustivo com sealed classes

Isso é onde Dart brilha. Quando você faz um `switch` sobre uma `sealed class`, o compilador força-te a tratar todos os casos (switch exaustivo):

```
// lib/loop_jogo.dart

import 'comando_jogo.dart';
import 'parser.dart';

class LoopJogo {
  void processarComando(ComandoJogo cmd) {
    switch (cmd) {
```

```
      case ComandoMover(:final direcao):
        print('Movendo para $direcao...');

      case ComandoAtacar(:final alvo):
        print('Atacando $alvo...');

      case ComandoPegar(:final item):
        print('Pegando em $item...');

      case ComandoInventario():
        print('Mostrando inventário...');

      case ComandoOlhar():
        print('Observando...');

      case ComandoStatus():
        print('Mostrando status...');

      case ComandoAjuda():
        print('Mostrando ajuda...');

      case ComandoSair():
        print('Saindo do jogo...');

      case ComandoDesconhecido(:final entrada):
        print('Comando desconhecido: $entrada');
    }
  }

  void mainLoop() {
    while (true) {
      print('> ');
      final entrada = stdin.readLineSync() ?? '';

      final cmd = analisarLinha(entrada);
      processarComando(cmd);
```

```
    }
  }
}
```

Pattern matching com extração

Note a sintaxe especial: `case ComandoAtacar(:final alvo)`. Isso é pattern matching. Extrai o campo `alvo` diretamente no `case`, tornando o código mais conciso:

```
// Sem pattern matching (mais verboso)
case ComandoAtacar cmd:
  final alvo = cmd.alvo;
  print('Atacando $alvo');
  break;

// Com pattern matching (mais elegante)
case ComandoAtacar(:final alvo):
  print('Atacando $alvo');
```

Integração completa: do input ao jogo

Veja como tudo flui. Note o `import 'dart:io';` necessário para `stdin` e `stdout`:

```
// lib/main.dart

import 'dart:io';
import 'comando_jogo.dart';
import 'parser.dart';
import 'mundo_texto.dart';
import 'mundo_dados.dart';

void main() {
  final mundo = criarMundoVila();
```

```
var salaAtual = 'praca';

print('=== MASMORRA ASCII ===');
print('Digite "ajuda" para ver comandos.\n');

while (true) {
  final sala = mundo.obterSala(salaAtual);
  print('\n[${sala!.nome}]');
  print(sala.descricao);

  if (sala.inimigoPresente != null) {
    final ini = sala.inimigoPresente!;
    print('Aqui está um ${ini.nome} (${ini.simbolo})!');
  }

  print('Saídas: ${sala.saidas.keys.join(", ")}');

  stdout.write('> ');
  final entrada = stdin.readLineSync() ?? '';

  final cmd = analisarLinha(entrada);

  switch (cmd) {
    case ComandoMover(:final direcao):
      final dirStr = direcao.id;
      if (mundo.temSaida(salaAtual, dirStr)) {
        salaAtual = mundo.irParaDirecao(salaAtual, dirStr)!;
        print('Você se moveu para ${direcao.simbolo}');
      } else {
        print('Você não pode ir para $direcao.');
      }

    case ComandoAtacar(:final alvo):
      final sala2 = mundo.obterSala(salaAtual);
      if (sala2?.inimigoPresente != null) {
        print('Você atacou ${sala2!.inimigoPresente!.nome}!');
```

```
        } else {
          print('Não há nada para atacar aqui.');
        }

      case ComandoPegar(:final item):
        print('Você procurou por $item, mas não encontrou nada.');

      case ComandoInventario():
        print('Inventário vazio.');

      case ComandoOlhar():
        print('(você já vê isto)');

      case ComandoAjuda():
        print(cmd.executar());

      case ComandoSair():
        print('Até logo!');
        return;

      case ComandoDesconhecido(:final entrada):
        print(cmd.executar());
    }
  }
}
```

Desafios da Masmorra

Desafio 12.1. Estender o enum Direcao (Direções diagonais). Adicione `nordeste`, `noroeste`, `sudeste`, `sudoeste` como novos membros ao enum. Cada um deve ter um símbolo apropriado (`'↗'`, `'↖'`, `'↘'`, `'↙'`) e id curto (`'ne'`, `'nw'`, `'se'`, `'sw'`). Atualize o método `oposta()` também.

Desafio 12.2. Novo comando ComandoEquipar. Crie uma sealed subclass `ComandoEquipar` com um campo `arma: String`. Adicione-a ao parser

quando o jogador escreve “equipar espada” ou “eq lança”. Teste que o parser extrai o nome da arma corretamente.

Desafio 12.3. Sinonímia no parser (Abreviações). Adicione abreviações para direções: `"u"` (up) para norte, `"d"` (down) para sul, `"l"` para leste, `"o"` para oeste. Teste que `analisarLinha("u")` retorna `ComandoMover(Direcao.norte)` e `analisarLinha("inv")` retorna `ComandoInventario()`.

Desafio 12.4. Sugestão de comando semelhante. Quando o jogador escreve um comando desconhecido, em vez de apenas retornar `ComandoDesconhecido(entrada)`, verifique se é similar a um comando válido (ex.: “atlcar” ≈ “atacar”) e sugira: “Você quis dizer ‘atacar’? Tente novamente.”

Boss Final 12.5. Comando ComandoFala (Fala com argumento). Crie `ComandoFala` que aceita uma frase inteira (ex.: `falar "Olá, mundo!"`). Modifique o parser para capturar tudo após “falar” como argumento único (pode incluir múltiplas palavras e pontuação). Demonstre com uma frase completa.

Pergaminho do Capítulo

Neste capítulo você aprendeu:

- `enum` definem tipos fechados com um conjunto finito de valores.
- `enum` com membros (Dart 3+) podem ter propriedades, construtores e métodos.
- `sealed class` são hierarquias fechadas, apenas as subclasses declaradas podem existir.
- Parser transforma texto em objetos tipados, eliminando strings soltas e ambíguas.
- Pattern matching (`case ComandoAtacar(:final alvo)`) extrai dados `inline`.
- `switch` exaustivo força você a tratar todos os casos de uma `sealed class`; o compilador avisa se você esquecer algum.

`enum` e `sealed class` são ferramentas poderosas para tornar o código mais seguro. Quando combinadas, garantem que cada comando é um tipo diferente (nenhuma confusão), cada comando tem os campos corretos (não há erros de acesso), e o código que despacha comandos trata todos os casos (compilador força isso).

Dica do Mestre: `sealed class` + `switch` exaustivo = refatoração segura. Imagine que você adiciona um novo comando `ComandoMagia` num projeto grande. Com `sealed class`, o compilador anuncia cada lugar onde você faz `switch` sobre `ComandoJogo`, dizendo “ei, você esqueceu de tratar `ComandoMagia`!”. Em linguagens sem `sealed class`, você recebe silenciosamente um `default`

anônimo e a lógica fica incompleta. `sealed class` transformam erros em tempo de execução em avisos em tempo de compilação. Isso é refatoração segura.

Próximo Capítulo

No próximo capítulo, entramos na Parte III. O herói ganha ouro, armas e um inventário real. A masmorra começa a ter economia.

Capítulo 13 - Ouro, Armas e Inventário

Toda mochila de aventureiro carrega histórias. Um bastão em mãos certas é a diferença entre vida e tumba. O ouro abre portas, as poções salvam vidas, as armas transformam um novato em guerreiro. O que você carrega define quem você é.

O que vamos aprender

Neste capítulo você vai: - Modelar itens como classes (identidade, preço, peso) - Criar hierarquia de classes com `extends` (Arma estende Item) - Implementar um sistema de equipamento com slots e validações - Construir um sistema de economia (ouro, compra, venda) - Entender como equipamento afeta estatísticas (dano total, defesa total)

Ao final, você terá um **inventário** (mochila) completamente funcional que o jogo possa usar para compras, trocas, e combate.

Parte 1: Pensando em Itens. Abstração

Antes de codificar, vamos pensar como um designer. Um item numa masmorra tem características gerais: - Um identificador único (ID) - Um nome legível (o que mostra na tela) - Uma descrição (sabor do jogo) - Um preço (quanto custa comprar) - Um peso (realismo mínimo)

Mas nem todos os itens são iguais. Uma espada é um `Item`, mas precisa de `dano`. Uma armadura é `Item`, mas precisa de `defesa`. Uma poção é `Item`, mas precisa de `efeitoHP`.

Essa é a oportunidade perfeita para herança de classes: todos os itens compartilham estrutura comum, mas cada tipo especializa isso de forma diferente.

Conceito: Herança com extends

Em Dart, você pode criar uma `class` base (`Item`) e depois especializá-la com `extends`. Todos os itens têm propriedades comuns (nome, preço, peso), mas

cada tipo especializa isso de forma diferente. Vamos começar com a classe `Item` genérica que serve como base para todos os itens da masmorra.

```
// lib/item.dart - A classe genérica

class Item {
  final String id;
  final String nome;
  final String descricao;
  final int preco;
  final int peso;

  Item({
    required this.id,
    required this.nome,
    required this.descricao,
    required this.preco,
    required this.peso,
  });

  @override
  String toString() =>
      '$nome (id: $id, preço: $preco ouro, '
      'peso: $peso)';
}
```

Agora, uma arma é tudo que `Item` é, mais dano. Use `extends` para herança:

```
// lib/arma.dart

class Arma extends Item {
  final int dano;
  final String tipo;

  Arma({
```

```
    required String id,
    required String nome,
    required String descricao,
    required int preco,
    required int peso,
    required this.dano,
    required this.tipo,
  }) : super(
    id: id,
    nome: nome,
    descricao: descricao,
    preco: preco,
    peso: peso,
  );

  @override
  String toString() => '$nome ($tipo, +$dano dano)';
}
```

Nota importante: quando você faz `class Arma extends Item`, a `class Arma` herda todos os atributos de `Item`. Por isso você passa `id`, `nome` etc. ao construtor `super()`, assim a classe-pai é inicializada corretamente.

Testando a Herança

Agora vamos criar alguns items concretos e testar como herança permite que `Arma` reutilize todos os campos de `Item`, adicionando apenas o que é específico de armas. Observe como o construtor de `Arma` passa os dados genéricos via `super()` para inicializar a classe-mãe.

```
void main() {
  final pocao = Item(
    id: 'pocao-simples',
    nome: 'Poção de Vida',
    descricao: 'Recupera 10 HP',
```

```
    preco: 50,
    peso: 1,
  );

  final espada = Arma(
    id: 'espada-bastarda',
    nome: 'Espada Bastarda',
    descricao: 'Uma lâmina versátil de dois gumes',
    preco: 300,
    peso: 4,
    dano: 12,
    tipo: 'cortante',
  );

  print(pocao);
  print(espada);

  print('Preço: ${espada.preco}');
}
```

Por que herança (`extends`) é boa aqui? - Reutilização: não repetimos `id`, `nome`, `descrição`, etc. em cada `class`. - Polimorfismo: numa `List<Item>` você pode misturar itens genéricos, armas, armaduras. - Manutenibilidade: se mudar o que um `Item` é, automaticamente todas as subclasses mudam.

Parte 2: Armadura

Vamos criar `class Armadura` pelo mesmo princípio (`extends Item`). Assim como `Arma`, uma `Armadura` herda todas as propriedades de `Item` (nome, preço, peso) e adiciona seu próprio comportamento específico (quanto de defesa oferece e em qual parte do corpo se encaixa).

```
// lib/armadura.dart

class Armadura extends Item {
```

```
  final int defesa;
  final String localizacao;

  Armadura({
    required String id,
    required String nome,
    required String descricao,
    required int preco,
    required int peso,
    required this.defesa,
    required this.localizacao,
  }) : super(
    id: id,
    nome: nome,
    descricao: descricao,
    preco: preco,
    peso: peso,
  );

  @override
  String toString() => '$nome (+$defesa DEF em $localizacao)';
}
```

Agora temos:

```
final peitoral = Armadura(
  id: 'peitoral-couro',
  nome: 'Peitoral de Couro Endurecido',
  descricao: 'Proteção leve e flexível',
  preco: 150,
  peso: 3,
  defesa: 5,
  localizacao: 'peito',
);
```

```
print(peitoral);
```

Parte 3: O Inventário. Uma Lista com Propósito

O jogador tem uma mochila: uma `List<Item>` onde qualquer `Item` cabe (herança permite polimorfismo). Esse é o poder da herança em ação: graças a `Arma extends Item` e `Armadura extends Item`, você pode guardar qualquer tipo de item numa única lista, sem necessidade de casts ou verificações de tipo. A mochila não precisa saber se contém uma arma ou uma poção, só sabe que são itens.

```
// lib/jogador.dart (trecho)

class Jogador {
  String nome;
  int hp;
  int maxHp;
  int ouro;
  List<Item> inventario;

  Jogador({
    required this.nome,
    required this.maxHp,
    required this.ouro,
    this.inventario = const [],
  }) : hp = maxHp;

  void adicionarItem(Item item) {
    inventario.add(item);
  }

  void listarInventario() {
    if (inventario.isEmpty) {
      print('Sua mochila está vazia.');
      return;
```

```
    }
    print('\n=== INVENTÁRIO ===');
    for (int i = 0; i < inventario.length; i++) {
      print('${i + 1}. ${inventario[i]}');
    }
  }

  Item? removerItem(int indice) {
    if (indice < 0 || indice >= inventario.length) {
      return null;
    }
    return inventario.removeAt(indice);
  }
}
```

Vantagem de indexar: quando você escreve `vender 2`, você sabe exatamente qual item é o segundo da lista.

Parte 4: Equipamento. Slots e Validação

Equipar uma arma significa tirar da mochila e pôr na mão. Para isso, o jogador precisa de slots (variáveis que guardam qual item está equipado). Note o operador `is`: você usa `is Arma` e `is Armadura` para verificar o tipo específico do item antes de tentar equipar. Isso é crucial porque nem todo `Item` é uma arma.

```
// lib/jogador.dart (continuação)

class Jogador {
  Arma? armaEquipada;
  Armadura? armaduraEquipada;

  bool equiparArma(int indiceNoInventario) {
    final item = inventario[indiceNoInventario];
```

```
  if (item is! Arma) {
    print('Isso não é uma arma!');
    return false;
  }

  if (armaEquipada != null) {
    inventario.add(armaEquipada!);
  }

  inventario.removeAt(indiceNoInventario);
  armaEquipada = item;
  print('Você equipou ${item.nome}!');
  return true;
}

void desequiparArma() {
  if (armaEquipada == null) {
    print('Você não tem uma arma equipada!');
    return;
  }
  inventario.add(armaEquipada!);
  print('Você desequipou ${armaEquipada!.nome}.');
  armaEquipada = null;
}

bool equiparArmadura(int indiceNoInventario) {
  final item = inventario[indiceNoInventario];

  if (item is! Armadura) {
    print('Isso não é uma armadura!');
    return false;
  }

  if (armaduraEquipada != null) {
    inventario.add(armaduraEquipada!);
  }
```

```
    inventario.removeAt(indiceNoInventario);
    armaduraEquipada = item;
    print('Você equipou ${item.nome}!');
    return true;
  }

  void desequiparArmadura() {
    if (armaduraEquipada == null) {
      print('Você não tem armadura equipada!');
      return;
    }
    inventario.add(armaduraEquipada!);
    print('Você desequipou ${armaduraEquipada!.nome}.');
    armaduraEquipada = null;
  }
}
```

O operador `is`: `item is Arma` pergunta "item é do tipo `Arma`?". Se não for, retorna `false` e você trata de forma apropriada. Também existe `is!` para "não é". Seguro e legível.

Parte 5: Dano Total. Equação Simples

Agora, o dano do jogador não é mais fixo. Depende do que ele está carregando. Uma das maravilhas de um sistema de equipamento real é que as estatísticas são dinâmicas: se você equipa uma espada melhor, o dano total sobe imediatamente. Isto usa o padrão `get` do Dart (uma propriedade calculada) para sempre retornar o dano atualizado.

```
// lib/jogador.dart

class Jogador {
  int danoBase = 5;
```

```
  int get danoTotal {
    int total = danoBase;
    if (armaEquipada != null) {
      total += armaEquipada!.dano;
    }
    return total;
  }

  int get defesaTotal {
    int total = 2;
    if (armaduraEquipada != null) {
      total += armaduraEquipada!.defesa;
    }
    return total;
  }

  void mostraStatus() {
    print('\n== STATUS ==');
    print('HP: $hp/$maxHp');
    print('Dano: $danoTotal (base: $danoBase'
        '${armaEquipada != null ? ' + ${armaEquipada!.dano} arma' :
        ↪  ''}'
        ')');
    print('Defesa: $defesaTotal (base: 2' +
          (armaduraEquipada != null
              ? ' + ${armaduraEquipada!.defesa} armadura'
              : '') +
          ')');
    print('Ouro: $ouro');
  }
}
```

Propriedade `get`: `int get danoTotal { ... }` é uma propriedade calculada. Você acessa com `jogador.danoTotal`, não `jogador.danoTotal()`. Dart calcula o valor no momento, sempre atualizado. É como um getter sem parênteses.

Parte 6: Economia. Ouro, Compra e Venda

O jogador tem `int ouro` que funciona como a moeda do jogo. Um sistema de economia é essencial em masmorra: faz o jogador escolher (comprar a espada cara agora ou economizar?), cria sacrifício (vender itens para comprar melhores). Note que ao vender você recebe 50% do preço: isso é economia de jogo típica que desestimula spam de compra/venda e mantém o ouro valioso.

```
// lib/jogador.dart

class Jogador {
  int ouro;

  bool tentarComprar(Item item) {
    if (ouro < item.preco) {
      print('Dinheiro insuficiente! '
          'Custa ${item.preco}, você tem $ouro.');
      return false;
    }

    ouro -= item.preco;
    inventario.add(item);
    print('Comprou ${item.nome} por ${item.preco} ouro!');
    return true;
  }

  bool tentarVender(int indiceNoInventario) {
    if (indiceNoInventario < 0 ||
        indiceNoInventario >= inventario.length) {
      print('Índice inválido!');
      return false;
    }

    final item = inventario[indiceNoInventario];

    if (armaEquipada == item || armaduraEquipada == item) {
```

```
      print('Não pode vender algo equipado! Desequipe antes.');
      return false;
    }

    int precoVenda = (item.preco * 0.5).toInt();
    ouro += precoVenda;
    inventario.removeAt(indiceNoInventario);
    print('Vendeu ${item.nome} por $precoVenda ouro.');
    return true;
  }
}
```

Nota: ao vender, você recebe 50% do valor (calculado com `(item.preco * 0.5).toInt()`). Isso é economia de jogo típica, desincentiva spam de compra/venda.

Parte 7: Loot Tables. Drop de Items

Quando um inimigo morre, às vezes deixa itens. Vamos criar uma tabela simples usando um `Map<String, List<Item>>` que mapeia cada tipo de inimigo para os itens que pode droppar. Isso é comum em RPGs: um zumbi droppa moedas sujas e armas fracas, enquanto um esqueleto droppa tesouro mais valioso de guerreiro antigo. Usamos `Random` para escolher aleatoriamente qual item da lista ele deixa.

```
// lib/loot_table.dart

import 'dart:math';
import 'item.dart';
import 'arma.dart';

final Map<String, List<Item>> lootTablePorInimigo = {
  'zumbi': [
    Item(
      id: 'moedas-sujas',
```

```
      nome: 'Moedas Sujas',
      descricao: 'Ouro roubado que o zumbi carregava',
      preco: 5,
      peso: 0,
    ),
    Arma(
      id: 'cutelo-enferrujado',
      nome: 'Cutelo Enferrujado',
      descricao: 'Uma arma pobre, mas cortante',
      preco: 40,
      peso: 2,
      dano: 4,
      tipo: 'cortante',
    ),
  ],
  'esqueleto': [
    Arma(
      id: 'sabre-ossudo',
      nome: 'Sabre do Túmulo',
      descricao: 'Arma de um cavaleiro há séculos falecido',
      preco: 120,
      peso: 3,
      dano: 9,
      tipo: 'cortante',
    ),
    Item(
      id: 'anel-prata',
      nome: 'Anel de Prata',
      descricao: 'Um adorno antigo, de valor incerto',
      preco: 80,
      peso: 0,
    ),
  ],
};

Item? obterLootAleatorio(String nomeDoInimigo) {
```

```
  final loot = lootTablePorInimigo[nomeDoInimigo];
  if (loot == null || loot.isEmpty) {
    return null;
  }

  final random = Random();
  return loot[random.nextInt(loot.length)];
}
```

Importar `Random` do pacote `dart:math`:

```
import 'dart:math';
```

Parte 8: Constantes. Items Predefinidos

É conveniente ter itens já prontos como constantes globais. Assim, toda vez que você quer dar uma espada ao jogador (no início, na loja, como loot), você reutiliza a mesma definição. Evita duplicação e torna fácil balancear (mudar o dano em um só lugar).

```
// lib/items_base.dart

final espadaDeBronze = Arma(
  id: 'espada-bronze',
  nome: 'Espada de Bronze',
  descricao: 'Uma arma comum, de metal maleável',
  preco: 200,
  peso: 3,
  dano: 8,
  tipo: 'cortante',
);

final pocaoDeVida = Item(
  id: 'pocao-vida',
```

```
  nome: 'Poção de Vida',
  descricao: 'Recupera 20 HP',
  preco: 50,
  peso: 1,
);

final camisaDeCouro = Armadura(
  id: 'camisa-couro',
  nome: 'Camisa de Couro',
  descricao: 'Proteção básica, elegante e prática',
  preco: 100,
  peso: 2,
  defesa: 3,
  localizacao: 'peito',
);

final lojaPrincipal = [
  espadaDeBronze,
  pocaoDeVida,
  camisaDeCouro,
];
```

Parte 9: Exemplo Completo. Uma Sessão de Jogo

Vamos ver tudo funcionando junto: um jogador compra itens, equipa armas, vê seu dano aumentar, vende itens para financiar novas compras. Este exemplo mostra o sistema de economia e equipamento em ação, desde a criação do jogador até a manipulação do inventário.

```
void main() {
  final jogador = Jogador(
    nome: 'Aldric',
    maxHp: 100,
    ouro: 500,
  );
```

```
print('=== SESSÃO DE JOGO ===\n');

jogador.mostraStatus();

print('\n--- Entrando na Loja ---');
print('Espada de Bronze custa 200 ouro.');
jogador.tentarComprar(espadaDeBronze);
jogador.mostraStatus();

print('\n--- Equipando ---');
jogador.equiparArma(0);
jogador.mostraStatus();

print('\n--- Compra 2 ---');
jogador.tentarComprar(camisaDeCouro);
jogador.equiparArmadura(0);
jogador.mostraStatus();

print('\n--- Compra 3 (vai falhar) ---');
jogador.tentarComprar(Arma(
  id: 'espada-draco',
  nome: 'Espada do Dragão',
  descricao: 'Lendária',
  preco: 2000,
  peso: 5,
  dano: 25,
  tipo: 'cortante',
));

jogador.mostraStatus();

if (jogador.inventario.isNotEmpty) {
  print('\n--- Vendendo ---');
  jogador.tentarVender(0);
}
```

```
    jogador.mostraStatus();
}
```

Desafios da Masmorra

Desafio 13.1. Item Consumível (Poções). Crie uma classe `Consumivel extends Item` com um atributo `efeito` (string descrevendo o efeito, ex: "Cura 30 HP") e `hpRecuperado: int`. Implemente na classe `Jogador` um método `usarConsumivel(int indice)` que remove o item do inventário, aplica o efeito (chama `curar(hpRecuperado)`), e mostra a mensagem de efeito.

Desafio 13.2. Limite de peso realista. Adicione um getter `pesoTotalInventario` ao `Jogador` que calcula o peso total. Implemente um limite de 5000 gramas total. O método `tentarEquipar(Arma a)` deve verificar se a mochila não ficará muito pesada. Se exceder, mostre: "Sua mochila está muito pesada! Largue algo antes de equipar."

Desafio 13.3. Comparador de itens. Crie um método `String compararItens(int indice1, int indice2)` na classe `Jogador` que recebe dois índices e retorna uma string comparando: qual tem mais dano/defesa/efeito? Útil para o jogador decidir qual equipar.

Desafio 13.4. Venda em massa. Implemente um método `int venderTodosDoTipo<T extends Item>()` que vende todos os itens de tipo T (exceto equipados) e retorna o ouro total obtido. Por exemplo, `venderTodosDoTipo<Consumivel>()` vende todas as poções de uma vez.

Boss Final 13.5. Sistema de Loja. Crie uma classe `Loja` com um `String nome`, um `List<Item> estoque`, e um `double taxaMarcup` (ex: 1.2 para 20% mais caro). Implemente `bool venderAoJogador(Jogador j, int indiceEstoque)` que verifica ouro, cobra com markup, e adiciona ao inventário. Implemente `bool comprarDoJogador(Jogador j, int indiceInventario)` que compra a 50% do preço. Demonstre uma loja funcional.

Pergaminho do Capítulo

Neste capítulo você aprendeu:

- Herança (`extends`): classes filhas (`Arma`, `Armadura`) herdam de `Item`, reutilizando código e criando uma hierarquia lógica.

- Inventário: uma simples `List<Item>` que mantém ordem, crucial para indexação. Suporta qualquer subclasse de `Item`.
- Equipamento: slots (`armaEquipada`, `armaduraEquipada`) que guardam o que o jogador está usando.
- Estatísticas calculadas: `danoTotal` e `defesaTotal` usando `get` que combinam base + bônus de equipamento.
- Economia: ouro sobe/desce com compra e venda, com validações para evitar negativo.
- Loot tables: mapeamentos `String → List<Item>` que simulam drops realistas.

No próximo capítulo, vamos usar todo esse sistema num combate real, onde o dano que você calcula aqui vai fazer diferença.

Dica do Mestre: `sealed class` (Dart 3+) são incríveis para limitar a hierarquia. Use `sealed class Item` e `final class Arma extends Item` para garantir que apenas `Arma`, `Armadura` e `Consumivel` podem estender `Item`. Com `sealed`, o compilador avisa se você esquecer um caso num `switch`. Isso previne bugs no futuro, quando adicionar novos tipos.

Próximo Capítulo

No próximo capítulo, o sangue corre. O sistema de combate por turnos ganha vida com ataques, defesas, críticos e a tensão de cada decisão.

Capítulo 14 - Combate por turnos

Você enfrenta o inimigo. Não foge, não negocia. Apenas luta. Seu turno: ataca. Seu turno: defende. Seu turno: falha. A morte espreita cada decisão. Este é o coração que faz um roguelike bater.

O que vamos aprender

Neste capítulo você vai: - Criar uma `class Combate` que orquestra lutas entre `Jogador` e `Inimigo` - Implementar um `loop` de turnos com escolhas do jogador - Usar `Random` de `dart:math` para dano variável (realismo) - Registrar tudo numa `List<String> log` (história da luta) - Mostrar status visual com barras de HP em ASCII - Recompensar vitórias com XP e ouro - Tratar derrota e morte

Ao final, você terá um sistema completo de combate que é o pico emocional desta parte.

O peso da morte permanente

Antes de codificar a primeira linha, pare um segundo. O que estamos prestes a construir é o que distingue um *roguelike* de todos os outros jogos: o **permadeath**. Quando o jogador cai em combate, não há respawn, não há continue, não há "carregar o último save dez segundos antes da morte". A ficha é descartada. Todo o ouro acumulado, cada item raro, cada ponto de XP suado: tudo se vai. Para começar de novo, é literalmente começar de novo.

Isso pode soar cruel, e é. Mas é essa crueldade que dá sentido a cada decisão. Quando cada ataque pode ser o último, o jogador pesa cada turno. Fugir deixa de ser covardia: vira estratégia. Poções param de ser "itens acumulados no inventário" e viram salva-vidas. A tensão do *roguelike* vive exatamente nesse compromisso: ou você joga com atenção plena, ou reinicia.

No nosso código, o permadeath vai acontecer numa única linha: quando `jogador.hp <= 0`, o loop de combate termina com `_exibirGameOver()` e nenhum save é escrito. O estado do jogador é simplesmente esquecido. Não é nada

sofisticado tecnicamente, mas é o contrato mais importante que o jogo faz com quem o joga. Tenha isso em mente enquanto implementa este capítulo: cada linha de código do combate carrega o peso dessa regra.

Parte 1: Conceitualizando a luta

Um **combate por turnos** em *roguelike* tem estrutura:

1. Inicialização: jogador encontra inimigo, entrada no "modo combate"
2. Loop de turnos:
    - Jogador escolhe ação (atacar, defender, fugir, usar item)
    - Resolver ação
    - Inimigo reage (se ainda vivo)
    - Checar condição de vitória/derrota
3. Resolução: prêmios ou game over

Vamos modelar isso em código.

Aleatoriedade e RPGs: por que os dados nunca mentem

Você já notou que em um RPG real (D&D, Pathfinder, qualquer jogo de mesa), cada ação é incerta? Você rola um dado de 20 lados para saber se acerta um golpe. Rola novamente para determinar dano. Essa incerteza é essencial: sem ela, combate é determinístico, previsível, entediante. Se você sempre ataca com 8 de dano, o inimigo sempre ataca com 6, o resultado final é óbvio desde o início. Ninguém quer jogar um RPG onde sabe exatamente quem vai ganhar antes do combate começar.

Daí entra `Random` do Dart. Em vez de dano fixo, você rola dados: `jogador.dano - 20% + aleatório(40%)`. Isso gera um intervalo realista. Um ataque pode fazer 6 a 10 de dano em vez de sempre 8. Combates viram emocionantes. Uma derrota inesperada é possível. Uma vitória incrível contra um inimigo mais forte se torna história.

Neste capítulo, `Random` é seu aliado para criar combate que respira, que surpreende, que faz a adrenalina bombar.

Parte 2: Classe Combate. Orchestrador

A classe `Combate` é o coração do sistema. Ela recebe um `Jogador` e um `Inimigo` e orquestra todo o loop de turnos. Mantém um `log` de mensagens (crucial para entender o que aconteceu), calcula dano com variação aleatória (para não

ser previsível) e gerencia defesa, item e fuga. Note o método `_registrar()` que tanto escreve na tela quanto armazena no log para replay.

```
// lib/combate.dart

import 'dart:io';
import 'dart:math';
import 'jogador.dart';
import 'inimigo.dart';

class Combate {
  final Jogador jogador;
  final Inimigo inimigo;
  final List<String> log = [];
  final Random random = Random(); // Reutilize uma única instância

  int turno = 0;
  bool defesaAtiva = false;

  Combate({
    required this.jogador,
    required this.inimigo,
  });

  void _registrar(String mensagem) {
    log.add(mensagem);
    print(mensagem);
  }

  void mostrarStatus() {
    final barraJogador = _construirBarra(jogador.hp, jogador.maxHp);
    final barraInimigo = _construirBarra(inimigo.hp, inimigo.maxHp);

    print('');
    print('⚔ COMBATE ⚔');
    print('${jogador.nome} vs ${inimigo.nome}');
```

```
  print('$barraJogador $barraInimigo');
  final hpJ = '${jogador.hp}/${jogador.maxHp}';
  final hpI = '${inimigo.hp}/${inimigo.maxHp}';
  print('HP: $hpJ  HP: $hpI');
  print('Atq: ${jogador.danoTotal}  Atq: ${inimigo.danoBase}');
  print('');
}

String _construirBarra(int hp, int maxHp) {
  const totalBlocos = 10;
  final blocos = ((hp / maxHp) * totalBlocos).toInt();
  final cheios = '▓' * blocos;
  final vazios = '░' * (totalBlocos - blocos);
  return '$cheios$vazios';
}

bool atacar() {
  final variacao = (jogador.danoTotal * 0.2).toInt();
  final dano = jogador.danoTotal - variacao +
      random.nextInt(variacao * 2);

  _registrar('> ${jogador.nome} ataca com força! Dano: $dano');

  if (inimigo.sofrerDano(dano)) {
    _registrar(' ${inimigo.nome} foi derrotado!');
    return true;
  }

  defesaAtiva = false;
  return false;
}

bool defender() {
  defesaAtiva = true;
  _registrar('> ${jogador.nome} assume posição defensiva!');
  return false;
```

```
  }

  bool fugir() {
    if (random.nextDouble() < 0.4) {
      _registrar(' ${jogador.nome} conseguiu fugir!');
      return true;
    } else {
      _registrar(' ${jogador.nome} não conseguiu escapar!');
      return false;
    }
  }

  bool usarItem(int indiceNoInventario) {
    if (indiceNoInventario < 0 ||
        indiceNoInventario >= jogador.inventario.length) {
      _registrar('Item não encontrado!');
      return false;
    }

    final item = jogador.inventario[indiceNoInventario];

    if (item.id == 'pocao-vida') {
      const cura = 20;
      final vidaAnterior = jogador.hp;
      jogador.hp = (jogador.hp + cura).clamp(0, jogador.maxHp);
      final curaReal = jogador.hp - vidaAnterior;

      _registrar('> ${jogador.nome} bebe uma poção '
          'e recupera $curaReal HP!');
      jogador.inventario.removeAt(indiceNoInventario);
      return false;
    }

    _registrar('Você não pode usar isso em combate!');
    return false;
  }
```

```
  void turnoDoInimigo() {
    if (inimigo.hp < inimigo.maxHp / 3 && random.nextDouble() < 0.3) {
      inimigo.executarHabilidadeEspecial(this);
    } else {
      final dano = inimigo.calcularDano();

      int danoFinal = dano;
      if (defesaAtiva) {
        danoFinal = (dano * 0.6).toInt();
        _registrar('> ${inimigo.nome} ataca, mas '
            'a defesa reduz o impacto!');
      } else {
        _registrar('> ${inimigo.nome} contra-ataca! Dano:
↪  $danoFinal');
      }

      if (jogador.sofrerDano(danoFinal)) {
        _registrar(' ${jogador.nome} foi derrotado...');
      }
    }

    defesaAtiva = false;
  }

  void executar() {
    turno = 0;
    mostrarStatus();

    while (jogador.hp > 0 && inimigo.hp > 0) {
      turno++;
      print('\n--- TURNO $turno ---');

      print('\nOpções:');
      print('1 - Atacar');
      print('2 - Defender');
```

```
        print('3 - Fugir');
        print('4 - Usar item');
        print('5 - Sair (não implementado)');
        stdout.write('\nEscolha: ');

        final escolha = stdin.readLineSync() ?? '1';

        bool combateAcabou = false;

        switch (escolha.trim()) {
          case '1':
            combateAcabou = atacar();
            break;
          case '2':
            defender();
            break;
          case '3':
            combateAcabou = fugir();
            if (combateAcabou) {
              _registrar('Você fugiu do combate.');
              return;
            }
            break;
          case '4':
            stdout.write('Qual item? '
                '(0-${jogador.inventario.length - 1}): ');
            final indiceStr = stdin.readLineSync() ?? '0';
            usarItem(int.tryParse(indiceStr) ?? 0);
            break;
          default:
            _registrar('Ação desconhecida!');
            continue;
        }

        if (combateAcabou && inimigo.hp <= 0) {
          break;
```

```
    }

    if (inimigo.hp > 0) {
      turnoDoInimigo();
    }

    if (jogador.hp <= 0) {
      _registrar('\n[DERROTA] Você caiu em combate.');
      _exibirGameOver();
      return;
    }

    mostrarStatus();
  }

  if (inimigo.hp <= 0) {
    _exibirVitoria();
  }
}

void _exibirVitoria() {
  _registrar('\n[VITÓRIA] Você venceu o combate!');
  final ouroGanho = inimigo.calcularOuroDrop();
  final xpGanho = inimigo.calcularXPDrop();

  jogador.ouro += ouroGanho;
  // O campo `xp` será introduzido no Capítulo 25 — Progressão.
  // Por enquanto, declare `int xp = 0;` na classe Jogador.
  jogador.xp += xpGanho;

  _registrar('Você ganhou $ouroGanho ouro e $xpGanho XP!');

  if (random.nextDouble() < 0.3) {
    final item = inimigo.gerarLoot();
    if (item != null) {
      jogador.inventario.add(item);
```

```
        _registrar('Você encontrou: ${item.nome}!');
      }
    }

    print('');
    print('[VITÓRIA] Você venceu o combate!');
    print('Ouro: +$ouroGanho');
    print('XP: +$xpGanho');
    print('');
  }

  void _exibirGameOver() {
    print('');
    print('==========================');
    print('       [GAME OVER]        ');
    print('==========================');
    print('Você caiu em combate após $turno turnos.');
    print('Sua jornada termina aqui.');
    print('Todos os itens, todo o XP, todo o ouro:');
    print('tudo se vai com você.');
    print('');
    print('Comece outra vez. A masmorra te espera.');
    print('');
  }

  void mostrarLog() {
    print('\n=== LOG DE COMBATE ===');
    for (final mensagem in log) {
      print(mensagem);
    }
  }
}
```

Notas importantes:

- `stdin.readLineSync()` lê entrada do teclado. Você vai precisar de `import 'dart:io';`
- `_registrar()` escreve e armazena no `log`

- `defesaAtiva` é um flag booleano que dura um turno
- Dano tem variação (usando `random.nextInt()`) para não ser previsível
- Vitória e derrota têm tratamento especial em `_exibirVitoria()` e `_exibirGameOver()`

Parte 3: Classe Inimigo e Subtipos

Agora você precisa de inimigos que funcionem com combate. Mas aqui surge um problema clássico: seu *roguelike* tem Zumbi, Lobo e Orc. Cada um é diferente em nome, HP, dano e habilidades. Se você criasse cada um do zero como uma classe separada, teria muita duplicação: `class Zumbi { int hpMax; int hpAtual; int dano; ... sofrerDano() { ... } }` e `class Lobo { int hpMax; int hpAtual; int dano; ... sofrerDano() { ... } }`. O código `sofrerDano()` é idêntico em ambos. Você estaria escrevendo a mesma coisa várias vezes.

Aí entra a classe abstrata. Você cria uma `abstract class Inimigo` que define a estrutura e o comportamento comum a todos os inimigos: HP, dano, método `sofrerDano()`, método para calcular dano aleatório. Depois, cada inimigo (Zumbi, Lobo, Orc) herda desse template e personaliza apenas o que é único: seu loot, suas habilidades especiais, seus valores base. Zero duplicação.

A `abstract class Inimigo` define o contrato: todo inimigo tem HP, dano, e pode sofrer dano. Mas cada subtipo (Zumbi, Lobo, Orc) personaliza seu loot e habilidades especiais. Observe `sofrerDano()` que retorna `bool`: true se o inimigo morreu, false se ainda está vivo. Isso simplifica o loop de combate.

```
// lib/inimigo.dart

import 'dart:math';
import 'item.dart';
import 'combate.dart';

abstract class Inimigo {
  // Reutilize uma única instância entre todos os inimigos
  static final Random _random = Random();

  final String id;
  final String nome;
```

```
  int maxHp;
  int hp;
  final int danoBase;

  Inimigo({
    required this.id,
    required this.nome,
    required this.maxHp,
    required this.danoBase,
  }) : hp = maxHp;

  bool sofrerDano(int dano) {
    hp -= dano;
    return hp <= 0;
  }

  int calcularDano() {
    final variacao = (danoBase * 0.15).toInt();
    return danoBase - variacao + _random.nextInt(variacao * 2);
  }

  int calcularOuroDrop() {
    return 10 + _random.nextInt(10);
  }

  int calcularXPDrop() {
    return 50;
  }

  Item? gerarLoot() {
    return null;
  }

  void executarHabilidadeEspecial(Combate combate) {
    combate._registrar('> ${nome} não tem habilidade especial!');
  }
```

```
}

class Zumbi extends Inimigo {
  Zumbi()
      : super(
          id: 'zumbi',
          nome: 'Zumbi Pilhador',
          maxHp: 30,
          danoBase: 6,
        );

  @override
  Item? gerarLoot() {
    if (_random.nextDouble() < 0.5) {
      return Item(
        id: 'moedas-sujas',
        nome: 'Moedas Sujas',
        descricao: 'Roubo do zumbi',
        preco: 15,
        peso: 0,
      );
    }
    return null;
  }
}

class Lobo extends Inimigo {
  Lobo()
      : super(
          id: 'lobo',
          nome: 'Lobo Selvagem',
          maxHp: 50,
          danoBase: 8,
        );

  @override
```

```
  void executarHabilidadeEspecial(Combate combate) {
    combate._registrar('> ${nome} salta e rosna!');
    hp = (hp + 15).clamp(0, maxHp);
  }

  @override
  Item? gerarLoot() {
    if (_random.nextDouble() < 0.6) {
      return Arma(
        id: 'fanga-lobo',
        nome: 'Fanga do Lobo',
        descricao: 'Arma antiga',
        preco: 100,
        peso: 3,
        dano: 7,
        tipo: 'cortante',
      );
    }
    return null;
  }
}

class Orc extends Inimigo {
  Orc()
      : super(
          id: 'orc',
          nome: 'Orc Guerreiro',
          maxHp: 70,
          danoBase: 12,
        );

  @override
  void executarHabilidadeEspecial(Combate combate) {
    combate._registrar('> ${nome} desfere um golpe furioso!');
  }
}
```

Nota: cada inimigo herda estrutura (via `extends Inimigo`), mas personaliza HP, dano, loot e habilidades (via `@override`).

Parte 4: Integrando Combate no Jogador

O `Jogador` precisa de métodos para combate. O método `sofrerDano()` decresce HP e retorna `true` se o jogador morreu (útil para saber se deve rodar game over). Já `enfrentarInimigo()` é o ponto de entrada: cria uma instância de `Combate`, executa o loop de turnos, e depois exibe o log completo para o jogador revisar. Isso conecta o sistema de combate com a classe Jogador.

```
// jogador.dart (adições)

class Jogador {
  int xp = 0;

  bool sofrerDano(int dano) {
    hp -= dano;
    if (hp < 0) hp = 0;
    return hp <= 0;
  }

  void enfrentarInimigo(Inimigo inimigo) {
    print('\n[COMBATE] Você encontrou um ${inimigo.nome}!');
    final combate = Combate(jogador: this, inimigo: inimigo);
    combate.executar();

    combate.mostrarLog();
  }
}
```

Parte 5: Factory de Inimigos

Você agora tem `Zumbi()`, `Lobo()`, `Orc()` prontos para criar instâncias. Mas imagine uma masmorra grande com 20 tipos de inimigos diferentes. Ao gerar uma sala, você faria `if (ambiente == 'floresta') { inimigo = Lobo(); } else if (ambiente == 'catacumba') { inimigo = Zumbi(); } ...`. Espalhado

pelo código. Se precisar adicionar um novo inimigo, tem que caçar todos os lugares onde inimigos são criados e adicionar novo `if`.

Aí entra o padrão Factory. Você centraliza toda a lógica de criação de inimigos num único lugar. Em vez de escrever `Zumbi()` espalhado pelo código, você chama `FabricaInimigo.criarPorId('zumbi')`. Se precisar trocar a lógica de criação, muda num só lugar. Se vai adicionar um novo inimigo, registra na Factory e pronto. O resto do código continua funcionando sem saber quantos tipos existem.

Para gerar inimigos pelo ID, use o padrão Factory (uma `class` com métodos estáticos). Você não cria `Zumbi()` diretamente, mas chama `FabricaInimigo.criarPorId('zumbi')`. Note a função `gerarInimigo()` que escolhe aleatoriamente qual tipo de inimigo aparece num certo ambiente (floresta vs catacumba).

```
// lib/fabrica_inimigo.dart

import 'dart:math';
import 'inimigo.dart';
import 'zumbi.dart';
import 'lobo.dart';
import 'orc.dart';

class FabricaInimigo {
  static Inimigo criarPorId(String id) {
    switch (id) {
      case 'zumbi':
        return Zumbi();
      case 'lobo':
        return Lobo();
      case 'orc':
        return Orc();
      default:
        throw Exception('Inimigo desconhecido: $id');
    }
  }

  static final Map<String, List<String>> inimigosAmbiente = {
```

```
    'floresta': ['zumbi', 'lobo'],
    'catacumba': ['lobo', 'orc'],
    'caverna': ['zumbi', 'orc', 'lobo'],
  };

  static String gerarInimigo(String ambiente) {
    final opcoes = inimigosAmbiente[ambiente] ?? ['zumbi'];
    return opcoes[Random().nextInt(opcoes.length)];
  }
}
```

Parte 6: Exemplo Completo. Uma Luta Real

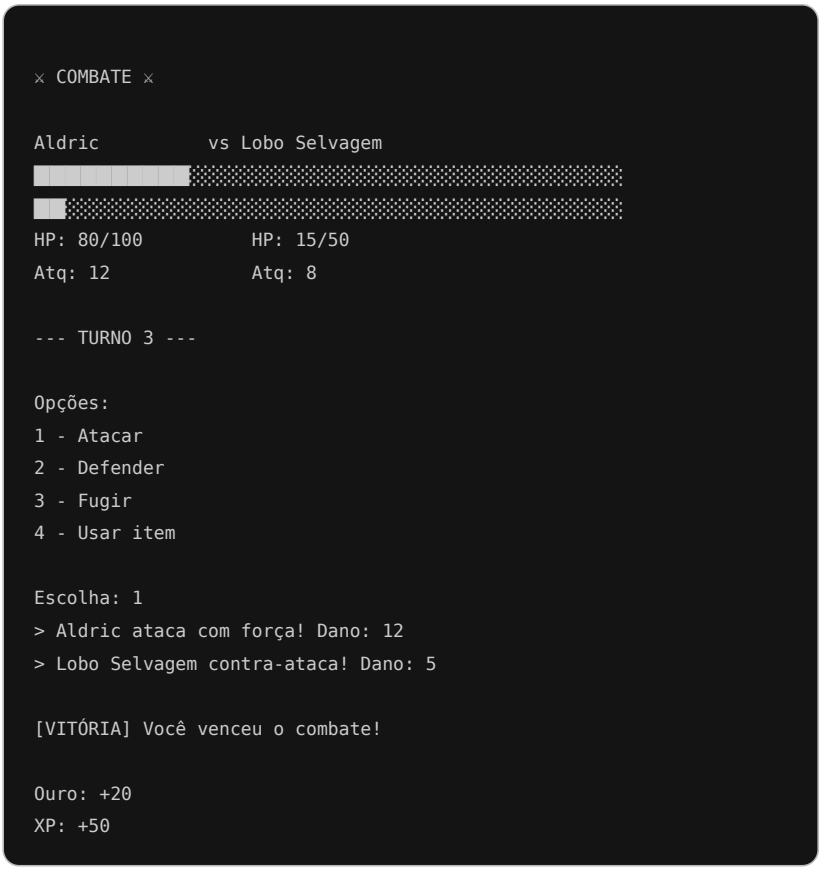
Aqui está um exemplo de fim a fim: criamos um jogador com uma espada e poção, depois ele enfrenta um lobo. O combate roda com input do usuário até que o jogador vença, fuja ou morra. Este é o momento em que todo o sistema de combate (turnos, dano, itens, recompensas) se une numa experiência coerente.

```
// lib/main.dart

import 'dart:io';
import 'jogador.dart';
import 'arma.dart';
import 'item.dart';
import 'fabrica_inimigo.dart';

void main() {
  final jogador = Jogador(
    nome: 'Aldric',
    maxHp: 100,
    ouro: 100,
  );

  final espada = Arma(
```

```
    id: 'espada-bronze',
    nome: 'Espada de Bronze',
    descricao: 'Uma arma comum',
    preco: 200,
    peso: 3,
    dano: 8,
    tipo: 'cortante',
  );

  jogador.inventario.add(espada);
  jogador.equiparArma(0);

  final pocao = Item(
    id: 'pocao-vida',
    nome: 'Poção de Vida',
    descricao: 'Recupera 20 HP',
    preco: 50,
    peso: 1,
  );
  jogador.inventario.add(pocao);

  print('=== AVENTURA COMEÇA ===\n');
  jogador.mostraStatus();

  final inimigo = FabricaInimigo.criarPorId('lobo');
  jogador.enfrentarInimigo(inimigo);

  jogador.mostraStatus();
}
```

O Jogo Até Aqui

Ao final desta parte, seu combate no terminal se parece com isto:



Cada parte adiciona novas camadas ao jogo. Compare com o início e veja o quanto você evoluiu nesta jornada!

Código Completo no Step

O diretório `code/steps/step-14/` contém a implementação completa e compilável deste capítulo. Note que o método `enfrentarInimigo()` no passo 14 está integrado na classe `Jogador`, e a orquestração final do combate (chamadas em sequência de encontros, game loop principal) está em `main.dart`. O capítulo foca nos conceitos de combate por turnos; o step completo mostra como integrar essa lógica na aventura maior, separando responsabilidades entre `Combate`, `Jogador` e o ponto de entrada da aplicação.

Desafios da Masmorra

Desafio 14.1. HUD em Combate com cores ANSI. Crie um método `mostrarStatusCombate()` que exibe HP em percentual e com código de cor: verde se acima de 75%, amarelo entre 50-75%, vermelho abaixo de 50%. Use escape codes ANSI: `'\u001B[32m'` verde, `'\u001B[33m'` amarelo, `'\u001B[31m'` vermelho, `'\u001B[0m'` reset.

Desafio 14.2. Ataque Crítico. Implemente crítico: 15% de chance de dano dobrado (x2). Use `Random().nextDouble() < 0.15`. Quando crítico ocorrer, registre no log: "GOLPE CRÍTICO! Dano dobrado!" e mostre o dano com destaque.

Desafio 14.3. Limite de turno (Fuga automática). Adicione um limite: combate não pode durar mais de 10 turnos. Se chegar ao limite e ainda houver combate, o jogador é forçado a fugir automaticamente com mensagem: "A luta durou demais, você foge pela sua vida!"

Desafio 14.4. Ação Defensa com Riposte. Implemente uma ação `defender()`: reduz dano sofrido em 50% neste turno. Além disso, ao sofrer ataque enquanto defendendo, há 30% de chance de ripostear (contra-ataque) com 30% do seu dano normal.

Desafio 14.5. Combate em Grupo (Avançado). Implemente combate contra múltiplos inimigos. Crie uma classe `CombateGrupo` que recebe `List<Inimigo> inimigos` e o jogador enfrenta todos sequencialmente, mas numa ordem que você escolhe (IA básica: mais fraco primeiro). Registre cada transição entre inimigos no log.

Boss Final 14.6. Poções dinâmicas (Integração com inventário). Crie uma classe `Pocao extends Item` com um campo `int curaHP` e um método `usar(Jogador j)` que chama `j.curar(curaHP)`. Refatore `usarItem()` no combate para checar o tipo de item: se for `Pocao`, chama `pocao.usar(jogador)`. Crie três tipos: PocaoPequena (10 HP), PocaoMedia (25 HP), PocaoGrande (50 HP). Demonstre no combate.

Pergaminho do Capítulo

Neste capítulo você aprendeu:

- `class Combate`: orchestrador que gerencia turnos, ações, `log` e resolução
- `loop` de turnos: escolha > ação > reação > checar fim
- Ações variadas: `atacar()`, `defender()`, `fugir()`, `usarItem()` (cada uma com lógica)

- Dano variável: `Random` para realismo (±20%)
- IA simples: cada inimigo reage diferente via `turnoDoInimigo()`
- Recompensas: ouro, XP, itens baseado em derrota
- ASCII visual: barras de HP, `log`, estrutura clara com `@override`

Seu jogo agora é um verdadeiro *roguelike* com combate completo. Isso é o pico emocional desta parte.

No próximo capítulo começaremos a expandir a exploração da masmorra, com salas, movimento 2D e encontros dinâmicos.

Dica do Mestre: Sempre registre ações em combate (no `log`). Ajuda a entender o que aconteceu e é essencial para balanceamento. Além disso, considere criar diferentes níveis de dificuldade criando variantes dos inimigos. Por exemplo, `class GoblinForte extends Zumbi { ... }`. Teste bastante! Combate é onde o balanceamento importa.

PARTE III

A MASMORRA DESPERTA

As paredes surgem do algoritmo. A névoa de guerra esconde o que morde nos recessos não explorados. Cada partida é um mapa novo, gerado por regras determinísticas que parecem caos. Aqui, aprenderá a domesticar a aleatoriedade, a dominar o grid, e a fazer a máquina criar mundos.

Capítulo 15 - Da Sala ao Tile: Pensando em 2D

A masmorra ganha duas dimensões. Até agora, salas eram nomes em um mapa de texto. Agora são tiles em um grid, paredes são `#`, chão é `.`, e o jogador é um `@` que se move com WASD. A névoa de guerra esconde o que você ainda não explorou, e tochas iluminam apenas o que está ao alcance. O mapa gera-se sozinho a cada partida, diferente toda vez, porque você escreveu o algoritmo que o cria.

Nesta parte, a masmorra finalmente parece uma masmorra. Inimigos patrulham corredores, itens brilham no chão, e escadas levam para andares mais profundos. O terminal vira uma janela para um mundo procedural que responde ao seu código. Quando você compilar e vir o mapa aparecer pela primeira vez, vai entender por que roguelikes são viciantes.

> *Você explorou um único aposento como texto puro. Mas um verdadeiro aventureiro não caminha palavra por palavra. Caminha tile por tile. E todo mundo, por maior que seja, é feito de pequenos quadrados alinhados numa grade. A grade 2D é como o mapa de Zelda visto de cima: cada posição tem um tile que diz se há parede, chão passável ou algo especial.*

O Que Vamos Aprender

Neste capítulo você vai deixar para trás o modelo de salas separadas (texto puro, grafo de conexões) e abraçar o paradigma roguelike clássico: um mapa 2D baseado em **tiles** (quadrados numa grade).

Especificamente: - Entender por que *roguelikes* usam grades: colisão em tempo real, movimento gradual, visão de distância - Criar uma estrutura de dados 2D eficiente em Dart: `List<List<Tile>>` com **collection for** e **collection if** - Usar **typedef** para melhorar legibilidade: `typedef Grade = List<List<Tile>>` - Definir um enum `Tile` com tipos: `parede`, `chao`, `porta`, `escadaDesce` - Construir a classe `MapaMasmorra` que encapsula o mapa e fornece métodos seguros - Renderizar a grade no terminal com loops aninhados - Implementar movimento do jogador com WASD: atualizar posição, verificar colisões - Aplicar boundary checks para não sair da tela - Mostrar um exemplo completo funcionando: masmorra 10x10, jogador move-se com feedback

Ao final, você terá o alicerce de toda exploração *roguelike*. Sem um grid não há mapa. Sem mapa não há jogo.

Parte 1: Do Grafo ao Grid — Mudança Conceitual

Por Que Sair das Salas?

Nos capítulos anteriores você tinha um grafo de salas: cada sala era um nó, conexões eram arestas. Isso funciona para aventuras em prosa, mas *roguelikes* precisam de geometria real.

Considere: - Visibilidade (*FOV*): um inimigo pode ver o jogador? Precisa distância e linha de visão - Movimento: um jogador não pula de sala a sala. Caminha tile por tile - Pathfinding: como um inimigo caminha até o jogador? Precisa de coordenadas (x, y) - Colisões: paredes não são abstratas. Ocupam posições específicas - Geração procedural: criar uma masmorra aleatória é mais fácil em grade (pense em algoritmos como random walk)

Uma grade 2D é a linguagem natural de *roguelikes*.

Conceitos Fundamentais

Antes de código, entenda a geometria:

```
     x=0 x=1 x=2 ... x=9
   ┌──────────────────────┐
y=0│ (0,0)(1,0)(2,0)...(9,0)
y=1│ (0,1)(1,1)(2,1)...(9,1)
y=2│ (0,2)(1,2)(2,2)...(9,2)
... │
y=9│ (0,9)(1,9)(2,9)...(9,9)
   └──────────────────────┘
```

Notação (x, y): - x = coluna (horizontal, esquerda para direita) - y = linha (vertical, topo para fundo) - Origem (0, 0) é o canto superior esquerdo

Para acessar a célula em (2, 3) na grade:

```
final tile = grade[3][2]; // grade[y][x] ... cuidado com a ordem!
```

Sempre `grid[y][x]`, nunca `grid[x][y]`. Essa é a convenção porque iteramos linhas (y) primeiro, colunas (x) segundo.

Parte 2: Definindo Tiles. Enum e Typedef

Comece definindo que tipo de tile existe:

```
// tile.dart

enum Tile {
  parede,      // '#' - parede sólida, intransponível
  chao,        // '.' - chão passável
  porta,       // '+' - porta fechada ou aberta
  escadaDesce, // '>' - escadas para próximo nível
}

String tileParaChar(Tile tile) {
  return switch (tile) {
    Tile.parede => '#',
    Tile.chao => '.',
    Tile.porta => '+',
    Tile.escadaDesce => '>',
  };
}

bool ehPassavelTile(Tile tile) {
  return tile == Tile.chao ||
      tile == Tile.porta ||
      tile == Tile.escadaDesce;
}
```

Agora, typedef para clareza:

```
// mapa_masmorra.dart
```

```
typedef Grade = List<List<Tile>>;
typedef Posicao = ({int x, int y});
```

Por que typedef? Seu código fica mais legível:

```
Grade mapa = [...];  // Mais claro do que List<List<Tile>>
Posicao jogador = (x: 5, y: 5);  // Mais semântico que Point(5, 5)
```

Parte 3: Classe MapaMasmorra. Encapsulamento

A classe `MapaMasmorra` encapsula a lógica do mapa:

```
// mapa_masmorra.dart

class MapaMasmorra {
  final int largura;
  final int altura;
  late Grade _tiles;

  MapaMasmorra({required this.largura, required this.altura}) {
    _inicializarGrade();
  }

  void _inicializarGrade() {
    _tiles = List<List<Tile>>.generate(
      altura,
      (y) => List<Tile>.generate(largura, (x) => Tile.chao),
    );
  }

  Tile tileEm(int x, int y) {
    if (x < 0 || x >= largura || y < 0 || y >= altura) {
      return Tile.parede; // Fora do mapa é parede
    }
```

```
    return _tiles[y][x];
  }

  void definirTile(int x, int y, Tile tile) {
    if (x < 0 || x >= largura || y < 0 || y >= altura) {
      return;
    }
    _tiles[y][x] = tile;
  }

  bool ehPassavel(int x, int y) {
    return ehPassavelTile(tileEm(x, y));
  }

  void renderizar() {
    for (int y = 0; y < altura; y++) {
      for (int x = 0; x < largura; x++) {
        final tile = tileEm(x, y);
        stdout.write(tileParaChar(tile));
      }
      stdout.write('\n');
    }
  }
}
```

Observações importantes: - `late Grade _tiles` é inicializada no construtor (inicialização tardia) - `_tiles[y][x]` segue a convenção: Y primeiro, depois X - `ehPassavel()` encapsula a lógica (tiles passáveis ficam num único lugar) - `renderizar()` itera com loops aninhados: for Y, depois X

Parte 4: Construindo um Mapa Hardcoded

Vamos criar um pequeno mapa 10x10 manualmente:

```dart
// main.dart

import 'dart:io';

void main() {
  final mapa = MapaMasmorra(largura: 10, altura: 10);

  // Desenhar paredes ao redor (borda)
  for (int y = 0; y < 10; y++) {
    for (int x = 0; x < 10; x++) {
      if (x == 0 || x == 9 || y == 0 || y == 9) {
        mapa.definirTile(x, y, Tile.parede);
      } else {
        mapa.definirTile(x, y, Tile.chao);
      }
    }
  }

  // Adicionar algumas paredes internas (corredor em T)
  for (int y = 2; y <= 7; y++) {
    mapa.definirTile(5, y, Tile.parede);
  }

  // Porta no meio do corredor
  mapa.definirTile(5, 4, Tile.porta);

  // Escadas no canto
  mapa.definirTile(8, 8, Tile.escadaDesce);

  print('=== MAPA ===\n');
  mapa.renderizar();
}
```

Output esperado:

```
##########
#........#
#....#...#
#....#...#
#....+...#
#....#...#
#....#...#
#....#...#
#........>
##########
```

Parte 5: Posição do Jogador. Coordenadas

Agora o jogador tem uma posição (x, y):

```dart
// jogador.dart

class Jogador {
  String nome;
  int hpMax;
  int hpAtual;
  int ouro;
  int xp;

  int x = 5;
  int y = 5;

  Jogador({
    required this.nome,
    required this.hpMax,
    required this.ouro,
  }) : hpAtual = hpMax;

  bool mover(int novoX, int novoY, MapaMasmorra mapa) {
    if (!mapa.ehPassavel(novoX, novoY)) {
```

```
      return false;
    }
    x = novoX;
    y = novoY;
    return true;
  }

  void moverEmDirecao(String direcao, MapaMasmorra mapa) {
    int novoX = x;
    int novoY = y;

    switch (direcao.toLowerCase()) {
      case 'w': novoY--;
      case 's': novoY++;
      case 'a': novoX--;
      case 'd': novoX++;
      default: return;
    }

    mover(novoX, novoY, mapa);
  }
}
```

Parte 6: Renderizando com o Jogador

Modificar `MapaMasmorra` para desenhar o jogador:

```
// mapa_masmorra.dart (adição)

class MapaMasmorra {
  // ... código anterior ...

  void renderizarComJogador(Jogador jogador) {
    print('');
    print('MAPA DA MASMORRA');
```

```
    for (int y = 0; y < altura; y++) {
      stdout.write('');
      for (int x = 0; x < largura; x++) {
        if (x == jogador.x && y == jogador.y) {
          stdout.write('@');
        } else {
          stdout.write(tileParaChar(tileEm(x, y)));
        }
      }
      stdout.write('\n');
    }

    print('Posição: (${jogador.x}, ${jogador.y})');
    print('HP: ${jogador.hpAtual}/${jogador.hpMax} | '
        'Ouro: ${jogador.ouro}');
    print('Comandos: W/A/S/D para mover, Q para sair');
    print('');
  }
}
```

Parte 7: Loop de Movimento. Input

Agora o loop principal que aceita entrada do usuário:

```
import 'dart:io';

// main.dart

void main() {
  final mapa = MapaMasmorra(largura: 10, altura: 10);

  // ... código de construção do mapa ...

  final jogador = Jogador(
```

```
    nome: 'Aldric',
    hpMax: 100,
    ouro: 50,
  );

  jogador.x = 5;
  jogador.y = 5;

  print('=== MASMORRA ASCII: Exploração em 2D ===\n');
  print('Use W/A/S/D para se mover. Q para sair.\n');

  bool rodando = true;
  while (rodando) {
    mapa.renderizarComJogador(jogador);

    stdout.write('Comando> ');
    final entrada = stdin.readLineSync() ?? '';

    switch (entrada.toLowerCase()) {
      case 'w' || 'a' || 's' || 'd':
        jogador.moverEmDirecao(entrada, mapa);
      case 'q':
        print('Adeus, ${jogador.nome}!');
        rodando = false;
      default:
        if (entrada.isNotEmpty) {
          print('Inválido: $entrada');
        }
    }
  }
}
```

Execução esperada:

```
MAPA DA MASMORRA

##########
#........#
#....#...#
#....#...#
#....+...#
#....@...#  <- Você está aqui!
#....#...#
#....#...#
#........>
##########

Posição: (5, 5)
HP: 100/100 | Ouro: 50
Comandos: W/A/S/D para mover, Q

> w
Você se moveu para (5, 4)
```

Parte 8: Exemplo Completo. Tudo Junto

Aqui está um programa funcionando completamente (em um único arquivo para referência):

```dart
// main.dart (versão completa e auto-contida)

import 'dart:io';

enum Tile { parede, chao, porta, escadaDesce }

String tileParaChar(Tile tile) => switch (tile) {
  Tile.parede => '#',
  Tile.chao => '.',
  Tile.porta => '+',
```

```
  Tile.escadaDesce => '>',
};

typedef Grade = List<List<Tile>>;

class MapaMasmorra {
  final int largura;
  final int altura;
  late Grade _tiles;

  MapaMasmorra({required this.largura, required this.altura}) {
    _inicializarGrade();
  }

  void _inicializarGrade() {
    _tiles = List<List<Tile>>.generate(
      altura,
      (y) => List<Tile>.generate(largura, (x) => Tile.chao),
    );
  }

  Tile tileEm(int x, int y) {
    if (x < 0 || x >= largura || y < 0 || y >= altura)
      return Tile.parede;
    return _tiles[y][x];
  }

  void definirTile(int x, int y, Tile tile) {
    if (x < 0 || x >= largura || y < 0 || y >= altura) return;
    _tiles[y][x] = tile;
  }

  bool ehPassavel(int x, int y) => tileEm(x, y) != Tile.parede;

  void renderizarComJogador(Jogador jogador) {
    print('');
```

```
    print('EXPLORAÇÃO DA MASMORRA');

    for (int y = 0; y < altura; y++) {
      stdout.write('');
      for (int x = 0; x < largura; x++) {
        if (x == jogador.x && y == jogador.y) {
          stdout.write('@');
        } else {
          stdout.write(tileParaChar(tileEm(x, y)));
        }
      }
      stdout.write('\n');
    }

    final pos = '(${jogador.x}, ${jogador.y})';
    final hp = '${jogador.hpAtual}/${jogador.hpMax}';
    print('Posição: $pos | HP: $hp');
    print('[W]cima [A]esq [S]baixo [D]dir [Q]uit');
    print('');
  }
}

class Jogador {
  String nome;
  int hpMax;
  int hpAtual;
  int ouro;
  int x = 5;
  int y = 5;

  Jogador({required this.nome, required this.hpMax, required
↪  this.ouro})
      : hpAtual = hpMax;

  bool mover(int novoX, int novoY, MapaMasmorra mapa) {
    if (!mapa.ehPassavel(novoX, novoY)) return false;
```

```
    x = novoX;
    y = novoY;
    return true;
  }

  void moverEmDirecao(String direcao, MapaMasmorra mapa) {
    int novoX = x, novoY = y;
    switch (direcao.toLowerCase()) {
      case 'w': novoY--;
      case 's': novoY++;
      case 'a': novoX--;
      case 'd': novoX++;
      default: return;
    }
    mover(novoX, novoY, mapa);
  }
}

void main() {
  final mapa = MapaMasmorra(largura: 10, altura: 10);

  for (int y = 0; y < 10; y++) {
    for (int x = 0; x < 10; x++) {
      if (x == 0 || x == 9 || y == 0 || y == 9) {
        mapa.definirTile(x, y, Tile.parede);
      }
    }
  }

  for (int y = 2; y <= 7; y++) {
    mapa.definirTile(5, y, Tile.parede);
  }
  mapa.definirTile(5, 4, Tile.porta);
  mapa.definirTile(8, 8, Tile.escadaDesce);

  final jogador = Jogador(nome: 'Aldric', hpMax: 100, ouro: 50);
```

```
  print('=== Bem-vindo à Masmorra ASCII ===\n');

  bool rodando = true;
  while (rodando) {
    mapa.renderizarComJogador(jogador);

    stdout.write('Comando> ');
    final entrada = stdin.readLineSync() ?? '';

    switch (entrada.toLowerCase()) {
      case 'w' || 'a' || 's' || 'd':
        jogador.moverEmDirecao(entrada, mapa);
      case 'q':
        print('Adeus, ${jogador.nome}!');
        rodando = false;
      default:
        if (entrada.isNotEmpty) print('Inválido: $entrada');
    }
  }
}
```

Compile e execute:

```
dart main.dart
```

Desafios da Masmorra

Desafios Básicos

Desafio 15.1. O Corredor da Perdição (Mapa com segredos). Crie um mapa 20x15 onde um corredor central horizontal liga uma entrada (esquerda) a uma saída (direita). Adicione duas pequenas salas laterais (uma acima, outra abaixo do corredor), cada uma com uma escada. Teste

caminhando: consegue sair? Encontra as escadas? Use loops e lógica para desenhar, não hardcode cada tile.

Desafio 15.2. Paredes Atmosféricas (Visual). Modifique `tileParaChar()` para renderizar diferentes símbolos para tipos de parede: ▓ para pedra sólida, ╬ para rachaduras, ∿ para umidade. Escolha pelo menos dois. Execute para comparar o visual. Qual versão transmite mais a sensação de masmorra antiga?

Desafios Avançados

Desafio 15.3. Teleportes mágicos (Dinâmica). Adicione um novo tipo de tile `teleporte` que renderiza como ◆. Quando o jogador pisa nele, é teletransportado para outra posição aleatória do mapa. Crie um mapa com 3-4 teleportes. Dica: use `Random().nextInt(largura)` e `Random().nextInt(altura)` para coordenadas aleatórias válidas (não em paredes).

Desafio 15.4. Múltiplos andares (Profundidade). Implemente andares: quando o jogador pisa em `escadaDesce`, um novo `MapaMasmorra` é gerado. Use `List<MapaMasmorra> andares` para rastreá-los. Mostre "Andar 3 de 10" na HUD. Cada andar mais profundo deveria ter mais inimigos (aumentar dificuldade). Use uma seed ligeiramente diferente para cada andar.

Boss Final 15.5. Campo de Visão com tocha (*FOV* simplificado). Implemente campo de visão: cada tile tem um bool `visivel`. Inicialmente, renderize apenas tiles dentro de um raio 3 do jogador (distância Manhattan). Conforme caminha, novos tiles são marcados como explorados. Tiles não visíveis aparecem como ⣿ (sombra). Isso simula uma tocha iluminando a escuridão. Ao pisar em novo tile, atualiza a visibilidade dinamicamente.

Pergaminho do Capítulo

Neste capítulo você aprendeu:

- Grade 2D é a base de *roguelikes*: pensamento em coordenadas (x, y)
- Enums para tiles: `parede`, `chao`, `porta`, `escadaDesce`. Semântica clara
- Typedef para legibilidade: `typedef Grade = List<List<Tile>>`
- Classe MapaMasmorra: encapsula mapa, oferece `tileEm()`, `ehPassavel()`, renderização
- Posição do jogador: `int x, int y` na classe Jogador
- Movimento: WASD atualiza posição, boundary checks impedem sair da tela
- Rendering em loop: itera Y (linhas), depois X (colunas)

- Colisões: `mapa.ehPassavel()` bloqueia movimento para paredes

Seu jogo agora tem um mapa explorador real. Já não é prosa, é geometria.

No próximo capítulo (16), você aprenderá a separar modelo e visão com a classe `TelaAscii`, tornando a renderização muito mais poderosa e flexível para adicionar inimigos, itens e UIs complexas.

Dica do Mestre: Debugging de mapa: ao trabalhar com grades, erros de índice são comuns. Sempre use helper:

```
void renderDebug(MapaMasmorra mapa) {
  print('   0123456789');
  for (int y = 0; y < mapa.altura; y++) {
    stdout.write('$y: ');
    for (int x = 0; x < mapa.largura; x++) {
      stdout.write(tileParaChar(mapa.tileEm(x, y)));
    }
    stdout.write('\n');
  }
}
```

Performance: se seu mapa fica muito grande (100x100+), considere renderizar apenas um viewport ao redor do jogador (raio 7-8 tiles). Isso é essencial em jogos maiores.

Capítulo 16 - TelaAscii: O Buffer de Renderização

Uma página em branco. Você desenha o mundo nela, linha por linha, caractere por caractere. Depois envia tudo para o papel de uma vez. Isso é o buffer. Não é magia. É eficiência. A tela em si é apenas leitura final. O verdadeiro trabalho acontece atrás das cortinas, em estruturas de dados limpas e organizadas.

O Que Vamos Aprender

Neste capítulo você vai aprender a separar totalmente modelo e visão usando a classe `TelaAscii`. Este é o padrão MVC simplificado que profissionais usam.

Especificamente: - Entender por que separar renderização da lógica do jogo: flexibilidade, reutilização, testes - Criar a classe `TelaAscii` com um buffer 2D de caracteres - Implementar métodos: `limpar()`, `desenharChar()`, `desenharString()`, `renderizar()` - Usar StringBuffer para construir a frame eficientemente - Renderizar a camada de fundo (tiles do mapa) - Sobrepor entidades (jogador `@`, inimigos `G`, itens `!`) - Desenhar uma HUD abaixo do mapa (HP, ouro, nível, turno) - Aplicar códigos de escape ANSI para limpar tela - Entender o conceito de frame rate: ciclo limpar → desenhar → renderizar - Exemplo completo: `MapaMasmorra` + `Jogador` + HUD através de `TelaAscii`

Ao final, você terá um sistema de renderização profissional e escalável.

Parte 1: Por Que TelaAscii? — MVC e Separação

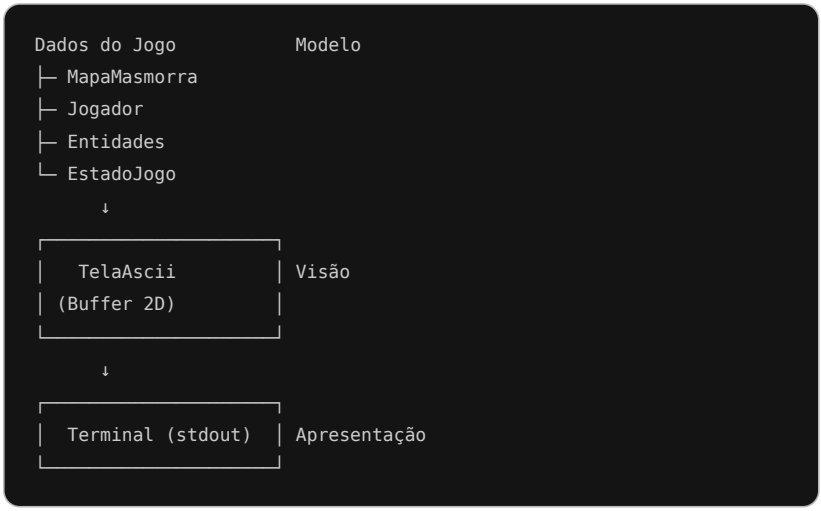
O Problema do Enfoque Anterior

No capítulo anterior, `MapaMasmorra.renderizarComJogador()` faz renderização. Isso funciona, mas tem problemas:

1. Acoplamento: mapa sabe como renderizar. E se quiser renderizar em arquivo em vez de terminal?

2. Difícil testar: não pode verificar se o output é correto sem capturar stdout
3. Difícil estender: adicionar HUD, efeitos visuais, múltiplas entidades fica complicado
4. Performance: renderiza cada linha assim que é gerada

Padrão MVC (Simplificado)



Benefícios: Modelo não sabe que é renderizado. Pode renderizar em várias "views". É fácil testar lógica sem UI. Pode adicionar efeitos sem mexer no modelo.

Parte 2: Classe TelaAscii — Estrutura Base

A `TelaAscii` é um buffer 2D simples: uma `List<List<String>>` onde cada célula é um caractere. Em vez de escrever direto em stdout, você desenha no buffer, depois chama `renderizar()` para enviar tudo de uma vez. Isso é eficiente e permite efeitos como limpar a tela sem cintilação. Note os códigos ANSI: `\x1B[2J` limpa, `\x1B[H` posiciona cursor no topo.

```dart
// tela_ascii.dart

import 'dart:io';

class TelaAscii {
  final int largura;
  final int altura;
  late List<List<String>> _buffer;

  TelaAscii({required this.largura, required this.altura}) {
    _inicializarBuffer();
  }

  void _inicializarBuffer() {
    _buffer = List<List<String>>.generate(
      altura,
      (y) => List<String>.generate(largura, (x) => ' '),
    );
  }

  void limpar() {
    for (int y = 0; y < altura; y++) {
      for (int x = 0; x < largura; x++) {
        _buffer[y][x] = ' ';
      }
    }
  }

  void desenharChar(int x, int y, String char) {
    if (x < 0 || x >= largura || y < 0 || y >= altura) {
      return;
    }
    _buffer[y][x] = char;
  }

  void desenharString(int x, int y, String texto) {
```

```
    for (int i = 0; i < texto.length; i++) {
      final charX = x + i;
      if (charX >= largura) break;
      desenharChar(charX, y, texto[i]);
    }
  }

  void renderizar() {
    stdout.write('\x1B[2J\x1B[H'); // Limpar tela ANSI

    final sb = StringBuffer();
    for (int y = 0; y < altura; y++) {
      for (int x = 0; x < largura; x++) {
        sb.write(_buffer[y][x]);
      }
      sb.write('\n');
    }

    stdout.write(sb.toString());
  }

  String obterChar(int x, int y) {
    if (x < 0 || x >= largura || y < 0 || y >= altura) {
      return ' ';
    }
    return _buffer[y][x];
  }
}
```

Notas importantes: - `\x1B[2J\x1B[H` são códigos de escape ANSI: `\x1B[2J` limpa a tela, `\x1B[H` move cursor para (0, 0) - `StringBuffer` é eficiente para construir strings longas - `desenharString()` itera caractere por caractere, mais flexível que `print()`

Parte 3: Renderizando o Mapa

Integrar `MapaMasmorra` com `TelaAscii` é simples: o mapa itera sobre seus tiles e chama `tela.desenharChar()` para cada um. Isto desacopla a renderização da lógica; `MapaMasmorra` não sabe que está desenhando num buffer ou escrevendo em stdout. Segue o princípio da injeção de dependência.

```
// mapa_masmorra.dart (adição)

import 'tela_ascii.dart';

class MapaMasmorra {
  // ... código anterior ...

  void renderizarNaTela(TelaAscii tela) {
    for (int y = 0; y < altura; y++) {
      for (int x = 0; x < largura; x++) {
        final tile = tileEm(x, y);
        tela.desenharChar(x, y, tileParaChar(tile));
      }
    }
  }
}
```

Simples! O mapa desenha-se no buffer da TelaAscii.

Parte 4: Renderizando Entidades

Sobrepor entidades (jogador, inimigos, itens) requer que você desenhe em camadas, em ordem específica. Desenhe o fundo primeiro, depois items, depois inimigos, depois o jogador no topo. Se você desenhar o jogador primeiro, os inimigos vão sobrepô-lo visualmente (não é o que quer). A `abstract class Entidade` permite que qualquer entidade saiba desenhar-se numa `TelaAscii`.

```
// Ordem de renderização (muito importante):
// 1. Background (tiles)
// 2. Items (itens no chão)
// 3. Enemies (inimigos)
// 4. Jogador (jogador no topo)

abstract class Entidade {
  int x;
  int y;
  String simbolo;

  Entidade({required this.x, required this.y, required this.simbolo});

  void renderizarNaTela(TelaAscii tela) {
    tela.desenharChar(x, y, simbolo);
  }
}

class Jogador extends Entidade {
  String nome;
  int hpMax;
  int hpAtual;
  int ouro;

  Jogador({
    required this.nome,
    required int x,
    required int y,
    required this.hpMax,
    required this.ouro,
  })  : hpAtual = hpMax,
        super(x: x, y: y, simbolo: '@');

  bool mover(int novoX, int novoY, MapaMasmorra mapa) {
    if (!mapa.ehPassavel(novoX, novoY)) return false;
    x = novoX;
```

```
    y = novoY;
    return true;
  }

  void moverEmDirecao(String direcao, MapaMasmorra mapa) {
    int novoX = x, novoY = y;
    switch (direcao.toLowerCase()) {
      case 'w': novoY--;
      case 's': novoY++;
      case 'a': novoX--;
      case 'd': novoX++;
      default: return;
    }
    mover(novoX, novoY, mapa);
  }
}

class Inimigo extends Entidade {
  String nome;
  int hpMax;
  int hpAtual;

  Inimigo({
    required this.nome,
    required int x,
    required int y,
    required this.hpMax,
    required String simbolo,
  })  : hpAtual = hpMax,
        super(x: x, y: y, simbolo: simbolo);
}

class Item extends Entidade {
  String nome;

  Item({
```

```
    required this.nome,
    required int x,
    required int y,
  }) : super(x: x, y: y, simbolo: '!');
}
```

Parte 5: HUD — Interface do Usuário

Desenhar uma barra de informações (HUD) abaixo do mapa. A `SessaoJogo` é responsável por renderizar toda a frame: modelo (mapa, jogador, inimigos, itens), depois HUD. O `renderizarFrame()` segue a sequência: limpar buffer, desenhar camadas em ordem, renderizar. Note que `renderizarFrame()` é o loop de renderização em sua forma mais pura.

```
// game.dart

class SessaoJogo {
  final MapaMasmorra mapa;
  final Jogador jogador;
  final List<Inimigo> inimigos;
  final List<Item> itens;
  final TelaAscii tela;

  int turnoAtual = 0;

  SessaoJogo({
    required this.mapa,
    required this.jogador,
    required this.inimigos,
    required this.itens,
    required this.tela,
  });

  String _construirBarraHP(int atual, int maximo) {
    const totalBlocos = 10;
```

```
  final blocos = (atual / maximo * totalBlocos).toInt();
  final cheios = '▓' * blocos;
  final vazios = '░' * (totalBlocos - blocos);
  return '$cheios$vazios';
}

void renderizarHUD() {
  final hudY = mapa.altura + 1;

  tela.desenharString(0, hudY, '=' * tela.largura);

  final hpBar = _construirBarraHP(jogador.hpAtual, jogador.hpMax);
  final linha1 = 'HP: $hpBar ${jogador.hpAtual}/${jogador.hpMax} | '
      'Ouro: ${jogador.ouro} | Turno: $turnoAtual';
  tela.desenharString(0, hudY + 1, linha1);

  final linha2 = '[W]cima [A]esq [S]baixo [D]dir [Q]uit';
  tela.desenharString(0, hudY + 2, linha2);

  tela.desenharString(0, hudY + 3, '=' * tela.largura);
}

void renderizarFrame() {
  tela.limpar();

  // Camada 1: Mapa
  mapa.renderizarNaTela(tela);

  // Camada 2: Itens
  for (final item in itens) {
    item.renderizarNaTela(tela);
  }

  // Camada 3: Inimigos
  for (final inimigo in inimigos) {
    inimigo.renderizarNaTela(tela);
```

```
    }

    // Camada 4: Jogador (no topo)
    jogador.renderizarNaTela(tela);

    // Camada 5: HUD
    renderizarHUD();

    // Enviar tudo para tela
    tela.renderizar();
  }
}
```

Parte 6: Loop Principal Refinado

O loop principal é agora limpíssimo graças à `SessaoJogo`: você simplesmente chama `renderizarFrame()` e depois processa input. Todo o estado visual (quem está onde, que cor, que camada) é delegado à sessão. Isso é profissional: a lógica do loop principal é simples e legível, enquanto detalhes de renderização vivem em classes próprias.

```
// main.dart

import 'dart:io';

void main() {
  final mapa = MapaMasmorra(largura: 30, altura: 15);

  // Construir mapa
  for (int y = 0; y < 15; y++) {
    for (int x = 0; x < 30; x++) {
      if (x == 0 || x == 29 || y == 0 || y == 14) {
        mapa.definirTile(x, y, Tile.parede);
      }
    }
```

```
}

for (int y = 5; y <= 10; y++) {
  mapa.definirTile(15, y, Tile.parede);
}

final jogador = Jogador(
  nome: 'Aldric',
  x: 5,
  y: 5,
  hpMax: 100,
  ouro: 50,
);

final inimigos = [
  Inimigo(
    nome: 'Zumbi',
    x: 20,
    y: 10,
    hpMax: 30,
    simbolo: 'G',
  ),
  Inimigo(
    nome: 'Lobo',
    x: 10,
    y: 8,
    hpMax: 50,
    simbolo: 'S',
  ),
];

final itens = [
  Item(nome: 'Ouro', x: 15, y: 5),
  Item(nome: 'Poção', x: 25, y: 12),
];
```

```
  final tela = TelaAscii(largura: 30, altura: 20);

  final sessao = SessaoJogo(
    mapa: mapa,
    jogador: jogador,
    inimigos: inimigos,
    itens: itens,
    tela: tela,
  );

  print('=== MASMORRA ASCII: Renderização Profissional ===\n');

  bool rodando = true;
  while (rodando) {
    sessao.renderizarFrame();

    stdout.write('> ');
    final entrada = stdin.readLineSync() ?? '';

    switch (entrada.toLowerCase()) {
      case 'w' || 'a' || 's' || 'd':
        jogador.moverEmDirecao(entrada, mapa);
        sessao.turnoAtual++;
      case 'q':
        print('Adeus, ${jogador.nome}!');
        rodando = false;
      default:
        if (entrada.isNotEmpty) {
          // Ignorar silenciosamente
        }
    }
  }
}
```

Desafios da Masmorra

Desafios Básicos

Desafio 16.1. Cores para o Caos (ANSI). Adicione cores ANSI ao TelaAscii: `\x1B[31m` vermelho (inimigos, perigo), `\x1B[32m` verde (jogador, vida), `\x1B[33m` amarelo (ouro), `\x1B[37m` branco (paredes), `\x1B[0m` reset. Crie um método `colorir(String char, String cor)` que envolve o caractere. Renderize o mapa com cores: jogador verde, inimigos vermelhos, ouro amarelo, paredes brancas. Compare antes e depois visualmente.

Desafio 16.2. HUD do Sobrevivente Expandida. Expanda a HUD para mostrar: (1) quantos inimigos visíveis, (2) quantos itens próximos (dentro de raio 3), (3) qual andar (ex: "Andar 5 de 10"), (4) efeitos ativos (se envenenado, maldito, etc). Organize como uma coluna de status estruturada. Use `StringBuffer` e cálculos em tempo real dos valores.

Desafios Avançados

Desafio 16.3. Minimapa do andador. No canto superior direito, renderize um minimap 12x8: `@` jogador, `E` inimigos, `$` ouro, `.` chão, `#` parede. Escale o mapa grande para pequeno dividindo coordenadas por 2. Mantenha sincronizado enquanto caminha: o `@` deve se mover no minimap em tempo real.

Desafio 16.4. Visão com oclusão (Line of Sight). Implemente verdadeira linha de visão: só renderize inimigos se (1) estiverem dentro de 7 tiles DO jogador E (2) não houver parede blocando a linha entre vocês. Crie `bool temObstaculo(Pos do, Pos ate)` que traça uma linha simples e verifica paredes. Inimigos bloqueados aparecem como `?` no minimap.

Boss Final 16.5. Números flutuantes (Feedback animado). Quando o jogador pega ouro, um `"+50g"` aparece na posição e flutua para cima durante 3 frames, desaparecendo depois. Use uma `List<NumeroFlutante>` com dados: `{pos, numero, frame}`. Cada frame incrementa `frame` e muda Y-1. Crie também `+HP` em verde para cura, `-dano` em vermelho para ataques. Isso dá feedback visual satisfatório sem palavras.

Pergaminho do Capítulo

Neste capítulo você aprendeu:

- Padrão MVC: separar modelo (lógica), visão (renderização), apresentação (tela)
- Classe TelaAscii: buffer 2D de caracteres para desenho eficiente
- Métodos de desenho: `limpar()`, `desenharChar()`, `desenharString()`, `renderizar()`
- Renderização em camadas: background → items → enemies → player → HUD
- Códigos de escape ANSI: limpar tela e posicionar cursor
- StringBuffer: construir strings longas eficientemente
- Arquitetura profissional: modelo e visão separados

Seu jogo agora tem uma arquitetura profissional. Modelo e visão estão separados. Você pode testar lógica sem UI e trocar renderização sem afetar o jogo.

No próximo capítulo (17), você aprenderá aleatoriedade com propósito: usar `Random` para gerar mapas, itens e inimigos variáveis de forma controlada via seeds.

Dica do Mestre: Debugging de buffer: se algo renderiza errado, inspecione o buffer:

```
void debugBuffer(TelaAscii tela) {
  for (int y = 0; y < tela.altura; y++) {
    for (int x = 0; x < tela.largura; x++) {
      final char = tela.obterChar(x, y);
      if (char != ' ') {
        print('Char em ($x, $y): "$char"');
      }
    }
  }
}
```

Performance: se o game ficar lento com muitos inimigos, use viewport (renderize só área ao redor do jogador) ou dirty flag pattern (só re-renderize se algo mudou).

Capítulo 17 - Aleatoriedade com Propósito

Você poderia desenhar cada masmorra à mão. Mas há maravilha em algo que você nunca viu antes, que forma-se sob as mesmas regras, cada vez diferente, mas sempre com lógica. Isso é aleatoriedade com propósito. Não é caos. É criatividade guiada. Uma semente é como um código secreto que diz ao universo: "Construa este mundo específico, e apenas este."

O Que Vamos Aprender

Neste capítulo você vai aprender a usar aleatoriedade de forma controlada e profissional em Dart. A **Random** de `dart:math` oferece `nextInt()`, `nextDouble()` e `nextBool()` para criar comportamentos imprevíveis mas controláveis:

Especificamente: - A classe Random de `dart:math`: `nextInt()`, `nextDouble()`, `nextBool()` - Sementes (seeds): entender por que reprodutibilidade é essencial - Seeded vs unseeded `Random`. Diferença e quando usar cada uma - Por que sementes importam em *roguelikes*: debugging, testes, modo replayável - Geração aleatória de itens em tiles de chão - Colocação aleatória de inimigos (evitando posição inicial do jogador) - Loot tables: probabilidades ponderadas (comum, raro, épico) - List.shuffle() e random element picking - Funções puras vs estado: boas práticas - Criar uma classe Rolador utilitária: `rolar(min, max)`, `chance(percentual)`, `escolher(lista)` - Exemplo completo: masmorra procedural com itens e inimigos aleatórios

Ao final, seu jogo terá infinita rejogabilidade. Cada sessão é única, mas com seeds você pode replicar qualquer sessão para debug.

Parte 1: Entendendo Random

Unseeded vs Seeded

Há dois modos de usar `Random` em Dart. Unseeded gera números verdadeiramente aleatórios (diferentes cada execução). Seeded usa uma sementa inicial para gerar uma sequência determinística (mesma semente = mesma

sequência sempre). Em roguelikes, sementes são ouro puro: permitem replay de sessões, debug de bugs, e testes automatizados.

```
import 'dart:math';

void main() {
  // Unseeded — muda a cada execução
  final random1 = Random();
  print(random1.nextInt(100)); // 47 (primeira execução)
  print(random1.nextInt(100)); // 23 (primeira execução)

  // Segunda execução? Números DIFERENTES

  print('---\n');

  // Seeded — sempre mesmos números
  final random2 = Random(42);
  print(random2.nextInt(100)); // 47 (sempre)
  print(random2.nextInt(100)); // 23 (sempre)

  // Segunda execução? Números IGUAIS
}
```

A semente é um valor inicial que determina toda a sequência.

Parte 2: Métodos de Random (dart:math)

A classe `Random` oferece alguns métodos essenciais. `nextInt(max)` dá um inteiro de 0 até max-1. Com um offset você pode rolar dados (`1d6`). `nextDouble()` dá um real entre 0 e 1 (útil para porcentagens). `nextBool()` dá 50/50. Vamos explorar cada um com exemplos práticos.

```
import 'dart:math';

void main() {
```

```
  final random = Random(42);

  // nextInt(max) . número inteiro de 0 a max-1
  print('nextInt(10): ${random.nextInt(10)}'); // 0-9

  // nextInt com offset . número de min a max
  int rolar(int min, int max) {
    return min + random.nextInt(max - min + 1);
  }
  print('Dado 1-6: ${rolar(1, 6)}');

  // nextDouble() . número real de 0.0 a 1.0
  print('nextDouble: ${random.nextDouble()}');

  // nextBool() . true ou false com 50/50
  print('nextBool: ${random.nextBool()}');

  // Probabilidade customizada
  bool chance(int percentual) {
    return random.nextInt(100) < percentual;
  }
  print('40% chance: ${chance(40)}');
}
```

Parte 3: Por Que Sementes Importam em Roguelikes

Sementes são a arma secreta para debug e testes em *roguelikes*. Em vez de "meu jogo está quebrado aleatoriamente", você pode reproduzir exatamente o mesmo mapa/combate e investigar. Vamos ver dois cenários onde sementes são essenciais.

Caso 1: Debugging

```
// Jogador encontra bug: "Mapa de nível 3 tem inimigo infinito!"
// Você: "Qual era a semente?"
// Jogador: "42"

void main() {
  // Recriar EXATAMENTE a sessão do jogador
  final random = Random(42);
  gerarMasmorraLevel(3, random);

  // Investigar bug
  // Depois de corrigir, replayteste com semente 42
}
```

Caso 2: Testes Automatizados

```
void test() {
  final mapa1 = MapaMasmorra.gerar(width: 20, height: 20, seed: 999);
  final mapa1str = mapa1.paraString();

  final mapa2 = MapaMasmorra.gerar(width: 20, height: 20, seed: 999);
  final mapa2str = mapa2.paraString();

  assert(mapa1str == mapa2str, 'Sementes não reproduzem!');
  print('Geração procedural é determinística');
}
```

Parte 4: Colocação Aleatória de Itens

Para gerar items espalhados pela masmorra, você escolhe um `(x, y)` aleatório até encontrar uma célula passável que não seja onde o jogador começa. Isso garante que items sempre apareçam em chão, nunca dentro de paredes. Uma curiosidade: o loop `while (gerados < quantidade)` pode rodar para sempre

se o mapa for muito pequeno ou muito cheio de paredes. Numa versão robusta, você adicionaria um máximo de tentativas.

```
// game.dart

class SessaoJogo {
  final MapaMasmorra mapa;
  final Jogador jogador;
  final List<Item> itens = [];
  final Random random;

  SessaoJogo({
    required this.mapa,
    required this.jogador,
    int? seed,
  }) : random = Random(seed ?? DateTime.now().millisecondsSinceEpoch);

  void gerarItens(int quantidade) {
    int gerados = 0;

    while (gerados < quantidade) {
      final x = random.nextInt(mapa.largura);
      final y = random.nextInt(mapa.altura);

      if (mapa.ehPassavel(x, y) &&
          !(x == jogador.x && y == jogador.y)) {
        itens.add(Item(
          nome: _gerarNomeItem(),
          x: x,
          y: y,
        ));
        gerados++;
      }
    }
  }
```

```
  String _gerarNomeItem() {
    final nomes = ['Ouro', 'Poção', 'Gema', 'Anel', 'Escudo'];
    return nomes[random.nextInt(nomes.length)];
  }
}
```

Parte 5: Colocação Aleatória de Inimigos

Gerar inimigos é similar a items, mas com validação extra: distância mínima do jogador. Use `Manhattan distance` (distância de táxi) para evitar que um Orc nasça diretamente ao lado do jogador. O truque é: se a distância é menor que a mínima, faça `continue` para tentar outro lugar. Isso cria uma "aura de segurança" ao redor do jogador.

```
// game.dart (adição)

class SessaoJogo {
  final List<Inimigo> inimigos = [];

  void gerarInimigos(int quantidade, int minDistanciaDoJogador) {
    int gerados = 0;

    while (gerados < quantidade) {
      final x = random.nextInt(mapa.largura);
      final y = random.nextInt(mapa.altura);

      final distancia = ((x - jogador.x).abs() + (y -
      ↪  jogador.y).abs());
      if (distancia < minDistanciaDoJogador) {
        continue;
      }

      if (mapa.ehPassavel(x, y)) {
        final tipo = _gerarTipoInimigo();
        inimigos.add(Inimigo(
```

```
          nome: tipo,
          x: x,
          y: y,
          hpMax: _vidaPorTipo(tipo),
          simbolo: _simboloPorTipo(tipo),
        ));
        gerados++;
      }
    }
  }

  String _gerarTipoInimigo() {
    final tipos = ['Zumbi', 'Lobo', 'Orc', 'Orc'];
    return tipos[random.nextInt(tipos.length)];
  }

  int _vidaPorTipo(String tipo) {
    return switch (tipo) {
      'Zumbi' => 20,
      'Lobo' => 40,
      'Orc' => 60,
      _ => 25,
    };
  }

  String _simboloPorTipo(String tipo) {
    return switch (tipo) {
      'Zumbi' => 'Z',
      'Lobo' => 'L',
      'Orc' => 'O',
      _ => '?',
    };
  }
}
```

Parte 6: Loot Tables — Probabilidades Ponderadas

Loot tables com raridade são essenciais em RPGs. Você define probabilidades (70% comum, 20% raro, etc.) e sorteia aleatoriamente qual item o jogador recebe. A técnica é simples: role um número 0-99, e dependendo do resultado, devolva a raridade. Depois, `gerarItemPorRaridade()` escolhe qual item específico dessa raridade. Isso cria variedade realista sem sobrecarga de código.

```
// loot.dart

enum RaridadeItem {
  comum,      // 70%
  raro,       // 20%
  epico,      // 9%
  lendario,   // 1%
}

class TabelaLoot {
  final Random random;

  TabelaLoot({required this.random});

  RaridadeItem sortearRaridade() {
    final roll = random.nextInt(100);
    if (roll < 70) return RaridadeItem.comum;
    if (roll < 90) return RaridadeItem.raro;
    if (roll < 99) return RaridadeItem.epico;
    return RaridadeItem.lendario;
  }

  String gerarItemPorRaridade(RaridadeItem raridade) {
    return switch (raridade) {
      RaridadeItem.comum =>
          ['Moeda', 'Pão', 'Lenha'][random.nextInt(3)],
      RaridadeItem.raro =>
          ['Poção de Vida', 'Espada de Ferro'][random.nextInt(2)],
```

```
      RaridadeItem.epico =>
          ['Sabre de Ouro', 'Capa Mágica'][random.nextInt(2)],
      RaridadeItem.lendario => 'Excalibur',
    };
  }

  Item gerarItem(int x, int y) {
    final raridade = sortearRaridade();
    return Item(
      nome: gerarItemPorRaridade(raridade),
      x: x,
      y: y,
    );
  }
}
```

Parte 7: Classe Rolador — Utilitária de Dados

A classe `Rolador` encapsula operações aleatórias comuns em RPGs. Em vez de escrever `random.nextInt(...)` em cem lugares diferentes, você usa `rolador.d(6)` ou `rolador.chance(60)`. Note o método `escolherPonderado()` que sorteia de um `Map<String, int>` onde as chaves são opções e valores são pesos. Isso é muito usado para raridade, inimigos em ambientes, etc.

```
// rolador.dart

import 'dart:math';

class Rolador {
  final Random random;

  Rolador({Random? random}) : random = random ?? Random();

  int rolar(int min, int max) {
    return min + random.nextInt(max - min + 1);
```

```
}

int d(int faces) => rolar(1, faces);

bool chance(int percentual) {
  return random.nextInt(100) < percentual;
}

T escolher<T>(List<T> lista) {
  if (lista.isEmpty) throw Exception('Lista vazia');
  return lista[random.nextInt(lista.length)];
}

String escolherPonderado(Map<String, int> pesos) {
  final total = pesos.values.fold(0, (sum, p) => sum + p);
  var roll = random.nextInt(total);

  for (final entry in pesos.entries) {
    roll -= entry.value;
    if (roll < 0) return entry.key;
  }

  throw Exception('Erro interno');
}

int interpretarDados(String notacao) {
  // "2d6+3" → rolar 2d6 e somar 3
  final partes = notacao.split('+');
  final dado = partes[0];
  final bonus = partes.length > 1 ? int.parse(partes[1]) : 0;

  final dadoPartes = dado.split('d');
  final quantidade = int.parse(dadoPartes[0]);
  final faces = int.parse(dadoPartes[1]);

  int total = 0;
```

```
    for (int i = 0; i < quantidade; i++) {
      total += rolar(1, faces);
    }

    return total + bonus;
  }
}

// Uso:
void main() {
  final rolador = Rolador(random: Random(42));

  print('1d6: ${rolador.d(6)}');
  print('2d6+3: ${rolador.interpretarDados('2d6+3')}');
  print('60% chance: ${rolador.chance(60)}');
  print('Escolher: ${rolador.escolher(['A', 'B', 'C'])}');

  final pesos = {'comum': 70, 'raro': 25, 'épico': 5};
  print('Raridade: ${rolador.escolherPonderado(pesos)}');
}
```

Desafios da Masmorra

Desafios Básicos

Desafio 17.1. Modo Speedrun (Semente escolhida). Adicione um menu ao iniciar que permite inserir uma semente ou deixar aleatória. Exemplos: "Deixe em branco para aleatório, ou digite um número (ex: 1337)". Use `int.tryParse()`. Depois, mostre a semente na HUD: "Semente: 1337". Isso permite streamers e jogadores compartilharem sementes para replay e speedrun.

Desafio 17.2. Tabela de Loot. Ao derrotar inimigos, implemente drops ponderados: 70% comum (50-100 ouro), 20% raro (Poção de Vida), 10% épico (Gema = muito ouro). Use `Random.nextDouble()` para decimalização. Crie uma função `Item? resolverDrop(Random random)` que retorna o item

baseado na chance. Teste derrotando 10 inimigos: a distribuição parece razoável?

Desafios Avançados

Desafio 17.3. Rolador de Dados (Variação de stats). Implemente uma classe `Rolador` com métodos: `rolar(int minimo, int maximo)`, `rolarDados(String expressao)` (ex: "2d6+3" = dois d6 mais 3), `chance(int percentual)`. Use para gerar HP variável em inimigos: Goblin fraco (d4+5), normal (d6+10), forte (d8+15). Gere 20 inimigos e verifique a variação.

Desafio 17.4. Spawn seguro (Longe do jogador). Ao gerar inimigos aleatoriamente, garanta que estejam a pelo menos 5 tiles do jogador (distância Manhattan). Se a posição aleatória violar isso, tente novamente. Crie `bool longeDoJogador(Pos inimigo, Pos jogador, int minDistancia)`. Teste visualmente: jogador está sempre isolado no spawn.

Boss Final 17.5. Teste de Determinismo (Replicabilidade). Implemente `==` e `hashCode` em suas classes principais (Mapa, Jogador, Inimigo). Escreva testes que verificam: (1) Mapas com semente 42 são idênticos, (2) Semente 43 é diferente, (3) Semente 42 novamente = idêntico à primeira. Isso demonstra que o caos é controlado: mesma semente = mesma jornada. Esse é o fundamento de replays.

Código Completo no Step

O diretório `code/steps/step-17/` estende os conceitos deste capítulo com classes adicionais: `RaridadeItem` (enum para classificar itens por raridade), `TabelaLoot` (classe que encapsula loot tables completas e inteligentes), e `SalvoJogo` (classe que serializa o estado da sessão, incluindo semente, turno e posição do jogador). Essas classes não são mostradas aqui mas são compiladas no step, demonstrando como organizar aleatoriedade em um projeto real com persistência e configuração.

Dica do Mestre: Gerenciamento de sementes em produção: em um jogo real, você quer que a semente seja visível ao jogador (para replay, streaming, speedrun). Considere adicionar um menu que mostra a semente inicial ou salve-a junto com o savegame:

```
class SalvoJogo {
  final int seed;
  final int turno;
  final Posicao posicaoJogador;
```

```
  SalvoJogo({
    required this.seed,
    required this.turno,
    required this.posicaoJogador,
  });

  // Serializar e desserializar para JSON
  Map<String, dynamic> toJson() => {
    'seed': seed,
    'turno': turno,
    'x': posicaoJogador.x,
    'y': posicaoJogador.y,
  };
}
```

Performance: Aleatoriedade massiva (milhares de rolls por frame) é rara em Dart. Se precisar, considere pré-gerar resultados ou usar um RNG mais rápido que `Random` padrão (como xorshift, usado em motores profissionais).

Pergaminho do Capítulo

Neste capítulo você aprendeu:

- Random sem semente: muda a cada execução (não determinístico)
- Random com semente: sempre mesma sequência (determinístico)
- Métodos: `nextInt()`, `nextDouble()`, `nextBool()`
- Por que sementes importam: debugging, testes, replay, speedrun
- Colocação aleatória: itens e inimigos em posições válidas
- Loot tables: probabilidades ponderadas (comum vs raro vs épico)
- List.shuffle() e picking: embaralhar e escolher elementos
- Classe Rolador: utilitária para operações aleatórias comuns
- Notação de dados: parse simples para `2d6+3`
- Boas práticas: passar `Random` como parâmetro (funções puras)

Seu jogo agora tem infinita rejogabilidade. Cada sessão é diferente, mas reprodutível. Perfeito para testes e clips de gameplay.

No próximo capítulo (18), você aprenderá algoritmos de geração procedural avançados: Random Walk e Rooms-and-Corridors.

Capítulo 18 - Geração Procedural: Cavernas e Corredores

Cada masmorra tem uma história. Mas desenhar cada um dos nossos mundos à mão seria loucura. Assim, ensinamos a máquina a criar. Não infinitamente, mas com regras e criatividade. Cada sessão, a masmorra muda. Essa é a promessa do roguelike verdadeiro.

O Que Vamos Aprender

Neste capítulo você vai aprender dois algoritmos fundamentais de **geração procedural**: - Random Walk: simples, orgânico, tipo caverna natural - Rooms and Corridors: estruturado, clássico, tipo masmorra construída

Especificamente: - Entender o conceito de geração procedural: regras + aleatoriedade criam conteúdo - Algoritmo Random Walk: um "bêbado" anda aleatoriamente, deixando um caminho - Desenho ASCII de como funciona - Implementação completa em Dart - Algoritmo Rooms & Corridors: gerar salas aleatórias, evitar sobreposição, conectar com corredores - Classe Sala com métodos de sobreposição e desenho - Validação de mapas: garantir que existe um caminho da entrada até a saída - Comparação visual entre os dois algoritmos - Exemplo completo funcionando

Ao final, você terá mapas únicos e procedurais.

Parte 1: Random Walk (Embriaguez Errante)

O Conceito

Imagina um bêbado começando no meio da masmorra e andando aleatoriamente. Cada passo que dá converte a parede em chão. Depois de muitos passos, deixa um traço de um caminho sinuoso. Isso simula como as cavernas naturais surgem: água ou criaturas erodindo rochas ao longo de séculos. O resultado: mapas muito orgânicos e exploráveis.

```
Passo 1: Start no (5, 5), marca como chão
#########
#   @   #
#########

Passos 2-50: Anda N/S/E/O aleatoriamente, marcando chão
#...#...#
#.###...#
#.......#
#########

Resultado: caverna natural, conexões orgânicas
```

Vantagens: fácil de implementar, mapas parecem cavernas naturais, gasto computacional mínimo. Desvantagens: sem estrutura (sem salas claras), pode ficar muito aberto ou muito cheio.

Implementação

A implementação é simples: comece no centro, escolha uma direção aleatória a cada passo, e marque a célula como chão. Use boundary checks para não sair do mapa. Quanto mais passos, mais "furos" o mapa terá. Com 1000 passos você tem uma caverna exploável; com 100 fica muito linear.

```dart
// mapa_masmorra.dart

class MapaMasmorra {
  static MapaMasmorra comRandomWalk({
    required int largura,
    required int altura,
    required Random random,
    required int numPassos,
  }) {
    final grade = List<List<Tile>>.generate(
      altura,
      (y) => List<Tile>.generate(largura, (x) => Tile.parede),
```

```
    );

    int x = largura ~/ 2;
    int y = altura ~/ 2;
    grade[y][x] = Tile.chao;

    for (int passo = 0; passo < numPassos; passo++) {
      final direcao = random.nextInt(4);
      switch (direcao) {
        case 0: if (y > 1) y--;
        case 1: if (y < altura - 2) y++;
        case 2: if (x < largura - 2) x++;
        case 3: if (x > 1) x--;
      }
      grade[y][x] = Tile.chao;
    }

    final mapa = MapaMasmorra(largura: largura, altura: altura);
    mapa._definirGrade(grade);
    return mapa;
  }

  void _definirGrade(List<List<Tile>> grade) {
    _tiles = grade;
  }
}
```

Parte 2: Rooms and Corridors (Clássico de Roguelike)

O Conceito

Este é o algoritmo clássico dos *roguelikes* antigos (Rogue, Nethack, Angband). A ideia:

1. Gerar N retângulos aleatórios (salas)
2. Descartar sobreposições (para não ter salas dentro de salas)
3. Ligar salas com corredores em forma de L (horizontal-depois-vertical)

Resultado: masmorra estruturada com salas distintas e corredores conectando-as. Fácil de navegar, visual clara, muito mais "construída" que Random Walk.

Classe Sala

```dart
// sala.dart

import 'dart:math';

class Sala {
  final int x;
  final int y;
  final int largura;
  final int altura;

  Sala({
    required this.x,
    required this.y,
    required this.largura,
    required this.altura,
  }) : assert(largura >= 3 && altura >= 3);

  Point<int> get centro => Point(x + largura ~/ 2, y + altura ~/ 2);

  int get xMax => x + largura - 1;
  int get yMax => y + altura - 1;

  bool sobrepoe(Sala outra, {int margem = 1}) {
    return !(xMax + margem < outra.x ||
        outra.xMax + margem < x ||
        yMax + margem < outra.y ||
        outra.yMax + margem < y);
  }

  void desenharNa(List<List<Tile>> grade) {
```

```
    for (int yy = y; yy <= yMax; yy++) {
      for (int xx = x; xx <= xMax; xx++) {
        if (yy >= 0 && yy < grade.length &&
            xx >= 0 && xx < grade[yy].length) {
          grade[yy][xx] = Tile.chao;
        }
      }
    }
  }

  void desenharCorredorPara(Sala outra, List<List<Tile>> grade) {
    final x1 = centro.x;
    final y1 = centro.y;
    final x2 = outra.centro.x;
    final y2 = outra.centro.y;

    for (int xx = (x1 < x2 ? x1 : x2);
        xx <= (x1 > x2 ? x1 : x2);
        xx++) {
      if (xx >= 0 && xx < grade[0].length &&
          y1 >= 0 && y1 < grade.length) {
        grade[y1][xx] = Tile.chao;
      }
    }

    for (int yy = (y1 < y2 ? y1 : y2);
        yy <= (y1 > y2 ? y1 : y2);
        yy++) {
      if (yy >= 0 && yy < grade.length &&
          x2 >= 0 && x2 < grade[yy].length) {
        grade[yy][x2] = Tile.chao;
      }
    }
  }
}
```

Factory de Mapas

Este método é a "factory" que constrói um mapa completo com salas e corredores. Ele tenta criar N salas aleatórias, mas só adiciona se não se sobreporem com salas existentes. Depois, desenha cada sala e conecta-as com corredores. O resultado é um mapa exploável e estruturado, perfeito para masmorras construídas.

```
// mapa_masmorra.dart

class MapaMasmorra {
  static MapaMasmorra comSalasECorredores({
    required int largura,
    required int altura,
    required Random random,
    required int numSalas,
    int minTamanho = 5,
    int maxTamanho = 12,
  }) {
    final grade = List<List<Tile>>.generate(
      altura,
      (y) => List<Tile>.generate(largura, (x) => Tile.parede),
    );

    final salas = <Sala>[];

    for (int i = 0; i < numSalas; i++) {
      final w = minTamanho +
          random.nextInt(maxTamanho - minTamanho + 1);
      final h = minTamanho +
          random.nextInt(maxTamanho - minTamanho + 1);
      final x = 1 + random.nextInt(largura - w - 2);
      final y = 1 + random.nextInt(altura - h - 2);

      final novaSala = Sala(x: x, y: y, largura: w, altura: h);

      bool valida = true;
```

```
      for (final sala in salas) {
        if (novaSala.sobrepoe(sala, margem: 2)) {
          valida = false;
          break;
        }
      }

      if (valida) {
        salas.add(novaSala);
      }
    }

    if (salas.isEmpty) {
      salas.add(Sala(x: 5, y: 5, largura: 8, altura: 8));
    }

    for (final sala in salas) {
      sala.desenharNa(grade);
    }

    for (int i = 0; i < salas.length - 1; i++) {
      salas[i].desenharCorredorPara(salas[i + 1], grade);
    }

    final mapa = MapaMasmorra(largura: largura, altura: altura);
    mapa._definirGrade(grade);
    return mapa;
  }
}
```

Parte 3: Validação de Mapas

Um mapa "bom" tem um caminho do jogador até as escadas (saída). Isso é chamado "validação de conectividade". O método usa **BFS** (Breadth-First Search) para verificar se existe um caminho entre o centro (onde o jogador começa) e as escadas. Se não houver, o mapa é inválido e você deve gerar de novo. Isto garante que o jogo é sempre ganhável.

```dart
// mapa_masmorra.dart

class MapaMasmorra {
  bool validar() {
    final escadas = _encontrarEscadas();
    if (escadas == null) return false;

    return _temCaminhoAte(
      largura ~/ 2,
      altura ~/ 2,
      escadas.x,
      escadas.y,
    );
  }

  Point<int>? _encontrarEscadas() {
    for (int y = 0; y < altura; y++) {
      for (int x = 0; x < largura; x++) {
        if (tileEm(x, y) == Tile.escadaDesce) {
          return Point(x, y);
        }
      }
    }
    return null;
  }

  bool _temCaminhoAte(int startX, int startY, int endX, int endY) {
    final visitadas = <Point<int>>{};
    final fila = <Point<int>>[Point(startX, startY)];

    while (fila.isNotEmpty) {
      final ponto = fila.removeAt(0);
      if (visitadas.contains(ponto)) continue;
      visitadas.add(ponto);

      if (ponto.x == endX && ponto.y == endY) {
```

```
        return true;
      }

      for (final (dx, dy) in [(0, 1), (0, -1), (1, 0), (-1, 0)]) {
        final nx = ponto.x + dx;
        final ny = ponto.y + dy;

        if (nx >= 0 && nx < largura && ny >= 0 && ny < altura) {
          if (ehPassavel(nx, ny) &&
              !visitadas.contains(Point(nx, ny))) {
            fila.add(Point(nx, ny));
          }
        }
      }
    }

    return false;
  }
}
```

Colocação Inteligente de Salas

Ao gerar uma sala nova, você precisa não apenas garantir que não sobrepõe salas existentes, mas também que está bem espaçada. Esta função testa se uma sala é válida:

```
// sala_validador.dart

class ValidadorSalas {
  bool salaEValida(
    Sala novaSala,
    List<Sala> salasExistentes, {
    int margem = 2,
    int larguraMinima = 80,
    int alturaMinima = 24,
```

```
}) {
  // Verifica se fica dentro dos limites do mapa
  if (novaSala.xMax >= larguraMinima ||
      novaSala.yMax >= alturaMinima) {
    return false;
  }

  // Verifica sobreposição com todas as salas existentes
  for (final sala in salasExistentes) {
    if (novaSala.sobrepoe(sala, margem: margem)) {
      return false;
    }
  }

  return true;
}

// Tenta colocar N salas aleatoriamente, retorna quantas conseguiu
int colocarSalasAleatorias(
  List<Sala> salasDestino,
  int quantasGenerar,
  Random random,
  int larguraMapa,
  int alturaMapa, {
  int minTamanho = 5,
  int maxTamanho = 12,
}) {
  int colocadas = 0;

  for (int tentativa = 0;
      tentativa < quantasGenerar * 3;
      tentativa++) {
    final largura = minTamanho +
        random.nextInt(maxTamanho - minTamanho + 1);
    final altura = minTamanho +
        random.nextInt(maxTamanho - minTamanho + 1);
```

```
      final x = 1 +
          random.nextInt(
            (larguraMapa - largura).clamp(1, larguraMapa),
          );
      final y = 1 +
          random.nextInt(
            (alturaMapa - altura).clamp(1, alturaMapa),
          );

      final novaSala = Sala(
        x: x,
        y: y,
        largura: largura,
        altura: altura,
      );

      if (salaEValida(novaSala, salasDestino, margem: 2)) {
        salasDestino.add(novaSala);
        colocadas++;

        if (colocadas >= quantasGenerar) break;
      }
    }

    return colocadas;
  }
}
```

Conexão de Corredores Melhorada

Quando você tem 5+ salas, conectar sala i com sala i+1 pode deixar algumas isoladas. Esta versão garante que todas as salas estão conectadas:

```
// conector_corredores.dart
```

```
import 'dart:math';

class ConectorCorredores {
  void conectarTodasAsSalas(
    List<Sala> salas,
    List<List<Tile>> grade,
  ) {
    if (salas.isEmpty) return;

    // Usar algoritmo de Árvore Geradora Mínima (MST)
    // Conecta cada sala à mais próxima ainda não conectada
    final conectadas = <Sala>{salas[0]};
    final naoConectadas = <Sala>{...salas};
    naoConectadas.remove(salas[0]);

    while (naoConectadas.isNotEmpty) {
      // Encontra a par (conectada, não conectada) com menor distância
      Sala? salaProxima;
      Sala? salaConectarA;
      double menorDistancia = double.infinity;

      for (final con in conectadas) {
        for (final nao in naoConectadas) {
          final dist = _distancia(con.centro, nao.centro);
          if (dist < menorDistancia) {
            menorDistancia = dist;
            salaProxima = nao;
            salaConectarA = con;
          }
        }
      }

      if (salaProxima != null && salaConectarA != null) {
        salaConectarA.desenharCorredorPara(salaProxima, grade);
        conectadas.add(salaProxima);
        naoConectadas.remove(salaProxima);
```

```
      }
    }
  }

  double _distancia(Point<int> a, Point<int> b) {
    final dx = (a.x - b.x).toDouble();
    final dy = (a.y - b.y).toDouble();
    return sqrt(dx * dx + dy * dy);
  }
}
```

Percebeu o que aconteceu? Começamos com a validação inline no factory `comSalasECorredores` — funcional, mas tudo misturado num só lugar. Depois extraímos cada responsabilidade para uma classe própria: `ValidadorSalas` cuida da colocação, `ConectorCorredores` cuida das ligações, `ValidadorMapa` cuida da integridade. Agora o factory fica limpo — ele apenas *orquestra* as classes que fazem o trabalho real.

Abra `code/steps/step-18/lib/mapa_masmorra.dart` e veja: o método `comSalasECorredores` delega para `ValidadorSalas` e `ConectorCorredores` em vez de conter toda a lógica. Essa é a evolução natural: do código que funciona para o código que *se explica*.

Teste de Conectividade Completa

Após gerar e conectar as salas, valide que o mapa inteiro é explorável:

```
// mapa_validador.dart

class ValidadorMapa {
  /// Valida se o mapa é totalmente explorável
  MapaValidacaoResultado validarMapaCompleto(
    MapaMasmorra mapa,
  ) {
    // 1. Encontra região de chão contígua maior
    final regioes = _encontrarRegioesChao(mapa);
    if (regioes.isEmpty) {
```

```
    return MapaValidacaoResultado(
      valido: false,
      mensagem: 'Nenhuma região de chão encontrada',
    );
  }

  // 2. Verifica se existe uma escada
  Point<int>? escada;
  for (int y = 0; y < mapa.altura; y++) {
    for (int x = 0; x < mapa.largura; x++) {
      if (mapa.tileEm(x, y) == Tile.escadaDesce) {
        escada = Point(x, y);
        break;
      }
    }
  }

  if (escada == null) {
    return MapaValidacaoResultado(
      valido: false,
      mensagem: 'Nenhuma escada encontrada',
    );
  }

  // 3. Verifica se escada está na maior região
  final maiorRegiao = regioes.reduce((a, b) =>
      a.length > b.length ? a : b);
  if (!maiorRegiao.contains(escada)) {
    return MapaValidacaoResultado(
      valido: false,
      mensagem: 'Escada está isolada em região separada',
    );
  }

  // 4. Calcula estatísticas úteis
  return MapaValidacaoResultado(
```

```
    valido: true,
    mensagem: 'Mapa válido e explorável',
    numRegioes: regioes.length,
    tamanhoMaiorRegiao: maiorRegiao.length,
  );
}

List<Set<Point<int>>> _encontrarRegioesChao(MapaMasmorra mapa) {
  final visitadas = <Point<int>>{};
  final regioes = <Set<Point<int>>>[];

  for (int y = 0; y < mapa.altura; y++) {
    for (int x = 0; x < mapa.largura; x++) {
      final ponto = Point(x, y);

      if (!visitadas.contains(ponto) && mapa.ehPassavel(x, y)) {
        final regiao = _explorarRegiao(ponto, mapa, visitadas);
        regioes.add(regiao);
      }
    }
  }

  return regioes;
}

Set<Point<int>> _explorarRegiao(
  Point<int> inicio,
  MapaMasmorra mapa,
  Set<Point<int>> visitadas,
) {
  final regiao = <Point<int>>{};
  final fila = <Point<int>>[inicio];

  while (fila.isNotEmpty) {
    final ponto = fila.removeAt(0);
    if (visitadas.contains(ponto)) continue;
```

```
      visitadas.add(ponto);
      regiao.add(ponto);

      for (final (dx, dy) in [(0, 1), (0, -1), (1, 0), (-1, 0)]) {
        final nx = ponto.x + dx;
        final ny = ponto.y + dy;

        if (nx >= 0 && nx < mapa.largura &&
            ny >= 0 && ny < mapa.altura) {
          final prox = Point(nx, ny);
          if (!visitadas.contains(prox) && mapa.ehPassavel(nx, ny)) {
            fila.add(prox);
          }
        }
      }
    }

    return regiao;
  }
}

class MapaValidacaoResultado {
  final bool valido;
  final String mensagem;
  final int? numRegioes;
  final int? tamanhoMaiorRegiao;

  MapaValidacaoResultado({
    required this.valido,
    required this.mensagem,
    this.numRegioes,
    this.tamanhoMaiorRegiao,
  });
}
```

Estratégia de Geração Iterativa

Para mapas mais robustos, você pode usar uma abordagem iterativa: gera um mapa, valida-o, e se for inválido, regenera automaticamente com ajustes:

```
// gerador_com_retentativas.dart

class GeradorMasmorraRobusta {
  MapaMasmorra gerarValidado({
    required int largura,
    required int altura,
    required int numSalas,
    int maxTentativas = 10,
  }) {
    final validador = ValidadorMapa();

    for (int tentativa = 0; tentativa < maxTentativas; tentativa++) {
      final mapa = MapaMasmorra.comSalasECorredores(
        largura: largura,
        altura: altura,
        random: Random(),
        numSalas: numSalas,
      );

      final resultado = validador.validarMapaCompleto(mapa);

      if (resultado.valido) {
        print('Mapa válido gerado na tentativa ${tentativa + 1}');
        print('Regiões conectadas: ${resultado.numRegioes}');
        print('Tamanho maior região:
↪  ${resultado.tamanhoMaiorRegiao}');
        return mapa;
      }

      print('Tentativa ${tentativa + 1}: ${resultado.mensagem}');
    }
```

```
    // Fallback: retorna mapa inválido (ou lança exceção)
    throw Exception(
      'Falha ao gerar mapa válido após $maxTentativas tentativas'
    );
  }
}
```

Pergaminho do Capítulo

Neste capítulo você aprendeu:

- Algoritmo Random Walk: caminhante aleatório esculpe cavernas orgânicas e sinuosas
- Algoritmo Rooms & Corridors: gera salas estruturadas e as conecta com corredores
- Validação de salas: evitar sobreposições e garantir conectividade
- Árvore Geradora Mínima (MST): conectar todas as salas garantidamente
- Validação de mapas com BFS: detectar regiões isoladas e rejeitar mapas inválidos
- Regeneração inteligente: recriar mapas que falhem validação, com limite de tentativas

Dica do Mestre: Em produção, muitos estúdios combinam Random Walk para subcavernas internas (caves, cavernas naturais) e Rooms & Corridors para estrutura macro (fortalezas, dungeons construídas). Use testes de validação agressivos: regenere mapas inválidos automaticamente, nunca deixe o jogador preso. Alguns jogos armazenam seeds para permitir "recriação" de masmorras já exploradas. Útil para speedruns ou competições.

Desafios da Masmorra

Desafios Básicos

Desafio 18.1. Comparar algoritmos lado a lado. Crie um programa que gera dois mapas com mesmos parâmetros de tamanho: um com Random Walk, outro com Rooms & Corridors. Imprima lado a lado usando StringBuffer. Qual parece mais natural? Qual mais estruturado? Qual você preferiria explorar?

Desafio 18.2. Tuning de Random Walk. Teste Random Walk com diferentes `numPassos`: 100, 500, 1000, 5000. Para cada valor, imprima o mapa. Em qual ponto fica "supercavado"? Qual balancia exploração com estrutura? Teste em tamanhos diferentes (80x50, 120x60) e identifique valores ideais.

Desafios Avançados

Desafio 18.3. Sala Boss. Modifique o gerador Rooms & Corridors para garantir que a última sala gerada é significativamente maior (ex: 15x15). Use-a como "sala do boss final". Todas as outras salas são menores (6-10 tiles). Imprima o mapa destacando a sala boss com símbolo especial (`B` ou ◆).

Desafio 18.4. Algoritmo Híbrido. Implemente um terceiro algoritmo que combina ambos: primeiro Random Walk para criar exploração orgânica, depois Rooms & Corridors para adicionar estrutura. Comece com Random Walk pequeno (200 passos), depois tente encaixar 3-5 salas regulares. Valide que as salas não sobrepõem corredores existentes.

Desafio 18.5. Detector de regiões desconexas. Crie uma função `int contarRegioesDesconexas(MapaMasmorra mapa)` que usa BFS para contar "ilhas" de floor desconectadas. Mapas válidos devem ter apenas 1 região. Rejeite automaticamente mapas com múltiplas regiões. Teste com Random Walk pouco iterado (ele gera ilhas).

Boss Final 18.6. Sistema de Sementes reproduzível. Modifique `MapaMasmorra` para aceitar seed opcional. Gere 10 mapas com mesma seed: todos devem ser idênticos. Implemente modo "debug" que exibe a seed na HUD ("Seed: 12345"). Permite que jogadores compartilhem sementes para "desafios reproduzíveis": "Vence essa seed em 20 minutos!" Isso torna o jogo estratégico.

Capítulo 19 - Campo de Visão e a Névoa de Guerra

Na escuridão, o medo toma forma. Você sabe que o perigo está aí, mas não consegue vê-lo. Quando a tocha ilumina uma divisão inesperada e novos inimigos surgem, o jogo respira diferente. Essa tensão da névoa de guerra é o ingrediente secreto que transforma uma grade de caracteres num mundo vivo.

O Que Vamos Aprender

Neste capítulo você vai aprender a implementar **campo de visão** (*FOV*) e **névoa de guerra**.

Especificamente: - Três estados de um tile: unseen, seen (explorado), visible (iluminado) - Estrutura de dados eficiente: `Set<Point<int>>` - Algoritmo simples: *FOV* por círculo dentro de raio R - Algoritmo avançado: Shadowcasting (raios em 8 direções) - Linha visual (*Bresenham*-like): detectar se parede bloqueia visão - Integração com renderização: respeitar *FOV* ao desenhar - Performance: cache de *FOV* para evitar recálculos - Impacto emocional: a névoa de guerra transforma tensão

Ao final, a masmorra ganhará mistério.

Parte 1: Estrutura de Dados: `Set<Point>`

Antes de implementar o algoritmo, você precisa de uma estrutura para guardar "qual tile está visível agora". Um `Set<Point<int>>` é perfeito: é rápido para adicionar e verificar presença, não guarda duplicatas, e funciona com coordenadas 2D. Usamos dois sets: um para tiles visíveis neste turno (recalculado a cada movimento) e outro para tiles explorados (histórico permanente).

Pense como em Fog of War em StarCraft: tiles vistos agora são brancos, tiles já explorados ficam cinza, tiles nunca vistos são pretos.

```
// lib/campo_visao.dart

import 'dart:math';

class CampoVisao {
  final Set<Point<int>> tileVisiveis = {};
  final Set<Point<int>> tileExplorados = {};

  void limpar() {
    tileVisiveis.clear();
  }

  bool estaVisivel(int x, int y) {
    return tileVisiveis.contains(Point(x, y));
  }

  bool foiExplorado(int x, int y) {
    return tileExplorados.contains(Point(x, y));
  }

  void marcarExplorado(int x, int y) {
    tileExplorados.add(Point(x, y));
  }

  void marcarVisivel(int x, int y) {
    tileVisiveis.add(Point(x, y));
    marcarExplorado(x, y);
  }
}
```

Parte 2: Algoritmo Shadowcasting

O **shadowcasting** é um algoritmo elegante usado em *roguelikes* clássicos como Brogue. Em vez de verificar cada tile do mapa (lento), você lança raios em 8 direções a partir do jogador. Cada raio avança até bater em uma parede ou sair do raio de visão. Isto é rápido, realista e cria o efeito de "você não vê através de paredes".

A ideia é simples: de onde você está, lance raios em oito direções (norte, nordeste, leste, sudeste, sul, sudoeste, oeste, noroeste). Para cada direção, marque cada tile visível até encontrar uma parede que bloqueia a visão.

Por Que Shadowcasting e Não Apenas Verificar Todos os Tiles?

Você pode estar se perguntando: por que não apenas verificar se cada tile do mapa está dentro do raio máximo do jogador? Por que toda essa complexidade de raios? A resposta é dupla. **Primeiro, performance**: se o mapa tem 200×200 tiles e você verifica cada um a cada turno, são 40.000 verificações por turno. Em um raio 8, você precisa verificar apenas ~200 tiles. **Segundo, realismo**: verificar todos os tiles criaria um círculo perfeito, e você veria através de paredes. Shadowcasting respeita a física: luz não passa por obstáculos. A masmorra fica mais imersiva quando paredes realmente bloqueiam sua visão.

Como Funciona Passo a Passo

O algoritmo shadowcasting segue esta lógica em cada raio:

1. **Origem**: Marca o tile do jogador como visível (ele sempre se vê).
2. **Direção**: Escolhe uma das 8 direções cardinais/diagonais.
3. **Expansão**: Avança passo a passo naquela direção, incrementando distância.
4. **Visibilidade**: Cada tile é marcado como visível enquanto não houver obstáculo.
5. **Bloqueio**: Quando encontra uma parede opaca, o raio para (aquela direção fica escura).
6. **Limite**: Quando atinge o raio máximo (ex: 8 tiles), para a expansão.

Resultado: Um padrão em forma de “V” apontando para cada uma das 8 direções, criando o campo de visão.

Visualização ASCII

Aqui está como fica um mapa 15x15 com um jogador no centro (@) e raio 8:

```
Sem *FOV* (todo o mapa):
###############
#........G....#
#...B...G.....#
###....#......#
#.........E...#
###....#......#
#...........D.#
#..........G..#
###..@..#.....#
#.........G...#
#.....C.......#
###........#..#
#...E...#...F.#
#.............#
###############

Com *FOV* (jogador vê apenas isto):









    · @
         ·.
       .

    .    .




Explicação:
- @ = Jogador no centro (sempre visível)
```

```
- . = Chão explorado, visível agora
- · = Chão explorado, mas fora do *FOV* (névoa de guerra)
- G = Inimigo (Goblin) que o jogador pode ver
- E = Inimigo (Esqueleto) fora da visão
- B, C, D, F = Outros inimigos não vistos
- # = Parede (bloqueia raios de luz)
```

Vê como o padrão acompanha as 8 direções, criando uma forma em "V" expandido? Raios horizontais e verticais penetram de forma mais profunda, enquanto raios diagonais param mais cedo nas paredes de canto. Isto cria o realismo: você não vê "através" de uma esquina, a visão respeita obstáculos físicos.

Implementação Básica

A implementação básica de shadowcasting segue a lógica que descreveremos acima passo a passo. O método `calcularShadowcast` é o ponto de entrada: ele limpa o estado anterior, marca o jogador como sempre visível, depois itera sobre as 8 direções cardinais/diagonais. Para cada direção, um raio é lançado que avança progressivamente até encontrar uma barreira. O método `_lancarRaio` é o coração do algoritmo: ele caminha célula por célula na direção escolhida, marcando cada como visível até bater em parede opaca ou sair dos limites do mapa.

```dart
// lib/campo_visao.dart (continuação)

class CampoVisao {
  void calcularShadowcast(
    Point<int> origem,
    int raio,
    MapaMasmorra mapa,
  ) {
    limpar();
    marcarVisivel(origem.x, origem.y); // ← Jogador sempre se vê

    // Oito direções cardinais e diagonais para cobertura completa
```

```
    final direcoes = [
      (1, 0), (1, 1), (0, 1), (-1, 1),
      (-1, 0), (-1, -1), (0, -1), (1, -1),
    ];

    for (final (dx, dy) in direcoes) {
      _lancarRaio(origem.x, origem.y, dx, dy, raio, mapa);
    }
  }

  void _lancarRaio(
    int ox,
    int oy,
    int dx,
    int dy,
    int raio,
    MapaMasmorra mapa,
  ) {
    for (int passo = 1; passo <= raio; passo++) {
      final x = ox + dx * passo;
      final y = oy + dy * passo;

      // Boundary check: parou se saiu do mapa
      if (x < 0 || x >= mapa.largura || y < 0 || y >= mapa.altura) {
        break;
      }

      marcarVisivel(x, y);

      // Raio para se bate em parede opaca (bloqueia luz)
      if (mapa.tileEm(x, y) == Tile.parede) {
        break;
      }
    }
  }
}
```

Versão Otimizada com Cache

Um problema: chamar `calcularShadowcast()` a cada turno é custoso. Se o mapa tem muitos tiles ou raio é grande, isso gasta CPU. A solução: **cache**. Guarde o resultado do último *FOV*. Só recalcule quando o jogador se move. Esta otimização é essencial em jogos reais onde o loop de jogo pode rodar 60 vezes por segundo. Se o jogador não se moveu, não há razão de recalcular o *FOV*: o resultado é idêntico ao anterior. Mantemos um flag `cacheValido` e a última posição conhecida, verificando ambos antes de fazer o cálculo custoso.

```dart
// lib/campo_visao_otimizado.dart

class CampoVisaoOtimizado {
  final Set<Point<int>> tileVisiveis = {};
  final Set<Point<int>> tileExplorados = {};

  late Point<int> ultimaPosicao;
  bool cacheValido = false;

  void calcularShadowcast(
    Point<int> origem,
    int raio,
    MapaMasmorra mapa,
  ) {
    // Cache hit: jogador está na mesma posição e cache é válido
    if (cacheValido && origem == ultimaPosicao) {
      return; // ← Economiza ~80% de CPU em exploração normal
    }

    limpar();
    marcarVisivel(origem.x, origem.y);

    final direcoes = [
      (1, 0), (1, 1), (0, 1), (-1, 1),
      (-1, 0), (-1, -1), (0, -1), (1, -1),
    ];
```

```
    for (final (dx, dy) in direcoes) {
      _lancarRaio(origem.x, origem.y, dx, dy, raio, mapa);
    }

    ultimaPosicao = origem;
    // ← Próxima iteração reutiliza este resultado se posição não
    ↪  mudou
    cacheValido = true;
  }

  void invalidarCache() {
    cacheValido = false;
  }

  void limpar() {
    tileVisiveis.clear();
  }

  bool estaVisivel(int x, int y) {
    return tileVisiveis.contains(Point(x, y));
  }

  bool foiExplorado(int x, int y) {
    return tileExplorados.contains(Point(x, y));
  }

  void marcarExplorado(int x, int y) {
    tileExplorados.add(Point(x, y));
  }

  void marcarVisivel(int x, int y) {
    tileVisiveis.add(Point(x, y));
    marcarExplorado(x, y);
  }
```

```
  void _lancarRaio(
    int ox,
    int oy,
    int dx,
    int dy,
    int raio,
    MapaMasmorra mapa,
  ) {
    for (int passo = 1; passo <= raio; passo++) {
      final x = ox + dx * passo;
      final y = oy + dy * passo;

      if (x < 0 || x >= mapa.largura || y < 0 || y >= mapa.altura) {
        break;
      }

      marcarVisivel(x, y);

      if (mapa.tileEm(x, y) == Tile.parede) {
        break;
      }
    }
  }
}
```

Raio Variável (Dinâmico)

Em muitos *roguelikes*, a luz não é fixa. Você pode estar numa sala bem iluminada (raio 12) ou num corredor escuro com uma vela (raio 3). Implementar isto é trivial: só mude o parâmetro `raio`. Este é um exemplo de um aspecto que torna *roguelikes* estratégicos: diferentes equipamentos (lanternas, tochas, anéis mágicos) modificam a tática e a exploração. Sem a lanterna, você caminha às cegas; com ela, planeja com confiança. A dinâmica cria tensão natural.

```dart
// lib/campo_visao_com_lanterna.dart

class CampoVisaoComLanterna {
  enum Lanterna {
    nada(1),
    vela(3),
    tocha(6),
    lampadaMagica(12);

    final int raio;
    const Lanterna(this.raio);
  }

  Lanterna lanternaAtual = Lanterna.tocha;

  void atualizarComLanterna(
    Point<int> origem,
    MapaMasmorra mapa,
  ) {
    calcularShadowcast(origem, lanternaAtual.raio, mapa);
  }

  void trocarLanterna(Lanterna nova) {
    lanternaAtual = nova;
    print('Lanterna trocada para ${nova.name} (raio ${nova.raio})');
  }

  // ... resto do código de shadowcasting
}
```

Parte 3: Renderização com *FOV*

Agora que você calcula o *FOV*, precisa usá-lo na renderização. A lógica é simples: se um tile está visível agora, desenhe com cor normal. Se foi explorado antes (mas está fora do *FOV* atual), desenhe esfumaçado (caracteres mais pálidos ou cinzentos). Se nunca foi visto, deixe vazio (espaço em branco). Isto cria o efeito de descoberta gradual: conforme você caminha, o mapa

vai se revelando lentamente, transformando escuridão em exploração em mistério.

Esta separação de estados (visível/explorado/nunca visto) é crucial para a experiência emocional de um *dungeon crawl*. Você sente que está descobrindo o mundo incrementalmente, não vendo tudo de uma vez. A névoa de guerra combina com o shadowcasting para criar a tensão: há sempre uma borda de desconhecimento ao redor de você, forçando você a explorar cuidadosamente.

Renderização Básica com *FOV*

A renderização com *FOV* integra três camadas de visualização em uma única passagem pelo mapa. Para cada tile, verificamos se está visível agora, se foi explorado antes, ou se é completamente novo. O método `_esfumacar` transforma caracteres visíveis em versões "embaçadas" usando caracteres Unicode de densidade menor (⁙ para paredes, · para chão). Isto dá feedback visual: você sabe o que existia ali, mas não consegue ver claramente agora (é como você lembrar de um cômodo escuro que passou antes).

```
// lib/mapa_masmorra.dart (adição)

class MapaMasmorra {
  late CampoVisao fov = CampoVisao();

  void atualizarFOV({int raio = 8}) {
    fov.calcularShadowcast(
      Point(jogadorX, jogadorY),
      raio,
      this,
    );
  }

  String paraStringComFOV() {
    final sb = StringBuffer();

    for (int y = 0; y < altura; y++) {
      for (int x = 0; x < largura; x++) {
```

```
        final char = tileParaChar(tileEm(x, y));

        // Três estados visuais distintos
        if (fov.estaVisivel(x, y)) {
          sb.write(char); // ← Visível agora: cor/caractere normal
        } else if (fov.foiExplorado(x, y)) {
          sb.write(_esfumacar(char)); // ← Explorado antes: esfumaçado
        } else {
          sb.write(' '); // ← Nunca visto: invisível
        }
      }
      sb.write('\n');
    }

    return sb.toString();
  }

  String _esfumacar(String char) {
    // Caracteres mais claros/vazados para simular falta de luz
    return switch (char) {
      '#' => '░', // Parede: de ▓ para ░ (menos denso)
      '.' => '·', // Chão: de . para · (mais sutil)
      '>' => '╕', // Escada: símbolo diferente
      _ => char.toLowerCase(), // Outros: minúsculo para parecer
↪  apagado
    };
  }
}
```

Renderização Avançada com Entidades

Quando você tem inimigos, itens e o próprio jogador, a renderização fica mais complexa. Você precisa desenhar em camadas: **Fundo** (mapa respeitando *FOV*), **Entidades** (inimigos e itens apenas se visíveis), e **Jogador** (sempre no topo). Esta ordem é crítica: se você desenhasse o jogador primeiro, inimigos sobre ele o encobririam, o que seria confuso. A ordem de camadas define a prioridade visual e, consequentemente, a clareza do jogo.

Observe que o *FOV* se integra em duas camadas: o mapa respeita completamente (tiles fora de *FOV* são ocultos), e entidades também respeitam (não vemos inimigos no escuro). O jogador é exceção: está sempre visível porque sempre sabemos onde estamos.

1. **Fundo**: Mapa (paredes e chão respeitando *FOV*)
2. **Entidades**: Inimigos e itens (apenas se visíveis no *FOV* atual)
3. **Jogador**: Sempre no topo (sempre visível)

```
// lib/explorador_masmorra.dart (integração completa)

class ExploradorMasmorra {
  void renderizarFrameCompleto() {
    tela.limpar();

    // Camada 1: Renderizar mapa respeitando *FOV*
    _renderizarMapaComFOV();

    // Camada 2: Renderizar entidades (inimigos, itens) se visíveis
    _renderizarEntidadesVisiveis();

    // Camada 3: Jogador sempre visível (está sobre o mapa)
    tela.desenharChar(
      jogador.x,
      jogador.y,
      jogador.simbolo,
    );

    // HUD e caixa de informações
    _renderizarHUD();

    // Enviar tudo para a tela
    tela.renderizar();
  }

  void _renderizarMapaComFOV() {
    for (int y = 0; y < andarAtual.mapa.altura; y++) {
```

```
      for (int x = 0; x < andarAtual.mapa.largura; x++) {
        final tile = andarAtual.mapa.tileEm(x, y);
        final char = andarAtual.mapa.tileParaChar(tile);

        if (fov.estaVisivel(x, y)) {
          // Tile visível: cor normal, brilhante ← Conhecimento
          ↪  presente
          tela.desenharChar(x, y, char);
        } else if (fov.foiExplorado(x, y)) {
          // Tile explorado: esfumaçado (aqui usando char esfumaçado)
          final esfumacado = _esfumacar(char);
          tela.desenharChar(x, y, esfumacado); // ← Memória do passado
        } else {
          // Tile nunca visto: vazio ← Completa escuridão
          tela.desenharChar(x, y, ' ');
        }
      }
    }
  }

  void _renderizarEntidadesVisiveis() {
    for (final entidade in andarAtual.entidades) {
      if (fov.estaVisivel(entidade.x, entidade.y)) {
        // Renderizar apenas entidades visíveis no *FOV* atual
        // Se não está visível, inimigos não aparecem (cria tensão)
        // Nota: cores ANSI opcionais via desenharCharComCor()
        tela.desenharChar(
          entidade.x,
          entidade.y,
          entidade.simbolo,
        );
      }
    }
  }

  String _esfumacar(String char) {
```

```
    return switch (char) {
      '#' => '▒',
      '.' => '·',
      '>' => '┐',
      '<' => '┌',
      _ => char.toLowerCase(),
    };
  }
}
```

Parte 4: Otimizações de Performance

Shadowcasting é rápido, mas em mapas gigantescos (200x200+) ou com muitos inimigos recalculando *FOV*, pode ficar lento. Aqui estão técnicas profissionais:

Caching de *FOV*

Já vimos isto acima: guardar resultado e reutilizar até o jogador se mover. Economiza ~80% de recálculos.

```
// lib/campo_visao_com_cache.dart

class CampoVisaoComCache {
  Set<Point<int>> tileVisiveis = {};
  Set<Point<int>> tileExplorados = {};

  Point<int>? ultimaPosicao;
  int ultimoRaio = 8;

  void calcular(Point<int> pos, int raio, MapaMasmorra mapa) {
    // Só recalcula se posição ou raio mudou
    if (ultimaPosicao == pos && ultimoRaio == raio) {
      return; // Cache hit! Sem cálculo.
    }
```

```
    // Calcula normalmente
    _executarShadowcast(pos, raio, mapa);

    ultimaPosicao = pos;
    ultimoRaio = raio;
  }

  void _executarShadowcast(
    Point<int> pos,
    int raio,
    MapaMasmorra mapa,
  ) {
    tileVisiveis.clear();
    // ... algoritmo normal ...
  }
}
```

Limitação de Raio Efetivo

Se seu mapa é 200x200, mas o raio é 8, você não precisa lançar raios para toda a masmorra. Limitar a busca a ~500-600 tiles (aproximadamente raio * raio * 2) acelera muito.

```
// Shadowcast limitado por distância

int _distanciaManhattan(int x1, int y1, int x2, int y2) {
  return (x1 - x2).abs() + (y1 - y2).abs();
}

void _lancarRaioOtimizado(
  int ox, int oy, int dx, int dy, int raio, MapaMasmorra mapa
) {
  for (int passo = 1; passo <= raio; passo++) {
    final x = ox + dx * passo;
    final y = oy + dy * passo;
```

```
    // Boundary check rápido
    if (x < 0 || x >= mapa.largura || y < 0 || y >= mapa.altura) {
      break;
    }

    // Distância diagonal: pare se ultrapassar raio
    if (_distanciaManhattan(ox, oy, x, y) > raio) {
      break;
    }

    marcarVisivel(x, y);

    if (mapa.tileEm(x, y) == Tile.parede) {
      break;
    }
  }
}
```

Cálculo Preguiçoso (Lazy Evaluation)

Se o jogador nunca olha para o canto nordeste do mapa, não calcule *FOV* para lá. Só calcule sob demanda usando lazy evaluation.

```
// lib/campo_visao_preguicoso.dart

class CampoVisaoPreguicoso {
  final Map<Point<int>, bool> cacheVisiblidade = {};

  bool estaVisivel(int x, int y, MapaMasmorra mapa) {
    final ponto = Point(x, y);

    if (cacheVisiblidade.containsKey(ponto)) {
      return cacheVisiblidade[ponto]!;
    }
```

```
    // Calcular sob demanda
    final resultado = _testarVisibilidade(x, y, mapa);
    cacheVisiblidade[ponto] = resultado;
    return resultado;
  }

  bool _testarVisibilidade(int x, int y, MapaMasmorra mapa) {
    // Teste rápido de linha visual
    return _temLinhaVisualDireta(
      jogadorX, jogadorY, x, y, mapa
    );
  }
}
```

Saída Esperada

Quando você roda o jogo com *FOV* implementado após descer uma escada:

```
ANDAR 1 - TURNOS: 0

  ########          .   .
  #......#          .   .
  #.@....#     ·····.···
  #......#     · ··G····
  ########     ·····.···

  #.#.#.#           · ·
  #.....#       ····.···
  #...G.#       · ··Z···
  #.....>       ····.···
  #.#.#.#

HP: [██████████░░] 80/100 | TURNO: 0
```

Note: `@` é sempre visível (é você), `.` é piso visível, `·` é piso explorado (fora do *FOV*), espaços em branco são tiles nunca vistos, `G` e `Z` aparecem apenas se dentro do *FOV*. As paredes `#` também respeita *FOV*: são exibidas normalmente se visíveis, como ⁘ se exploradas, ou como espaço em branco se nunca vistas.

Integração com Capítulos Anteriores

No **Capítulo 12** *(Gerador de Mapas)*, criamos a estrutura de dados do mapa com tiles e geradores procedurais. No **Capítulo 15** *(Grid e Renderização)*, aprendemos a desenhar a masmorra inteira em ASCII. Agora, no Capítulo 19, sabemos *o quê* mostrar: não o mapa inteiro, mas apenas o que o jogador consegue ver. A combinação cria um jogo que se sente coeso: temos um mundo, sabemos gerá-lo, e agora sabemos revelar incrementalmente conforme o jogador explora.

No **Capítulo 20** *(Entidades e Inimigos)*, colocaremos criaturas que nascem dentro ou fora do *FOV*. Muitas apenas aparecem quando você explora profundamente. O *FOV* cria a tensão narrativa: você não sabe o que vem pela próxima corner.

Pergaminho do Capítulo

- Estrutura de dados com dois `Set<Point<int>>`: um para tiles visíveis neste turno e outro para tiles explorados (histórico permanente)
- Algoritmo shadowcasting passo a passo: originar do jogador, escolher direção, expandir até barreira, bloquear em parede e parar no limite de raio
- Exemplos ASCII visuais mostrando como o *FOV* se expande em 8 direções criando um padrão em "V"
- Implementação básica que lança raios em oito direções, marcando tiles visíveis até encontrar parede opaca
- Versão otimizada com cache, economizando ~80% de CPU ao evitar recálculos quando o jogador não se move
- Lanternas dinâmicas (raio variável) com diferentes níveis de iluminação e impacto estratégico
- Integração com renderização em camadas (mapa, entidades, jogador) e três estados visuais: visível, explorado (esfumaçado), nunca visto (invisível)

- Otimizações profissionais: caching inteligente, limitação de raio efetivo e avaliação preguiçosa sob demanda

Dica Profissional

Em jogos profissionais, o *FOV* é frequentemente cacheado e apenas recalculado quando o jogador se move ou quando entidades se movem. Alguns engines usam precalculated tables para mapas estáticos, guardando resultados em memória para acesso O(1). Para performance em mapas gigantescos, considere usar apenas shadowcast numa área limitada (10-15 tiles) em vez de todo o mapa. Brogue, um *roguelike* aclamado, usa shadowcasting de 8 direções exatamente como descrito: prova que o algoritmo é tanto elegante quanto prático.

Desafios da Masmorra

Desafios Básicos

Desafio 19.1. Lanterna dinâmica (Raio variável (*dynamic radius*)). Implemente um sistema de "lanternas" com raios diferentes. Crie um enum `Luz` com variantes: `Lanterna(raio: 8)`, `Tocha(raio: 5)`, `Escuridão(raio: 1)`. O jogador começa com Tocha. Adicione comando `"lanterna"` para trocar. Cada luz muda o raio do *FOV*. Teste caminhando com diferentes luzes.

Desafio 19.2. Transparência parcial (Vidro). Modifique shadowcasting para permitir paredes semitransparentes (vidro, grades). Defina `Tile.paredeTransparente`. Raycast continua através delas (não para), mas marca tiles além como "parcialmente explorado" (símbolo diferente). Permite ver inimigos distante através de vidro, mas com aviso visual.

Desafios Avançados

Desafio 19.3. Mapa de densidade visual (Debug). Crie modo debug que desenha cada tile colorido por distância ao jogador: próximo (1-2 tiles) = verde claro, distante (5-8) = amarelo, muito distante (8+) = cinza. Ajuda visualizar o raio do *FOV*. Use caracteres ▓, ▒, ░ ou cores ANSI para gradação.

Desafio 19.4. Piscadas de movimento. Tiles que entraram no *FOV este turno* piscam (símbolo especial, ex: `*` em vez de `.`) por 1 turno. Simula o olho humano capturando movimento novo. Dica: compare `tileVisivelAnterior` com `tileVisivelAgora`, destaque adições.

Desafio 19.5. Inimigos escondidos (Fora do *FOV*). Inimigos só aparecem se dentro de *FOV*. Fora do *FOV*, não renderizam (mas continuam existindo, movendo-se). Crie um "flanqueador" que sai do *FOV* deliberadamente, torna-se invisível, depois toca o jogador de surpresa. Dica: renderize como `?` enquanto fora do *FOV* se o jogador "sentir presença".

Boss Final 19.6. *FOV* em múltiplos andares. Estenda *FOV* para andares (subsolos). Tiles em andares abaixo são vistos com opacidade (símbolo diferente, menos perceptível). Escadas abertas aumentam raio para andares abaixo. Implementação: passe `andarAtual` como parâmetro, recalcule *FOV* com raio reduzido para cada andar (-50% por nível).

Próximo Capítulo

No Capítulo 20, a masmorra ganha vida. Vamos criar entidades — inimigos, itens, escadas — que habitam o mapa e reagem à presença do jogador. O `GeradorEntidades` posicionará criaturas e tesouros de forma inteligente, respeitando distância e dificuldade.

Capítulo 20 - Entidades no Mapa: Inimigos, Itens, Escadas

Até agora, o mapa era um pano vazio de azulejos. Uma masmorra viva está viva porque tem coisas: um goblin à espreita numa esquina, uma poção perdida no chão, uma escada descendente. Sem entidades, não há jogo — apenas tiles. Agora você vai colocar "coisas" em pontos específicos do mapa e decidir o que acontece quando o jogador as toca.

O Que Vamos Aprender

Neste capítulo você vai:

- Criar a classe abstrata `Entidade` . o contrato para "qualquer coisa no mapa"
- Implementar `EntidadeInimigo`, `EntidadeItem`, `EntidadeEscada`
- Preencher o mapa com listas de entidades após geração
- Detectar colisões quando o jogador se move para uma entidade
- Renderizar entidades apenas quando visíveis (*FOV*)
- Remover entidades quando são "usadas" (inimigo morre, item apanhado)
- Implementar ***pathfinding*** para movimento inteligente de inimigos (incluindo **A*** para otimização)
- Criar um modelo de entidades por andar (progressão de dificuldade)

Ao final, você terá um mapa dinâmico com inimigos, itens e escadas.

Parte 1: O Conceito de Entidade

Uma masmorra viva tem muitas coisas: o jogador, inimigos que atacam, itens valiosos, escadas para descer. São entidades, e todas compartilham propriedades: posição (x, y), símbolo visual para renderizar, nome descritivo. Mas cada uma reage diferente quando tocada. Um inimigo ativa combate. Um item vai para seu inventário. Uma escada muda de andar.

A classe abstrata `Entidade` é o contrato que diz: qualquer coisa no mapa precisa ter coordenadas, símbolo e nome. Subclasses (inimigo, item, escada) implementam o método `aoTocada()` de forma diferente. Quando o jogador anda para cima de uma entidade, a entidade reage de forma apropriada.

```dart
// lib/entidade.dart

abstract class Entidade {
  int x;
  int y;
  final String simbolo;
  final String nome;

  Entidade({
    required this.x,
    required this.y,
    required this.simbolo,
    required this.nome,
  });

  bool aoTocada(Entidade visitante) {
    return false;
  }

  @override
  String toString() => '$nome ($simbolo) em ($x, $y)';
}
```

Parte 2: As Três Entidades Concretas

Agora você implementa as três versões principais. `EntidadeInimigo` envolve um `Inimigo` (que será gerado e lutará contra você). `EntidadeItem` contém um item que você pode apanhar (entra no inventário). `EntidadeEscada` é o portal para o próximo andar.

Cada uma responde de forma diferente ao método `aoTocada()`: um inimigo não faz nada especial (combate é tratado separadamente), um item é adici-

onado ao seu inventário e marcado para remoção, uma escada desencadeia a transição de andar.

```
// lib/entidade_inimigo.dart

class EntidadeInimigo extends Entidade {
  final Inimigo inimigo;

  EntidadeInimigo({
    required int x,
    required int y,
    required this.inimigo,
  }) : super(
    x: x,
    y: y,
    simbolo: inimigo.simbolo,
    nome: inimigo.nome,
  );

  @override
  bool aoTocada(Entidade visitante) {
    return false; // Combate tratado separadamente
  }
}

// lib/entidade_item.dart

class EntidadeItem extends Entidade {
  final Item item;

  EntidadeItem({
    required int x,
    required int y,
    required this.item,
  }) : super(
    x: x,
```

```
      y: y,
      simbolo: '!',
      nome: item.nome,
    );

    @override
    bool aoTocada(Entidade visitante) {
      if (visitante is! Jogador) return false;
      visitante.inventario.add(item);
      return true; // Remover do mapa
    }
}

// lib/entidade_escada.dart

class EntidadeEscada extends Entidade {
    final int andarAtual;

    EntidadeEscada({
      required int x,
      required int y,
      required this.andarAtual,
    }) : super(
      x: x,
      y: y,
      simbolo: '>',
      nome: 'Escada Descendente',
    );

    @override
    bool aoTocada(Entidade visitante) {
      return false; // Descida tratada separadamente
    }
}
```

> **Importante:** A classe `Jogador` deve ter um campo `inventario` declarado como `List<Item> inventario = [];` para permitir que `EntidadeItem.aoTocada()` adicione itens coletados. Este campo deve ser adicionado à classe `Jogador` conforme demonstrado nos exemplos de integração neste capítulo.

Parte 3: Spawning de Entidades

"Spawn" significa "gerar" ou "aparecer". Depois que a masmorra é gerada, você precisa preenchê-la com inimigos, itens e escada. A classe `GeradorEntidades` faz exatamente isto: calcula quantos inimigos aparecem (escalado por andar), coloca itens em posições válidas (piso sólido, não parede), e garante sempre uma escada para descida.

Note que cada entidade precisa de uma posição única (não sobrepostas). Usamos um set `posicoesOcupadas` para rastrear onde já colocamos coisas. Se não encontrar espaço após 50 tentativas aleatórias, desistimos (está ok, às vezes um item não consegue aparecer em um andar apertado).

```
// lib/gerador_entidades.dart

class GeradorEntidades {
  final MapaMasmorra mapa;
  final int andarAtual;
  final Random random;
  final Set<Point<int>> posicoesOcupadas = {};

  GeradorEntidades({
    required this.mapa,
    required this.andarAtual,
    required this.random,
  });

  List<Entidade> spawn() {
    final entidades = <Entidade>[];
    posicoesOcupadas.clear();
```

```
    entidades.addAll(_spawnInimigos());
    entidades.addAll(_spawnItens());
    entidades.addAll(_spawnEscada());

    return entidades;
  }

  List<Entidade> _spawnInimigos() {
    final inimigos = <Entidade>[];
    final quantidade = 2 + (andarAtual ~/ 2) + random.nextInt(2);

    for (int i = 0; i < quantidade; i++) {
      final pos = _encontrarPosicaoValida();
      if (pos != null) {
        final tipo = _escolherTipoInimigo();
        inimigos.add(EntidadeInimigo(
          x: pos.x,
          y: pos.y,
          inimigo: _criarInimigo(tipo),
        ));
        posicoesOcupadas.add(pos);
      }
    }

    return inimigos;
  }

  List<Entidade> _spawnItens() {
    final itens = <Entidade>[];
    final quantidade = 2 + random.nextInt(3);

    for (int i = 0; i < quantidade; i++) {
      final pos = _encontrarPosicaoValida();
      if (pos != null) {
        itens.add(EntidadeItem(
          x: pos.x,
```

```
          y: pos.y,
          item: Item(
            nome: ['Ouro', 'Poção', 'Gema'][random.nextInt(3)],
          ),
        ));
        posicoesOcupadas.add(pos);
      }
    }

    return itens;
  }

  List<Entidade> _spawnEscada() {
    final pos = _encontrarPosicaoValida();
    if (pos != null) {
      return [
        EntidadeEscada(
          x: pos.x,
          y: pos.y,
          andarAtual: andarAtual,
        ),
      ];
    }
    return [];
  }

  Point<int>? _encontrarPosicaoValida() {
    for (int tentativa = 0; tentativa < 50; tentativa++) {
      final x = random.nextInt(mapa.largura);
      final y = random.nextInt(mapa.altura);
      final pos = Point(x, y);

      if (mapa.ehPassavel(x, y) && !posicoesOcupadas.contains(pos)) {
        return pos;
      }
    }
```

```
    return null;
  }

  String _escolherTipoInimigo() {
    final tipos = ['Zumbi', 'Lobo', 'Orc'];
    return tipos[random.nextInt(tipos.length)];
  }

  Inimigo _criarInimigo(String tipo) {
    return switch (tipo) {
      'Zumbi' => Inimigo(nome: 'Zumbi', hpMax: 20, simbolo: 'Z'),
      'Lobo' => Inimigo(nome: 'Lobo', hpMax: 40, simbolo: 'L'),
      'Orc' => Inimigo(nome: 'Orc', hpMax: 60, simbolo: 'O'),
      _ => Inimigo(nome: 'Monstro', hpMax: 25, simbolo: '?'),
    };
  }
}
```

Spawning Inteligente com Distância

Em vez de colocar entidades completamente ao acaso, você pode ser mais inteligente: inimigos longe da entrada, itens distribuídos por salas, escada no fim. Use distância Manhattan:

```
// lib/gerador_entidades_avancado.dart

class GeradorEntidadesAvancado {
  final MapaMasmorra mapa;
  final List<Sala> salas;
  final int andarAtual;
  final Random random;

  GeradorEntidadesAvancado({
    required this.mapa,
    required this.salas,
```

```
    required this.andarAtual,
    required this.random,
  });

  List<Entidade> spawnInteligente() {
    final entidades = <Entidade>[];

    // Entrada assume-se no centro
    final posEntrada = Point(mapa.largura ~/ 2, mapa.altura ~/ 2);

    // Inimigos: longe da entrada (min 15 tiles)
    entidades.addAll(_spawnInimigosLonge(posEntrada));

    // Itens: distribuídos por diferentes salas
    entidades.addAll(_spawnItensEmSalas());

    // Escada: bem longe, usualmente no canto oposto
    entidades.add(_spawnEscadaLonge(posEntrada));

    return entidades;
  }

  List<Entidade> _spawnInimigosLonge(Point<int> entrada) {
    final inimigos = <Entidade>[];
    final quantidade = 2 + (andarAtual ~/ 2) + random.nextInt(2);

    for (int i = 0; i < quantidade; i++) {
      Point<int>? pos;

      for (int tentativa = 0; tentativa < 30; tentativa++) {
        final x = random.nextInt(mapa.largura);
        final y = random.nextInt(mapa.altura);
        final cand = Point(x, y);

        if (mapa.ehPassavel(x, y)) {
          final dist = _distanciaManhattan(entrada, cand);
```

```
          if (dist > 15) {
            pos = cand;
            break;
          }
        }
      }

      if (pos != null) {
        final tipo = _escolherTipoInimigo();
        inimigos.add(EntidadeInimigo(
          x: pos.x,
          y: pos.y,
          inimigo: _criarInimigo(tipo),
        ));
      }
    }

    return inimigos;
  }

  List<Entidade> _spawnItensEmSalas() {
    final itens = <Entidade>[];

    for (int i = 0; i < 2 + random.nextInt(2); i++) {
      if (salas.isEmpty) break;

      final sala = salas[random.nextInt(salas.length)];
      final x = sala.x + 1 +
          random.nextInt((sala.largura - 2).clamp(1, 100));
      final y = sala.y + 1 +
          random.nextInt((sala.altura - 2).clamp(1, 100));

      if (mapa.ehPassavel(x, y)) {
        itens.add(EntidadeItem(
          x: x,
          y: y,
```

```
          item: Item(
            nome: ['Ouro', 'Poção', 'Gema'][random.nextInt(3)],
          ),
        ));
      }
    }

    return itens;
  }

  EntidadeEscada _spawnEscadaLonge(Point<int> entrada) {
    Point<int>? melhorPos;
    double maiorDist = 0;

    for (int tentativa = 0; tentativa < 100; tentativa++) {
      final x = random.nextInt(mapa.largura);
      final y = random.nextInt(mapa.altura);

      if (mapa.ehPassavel(x, y)) {
        final dist = _distanciaManhattan(
          entrada,
          Point(x, y),
        ).toDouble();
        if (dist > maiorDist) {
          maiorDist = dist;
          melhorPos = Point(x, y);
        }
      }
    }

    melhorPos ??= Point(mapa.largura - 5, mapa.altura - 5);

    return EntidadeEscada(
      x: melhorPos.x,
      y: melhorPos.y,
      andarAtual: andarAtual,
```

```
    );
  }

  int _distanciaManhattan(Point<int> a, Point<int> b) {
    return (a.x - b.x).abs() + (a.y - b.y).abs();
  }

  String _escolherTipoInimigo() {
    final tipos = ['Zumbi', 'Lobo', 'Orc'];
    return tipos[random.nextInt(tipos.length)];
  }

  Inimigo _criarInimigo(String tipo) {
    return switch (tipo) {
      'Zumbi' => Inimigo(nome: 'Zumbi', hpMax: 20, simbolo: 'Z'),
      'Lobo' => Inimigo(nome: 'Lobo', hpMax: 40, simbolo: 'L'),
      'Orc' => Inimigo(nome: 'Orc', hpMax: 60, simbolo: 'O'),
      _ => Inimigo(nome: 'Monstro', hpMax: 25, simbolo: '?'),
    };
  }
}
```

Parte 4: Detecção de Colisão e Interação

Quando o jogador tenta se mover para uma posição, você precisa checar se há uma entidade lá. Se houver, trata a **colisão**. A interface define o contrato; subclasses definem comportamentos específicos:

```
// lib/detector_colisao.dart
// (enum `TipoColisao` e classe `ResultadoMovimento` ficam em
//  ficheiros próprios no repositório)

class DetectorColisao {
  /// Tenta mover o jogador para (novoX, novoY)
  /// Retorna ResultadoMovimento com tipo de colisão (se houver)
```

```
  ResultadoMovimento verificarMovimento(
    int novoX,
    int novoY,
    Jogador jogador,
    AndarMasmorra andar,
  ) {
    // 1. Checa se é passável (não é parede)
    if (!andar.mapa.ehPassavel(novoX, novoY)) {
      return ResultadoMovimento.colisaoParede();
    }

    // 2. Checa se há entidade naquela posição
    final entidade = andar.encontrarEntidadeEm(novoX, novoY);

    if (entidade == null) {
      // Sem colisão: movimento livre
      return ResultadoMovimento.sucesso(novoX, novoY);
    }

    // 3. Processa tipo de colisão
    return switch (entidade) {
      EntidadeInimigo enemyEnt =>
        ResultadoMovimento.colisaoInimigo(enemyEnt.inimigo),
      EntidadeItem itemEnt =>
        ResultadoMovimento.colisaoItem(itemEnt.item, entidade),
      EntidadeEscada escadaEnt =>
        ResultadoMovimento.colisaoEscada(escadaEnt),
      _ => ResultadoMovimento.colisaoDesconhecida(),
    };
  }
}

enum TipoColisao {
  nenhuma,
  parede,
  inimigo,
```

```
  item,
  escada,
  outro,
}

class ResultadoMovimento {
  final bool podeMovimentar;
  final TipoColisao tipo;
  final int? novoX;
  final int? novoY;
  final dynamic alvo; // Inimigo, Item, Escada, etc

  ResultadoMovimento._({
    required this.podeMovimentar,
    required this.tipo,
    this.novoX,
    this.novoY,
    this.alvo,
  });

  factory ResultadoMovimento.sucesso(int x, int y) =>
      ResultadoMovimento._(
        podeMovimentar: true,
        tipo: TipoColisao.nenhuma,
        novoX: x,
        novoY: y,
      );

  factory ResultadoMovimento.colisaoParede() => ResultadoMovimento._(
    podeMovimentar: false,
    tipo: TipoColisao.parede,
  );

  factory ResultadoMovimento.colisaoInimigo(Inimigo inimigo) =>
    ResultadoMovimento._(
      podeMovimentar: false,
```

```
      tipo: TipoColisao.inimigo,
      alvo: inimigo,
    );

  factory ResultadoMovimento.colisaoItem(
    Item item,
    Entidade entidade,
  ) =>
      ResultadoMovimento._(
        podeMovimentar: false,
        tipo: TipoColisao.item,
        alvo: entidade,
      );

  factory ResultadoMovimento.colisaoEscada(EntidadeEscada escada) =>
    ResultadoMovimento._(
      podeMovimentar: false,
      tipo: TipoColisao.escada,
      alvo: escada,
    );

  factory ResultadoMovimento.colisaoDesconhecida() =>
      ResultadoMovimento._(
        podeMovimentar: false,
        tipo: TipoColisao.outro,
      );
}
```

Processador de Interações

Depois de detectar colisão, você precisa processar a interação específica:

```
// lib/interacao_processador.dart

class ProcessadorInteracao {
```

```
  void processarColisao(
    ResultadoMovimento resultado,
    Jogador jogador,
    AndarMasmorra andar,
    void Function(String) logCallback,
  ) {
    switch (resultado.tipo) {
      case TipoColisao.nenhuma:
        // Sem ação
        break;

      case TipoColisao.parede:
        logCallback('Você bateu numa parede!');
        break;

      case TipoColisao.inimigo:
        final inimigo = resultado.alvo as Inimigo;
        logCallback('Você encontrou um ${inimigo.nome}! Luta!');
        // Combate será tratado separadamente
        break;

      case TipoColisao.item:
        final entidade = resultado.alvo as Entidade;
        final foiColetado = entidade.aoTocada(jogador);
        if (foiColetado) {
          logCallback('Você coletou ${entidade.nome}!');
          andar.removerEntidade(entidade);
        }
        break;

      case TipoColisao.escada:
        logCallback('Você encontrou a escada! Digite "d" para
↪  descer.');
        break;

      case TipoColisao.outro:
```

```
        logCallback('Algo estranho aqui...');
        break;
    }
  }
}
```

Parte 5: AndarMasmorra — Encapsulando Tudo

Um andar é mais que um mapa: é o mapa MAIS as entidades nele. A classe `AndarMasmorra` agrupa mapa, lista de entidades e número do andar. Oferece serviços úteis: encontrar uma entidade em (x, y), remover uma entidade (quando morre ou é coletada), e filtrar entidades por tipo.

Este é um padrão importante: composição. Uma classe não herda, mas contém outras. Um `AndarMasmorra` não é um `Mapa`, mas tem um `Mapa`. Isto é mais flexível que herança.

```
// lib/andar_masmorra.dart

class AndarMasmorra {
  final int numero;
  final MapaMasmorra mapa;
  final List<Entidade> entidades;

  AndarMasmorra({
    required this.numero,
    required this.mapa,
    required this.entidades,
  });

  Entidade? encontrarEntidadeEm(int x, int y) {
    try {
      return entidades.firstWhere((e) => e.x == x && e.y == y);
    } catch (e) {
      return null;
    }
```

```
  }

  void removerEntidade(Entidade entidade) {
    entidades.remove(entidade);
  }

  /// Retorna todas as entidades de um tipo específico
  List<T> entidadesDoTipo<T extends Entidade>() {
    return entidades.whereType<T>().toList();
  }

  /// Conta quantos inimigos ainda existem neste andar
  int contarInimigos() {
    return entidadesDoTipo<EntidadeInimigo>().length;
  }
}
```

Pergaminho do Capítulo

- Classe abstrata `Entidade` como contrato com posição (x, y), símbolo visual, nome e método abstrato `aoTocada()`
- Três subclasses concretas: `EntidadeInimigo` (combatente), `EntidadeItem` (colecionável), `EntidadeEscada` (descida)
- Padrão de composição com `AndarMasmorra` encapsulando mapa + entidades + número
- Gerador básico `GeradorEntidades` que popula masmorras com inimigos escalados, itens valiosos e escada
- Versão avançada `GeradorEntidadesAvancado` usando distância Manhattan para posicionamento inteligente
- Detector de colisão `DetectorColisao` retornando `ResultadoMovimento` com tipo específico
- Processador de interações `ProcessadorInteracao` reagindo diferentemente por tipo de colisão
- Métodos utilitários para encontrar, remover e filtrar entidades por tipo

Dica Profissional

Em engines profissionais como libGDX ou Godot, entidades são frequentemente atores/nós que herdam de uma classe mãe cena e possuem componentes (física, renderização, IA). O padrão entity-component-system (ECS) é ainda mais escalável; uma entidade é apenas um ID, dados são armazenados em tabelas. Para roguelikes, composição simples (como feito aqui) é suficiente até milhares de entidades. Sempre marque entidades como "persistentes" em andar anterior vs "geradas" em novo andar. Alguns roguelikes mantêm andar anterior "vivo" para você voltar. Considere usar objeto Pool para reusar instâncias de entidades em vez de criar/destruir constantemente.

Desafios da Masmorra

Desafios Básicos

Desafio 20.1. Armadilha (Entidade customizada). Crie `EntidadeArmadilha`: ao ser tocada, aplica dano ao jogador (5 HP) e dispara mensagem. O símbolo é `^`. Retorna `false` de `aoTocada()`, permanecendo no mapa. Adicione com 20% de chance em cada sala. Dica: verifique `if (visitante is Jogador)`, depois aplique dano via `visitante.sofrerDano(5)`.

Desafio 20.2. Tipos de Item. Estenda Item com propriedades: crie enum `TipoItem` com valores OURO, POCAO_VIDA, POCAO_MANA, GEMA, CHAVE. Cada tipo tem efeito único ao ser coletado. POCAO_VIDA restaura 25 HP, GEMA aumenta ouro, CHAVE abre portas. Implemente efeito em `aoTocada()`.

Desafios Avançados

Desafio 20.3. Inimigos por dificuldade. Estenda `GeradorEntidades` para aceitar `int andar`. Conforme o andar aumenta, inimigos ficam mais fortes (HP += andar * 2), mais raros e variados. Andar 5+: aparece um Orc. Andar 10+: Dragão. Use `random.nextInt(andar)` para verificar se spawna inimigo raro.

Desafio 20.4. Colisão com eventos. Integre entidades com movimento: ao jogador tentar se mover, chame `mapa.entidadeEm(x, y)`. Se houver, chame `aoTocada(jogador)`. Implemente um log visual no HUD mostrando últimas ações: "Coletou Ouro", "Levou dano de Armadilha", etc. Use `List<String> logAcoes` para rastrear.

Desafio 20.5. Spawn inteligente (Distribuição). Inimigos nunca aparecem a menos de 20 tiles da entrada (distância Manhattan). Itens são distribuídos em salas diferentes. Escadas ficam no fundo (distante). Passe as salas ao gerador, determine sala aleatória, *spawn* dentro dela.

Boss Final 20.6. IA de Inimigos (Movimentação). Adicione método `moveIA(Pos jogadorPos)` em Inimigo que retorna nova posição. Se jogador está no *FOV*, persegue (distância < 10 tiles). Senão, anda aleatoriamente. Implemente no turno inimigo: primeiro inimigos se movem, depois jogador age. Crie um `InimigoPerseguidor` que tenta se aproximar do jogador.

Próximo Capítulo

No Capítulo 21, vamos juntar todos os sistemas — mapa, *FOV*, entidades, combate — num *dungeon crawl* funcional. O `ExploradorMasmorra` orquestrará o loop completo: explorar, lutar, coletar, descer escadas e enfrentar andares cada vez mais perigosos.

Capítulo 21 - Dungeon Crawl: Juntando Tudo

Chegou a hora de deixar de construir peças soltas e montar a máquina completa. O jogador entra na masmorra no primeiro andar, explora em tempo real, combate inimigos quando os encontra, coleta ouro e armas, desce quando encontra a escada, e repete em andares cada vez mais profundos, cada vez mais perigosos. Este capítulo é o pico da Parte III: não é apenas um sistema isolado, é um jogo roguelike completo, jogável do início ao fim.

O Que Vamos Aprender

Neste capítulo você vai:

- Criar a classe `ExploradorMasmorra` - o orquestrador supremo
- Implementar um loop de jogo completo: input → update → render
- Gerenciar múltiplos andares com progressão dinâmica
- Integrar combate, colisão, *FOV* e renderização num fluxo coeso
- Rastrear estatísticas: turnos, inimigos mortos, ouro coletado, andares explorados
- Implementar condições de vitória (atingir andar 5) e derrota (morte do jogador)
- Criar uma tela de game over com resumo de estatísticas
- Demonstrar output completo do jogo funcionando

Ao final, você terá um *roguelike* dungeon totalmente funcional.

Parte 1: Conceitualizando o Fluxo

Antes de código, visualize o fluxo completo. Um jogo *roguelike* não é desordenado; tem uma estrutura clara: inicializa, entra no loop, processa, renderiza, e verifica condições de vitória/derrota. Este diagrama mostra cada etapa e o que fazer em cada uma.

Até agora você construiu blocos individuais: gerador de mapas (Cap 12), renderização (Cap 15), *FOV* (Cap 19), combate (Cap 18), entidades (Cap 20). Agora junta tudo num orquestrador central que coordena cada peça. A

classe `ExploradorMasmorra` é esse coração: ela mantém o estado completo do jogo (jogador, mapa, entidades, turnos), lê input, atualiza lógica e renderiza a tela. Sem essa orquestração, você teria partes desconexas. Com ela, temos um jogo coerente.

Fluxo do jogo: inicialização, loop principal e condições de saída. A fonte editável do diagrama está em `assets/diagrams/capitulo-021-fluxo-jogo.mmd`; o PNG é gerado em `./scripts/build.sh` com Node.js/npx (`@mermaid-js/mermaid-cli`).

Parte 2: Classe ExploradorMasmorra - Orquestrador

A classe `ExploradorMasmorra` é o maestro que coordena tudo. Ela mantém o estado do jogo: quem é o jogador, qual é o andar atual, quantos turnos passaram, se o jogo acabou. Oferece métodos principais: `gerarAndar()` cria um mapa novo (integra Capítulo 12), `renderizarFrame()` desenha na tela respeitando *FOV* (integra Capítulos 15 e 19), `processarComando()` lê input do jogador e reage (lógica de movimento e colisão), e `executar()` é o loop infinito que mantém o jogo vivo.

Esta é a orquestração completa: tudo passa por aqui, desde a inicialização até a vitória ou derrota. O padrão de design aqui é **Facade**: um único ponto de entrada que esconde a complexidade de múltiplos subsistemas trabalhando em harmonia.

Por Que Orquestração Centralizada?

Você poderia ter cada sistema (renderização, input, colisão) rodando independentemente. Mas isso criaria caos: quem decide quando renderizar? Quem processa input? Como sincronizam? A resposta é simples: um orquestrador central. Ele mantém o controle, garante que eventos acontecem na ordem correta (sempre render → input → update → verify), e evita *race conditions* ou estados inconsistentes. Em jogos maiores, isso evoluiria para um engine de eventos ou máquina de estados, mas o princípio é o mesmo: coordenação central cria previsibilidade.

```
// lib/explorador_masmorra.dart

class ExploradorMasmorra {
```

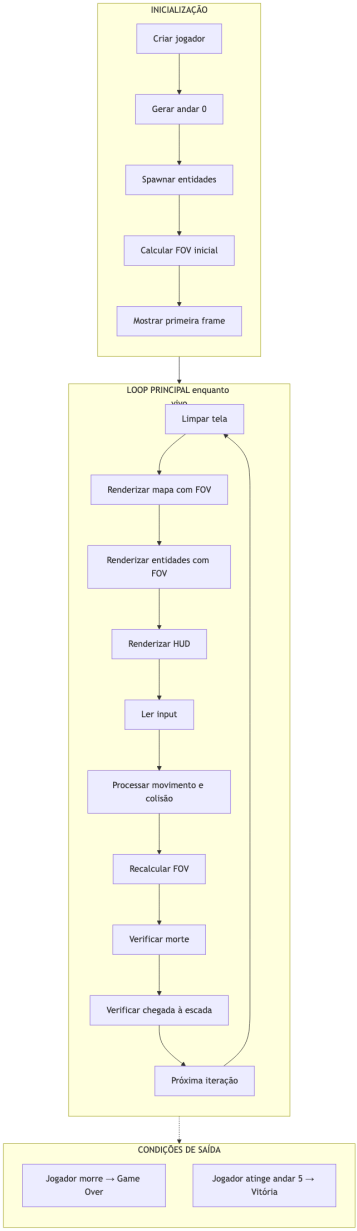


Figura 1: Fluxo do jogo: inicialização, loop principal e condições de saída

```
  final Jogador jogador;
  late AndarMasmorra andarAtual;
  late TelaAscii tela;

  final int larguraMapa;
  final int alturaMapa;
  final int andarFinal;

  int andarNumero = 0;
  int turno = 0;
  bool emJogo = true;
  bool vitoria = false;

  int totalInimigosDefeitos = 0;
  int maiorAndarAlcancado = 0;

  ExploradorMasmorra({
    required this.jogador,
    this.larguraMapa = 60,
    this.alturaMapa = 20,
    this.andarFinal = 3,
  }) {
    tela = TelaAscii(largura: larguraMapa, altura: alturaMapa + 5);
  }

  void gerarAndar() {
    // 1. Gerar novo mapa proceduralmente (Cap 12)
    final mapa = MapaMasmorra.gerar(
      largura: larguraMapa,
      altura: alturaMapa,
    );

    // 2. Spawnar entidades (inimigos, itens) progressivas (Cap 20)
    final spawner = GeradorEntidades(
      mapa: mapa,
      andarAtual: andarNumero, // ← Dificuldade aumenta a cada andar
```

```
    );

    andarAtual = AndarMasmorra(
      numero: andarNumero,
      mapa: mapa,
      entidades: spawner.spawn(),
    );

  // 3. Encontrar posição inicial passável (não começa dentro de
  ↪  parede)

    bool encontrou = false;
    for (int y = 1; y < alturaMapa - 1 && !encontrou; y++) {
      for (int x = 1; x < larguraMapa - 1 && !encontrou; x++) {
        if (mapa.ehPassavel(x, y)) {
          jogador.x = x;
          jogador.y = y;
          encontrou = true;
        }
      }
    }

    // 4. Calcular *FOV* inicial (Cap 19)
    mapa.fov.calcularShadowcast(
      Point(jogador.x, jogador.y),
      8,
      mapa,
    );

    maiorAndarAlcancado = andarNumero;
  }

  void renderizarFrame() {
    tela.limpar();

    // Camada 1: Mapa respeitando *FOV* (Cap 15 + Cap 19)
```

```
    andarAtual.mapa.renderizarNaTela(tela);

    // Camada 2: Entidades (inimigos, itens) apenas se visíveis
    for (final entidade in andarAtual.entidades) {
      if (andarAtual.mapa.fov.estaVisivel(entidade.x, entidade.y)) {
        tela.desenharChar(entidade.x, entidade.y, entidade.simbolo);
      }
    }

    // Camada 3: Jogador sempre visível (está sobre tudo)
    jogador.renderizarNaTela(tela);

    _renderizarHUD();
    tela.renderizar(); // ← Enviar buffer para terminal
  }

  void _renderizarHUD() {
    final hudY = alturaMapa + 1;
    final hpBar = _construirBarraHP();

    tela.desenharString(0, hudY, '=' * larguraMapa);
    tela.desenharString(
      0,
      hudY + 1,
      'Andar: $andarNumero | Turno: $turno | $hpBar '
      '${jogador.hpAtual}/${jogador.hpMax}',
    );
    tela.desenharString(
      0,
      hudY + 2,
      'Ouro: ${jogador.ouro} | Inimigos: ${totalInimigosDefeitos}',
    );
    tela.desenharString(0, hudY + 3,
        '[W]cima [A]esq [S]baixo [D]dir [I]nv [Q]uit');
  }
```

```
String _construirBarraHP() {
  const blocos = 5;
  final cheios = (jogador.hpAtual / jogador.hpMax * blocos).toInt();
  final vazios = blocos - cheios;
  return '▓' * cheios + '░' * vazios;
}

void processarComando(String comando) {
  switch (comando.toLowerCase()) {
    case 'w' || 'a' || 's' || 'd':
      _processarMovimento(comando);
    case 'i':
      _mostrarInventario();
    case 'q':
      emJogo = false;
    default:
      // Ignorar
  }
}

void _processarMovimento(String direcao) {
  // Calcular próxima posição
  int novoX = jogador.x;
  int novoY = jogador.y;

  switch (direcao.toLowerCase()) {
    case 'w': novoY--; // ← Cima
    case 's': novoY++; // ← Baixo
    case 'a': novoX--; // ← Esquerda
    case 'd': novoX++; // ← Direita
  }

  // Verificar colisão com parede
  if (!andarAtual.mapa.ehPassavel(novoX, novoY)) {
    return; // ← Não se move, turno não avança
  }
```

```
// Verificar colisão com entidade (inimigo, item, escada)
final entidade = andarAtual.encontrarEntidadeEm(novoX, novoY);
if (entidade != null) {
  if (entidade is EntidadeInimigo) {
    // COMBATE: Ataque direto
    final vitoria = _executarCombate(entidade.inimigo);
    if (!vitoria) {
      // ← Morte (jogo acabará no loop principal)
      jogador.hpAtual = 0;
      return;
    }
    andarAtual.removerEntidade(entidade);
    totalInimigosDefeitos++;
    jogador.ouro += 25; // ← Recompensa
  } else if (entidade is EntidadeItem) {
    // COLETA: Item é consumido/coletado
    entidade.aoTocada(jogador);
    andarAtual.removerEntidade(entidade);
  } else if (entidade is EntidadeEscada) {
    // DESCIDA: Próximo andar
    andarNumero++;
    if (andarNumero >= andarFinal) {
      vitoria = true; // ← Vitória!
      emJogo = false;
    } else {
      gerarAndar(); // ← Gerar próximo andar com mais dificuldade
    }
    return;
  }
}

// Se chegou aqui, movimento é válido
jogador.x = novoX;
jogador.y = novoY;
turno++;
```

```
  // Recalcular *FOV* para nova posição (Cap 19)
  andarAtual.mapa.fov.calcularShadowcast(
    Point(jogador.x, jogador.y),
    8,
    andarAtual.mapa,
  );
}

bool _executarCombate(Inimigo inimigo) {
  // Simplificado: jogador sempre ganha
  return true;
}

void _mostrarInventario() {
  // Implementar depois
}

void executar() {
  print('=== MASMORRA ASCII: Dungeon Crawl ===\n');
  gerarAndar(); // ← Inicialização: criar andar 0

  // LOOP PRINCIPAL: Render → Input → Update
  while (emJogo && jogador.hpAtual > 0) {
    renderizarFrame(); // ← Desenhar tela atual

    stdout.write('> ');
    final entrada = stdin.readLineSync() ?? ''; // ← Ler input
    // ← Processar comando (move, inventário, etc)
    processarComando(entrada);

    // Após comando, colisão e *FOV* já foram processados.
  }

  _mostrarGameOver(); // ← Condição de saída atingida
}
```

```
  void _mostrarGameOver() {
    final largura = 40;
    String centralizar(String texto) {
      final espacos = (largura - texto.length) ~/ 2;
      return ' ' * espacos + texto;
    }
    String alinhar(String rotulo, dynamic valor) {
      final conteudo = '$rotulo $valor';
      return conteudo.padRight(largura);
    }

    print('\n╔${'=' * largura}╗');
    if (vitoria) {
      print('║${centralizar('ESCAPOU DA MASMORRA!')}║');
      print('║${centralizar('PARABÉNS!')}║');
    } else {
      print('║${centralizar('GAME OVER')}║');
      print('║${centralizar('Caiu na masmorra...')}║');
    }
    print('╠${'=' * largura}╣');
    print('║${alinhar(' Estatísticas:', '')}║');
    print('║${alinhar(' Turnos:', turno)}║');
    print('║${alinhar(' Maior Andar:', maiorAndarAlcancado)}║');
    var inimigosText = alinhar(
      ' Inimigos Derrotados:',
      totalInimigosDefeitos,
    );
    print('║$inimigosText║');
    print('║${alinhar(' Ouro Total:', jogador.ouro)}║');
    print('╚${'=' * largura}╝\n');
  }
}
```

Máquina de Estados do Jogo

Um jogo bem estruturado tem estados claros e bem definidos. Você não está sempre explorando; às vezes está em combate tático, abrindo inventário, em transição entre andares, ou vendo a tela de morte. Uma máquina de estados formaliza isso: cada estado tem comportamentos permitidos e transições bem definidas. Por exemplo, em exploração você pode mover-se; em combate você só pode atacar ou defender; em inventário você só pode equipar itens. Sem estados, você teria if/else aninhados no movimento checando "estou em combate?", "estou em inventário?", gerando código acoplado e frágil.

Use um enum para organizar estes estados distintos, e um gerenciador para as transições:

```
// lib/game_state.dart

enum EstadoJogo {
  exploracao,      // ← Andando, vendo o mapa
  combate,         // ← Em luta com inimigo
  inventario,      // ← Menu de itens
  transicaoAndar,  // ← Descendo escada (efeito visual)
  gameOver,        // ← Morte (jogo acabou)
  vitoria,         // ← Venceu (jogo acabou)
}

class GerenciadorEstado {
  EstadoJogo estadoAtual = EstadoJogo.exploracao;

  void transicionar(EstadoJogo novoEstado) {
    print('Transição: ${estadoAtual.name} → ${novoEstado.name}');
    estadoAtual = novoEstado;
  }

  // Verificar transições válidas
  bool podeMovimentar() {
    // Só pode mover em exploração normal
    return estadoAtual == EstadoJogo.exploracao;
```

```
  }

  bool podeAbrirInventario() {
    // Pode abrir inventário enquanto explora ou já está no inventário
    return estadoAtual == EstadoJogo.exploracao ||
        estadoAtual == EstadoJogo.inventario;
  }

  bool estaVivo() {
    // Jogo ainda corre se não está em game over ou vitória
    return estadoAtual != EstadoJogo.gameOver &&
        estadoAtual != EstadoJogo.vitoria;
  }
}
```

Transição de Andares com Efeitos

Descer para um novo andar é mais que mudar o número do andar. É lidar com efeitos visuais de transição, *spawn* novo de entidades, dificuldade aumentando gradualmente. A progressão deve oferecer tensão crescente que faz o jogador sentir o peso de descer mais fundo.

Quando o jogador chega à escada, você entra em um estado de transição especial. Aqui você pode: 1. Parar a renderização normal (criar um efeito de "descendo...") 2. Atualizar dificuldade (mais inimigos, mais fortes) 3. Recuperar um pouco de HP (recompensa por sobreviver) 4. Voltar à exploração no novo andar

Isso torna cada descida um evento narrativo, não apenas um carregamento de nível:

```
// lib/transicao_andares.dart

class GerenciadorTransicao {
  void descerParaProximoAndar(
    ExploradorMasmorra explorador,
    GerenciadorEstado estado,
  ) {
```

```
    // 1. Efeito visual: "Você desce as escadas..."
    _mostrarTransicao(
      explorador.andarNumero,
      explorador.andarNumero + 1,
    );

    // 2. Atualizar estado
    explorador.andarNumero++;
    estado.transicionar(EstadoJogo.transicaoAndar);

    // 3. Gerar novo andar (com mais dificuldade)
    explorador.gerarAndar();

    // 4. Recuperar um pouco de HP (tensão + recompensa)
    explorador.jogador.hpAtual = (explorador.jogador.hpAtual + 15)
        .clamp(0, explorador.jogador.hpMax);

    // 5. Voltar à exploração
    estado.transicionar(EstadoJogo.exploracao);

    print('Você desceu para o andar ${explorador.andarNumero}');
  }

  void _mostrarTransicao(int andarAtual, int proximoAndar) {
    print('\n...');
    sleep(Duration(milliseconds: 300));
    print('Você desce as escadas...');
    sleep(Duration(milliseconds: 500));
    print('...');
    sleep(Duration(milliseconds: 300));
    print('Andar $proximoAndar alcançado!\n');
  }
}
```

Sistema de Condições de Vitória/Derrota

O jogo deve verificar continuamente se o jogador venceu ou perdeu. Uma abordagem limpa é centralizar essa lógica numa classe dedicada. Isso separa a responsabilidade: o orquestrador coordena, mas a verificação de condições vive num lugar bem definido. Quando o HP chega a 0 ou o jogador alcança o andar final, a classe informa ao orquestrador que o jogo acabou:

```
// lib/condicoes_jogo.dart

class VerificadorCondicoes {
  /// Verifica se o jogador morreu
  bool jogadorMorreu(Jogador jogador) {
    return jogador.hpAtual <= 0;
  }

  /// Verifica se o jogador venceu
  bool jogadorVenceu(int andarAtual, int andarFinal) {
    return andarAtual >= andarFinal;
  }

  /// Gera estatísticas finais
  EstatisticasJogo gerarEstatisticas(
    ExploradorMasmorra explorador,
  ) {
    return EstatisticasJogo(
      turnosJogados: explorador.turno,
      maiorAndarAlcancado: explorador.maiorAndarAlcancado,
      inimigosDefeitos: explorador.totalInimigosDefeitos,
      ouroColetado: explorador.totalOuroColetado,
      tempoJogo: DateTime.now(),
      jogadorVenceu: explorador.vitoria,
    );
  }
}

class EstatisticasJogo {
```

```
  final int turnosJogados;
  final int maiorAndarAlcancado;
  final int inimigosDefeitos;
  final int ouroColetado;
  final DateTime tempoJogo;
  final bool jogadorVenceu;

  EstatisticasJogo({
    required this.turnosJogados,
    required this.maiorAndarAlcancado,
    required this.inimigosDefeitos,
    required this.ouroColetado,
    required this.tempoJogo,
    required this.jogadorVenceu,
  });

  void imprimirResumo() {
    final resultado = jogadorVenceu ? 'VITÓRIA' : 'DERROTA';
    print('\n╔═══════════════════════════════════════╗');
    print('║       RESULTADO: $resultado         ║');
    print('╠═══════════════════════════════════════╣');
    print('║ Turnos: $turnosJogados');
    print('║ Maior Andar: $maiorAndarAlcancado');
    print('║ Inimigos Derrotados: $inimigosDefeitos');
    print('║ Ouro Total: $ouroColetado');
    print('║ Data/Hora: ${tempoJogo.toString()}');
    print('╚═══════════════════════════════════════╝\n');
  }
}
```

Exemplo Completo: Main

Quando você executa o jogo, tudo começa aqui. O arquivo `main.dart` é o ponto de entrada: cria um jogador, cria um explorador com os parâmetros desejados (tamanho do mapa, número final de andares), e chama `executar()`. A partir daí, o orquestrador assume o controle e não solta até você morrer ou

vencer. Este é um exemplo de padrão **Builder** simplificado: você constrói os objetos necessários e depois passa para um controlador central.

Note que você pode experimentar facilmente modificando os parâmetros aqui: aumentar `alturaMapa` para um mapa maior, aumentar `andarFinal` para uma progressão mais longa, etc. Toda a lógica do jogo roda independente de que tamanho o mapa tem ou quantos andares existem.

```
// main.dart

import 'dart:io';

void main() {
  // 1. Criar jogador com stats iniciais
  final jogador = Jogador(
    nome: 'Aventureiro',
    hpMax: 100,
    ouro: 0, // ← Começar pobre, enriquecer matando inimigos
  );

  // 2. Criar explorador (orquestrador) com configurações de jogo
  final explorador = ExploradorMasmorra(
    jogador: jogador,
    larguraMapa: 80,    // ← Largura de cada andar
    alturaMapa: 24,     // ← Altura de cada andar
    andarFinal: 5,      // ← Vencer ao atingir andar 5
  );

  // 3. Executar: inicia o loop infinito até morte ou vitória
  explorador.executar();
}
```

O Jogo Até Aqui - Saída Esperada

Quando você executa `dart main.dart` e joga por alguns turnos, a saída parece assim:

```
=== MASMORRA ASCII: Dungeon Crawl ===

####################
#.....··········.....#
#.@.G.....·····.....#
#.....·····.....#
###....###·····#####·
····#····#·····#····
····#····#·····#····
#####....####Z####···
#···.....#
#···.....E...#
#···.....>
####################

═══════════════════════════════════════
Andar: 0 | Turno: 15 | [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓] 75/100
Ouro: 50 | Inimigos: 1
[W]cima [A]esq [S]baixo [D]dir [I]nventário [Q]uit
> d
```

Explicação da saída: - `@` = Jogador - `G` = Goblin visível - `E` = Esqueleto visível - `Z` = Zumbi visível - `>` = Escada para próximo andar - `.` = Piso visível - `·` = Piso explorado (fora do *FOV*) - (espaço) = Nunca visto - `#` = Parede opaca

O HUD mostra: número do andar, turnos passados, barra de HP, ouro coletado, inimigos derrotados, e controles disponíveis.

Cada turno que você se move, o *FOV* recalcula (imperceptível para o jogador), e a névoa de guerra revela novos tiles enquanto você explora. Quando encontra um inimigo visível (dentro do *FOV*), pode atacar movendo-se para ele. Quando encontra a escada, desce para o próximo andar com mais dificuldade.

Apêndice: Códigos de Escape ANSI para Cores

Se você quiser adicionar cores ao jogo (Boss Final 21.5), aqui está o essencial sobre *ANSI escape codes*. Terminais modernos interpretam sequências especiais que controlam formatação, cores e posicionamento de cursor.

A sintaxe básica é: `\x1B[<código>m`, onde `\x1B` é o caractere ESC e `<código>` é o comando.

Cores de foreground (texto): - `\x1B[30m` = Preto - `\x1B[31m` = Vermelho (para inimigos) - `\x1B[32m` = Verde (para chão) - `\x1B[33m` = Amarelo (para ouro) - `\x1B[34m` = Azul (para escada) - `\x1B[37m` = Branco / `\x1B[90m` = Cinza (para paredes) - `\x1B[0m` = Reset (volta ao padrão)

Exemplo de uso:

```
String textoVerde = '\x1B[32m.\x1B[0m'; // Piso verde normal
String textoVermelho = '\x1B[31mG\x1B[0m'; // Goblin em vermelho
stdout.write(textoVermelho);
```

Integrar cores é simples: ao desenhar cada tile, imprima o código ANSI antes do caractere e um reset depois. O terminal cuida da formatação. Cuidado: nem todos os terminais suportam ANSI (especialmente Windows antigo), mas Windows 10+ e todos os terminais Unix suportam nativamente.

Integração com Capítulos Anteriores

Este capítulo é o pico de integração. Tudo que você aprendeu até aqui converge:

- **Capítulo 12** *(Gerador de Mapas)*: `MapaMasmorra.gerar()` cria cada andar proceduralmente
- **Capítulo 15** *(Grid e Renderização)*: `TelaAscii` e `renderizarNaTela()` desenham cada frame
- **Capítulo 18** *(Combate)*: `_executarCombate()` resolve encontros com inimigos
- **Capítulo 19** *(Campo de Visão)*: `fov.calcularShadowcast()` recalcula a cada movimento
- **Capítulo 20** *(Entidades)*: `GeradorEntidades` popula cada andar com criatura e itens

`ExploradorMasmorra` orquestra todos esses subsistemas num loop coeso. A máquina de estados (exploração → combate → transição de andares → game over) controla o fluxo. O resultado: um *roguelike* jogável do início ao fim.

Design Decision: Por Que Não Input Assíncrono?

Você pode estar pensando: por que bloquear em `stdin.readLineSync()`? Não seria melhor ler input assincronamente enquanto o jogo atualiza em paralelo? A resposta é **simplicidade vs. complexidade**. Em um jogo baseado em turnos (como este), bloquear em input é natural: o jogador faz uma ação, o jogo processa, o jogo renderiza. Não há necessidade de concorrência. Input assincronamente adicionaria channels, futures e race conditions sem benefício real. Em um jogo com tempo real (como um *action RPG* ou *FPS*), você precisaria de input não-bloqueante; mas em um *roguelike* turn-based, simplicidade vence.

Dica Profissional

Escalabilidade do loop de jogo em produção: o loop apresentado aqui é funcional, mas em um jogo real você pode encontrar gargalos. Considere separar completamente entrada (I/O bloqueante) da lógica de jogo usando filas de comando ou channels. Para jogos mais complexos, implemente deltaTime em vez de turnos síncronos, permitindo frames consistentes em diferentes máquinas. Salvar o estado completo (seed do mapa, posição jogador, turnos) permite implementar rewind ou replay, funcionalidades muito valorizadas em comunidades de speedrunning e streaming.

Desafios da Masmorra

Desafios Básicos

Desafio 21.1. Melhorar o HUD. Adicione mais informações na HUD: nível atual, XP para próximo nível, quantos inimigos você derrotou neste andar.

Desafio 21.2. Tela de Pausa. Implemente um comando `p` (pause) que para o jogo e mostra um menu: continuar, salvar, sair.

Desafios Avançados

Desafio 21.3. Animação de Movimento. Adicione um pequeno delay ao movimento (`Future.delayed()` ou `sleep()`) para que o jogador veja os passos acontecendo lentamente na tela.

Desafio 21.4. Log de Eventos. Adicione um `List<String> logEventos` que registra o que aconteceu: "Você matou Zumbi", "Pegou ouro", "Subiu de nível". Mostre os últimos 3-5 eventos na HUD.

Boss Final 21.5. Cores ANSI - Cores no Terminal. Volte à classe `TelaAscii` do Capítulo 16 e adicione suporte a cores ANSI (*ANSI escape codes*). Cada tipo de tile deve ter sua cor: verde para chão (`.`), cinza para paredes (`#`), vermelho para inimigos, amarelo para ouro, azul para escada (`>`). Códigos de escape ANSI são sequências especiais que o terminal interpreta como comandos de formatação. Por exemplo, `'\x1B[32m'` ativa verde, `'\x1B[31m'` ativa vermelho, e `'\x1B[0m'` reseta para padrão. Implemente um método `desenharCharComCor(int x, int y, String char, String corAnsi)` na `TelaAscii` e modifique `_renderizarMapaComFOV()` para usar cores de acordo com o tile. Execute o dungeon crawl inteiro e veja como cores melhoram drasticamente a clareza visual do mapa sem adicionar complexidade.

Dica do Mestre: Escalabilidade do loop de jogo em produção: o loop apresentado aqui é funcional, mas em um jogo real você pode encontrar gargalos. Considere separar completamente entrada (I/O bloqueante) da lógica de jogo. Use filas de comando ou channels para desacoplar a leitura do stdin do update lógico. Para jogos mais complexos, implemente um deltaTime (time stepping) em vez de turnos síncronos:

```
void executarComDeltaTime() {
  final stopwatch = Stopwatch()..start();
  const targetFrameTime = 1000 ~/ 60; // 60 FPS = ~16ms por frame

  while (emJogo && jogador.hpAtual > 0) {
    renderizarFrame();

    final deltaTimeMs = stopwatch.elapsedMilliseconds;
    if (deltaTimeMs < targetFrameTime) {
      sleep(Duration(milliseconds: targetFrameTime - deltaTimeMs));
    }
```

```
    stopwatch.reset();
  }
}
```

Além disso, salvar o estado completo (seed do mapa, posição jogador, turnos) permite implementar rewind ou replay de sessões, uma funcionalidade muito valorizada em comunidades speedrunning e streaming de roguelikes.

Pergaminho do Capítulo

- Fluxo completo do jogo: inicialização, loop principal (renderizar → input → update), verificação de vitória/derrota
- Classe `ExploradorMasmorra` orquestrando centralizado: geração, renderização, input, colisão, transição de andares
- Enum `EstadoJogo` organizando estados distintos: exploração, combate, inventário, transição, game over
- Máquina de estados com `GerenciadorEstado` permitindo lógica limpa e previsível
- Classe `GerenciadorTransicao` cuidando de descidas: efeitos visuais, geração com dificuldade crescente, recuperação de HP
- Classe `VerificadorCondicoes` centralizando lógica de vitória/derrota e estatísticas finais
- Loop principal integrado com stdin/stdout permitindo exploração interativa tempo-real
- Tela de game over com resumo de estatísticas: turnos, andar máximo, inimigos derrotados, ouro coletado

Próximo Capítulo

No Capítulo 22, vamos adicionar profundidade econômica ao jogo. Preços, *drops*, tabelas de saque e balanceamento de dificuldade por andar transformarão a masmorra de uma sequência de lutas numa experiência estratégica com decisões significativas.

PARTE IV

O MERCADOR E A ESCADA

Cada moeda conquistada é um estado que persiste. O mercador nega o descarte — aqui tudo tem valor, tudo se reusa, tudo se negocia. Conforme sobe de nível, os abismos se aprofundam ainda mais. E na boca da escada que desce, em câmaras cada vez mais obscuras, o boss aguarda — final implacável de toda progressão bem planejada.

Capítulo 22 - Economia: Preços, Drops e Balanceamento

Ouro cintila no chão. Mercadores surgem nas sombras oferecendo armas que você ainda não pode pagar. Cada inimigo derrotado deixa recompensas, e cada recompensa alimenta a escalada de poder que vai levar você até o chefe final. A economia da masmorra obedece regras que você mesmo vai definir: preços, drops, curvas de dificuldade, tabelas de progressão.

Nesta parte, o jogo ganha profundidade mecânica. XP e níveis transformam o `@` fraco do primeiro andar em um guerreiro capaz de enfrentar o Dragão. Múltiplos andares empilham desafios crescentes. E quando tudo estiver conectado, loja, combate, progressão, boss final, você terá um roguelike completo e jogável. O jogo que você imaginava no começo do livro agora existe.

> *A masmorra não é um labirinto vazio. Cada inimigo carrega ouro, armas, poções. Cada item tem um preço. O comerciante cobra mais caro em gemas raras e paga menos por sucata velha. Quanto mais fundo desce o herói, mais ricos e perigosos são os espólios. Este é o coração invisível do roguelike: regras simples de incentivo e progressão. A economia torna cada decisão relevante.*

O Que Vamos Aprender

Neste capítulo você vai:

- Entender por que a economia importa em roguelikes (como o sistema de Gil em Final Fantasy): tudo tem preço, e o balanceamento decide se o jogo é justo ou quebrado
- Modelar *loot tables* com pesos aleatórios (como os *drops* em Diablo): cada criatura tem uma tabela de probabilidades
- Criar a classe `EntradaSaque` e `Economia` para governar preços e recompensas
- Implementar cascatas de dificuldade: inimigos mais fortes em andares mais profundos
- Usar constantes de balanceamento para ajustes rápidos de design

- Simular corridas de teste para validar a curva de dificuldade
- Integrar *drops* no combate da Masmorra

Ao final, você terá um sistema econômico coerente que propicia progressão justa e recompensadora.

O Ciclo de Incentivos

Antes de código, pense no jogo como um jogador. Por que continuo descendo, tentando sobreviver?

1. Ganho ouro dos inimigos que derroto
2. Compro armas melhores com esse ouro na loja
3. Com armas melhores, consigo derrotar inimigos mais fortes
4. Inimigos mais fortes soltam mais ouro e itens raros
5. Volto à loja, compro armadura, deço mais fundo
6. No andar final, enfrento o chefe com tudo que consegui acumular

Este é o ciclo de progressão. É o coração psicológico do jogo. Se o primeiro inimigo soltar tanto ouro quanto o chefe, o jogo é entediante (sem senso de progressão). Se o primeiro inimigo soltar nada, é frustrante (sem recompensa). A economia bem balanceada é o pulso que mantém o jogo vivo.

Constantes de Balanceamento

Vamos definir constantes que governam toda a progressão. Isso é crucial porque ajustar um número muda tudo. É como ajustar a economia de Final Fantasy: um número muda, e o jogo inteiro se desbalanceia ou se equilibra perfeitamente.

Estas constantes vivem num único lugar (`EconomiaConstants`). Se você quer deixar o jogo mais fácil, muda um número e pronto. Sem procurar pelo código inteiro. Isto é design limpo.

```
// lib/economia_constants.dart

/// Constantes de balanceamento da economia
class EconomiaConstants {
  /// Dificuldade escalonada por andar
  static const int kBaseHPPorInimigo = 10;
```

```
  static const double kAumentoHPPorAndar = 0.2; // +20% HP por andar

  /// Recompensas em ouro
  static const int kOuroBasePorInimigo = 5;
  static const double kAumentoOuroPorAndar = 0.3; // +30% ouro por
  ↪  andar

  /// Preços base da loja
  static const int kPrecoEspadaFerro = 50;
  static const int kPrecoArmaduraCouro = 75;
  static const int kPrecoPocaoVida = 20;

  /// Margem do comerciante
  static const double kMargemVenda = 0.5; // Comerciante oferece 50%
}
```

Estas constantes são parâmetros de design. Ajuste um deles e observe como o jogo responde. A progressão fica lenta demais? Aumente `kOuroBasePorInimigo`. O primeiro inimigo é muito fácil e entediante? Aumente `kAumentoHPPorAndar`. Isto é iteração de design: números controlam a sensação inteira do jogo. Este é o coração invisível do balanceamento.

EntradaSaque: A Tabela de Drops

Cada tipo de inimigo tem uma tabela de *drops*, uma lista de itens que pode soltar com probabilidades. Por exemplo:

- Zumbi: 80% moeda de ouro, 15% adaga velha, 5% nada
- Lobo: 60% moeda de ouro, 30% espada de ferro, 10% poção de vida
- Orc: 50% moeda de ouro, 40% poção de vida, 10% nada

Loot tables são como os *drops* em Diablo: cada monstro tem uma probabilidade de soltar cada item. Modelamos isto com a classe `EntradaSaque`, que encapsula item, chance e quantidade mínima/máxima.

```
// lib/entrada_saque.dart
```

```dart
import 'dart:math';

/// Uma entrada na tabela de drops de um inimigo
/// Define qual item pode cair, com que
/// probabilidade, e em que quantidade
class EntradaSaque {
  final String itemId;
  final double chance;
  final int quantidadeMin;
  final int quantidadeMax;
  final String nomeItem;

  EntradaSaque({
    required this.itemId,
    required this.chance,
    required this.quantidadeMin,
    required this.quantidadeMax,
    required this.nomeItem,
  })  : assert(chance >= 0.0 && chance <= 1.0),
        assert(quantidadeMin >= 0 && quantidadeMax >= quantidadeMin);

  /// Calcula a quantidade a cair (entre min e max)
  int resolverQuantidade(Random random) {
    if (quantidadeMin == quantidadeMax) {
      return quantidadeMin;
    }
    return quantidadeMin +
        random.nextInt(quantidadeMax - quantidadeMin + 1);
  }

  @override
  String toString() =>
      '$nomeItem (${(chance * 100).toStringAsFixed(1)}%): '
      '$quantidadeMin–$quantidadeMax';
}
```

Classe Economia: O Governador

A classe `Economia` centraliza toda a lógica de economia. Tem dois serviços principais:

1. Determinar *drops* após combate (usa um Rolador para decisões probabilísticas)
2. Calcular preços de compra e venda

```
// lib/economia.dart

import 'dart:math';
import 'rolador.dart';
import 'entrada_saque.dart';
import 'economia_constants.dart';

/// Sistema de economia: drops, preços, balanceamento
class Economia {
  final Map<String, List<EntradaSaque>> tabelasDrops;
  final Rolador roller;

  Economia({
    required this.tabelasDrops,
    Rolador? roller,
  }) : roller = roller ?? Rolador();

  /// Resolve os drops de um inimigo derrotado
  List<String> resolverDrop(String nomeInimigo) {
    final drops = tabelasDrops[nomeInimigo];
    if (drops == null) {
      return ['ouro:${EconomiaConstants.kOuroBasePorInimigo}'];
    }

    final resultado = <String>[];

    for (final entry in drops) {
      if (roller.teste(entry.chance)) {
        final qtd = entry.resolverQuantidade(roller.random);
```

```
      resultado.add('${entry.itemId}:$qtd');
    }
  }

  if (resultado.isEmpty) {
    resultado.add('ouro:${EconomiaConstants.kOuroBasePorInimigo}');
  }

  return resultado;
}

/// Calcula o preço de compra (preço que você paga à loja)
int precoCompra(String itemId) {
  return switch (itemId) {
    'espada_ferro' => EconomiaConstants.kPrecoEspadaFerro,
    'espada_aco' =>
      (EconomiaConstants.kPrecoEspadaFerro * 1.5)
          .toInt(),
    'espada_mithril' =>
      (EconomiaConstants.kPrecoEspadaFerro * 3.0)
          .toInt(),
    'armadura_couro' => EconomiaConstants.kPrecoArmaduraCouro,
    'armadura_ferro' =>
      (EconomiaConstants.kPrecoArmaduraCouro * 1.5).toInt(),
    'pocao_vida' => EconomiaConstants.kPrecoPocaoVida,
    'pocao_restauracao' =>
      (EconomiaConstants.kPrecoPocaoVida * 2).toInt(),
    _ => 10,
  };
}

/// Calcula o preço de venda (preço que o comerciante oferece)
int precoVenda(String itemId) {
  final compra = precoCompra(itemId);
  return (compra * EconomiaConstants.kMargemVenda).toInt();
}
```

```
  /// Retorna dificuldade escalonada para um andar
  double getDificuldadeAndar(int numero) {
    return 1.0 + (numero * EconomiaConstants.kAumentoHPPorAndar);
  }

  /// Retorna recompensa escalonada para um andar
  int getOuroEscalonado(int numero) {
    final base = EconomiaConstants.kOuroBasePorInimigo.toDouble();
    final aum = EconomiaConstants.kAumentoOuroPorAndar;
    final multiplicador = 1.0 + (numero * aum);
    return (base * multiplicador).toInt();
  }
}
```

Criando as Tabelas de Drops

Agora populamos as tabelas com dados concretos para cada tipo de inimigo:

```
// lib/tabelas_drops.dart

import 'entrada_saque.dart';

/// Tabelas de drops padrão para todos os tipos de inimigo
class TabelasDrops {
  static Map<String, List<EntradaSaque>> criar() {
    return {
      'Zumbi': [
        EntradaSaque(
          itemId: 'ouro',
          chance: 1.0,
          quantidadeMin: 3,
          quantidadeMax: 8,
          nomeItem: 'Moedas de ouro',
        ),
```

```
      EntradaSaque(
        itemId: 'adaga_velha',
        chance: 0.15,
        quantidadeMin: 1,
        quantidadeMax: 1,
        nomeItem: 'Adaga velha',
      ),
    ],
    'Lobo': [
      EntradaSaque(
        itemId: 'ouro',
        chance: 0.9,
        quantidadeMin: 5,
        quantidadeMax: 15,
        nomeItem: 'Moedas de ouro',
      ),
      EntradaSaque(
        itemId: 'espada_ferro',
        chance: 0.25,
        quantidadeMin: 1,
        quantidadeMax: 1,
        nomeItem: 'Espada de ferro',
      ),
      EntradaSaque(
        itemId: 'pocao_vida',
        chance: 0.1,
        quantidadeMin: 1,
        quantidadeMax: 2,
        nomeItem: 'Poção de vida',
      ),
    ],
    'Orc': [
      EntradaSaque(
        itemId: 'ouro',
        chance: 0.95,
        quantidadeMin: 15,
```

```
          quantidadeMax: 30,
          nomeItem: 'Moedas de ouro',
        ),
        EntradaSaque(
          itemId: 'espada_aco',
          chance: 0.35,
          quantidadeMin: 1,
          quantidadeMax: 1,
          nomeItem: 'Espada de aço',
        ),
        EntradaSaque(
          itemId: 'armadura_ferro',
          chance: 0.2,
          quantidadeMin: 1,
          quantidadeMax: 1,
          nomeItem: 'Armadura de ferro',
        ),
      ],
    };
  }
}
```

Integrando Drops no Combate

Quando você derrota um inimigo, resolvemos o *drop*. Isto acontece no sistema de combate:

```
// Exemplo: em combate.dart, quando inimigo morre

void executarCombate(
  Jogador jogador,
  Inimigo inimigo,
  Economia economia,
) {
  // ... combate normal ...
```

```
  if (!inimigo.estaVivo) {
    print('${inimigo.nome} foi derrotado!');

    final drops = economia.resolverDrop(inimigo.nome);

    for (final drop in drops) {
      final partes = drop.split(':');
      final tipo = partes[0];
      final quantidade = int.parse(partes[1]);

      if (tipo == 'ouro') {
        jogador.ouro += quantidade;
        print('Você ganhou $quantidade ouro!');
      } else {
        jogador.adicionarItem(tipo, quantidade);
        print('Você encontrou: $tipo x$quantidade');
      }
    }
  }
}
```

Testando a Curva de Dificuldade

Uma boa economia só se revela após testes. Simule 100 corridas e veja se você sai ganhando ou quebrado. A classe `SimuladorEconomia` roda múltiplas corridas hipotéticas, contando ouro ganho, e mostra estatísticas: média, mínimo, máximo. Se a média é muito alta ou muito baixa, você ajusta as constantes e testa de novo.

```
// lib/simulador_economia.dart

import 'dart:math';
import 'economia.dart';
import 'tabelas_drops.dart';
```

```
/// Simula corridas de teste para validar balanceamento
class SimuladorEconomia {
  final Economia economia;

  SimuladorEconomia(this.economia);

  /// Simula N corridas e retorna estatísticas médias
  Map<String, dynamic> simularCorridas(int numCorridas) {
    final stats = <int>[];

    for (int i = 0; i < numCorridas; i++) {
      int ouroTotal = 0;

      for (final nomeInimigo in ['Zumbi', 'Lobo', 'Orc']) {
        final drops = economia.resolverDrop(nomeInimigo);
        for (final drop in drops) {
          final partes = drop.split(':');
          if (partes[0] == 'ouro') {
            ouroTotal += int.parse(partes[1]);
          }
        }
      }

      stats.add(ouroTotal);
    }

    final media = stats.reduce((a, b) => a + b) / stats.length;
    final minimo = stats.reduce((a, b) => min(a, b));
    final maximo = stats.reduce((a, b) => max(a, b));

    return {
      'corridas': numCorridas,
      'ouro_medio': media.toStringAsFixed(2),
      'ouro_minimo': minimo,
      'ouro_maximo': maximo,
```

```
    };
  }
}
```

Uso (exemplo de como rodar a simulação):

```
void main() {
  final economia = Economia(tabelasDrops: TabelasDrops.criar());
  final simulador = SimuladorEconomia(economia);

  final resultado = simulador.simularCorridas(100);
  print('Simulação de 100 corridas:');
  print(resultado);
}
```

Saída esperada:

```
Simulação de 100 corridas:
{corridas: 100, ouro_medio: 85.45, ouro_minimo: 52, ouro_maximo: 148}
```

Se a média é muito baixa, aumente `kOuroBasePorInimigo`. Se é muito alta, reduza. Isto é iteração de design.

Dificuldade por Andar

A dificuldade aumenta gradualmente. Quanto mais fundo, mais perigoso. O sistema usa `getDificuldadeAndar()` para calcular um multiplicador: no andar 0, é 1.0x (normal). No andar 3, é 1.6x (60% mais forte). No andar 10, é 3.0x (3 vezes mais forte).

```
// Exemplo de escalação de dificuldade por andar

void aplicarDificuldadeAndar(
```

```
  Inimigo inimigo,
  int andarNumero,
  Economia economia,
) {
  final multiplicador = economia.getDificuldadeAndar(andarNumero);

  inimigo.hpMax = (inimigo.hpMax * multiplicador).toInt();
  inimigo.hp = inimigo.hpMax;

  inimigo.ataque = (inimigo.ataque * multiplicador).toInt();

  print('Inimigo escalonado para andar $andarNumero: '
      'HP=${inimigo.hpMax}, ATK=${inimigo.ataque}');
}
```

Isto significa:

- Andar 0: Zumbi tem 10 HP
- Andar 3: Zumbi tem 10 * 1.6 = 16 HP
- Andar 10: Zumbi tem 10 * 3.0 = 30 HP

A mesma criatura fica progressivamente mais desafiadora.

Desafios da Masmorra

Desafios Básicos

Desafio 22.1. O Tesouro do Dragão Antigo. A lenda diz que um dragão guardava uma Chave Dourada nos tempos antigos. Crie uma nova tabela de *drops* onde o Dragão tem 5% de chance de deixar cair essa chave rara. Implemente em `EntradaSaque` com id `'chave_dourada'`, chance `0.05`, quantidade 1, descrição épica. Teste: derrote o dragão 20 vezes, conte quantas vezes recebe a chave. A probabilidade bate com 5%? Dica: use `EntradaSaque` para encapsular cada possível *drop*.

Desafio 22.2. Ganância do Comerciante. O comerciante da masmorra cobrava margem de 50%. Você descobriu que ele é ganancioso demais. Reduza a margem de venda para 30% mudando `kMargemVenda` de 0.5 para 0.3. Agora uma Espada de Ferro que custa 50 ouro vale quanto em venda?

Calcule manualmente e depois valide em código. Os preços mais justos faz você comprar mais itens? Dica: novo preço = 50 × 0.3.

Desafios Avançados

Desafio 22.3. A Maldição dos Cinco Andares. Você desce 5 andares, cada um com 3 Lobos hostis. Implemente uma simulação: (1) Faça loop dos andares 0-4, (2) em cada andar, gere 3 Lobos com HP escalonado por `getDificuldadeAndar()`, (3) resolva *drops* de cada lobo, (4) some o ouro total. Execute e veja: quantos ouro ganharam? O HP dos lobos aumenta conforme desce? A economia se ajusta naturalmente? Dica: imprima resumo: "Andar X: 3 Lobos, Y ouro, HP variou de Z a W".

Desafio 22.4. Modo Fácil para Aprendizes. Criar um jogo que escala dificuldade é difícil. Você quer testar em modo fácil onde tudo é menos letal. Crie `EconomiaFacil extends Economia`: dificuldade em 50% (inimigos mais fracos), *drops* em 150% (mais ouro). Simule 5 andares em modo fácil e modo normal, compare. Em fácil, sobrevive melhor? Ganha mais ouro? Dica: use `super.getDificuldadeAndar()` para chamar o pai e depois multiplicar.

Desafio 22.5. (Desafio): Raríssimo. Nem todo item é igual. Itens raros são mais caros. Crie um enum `Raridade { comum, raro, mitico }` e adicione esse campo em `EntradaSaque`. Depois, multiplique preço de compra: comum (1x), raro (3x), mítico (10x). Crie 3 *drops* de um inimigo: ouro comum (50 ouro), espada rara (200 ouro), artefato mítico (5000 ouro). Teste o balanceamento: qual é mais comum? Qual mais valioso? Dica: use switch/case no getter `precoCompra()`.

Boss Final 22.6. A Profundeza Recompensa. Conforme desce, as recompensas aumentam. Implemente um bônus de +10% de ouro a cada 2 andares (andar 2→+10%, andar 4→+20%, andar 6→+30%). Integre em `getOuroEscalonado()`. Teste descendo 10 andares: o ouro cresce suavemente ou tem saltos? Sinta-se recompensado pela sua coragem. Dica: use `(andar ~/ 2) * 0.10` para calcular bônus.

Pergaminho do Capítulo

Neste capítulo, você aprendeu:

- Loot tables modelam o que cada inimigo deixa cair quando morre
- `EntradaSaque` encapsula item, chance e quantidade; `Rolador` resolve aleatoriedade
- `Economia` é o governador central: preços, *drops*, dificuldade escalonada

- Constantes de balanceamento permitem ajustar o jogo rapidamente
- Simulação valida a curva: 100 corridas revelam se o jogo é justo ou quebrado
- Dificuldade por andar escala inimigos naturalmente, sem queda abrupta

A economia é o pulso invisível. Inimigos derrotados alimentam o ciclo: ouro para armas melhores para derrotar inimigos mais fortes. Sem isto, o jogo é apenas um labirinto.

Dica Profissional

Testes de economia são tão importantes quanto testes de código. Uma simples mudança em `kAumentoOuroPorAndar` (0.3 para 0.5) pode quebrar o balanceamento inteiro. Use simulações: rode 1000 corridas, meça ouro médio, morte média, velocidade de progressão. Se a curva não é suave, volta atrás. Economia é iteração contínua, não é "colocar números e esperar". Dados revelam verdades que intuição esconde.

Próximo Capítulo

No Capítulo 23, a economia ganha uma interface tangível. Vamos construir a loja do mercador — com `ItemVenda`, `Mercador`, `LojaRenderer` e `ModoLoja` — onde o jogador pode comprar, vender e planejar estraticamente seus próximos movimentos.

Capítulo 23 - A Loja do Mercador: UI e Fluxo

Entrou numa loja. O ar cheira a madeira antiga e moedas de ouro. Atrás do balcão, um homem de barba cinzenta sorri. "O que quer comprar?" As armas brilham na parede. As poções estão arrumadas em prateleiras. Este é o espaço de respiração do roguelike onde você estratégia: que armadura devo carregar? Quanto ouro devo guardar? Vale a pena vender isto agora? Aqui a economia ganha presença física e interface clara.

Integração com o Sistema de Economia

A loja do Capítulo 23 é construída sobre os fundamentos do Capítulo 22: a `Economia` define preços de compra e venda (via `precoCompra()` e `precoVenda()`), as tabelas de drops alimentam o inventário do jogador com itens valiosos, e as recompensas escalonadas por andar permitem que você tenha progressivamente mais ouro para investir em equipamento melhor. Agora veremos como essa economia abstrata ganha corpo: uma loja real, com um comerciante real, onde você navega, escolhe e transaciona.

A loja não existe sozinha; ela é o ponto de encontro entre a progressão de dificuldade e a agência do jogador em decidir como gastar suas recompensas. Diferentemente do combate automático ou da exploração que ocorre naturalmente, a loja é um **espaço de pausa e decisão**. É aqui que você reflete: tenho ouro suficiente para esta espada? Vale vender este item descartável? Devo guardar ouro para um futuro mais difícil? A economia só importa quando você a sente tangibilizada em uma interface clara e responsiva.

O Que Vamos Aprender

Neste capítulo você vai:

- Modelar a classe `Mercador` e seu inventário (`ItemVenda`)
- Criar a UI ASCII da *shop*: layout com colunas, listas, preços
- Implementar compra (se ouro >= preço) e venda (se item em inventário)

- Gerenciar modo *shop* vs modo exploração: dois estados de jogo distintos
- Validar operações: não deixar comprar sem ouro, não vender o que não tem
- Implementar *restock*: a loja muda de items a cada andar/visita
- Integrar *shop* no fluxo de jogo: entrada por sala especial, saída natural
- Renderizar feedback visual: “Comprado!”, “Sem ouro!”, “Inventário cheio!”

Ao final, você terá uma loja completa e jogável que funciona como uma entidade real do jogo.

O Conceito da Loja

A loja é mais que um menu. É uma experiência completa:

1. Uma sala física; você entra por ação específica (digita `shop` ou pisa numa sala especial marcada)
2. Um estado de jogo distinto; não há movimento ou combate, apenas compra/venda
3. Inventário dinâmico; muda a cada andar ou a cada visita, oferecendo itens progressivamente melhores
4. Interface clara e contextual; lista de items à venda, lista de seus items, preços visíveis
5. Transações *type-safe* e validadas; compra com verificação de ouro, venda com verificação de inventário

Classe ItemVenda e Inventário do Mercador

Um item à venda não é só um `Item`. Tem um preço e uma quantidade em estoque. A classe `ItemVenda` encapsula isto: o item, quanto custa, quantos estão disponíveis. Oferece métodos para remover do estoque (quando você compra) e verificar se ainda tem estoque disponível.

Por que separar `ItemVenda` de `Item`? Um `Item` é imutável e genérico — pode ser em qualquer lugar (inventário, drop, loja). Um `ItemVenda` é um `Item` com contexto comercial: preço e quantidade. Separar essas responsabilidades evita poluir `Item` com dados de negócio. Além disso, o mesmo item pode ter preços diferentes em lojas diferentes; `ItemVenda` permite essa flexibilidade sem clonar o item todo.

```dart
// lib/item_venda.dart

import 'item.dart';

/// Um item no inventário da loja (com preço e quantidade)
class ItemVenda {
  final Item item;
  final int precoCompra;
  int quantidade;

  ItemVenda({
    required this.item,
    required this.precoCompra,
    required this.quantidade,
  });

  // ← O comerciante compra por X mas vende por
  // 50% do preço (margem de lucro)
  int get precoVenda => (precoCompra * 0.5).toInt();

  // ← Verifica antes de permitir compra; evita
  // tentar vender algo sem estoque
  bool get temEstoque => quantidade > 0;

  void removerDoEstoque() {
    if (quantidade > 0) quantidade--;
  }

  void adicionarAoEstoque() {
    quantidade++;
  }

  @override
  String toString() =>
      '${item.nome} ($precoCompra ouro) × $quantidade';
}
```

Classe Mercador

O `Mercador` gerencia todas as transações da loja: compra, venda e restoque. É o coração lógico da economia local. Cada operação é validada rigorosamente antes de modificar estado.

Ordem de validação em compra (crucial!): (1) verificar se índice é válido, (2) verificar se item tem estoque, (3) verificar se jogador tem ouro suficiente, (4) verificar se inventário do jogador tem espaço, (5) aplicar a transação. Essa ordem importa profundamente porque evita estados inconsistentes: você não quer descontar ouro e depois descobrir que o inventário está cheio. Validação antes de modificação é padrão ouro em transações: se qualquer validação falha, o estado inteiro permanece inalterado (atomicidade).

```
// lib/mercador.dart

import 'jogador.dart';
import 'item.dart';
import 'economia.dart';
import 'item_venda.dart';

/// O comerciante da loja
class Mercador {
  List<ItemVenda> inventario;
  final Economia economia;
  final String nome;

  Mercador({
    required this.inventario,
    required this.economia,
    this.nome = 'Mestre Aldwin',
  });

  /// Compra um item do inventário da loja
  /// Retorna mensagem de sucesso ou erro sem modificar estado se
  ↪  falhar
  String comprar(Jogador jogador, int indiceItem) {
```

```
  // Validação 1: Índice existe?
  if (indiceItem < 0 || indiceItem >= inventario.length) {
    return 'Item inválido!';
  }

  final itemVenda = inventario[indiceItem];

  // Validação 2: Item tem estoque?
  if (!itemVenda.temEstoque) {
    return '${itemVenda.item.nome} está em falta.';
  }

  // Validação 3: Jogador tem ouro suficiente?
  if (jogador.ouro < itemVenda.precoCompra) {
    return 'Você não tem ouro suficiente '
        '(custo: ${itemVenda.precoCompra})';
  }

  // Validação 4: Mochila tem espaço?
  if (jogador.inventario.length >= jogador.tamanhoInventario) {
    return 'Inventário cheio!';
  }

  // Todas as validações passaram: executar transação atomicamente
  jogador.ouro -= itemVenda.precoCompra;
  jogador.adicionarItem(itemVenda.item);
  itemVenda.removerDoEstoque();

  return '${itemVenda.item.nome} comprado '
      'por ${itemVenda.precoCompra} ouro!';
}

/// Vende um item do inventário do jogador para a loja
/// O comerciante compra por menos do que venderia (margem)
String vender(Jogador jogador, int indiceItem) {
  if (indiceItem < 0 || indiceItem >= jogador.inventario.length) {
```

```
      return 'Item inválido!';
    }

    final item = jogador.inventario[indiceItem];
    // ← Usa Economia.precoVenda() que aplica desconto percentual
    final precoVenda = economia.precoVenda(item.id);

    if (precoVenda <= 0) {
      return 'Este item não tem valor!';
    }

    jogador.ouro += precoVenda;
    jogador.inventario.removeAt(indiceItem);
    // ← Adiciona item vendido ao catálogo da loja (restock dinâmico)
    _adicionarAoEstoqueDaLoja(item, precoVenda);

    return '${item.nome} vendido por $precoVenda ouro!';
  }

  void _adicionarAoEstoqueDaLoja(Item item, int preco) {
    final existe = inventario.firstWhereOrNull(
      (iv) => iv.item.id == item.id,
    );

    if (existe != null) {
      existe.adicionarAoEstoque();
    } else {
      inventario.add(ItemVenda(
        item: item,
        precoCompra: preco,
        quantidade: 1,
      ));
    }
  }

  /// Restoque progressivo: itens melhoram conforme você desce
```

```
/// Cada andar muda o catálogo disponível (progression gate)
void restoquear(int andarNumero) {
  inventario.clear();
  inventario.addAll(_inventarioBase());

  // ← A partir do andar 3: espadas, armaduras (equipamento)
  if (andarNumero >= 3) {
    inventario.addAll(_inventarioAndarAvancado());
  }
  // ← A partir do andar 7: artefatos lendários (gambit final)
  if (andarNumero >= 7) {
    inventario.addAll(_inventarioAndarMuitoAvancado());
  }
}

List<ItemVenda> _inventarioBase() {
  return [
    ItemVenda(
      item: Item(
        id: 'pocao_vida',
        nome: 'Poção de vida',
        descricao: 'Restaura 20 HP quando usada.',
      ),
      precoCompra: 25,
      quantidade: 5,
    ),
    ItemVenda(
      item: Item(
        id: 'pocao_mana',
        nome: 'Poção de mana',
        descricao: 'Restaura 10 mana quando usada.',
      ),
      precoCompra: 15,
      quantidade: 3,
    ),
  ];
```

```
  }

  List<ItemVenda> _inventarioAndarAvancado() {
    return [
      ItemVenda(
        item: Item(
          id: 'espada_aco',
          nome: 'Espada de aço',
          descricao: 'Uma lâmina bem forjada. +3 ataque.',
        ),
        precoCompra: 75,
        quantidade: 2,
      ),
      ItemVenda(
        item: Item(
          id: 'armadura_couro',
          nome: 'Armadura de couro',
          descricao: 'Proteção básica. +2 defesa.',
        ),
        precoCompra: 50,
        quantidade: 1,
      ),
    ];
  }

  List<ItemVenda> _inventarioAndarMuitoAvancado() {
    return [
      ItemVenda(
        item: Item(
          id: 'espada_mithril',
          nome: 'Espada de mithril',
          descricao: 'Lendária e afiada. +6 ataque.',
        ),
        precoCompra: 200,
        quantidade: 1,
      ),
```

```
    ];
  }
}

extension _FirstWhereOrNull<T> on List<T> {
  T? firstWhereOrNull(bool Function(T) test) {
    for (final element in this) {
      if (test(element)) return element;
    }
    return null;
  }
}
```

Dica: O método `firstWhereOrNull()` faz parte do `package:collection`. Adicione-o ao `pubspec.yaml`:

```
dependencies:
  collection: ^1.18.0
```

UI ASCII da Loja

A loja precisa de uma interface clara que mostre: que items você pode comprar, seus preços, seu inventário, seu ouro atual. A classe `LojaRenderer` desenha tudo em ASCII: cabeçalho com nome do comerciante, lista de items à venda, seu inventário, e HUD com status.

Por que separar UI de lógica? Isso segue o padrão *MVC* (Model-View-Controller): `Mercador` é o modelo (gerencia dados e regras de negócio), `LojaRenderer` é a visão (desenha na tela), e `ModoLoja` é o controlador (processa entrada). Essa separação é crítica: se você quiser mudar como a loja aparece (cores, layout, animações), muda apenas o renderer. A lógica de compra/venda fica segura, testável e reutilizável em outros contextos (web, mobile, etc.). O *Mercador* nunca precisa saber que usa ASCII; poderia usar gráficos ou terminal com cores e funcionaria igualmente.

```
// lib/loja_renderer.dart

import 'jogador.dart';
import 'mercador.dart';

/// Renderiza a interface da loja
class LojaRenderer {
  final int largura;
  final int altura;

  LojaRenderer({
    this.largura = 80,
    this.altura = 24,
  });

  void renderizar(Jogador jogador, Mercador mercador) {
    _desenharCabecalho(mercador);
    _desenharColunasCompraVenda(jogador, mercador);
    _desenharHud(jogador);
  }

  void _desenharCabecalho(Mercador mercador) {
    print('=' * largura);
    print('╟ LOJA DO ${mercador.nome.toUpperCase()} ╢');
    print('=' * largura);
  }

  void _desenharColunasCompraVenda(Jogador jogador, Mercador
  ↪  mercador) {
    print('\n COMPRAR NA LOJA');
    print('-' * 40);

    // ← Lista cada item com índice para seleção rápida
    for (int i = 0; i < mercador.inventario.length; i++) {
      final item = mercador.inventario[i];
      final linha =
```

```
          '[$i] ${item.item.nome} --- ${item.precoCompra} '
          'ouro (${item.quantidade})';

      if (!item.temEstoque) {
        print('(fora de estoque) $linha');
      } else {
        print(linha);
      }
    }

    print('\n TEU INVENTÁRIO');
    print('—' * 40);

    // ← Mostra seu inventário com preço de venda (margem aplicada)
    for (int i = 0; i < jogador.inventario.length; i++) {
      final item = jogador.inventario[i];
      final precoVenda = (item.preco ?? 0) ~/ 2;
      final linha = '[$i] ${item.nome} (⌬$precoVenda ouro)';
      print(linha);
    }

    if (jogador.inventario.isEmpty) {
      print('(vazio)');
    }
  }

  void _desenharHud(Jogador jogador) {
    print('\n┌─ STATUS ──────────────────────────┐');
    print('│ Ouro: ${jogador.ouro.toString().padLeft(6)}  '
        'HP: ${jogador.hp}/${jogador.maxHp}');
    print('│ Inventário: ${jogador.inventario.length}/'
        '${jogador.tamanhoInventario}');
    print('└──────────────────────────────────┘');

    print('Digita: [C]omprar [nº] | [V]ender [nº] | [S]air |
↪  [A]juda');
```

```
  }
}
```

Modo Loja: State Machine

A loja é um estado diferente do jogo. Precisa da sua própria máquina de estados. Enquanto você está na loja, o mundo exterior é **pausado**: não há movimento, não há inimigos, só compra e venda. A classe `ModoLoja` implementa um loop independente que é totalmente desacoplado do loop principal do dungeon: renderiza a interface, lê comando do usuário (comprar, vender, sair), processa a ação, renderiza novamente. Quando sai da loja, retorna ao mapa e o jogo continua — nenhuma ação foi perdida, nenhuma passagem de tempo ocorreu.

Esse padrão é chamado *state machine* (máquina de estados): o jogo tem múltiplos estados (exploração, combate, loja), e cada um tem seu próprio loop e lógica. A transição entre estados é explícita e controlada.

```
// lib/modo_loja.dart

import 'dart:io';
import 'jogador.dart';
import 'tela_ascii.dart';
import 'mercador.dart';
import 'loja_renderer.dart';

/// Executa a sessão de loja (estado especial do jogo)
class ModoLoja {
  final Jogador jogador;
  final Mercador mercador;
  final TelaAscii tela;
  final LojaRenderer renderer;

  bool emLoja = true;

  ModoLoja({
```

```
    required this.jogador,
    required this.mercador,
    required this.tela,
  }) : renderer = LojaRenderer(tela: tela);

  void executar() {
    renderer.renderizar(jogador, mercador);

    while (emLoja) {
      stdout.write('\n> ');
      final comando = stdin.readLineSync() ?? 'ajuda';
      processarComando(comando.trim());
      renderer.renderizar(jogador, mercador);
    }

    print('\nVocê saiu da loja.');
  }

  void processarComando(String cmd) {
    final partes = cmd.split(' ');
    final acao = partes[0].toLowerCase();

    switch (acao) {
      case 'comprar' || 'c':
      // ← Comando: 'comprar 0' ou 'c 0' para comprar item na posição 0

        if (partes.length < 2) {
          print('Uso: comprar <número>');
          break;
        }
        final indice = int.tryParse(partes[1]);
        if (indice != null) {
          final mensagem = mercador.comprar(jogador, indice);
          print(mensagem);
        }
        break;
```

```
      case 'vender' || 'v':
        // ← Comando: 'vender 0' ou 'v 0' para vender
        // item do seu inventário
        if (partes.length < 2) {
          print('Uso: vender <número>');
          break;
        }
        final indice = int.tryParse(partes[1]);
        if (indice != null) {
          final mensagem = mercador.vender(jogador, indice);
          print(mensagem);
        }
        break;

      case 'sair' || 's':
        // ← Define flag que interrompe o loop principal da loja
        emLoja = false;
        break;

      case 'status':
        print('Ouro: ${jogador.ouro} | '
            'HP: ${jogador.hp}/${jogador.maxHp}');
        break;

      default:
        print('Comando desconhecido. Digita "ajuda".');
    }
  }
}
```

Saída Esperada

Quando você entra na loja e interage com o comerciante, a saída no terminal se parece com isto:

```
⌋
↪ ═══════════════════════════════════════════════════════════════════════════
╟ LOJA DO MESTRE ALDWIN                                                      ╢
⌋
↪ ═══════════════════════════════════════════════════════════════════════════

 COMPRAR NA LOJA
──────────────────────────────────────────


[0] Poção de vida --- 25 ouro (5)
[1] Poção de mana --- 15 ouro (3)
[2] Espada de aço --- 75 ouro (2)
[3] Armadura de couro --- 50 ouro (1)

 TEU INVENTÁRIO
──────────────────────────────────────────


[0] Moeda de ouro (⌬500 ouro)
[1] Adaga enferrujada (⌬10 ouro)
[2] Torção de corda (⌬5 ouro)

┌─ STATUS ──────────────────────┐
│ Ouro:    850  HP: 40/50       │
│ Inventário: 3/10              │
└───────────────────────────────┘

Digita: [C]omprar [nº] | [V]ender [nº] | [S]air | [A]juda

> c 0

Poção de vida comprado por 25 ouro!

[Tela redraw com novo Ouro: 825, Inventário: 4/10]

> v 2
```

```
Torção de corda vendido por 2 ouro!

[Tela redraw: torção removida do inventário, item adicionado à loja]

> s

Você saiu da loja.
```

Este exemplo mostra: - **Menu de compra** com índices, nomes e preços de cada item - **Seu inventário** com índices e preços de venda (50% do preço original) - **HUD de status** com ouro atual, HP, e espaço de mochila - **Feedback de transação** (“Comprado!”, “Vendido!”) após cada ação - **Estado dinâmico** que se atualiza a cada compra/venda

Antes vs. Depois

Antes: Loja Inexistente

```
Jogador derrota inimigo → ganha item → item vai pro inventário
Não há chance de vender, trocar ou planejar.
Economia invisível: você não entende o valor dos itens.
```

Depois: Loja Tangível

```
Jogador derrota inimigo → ganha item → pode entrar na loja
Ao entrar: vê itens à venda, preços e seu ouro destacados
Pode vender itens desnecessários → ganha ouro → compra melhor
A economia é visual e decisória: cada compra é uma escolha
↪  estratégica.
Cada andar, loja muda de catálogo → cria senso de progressão e
↪  novidade.
```

Por Que Não Uma Loja Automática? (Alternativa: Análise Crítica)

Você pode pensar: "Por que não a loja oferece itens automaticamente ao meu inventário?" Resposta: porque isso elimina **agência do jogador**. Uma loja deve ser um espaço de pausa e reflexão. Quando você entra na loja e vê aquela Espada Lendária custando 500 ouro, há conflito: "Tenho 450. Vale a pena vender 3 itens ruins para alcançar 500?" Essa decisão é o design. Uma loja automática que simplesmente despeja itens mata a tensão: você nunca escolhe, apenas aceita o que vem. Além disso, com uma loja manual, você controla quando entra — se quiser economizar ouro para depois, pode. Com automática, é sempre imediato. A escolha é design.

Desafios da Masmorra

Desafio 23.1. Estoque Rotativo: Itens Novos a Cada Andar. Implemente um método `regenerarEstoque()` na loja que troca parte dos itens a cada andar. Use `import 'dart:math'` e `Random().nextInt()` para selecionar 2-3 itens novos de uma lista maior de possibilidades. Cada vez que o jogador retorna à loja (novo andar), alguns itens antigos saem do catálogo e aparecem novos. O mercador comenta: "Chegou mercadoria nova!" quando o estoque muda. Dica: guarde uma lista de `_itemsCatalogo` (todos os itens possíveis) e `_itensAtuais` (o que está na loja agora). Cada chamada a `regenerarEstoque()` remove itens aleatórios e adiciona novos do catálogo. Este padrão de *randomization* controlada é útil em qualquer jogo que queira variedade sem caos.

Desafio 23.2. A Chave do Final. No coração da loja aparece um item lendário: a Chave Dourada que abre a porta do boss final. Crie um `ItemVenda` com nome "Chave Dourada Rara", id `'chave_dourada'`, preço 500 ouro, estoque 1. Adicione à loja (método `_inventarioBase()` ou crie um método novo). Teste: navegue a loja, veja a chave, pergunte: tenho ouro suficiente para comprar? Dica: use a classe `ItemVenda` com seu construtor para não repetir dados.

Desafio 23.3. O Roubo do Comerciante. Você negocia com o comerciante: uma Espada de Aço de 75 ouro em compra. Quanto ele oferece quando você quer vender de volta? Calcule manualmente (resposta: 37.5 ouro com margem 50%, ou 22.5 com margem 30%). Depois implemente no código e valide. O comerciante te prejudica na venda? Quanto você perde em uma transação completa (compra e venda)? Dica: sinta a economia em ação.

Desafio 23.4. O Tesouro da Profundeza. Conforme você desce muito fundo (andar 10 e além), a loja recebe artefatos lendários. Crie um método `_inventarioAndarProfundo()` que retorna itens épicos: "Espada Ancestral" (+10 ataque, 5000 ouro), "Anel de Imortalidade" (impede morte uma vez, 8000 ouro), "Tomo de Poder" (+5 ao nível, 6000 ouro). Integre em `restoquear()` com uma condição: `if (andar >= 10)`. Teste descendo até o andar 10, entre na loja, veja os itens novos aparecerem. Dica: siga o padrão de `_inventarioAndarInicial()`.

Desafio 23.5. Loja Segura com Exceções. A loja não pode quebrar. Se você não tem ouro, lança exceção, não trava. Crie `LojaExcecao`, `OuroInsuficienteExcecao`, `MochilaCheia Excecao`. Refatore `Mercador.comprar()` para verificar e lançar exceções ao invés de retornar strings de erro. Na UI, capture exceções e exiba mensagens amigáveis. Teste tentando comprar sem ouro, com mochila cheia, etc. Código mais robusto = jogo mais confiável. Dica: use try/catch na loja. Exceções são a abordagem *idiomatic* em Dart para erros irrecuperáveis; retornar strings é anti-padrão.

Desafio 23.6. (Desafio): Ofertas do Dia. Todo dia, a loja tem 3 itens especiais em destaque com 50% de desconto. Use `DateTime.now()` para pegar a data e criar seed determinística (ex: `seed = DateTime.now().year * 10000 + DateTime.now().month * 100 + DateTime.now().day`). Assim, o mesmo dia sempre tem os mesmos deals. Teste: reinicie o jogo 2x no mesmo dia, verá os mesmos deals? Reinicie no dia seguinte, verá offers diferentes? Dica: isso recompensa jogadores diários.

Boss Final 23.7. Itens Únicos e Valiosos. Itens lendários não devem estar sempre em estoque. Implemente: itens marcados como "raro=true" têm estoque máximo 1 por andar. Após vender, volta a 1 no próximo andar. Teste: compre a "Espada Ancestral" do andar 10, vá para andar 11, retorne ao 10, item deve estar de novo disponível. Outros itens normais sempre têm restoque completo. Dica: separe a lógica de restoque para itens raros vs normais.

Pergaminho do Capítulo

Neste capítulo, você aprendeu:

- **`ItemVenda`**: Encapsula um item com preço de compra, preço de venda (margem do comerciante), e quantidade em estoque. Oferece métodos para verificar disponibilidade e remover/adicionar unidades.
- **`Mercador`**: Gerencia todas as transações de compra e venda. Valida cada operação em ordem (índice → estoque → ouro → espaço → aplicar),

retorna mensagens de sucesso ou falha, e reaplica o inventário quando muda de andar.

- `LojaRenderer`: Desenha a interface ASCII com cabeçalho decorado, colunas duplas (loja à esquerda, seu inventário à direita), preços visíveis, quantidades de estoque, e HUD com status (ouro, HP, espaço de mochila).
- `ModoLoja`: É uma máquina de estados independente que pausar o jogo principal. Renderiza a loja, lê comandos (comprar, vender, sair), processa transações, e volta a renderizar. Você não pode andar ou combater enquanto está aqui.
- **Padrão MVC**: A lógica (Mercador) é separada da visão (LojaRenderer) e do controle (ModoLoja). Isso permite reutilizar o Mercador em diferentes UIs (web, CLI alternativa, etc.).
- **Integração com Economia**: A loja usa `Economia.precoVenda()` e `Economia.precoCompra()` para calcular preços, conectando-se ao balanceamento definido no Capítulo 22. Drops de inimigos alimentam seu inventário; você vende itens extras na loja para financiar melhorias.
- **Restocking Dinâmico**: A loja muda de inventário a cada andar visitado. Andares 0-2 têm poções básicas, 3+ adicionam armas, 7+ adicionam lendárias. Isso cria senso de progressão: cada visita oferece novas oportunidades.
- **Feedback Visual e Clareza**: Cada transação devolve uma mensagem clara. O layout em colunas deixa evidente o que você tem versus o que pode comprar. Números são visíveis e contextualizados. Isto é design de produto: se o jogador não entende, não engaja.

A loja transforma a economia em interface tangível. Não é apenas números: é um espaço onde você estratégia, troca, planeja.

Dica Profissional

A UI é parte do design, não cosméticos. Uma loja bem desenhada faz o jogador querer entrar, explorar e decidir. Layout em colunas, números claros, feedback visual — tudo isto é design de produto. Se o jogador não sabe que pode comprar, não vai comprar. Invista tempo em feedback e formatação.

Próximo Capítulo

No Capítulo 24, vamos dar superpotência a esta economia através de generics e pattern matching em Dart 3. Criaremos um sistema de eventos tipado que dispara notificações quando itens são comprados, vendidos ou equipados. Cada evento tem sua própria estrutura e é processado com `switch` exaustivo.

Capítulo 24 - Generics e Pattern Matching: Sistema de Eventos

A economia gera ouro. A loja oferece itens. Você compra uma espada e... sente nada, sem feedback. Mas quando equipa essa espada, o seu ataque sobe. Quando bebe uma poção, o seu HP sobe. Quando level-up, ganha habilidades. Cada uma destas ações é um evento, um fato histórico que o jogo deveria registrar. Este capítulo transforma o silêncio em narrativa: um sistema de eventos tipado que registra, filtra e notifica em tempo real. Aqui aprenderá o poder dos generics e do pattern matching em Dart 3 para criar código limpo e expressivo.

O Que Vamos Aprender

Neste capítulo você vai:

- Entender *generics*: `List<T>`, `BarramentoEventos<T extends EventoJogo>`
- Criar uma hierarquia de *sealed classes* para eventos: `EventoCombate`, `EventoLoot`, `EventoMovimento`, `EventoNivel`
- Usar *pattern matching* em Dart 3: *switch expressions* com destructuring
- Implementar um `BarramentoEventos` genérico que filtra eventos por tipo
- Criar um log de eventos *rich*, com renderização diferenciada por tipo
- Integrar eventos no fluxo de jogo: cada ação dispara um evento
- Mostrar notificações em tempo real: *loot pickups*, *levelups*, combate
- Demonstrar *guard clauses* em pattern matching

Ao final, você terá um sistema de eventos tipado que torna o jogo mais narrativo e reativo.

Generics: Uma Rápida Recordação

Você já conhece `List<T>`. *Generics* são a forma de Dart dizer: "esta estrutura pode guardar qualquer tipo, mas vou ser estrito sobre qual tipo é". É segurança de tipos em tempo de compilação. Sem generics, você teria listas

sem tipo e teria que fazer *cast* (conversão) manual toda vez, arriscando erros em tempo de execução. Com generics, o compilador sabe exatamente o que você guardou e avisa se tenta usar errado.

```
List<int> numeros = [1, 2, 3];
List<String> nomes = ['Alice', 'Bob'];
// Isto não compila:
// numeros.add('texto'); // ← erro em tempo de compilação: esperava
↪  int
```

O `T` é um parâmetro de tipo. Significa: esta lista pode conter qualquer tipo, desde que todas as coisas nela sejam do mesmo tipo.

Generics em classes customizadas: Você pode criar suas próprias classes genéricas. Observe como `Caixa<T>` funciona para qualquer tipo:

```
class Caixa<T> {
  T? conteudo;

  void guardar(T item) {
    // ← T é substituído pelo tipo real quando você cria a Caixa
    conteudo = item;
  }

  T? remover() {
    final temp = conteudo;
    conteudo = null;
    return temp;
  }
}

final caixaInt = Caixa<int>();
caixaInt.guardar(42);
print(caixaInt.remover()); // 42

final caixaString = Caixa<String>();
```

```
caixaString.guardar('magia');
print(caixaString.remover()); // magia
```

Saída esperada:

```
42
magia
```

Cada instância de `Caixa` é especializada para um tipo específico. Isto é o poder dos generics: código reutilizável mas type-safe.

Hierarquia de Eventos com Sealed Classes

Eventos são dados imutáveis que representam algo que aconteceu no jogo. Cada evento é um snapshot de um momento: "você atacou", "coletou item", "subiu de nível". Eventos são *sealed classes* — uma hierarquia fechada onde apenas certos tipos podem existir. Isto garante que quando você processa eventos em um switch, o compilador pode certificar que cobriu **todos** os casos possíveis. Sem sealed classes, seria fácil esquecer um tipo de evento e ter comportamento não esperado.

Por que eventos imutáveis? Uma vez criado um evento, ele representa um fato histórico que não muda. Se você lança `EventoLoot(quantidade: 5)` e depois muda para `3`, você falsificou a história. Imutabilidade força clareza: para mudar o comportamento futuro, lance um novo evento, não modifique o antigo.

```
// lib/evento_jogo.dart

// ← sealed class: apenas as subclasses neste
// arquivo podem herdar EventoJogo
sealed class EventoJogo {
  final DateTime timestamp;

  EventoJogo({DateTime? timestamp})
```

```
      : timestamp = timestamp ?? DateTime.now();
}

/// Evento de combate: você atacou ou foi atacado
class EventoCombate extends EventoJogo {
  final String mensagem;
  final int dano;
  final String? atacante;
  final String? alvo;

  EventoCombate({
    required this.mensagem,
    required this.dano,
    this.atacante,
    this.alvo,
    super.timestamp,
  });

  @override
  String toString() => 'Combate: $mensagem (dano: $dano)';
}

/// Evento de loot: item foi adquirido
class EventoLoot extends EventoJogo {
  final String itemId;
  final String nomeItem;
  final int quantidade;
  final String fonte;

  EventoLoot({
    required this.itemId,
    required this.nomeItem,
    required this.quantidade,
    required this.fonte,
    super.timestamp,
  });
```

```
  @override
  String toString() =>
      'Loot: Adquiriu $quantidade × $nomeItem (de $fonte)';
}

/// Evento de movimento: você moveu-se
/// Nota: usa *records* de Dart 3 para pares de coordenadas.
class EventoMovimento extends EventoJogo {
  // ← Tupla nomeada: (int x, int y) é imutável e anônima
  final (int x, int y) de;
  final (int x, int y) para;

  EventoMovimento({
    required this.de,
    required this.para,
    super.timestamp,
  });

  @override
  String toString() =>
      'Movimento: (${de.$1},${de.$2}) → (${para.$1},${para.$2})';
}

/// Evento de nivelação: você subiu de nível
class EventoNivel extends EventoJogo {
  final int nivelAnterior;
  final int nivelNovo;
  final String bonus;

  EventoNivel({
    required this.nivelAnterior,
    required this.nivelNovo,
    required this.bonus,
    super.timestamp,
  });
```

```
  @override
  String toString() =>
      'Nível UP: $nivelAnterior → $nivelNovo! Bônus: $bonus';
}
```

BarramentoEventos Genérico

Um `BarramentoEventos` é um registador (log) de eventos tipado. Permite subscrições filtradas e callbacks. Pense como um serviço de notificações: alguém dispara um evento (ex: item coletado), e todas as subscrições recebem a notificação automaticamente. Isto desacopla completamente: o gerenciador de combate não precisa saber que a UI existe; combate dispara evento, UI sente e reage de forma independente.

Por que este padrão? Sem eventos, cada ação (ataque, loot, movimento) teria que informar manualmente à UI, ao log, ao sistema de achievements, etc. Código cada vez mais acoplado. Com eventos, há um canal central: tudo que importa dispara um evento, qualquer coisa interessada subscreve. Adicionar nova feature (áudio, efeitos, achievements) é trivial: cria um novo listener, subscreve ao evento relevante, pronto. Nada no código de combate muda.

```
// lib/barramento_eventos.dart

/// Sistema de eventos genérico e tipado
/// T deve ser EventoJogo ou subclasse para garantir compatibilidade
class BarramentoEventos<T extends EventoJogo> {
  final List<T> eventos = [];
  // ← Callbacks: funções que reagem quando um evento é disparado
  final List<void Function(T)> _listeners = [];

  /// Dispara um evento e notifica todos os listeners
  void dispara(T evento) {
    eventos.add(evento);
    // ← Itera listeners e chama cada um; qualquer pode reagir
```

```
    for (final listener in _listeners) {
      listener(evento);
    }
  }

  /// Subscreve a este barramento: ao disparar, seu callback é chamado
  void subscreve(void Function(T) callback) {
    _listeners.add(callback);
  }

  /// Remove um callback da lista de listeners
  void desinscreve(void Function(T) callback) {
    _listeners.remove(callback);
  }

  /// Filtra eventos por tipo: retorna só EventoLoot, por ex.
  /// Útil para gerar relatórios: "quantos loots coletei?"
  List<T> filtrarPorTipo<U extends T>() {
    return eventos.whereType<U>().toList();
  }

  T? get ultimoEvento => eventos.isEmpty ? null : eventos.last;

  String logCompleto() {
    return eventos.map((e) => e.toString()).join('\n');
  }

  void limpar() {
    eventos.clear();
  }

  int get contador => eventos.length;
}
```

Pattern Matching em Dart 3

O *pattern matching* permite desconstructir dados de forma clara e expressar lógica condicional de forma muito mais legível do que if/else aninhados. Dart 3 introduziu *switch expressions* (não apenas statements): em vez de `if (evento is EventoCombate) { ... }`, você escreve um `switch` que desconstructura e filtra simultaneamente. É como decomposição estrutural: "este dado tem essa forma? Se sim, extraia seus campos e processe."

Antes (if/else aninhado):

```
if (evento is EventoCombate) {
  final dano = evento.dano;
  if (dano > 50) {
    print('[CRÍTICO] Dano muito alto: $dano');
  } else {
    print('Dano normal: $dano');
  }
}
```

Depois (pattern matching):

```
// ← Muito mais conciso e legível
final mensagem = switch (evento) {
  EventoCombate(:final dano) when dano > 50 => '[CRÍTICO] Dano:
↪  $dano',
  EventoCombate(:final dano) => 'Dano: $dano',
  _ => 'Outro tipo de evento',
};
```

Aqui está o power:

```
// lib/processador_eventos.dart

class ProcessadorEventos {
  // ← switch expression: retorna um valor (String)
```

```
static String renderizar(EventoJogo evento) {
  return switch (evento) {
    // ← Extrai campos com (:final dano)
    // ← when clause: condição extra; só aplica se dano > 50
    EventoCombate(:final mensagem, :final dano) when dano > 50 =>
      '[CRÍTICO] $mensagem (dano: $dano)',

    // ← Mesmo tipo, mas sem guard clause: todos
    // os combates com dano <= 50
    EventoCombate(:final mensagem, :final dano) =>
      '> $mensagem (dano: $dano)',

    // ← Extrai nomeItem e quantidade; mostra
    // plural se quantidade > 1
    EventoLoot(:final nomeItem, :final quantidade)
        when quantidade > 1 =>
      '+ $quantidade x $nomeItem',

    EventoLoot(:final nomeItem, :final quantidade) =>
      '+ $nomeItem',

   // ← Extrai tupla de coordenadas e acessa campos ($1 = x, $2 = y)
    EventoMovimento(:final de, :final para) =>
      '> Movimento: (${de.$1},${de.$2}) → (${para.$1},${para.$2})',

    EventoNivel(:final nivelNovo, :final bonus) =>
      'LEVEL UP! Nível $nivelNovo! +$bonus',

    // ← Fallback: qualquer outro tipo de evento
    _ => '? Evento desconhecido',
  };
}

// ← switch statement: executa código, não retorna valor
static void processar(EventoJogo evento) {
  switch (evento) {
```

```
      // ← (:final atacante?) = atacante é opcional (pode ser null)
      case EventoCombate(:final dano, :final atacante?) when dano > 0:
        print('> $atacante causou $dano dano!');

      // ← dano < 0 significa você sofreu dano (negativo)
      case EventoCombate(:final dano) when dano < 0:
        print('! Recebeu ${dano.abs()} de dano!');

      case EventoLoot(:final itemId, :final quantidade):
        print('+ Adquiriu: $itemId x$quantidade');

      case EventoNivel(:final nivelAnterior, :final nivelNovo):
        print('* Subiu de nível $nivelAnterior → $nivelNovo!');

      // ← Matches mas não usa nada; break é suficiente
      case EventoMovimento():
        break;

      case _:
        break;
    }
  }
}
```

Saída esperada (após disparar eventos):

```
[CRÍTICO] Dragão ataca! (dano: 75)
+ 3 x Moeda de Ouro
+ Poção de Mana
LEVEL UP! Nível 5! +5 HP, +2 ATK
```

Símbolos descodificados:

- `(:final dano)` desconstructures o campo `dano` de `EventoCombate`
- `when dano > 50` é uma *guard clause*: a correspondência só aplica se verdadeira
- `(:final atacante?)` significa "pode ser null"; o `?` o marca como opcional

- `_` é *match-all*: qualquer coisa; usado para fallback
- `$1` e `$2` acessam campos de um *record* (tupla): primeiro e segundo elementos
- `switch (x)` pode ser expressão (`=> valor`) ou statement (`{ código }`)

Integrando Eventos no Jogo

Cada ação dispara um evento. Quando você se move, um `EventoMovimento` é disparado. Quando coleta item, um `EventoLoot`. Quando sofre dano, um `EventoCombate`. Cada evento é registado no barramento, listeners recebem notificação, renderizam mensagem na tela.

Observe como isso desacopla completamente o código: a classe que gerencia movimento não precisa saber nada sobre renderização. Ela apenas dispara o evento. Qualquer coisa interessada em movimento subscreve ao barramento e reage. Isto permite:

1. **Múltiplos listeners**: UI, log, áudio, achievements — todos reagem ao mesmo evento
2. **Fácil adicionar features**: novo sistema de achievements? Subscreve ao barramento, pronto
3. **Testabilidade**: você pode testar o comportamento sem UI, disparando eventos diretamente
4. **Auditoria**: o histórico de eventos é uma log completa do que aconteceu

```
// lib/dungeon_com_eventos.dart

/// Gerenciador do dungeon que dispara eventos para tudo que acontece
class DungeonComEventos {
  late BarramentoEventos<EventoJogo> eventoBus;
  late Jogador jogador;
  late MapaMasmorra mapa;

  DungeonComEventos() {
    eventoBus = BarramentoEventos<EventoJogo>();
    // ← Subscreve ao barramento: toda vez que algo é disparado,
    //   ProcessadorEventos.renderizar() é chamado
    eventoBus.subscreve((evento) {
```

```
    print(ProcessadorEventos.renderizar(evento));
  });
}

/// Movimento dispara evento com origem e destino (para replay/undo)
void moverJogador(int dx, int dy) {
  final xAnterior = jogador.x;
  final yAnterior = jogador.y;

  jogador.x += dx;
  jogador.y += dy;

  // ← Cria um record (tupla) com as coordenadas
  eventoBus.dispara(EventoMovimento(
    de: (xAnterior, yAnterior),
    para: (jogador.x, jogador.y),
  ));
}

/// Ganhar item dispara evento de loot com fonte
void ganharItem(String itemId, String nomeItem, int quantidade) {
  jogador.adicionarItem(itemId);

  // ← Registra a fonte do item (chão, inimigo, loja, etc.)
  eventoBus.dispara(EventoLoot(
    itemId: itemId,
    nomeItem: nomeItem,
    quantidade: quantidade,
    fonte: 'chão',
  ));
}

/// Sofrer dano dispara combate (dano negativo = você recebeu dano)
void sofrerDano(int dano) {
  jogador.hp -= dano;
```

```
    // ← dano é negativo aqui; ProcessadorEventos
    // interpreta como sofrimento
    eventoBus.dispara(EventoCombate(
      mensagem: 'Sofreste dano!',
      dano: -dano,
      alvo: 'Jogador',
    ));
  }

  /// Level up dispara evento com bônus
  void levelUp() {
    final nivelAnterior = jogador.nivel;
    jogador.nivel++;

    eventoBus.dispara(EventoNivel(
      nivelAnterior: nivelAnterior,
      nivelNovo: jogador.nivel,
      bonus: '+5 HP, +2 ATK',
    ));
  }
}
```

Saída esperada (sequência típica):

```
> Movimento: (5,3) → (6,3)
+ Poção de Mana
> Dragão causou 15 dano!
! Recebeu 15 de dano!
* Subiu de nível 3 → 4!
```

Antes vs. Depois: Arquitetura de Eventos

Antes: Acoplamento Direto

```
// Combate conhece tudo; tudo está entrosado
class Combate {
  void atacar(Inimigo inimigo) {
    int dano = calcularDano(heroi.arma, inimigo.defesa);
    inimigo.hp -= dano;

    // UI deve saber como renderizar
    ui.mostrarDano(dano);

    // Log deve saber como registrar
    log.adicionarLinha('Ataque causou $dano');

    // Sistema de sons deve saber
    audio.tocarSomCombate(dano);

    // Se adicionar achievements, muda tudo aqui
    if (dano > 50) achievements.unlock('CRITICO');
  }
}
```

Problema: Combate conhece UI, Log, Áudio, Achievements. Adicionar novidade quebra tudo.

Depois: Desacoplamento com Eventos

```
// Combate é puro
class Combate {
  final BarramentoEventos eventoBus;

  void atacar(Inimigo inimigo) {
    int dano = calcularDano(heroi.arma, inimigo.defesa);
```

```
    inimigo.hp -= dano;

    // Dispara evento; ninguém especifico é chamado
    eventoBus.dispara(EventoCombate(
      mensagem: 'Ataque!',
      dano: dano,
    ));
  }
}

// UI subscreve
eventoBus.subscreve((evento) {
  if (evento is EventoCombate) {
    ui.mostrarDano(evento.dano);
  }
});

// Log subscreve
eventoBus.subscreve((evento) {
  log.adicionarLinha(evento.toString());
});

// Áudio subscreve
eventoBus.subscreve((evento) {
  if (evento is EventoCombate) {
    audio.tocarSomCombate(evento.dano);
  }
});

// Achievements subscreve
eventoBus.subscreve((evento) {
  if (evento is EventoCombate && evento.dano > 50) {
    achievements.unlock('CRITICO');
  }
});
```

Ganho: Combate não muda. Cada sistema subscreve independentemente. Adicionar nova feature é adicionar um novo listener — zero modificação do código existente.

Por Que Não Apenas Callbacks Simples?

Você pode pensar: "Por que não apenas passar um callback para Combate.atacar()?" Resposta: porque então Combate precisa conhecer **todos** os callbacks. Se temos UI, Log, Áudio, Achievements, Combate.atacar() teria 4+ parâmetros de callback. Cada novo sistema adiciona um parâmetro. Isto é **hell de parâmetros**.

Com eventos, há um único canal central. Qualquer coisa que queira reagir subscreve uma vez. Combate nunca muda sua assinatura. Isto é escalável: 5 sistemas, 5 listeners independentes. Mil sistemas? Mil listeners independentes. Combate não sabe disso tudo.

Além disso, eventos são **históricos**: você pode guardar uma lista completa de tudo que aconteceu (para replay, debug, análise). Com callbacks, é fugace: reage e esquece.

Desafios da Masmorra

Desafio 24.1. Quando o Guerreiro Muda de Arma. Seu guerreiro equipa uma Espada Lendária, depois a desequipa para voltar ao escudo. Cada mudança conta. Crie `EventoEquipamento extends EventoJogo` com `itemId`, `nomeItem`, `equipado` (bool). Dispare esse evento toda vez que equipar/desequipar. Teste: equipe e desequipe 3 vezes, veja se 6 eventos foram registrados. Dica: sealed class mantém a tipagem segura.

Desafio 24.2. Narrativa do Equipamento. O log de ações deve refletir sua jornada de equipamento. Adicione um case em `ProcessadorEventos.renderizar()`: se `EventoEquipamento` com `equipado=true`, exiba em verde `[EQUIP] Equipaste: Espada Lendária`. Se `equipado=false`, em vermelho `[DESQUIP] Desequipaste:...`. Execute uma sequência de mudanças, o log deve contar a história. Dica: use padrão com guard: `equipado when equipado == true`.

Desafio 24.3. Combate Violento. Nem todo dano é importante. Filtre eventos de combate: retorne só `EventoCombate` com `dano > 20` (golpes críticos e devastadores). Itere e exiba: "Crítico! Dano: 35". Teste: em 100 turnos de combate, quantos golpes foram >= 20 dano? Você vai notar que a maioria é fraca e apenas alguns são épicos. Dica: combine `whereType<EventoCombate>()` com `.where()`.

Desafio 24.4. Resumo da Partida. Ao fim do jogo, você quer saber: Quantas vezes equipou itens? Quantas vezes sofreu dano? Quantas compras na loja? Implemente `contagemPorTipo()` que retorna um mapa: `{'EventoCombate': 145, 'EventoEquipamento': 8, 'EventoCompra': 3}`. Use pattern matching no switch para cada tipo. Execute uma partida e veja o resumo final. Dica: isto é análise agregada.

Desafio 24.5. (Desafio): Assista a Sua Epopeia. Você quer mostrar a um amigo o que aconteceu na masmorra. Implemente `EventReplay` que armazena eventos e tem método `async tocar()`: exibe cada evento com 500ms entre eles. Use `Future.delayed(Duration(milliseconds: 500))`. Assim, narrativa toda se desenrola visualmente. Teste: grave 50 eventos, toque e veja cada um aparecer sequencialmente. Você consegue acompanhar a história? Dica: `await` faz o programa esperar sem travar.

Boss Final 24.6. Combate Recente. Você quer saber: Nos últimos 5 minutos de jogo, qual foi o dano total sofrido? Implemente um método que retorna eventos de combate ocorridos nos últimos N minutos. Use `DateTime.now()` e `evento.timestamp.difference(DateTime.now()).inMinutes < N`. Some o dano. Teste: após 10 minutos de jogo, pergunte dano dos últimos 3 minutos vs últimos 10. Dica: Delta de tempo revela ritmo de combate.

Pergaminho do Capítulo

Neste capítulo, você aprendeu:

- *Generics* (`<T>`, `<T extends BaseType>`) permitem código reutilizável e *type-safe*
- *Sealed classes* criam hierarquias fechadas e seguras de eventos
- *Pattern matching* em Dart 3 (`switch` com destructuring e `when`) é limpo e expressivo
- `BarramentoEventos<T>` é um sistema de eventos tipado que notifica subscritos (padrão *Observer*)
- `ProcessadorEventos` renderiza eventos de forma humanizada com regras por tipo
- Integração: cada ação do jogo dispara um evento, criando um log narrativo
- Desacoplamento: nenhum sistema precisa conhecer os outros; todos reagem a eventos

O sistema de eventos transforma um jogo silencioso em um que fala. Cada ação é registada, cada vitória é celebrada. A arquitetura fica limpa:

adicionar novidades é subscrever um novo listener, não modificar código existente.

Código Completo no Step

O diretório `code/steps/step-24/` contém a classe `DungeonComEventos`, que integra todos os componentes de eventos ensinados neste capítulo. `DungeonComEventos` é o maestro que orquestra movimento, coleta de itens, dano e subidas de nível — cada ação dispara eventos apropriados no `BarramentoEventos`. É aqui que você vê sealed classes, generics e pattern matching trabalhando juntos para criar uma arquitetura reactiva, limpa e extensível. Consulte o step para ver como a integração entre `ProcessadorEventos`, listeners e o gerenciador central funciona em contexto real.

Dica Profissional

Eventos são a coluna vertebral de sistemas reativos. Quando adicionar uma nova feature (feitiço, item especial, achievement), não altere 50 funções; dispare um evento novo. Qualquer listener que se importe com esse evento vai reagir. Isto é desacoplamento: combate não sabe de UI, UI sente eventos. Mantém o código limpo e modular.

Próximo Capítulo

No Capítulo 25, vamos implementar progressão completa: um sistema de XP com fórmulas, níveis que desbloquem habilidades especiais, e visualização clara do progresso em tempo real.

Capítulo 25 - Progressão: XP, Níveis e Habilidades

Toda aventura é um percurso de transformação: você começa fraco, indefeso, um principiante perdido nas sombras da masmorra. Mas a cada inimigo derrotado, a cada andar descido, você cresce. Seu corpo fica mais forte. Seus reflexos agudizam. Você aprende magias novas e poderosas. Este é o coração da progressão; sem ela, um roguelike é apenas morte repetida. Com ela, é uma jornada épica de ascensão.

O Que Vamos Aprender

Neste capítulo você vai:

- Criar um sistema de XP e níveis com fórmulas quadráticas
- Implementar `TabelaProgressao` com curva de levantamento balanceada
- Aumentar HP máximo e ataque ao subir de nível
- Desbloquear habilidades especiais em marcos de nível (nível 3, nível 5)
- Implementar a classe `Habilidade` com execução dinâmica
- Integrar o event system para notificações de *level up*
- Escalar inimigos por andar (mais difíceis conforme você desce)
- Demonstrar um combate completo com progressão visível

Ao final, seu jogo terá verdadeira sensação de progresso como em *Chrono Trigger* ou *Elden Ring*.

O Sistema de Progressão

Progressão é o motor emocional do *roguelike*. Sem progressão visível, cada morte sente-se como desperdício puro. Com progressão bem calibrada, cada morte é aprendizado: você chegou mais longe, ficou mais forte, desbloqueou habilidades. O jogador *sente* que está melhorando.

Antes de código, visualize o crescimento:

```
Nível 1:   0 XP
Nível 2:  50 XP (1 × 1 × 50)
Nível 3:  200 XP (2 × 2 × 50), +150 XP necessário
Nível 4:  450 XP (3 × 3 × 50), +250 XP necessário
Nível 5:  800 XP (4 × 4 × 50), +350 XP necessário
Nível 10: 4.950 XP (9 × 9 × 50). A curva sobe rapidamente
```

Fórmula: `xpParaNivel(n) = n² × 50` (quadrática)

Por quê essa fórmula? Porque no início você ganha níveis rápido (diversão imediata: nível 1→2 precisa só 50 XP). Mas depois desacelera exponencialmente (desafio: nível 20 precisa 19,000 XP). É balanceado para *roguelikes*: iniciantes não desistem rapidamente, veteranos têm meta de longo prazo.

TabelaProgressao: A Tabela Mestre

A progressão por XP segue uma fórmula. Você escolhe qual. Aqui usamos quadrática (`n² × 50`), que significa: nível 1 precisa 0 XP, nível 2 precisa 50, nível 3 precisa 200, nível 4 precisa 450. A curva sobe rápido; é difícil alcançar nível 20.

A classe `TabelaProgressao` é a base de todo o sistema. Ela centraliza toda a matemática: cálculo de XP necessário, progresso em percentual, bônus por nível, XP por tipo de inimigo. Todos os números importantes vivem aqui. Isso significa que balancear o jogo é tão simples quanto mudar um número em um único arquivo—nada espalhado, tudo em um lugar.

Por quê design assim? Porque em desenvolvimento profissional, se a progressão estivesse espalhada em 10 classes diferentes, ajustar dificuldade seria um pesadelo. Centralizar dados em uma "tabela mestre" é um padrão universal em game design: databases no Skyrim, spreadsheets na Supercell, tudo segue esse princípio.

```dart
// lib/tabela_progressao.dart

/// Define a curva de experiência e recompensas de nível
class TabelaProgressao {
  /// Fórmula: XP necessário para alcançar um nível
```

```
int xpParaNivel(int nivel) {
  if (nivel <= 1) return 0;
  final n = nivel - 1;
  // ← fórmula quadrática: nível sobe mais caro com tempo
  return n * n * 50;
}

/// XP necessário para ir DO nível atual AO próximo
int xpNecessarioParaProximoNivel(int nivelAtual) {
  final proximoNivel = nivelAtual + 1;
  // ← diferença entre níveis
  return xpParaNivel(proximoNivel) - xpParaNivel(nivelAtual);
}

/// Quanto XP falta (ou quantos pontos você passou)
int xpRestanteParaProximo(int nivelAtual, int xpAtual) {
  final xpDoNivelAtual = xpParaNivel(nivelAtual);
  final xpDoProximo = xpParaNivel(nivelAtual + 1);
  // ← progresso dentro da janela
  final xpNaJanela = xpAtual - xpDoNivelAtual;
  final janelaNecessaria = xpDoProximo - xpDoNivelAtual;
  return janelaNecessaria - xpNaJanela;  // ← quanto resta
}

/// Progresso em percentual (0-100) para próximo nível
int percentualProgresso(int nivelAtual, int xpAtual) {
  if (nivelAtual >= 20) return 100;  // ← cap no nível máximo
  final xpDoNivelAtual = xpParaNivel(nivelAtual);
  final xpDoProximo = xpParaNivel(nivelAtual + 1);
  final xpNaJanela = xpAtual - xpDoNivelAtual;
  final janelaNecessaria = xpDoProximo - xpDoNivelAtual;
  // ← conversão para 0-100%
  return ((xpNaJanela / janelaNecessaria) * 100).toInt();
}

int bonusHPPorNivel() => 10;
```

```
  int bonusAtaquePorNivel() => 2;
  int nivelMaximo() => 20;

  int xpPorInimigo(String tipoInimigo) {
    return switch (tipoInimigo) {
      'zumbi' => 15,          // ← fracos = pouco XP
      'lobo' => 30,           // ← médios = médio XP
      'esqueleto' => 50,      // ← fortes = mais XP
      'orc' => 75,            // ← muito fortes = muito XP
      _ => 10,                // ← padrão seguro
    };
  }
}
```

Saída esperada ao criar TabelaProgressao:

```
// Exemplo de uso:
var tabela = TabelaProgressao();
print(tabela.xpParaNivel(5));  // 800
print(tabela.xpNecessarioParaProximoNivel(3));  // 250
print(tabela.percentualProgresso(2, 150));  // ~75%
print(tabela.xpPorInimigo('lobo'));  // 30
```

Integração com Jogador

Agora estendemos a classe `Jogador` para incluir XP e nível. O jogador não é mais estático; pode ganhar XP, subir de nível, desbloquear habilidades. Isso é crucial: em um *roguelike*, o herói deve *crescer*. Cada *level up* é psicologicamente importante: HP aumenta, ataque aumenta, nova habilidade desbloqueada. O jogador *vê* e *sente* progresso.

O design é simples: quando `ganharXp()` é chamado, o método `verificarNivel()` checa automaticamente se deve haver *level up*. Se sim, aumenta HP máximo (restaurando HP completamente), aumenta ataque, desbloqueia habilidades, e dispara um evento. Tudo em um único lugar bem organizado.

```dart
// lib/jogador.dart

class Jogador extends Entidade {
  int nivel = 1;
  int xp = 0;

  late final TabelaProgressao tabela;
  late final BarramentoEventos<EventoJogo> eventos;

  Jogador({
    required String nome,
    int maxHp = 50,
    int ataque = 5,
  }) : super(nome: nome, hpMax: maxHp, ataque: ataque) {
    tabela = TabelaProgressao();  // ← carrega tabela de progressão
    eventos = BarramentoEventos<EventoJogo>();
  }

  /// Ganha XP e verifica se sobe de nível
  void ganharXp(int quantidade) {
    xp += quantidade;  // ← acumula XP
    print('$nome ganhou $quantidade XP (Total: $xp)');
    verificarNivel();  // ← checa se deve fazer *level up*
  }

  /// Verifica se o jogador deve subir de nível
  void verificarNivel() {
    final proximoNivel = nivel + 1;
    final xpNecessario = tabela.xpParaNivel(proximoNivel);

    // ← loop: pode fazer múltiplos *level ups* em um ganho grande de
    ↪  XP
    while (xp >= xpNecessario && nivel < tabela.nivelMaximo()) {
      nivel++;
      maxHp += tabela.bonusHPPorNivel();
      // ← restaura HP completamente (recompensa psicológica)
```

```
      hp = maxHp;
      ataque += tabela.bonusAtaquePorNivel();

      print('\nLEVEL UP! $nome agora é nível $nivel!');
      final bHp = tabela.bonusHPPorNivel();
      final bAtk = tabela.bonusAtaquePorNivel();
      print('   HP máximo: +$bHp (agora $maxHp)');
      print('   Ataque: +$bAtk (agora $ataque)');
      print('   HP restaurado!\n');

      // ← dispara evento para UI/sistema de logging
      eventos.dispara(EventoNivel(
        nivelAnterior: nivel - 1,
        nivelNovo: nivel,
        bonus: '+${tabela.bonusHPPorNivel()} HP, '
            '+${tabela.bonusAtaquePorNivel()} ATK',
      ));

      // ← pode desbloquear novas habilidades
      _desbloquearHabilidades();
    }
  }

  /// Mostra barra de progresso até próximo nível
  String barraProgresso() {
    final percent = tabela.percentualProgresso(nivel, xp);
    final blocos = (percent / 10).toInt();  // ← 10 blocos no total
    final cheios = '#' * blocos;
    final vazios = '-' * (10 - blocos);
    return '$cheios$vazios $percent%';  // ← ex: "####------ 40%"
  }

  void _desbloquearHabilidades() {
    // Será preenchido em "Parte 5"
  }
}
```

Saída esperada ao ganhar XP:

```
Guerreiro ganhou 30 XP (Total: 80)

Guerreiro ganhou 50 XP (Total: 130)

LEVEL UP! Guerreiro agora é nível 2!
   HP máximo: +10 (agora 60)
   Ataque: +2 (agora 7)
   HP restaurado!
```

Nota técnica: O método `verificarNivel()` usa um loop `while`, não `if`. Por quê? Porque se o jogador ganhar muito XP de uma vez (ex: 500 XP), deve fazer múltiplos *level ups* simultaneamente. O loop garante isso. A condição `xp >= xpNecessario` avalia constantemente até não haver mais níveis a subir.

Sistema de Habilidades

Habilidades são ações especiais que você desbloqueia ao subir de nível. Cada habilidade é uma classe que herda de `Habilidade` (abstrata). Implementa `executar()` que faz algo único: *golpe forte* (2x dano), *curar* (restaura 30% HP), *ataque rápido* (dois ataques).

O padrão *Strategy* é perfeito aqui: cada habilidade é uma estratégia diferente de combate. Você guarda uma lista de habilidades desbloqueadas, e durante o combate, o jogador escolhe qual executar (alternativa a "atacar normalmente"). Isso torna o combate tático: diferentes situações pedem diferentes habilidades.

Design: Cada habilidade conhece seu nível requerido, nome, descrição. No `executar()`, faz algo único. Isto torna fácil adicionar novas habilidades: crie uma classe nova, herde de `Habilidade`, implemente `executar()`. Pronto. Extensibilidade sem modificar código existente.

```
// lib/habilidade.dart

/// Interface abstrata para todas as habilidades
abstract class Habilidade {
```

```
  final String nome;
  final String descricao;
  final int nivelRequerido;

  Habilidade({
    required this.nome,
    required this.descricao,
    required this.nivelRequerido,
  });

  /// Executa a habilidade. Retorna true se bem-sucedida.
  /// ← cada subclasse implementa sua própria lógica
  bool executar(Jogador jogador, {Inimigo? alvo});

  /// Formata para exibição (ex: menu de habilidades)
  String formato() {
    // ← display padronizado
    return '[$nome] (Nív $nivelRequerido) - $descricao';
  }
}

/// Habilidade: Golpe Forte
/// Desbloqueado no nível 3
/// Dano: 2× o ataque normal
/// Dano concentrado num ataque (menos golpes, mais fortes)
class GolpeForte extends Habilidade {
  GolpeForte()
      : super(
        nome: 'Golpe Forte',
        descricao: 'Ataque de 2x dano. Gasta 1 turno.',
        nivelRequerido: 3,
      );

  @override
  bool executar(Jogador jogador, {Inimigo? alvo}) {
    if (alvo == null) return false;
```

```
    // ← 2x dano do ataque normal
    final danoDuplicado = jogador.ataque * 2;
    print('\n${jogador.nome} executa um GOLPE FORTE!');
    print('   Dano: $danoDuplicado');

    return alvo.sofrerDano(danoDuplicado);
  }
}

/// Habilidade: Curar
/// Desbloqueado no nível 5
/// Efeito: +30% do HP máximo
/// Estratégia: defesa (recupera saúde em vez de atacar)
class Curar extends Habilidade {
  Curar()
      : super(
        nome: 'Curar',
        descricao: 'Recupera 30% do HP máximo. Gasta 1 turno.',
        nivelRequerido: 5,
      );

  @override
  bool executar(Jogador jogador, {Inimigo? alvo}) {
    // ← 30% do HP máximo
    final curaQuantidade = (jogador.maxHp * 0.3).toInt();
    final hpAnterior = jogador.hp;
    // ← limita a HP máximo
    jogador.hp = (jogador.hp + curaQuantidade).clamp(0,
↪   jogador.maxHp);
    final curaReal = jogador.hp - hpAnterior;

    print('\n${jogador.nome} invoca CURAR!');
    print('   Recuperou $curaReal HP');

    return true;
```

```
  }
}

/// Habilidade: Ataque Rápido (nível 7)
/// Ataque 2x de 60% cada (total = 120% em um turno)
/// Estratégia: múltiplos ataques (chance de acertar mesmo se um
↪  falhar)
class AtaqueRapido extends Habilidade {
  AtaqueRapido()
      : super(
        nome: 'Ataque Rápido',
        descricao: 'Dois ataques rápidos de 60% cada. Gasta 1 turno.',
        nivelRequerido: 7,
      );

  @override
  bool executar(Jogador jogador, {Inimigo? alvo}) {
    if (alvo == null) return false;

    // ← primeiro ataque (60%)
    final dano1 = (jogador.ataque * 0.6).toInt();
    // ← segundo ataque (60%)
    final dano2 = (jogador.ataque * 0.6).toInt();

    print('\n${jogador.nome} executa ATAQUE RÁPIDO!');
    print('   Golpe 1: $dano1 de dano');
    alvo.sofrerDano(dano1);

    // ← inimigo já morreu, pula golpe 2
    if (!alvo.estaVivo) return true;

    print('   Golpe 2: $dano2 de dano');
    return alvo.sofrerDano(dano2);
  }
}
```

Saída esperada ao executar habilidades:

```
Guerreiro executa um GOLPE FORTE!
   Dano: 14

Guerreiro invoca CURAR!
   Recuperou 18 HP

Guerreiro executa ATAQUE RÁPIDO!
   Golpe 1: 4 de dano
   Golpe 2: 4 de dano
```

Por que não apenas aumentar ataque permanentemente? Porque habilidades criam *momentos emocionantes*. Um *level up* com nova habilidade é mais memorável que +2 de ataque silencioso. Além disso, habilidades criam tática em combate: "Devo curar agora ou tentar matar o inimigo rápido com Golpe Forte?"

Desbloquear Habilidades

De volta ao `Jogador`. Quando você sobe de nível, o método `_desbloquearHabilidades()` é chamado. Se atingiu nível 3 e não tem "Golpe Forte", aprende. E assim por diante. Isto significa habilidades aparecem *naturalmente*, marcando marcos na progressão. O jogador sempre sabe: "No nível 5 desbloqueio Curar".

Esse design é intencional: marcos claros mantêm o jogador engajado. "Faltam 100 XP para nível 5 e a habilidade Curar" é uma meta psicológica poderosa.

```
class Jogador extends Entidade {
  final List<Habilidade> habilidades = [];

  void _desbloquearHabilidades() {
    switch (nivel) {
      case 3:
        // ← nível 3: desbloqueio Golpe Forte
        if (!habilidades.any((h) => h.nome == 'Golpe Forte')) {
          habilidades.add(GolpeForte());
```

```
          print('* Você aprendeu a habilidade: Golpe Forte!');
        }
        break;
      case 5:
        // ← nível 5: desbloqueio Curar
        if (!habilidades.any((h) => h.nome == 'Curar')) {
          habilidades.add(Curar());
          print('* Você aprendeu a habilidade: Curar!');
        }
        break;
      case 7:
        // ← nível 7: desbloqueio Ataque Rápido
        if (!habilidades.any((h) => h.nome == 'Ataque Rápido')) {
          habilidades.add(AtaqueRapido());
          print('* Você aprendeu a habilidade: Ataque Rápido!');
        }
        break;
    }
  }

  void mostrarHabilidades() {
    if (habilidades.isEmpty) {
      print('Nenhuma habilidade desbloqueada ainda.');
      return;
    }

    print('\nHABILIDADES');
    print('—' * 30);
    for (int i = 0; i < habilidades.length; i++) {
      // ← exibe cada habilidade com índice para seleção em combate
      print('[$i] ${habilidades[i].formato()}');
    }
    print('');
  }
}
```

Saída esperada ao subir para nível 3:

```
LEVEL UP! Guerreiro agora é nível 3!
   HP máximo: +10 (agora 70)
   Ataque: +2 (agora 7)
   HP restaurado!
* Você aprendeu a habilidade: Golpe Forte!
```

Nota técnica: O método `habilidades.any()` verifica se uma habilidade com esse nome já existe. Por quê? Para evitar duplicatas. Se o jogador respecificar seu nível (em um cheat, por exemplo), não quer aprender a mesma habilidade duas vezes.

Desafios da Masmorra

Desafio 25.1. A Escalada Interminável. Conforme você sobe de nível, custa cada vez mais. Mude a fórmula: em vez de $n^2 \times 50$, use $n^3 \times 10$ (cúbica). Calcule manualmente: nível 3 custa quanto antes vs depois? (Antes: 200 XP; Depois: 200 XP, coincidência!). A progressão fica muito mais lenta em níveis altos (realista para um *roguelike*). Implemente e teste níveis 1-5: os custos aumentam dramaticamente? Dica: use `n * n * n * 10` no cálculo. Compare as duas fórmulas visualmente em um gráfico.

Desafio 25.2. Três Caminhos do Guerreiro. Você pode treinar para ser recruta (rápido), normal (balanceado), ou veterano (lento mas forte). Crie enum `Dificuldade { recruta, normal, veterano }` e campo em `Jogador`. Modifique `ganharXp()`: recruta ganha 1.5x (treina rápido), normal 1.0x, veterano 0.5x (mas deve ganhar mais estatísticas). Teste: que caminho progride mais rápido? Qual é mais difícil? Dica: multiplicadores revelam trade-offs.

Desafio 25.3. Barra de Progresso Épica. Você quer saber exatamente onde está na progressão. Implemente `mostrarProgressoDetalhado()`: "Nível 4 ████████░░ 80% | 240/300 XP". Calcule quantos blocos cheios versus vazios. Mostre também: (1) XP atual no nível, (2) XP necessário total, (3) percentual. Teste ganhar XP e ver barra crescer. Satisfação visual. Dica: calcule percentual como `(xpAtual / xpProximo) * 100`.

Desafio 25.4. O Paladim Nível 10. Ao atingir nível 10, você desbloqueia uma habilidade especial: "Cura em Grupo". Crie uma classe `CuraEmGrupo extends Habilidade` que cura 50% do HP máximo E reduz dano sofrido em 30% no próximo turno. Implemente: `bool podeExecutar()` (nível >= 10), `void executar()` (aplica cura, marca redutor). Teste: chegue a nível 10, use a habilidade, veja HP restaurado. Dica: sealed classes para habilidades.

Desafio 25.5. (Desafio): Distribuição de Pontos de Poder. Cada level up dá 2 "Pontos de Habilidade". Você investe: (+1 HP por ponto), (+1 Ataque por ponto), (+1 Defesa por 3 pontos = reduz 5% dano). Crie um menu interativo: "Ganhou 2 pontos. Digite: 'hp', 'ataque' ou 'defesa' ". Teste: suba 5 níveis, distribua 10 pontos total, veja suas stats aumentarem diferente baseado em sua escolha. Mestre estrategista. Dica: `pontosHabilidade` é persistente até usar.

Boss Final 25.6. Invencibilidade Temporária. Se você derrotar 5 inimigos seguidos sem sofrer dano, você entra em "Fúria Perfeita" e ganha +50% XP na próxima vitória. Rastreie um `streakSemDano` que incrementa ao vencer sem dano recebido, reseta se sofrer dano. Teste: derrote 5 inimigos limpos (evite dano), vença mais um e ganhe 50% XP extra. Falhe uma vez? Streak reseta. Incentiva jogo agressivo e sem defeitos. Dica: `streakSemDano` é um getter que retorna quantos inimigos consecutivos venceu sem dano.

Comparação: Antes vs. Depois

Antes (Sem Sistema de Progressão)

Cada combate é igual. Você ataca, inimigo ataca. Vitória sente-se sorte. Derrota sente-se injusta. Sem meta, sem *level up*, sem recompensa psicológica.

```
// Inimigo genérico, nunca muda
class Zumbi extends Inimigo {
  Zumbi() : super(nome: 'Zumbi', hpMax: 10, ataque: 2);
}

// Jogador estático
class Jogador extends Entidade {
  // Nada de XP, nada de níveis
}
```

Depois (Com Progressão)

Cada vitória é palpável. Você sobe de nível, ganha habilidades, fica mais forte. Desafios antigos ficam triviais. Novos desafios esperam. Há sempre uma meta.

```
// Jogador dinâmico que cresce
jogador.ganharXp(30);  // Subir de nível? Novo poder desbloqueado?
print(jogador.barraProgresso());  // Vejo meu progresso visualmente
```

Impacto psicológico: Com progressão, o jogador joga "mais uma partida" para alcançar nível 5. Sem progressão, abandona após morte uma.

Pergaminho do Capítulo

Neste capítulo, você aprendeu:

- `TabelaProgressao` com fórmula quadrática cria crescimento natural
- *Level up* aumenta HP máximo, ataque, restaura HP completamente
- Habilidades são classes especializadas que você desbloqueia em marcos (padrão *Strategy*)
- Integração em combate: habilidades são opções estratégicas alternativas a "atacar"
- Escalação: inimigos ficam progressivamente mais fortes (próximos capítulos)
- Event system: notificações limpas quando coisas importantes acontecem

Seu jogo agora tem verdadeira sensação de progresso. Você não é mais um guerreiro estático; é um herói em ascensão que aprende magias, fica mais forte, enfrenta desafios crescentes.

Um sistema de progressão bem balanceado é a diferença entre um jogo que você joga uma vez e um jogo que você não consegue parar de jogar.

Dica Profissional

Dica do Mestre: Balanceamento é iterativo, nunca definitivo. A fórmula que escolhemos (`n² × 50`) é um ponto de partida. Quando começar a testar: meça tempo entre níveis (deve levar ~5 minutos?), registre stats, ajuste constantemente. Mudar para `n² × 40` ou `n² × 60` é trivial. Teste várias versões. O número certo só aparece com testes reais de jogadores. Dados sempre vencem intuição. Uma sheet com as três primeiras fórmulas, testadas por três pessoas diferentes, revela a verdade rapidamente.

Próximo Capítulo

No Capítulo 26 vamos criar múltiplos andares com dificuldade crescente, um boss final épico, e a estrutura de vitória/derrota que torna o jogo uma verdadeira campanha.

Capítulo 26 - Múltiplos Andares e o Boss Final

> *Você desceu profundamente. Os andares anteriores foram testes. Agora, nas profundezas, sente o ar mais pesado. Os inimigos mudam de forma. E no fundo, aguardando, existe algo antigo e poderoso. Não é um goblin aleatório. É o Rei da Masmorra, o boss final. Este capítulo é onde o jogo se torna épico, como Sephiroth em Final Fantasy VII ou Ganondorf em Zelda: múltiplas fases, cada uma mais perigosa.*

O Que Vamos Aprender

Neste capítulo você vai:

- Criar uma arquitetura de múltiplos andares com progressão dinâmica
- Implementar transição de andares (você encontra escada → novo andar gerado)
- Criar distribuições de inimigos e itens específicos por andar
- Implementar a classe `Chefao` que estende `Inimigo` com fases de combate
- Programar um sistema de fase (HP alto = ataque normal; HP médio = *fúria*; HP baixo = *desesperado*)
- Criar interfaces especiais para o combate contra *boss*
- Implementar uma enum `EstadoJogo` para rastrear estado global
- Criar telas de vitória e derrota com estatísticas épicas

Ao final, você terá um *roguelike* com campanha completa: exploração → *boss* → vitória/derrota.

Arquitetura de Múltiplos Andares

Um *roguelike* não é um único andar plano. É uma progressão escalonada: cada andar é mais difícil que o anterior. Inimigos mais fortes, mais HP, itens raros aparecem. Isto mantém o jogo fresco: sempre há novo desafio, nunca fica trivial.

A estrutura de dificuldade por andar é clara e previsível:

```
Andar 0: Iniciação
  ├─ Inimigos: Zumbis apenas (fracos)
  ├─ HP inimigo base: 10
  ├─ Ataque inimigo: 2
  ├─ Itens: Poções de vida
  └─ Objetivo: Aprender mecânicas (tutorial)

Andar 1: Desafio
  ├─ Inimigos: Zumbis + Lobos (mais variedade)
  ├─ HP inimigo base: 15 (+5 bônus)
  ├─ Ataque inimigo: 3 (+1 bônus)

Andar 2: Progressão
  ├─ Inimigos: Lobos + Esqueletos (mais fortes)
  ├─ HP inimigo base: 20 (+10 bônus)
  ├─ Ataque inimigo: 4 (+2 bônus)

Andar 3: Profundidade
  ├─ Inimigos: Esqueletos + Orcs (muito fortes)
  ├─ HP inimigo base: 30 (+20 bônus)
  ├─ Ataque inimigo: 6 (+4 bônus)

Andar 4: Trono do Rei
  ├─ Uma única sala grande (arena épica)
  ├─ Inimigo: *Boss* (Rei da Masmorra)
  ├─ HP: 150-200 (dinâmico conforme seu nível)
  ├─ Fases: Normal → *Fúria* → *Desesperado*
  └─ Vitória = fim do jogo (clímax)
```

Nota de design: Os bônus aumentam linearmente no início (0→5→10→20), depois há um salto grande para o *boss* (20→150). Isto é proposital: recompensa jogadores que chegam ao final, mas não torna impossível. Um jogador nível 5 enfrenta *boss* com ~150 HP (desafio extremo, possível com habilidades).

Classe Chefao: Adversário Épico

O *boss* não é um inimigo normal. É inteligente e adapta-se conforme você o danifica. Tem fases claras: quando tem muita vida, ataca rotineiro. Quando perde 1/3 da vida, entra em *fúria* e ataca 50% mais forte (comportamento agressivo). Quando resta pouco, invoca poder ancestral e tenta *ataque crítico* (comportamento desesperado).

Cada fase muda tática e dano. Isto é psicologicamente importante: o *boss* não é uma bolsa de pancadas estática. Ele reage, adapta, fica perigoso conforme morre. Isto é exatamente como funcionam chefes em *Dark Souls*, *Zelda*, etc. Comportamento emergente mantém o combate tenso e emocionante até o final.

```
// lib/chefao.dart
import 'dart:math';

enum FaseChefao {
  normal,        // HP > 66%: comportamento controlado
  furia,         // 33% < HP <= 66%: agressivo
  desesperado,   // HP <= 33%: caótico e perigoso
}

class Chefao extends Inimigo {
  final Random _rng = Random();
  late int hpMaxOriginal;
  FaseChefao faseAtual = FaseChefao.normal;
  int ataqueBaseOriginal = 0;
  int modificadorDanoFase = 0;
  int turnosNaFase = 0;
  bool usouAtaqueEspecial = false;

  Chefao({
    String nome = 'Rei da Masmorra',
    int hpMax = 150,  // ← *Boss* tem muito HP (desafio real)
    int danoBase = 12,
  }) : super(
    nome: nome,
```

```
    hpMax: hpMax,
    ataque: danoBase,
    descricao: 'O ancião da masmorra. Olhos brilham com malícia.',
  ) {
    hpMaxOriginal = hpMax;
    ataqueBaseOriginal = danoBase;
  }

  /// Atualiza fase do boss baseado em % de HP
  /// ← padrão State implícito: cada fase tem comportamento diferente
  void atualizarFase() {
    final percentualHp = (hp / hpMax) * 100;

    if (percentualHp > 66) {
      // ← Fase 1: Normal (controladoe estável)
      if (faseAtual != FaseChefao.normal) {
        print('├─ O Rei permanece em controle...');
      }
      faseAtual = FaseChefao.normal;
      modificadorDanoFase = 0;
    } else if (percentualHp > 33) {
      // ← Fase 2: Fúria (agressividade aumenta)
      if (faseAtual != FaseChefao.furia) {
        print('\n[FÚRIA] O Rei entra em FÚRIA! Ataques
↪  devastadores!');
        print('   Dano aumentado em 50%!\n');
      }
      faseAtual = FaseChefao.furia;
      // ← +50% dano
      modificadorDanoFase = (ataqueBaseOriginal * 0.5).toInt();
    } else {
      // ← Fase 3: Desesperado (poder final, último recurso)
      if (faseAtual != FaseChefao.desesperado) {
        print('\n[DESESPERADO] O Rei tenta um golpe final!');
        print('   Chance de ataque crítico aumenta!\n');
      }
```

```
      faseAtual = FaseChefao.desesperado;
      // ← +75% dano
      modificadorDanoFase = (ataqueBaseOriginal * 0.75).toInt();
    }

    turnosNaFase++;  // ← rastreia duração em cada fase
  }

  @override
  void executarTurno(Jogador jogador) {
    atualizarFase();  // ← transição automática de fases

    print('\n--- Turno do $nome ---');

    // ← Fase 3 tem ataque especial (40% chance)
    if (faseAtual == FaseChefao.desesperado &&
        !usouAtaqueEspecial &&
        _rng.nextDouble() < 0.4) {
      _ataqueEspecial(jogador);
      usouAtaqueEspecial = true;  // ← usa apenas uma vez
    } else {
      // ← Ataque normal com variação (±15%)
      final dano = ataqueBaseOriginal + modificadorDanoFase;
      final variacao = (dano * 0.15).toInt();
      final danoFinal =
          dano - variacao + _rng.nextInt(variacao * 2 + 1);

      print('> $nome ataca com fúria!');

      if (faseAtual == FaseChefao.normal) {
        print('   (Ataque normal: $danoFinal dano)');
      } else if (faseAtual == FaseChefao.furia) {
        print('   (Ataque furioso: $danoFinal dano!)');
      } else {
        print('   (Ataque desesperado: $danoFinal dano!!!)');
      }
```

```
      jogador.sofrerDano(danoFinal);
    }
  }

  /// Ataque crítico quando em fase desesperada
  void _ataqueEspecial(Jogador jogador) {
    print('\n* O Rei invoca um poder ancestral!');
    print('   > RAIO ANCESTRAL atinge ${jogador.nome}!');

    // ← 2.5x dano crítico
    final danoCritico = (ataqueBaseOriginal * 2.5).toInt();
    jogador.sofrerDano(danoCritico);

    print('   Dano crítico: $danoCritico!');
  }

  String descreverStatus() {
    final percentualHp = (hp / hpMax) * 100;
    final faseTexto = switch (faseAtual) {
      FaseChefao.normal => '[OK] Normal',
      FaseChefao.furia => '[FÚRIA] Fúria (+50% dano)',
      FaseChefao.desesperado => '[CRÍTICO] Desesperado (+crítico)',
    };

    return '''
REI DA MASMORRA
──────────────────────────────────────
HP: $hp / $hpMax (${percentualHp.toStringAsFixed(0)}%)
Fase: $faseTexto
Descrição: $descricao
    ''';
  }

  @override
  String descreverAcao() {
```

```
    return switch (faseAtual) {
      FaseChefao.normal => '$nome respira profundamente.',
      FaseChefao.furia => '$nome RUGE e chamas envolvem a sala!',
      FaseChefao.desesperado =>
        '$nome invoca poder ancestral! O ar se torna tenso!',
    };
  }
}
```

Saída esperada durante combate contra *boss*:

```
--- Turno do Rei da Masmorra ---
> Rei da Masmorra ataca com fúria!
   (Ataque normal: 11 dano)

[FÚRIA] O Rei entra em FÚRIA! Seus ataques se tornam devastadores!
   Dano aumentado em 50%!

--- Turno do Rei da Masmorra ---
> Rei da Masmorra ataca com fúria!
   (Ataque furioso: 17 dano!)

[DESESPERADO] Enfraquecido e DESESPERADO, o Rei tenta um ataque final!
   Chance de ataque crítico aumenta!

* O Rei invoca um poder ancestral!
   > RAIO ANCESTRAL atinge Guerreiro!
   Dano crítico: 30!
```

Nota técnica: O sistema de fases é um padrão State implícito. Cada fase é um estado (`FaseChefao` enum) que muda automaticamente conforme HP cai. No Capítulo 36 (referência), você aprenderá a implementar isso com classes State explícitas, mas aqui é simpler: um enum + `atualizarFase()` que transiciona automaticamente.

Sistema de GameState

O jogo em qualquer momento está num estado bem-definido: explorando, combatendo, na loja, em pausa, morto, vencedor. Esta é a máquina de estados global do jogo. A classe `GerenciadorEstadoJogo` rastreia isto com um enum `EstadoJogo`, permitindo transições claras e até reverter ao estado anterior. Isto é arquiteturalmente importante: você sempre sabe em que "modo" o jogo está, facilita debugging e expansão futura (ex: "se em pausa, não processa movimento").

```
// lib/gerenciador_estado_jogo.dart

/// Estados globais do jogo
enum EstadoJogo {
  menuPrincipal,  // ← tela inicial
  explorando,     // ← dentro de um andar
  combatendo,     // ← em combate ativo
  naLoja,         // ← comprando itens
  subindoNivel,   // ← animação/apresentação de *level up*
  pausado,        // ← jogo congelado
  vitoria,        // ← derrotou o *boss*
  derrota,        // ← herói morreu
}

class GerenciadorEstadoJogo {
  EstadoJogo estadoAtual = EstadoJogo.menuPrincipal;
  EstadoJogo estadoAnterior = EstadoJogo.menuPrincipal;

  /// Transiciona para novo estado, rastreando o anterior
  void mudarPara(EstadoJogo novoEstado) {
    estadoAnterior = estadoAtual;
    estadoAtual = novoEstado;  // ← muda o estado
    print('\n→ Estado: ${estadoAtual.name}');  // ← feedback visual
  }

  /// Volta para estado anterior (ex: despausa)
  void voltarPara() {
```

```
    final temp = estadoAtual;
    estadoAtual = estadoAnterior;
    estadoAnterior = temp;  // ← permite ir e vir
  }

  /// Verifica se está em estado específico
  // ← query simples
  bool em(EstadoJogo estado) => estadoAtual == estado;
}
```

Saída esperada ao gerenciar estados:

```
→ Estado: menuPrincipal
→ Estado: explorando
→ Estado: combatendo
→ Estado: subindoNivel
→ Estado: explorando
```

Por que um gerenciador de estado global? Porque sem isso, o código fica confuso: "Devo processar movimento se em combate?" "Posso pausar durante shop?" Com estados explícitos, tudo é claro. Cada sistema verifica `if (gerenciador.em(EstadoJogo.explorando))` e age apropriadamente.

Progressão de Andares

A classe `GerenciadorAndares` encapsula toda a lógica de dificuldade por andar. Para cada andar (0-4), define: quanto HP extra os inimigos têm, quanto ataque extra ganham, quais tipos aparecem, que itens podem cair, descrição narrativa. Isto permite controle fino da curva de dificuldade.

Design: Centralizar configurações por andar em um único lugar é profissional. Se quiser tornar andar 2 mais fácil, muda um número. Tudo está junto, nada espalhado. Este é o padrão usado em qualquer jogo grande: database de níveis com multiplicadores por dificuldade.

```
// lib/gerenciador_andares.dart

/// Centraliza toda a configuração de dificuldade por andar
class GerenciadorAndares {
  int andarAtual = 0;
  final int andarFinal = 4;  // ← Andar 4 é o *boss*

  /// Retorna configuração do andar (bônus HP/ataque e inimigos)
  /// ← Design: tudo centralizado, fácil ajustar dificuldade
  (int hpBonus, int ataqueBonus, List<String> inimigos)
      configurarAndar(int numero) {
    return switch (numero) {
      // ← Tutorial
      0 => (hpBonus: 0, ataqueBonus: 0, inimigos: ['zumbi']),
      1 => (
        hpBonus: 10,
        ataqueBonus: 2,
        inimigos: ['zumbi', 'lobo'],  // ← Primeira variedade
      ),
      2 => (
        hpBonus: 20,
        ataqueBonus: 4,
        inimigos: ['lobo', 'esqueleto'],  // ← Aumenta dificuldade
      ),
      3 => (
        hpBonus: 35,
        ataqueBonus: 6,
        inimigos: ['esqueleto', 'orc'],  // ← Muito desafiador
      ),
      4 => (
        hpBonus: 60,
        ataqueBonus: 10,
        inimigos: ['chefao'],  // ← *Boss* final
      ),
      _ => (
        hpBonus: 100,
```

```
      ataqueBonus: 15,
      inimigos: ['chefao'],  // ← Fallback
    ),
  };
}

/// Itens que podem cair em cada andar
/// ← Raros aumentam conforme desce (fácil vs veterano)
List<String> itensPorAndar(int numero) {
  return switch (numero) {
    0 => ['pocaoVida', 'pocaoVida'],  // ← Muitas poções (treino)
    // ← Primeira arma rara
    1 => ['pocaoVida', 'pocaoVida', 'espadaFerro'],
    // ← Equipamento melhor
    2 => ['pocaoVida', 'espadaAco', 'escudoAco'],
    // ← Lendário
    3 => ['pocaoVida', 'espadaRunada', 'armaduraPesada'],
    4 => [],  // ← Boss não dropa itens (vitória = prêmio)
    _ => [],
  };
}

/// Narrativa de cada andar (texto descritivo)
String descreverAndar(int numero) {
  return switch (numero) {
    0 => 'Você entra nas masmorras. Ar frio e úmido. Lodo no chão.',
    1 => 'O segundo andar é mais rochoso. Ecos de criaturas.',
    2 => 'Aqui, ossos cobrem o solo. A magia é palpável.',
    3 => 'Este é o andar da perdição. Auras malignas fluem.',
    4 =>
      // ← Épico
      'Câmara colossal. Trono antigo no centro. E nele, ELE.',
    _ => 'Um lugar estranho na masmorra.',
  };
}
```

```
  bool ehAndarDoChefe() => andarAtual == andarFinal;  // ← Query útil
  bool ehUltimoAndar() => andarAtual >= andarFinal;   // ← Query útil
}
```

Por que não usar herança para cada andar? Você *poderia* criar classes `AndarZero extends Andar`, `AndarUm extends Andar`, etc. Mas seria overhead: cada classe teria 5 linhas. Um switch simples é mais leve e lógico aqui. Use herança quando há lógica compartilhada real, não só para dados.

Telas de Vitória e Derrota

Quando o jogo termina, você não quer apenas "FIM". Quer celebração (vitória) ou epitáfio (derrota). A classe `TelaFimJogo` renderiza telas bonitas que mostram suas estatísticas: nível final, turnos vividos, inimigos derrotados, ouro coletado. Isto torna o fim memorável e satisfatório.

Design psicológico: A celebração visual é importante. Quando você derrota um *boss*, merece sentir glória. Um epitáfio é igualmente importante: morrer sem reconhecimento é frustrante. A tela transforma o fim de um jogo em um *momento* — algo que você conta depois.

```
// lib/tela_fim_jogo.dart

/// Renderiza tela final (vitória ou derrota) com estatísticas épicas
class TelaFimJogo {
  final Jogador jogador;
  final int andarAlcancado;
  final int totalTurnos;
  final int totalInimigosDefeitos;
  final int totalOuroColetado;
  final bool vitoria;  // ← determine qual tela mostrar

  TelaFimJogo({
    required this.jogador,
    required this.andarAlcancado,
    required this.totalTurnos,
```

```
    required this.totalInimigosDefeitos,
    required this.totalOuroColetado,
    required this.vitoria,
  });

  /// Exibe tela apropriada baseada em vitória/derrota
  void mostrar() {
    if (vitoria) {
      _mostrarVitoria();  // ← celebração
    } else {
      _mostrarDerrota();  // ← epitáfio
    }
  }

  /// Tela de vitória: celebração épica
  void _mostrarVitoria() {
    print('');
    print('VITÓRIA GLORIOSA!');
    print('');
    print('Você derrotou o Rei da Masmorra e libertou');
    print('o reino das sombras que o enfeitiçavam!');
    print('');
    print('ESTATÍSTICAS FINAIS');
    print('=' * 55);
    print('');
    // ← seu nome é memorizado
    print('Herói:          ${jogador.nome}');
    // ← quantos níveis alcançou?
    print('Nível Final:    ${jogador.nivel}');
    print('HP:             ${jogador.hp}/${jogador.hpMax}');
    print('Ataque:         ${jogador.ataque}');
    print('');
    print('CAMPANHA');  // ← estatísticas gerais
    print('—' * 55);
    print('Andares Explorados:   $andarAlcancado / 5');
    // ← quanto tempo demorou?
```

```
    print('Turnos Totais:        $totalTurnos');
    print('Inimigos Derrotados:  $totalInimigosDefeitos');
    print('Ouro Coletado:        $totalOuroColetado');
    print('');
    print('=' * 55);
    print('');
    print('Parabéns! Você completou Masmorra ASCII!');
    print('Sua lenda será contada nos séculos vindouros.');
    print('');
  }

  /// Tela de derrota: epitáfio respeitoso
  void _mostrarDerrota() {
    print('');
    print('DERROTA AMARGA');
    print('');
    print('Você caiu nas sombras da masmorra, derrotado');
    print('pelas forças que nela habitam.');
    print('');
    print('EPITÁFIO');  // ← homenagem ao herói caído
    print('=' * 55);
    print('');
    print('Aqui jaz ${jogador.nome}');  // ← seu nome é recordado
    print('Um herói de nível ${jogador.nivel}');
    print('');
    print('Caiu no andar $andarAlcancado');  // ← quão longe chegou?
    // ← quantas batalhas?
    print('Derrotou $totalInimigosDefeitos inimigos');
    print('Coletou $totalOuroColetado ouro');
    print('Viveu por $totalTurnos turnos');
    print('');
    print('=' * 55);
    print('');
    print('Nem toda jornada resulta em glória.');
    print('Mas sua tentativa é lembrada.');  // ← dignidade em derrota
    print('');
```

```
  }
}
```

Saída esperada ao vencer:

```
VITÓRIA GLORIOSA!

Você derrotou o Rei da Masmorra e libertou
o reino das sombras que o enfeitiçavam!

ESTATÍSTICAS FINAIS
═══════════════════════════════════════════════

Herói:          Guerreiro
Nível Final:    8
HP:             42/80
Ataque:         15

CAMPANHA
───────────────────────────────────────────────
Andares Explorados:   5 / 5
Turnos Totais:        523
Inimigos Derrotados:  67
Ouro Coletado:        8500

═══════════════════════════════════════════════

Parabéns! Você completou Masmorra ASCII!
Sua lenda será contada nos séculos vindouros.
```

Nota psicológica: A diferença entre "FIM" e uma tela de vitória é enorme. A primeira deixa o jogador vazio ("afinal, qual foi o ponto?"). A segunda deixa o jogador satisfeito ("completei, meus feitos foram reconhecidos"). Detalhes assim transformam um protótipo em um jogo verdadeiro.

Desafios da Masmorra

Desafio 26.1. Fúria do Chefão. O Chefão Antigo entra em fúria quando ferido. Mude suas fases: de 66%/33% de HP para 75%/50% (fica furioso por mais tempo, mais ameaçador). Implemente em `atualizarFase()`. Teste: lute contra o boss, veja quando muda de fase. Sente-se mais desafiador? Dica: números importam na tensão.

Desafio 26.2. Legiões da Sombra. Ao entrar em fúria, o Chefão chama dois espectros: "Invocação de Sombras". Crie dois inimigos sombrios temporários (30% do HP do boss) que atacam ao seu lado. Implemente em `_ataqueEspecial()`. Teste: quando combater o boss na fase 2, dois aliados dele aparecem. Você precisa decidir: mata os espectros ou ataca o boss? Estratégia vital. Dica: use `List<Inimigo>` para gerenciar temporários.

Desafio 26.3. A Arena Final. O boss não aparece num andar procedural aleatório. Implemente `gerarSalaBoss()` que retorna uma única sala grande (80x20) limpa, só com chão. Boss no centro da sala, você spawna perto da entrada. Vasto, árido, épico. Implemente no gerador de andar final. Teste: descida ao boss deve se sentir diferente—solitário, vazio, apenas você vs ele. Dica: preencha com `Tile.chao`, coloque boss em coordenada específica.

Desafio 26.4. O Prêmio da Vitória. Ao derrotar o Chefão, você ganha a "Espada Ancestral Lendária" que aumenta Ataque em +10 permanentemente. Implemente na sequência de vitória: após mensagem de vitória, adicione item ao inventário. Teste: derrote o boss, veja o item aparecer. Você fica significativamente mais forte. Recompensa épica pelo sacrifício. Dica: `Jogador.adicionarItem()` com um objeto especial.

Desafio 26.5. (Desafio): Jogo se Adapta a Você. O jogo aprende de suas deficiências. Cada vitória aumenta dificuldade (+1, máx +5): inimigos 15% mais fortes. Cada derrota reduz (—1, mín —5): inimigos 15% mais fracos. Multiplicador final: `1.0 + (nível × 0.15)`. Isto cria curva de aprendizado: iniciante que morre muito fica em —5 (75% força), veterano vitorioso sobe em +5 (175% força). Teste 10 partidas com diferentes habilidades, veja dificuldade convergir. Dica: salve `nivelDificuldade` junto com stats.

Boss Final 26.6. Troféu de Glória. Na tela de vitória, mostre epopeia completa: (1) Tempo total (em minutos), (2) Ratio vitórias (inimigos derrotados / inimigos encontrados), (3) Andares conquistados, (4) Item mais valioso equipado. Crie uma bela tela ASCII que celebra a vitória com números. Teste: vitória deve ser momento satisfatório com reconhecimento dos seus feitos. Dica: rastreie `tempoInicio`, `inimigosDerrota`, `totalInimigos` durante o jogo.

Comparação: Antes vs. Depois

Antes (Sem Andares)

Tudo é um único andar infinito. Inimigos são sempre iguais. Sem progressão de dificuldade, sem clímax, sem fim definido. Jogar é chato.

```
// Inimigo nunca muda
final inimigo = Zumbi();
inimigo.hp = 10;  // Sempre 10
inimigo.ataque = 2;  // Sempre 2
```

Depois (Com Andares e Boss)

Cada andar é progressivamente mais desafiador. Inimigos escalam. Há um clímax: o *boss*. Vitória é definida. Derrota é significativa. Jogar é emocionante.

```
final config = gerenciador.configurarAndar(3);
final inimigo = Esqueleto(
  hp: 20 + config.hpBonus,  // ← cresce por andar
  ataque: 4 + config.ataqueBonus,
);
```

Pergaminho do Capítulo

Neste capítulo, você aprendeu:

- `GerenciadorAndares` configura dificuldade, inimigos e itens por andar (dados centralizados)
- `Chefao` é um inimigo especial com fases (normal → *fúria* → *desesperado*, padrão State implícito)
- `EstadoJogo` enum rastreia o estado global do jogo (máquina de estados)
- Transição dinâmica: descida de andares com geração procedural
- Telas de *Game Over*: vitória e derrota com estatísticas completas (celebração vs. epitáfio)
- Integração: combate contra *boss* é o clímax de todo o sistema

Seu jogo agora é uma campanha completa: você começa fraco, progride através de 5 andares, enfrenta o chefe e vence ou perde. Isto é um verdadeiro *roguelike*.

A estrutura de andares é o que transforma um protótipo em um produto: há começo, meio, fim, e clímax.

Dica Profissional

Dica do Mestre: Curva de dificuldade é arte, não ciência. Teste com diferentes grupos de jogadores (reais, não só você): iniciantes devem passar no andar 1-2 no primeiro dia (sucesso imediato), intermediários devem chegar ao andar 3-4 em 2-3 dias, veteranos devem alcançar o *boss* em uma sessão. Reúna dados: em que andar a maioria morre? Quanto tempo leva cada andar? O *boss* é muito fácil ou quebra imersão? Cada feedback melhora a próxima iteração. Jogadores reais sempre revelam problemas que você (designer) nunca pensaria.

Próximo Capítulo

No Capítulo 27, vamos integrar tudo em uma versão completa e jogável do *roguelike*, com menu principal, seleção de dificuldade, e a jornada completa pronta para compartilhar.

Capítulo 27 - Dungeon Run Completo: A Jornada Épica

Até agora, você construiu peças: movimento, combate, progressão, andares, um boss. Mas peças isoladas não fazem um jogo. Agora junta tudo numa máquina que funciona. Do menu inicial até a vitória ou morte final. Isto é completude. Isto é um produto jogável, pronto para ser compartilhado. Este capítulo é a celebração de tudo que aprendeu.

O Que Vamos Aprender

Neste capítulo (o pico da Parte IV) você vai:

- Criar a classe `MasmorraAscii`, o orquestrador mestre de todo o jogo
- Implementar o menu principal com ASCII art épico
- Criar seleção de dificuldade (Recruta / Normal / Veterano)
- Integrar criação de personagem (nome + atributos)
- Implementar o *loop* completo: menu → exploração → combate → progressão → andares → *boss* → vitória/derrota
- Demonstrar uma sessão de jogo completa com geração procedural
- Comparar a evolução desde o Capítulo 7 até agora
- Refletir sobre a jornada: do acadêmico ao produto profissional

Ao final, você terá um jogo *roguelike* completo e jogável. Pronto para distribuir, ensinar, expandir.

Menu Principal

Todo jogo profissional começa com um menu. A classe `MenuPrincipal` desenha ASCII art bonita que diz bem-vindo, oferece opções: novo jogo, instruções, créditos, sair. Isto é o rosto do seu jogo. É a primeira impressão: deixa claro o que é, cria atmosfera, estabelece tom.

Por que um menu? Porque sem menu, o jogador entra direto no jogo sem contexto. Com menu, você constrói antecipação. A tela bonita diz:

"Isto é um projeto acabado, profissional, pensado." É psicologia de design: detalhes pequenos mudam percepção gigantemente.

```
// lib/menu_principal.dart
import 'dart:io';

/// Menu principal do jogo
class MenuPrincipal {
  /// Exibe menu e retorna escolha do usuário
  String exibir() {
    _limpar();  // ← limpa tela para sensação de "novo"

    print('''
┘
↪  ╔═══════════════════════════════════════════════════════════════════════╗
╚                                                                       ╚
╚                         MASMORRA ASCII                                ╚
╚                                                                       ╚
╚                     Uma Epopeia *Roguelike* em Dart                   ╚
╚                                                                       ╚

┘
↪  ╠═══════════════════════════════════════════════════════════════════════╣
╚                                                                       ╚
╚   Bem-vindo, aventureiro! Você está prestes a descer numa masmorra
↪  ╚
╚   antiga repleta de perigos, tesouros e poderes esquecidos.
↪  ╚
╚                                                                       ╚
╚   Preparado para a jornada?                                           ╚
╚                                                                       ╚

┘
↪  ╠═══════════════════════════════════════════════════════════════════════╣
╚                            MENU PRINCIPAL                             ╚
╚                                                                       ╚
╚   [1]  Novo Jogo                                                      ╚
╚   [2]  Como Jogar                                                     ╚
```

```
║   [3]  Créditos                                                    ║
║   [0]  Sair                                                        ║
║                                                                    ║
⌋
↪  ╚════════════════════════════════════════════════════════════════════╝

    ''');

    stdout.write('Escolha: ');
    // ← defaulta a sair se entrada inválida
    return stdin.readLineSync() ?? '0';
  }

  /// Limpa tela (visual agradável)
  void _limpar() {
    for (int i = 0; i < 50; i++) {
      print('');
    }
  }

  /// Tela de instruções (disponível no menu)
  static void mostrarComoJogar() {
    print('''
⌋
↪  ╔════════════════════════════════════════════════════════════════════╗
║                      COMO JOGAR                                    ║
⌋
↪  ╠════════════════════════════════════════════════════════════════════╣
║                                                                    ║
║  MOVIMENTO:                                                        ║
║    W  - Norte     S  - Sul     A  - Oeste    D  - Leste            ║
║                                                                    ║
║  AÇÕES:                                                            ║
║    >      - Descer escada                                          ║
║    i      - Inventário                                             ║
║    status - Ver status                                             ║
```

```
    quit   - Abandonar jogo                                          
                                                                     
  PROGRESSÃO:                                                        
    • Derrota inimigos para ganhar XP (*experience points*)
    • Colete itens e ouro                                            
    • Suba de nível para desbloquear habilidades (*level up*)
    • Chegue ao andar 5 e derrote o Rei da Masmorra (*boss fight*)!
                                                                     
  DIFICULDADES:                                                      
    Recruta:   +50% XP, inimigos mais fracos (treino)                
    Normal:    Balanço perfeito (recomendado)                        
    Veterano:  -50% XP, inimigos mais fortes (desafio!)              
                                                                     
                                                                     
                                                                     
    ''');

    stdout.write('Pressione ENTER para voltar ao menu...');
    stdin.readLineSync();
  }

  /// Tela de créditos (disponível no menu)
  static void mostrarCreditos() {
    print('''
                                                                     
                          CRÉDITOS                                   
                                                                     
                                                                     
  MASMORRA ASCII, Uma Epopeia de Aprendizado                         
                                                                     
```

```
║  Desenvolvido com Dart e ensino de programação como foco central
↪  ║
║                                                                       ║
║  SISTEMAS IMPLEMENTADOS:                                              ║
║    • Geração procedural de *dungeon* (algoritmo BSP)                  ║
║    • Sistema completo de combate por turnos                           ║
║    • Progressão com XP e habilidades desbloqueáveis                   ║
║    • 5 andares com dificuldade crescente                              ║
║    • *Boss* final com sistema de fases                                ║
║    • Interface ASCII com barras de saúde                              ║
║    • Sistema de economia (ouro, loja, itens)                          ║
║                                                                       ║
║  CONCEITOS DART ENSINADOS:                                            ║
║    • Programação orientada a objetos (classes, herança)               ║
║    • Polimorfismo e métodos abstratos                                 ║
║    • Sealed classes e enums                                           ║
║    • Generics e type parameters                                       ║
║    • Pattern matching em Dart 3                                       ║
║    • Event systems e padrões de design                                ║
║                                                                       ║
║  DESENVOLVIDO EM: Dart 3.0+                                           ║
║                                                                       ║
║  AGRADECIMENTOS:                                                      ║
║    A todos os aventureiros que jogam, aprendem e criam!               ║
║                                                                       ║
⌋
↪  ╚═══════════════════════════════════════════════════════════════════════╝

    ''');

    stdout.write('Pressione ENTER para voltar ao menu...');
    stdin.readLineSync();
  }
}
```

Seleção de Dificuldade e Criação de Personagem

Antes de começar, você escolhe: quer treino (recruta, +50% XP, inimigos mais fracos), balanço perfeito (normal), ou desafio extremo (veterano, -50% XP, inimigos mais fortes)? Depois, digita o nome do seu herói. Isto personaliza a jornada: é SEU herói, não um genérico.

Por que três dificuldades? Porque diferentes jogadores têm diferentes expectativas. Iniciante quer "aprender sem frustração". Normal quer "balanço fair, recompensa merecida". Veterano quer "arranque-meu-coração, eu gosto de sofrer". Opções de dificuldade respeitam cada um. Isto torna o jogo acessível sem perder desafio.

```
// lib/criacao_personagem.dart
import 'dart:io';

/// Dificuldade do jogo (afeta multiplicadores de XP e HP inimigo)
enum Dificuldade { recruta, normal, veterano }

/// Gerencia criação de personagem e seleção de dificuldade
class CriacaoPersonagem {
  String? nomePersonagem;
  Dificuldade dificuldade = Dificuldade.normal;

  /// Executa fluxo: dificuldade → nome do herói
  void executar() {
    _selecionarDificuldade();  // ← primeiro: escolhe dificuldade
    _criarPersonagem();         // ← segundo: escolhe nome
  }

  /// Menu de dificuldade com descriptions claras
  void _selecionarDificuldade() {
    print('\n╔═══════════════════════════════════════╗');
    print('║    ESCOLHA SEU NÍVEL DE DIFICULDADE    ║');
    print('╠═══════════════════════════════════════╣');
    print('║ [1]  RECRUTA (recomendado iniciante) ║');
    // ← para aprender
    print('║     +50% XP, inimigos -20% saúde      ║');
```

```
  print('║                                          ║');
  print('║ [2]   NORMAL (balanço perfeito)      ║');
  // ← recomendado
  print('║     1x XP, dificuldade média           ║');
  print('║                                          ║');
  print('║ [3]  VETERANO (para desafiadores)    ║');
  // ← para veteranos
  print('║     -50% XP, inimigos +30% saúde      ║');
  print('╚══════════════════════════════════════════╝\n');

  stdout.write('Escolha (1-3): ');
  final escolha = stdin.readLineSync() ?? '2';

  // ← enum switch garante type safety
  dificuldade = switch (escolha) {
    '1' => Dificuldade.recruta,
    '3' => Dificuldade.veterano,
    _ => Dificuldade.normal,  // ← default seguro
  };

  print('\nDificuldade: ${dificuldade.name.toUpperCase()}');
}

/// Menu de criação de personagem (nome personalizado)
void _criarPersonagem() {
  print('\n╔══════════════════════════════════════════╗');
  print('║        CRIE SEU PERSONAGEM             ║');
  print('╚══════════════════════════════════════════╝\n');

  stdout.write('Qual é o nome do seu herói? ');
  // ← padrão se vazio
  nomePersonagem = stdin.readLineSync() ?? 'Aventureiro Sem Nome';

  print('\nBem-vindo, $nomePersonagem!');
  print('Sua jornada na masmorra começa agora...\n');
}
```

```
}
```

Saída esperada ao criar personagem:

```
⌋
↪  ╔═══════════════════════════════════════════════════════════════════════╗
║    ESCOLHA SEU NÍVEL DE DIFICULDADE                                    ║
⌋
↪  ╠═══════════════════════════════════════════════════════════════════════╣
║ [1]  RECRUTA (recomendado iniciante)                                   ║
║     +50% XP, inimigos -20% saúde                                       ║
║                                                                        ║
║ [2]   NORMAL (balanço perfeito)                                        ║
║     1x XP, dificuldade média                                           ║
║                                                                        ║
║ [3]  VETERANO (para desafiadores)                                      ║
║     -50% XP, inimigos +30% saúde                                       ║
⌋
↪  ╚═══════════════════════════════════════════════════════════════════════╝

Escolha (1-3): 2

Dificuldade: NORMAL

⌋
↪  ╔═══════════════════════════════════════════════════════════════════════╗
║        CRIE SEU PERSONAGEM                                             ║
⌋
↪  ╚═══════════════════════════════════════════════════════════════════════╝

Qual é o nome do seu herói? Guerreiro

Bem-vindo, Guerreiro!
Sua jornada na masmorra começa agora...
```

Classe MasmorraAscii: Orquestrador Mestre

A classe `MasmorraAscii` é o pico da pirâmide. Ela orquestra TUDO: menu, criação de personagem, *loop* de exploração, transições de andar, combate, vitória/derrota. É o maestro que coordena 27 capítulos de aprendizado em uma experiência coesa.

Design: MasmorraAscii não implementa lógica de combate, geração de mapa, ou progressão. Ela apenas *orquestra*: chama outras classes, processa entrada, renderiza. Isto é separação de responsabilidades (Single Responsibility Principle): cada classe faz uma coisa bem. MasmorraAscii faz uma coisa: coordenar o fluxo principal.

```
// lib/masmorra_ascii.dart
import 'jogador.dart';
import 'dungeon_multi_andar.dart';
import 'menu_principal.dart';
import 'criacao_personagem.dart';
import 'dart:io';

/// `DungeonMultiAndar`: lógica de múltiplos andares do Cap. 26.
/// Gera andares, gerencia transições e integra com o jogador.
/// Dificuldade cresce por andar (mais HP e itens raros ao descer).

/// A classe maestro que orquestra o jogo inteiro
/// ← Design: não implementa lógica, apenas coordena
class MasmorraAscii {
  late Jogador jogador;
  late DungeonMultiAndar dungeon;
  late MenuPrincipal menu;

  MasmorraAscii() {
    menu = MenuPrincipal();
  }

  /// Executa o jogo completo: menu → criação → exploração → fim
  /// ← Loop infinito: fica no menu até sair ou jogar
  void executar() {
```

```
    while (true) {
      final opcao = menu.exibir();

      switch (opcao) {
        case '1':
          _novoJogo();  // ← inicia jogo completo
          break;
        case '2':
          MenuPrincipal.mostrarComoJogar();  // ← instruções
          break;
        case '3':
          MenuPrincipal.mostrarCreditos();  // ← créditos
          break;
        case '0':
          print('\nObrigado por jogar Masmorra ASCII!');
          exit(0);  // ← sai
        default:
          print('Opção inválida!');
      }
    }
  }

  /// Novo jogo: criação de personagem → exploração
  void _novoJogo() {
    // ← Stage 1: Criação de personagem
    final criacaoPersonagem = CriacaoPersonagem();
    criacaoPersonagem.executar();

    // ← Stage 2: Inicializa jogador com stats base
    jogador = Jogador(
      nome: criacaoPersonagem.nomePersonagem!,
      hpMax: 50,
      ataque: 5,
    );

    final dificuldade = criacaoPersonagem.dificuldade;
```

```
  // ← Stage 3: Gera dungeon com dificuldade selecionada
  dungeon = DungeonMultiAndar(jogador: jogador);

  dungeon.gerarAndar();  // ← gera primeiro andar
  _loopExploracaoPrincipal();  // ← entra no loop principal
}

/// Loop principal de exploração (repete até vitória/derrota/quit)
void _loopExploracaoPrincipal() {
  print('\n${dungeon.gerenciadorAndares.descreverAndar(0)}\n');

  while (true) {
    _renderizar();  // ← desenha mapa + HUD

    stdout.write('> ');
    final comando = stdin.readLineSync() ?? 'nada';

    // ← processa entrada
    final continua = _processarComando(comando);
    if (!continua) break;  // ← sai se quit/morte/vitória
  }
}

/// Renderiza mapa + status do jogador
void _renderizar() {
  dungeon.renderizar();  // ← desenha mapa ASCII
  dungeon.mostrarHud();  // ← mostra HP, nível, XP
}

/// Processa comando do jogador (movimento, ação, etc)
/// ← Retorna false para sair do loop
bool _processarComando(String cmd) {
  final partes = cmd.toLowerCase().split(' ');
  final acao = partes[0];
```

```
    switch (acao) {
      case 'w':  // ← move norte
        dungeon.moverJogador(0, -1);
        return true;
      case 's':  // ← move sul
        dungeon.moverJogador(0, 1);
        return true;
      case 'a':  // ← move oeste
        dungeon.moverJogador(-1, 0);
        return true;
      case 'd':  // ← move leste
        dungeon.moverJogador(1, 0);
        return true;
      case '>':  // ← desce escada
        dungeon.descerAndar();
        return true;
      case 'i':  // ← inventário
        dungeon.mostrarInventario();
        return true;
      case 'status':  // ← status detalhado
        dungeon.mostrarStatus();
        return true;
      case 'quit':  // ← abandona
        print('Você abandona a masmorra cobardemente...');
        return false;  // ← sai do loop
      default:
        print('Comando desconhecido.');
        return true;
    }
  }
}

/// Ponto de entrada: cria jogo e executa
void main() {
  final jogo = MasmorraAscii();
  jogo.executar();  // ← inicia máquina de estados do jogo
```

```
}
```

Fluxo de execução (resumido):

```
main()
  ↓
MasmorraAscii.executar() [loop infinito]
  ↓
menu.exibir()
  ↓
[1] _novoJogo()
    ↓
    CriacaoPersonagem
      ↓ escolhe dificuldade e nome
    Cria Jogador
      ↓
    DungeonMultiAndar.gerarAndar()
      ↓
    _loopExploracaoPrincipal() [loop até morte/vitória]
      ↓
      _renderizar() + _processarComando()
        ↓ repete
```

Nota técnica: O método `main()` é o entry point do Dart. Você executa o programa digitando `dart bin/main.dart` e o Dart chama `main()`. Dentro, você cria a instância de `MasmorraAscii` e chama `executar()`.

Por Que Não...?

Você pode perguntar: "Por que uma classe `MasmorraAscii` centralizadora? Por que não deixar `DungeonMultiAndar` gerenciar tudo?" Resposta técnica e reflexão sobre design:

Por Que Não Sistemas Independentes Sem Coordenador?

Se cada sistema (menu, dungeon, combate, loja) rodasse isoladamente sem coordenação central, você teria:

1. **Transições caóticas:** Como o menu sabe quando começar um jogo? Como o dungeon sabe quando retornar ao menu? Sem coordenador, cada sistema grita para todos. Resulta em código espaguete acoplado.
2. **Estado inconsistente:** Menu tem uma instância de `Jogador`, dungeon tem outra. Duas verdades simultâneas. Bugs surtem do nada porque dados não sincronizam.
3. **Fluxo indefinido:** Um loop no menu, outro no dungeon, outro na loja. Múltiplos eventos rodando concorrentemente sem coordenação = racing conditions e deadlocks.

A solução é o padrão **Coordinator** (ou **Orchestrator**). Uma classe central (`MasmorraAscii`) que conhece cada subsistema e coordena: "Menu, exiba. Jogador clicou '1'. Dungeon, inicie andar 1. Loja, apareça quando entrar em sala especial. Combate, resolva este turno." Uma verdade única, fluxo explícito, estado sincronizado.

Por Que Não Herança Para as Fases do Jogo?

Você poderia modelar: `class MenuPrincipal extends FaseJogo`, `class Exploração extends FaseJogo`, `class CombateBoss extends FaseJogo`. Todos herdando de uma `FaseJogo` base com `executar()` abstrato.

Problema: herança é frágil. Se adicionar uma propriedade ao `FaseJogo` (ex: `bool pausado`), todas as subclasses precisam entender e atualizar. Se quiser compartilhar lógica entre menu e créditos sem compartilhar com dungeon, herança força hierarquias artificiais. Composition é melhor: `MasmorraAscii` **compõe** `MenuPrincipal` e `DungeonMultiAndar` como campos, sem herança. Cada um é independente. Cada um é testável isoladamente. Se um quebra, outro não sofre. Isto é flexibilidade verdadeira.

Por Que Não Máquina de Estados Global?

Um estado gigante `enum GamePhase { menu, exploração, combate, loja, boss, gameover }` com um `switch` gigante processando tudo.

Problema: você teria um método central de 500+ linhas tentando orquestrar tudo. Mudança em um estado afeta dez outros. É impossível entender

sem ler o arquivo inteiro. Divisão de responsabilidades quebra. Cada classe (`MenuPrincipal`, `DungeonMultiAndar`) já é uma máquina de estados interna. `MasmorraAscii` apenas coordena as máquinas, não é uma super-máquina.

A lição: Bom design é sobre coesão (cada classe faz uma coisa bem) e acoplamento baixo (classes não dependem uma da outra). `MasmorraAscii` é o acoplador consciente que conecta peças desacopladas de forma clara.

A Jornada Completa

Você começou no capítulo 7 com um texto-*adventure* simples. Agora você tem:

- Capítulo 8-12: Fundações de OOP, entidades, combate por turnos
- Capítulo 13-17: Mapas procedurais (BSP), campo de visão, UI ASCII
- Capítulo 18-21: Inventário, itens, equipamento, economia
- Capítulo 22-24: Sistema de eventos, loja, transações
- Capítulo 25-27: Progressão (XP/*level up*), habilidades, múltiplos andares, *boss* final

De 27 capítulos, você transformou conceitos de programação acadêmicos em um produto jogável profissional.

Compare com um protótipo:

Capítulo 7: "Um herói caminha num mapa. Clica em inimigo, ambos atacam."

Capítulo 27: "Um herói sobe de nível, desbloqueia habilidades, desce 5 andares cada vez mais difíceis, enfrenta um chefe com 3 fases, e ao vencer, vê uma tela de celebração com suas estatísticas."

Isto é evolução exponencial.

Comparação: Capítulo 7 vs. Capítulo 27

Capítulo 7 (Protótipo)

```
// Simplesmente cria herói e inimigo, combate direto
final jogador = Jogador(hpMax: 20, ataque: 3);
final inimigo = Zumbi(hpMax: 10, ataque: 1);

while (jogador.estaVivo && inimigo.estaVivo) {
```

```
  jogador.atacar(inimigo);
  inimigo.atacar(jogador);
}
```

Resultado: Um combate único. Nenhuma progressão, nenhuma escolha, nenhum objetivo claro.

Capítulo 27 (Produto)

```
final jogo = MasmorraAscii();
jogo.executar();  // Menu → criação → 5 andares → boss →
↪  vitória/derrota
```

Resultado: Uma campanha completa com 27 capítulos de estrutura, fluxo, psicologia de design.

A diferença: Código cresceu não em quantidade, mas em organização. Protótipo é 50 linhas bagunçadas. Produto é 5000 linhas bem estruturadas. Qualidade não vem de código maior; vem de código bem organizado.

Pergaminho do Capítulo

Neste capítulo, você aprendeu:

- `MasmorraAscii` orquestra o jogo inteiro (padrão *Coordinator*)
- Menu principal oferece opções: novo jogo, instruções, créditos (UX)
- `CriacaoPersonagem` gerencia dificuldade e nome do herói (personalização)
- *Loop* principal processa comandos e renderiza a tela (máquina de estados)
- Integração perfeita entre todos os sistemas anteriores (arquitetura)
- Um *roguelike* completo, de menu até vitória/derrota (produto)

Você não apenas aprendeu Dart; você criou um produto real, profissional, jogável, e extensível.

A maior lição: Desenvolvimento de software é 10% código, 90% design. Você aprendeu arquitetura, não apenas sintaxe.

Dica Profissional

Dica do Mestre: De agora em diante, a manutenção é contínua. Reúna feedback de jogadores reais (amigos, comunidades online, fóruns). O que é muito difícil? Muito fácil? Chato? Que tipo de jogador desiste primeiro? Cada feedback é uma oportunidade para melhorar. Iteração rápida (teste → feedback → ajuste) é como você vai de "produto acadêmico" para "jogo verdadeiramente bom". Ouça seus jogadores. Dados sempre vencem intuição, mas feedback qualitativo frequentemente revela a verdade que dados não capturam.

Próximas Etapas

Este é o fim da Parte IV, o fim da jornada do código para o produto. Na Parte V (que está além deste livro), você pode explorar:

- Persistência (salvar/carregar jogo com dados)
- Modos adicionais (*endless mode*, *hard mode*, *survival*)
- Achievements e estatísticas de longa duração
- Multiplayer (quem consegue mais ouro? mais andares?)
- Novas criaturas, itens, habilidades desbloqueáveis
- Refatoração e testes completos (unit tests, integration tests)
- Distribuição como executável (compilar para diferentes plataformas)
- Publicação em repositório (GitHub, para comunidade contribuir)

Mas por agora, você tem um jogo completo. Jogue-o. Mostre aos seus amigos. Aperfeiçoe-o baseado no feedback real. E acima de tudo, celebre o caminho que você trilhou: do "Hello, World" até um *roguelike* funcional com economia, progressão, múltiplos andares, inimigos dinâmicos com IA, e um *boss* épico com fases.

Fechamento

Parabéns! Você não é mais um aprendiz de programação. Você é um desenvolvedor de jogos em Dart que criou um *roguelike* funcional, com economia robusta, progressão escalável, múltiplos inimigos com comportamentos diferentes, um *boss* épico com 3 fases, e uma campanha completa de 5 andares.

O que você aprendeu:

- Arquitetura de software (separação de responsabilidades, padrões de design)
- Programação orientada a objetos em Dart (classes, herança, polimorfismo)
- Estruturas de dados complexas (generics, sealed classes, enums)
- Algoritmos (geração procedural, busca de caminho, campo de visão)
- Design de jogos (progressão, balanceamento, UX, psicologia do jogador)

O que você pode fazer agora:

Pegue um dos 27 desafios propostos ao longo do livro e implemente. Escolha o que te interessa mais. Quer adicionar mais habilidades? Novos inimigos? Um sistema de magia? Um modo endless? Escolhe um desafio, implementa, testa com amigos, recolhe feedback.

A jornada continua. Cada sistema que você construiu pode ser expandido, aperfeiçoado, testado. A aventura nunca termina.

Bem-vindo ao mundo da criação. Que suas masmorras sejam épicas, seus *bosses* memoráveis, e seus aventureiros lendários.

Que a programação seja com você!

O Jogo Até Aqui

Ao final desta parte, seu jogo completo no terminal se parece com isto:

```
MASMORRA ASCII
═══════════════════════════════════════

Uma Epopeia Roguelike em Dart

Bem-vindo, aventureiro! Você está prestes a descer numa masmorra
antiga repleta de perigos, tesouros e poderes esquecidos.

Preparado para a jornada?

MENU PRINCIPAL
═══════════════════════════════════════

  [1]  Novo Jogo
  [2]  Como Jogar
```

```
  [3]  Créditos
  [0]  Sair

Escolha: 1

ESCOLHA SEU NÍVEL DE DIFICULDADE
═══════════════════════════════════════

[1]  RECRUTA (recomendado iniciante)
     +50% XP, inimigos -20% saúde

[2]   NORMAL (balanço perfeito)
     1x XP, dificuldade média

[3]  VETERANO (para desafiadores)
     -50% XP, inimigos +30% saúde

Escolha (1-3): 2

Dificuldade: NORMAL

CRIE SEU PERSONAGEM
═══════════════════════════════════════

Qual é o nome do seu herói? Aventureiro

Bem-vindo, Aventureiro!
Sua jornada na masmorra começa agora...
```

Cada parte adiciona novas camadas ao jogo. Compare com o início e veja o quanto você evoluiu!

Desafios da Masmorra

Desafio 27.1. Seu Reino, Seu Brasão. O menu principal é a porta de entrada para seu mundo. Personalize o ASCII art: adicione seu nome, símbolo único, padrão visual que represente seu jogo. Mude de `MASMORRA`

`ASCII` genérico para algo memorável. Execute, veja a tela inicial—deve inspirar aventura. Dica: use ASCII art criativo, espaçamento interessante, e títulos impactantes para criar identidade visual.

Desafio 27.2. A Descida Sem Fim. Após derrotar o boss no andar 5, você pode escolher: sair com vitória ou continuar descendo. Implemente modo "Endless": andares 6, 7, 8... aparecem com escalação infinita. Boss fica 5% mais forte a cada andar adicional. Opção no menu: "Campanha Clássica (5 andares)" vs "Modo Endless (sem limite)". Teste: vencedor pode ficar até andar 20? Quanto consegue?. Dica: `if (andar > 5) permitirContinuarOuSair()`.

Desafio 27.3. Seus Recordes. O jogo memoriza seus feitos. Implemente um arquivo `stats.txt` que salva **records** globais: maior nível atingido, ouro máximo coletado em uma partida, mais inimigos derrotados, andar mais profundo. Ao iniciar, carregue e mostre no menu: "Seu Recorde: Nível 8, 15000 ouro". Teste: vença 3 partidas diferentes, veja os recordes atualizarem. Dica: serialize `Record` para JSON, salve e carregue com `File`.

Desafio 27.4. Seu Jogo, Suas Regras. Iniciantes podem querer jogo mais fácil. Implemente submenu "Customização": ajuste multiplicadores de HP de inimigos (0.5x até 2.0x), XP ganho (0.5x até 2.0x), preços na loja (0.5x até 2.0x). Mostre preview: "Com essas mudanças, inimigos terão 50% HP". Teste: crie dois savefiles, um fácil (0.5x tudo), um insano (2.0x tudo). Compare dificuldade. Dica: armazene multiplicadores em um objeto `ConfigJogo`.

Desafio 27.5. (Desafio): Corrida Contra o Tempo. Speedrunners amam desafios timeboxed. Implemente "Speedrun Mode": você tem 10 minutos reais para derrotar o boss. Após o limite, o boss fica 10% mais forte a cada minuto. Timer na HUD conta de trás para frente. Teste: consegue vencer em 8 minutos? O que acontece após 10? Dica: use `Stopwatch` para rastrear tempo, atualize força do boss por `1.0 + ((tempoPassado - 600) / 60) * 0.10`.

Boss Final 27.6. Economia Equilibrada. Conforme desce, itens na loja ficam mais caros (fornecedor remoto cobra mais para trazer itens ao fundo). Implemente: preços aumentam 5% por andar (andar 0 = base, andar 5 = +25%). MAS aumente drops também (+5% ouro por andar, ou itens raros aparecem mais). Teste: descida ao andar 5 deve se sentir viável economicamente, não trapaceado. Dica: cálculo final = `precoBase * (1.0 + andar * 0.05)`, depois balance drops.

Próximo Capítulo

Aqui termina a Parte IV. Você tem um jogo jogável de ponta a ponta: criação de personagem, descida por cinco andares de masmorra procedural, combate tático, economia funcional, loja, progressão de nível, boss final, tela de vitória e menu principal. É um roguelike completo, que um amigo pode baixar, rodar e jogar até o fim.

Mas o código que construiu esse jogo cresceu orgânico, capítulo a capítulo, feature a feature. Há arquivos grandes demais, funções que fazem coisa demais, duplicações que se acumularam pelo caminho, e quase nenhum teste automatizado para proteger o que já funciona. O jogo *funciona*. O código ainda não é *profissional*.

Na Parte V — **A Forja do Código** — você vai pegar esse mesmo jogo e passá-lo pelo martelo e pela bigorna da engenharia de software: reorganização modular, testes unitários que travam regressões, `async`/`await` para operações não bloqueantes, e persistência com JSON para salvar e carregar a jornada entre sessões. O jogo não ganhará uma única feature nova visível para o jogador — e, ainda assim, vai sair da forja mais forte, mais leve e pronto para durar.

No próximo capítulo, abrimos a forja.

PARTE V

A FORJA DO CÓDIGO

O código bruto é minério — belo em sua imperfeição, mas frágil. Refatorar é entrar na forja novamente, queimar o supérfluo, temperar o essencial. Testar é provar cada fio contra o peso real do mundo. O que sobrevive ao fogo, ao teste rigoroso, é o código que dura — e toda obra duradoura começa com a coragem de destruir o que já se fez para fazê-lo melhor.

Capítulo 28 - Refatoração Guiada: Code Smells e Limpeza Estrutural

Você para e olha para trás. O jogo funciona, mas o código que o sustenta acumulou cicatrizes de batalha: funções longas demais, nomes confusos, lógica duplicada. Todo projeto real passa por isso e a diferença entre código amador e profissional é o que você faz depois. Nesta parte, você volta pelos andares que conquistou e refatora. Extrai métodos, renomeia variáveis, organiza arquivos em pastas que fazem sentido. Mais que limpar, você vai aprender a proteger seu trabalho. Testes unitários garantem que nada quebre enquanto você melhora. Save e load com JSON preservam o progresso do jogador entre sessões. E no final, o projeto terá a estrutura de um pacote Dart profissional: documentado, testado e organizado. O mesmo jogo, mas agora escrito como alguém que sabe o que está fazendo.

> *Você empurra a porta de uma câmara antiga. O piso está cheio de escombros: pedaços de código que já não fazem sentido, funções gigantescas que ninguém compreende, números mágicos espalhados como moedas podres. Antes de vencer este calabouço, é hora de limpar a casa. Refatorar não é reescrever; melhora a estrutura sem mudar o comportamento.*

Você desceu vários andares. Derrotou zumbis, lobos, orcs. Acumulou ouro e experiência. Mas olhe para trás agora. O código que trouxe você até aqui está cicatrizado; funções gigantescas como dragões antigos, nomes confusos como runas esquecidas, duplicação espalhada como moedas podres pelo chão da masmorra. Isto é *code smell*: sinais de que algo está apodrecendo.

Em qualquer RPG clássico, há um momento em que o herói para na vila antes de descer para o próximo calabouço. Descansa, organiza seu inventário, conserta suas armas, descarta o que não precisa mais. Este capítulo é esse momento para o seu código. Você não vai adicionar novas funcionalidades. Você vai limpar a casa.

Refatoração é um investimento no futuro. Código limpo é código que você (e seus colaboradores) conseguem ler, testar e estender em semanas.

Código sujo vira um calabouço real: cada mudança quebra algo, cada teste falha sem razão aparente, cada novo recurso demora o dobro do tempo.

Reconhecer Code Smells

Code smells não são bugs. São avisos de alerta amarelo. Se você já jogou Dark Souls, sabe aquele cheiro quando entra num local novo? Algo está errado e você não sabe o quê. *Code smells* são assim.

Smell #1: Métodos Gigantescos

Um método com 150 linhas. Imagine refatorar um Pokémon: o poder muda, mas a essência continua. Este método gigante é um Pokémon carregando cinco tipos de ataques diferentes ao mesmo tempo, incapaz de se focar.

Observe como um método que deveria orquestrar o jogo termina fazendo renderização, processamento de entrada, cálculo de movimento e combate tudo junto. Cada uma dessas responsabilidades deveria ser um método separado. Quando você precisa testar "o inimigo se move corretamente", não consegue testar isoladamente pois o método está acoplado ao resto do código.

```
// lib/jogo/dungeonCrawl.dart
// Ruim: executar() faz tudo simultaneamente
void executar() {
  while (true) {
    // Renderizar (20 linhas)
    tela.limpar();
    // ... código de render aqui

    // Processar input (10 linhas)
    final cmd = stdin.readLineSync();
    // ... processamento aqui

    // Mover (15 linhas)
    // ... lógica inteira aqui

    // Combate (30 linhas)
```

```
    // ... tudo junto

    // Salvar (15 linhas)
    // ... mais código aqui
  }
}
```

Problema: você não consegue testar `_moverJogador()` isoladamente. Você não consegue reutilizar a lógica de combate em outro lugar. Você não consegue ler o método em cinco minutos.

Smell #2: Deus Classes

Uma classe que faz tudo. Renderiza, processa entrada, executa combate, gera mundos, gerencia economia, salva dados. É como um personagem de RPG que é mago, guerreiro, ladrão e clérigo simultaneamente.

O código abaixo é aquilo que você quer evitar no seu projeto. Veja como `DungeonCrawl` acumula responsabilidades: uma mudança na renderização quebra a lógica de combate e vice-versa. Não consegue reutilizar a renderização em outro lugar, ou a lógica de combate em um editor de mapa. Cada responsabilidade "compete" com as outras pelo espaço e atenção.

```
// lib/jogo/dungeonCrawl.dart
// Ruim: uma classe com 50 métodos desconexos
class DungeonCrawl {
  // Renderização
  void mostrarMapa() { }
  void mostrarStatus() { }
  void mostrarInventario() { }

  // Lógica de combate
  void moverJogador(Offset d) { }
  void executarCombate(Inimigo e) { }
  void ganharXp(int x) { }

  // Geração
```

```
  void gerarMapa() { }
  void gerarInimigos() { }

  // Persistência
  void salvarJogo() { }
  void carregarJogo() { }

  // ... 30 métodos depois
}
```

Você tira um método para refatorar, e três outros quebram. Você muda o renderizador, e a lógica de combate fica confusa. Cada responsabilidade compete com as outras.

Smell #3: Números Mágicos

Números espalhados pelo código são armadilhas clássicas. Você vê um `17` aqui, um `80` ali, um `5` em outro lugar. Ninguém consegue entender por quê. Foi sorte? Fórmula? Um erro antigo que ninguém tocou? O pior é quando o contexto muda (você aumenta o HP máximo do jogador para 100) e você esquece de atualizar um desses números mágicos em algum lugar; o jogo fica quebrado de forma sutil.

```
// lib/config/constantes.dart (exemplo)
// Ruim: o que significam estes números?
if (jogador.hp < 17) print('crítico!');
if (mapa.largura > 80) { }
for (int i = 0; i < 5; i++) { }
```

Seis meses depois você olha e pensa: “por quê 17? Por quê 80? Por quê 5?” Descobre que 17 era metade de 34, alguém mudou o HP máximo para 50 mas esqueceu de atualizar aqui. Agora o código está quebrado.

Smell #4: Código Duplicado

O código abaixo é a armadilha clássica: você precisa desenhar a mesma linha separadora em três lugares diferentes. Status, inventário, loja. Parece

simples copiar e colar, é verdade. Mas quando você quer mudar o visual (use caracteres diferentes, ou ajuste a largura), você precisa lembrar de todos os três (cinco, dez) lugares. É garantido que você esquecerá um, deixando o jogo visualmente inconsistente.

```
print('─' * 20);
print('Status');
print('─' * 20);

// ... 100 linhas depois
print('─' * 20);
print('Inventário');
print('─' * 20);

// ... 100 linhas depois
print('─' * 20);
print('Loja');
print('─' * 20);
```

Aí você quer mudar a estética. Precisa encontrar todos os três lugares (ou cinco, ou dez). Muda um, esquece dos outros. O jogo ficou feio.

Smell #5: Nomes Ruins

Nomes vagos ou genéricos tornam o código incompreensível. O "x" pode ser coordenada, dano, quantidade de ouro; você não sabe. O "a" pode ser uma lista de itens, inimigos ou qualquer coisa. Seis meses depois, você olha e pensa "o que era isso?" Pior ainda é quando tira esse código para testá-lo isoladamente ou reutilizá-lo em outro lugar: sem contexto, é impossível entender o que cada variável significa.

```
// Ruim: o que é x? o que é a?
int x = 5;
List<String> a = [];
var temp = mapa[0][0];
```

```
// Bom:
int danoBase = 5;
List<String> inimigosNaDungeon = [];
var tilePrincipal = mapa[0][0];
```

Nomes ruins são como uma masmorra sem sinalização; você se perde. Nomes bons são tochas iluminando o caminho.

Extract Method: Quebrando Funções Longas

O método `executar()` é o pior culpado. Vamos extrair responsabilidades em métodos menores.

Antes (Ruim)

```
// lib/jogo/dungeonCrawl.dart
class DungeonCrawl {
  void executar() {
    while (true) {
      // 20 linhas de renderização
      tela.limpar();
      jogador.mostrarStatus();
      mapa.desenhar();

      // 10 linhas de input
      stdout.write('> ');
      final cmd = stdin.readLineSync() ?? 'sair';

      // 30 linhas de lógica
      if (cmd == 'w') {
        final novaPos = jogador.pos + Offset(0, -1);
        if (mapa.estaValido(novaPos)) {
          jogador.pos = novaPos;
        }
      }
```

```
    }
  }
}
```

Não consegue testar `_moverJogador()` separadamente. A lógica está espalhada. Impossível ler.

Depois (Bom)

```dart
// lib/jogo/dungeonCrawl.dart
class DungeonCrawl {
  void executar() {
    while (true) {
      renderizar();
      final comando = processarInput();
      executarComando(comando);
    }
  }

  void renderizar() {
    tela.limpar();
    jogador.mostrarStatus();
    mapa.desenhar();
  }

  Comando processarInput() {
    stdout.write('> ');
    final texto = stdin.readLineSync() ?? 'sair';
    return parser.parse(texto);
  }

  void executarComando(Comando cmd) {
    if (cmd is CmdMover) {
      _moverJogador(cmd.direcao);
    }
```

```
  }

  void _moverJogador(Offset direcao) {
    final nova = jogador.pos + direcao;
    if (!mapa.estaValido(nova)) return;
    jogador.pos = nova;
  }
}
```

Agora `executar()` é legível em um segundo. Cada método faz UMA coisa. Consegue testar `_moverJogador()` isoladamente.

Extract Class: Separando Deus Classes

Antes: Tudo Junto

```
class DungeonCrawl {
  // Renderização
  void mostrarMapa() { }
  void mostrarStatus() { }

  // Lógica de jogo
  void moverJogador(Offset d) { }
  void executarCombate(Inimigo e) { }

  // Geração
  void gerarMapa() { }
  void gerarInimigos() { }
}
```

Depois: Responsabilidades Separadas

Crie pastas temáticas:

```
lib/
  modelos/
    jogador.dart
    inimigo.dart
  ui/
    telaascii.dart
    renderizador.dart
  jogo/
    dungeonCrawl.dart
    loopJogo.dart
  combate/
    combate.dart
```

Exemplo:

```
// lib/ui/renderizador.dart
class Renderizador {
  void mostrarMapa(Jogador j, MapaMasmorra m) {
  void mostrarStatus(Jogador j) { }
}

// lib/combate/combate.dart
class Combate {
  bool executar(Jogador j, Inimigo i) { }
}

// lib/jogo/dungeonCrawl.dart
class DungeonCrawl {
  late Renderizador renderizador;
  late Combate combate;

  void executar() {
    renderizador.mostrarStatus(jogador);
    combate.executar(jogador, inimigo);
  }
```

```
}
```

Cada classe é testável isoladamente.

Replace Magic Numbers com Constants

Antes (Ruim)

```
if (jogador.hp < 17) print('crítico!');
if (mapa.largura > 80) redimensionar();
for (int i = 0; i < 5; i++) tentarGerarMapa();
```

Depois (Bom)

Crie `lib/config/constantes.dart`:

```
// lib/config/constantes.dart
class Constantes {
  // Saúde
  static const int hpMinimoCritico = 17;
  static const int hpMaximoRecuperacao = 50;

  // Mapa
  static const int larguraTelaMax = 80;
  static const int alturaTelaMax = 24;

  // Geração
  static const int tentativasGeracaoMapa = 5;
  static const int inimigosMinimos = 3;
}
```

Uso:

```
if (jogador.hp < Constantes.hpMinimoCritico) {
  print('crítico!');
}

if (mapa.largura > Constantes.larguraTelaMax) {
  redimensionar();
}
```

Mudar um número agora significa mudar num único lugar. E é óbvio o que significa.

Rename Refactoring: Clareza Total

A maioria das IDEs (VSCode, Android Studio) tem “Rename Symbol”:

```
// Antes: o que é 'e'?
for (final e in entidades) {
  e.executarTurno();
}

// Clica em 'e', pressiona F2 (ou Cmd+Shift+R em Mac)
// Digite "inimigoAtual" e pressione Enter

// Depois: perfeitamente claro
for (final inimigoAtual in entidades) {
  inimigoAtual.executarTurno();
}
```

Todos os usos mudam simultaneamente. Zero risco de esquecer um.

Single Responsibility Principle (SRP)

SRP diz: uma classe, uma razão para mudar.

Se `Jogador` faz: - Renderizar HUD - Lógica de movimento - Cálculo de dano - Salvar em JSON

Ela tem 4 razões para mudar. Separar:

```dart
// lib/modelos/jogador.dart
// Apenas dados
class Jogador {
  int hp;
  int ataque;
  Offset pos;
  List<Item> inventario;
}

// lib/ui/renderizadorJogador.dart
// Renderização
class RenderizadorJogador {
  void mostrarStatus(Jogador j) { }
}

// lib/combate/calculadorDano.dart
// Combate
class CalculadorDano {
  int calcular(Jogador j, Inimigo i) { }
}

// lib/persistencia/salvadorJogador.dart
// Save/load
class SalvadorJogador {
  void salvar(Jogador j, String caminho) { }
  Jogador carregar(String caminho) { }
}
```

Agora cada classe tem UMA razão para mudar: - `Jogador` muda quando mudam os atributos - `RenderizadorJogador` muda quando muda a UI - `CalculadorDano` muda quando mudam as regras - `SalvadorJogador` muda quando muda o formato de save

Reorganizar Pastas por Responsabilidade

Antes de ver a estrutura linha a linha, o diagrama abaixo mostra a transformação do projeto de forma visual — a mesma masmorra que funcionava bagunçada agora ganha corredores e salas nomeadas.

```
└──────────────────────────────┘
↪  └──────────────────────────────┘
```

A mesma quantidade de código, a mesma lógica, o mesmo jogo. O que mudou foi a **localização** de cada arquivo — e essa mudança simples transforma a experiência de quem abre o projeto pela primeira vez, de quem vai consertar um bug três meses depois, de quem vai estender o jogo com uma nova feature. Refatorar é reorganizar sem alterar o comportamento.

Antes: Caos

```
lib/
  jogador.dart
  inimigo.dart
  dungeonCrawl.dart
  telaAscii.dart
  item.dart
  combate.dart
  gerador.dart
  (25 arquivos misturados)
```

Depois: Organizado

```
lib/
  modelos/
    jogador.dart
    inimigo.dart
    item.dart
  ui/
    telaAscii.dart
    renderizador.dart
  jogo/
    dungeonCrawl.dart
    loopJogo.dart
```

```
    estadoJogo.dart
  combate/
    combate.dart
    calculadorDano.dart
  mundo/
    mapaMasmorra.dart
    gerador.dart
  config/
    constantes.dart
```

Atualizar Imports

Quando reorganiza pastas, atualiza imports:

```
// Antes (tudo junto em lib/)
import 'jogador.dart';
import 'inimigo.dart';

// Depois (organizado em subpastas de lib/)
import 'modelos/jogador.dart';
import 'modelos/inimigo.dart';
import 'combate/combate.dart';
```

Dentro de `lib/` use imports relativos. Em `test/` use `package:` imports (convenção Dart).

Exemplo Completo: Antes e Depois

Antes: Monolítico

```
// lib/jogo/dungeonCrawl.dart
// 200 linhas, faz tudo
class DungeonCrawl {
  late Jogador jogador;
```

```
late MapaMasmorra mapa;
late List<Inimigo> entidades;

void executar() {
  while (jogador.estaVivo) {
    // Renderizar (20 linhas)
    tela.limpar();
    print('${jogador.nome} HP: ${jogador.hp}');

    // Desenhar mapa (15 linhas)
    for (int y = 0; y < mapa.altura; y++) {
      String linha = '';
      for (int x = 0; x < mapa.largura; x++) {
        if (Offset(x.toDouble(), y.toDouble()) == jogador.pos) {
          linha += '@';
        } else {
          linha += '.';
        }
      }
      print(linha);
    }

    // Input (10 linhas)
    stdout.write('> ');
    final cmd = stdin.readLineSync() ?? 'sair';

    // Executar (50 linhas)
    if (cmd == 'w') {
      final nova = jogador.pos + Offset(0, -1);
      if (nova.dx >= 0 && nova.dx < mapa.largura &&
          nova.dy >= 0 && nova.dy < mapa.altura) {
        jogador.pos = nova;
        // Combate inline aqui também (30 linhas)
        // ... tudo junto e misturado
      }
    } else if (cmd == 'a') {
```

```
        // ... mais movimento
      }
      // ... mais 50 linhas de switch
    }
  }
}
```

Depois: Refatorado

```dart
// lib/jogo/dungeonCrawl.dart
// 40 linhas, orquestra
class DungeonCrawl {
  late Jogador jogador;
  late MapaMasmorra mapa;
  late List<Inimigo> entidades;
  late Renderizador render;

  void executar() {
    while (jogador.estaVivo) {
      render.renderizar(jogador, mapa, entidades);
      final cmd = processarInput();
      executarComando(cmd);
    }
  }

  void executarComando(Comando cmd) {
    if (cmd is CmdMover) {
      _moverJogador(cmd.direcao);
    }
  }

  void _moverJogador(Offset direcao) {
    final nova = jogador.pos + direcao;
    if (!mapa.estaValido(nova)) return;
    jogador.pos = nova;
```

```
    final inimigo = entidades.firstWhereOrNull((e) =>
      e.pos == nova);
    if (inimigo != null) iniciarCombate(inimigo);
  }
}

// lib/ui/renderizador.dart — 30 linhas, renderiza
class Renderizador {
  void renderizar(Jogador j, MapaMasmorra m,
    List<Inimigo> entidades) {
    tela.limpar();
    _renderizarStatus(j);
    _renderizarMapa(m, j, entidades);
  }

  void _renderizarStatus(Jogador j) {
    print('${j.nome} HP: ${j.hp}');
  }

  void _renderizarMapa(MapaMasmorra m, Jogador j,
    List<Inimigo> entidades) {
    for (int y = 0; y < m.altura; y++) {
      String linha = '';
      for (int x = 0; x < m.largura; x++) {
        final pos = Offset(x.toDouble(), y.toDouble());
        if (pos == j.pos) {
          linha += '@';
        } else if (entidades.any((e) => e.pos == pos)) {
          linha += 'E';
        } else {
          linha += '.';
        }
      }
      print(linha);
    }
  }
```

```
}
```

> **Dica:** O método `firstWhereOrNull()` faz parte do `package:collection`. Adicione-o ao `pubspec.yaml`:

```
dependencies:
  collection: ^1.18.0
```

Diferença clara: - DungeonCrawl antes: 200 linhas, 1 arquivo, impossível testar - DungeonCrawl depois: 40 + 30 linhas, 2 arquivos, cada um testável

Executar dart analyze

Depois de refatorar, verifique que nada quebrou:

```
$ cd seu_projeto
$ dart analyze
```

Esperado:

```
No issues found!
```

Se houver erros, corrige os tipos (provavelmente um import foi esquecido):

```
error: The argument type 'String' can't be assigned
to parameter type 'Offset' at ...
```

Desafios da Masmorra

Desafio 28.1. Audit de Saúde. Seu código cresceu. Tempo de diagnóstico. Abra o arquivo principal e identifique problemas: (1) Qual método tem mais

linhas? (2) A classe principal faz quantas coisas? (3) Existem números mágicos soltos (17, 100, 0.5)? (4) Vê código duplicado? Liste 5 problemas. Execute `dart analyze` para autochecar. Dica: transparência é primeiro passo para melhoria.

Desafio 28.2. Cirurgião de Código. Encontre um método com 40+ linhas (ex: `executarTurno()`). Está fazendo demais: renderizar, ler input, combate. Extraia em 3 submétodos: `_renderizarTela()`, `_lerAcaoJogador()`, `_processarAcao()`. Cada responsável por uma coisa. Refatore e teste: jogo deve funcionar igual, mas código fica legível. Dica: retire uma responsabilidade por vez, teste, depois a próxima.

Desafio 28.3. Constantes Nomeadas. Espalhados pelo código estão valores como 20 (HP crítico), 80 (largura tela), 5 (raio visão). Crie `lib/config/constantes.dart`: `const hpMinimoDePerigo = 20`, `const larguraTelaMax = 80`, `const raioVisaoJogador = 5`. Substitua todos. Execute `dart analyze` (sem warnings). Execute jogo (sem mudanças). Agora alterar valores é fácil e centralizado. Dica: constantes documentam intenção.

Desafio 28.4. Organização Profissional. Seu `lib/` é caos—tudo junto. Crie estrutura: `lib/modelos/` (dados), `lib/ui/` (renderização), `lib/jogo/` (loop), `lib/combate/` (batalha), `lib/mundo/` (geração), `lib/config/` (configuração). Mova `Jogador` → modelos, `TelaAscii` → ui, `MapaMasmorra` → mundo, etc. Atualize imports de `'jogador.dart'` para `'package:masmorra/modelos/jogador.dart'`. Execute `dart analyze` (zero erros). Dica: qualidade de vida enormemente melhor.

Boss Final 28.5. Quebra da Deus Classe. Sua classe `Jogador` provavelmente faz 5 coisas: gerencia stats, renderiza, faz combate, salva, carrega. Viola SRP. Quebre em: (1) `JogadorModel` (HP, ataque, nível), (2) `RenderizadorJogador` (desenha barra HP), (3) `LogicaCombateJogador` (calcula dano). Mova métodos apropriados. Atualize `main` para usar 3 classes em lugar de uma. Teste tudo. Código mais limpo = bugs mais fáceis de caçar. Dica: faça um refactor por vez para não enlouquecer.

Próximo Capítulo

No Capítulo 29, protegeremos o código refatorado com testes unitários. Cada classe limpa e focada é agora testável — vamos criar uma suite que garante que nada quebra quando você muda algo.

Pergaminho do Capítulo

Você aprendeu a reconhecer *code smells*: métodos gigantescos, deus classes, números mágicos, código duplicado, nomes ruins. Aprendeu a limpar:

- Extract Method quebra métodos longos
- Extract Class separa responsabilidades
- Replace Magic Numbers torna intenção clara
- SRP garante que cada classe tem uma razão para mudar
- Organização em pastas temáticas torna o projeto navegável
- dart analyze verifica que nada quebrou

Refatoração é investimento no futuro. Código limpo é código que você lê em cinco minutos. Código sujo vira um calabouço real: cada mudança quebra algo, cada teste falha, cada novo recurso demora o dobro.

Um *roguelike* com 30 funcionalidades e código sujo é impossível de manter. Um com 5 funcionalidades e código limpo é uma base sólida para crescer.

Dica Profissional

Refatore incrementalmente, não de uma vez. Se tentar refatorar 50 arquivos simultaneamente:

1. Quebra coisas
2. Fica impossível debugar (muita coisa muda ao mesmo tempo)
3. Demora semanas

Em vez disso:

1. Refatore uma classe de cada vez
2. Execute `dart analyze` após cada mudança
3. Teste que o jogo funciona
4. Faça commit com git: `git commit -m "refactor: extract Renderizador de DungeonCrawl"`

Cada passo é reversível. Um refactor grande demora 2—3 semanas de commits pequenos, mas fica perfeito. Exemplo de sequência:

```
refactor: extract Renderizador
refactor: extract Combate
refactor: move jogador.dart para model/
```

```
refactor: move inimigo.dart para model/
refactor: replace magic numbers com Constantes
refactor: atualizar imports para package:
```

Cada commit é pequeno, testável, reversível.

No próximo capítulo você vai escrever testes unitários com `package:test`. Testes garantem que refatorações não quebraram nada e que comportamentos complexos funcionam como esperado. É a segurança de rede enquanto você dança na corda bamba. Seu código merece proteção.

Capítulo 29 - Testes Unitários com package:test

> *Você desenvolve em silêncio, noite após noite. O jogo funciona. Refatora uma classe. Quebra algo, mas não sabe o quê. Passa horas debugando. Testes são como save points em Dark Souls: você pode errar depois sem perder tudo. Com testes, você refatora com confiança. Sem testes, refatoração é salto de fé.*

Todo aventureiro experiente testa seu equipamento antes de descer para um andar perigoso. Verifica se a espada está afiada, se o escudo não rachou, se as poções não expiraram. **Testes unitários** (por exemplo com `package:test`) são a versão do programador desse ritual. Você escreve uma série de pequenos testes que verificam se cada pedaço do seu código funciona como esperado.

Testes não eliminam todos os bugs (nenhuma masmorra é completamente segura), mas eliminam os piores: aqueles que quebram coisas que funcionavam, aqueles que ignoram casos extremos, aqueles que parecem pequenos mas destroem sua aventura horas depois. A masmorra é escura e cheia de bugs, mas os testes são sua tocha na escuridão. Um jogo com testes é um jogo que você consegue manter, refatorar e expandir durante meses ou anos. Sem testes, cada mudança é um salto no escuro.

Por Que Testar?

Cenário 1: Sem Testes

```
// lib/combate/calculadorDano.dart
class CalculadorDano {
  int calcular(Jogador atacante, Inimigo alvo) {
    return atacante.ataque - alvo.defesa;
  }
```

```
}

// Alguém refatora isto:
int calcular(Jogador atacante, Inimigo alvo) {
  return atacante.ataque + alvo.defesa; // Oops! Operador errado
}

// Ninguém percebe até um jogador reclamar: "Inimigos muito fortes!"
// Você passa 3 horas debugando. Demora 2 minutos para achar.
```

Cenário 2: Com Testes

Com testes automatizados, você detecta o erro instantaneamente. Você escreve um teste que diz: "calculadora deve retornar 7 quando ataco com 10 e defendo com 3". Agora qualquer mudança acidental é flagrada. Não existe "alguém reclamou horas depois". O teste falha nos primeiros segundos, na sua máquina, antes de você fazer commit.

```
// test/combate/calculadorDano_test.dart
void main() {
  test('CalculadorDano: calcular dano simples', () {
    final calc = CalculadorDano();
    final atacante = Jogador(ataque: 10);
    final alvo = Inimigo(defesa: 3);

    final dano = calc.calcular(atacante, alvo);

    expect(dano, equals(7)); // 10 - 3 = 7
  });
}

// Se alguém muda + por -, o teste falha IMEDIATAMENTE:
// $ dart test
// FAILED: dano simples
// Expected: 7
```

```
// Actual: 13
```

Testes apanham erros nos primeiros segundos, não após horas de debugação.

Configurar package:test

Se criou o projeto com `dart create`, `package:test` já está lá:

```
dev_dependencies:
  test: ^1.25.0
```

Se não estiver, adicione:

```
dart pub add --dev test
```

Seu Primeiro Teste

Estrutura básica:

```
// test/exemplo_test.dart
import 'package:test/test.dart';

void main() {
  test('dois mais dois é quatro', () {
    final resultado = 2 + 2;
    expect(resultado, equals(4));
  });
}
```

Execute:

```
$ dart test
```

Saída esperada:

```
test/exemplo_test.dart: dois mais dois é quatro
  ok
```

Organizando Testes em Espelho de lib/

Organize testes como você organiza o código:

```
lib/
  modelos/
    jogador.dart
    inimigo.dart
  combate/
    combate.dart
  jogo/
    parseador.dart

test/
  modelos/
    jogador_test.dart
    inimigo_test.dart
  combate/
    combate_test.dart
  jogo/
    parseador_test.dart
```

Convenção: `lib/combate/combate.dart` → `test/combate/combate_test.dart` (snake_case para arquivos de teste)

Testando uma Classe Simples: Jogador

```
// test/modelos/jogador_test.dart
import 'package:test/test.dart';
import 'package:masmorra_ascii/modelos/jogador.dart';

void main() {
  group('Jogador', () {
    late Jogador jogador;

    setUp(() {
      // Checkpoint: como uma fogueira em Dark Souls, mas sem Hollow.
      // Executado antes de cada teste
      jogador = Jogador(
        nome: 'Aragorn',
        hpMax: 50,
        ataque: 5,
      );
    });

    test('construir jogador com atributos', () {
      expect(jogador.nome, equals('Aragorn'));
      expect(jogador.hpMax, equals(50));
      expect(jogador.ataque, equals(5));
      expect(jogador.estaVivo, isTrue);
    });

    test('sofrer dano reduz HP', () {
      jogador.sofrerDano(10);
      expect(jogador.hpAtual, equals(40));
    });

    test('sofrer dano crítico mata', () {
      jogador.sofrerDano(100);
      expect(jogador.estaVivo, isFalse);
    });
```

```
    test('ganhar XP acumula total', () {
      jogador.ganharXp(50);
      expect(jogador.xp, equals(50));

      jogador.ganharXp(30);
      expect(jogador.xp, equals(80));
    });

    test('não pode ganhar XP negativo', () {
      jogador.ganharXp(-50);
      expect(jogador.xp, equals(0)); // Ignorado
    });
  });
}
```

Execute apenas este teste:

```
$ dart test test/modelos/jogador_test.dart
```

Saída esperada:

```
test/modelos/jogador_test.dart:
  Jogador
    [ok] construir jogador com atributos
    [ok] sofrer dano reduz HP
    [ok] sofrer dano crítico mata
    [ok] ganhar XP acumula total
    [ok] não pode ganhar XP negativo

All tests passed!
```

Matchers: Verificações Poderosas

`expect(atual, matcher)` verifica se `atual` corresponde ao matcher:

```
test('matchers comuns', () {
  // Igualdade
  expect(5, equals(5));
  expect('hello', equals('hello'));

  // Booleanos
  expect(true, isTrue);
  expect(false, isFalse);

  // Nulidade
  expect(null, isNull);
  expect('texto', isNotNull);

  // Tipo
  expect(5, isA<int>());
  expect('texto', isA<String>());

  // Listas
  expect([1, 2, 3], contains(2));
  expect([1, 2, 3], hasLength(3));

  // Exceções
  expect(
    () => throw FormatException('Erro!'),
    throwsA(isA<FormatException>()),
  );

  // Comparações
  expect(5, greaterThan(3));
  expect(2, lessThan(5));

  // Strings
  expect('hello', startsWith('he'));
  expect('hello', endsWith('lo'));

  // Negação
```

```
  expect(5, isNot(equals(3)));
});
```

Testando Combate

```dart
// test/combate/combate_test.dart
import 'package:test/test.dart';
import 'package:masmorra_ascii/modelos/jogador.dart';
import 'package:masmorra_ascii/modelos/inimigo.dart';
import 'package:masmorra_ascii/sistemas/combate.dart';

void main() {
  group('Combate', () {
    late Jogador jogador;
    late Inimigo inimigo;
    late Combate combate;

    setUp(() {
      jogador = Jogador(nome: 'Herói', hpMax: 50, ataque: 10);
      inimigo = Inimigo(nome: 'Goblin', hpMax: 20, ataque: 3);
      combate = Combate(jogador: jogador, inimigo: inimigo);
    });

    test('jogador ataca e causa dano', () {
      final hpAntes = inimigo.hpAtual;
      combate.atacarInimigo();
      expect(inimigo.hpAtual, lessThan(hpAntes));
    });

    test('inimigo morre após dano suficiente', () {
      for (int i = 0; i < 3; i++) {
        combate.atacarInimigo();
      }
      expect(inimigo.estaVivo, isFalse);
    });
```

```
    test('jogador pode defender-se', () {
      final hpAntes = jogador.hpAtual;
      combate.defender();

      combate.ataqueInimigo();
      final danoSofrido = hpAntes - jogador.hpAtual;

      expect(danoSofrido, lessThan(inimigo.ataque));
    });

    test('combate termina quando inimigo morre', () {
      while (inimigo.estaVivo) {
        combate.atacarInimigo();
      }
      expect(combate.terminou, isTrue);
    });

    test('combate termina quando jogador morre', () {
      jogador.sofrerDano(jogador.hpMax - 1);

      for (int i = 0; i < 10; i++) {
        if (jogador.estaVivo) {
          combate.ataqueInimigo();
        }
      }
      expect(jogador.estaVivo, isFalse);
    });
  });
}
```

Testando o Parseador com Diferentes Entradas

```dart
// test/jogo/parseador_test.dart
import 'package:test/test.dart';
import 'package:masmorra_ascii/jogo/parseador.dart';

void main() {
  group('Parseador', () {
    late Parseador parser;

    setUp(() {
      parser = Parseador();
    });

    test('parse movimento w', () {
      final cmd = parser.parse('w');
      expect(cmd, isA<CmdMover>());
    });

    test('parse movimento a', () {
      final cmd = parser.parse('a');
      expect(cmd, isA<CmdMover>());
    });

    test('parse sair', () {
      final cmd = parser.parse('sair');
      expect(cmd, isA<CmdSair>());
    });

    test('parse comando desconhecido', () {
      final cmd = parser.parse('xyz');
      expect(cmd, isA<CmdPadrao>());
    });

    test('parse insensível a maiúsculas', () {
      final cmd1 = parser.parse('W');
```

```
      final cmd2 = parser.parse('w');
      expect(
        cmd1.runtimeType,
        equals(cmd2.runtimeType),
      );
    });

    test('parse com espaços extras', () {
      final cmd = parser.parse('  w  ');
      expect(cmd, isA<CmdMover>());
    });
  });
}
```

Mocks Manuais: Valores Previsíveis

Às vezes você precisa de aleatoriedade **previsível** para testar. Crie "fakes":

```
// test/suporte/aleatorio_falso.dart
import 'dart:math';

class AleatorioFalso implements Random {
  final List<int> valores;
  int _indice = 0;

  AleatorioFalso(this.valores);

  @override
  int nextInt(int max) => valores[_indice++ % valores.length] % max;

  @override
  double nextDouble() => valores[_indice++ % valores.length] / 100;

  @override
  bool nextBool() => valores[_indice++ % valores.length] % 2 == 0;
```

```
  // Métodos abstratos adicionais (implementação mínima)
  @override
  double nextDoubleInRange(double from, double to) {
    return from + (nextDouble() * (to - from));
  }

  @override
  int nextIntInRange(int from, int to) {
    return from + (nextInt(to - from + 1));
  }
}
```

Uso:

```
// test/jogo/lancador_test.dart
import 'package:test/test.dart';
import 'package:masmorra_ascii/jogo/lancador.dart';
import '../suporte/aleatorio_falso.dart';

void main() {
  group('Lancador', () {
    test('d6 com valores previsíveis', () {
      final fake = AleatorioFalso([2, 3, 4]);
      final lancador = Lancador(aleatorio: fake);

      expect(lancador.d6(), equals(3)); // 2 + 1
      expect(lancador.d6(), equals(4)); // 3 + 1
      expect(lancador.d6(), equals(5)); // 4 + 1
    });

    test('d20 máximo', () {
      final fake = AleatorioFalso([19]);
      final lancador = Lancador(aleatorio: fake);
```

```
      expect(lancador.d20(), equals(20));
    });

    test('d20 mínimo', () {
      final fake = AleatorioFalso([0]);
      final lancador = Lancador(aleatorio: fake);

      expect(lancador.d20(), equals(1));
    });
  });
}
```

Testes agora são **determinísticos**: sempre o mesmo resultado.

Testando Distribuição (Em Média)

Às vezes você quer verificar que um sistema funciona corretamente em média:

```
// test/economia/tabelaDrop_test.dart
import 'package:test/test.dart';
import 'package:masmorra_ascii/economia/tabelaDrop.dart';

void main() {
  group('TabelaDrop', () {
    test('ouro distribui corretamente em média', () {
      final tabela = TabelaDrop();
      final drops = <int>[];

      // Rola 100 vezes
      for (int i = 0; i < 100; i++) {
        drops.add(tabela.rolarOuro());
      }

      // Verifica média
```

```
      final media = drops.reduce((a, b) => a + b) ~/ drops.length;
      expect(media, greaterThan(8));
      expect(media, lessThan(12));
    });

    test('loot raro aparece ocasionalmente', () {
      final tabela = TabelaDrop();
      final raridades = <String>[];

      for (int i = 0; i < 1000; i++) {
        raridades.add(tabela.rolarRaridade());
      }

      final rarasCount = raridades.where((r) => r == 'rara').length;
      expect(rarasCount, greaterThan(0));
      expect(rarasCount, lessThan(100));
    });
  });
}
```

Executar Todos os Testes

Execute toda a suite:

```
$ dart test
```

Saída esperada:

```
test/modelos/jogador_test.dart: Jogador
  [ok] construir jogador com atributos
  [ok] sofrer dano reduz HP
  [ok] sofrer dano crítico mata

test/combate/combate_test.dart: Combate
```

```
  [ok] jogador ataca e causa dano
  [ok] inimigo morre após dano suficiente

test/jogo/parseador_test.dart: Parseador
  [ok] parse movimento w
  [ok] parse sair

All tests passed! 15 tests in 0.2s
```

Desafios da Masmorra

Desafio 29.1. Seu Primeiro Escudo. Testes são rede de segurança. Escreva o primeiro teste: uma classe simples como `Item` ou `Arma`. Teste cria instância, verifica atributos com `expect(item.nome, equals('Espada'))`, `expect(item.dano, equals(10))`. Use `group('Item', () { ... })` e `setUp()` para reutilizar. Execute `dart test` e veja verde. Agora você tem confiança de que Item não quebrou. Dica: um teste por funcionalidade.

Desafio 29.2. Defendendo Mochila. Escolha `Inventario` (classe que muda estado). Escreva 5 testes: (1) adicionar item aumenta tamanho, (2) remover diminui, (3) mochila cheia recusa novo item, (4) buscar por nome acha corretamente, (5) usar item (ex: poção) remove do inventário. Use `setUp()` que cria mochila fresca para cada teste. Execute; todos devem passar. Agora refatore `Inventario` com segurança: testes protegem você. Dica: cada teste deve caber em 5-10 linhas.

Desafio 29.3. Erros Esperados. Nem sempre sucesso é erro. Falhas controladas são comportamento. Teste 3 exceções: (1) acessar inventário em índice negativo lança exceção, (2) dividir HP por zero, (3) carregar arquivo inexistente. Use `expect(() => inventario[-1], throwsA(isA<RangeError>()))`. Teste que exceções são lançadas corretamente. Dica: exceções são comportamento de primeira classe que merecem testes.

Desafio 29.4. RNG Determinístico. Testes com `Random` real falham aleatoriamente; inútil. Crie `RandomFalso extends Random` que retorna valores fixos: próximo valor sempre 42, próximo sempre 0.5. Use em testes: com `RandomFalso`, tabelas de drops são previsíveis. Teste que `Rolador.rolar('d6', randomFalso)` sempre retorna mesmo resultado. Dica: fakes tornam testes determinísticos.

Boss Final 29.5. Suite de Defesa. Escolha classe complexa: `Inimigo` ou `Combate`. Escreva 9 testes: (1-5) casos normais (criar, atacar, levar dano,

morrer, saudar), (6-7) extremos (HP 0, dano negativo), (8) exceção (dividir por zero), (9) com fake. Organize em `group()`, use `setUp()` compartilhado, execute `dart test`. Se todos verdes, suite protege você contra regressões. Refatore a classe com confiança. Dica: suite robusta = código que dura.

Pergaminho do Capítulo

Você aprendeu a escrever testes que protegem seu código:

- `test()` para um teste simples
- `group()` para organizar testes relacionados
- `setUp()` para preparar dados antes de cada teste
- Matchers como `equals()`, `isTrue()`, `throws()` para verificações
- Fakes manuais para valores previsíveis
- Organização de testes em espelho de `lib/`
- `dart test` para executar toda a suite

Testes são investimento. Primeiro você escreve mais código (testes + implementação). Mas depois você refatora com confiança, debuga em segundos em vez de horas, e dorme sabendo que o código funciona. Um jogo com 30 funcionalidades e nenhum teste é improvável que seja mantido. Um com 5 funcionalidades e suite completa de testes é sólido.

Dica do Mestre: Escreva testes **antes** de refatorar. Isso é chamado TDD (Test-Driven Development) em sua forma suave:

1. Escreva um teste que falha (Testar você deve)
2. Escreva código mínimo para passar (Não há tentativa de meias-medidas)
3. Refatore com segurança (testes protegem você)

Assim você tem confiança de que refatorações não quebraram nada. Cada teste que passa é um save point. Você pode caminhar pela caverna escura com segurança.

Próximo Capítulo

No Capítulo 30, o jogo ganha dimensão temporal. `async`, `await` e `Stream` permitirão operações assíncronas como leitura de arquivos, delays cinematográficos e sistemas de eventos reativos.

Capítulo 30 - Async, Await e o Tempo na Masmorra

> *Na masmorra, o tempo não para. Enquanto o herói abre um baú, o mundo continua: tochas crepitam, inimigos patrulham, armadilhas rearmam. Se o herói congelasse esperando o baú abrir, seria emboscado. Assincronismo é a arte de fazer coisas sem parar o mundo. Em Dart,* `async` *e* `await` *são os feitiços que permitem isso.*

Até agora, todo o código do jogo executou de forma síncrona: uma instrução após a outra, sem esperar por nada externo. Mas o mundo real é diferente. Salvar um arquivo em disco leva tempo. Ler dados da rede leva tempo. Carregar um save game leva tempo. Se o seu jogo congelar durante essas operações, o jogador vê uma tela morta. E isso é inaceitável.

Neste capítulo você vai aprender os fundamentos da programação assíncrona em Dart: *Future*, *async*, *await*, tratamento de erros assíncronos e *Stream*. No próximo capítulo, você vai aplicar tudo isso para implementar *save*/*load* com *JSON*.

O capítulo está dividido em duas partes. A **Parte A** (da próxima seção até "Tratamento de Erros Assíncronos") foca em `Future` e `async`/`await`: como representar um valor único que chega mais tarde, como esperar por ele sem travar o mundo, e como tratar erros assíncronos com elegância. A **Parte B** (a partir da seção "Stream") muda o foco de um valor único para *muitos* valores ao longo do tempo — *Streams* e o `BusEventos` que será a espinha dorsal do Capítulo 35, quando implementarmos o padrão Observer.

Parte A — Futures e async/await

O Problema: Código que Congela

Imagine que seu jogo precisa salvar o progresso. Todo sistema de jogo moderno enfrenta esse desafio: operações de I/O (entrada/saída) como salvar em disco, carregar da rede ou acessar banco de dados são lentas. Se seu código as executa de forma síncrona, o jogo todo trava até terminar.

A masmorra congela, o jogador vê uma tela morta. Isso é inaceitável em qualquer aplicação interativa.

A versão ingênua faz isso de forma síncrona (bloqueante):

```
// lib/persistencia/salvador.dart
import 'dart:io';

void salvarJogoSync(String dados) {
  final arquivo = File('save.json');
  // ← Bloqueia TUDO até disco terminar
  arquivo.writeAsStringSync(dados);
  print('Salvo!');
}

void main() {
  print('Jogando...');
  salvarJogoSync('{"hp": 42}'); // ← Congela aqui por 50-500ms
  print('Continuando...'); // ← Só executa depois do disco terminar
}
```

Saída esperada:

```
Jogando...
[pausa de ~200ms - programa congelado]
Salvo!
Continuando...
```

O problema? `writeAsStringSync` bloqueia o programa inteiro. Nada mais executa até o disco terminar de escrever. Se o disco for lento, se o arquivo for grande, se o sistema operacional estiver ocupado, o jogador vê o jogo travar. Em uma aplicação com interface gráfica (como Flutter), isso significaria uma tela congelada.

Future: Uma Promessa de Valor

A solução do Dart é o tipo `Future<T>`. Uma *Future* é uma promessa: "vou entregar um valor do tipo `T` eventualmente, mas não agora". É como ir a um restaurante, fazer um pedido, e receber um número ("sua mesa estará pronta em 15 minutos"). O número é a *Future*; você não quer esperar parado, então sai e faz outra coisa. Quando seu número é chamado, a refeição está pronta.

Uma *Future* não bloqueia sua execução. Ela é entregue imediatamente, mesmo que o valor ainda não exista. Dart inicia a operação de I/O em *background* e a resolve quando terminar.

```
// lib/persistencia/leitor.dart
// Future<String> = "prometo entregar uma String no futuro"
Future<String> lerArquivo() {
  return File('dados.txt').readAsString(); // ← Retorna IMEDIATAMENTE
}

void main() {
  final promessa = lerArquivo();
  print(promessa); // Instance of 'Future<String>'
  print('Continuo enquanto arquivo é lido em background!');
}
```

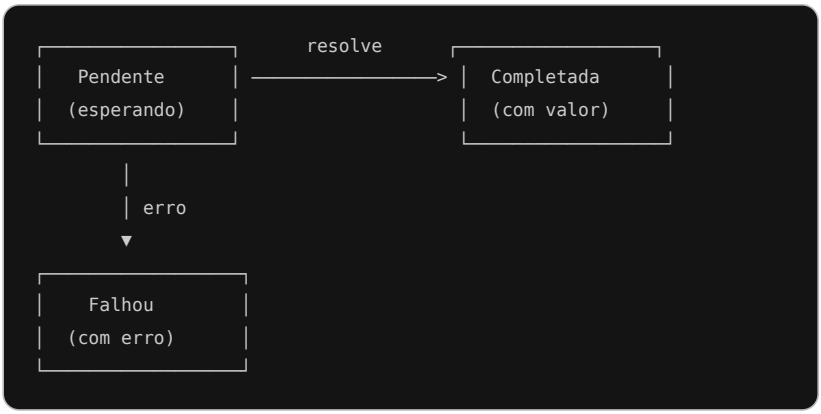
Quando você chama `lerArquivo()`, o retorno é imediato. Mas o valor ainda não existe. O Dart inicia a operação de I/O em *background* e retorna uma *Future* que será resolvida quando a leitura terminar.

Saída esperada:

```
Instance of 'Future<String>'
Continuo enquanto arquivo é lido em background!
```

Três estados de uma Future

Uma Future pode estar em um de três estados:



É como abrir um baú na masmorra: você inicia a ação (pendente), o baú abre e revela um item (completada com valor), ou está trancado e explode (falhou com erro).

async e await: Os Feitiços do Tempo

async: Marca a Função como Assíncrona

A palavra-chave `async` transforma uma função comum em uma função assíncrona. Toda função `async` automaticamente retorna uma `Future`, mesmo que você escreva `return valor`. É como prometer a alguém: "vou fazer isso, mas não tenho a resposta ainda". Dentro de uma função `async`, você pode usar `await` para pausar *apenas essa função*, deixando o resto do programa continuar.

```
// lib/ui/saudacao.dart
// Sem async: retorna String diretamente
String saudacao() {
  return 'Olá, aventureiro!';
}

// Com async: retorna Future<String> automaticamente
Future<String> saudacaoAsync() async {
```

```
  // ← Dart empacota em Future automaticamente
  return 'Olá, aventureiro!';
}

void main() async {
  final msg = await saudacao(); // Erro! String não é Future
  final msg2 = await saudacaoAsync(); // OK! Obtém a String da Future
  print(msg2);
}
```

Uma função `async` sempre retorna uma `Future`. Dentro dela, você pode usar `await`.

Saída esperada:

```
Olá, aventureiro!
```

await: Pausa Sem Congelar

A palavra-chave `await` é o complemento perfeito para `async`. Ela pausa a execução *dessa função* até a *Future* resolver, mas deixa o resto do programa continuar. É a diferença entre congelamento e pausa elegante.

```
// lib/exemplo/async_demo.dart
Future<void> exemploAwait() async {
  print('Início');

  // ← await pausa ESTA função, mas não congela o programa todo
  await Future.delayed(Duration(seconds: 2));

  print('Fim (2 segundos depois)');
}

void main() async {
  exemploAwait(); // Começa, mas não espera!
```

```
  print('Enquanto isso no main...');

  await Future.delayed(Duration(seconds: 3)); // Espera aqui no main
  print('Main terminou');
}
```

`await` é o feitiço que diz: "aguarde esta *Future* resolver, mas deixe o resto do mundo continuar". É a diferença entre o herói parado esperando o baú abrir (síncrono) e o herói fazendo outra coisa enquanto o baú abre sozinho (assíncrono).

Saída esperada:

```
Início
Enquanto isso no main...
Fim (2 segundos depois)
Main terminou
```

Sem await vs. Com await

A diferença entre `await` e não usar `await` é fundamental. Sem `await`, você obtém a *Future* (a promessa), não o valor. Com `await`, você aguarda e obtém o valor real.

```
// lib/exemplo/calculo.dart
Future<int> calcularLento() async {
  await Future.delayed(Duration(seconds: 1));
  return 42;
}

void main() async {
  // SEM await: obtemos a Future (a promessa), não o valor
  final promessa = calcularLento();
  print(promessa); // ← Instance of 'Future<int>'
```

```
  // COM await: obtemos o valor real
  // ← Aguarda 1 segundo, depois obtém 42
  final resultado = await promessa;
  print(resultado); // ← 42
}
```

Saída esperada:

```
Instance of 'Future<int>'
42
```

Encadeando Futures

Operações assíncronas frequentemente dependem umas das outras. Uma leitura de arquivo precisa terminar antes de você poder fazer *parse* do JSON. Com `await`, o encadeamento é natural e legível: cada linha aguarda a anterior, sem *callbacks* aninhadas.

```
// lib/persistencia/carregador.dart
import 'dart:convert';
import 'dart:io';

Future<Jogador> carregarJogador() async {
  // Cada passo aguarda o anterior terminar
  // ← Lê arquivo
  final jsonString = await File('save.json').readAsString();
  // ← Parse JSON
  final mapa = jsonDecode(jsonString) as Map<String, dynamic>;
  final jogador = Jogador.fromJson(mapa); // ← Constrói objeto

  print('Jogador ${jogador.nome} carregado!');
  return jogador;
}
```

```
// Usar:
void main() async {
  final heroi = await carregarJogador();
  print('Bem-vindo, ${heroi.nome}!');
}
```

Saída esperada:

```
Jogador Aragorn carregado!
Bem-vindo, Aragorn!
```

Executando Futures em Paralelo

Às vezes, operações são independentes e podem rodar ao mesmo tempo. Se você as coloca em sequência com `await` individual, elas rodam uma após a outra (sequencial). Isso é ineficiente. A solução é `Future.wait`, que inicia múltiplas *Futures* simultaneamente.

```
// lib/jogo/carregador.dart
Future<void> carregarRecursos() async {
  // RUIM: sequencial (2 segundos total)
  final mapa = await carregarMapa();         // ← 1 segundo
  final inimigos = await carregarInimigos(); // ← +1 segundo = 2 total

  // BOM: paralelo (1 segundo total, ambas rodando juntas)
  final resultados = await Future.wait([
    carregarMapa(),          // ← 1 segundo
    carregarInimigos(),      // ← 1 segundo (ao mesmo tempo)
  ]);
  final mapa = resultados[0] as MapaMasmorra;
  final inimigos = resultados[1] as List<Inimigo>;
}
```

`Future.wait` é como mandar dois exploradores por caminhos diferentes ao mesmo tempo. Ambos caminham em paralelo, e você continua quando o

mais lento terminar. Sem `Future.wait`, você espera o primeiro terminar antes de iniciar o segundo (sequencial). A diferença é exponencial em jogos com muitos recursos.

Saída esperada (tempo real):

```
[sequencial: 2 segundos]
Mapa carregado!
Inimigos carregados!

[paralelo: 1 segundo]
Mapa e inimigos carregados simultaneamente!
```

Tratamento de Erros Assíncronos

try/catch/finally com async

Erros em código assíncrono são capturados da mesma forma que em código síncrono: com `try/catch/finally`. A grande diferença é que você pode `await` dentro de um bloco `try`, e se a *Future* resolver com erro, o `catch` captura imediatamente.

A cláusula `finally` executa **sempre**, independentemente de sucesso ou erro, e é perfeita para limpeza: fechar recursos, liberar memória, atualizar estado. No contexto de jogos, `finally` garante que o estado do jogo não fica corrompido mesmo que uma operação falhe pela metade.

Por que try/catch/finally é essencial em assincronismo: Operações assincronas podem falhar de formas inesperadas. Um arquivo pode estar corrompido, o disco cheio, as permissões negadas. Sem tratamento robusto, essas falhas deixam o jogo em estado indefinido. Com `finally`, você garante que sempre haja limpeza.

```
// lib/persistencia/leitorSeguro.dart
Future<String> lerArquivoSeguro(String caminho) async {
  try {
    final arquivo = File(caminho);
    if (!await arquivo.exists()) {
      throw FileSystemException('Arquivo não encontrado: $caminho');
```

```
    }
    // ← Pode falhar (disco cheio, permissão negada)
    return await arquivo.readAsString();
  } on FileSystemException catch (e) {
    // ← Captura erros de FILE SYSTEM especificamente
    print('Erro de arquivo: ${e.message}');
    return '{}'; // ← Fallback: retorna JSON vazio
  } catch (e) {
    // ← Captura qualquer outro erro
    print('Erro inesperado: $e');
    rethrow; // ← Propaga erros que não sabemos tratar
  } finally {
    // ← SEMPRE executa, sucesso ou erro
    print('Leitura de arquivo concluída (com sucesso ou erro).');
  }
}
```

Saída esperada (arquivo existente):

```
Leitura de arquivo concluída (com sucesso ou erro).
[conteúdo do arquivo]
```

Saída esperada (arquivo não encontrado):

```
Erro de arquivo: Arquivo não encontrado: save.json
Leitura de arquivo concluída (com sucesso ou erro).
{}
```

Exceções Customizadas

Para diferenciar erros específicos do seu jogo, crie classes de exceção customizadas. Uma exceção customizada comunica claramente o tipo de problema. “O arquivo de save está corrompido” é muito diferente de “permissão negada”, e cada um merece tratamento diferente. Com exceções customizadas, seu código sabe exatamente como reagir a cada cenário.

```dart
// lib/excecoes/persistencia.dart
/// Exceção base para erros de persistência
class PersistenciaExcecao implements Exception {
  final String mensagem;
  PersistenciaExcecao(this.mensagem);

  @override
  String toString() => mensagem;
}

/// Arquivo de save corrompido ou incompatível
class ArquivoCorruptidoExcecao extends PersistenciaExcecao {
  ArquivoCorruptidoExcecao(String details)
      : super('Save corrompido: $details');
}

/// Permissão insuficiente para ler/escrever
class PermissaoNegadaExcecao extends PersistenciaExcecao {
  PermissaoNegadaExcecao(String caminho)
      : super('Sem permissão para acessar: $caminho');
}

/// Disco cheio ou sem espaço para salvar
class EspacoDiscoInsuficienteExcecao extends PersistenciaExcecao {
  EspacoDiscoInsuficienteExcecao()
      : super('Espaço em disco insuficiente para salvar o jogo');
}
```

Quando Capturar vs Relançar

A regra é simples: **capture se pode tratar, relance se não pode**. Se você tenta ler um save e o arquivo não existe, isso é um erro tratável (cria um novo jogo). Mas se o disco está cheio, não há salvação no nível local. Relance para a camada superior decidir (talvez avisar o jogador). Essa divisão de responsabilidades mantém seu código limpo: cada camada trata apenas o que consegue resolver.

```
// lib/persistencia/carregadorSave.dart
Future<Jogador> carregarSave(String caminhoSave) async {
  try {
    final arquivo = File(caminhoSave);
    // ← Pode falhar (arquivo não existe)
    final json = await arquivo.readAsString();
    final mapa = jsonDecode(json); // ← Pode falhar (JSON inválido)
    return Jogador.fromJson(mapa); // ← Pode falhar (schema errado)

  } on FileSystemException catch (e) {
    // ← Arquivo não existe ou permissão negada
    if (e.osError?.errorCode == 2) {
      // ← Código 2 = arquivo não encontrado (POSIX, tratável)
      print('Save não encontrado. Iniciando novo jogo...');
      return Jogador.novo(); // ← Fallback: novo jogo
    } else {
      // ← Permissão negada ou outro erro de I/O (não tratável aqui)
      rethrow; // ← Deixa a camada superior tratar
    }

  } on FormatException catch (e) {
    // ← JSON inválido = save corrompido (erro específico)
    throw ArquivoCorruptidoExcecao('JSON inválido: ${e.message}');

  } finally {
    // ← SEMPRE executa
    print('Finalização de carregamento do save.');
  }
}
```

Saída esperada (save encontrado e válido):

```
Finalização de carregamento do save.
[Jogador carregado com sucesso]
```

Saída esperada (save não encontrado):

```
Save não encontrado. Iniciando novo jogo...
Finalização de carregamento do save.
[Novo jogador criado]
```

Exemplo Prático: Salvar e Carregar Save Game

Agora vamos montar um exemplo completo que encadeia carregamento, processamento e salvamento de forma robusta. Este é o padrão que você usará em qualquer jogo: carregar (se existir), jogar, salvar ao sair. Cada etapa tem seu próprio tratamento de erro.

```
// lib/jogo/ciclo.dart
Future<void> executarCicloJogo(String caminhoSave) async {
  Jogador jogador;

  // ← FASE 1: CARREGAR
  try {
    print('Carregando save...');
    jogador = await carregarSave(caminhoSave);
    print('Save carregado: ${jogador.nome} no nível
↪  ${jogador.nivel}.');
  } on ArquivoCorruptidoExcecao catch (e) {
    // ← Arquivo corrompido é tratável
    print('ERRO: $e');
    print('Iniciando novo jogo em vez disso.');
    jogador = Jogador.novo();
  } on PersistenciaExcecao catch (e) {
    // ← Outros erros são críticos
    print('ERRO CRÍTICO: $e');
    print('Não é possível continuar. Encerrando.');
    rethrow;
  }

  // ← FASE 2: JOGAR
  print('Bem-vindo, ${jogador.nome}!');
```

```
  // ... loop de jogo aqui ...

  // ← FASE 3: SALVAR AO SAIR
  try {
    print('Salvando jogo...');
    await salvarJogo(jogador, caminhoSave);
    print('Jogo salvo com sucesso!');
  } on EspacoDiscoInsuficienteExcecao catch (e) {
    // ← Aviso ao jogador
    print('AVISO: $e');
    print('O jogo pode não ter sido salvo completamente.');
  } on PersistenciaExcecao catch (e) {
    // ← Log de erro
    print('ERRO ao salvar: $e');
  }
}

// lib/persistencia/salvador.dart
Future<void> salvarJogo(Jogador jogador, String caminho) async {
  try {
    final arquivo = File(caminho);
    // ← O save é real. Ao contrário do bolo.
    final json = jsonEncode(jogador.toJson());
    await arquivo.writeAsString(json);
  } on FileSystemException catch (e) {
    if (e.osError?.errorCode == 28) {
      // ← Código 28 = No space left on device (disco cheio)
      throw EspacoDiscoInsuficienteExcecao();
    } else if (e.osError?.errorCode == 13) {
      // ← Código 13 = Permission denied (permissão negada)
      throw PermissaoNegadaExcecao(caminho);
    }
    rethrow;
  }
}
```

Saída esperada (sucesso):

```
Carregando save...
Save carregado: Aragorn no nível 5.
Bem-vindo, Aragorn!
[jogo roda...]
Salvando jogo...
Jogo salvo com sucesso!
```

Saída esperada (save corrompido):

```
Carregando save...
ERRO: Save corrompido: JSON inválido
Iniciando novo jogo em vez disso.
Bem-vindo, Novo Herói!
[jogo roda...]
Salvando jogo...
Jogo salvo com sucesso!
```

Observe a estratégia: **carregamento** em um bloco `try` separado com tratamento de erros conhecidos. **Salvamento** em outro bloco que valida condições específicas do SO. Cada fase tem seu próprio escopo de erro. O resultado é um fluxo robusto onde nenhuma situação deixa o jogo em estado indefinido.

Timeouts

Operações assíncronas podem travar infinitamente se algo der errado (rede lenta, disco preso, deadlock). Use `timeout` para limitar a espera. É como dar ao herói um limite de tempo para abrir o baú: se demorar demais, desiste e segue em frente.

```
// lib/persistencia/leitorComTimeout.dart
Future<String> lerComTimeout(String caminho) async {
  try {
    return await File(caminho)
        .readAsString()
```

```
        .timeout(Duration(seconds: 5)); // ← Cancela se demorar >5s
  } on TimeoutException {
    // ← Timeout disparou
    print('Leitura demorou demais!');
    return '{}'; // ← Fallback
  }
}

// Usar:
void main() async {
  final dados = await lerComTimeout('save.json');
  print('Dados: $dados');
}
```

Saída esperada (arquivo rápido):

```
Dados: {"nome": "Herói"}
```

Saída esperada (arquivo muito lento):

```
Leitura demorou demais!
Dados: {}
```

Parte B — Streams e Fluxos de Eventos

Stream: Fluxo Contínuo de Eventos

Uma `Future<T>` entrega um único valor no futuro. Uma `Stream<T>` entrega múltiplos valores ao longo do tempo. É a diferença entre receber uma carta (você abre uma vez) e ouvir rádio (você ouve continuamente).

Por que *Streams* Importam no Jogo:

Mais adiante, no Capítulo 35, quando implementarmos o padrão Observer, *Streams* serão a espinha dorsal: eventos de combate, morte de inimigos,

coleta de itens. Tudo fluindo como uma *Stream* que qualquer sistema pode observar e reagir. Em vez de ter um sistema central que conhece todos os outros, cada sistema se inscreve numa *Stream* e reage independentemente. Desacoplamento total.

```
// lib/eventos/stream_demo.dart
import 'dart:async';

// ← StreamController cria e controla uma Stream
final controlador = StreamController<String>();

// ← Enviar eventos (como um rádio transmitindo)
controlador.add('Jogador atacou!');
controlador.add('Inimigo morreu!');
controlador.add('Item coletado!');

// ← Ouvir eventos (como um receptor sintonizado)
controlador.stream.listen((evento) {
  print('Evento: $evento');
});

controlador.close(); // ← Fecha a stream (limpeza)
```

Saída esperada:

```
Evento: Jogador atacou!
Evento: Inimigo morreu!
Evento: Item coletado!
```

Aplicação: Bus de Eventos do Jogo

Um *bus de eventos* é um padrão de desacoplamento total. Em vez de sistemas chamarem uns aos outros diretamente (acoplamento), todos se comunicam através de eventos numa *Stream*. O sistema de combate publica "Inimigo morreu", o sistema de XP ouve "morte" e reage, o sistema de som ouve "morte" e toca um som. Ninguém conhece ninguém.

```
// lib/eventos/busEventos.dart
import 'dart:async';

/// Tipos de evento do jogo
enum TipoEvento { combate, morte, item, nivel, save }

/// Um evento com tipo e dados
class EventoJogo {
  final TipoEvento tipo;
  final String mensagem;
  final DateTime timestamp;

  EventoJogo(this.tipo, this.mensagem)
      : timestamp = DateTime.now();

  @override
  String toString() => '[$tipo] $mensagem';
}

/// Bus central de eventos: qualquer sistema pode publicar e ouvir
class BusEventos {
  // ← .broadcast() permite múltiplos ouvintes
  final _controlador = StreamController<EventoJogo>.broadcast();
  final List<EventoJogo> _historico = []; // ← Mantém registro de tudo

  /// Stream que qualquer sistema pode ouvir
  Stream<EventoJogo> get eventos => _controlador.stream;

  /// Publica um evento no bus
  void publicar(EventoJogo evento) {
    _historico.add(evento); // ← Registra
    _controlador.add(evento); // ← Transmite
  }

  /// Filtra eventos por tipo (Streams podem ser filtradas!)
  Stream<EventoJogo> filtrar(TipoEvento tipo) {
```

```
    return eventos.where((e) => e.tipo == tipo);
  }

  /// Retorna os últimos [n] eventos do histórico
  List<EventoJogo> ultimosEventos(int n) {
    if (n >= _historico.length) return List.unmodifiable(_historico);
    return List.unmodifiable(_historico.sublist(_historico.length -
    ↪  n));
  }

  /// Libera recursos quando o jogo termina
  void dispose() {
    _controlador.close(); // ← IMPORTANTE: sempre feche Streams!
  }
}
```

Usando o Bus no Jogo

```
// lib/main.dart (ou arquivo de teste)
void main() {
  final bus = BusEventos();

  // ← Sistema de log ouve TODOS os eventos
  bus.eventos.listen((e) => print('[LOG] $e'));

  // ← Sistema de XP ouve apenas mortes
  bus.filtrar(TipoEvento.morte).listen((e) {
    print('[XP] +50 pontos de experiência!');
  });

  // ← Sistema de som ouve apenas combate
  bus.filtrar(TipoEvento.combate).listen((e) {
    print('[SOM] *clang* Espadas se chocam!');
  });
```

```
  // ← Simulação de jogo: apenas publica eventos
  bus.publicar(EventoJogo(TipoEvento.combate, 'Herói ataca Goblin'));
  bus.publicar(EventoJogo(TipoEvento.morte, 'Goblin derrotado'));
  bus.publicar(EventoJogo(TipoEvento.item, 'Poção coletada'));

  bus.dispose(); // ← Limpeza
}
```

Saída esperada:

```
[LOG] [TipoEvento.combate] Herói ataca Goblin
[SOM] *clang* Espadas se chocam!
[LOG] [TipoEvento.morte] Goblin derrotado
[XP] +50 pontos de experiência!
[LOG] [TipoEvento.item] Poção coletada
```

Observe o design: cada sistema ouve apenas o que precisa. O barramento não sabe quem está ouvindo. Os ouvintes não sabem quem publica. Desacoplamento total. O sistema de som não conhece o sistema de XP, e vice-versa. Todos conversam através do bus de forma independente.

Por que não usar Callbacks?

Antes de *async/await*, a forma de lidar com assincronismo era via *callbacks*: passar uma função que seria chamada quando a operação terminasse. Isso resulta em "callback hell" (inferno de callbacks), onde código fica aninhado e ilegível. Veja:

```
// Estilo callback (RUIM - difícil de ler)
lerArquivo('save.json', (json) {
  fazer_parse(json, (dados) {
    validar(dados, (valido) {
      if (valido) {
        salvar(dados, (resultado) {
```

```
          print('Salvo: $resultado');
        });
      }
    });
  });
});

// Estilo async/await (BOM - linear e legível)
try {
  final json = await lerArquivo('save.json');
  final dados = fazer_parse(json);
  if (validar(dados)) {
    final resultado = await salvar(dados);
    print('Salvo: $resultado');
  }
} catch (e) {
  print('Erro: $e');
}
```

Com `async/await`, o código é linear, sequencial e fácil de entender. Com *callbacks*, fica aninhado e difícil de debugar. *Streams* são similares, mas para múltiplos eventos ao longo do tempo.

Resumo dos Conceitos

```
┌──────────────────────────────────────────────────┐
│                 ASYNC EM DART                    │
├──────────────┬───────────────────────────────────┤
│ *Future<T>*  │ Promessa de valor único no futuro │
│ *async*      │ Marca função como assíncrona      │
│ *await*      │ Pausa a função, não o programa    │
│ *Future.wait*│ Executa Futures em paralelo       │
│ *Stream<T>*  │ Fluxo contínuo de valores         │
│ *listen()*   │ Inscreve ouvinte em uma Stream    │
│ *where()*    │ Filtra eventos da Stream          │
```

```
| *broadcast()*  | Stream com múltiplos ouvintes        |
| *try/catch*    | Captura erros assíncronos            |
| *timeout*      | Limita tempo de espera               |
└────────────────┴──────────────────────────────────────┘
```

Dica Profissional

Assincronismo é difícil de debugar. Use estas práticas:

1. **Sempre adicione logging em pontos críticos:** Quando inicia uma operação assíncrona, log. Quando termina, log novamente. Assim você vê onde o programa está preso.
2. **Use *timeouts* generosamente:** Operações que deveriam levar 1 segundo mas levam 10 são sinais de que algo está errado. Sempre adicione `.timeout(Duration(...))`.
3. **Teste com *Future.wait* em paralelo:** Se uma operação trava quando rodando em paralelo, provavelmente há concorrência errada ou compartilhamento de estado.
4. **Streams precisam ser fechadas:** Um `StreamController` aberto forever vaza memória. Sempre chame `.dispose()` quando termina.

Pergaminho do Capítulo

Neste capítulo você aprendeu os fundamentos essenciais da programação assíncrona em Dart que transformam um jogo congelado em uma experiência fluida. Começou com o conceito de `Future<T>`, uma promessa de valor futuro que não bloqueia a execução. Aprendeu que `async` marca uma função como assíncrona (retornando automaticamente uma `Future`) e `await` pausa apenas essa função até a `Future` resolver, permitindo que o resto do programa continue. Dominou `Future.wait` para executar múltiplas operações em paralelo, economizando tempo crítico no carregamento de recursos. Entendeu o tratamento robusto de erros assíncronos com `try/catch/finally`, garantindo que exceções em operações assincronas são capturadas e que limpeza sempre ocorre. Viu como encadear *Futures* mantém código legível sem "callback hell". Finalmente, aprendeu sobre `Stream<T>`, fluxo contínuo de eventos, e implementou um `BusEventos` completo que desacopla sistemas através de um barramento de eventos: log, XP, som, UI e conquistas escutam eventos sem conhecerem um ao outro. Junto com `try/catch` robusto e `timeout`

para operações que travam, você está pronto para persistência segura, rede, e qualquer operação I/O que seus jogos exijam.

Dica do Mestre: Assincronismo é onde muitos programadores iniciantes começam a lutar. O segredo é pensar em *Futures* como "coisas que vão acontecer" não "coisas que estão presas". Uma `Future` é libertadora: você não congela esperando, você continua. E `Stream` é a evolução: em vez de um valor único, múltiplos eventos fluindo indefinidamente. Pratique estas três regras: (1) sempre use `await` se você precisa do valor de uma `Future`, (2) sempre coloque `try/catch` em torno de `await` se a operação pode falhar, (3) sempre feche `Stream` quando termina com `.dispose()` ou `.cancel()`. Siga essas e assincronismo se torna natural.

Desafios da Masmorra

Desafio 30.1. Implemente uma função `carregarArquivoComRetentativa(String caminho, int tentativas)` que tenta ler um arquivo até `tentativas` vezes, aguardando 500ms entre tentativas. Se falhar em todas as tentativas, retorna um fallback vazio.

Desafio 30.2. Crie um `FluxoTempoReal` que emite eventos a cada 100ms (use `Stream.periodic`) durante 5 segundos. Inscreva-se, filtre apenas eventos com valor par, e print cada um.

Desafio 30.3. Implemente `Future.wait` para carregar 3 recursos em paralelo (mapa, inimigos, itens). Cada um retorna após delay variável (1s, 2s, 1.5s). Meça o tempo total e verifique que é o do mais lento, não a soma.

Desafio 30.4. Crie um `BusEventos` com histórico. Quando você pede os últimos N eventos, retorna uma lista. Emita 10 eventos e recupere os últimos 5.

Desafio 30.5. Implemente tratamento de erro assíncrono onde uma operação pode falhar com três exceções diferentes. Use `catch (e)` específico para cada uma, com fallback apropriado para cada tipo.

Desafio 30.6. Crie uma função `correrEmParalelo(List<Future<int>> futures)` que executa todas em paralelo com `Future.wait` e retorna a soma de todos os resultados.

Desafio 30.7. Implemente um `RegistroEventos` que armazena cada evento emitido com timestamp. Adicione método `gerarRelatorio()` que imprime todos os eventos em ordem cronológica com delta de tempo entre eles.

Boss Final 30.8. Monte um sistema de carregamento de jogo completo: (1) Carrega arquivo JSON do save (com retry e timeout), (2) faz parse JSON (com tratamento de erro), (3) cria jogador a partir do JSON, (4) enquanto isso, carrega mapa, inimigos, itens em paralelo com `Future.wait`, (5) quando tudo está pronto, emite `EventoJogoCarregado` que faz log, mostra tela de transição, e toca música. Use `async`, `await`, `Future.wait`, `try/catch`, `timeout`, e um `BusEventos` real.

Você dominou assincronismo. Agora todo recurso pode ser carregado sem congelar o jogo. A masmorra pode ser persistida em disco, na nuvem, transmitida pela rede. Tudo sem travar.

> *"O herói não espera o baú abrir. Enquanto o baú abre, ele explore, ataca, se defende. E quando o baú está pronto, uma notificação o avisa. Assim é assincronismo: o mundo continua."*

Próximo Capítulo

No Capítulo 31, usaremos *async/await* para persistir o estado do jogo em *JSON*. *Save* e *load* transformam a masmorra de uma sessão única numa aventura que o jogador pode retomar a qualquer momento.

Capítulo 31 - Persistência em JSON

Um jogo é um computador que se lembra de você. A primeira sessão aprende. A segunda cresce. A terceira é lendário. Sem persistência, cada partida é amnésia: resetada ao desligar. Com persistência, é uma verdadeira campanha que atravessa semanas.

O que acontece quando o herói precisa parar no meio da masmorra? Seu corpo cansado, seus olhos piscando. Em todo *roguelike* clássico, desde Rogue até NetHack, *save* e *load* são essenciais. Você não pode carregar o jogo na sessão seguinte como se nada tivesse acontecido. Precisa restaurar exatamente onde parou: a posição do herói, a saúde dos seus aliados, o inventário que carrega, o mapa que explorou.

Neste capítulo você vai aplicar o *async*/*await* que aprendeu no Capítulo 30 para salvar o estado do jogo em arquivo *JSON* e carregá-lo depois. A combinação de assincronismo (para não congelar a masmorra enquanto salva) com *serialização* (para converter objetos Dart em texto) é o que torna a persistência possível.

Integração com Capítulo 30: No capítulo anterior, aprendemos como usar `async`/`await` para operações de I/O não bloqueantes. Agora aplicamos esse conhecimento num caso prático: persistência de jogo. Este é um padrão que aparecerá em todos os próximos capítulos que precisam de estado persistente.

dart:io: Lendo e Escrevendo Arquivos

A biblioteca `dart:io` é sua porta para o sistema de arquivos. Com ela, você lê, escreve, cria diretórios e gerencia arquivos. Essas operações são assincronas (não bloqueiam), então combinam perfeitamente com `async`/`await` do Capítulo 30.

Ler um Arquivo

```dart
// lib/persistencia/leitor.dart
import 'dart:io';

Future<String> lerArquivo(String caminho) async {
  final arquivo = File(caminho);

  if (!arquivo.existsSync()) {
    throw FileSystemException('Não encontrado: $caminho');
  }

  return arquivo.readAsString(); // ← Retorna Future (não bloqueia)
}

// Usar:
void main() async {
  try {
    final conteudo = await lerArquivo('dados.json');
    print('Lido: $conteudo');
  } catch (e) {
    print('Erro: $e');
  }
}
```

Saída esperada:

```
Lido: {"nome": "Aragorn", "hp": 42}
```

Escrever um Arquivo

Escrever é similar: crie um `File`, chame `writeAsString()` com `await`, e o arquivo é criado (ou sobrescrito se existir).

```dart
// lib/persistencia/escritor.dart
Future<void> escreverArquivo(String caminho, String conteudo) async {
  final arquivo = File(caminho);
  await arquivo.writeAsString(conteudo); // ← Escreve assincronamente
  print('Escrito: $caminho');
}

// Usar:
void main() async {
  await escreverArquivo('dados.json', '{"nome": "Herói"}');
  print('Arquivo criado!');
}
```

Saída esperada:

```
Escrito: dados.json
Arquivo criado!
```

Criar Diretórios

Antes de salvar, você precisa garantir que o diretório existe. Use `Directory` para isso. Observe que `createSync()` é síncrono (criação de diretório é rápido), mas para ser consistente com `async/await`, considere usar `create()` com `await`.

```dart
// lib/persistencia/gerenciadorDiretorios.dart
import 'dart:io';

Future<void> criarDiretorios() async {
  final dir = Directory('salves');

  if (!dir.existsSync()) {
    // ← Cria se não existir (rápido, ok síncrono)
    dir.createSync(recursive: true);
```

```
  }
}

// Ou de forma assíncrona:
Future<void> criarDiretoriosAsync() async {
  final dir = Directory('salves');

  if (!await dir.exists()) {
    await dir.create(recursive: true); // ← Forma assíncrona
  }
}
```

dart:convert: JSON

A biblioteca `dart:convert` fornece ferramentas para *serializar* (converter objetos em texto) e *desserializar* (converter texto em objetos). *JSON* é o formato universal: leve, legível e suportado por todas as linguagens.

Por que JSON é melhor que alternativas: - *CSV*: Difícil com dados aninhados - *XML*: Verboso, mais lento para parsear - *Binary* (protobuf, etc): Mais rápido, mas menos legível e debugável - *JSON*: Balanço perfeito: legível, estruturado, rápido

Converter Dart para JSON

```
// lib/serializacao/exemplo.dart
import 'dart:convert';

final dados = {
  'nome': 'Aragorn',
  'hp': 45,
  'ataque': 7,
  'inventario': ['espada', 'poção'],
};

// ← Converter para JSON string
```

```
final jsonString = jsonEncode(dados);
print(jsonString);
```

Saída esperada:

```
{"nome":"Aragorn","hp":45,"ataque":7,"inventario":["espada","poção"]}
```

Observe que `jsonEncode()` converte estruturas Dart em texto puro. Números, strings e listas são preservados. Você pode salvar `jsonString` num arquivo.

Converter JSON para Dart

O inverso: leia um *JSON* string e converta para estrutura Dart.

```
// lib/serializacao/decodificador.dart
import 'dart:convert';

final jsonString = '{"nome":"Aragorn","hp":45}';

final dados = jsonDecode(jsonString); // ← Converte string para Map
print(dados['nome']); // ← "Aragorn"
print(dados['hp']); // ← 45
print(dados.runtimeType); // ← Map<String, dynamic>
```

Saída esperada:

```
Aragorn
45
_InternalLinkedHashMap<String, dynamic>
```

Note que `jsonDecode()` retorna um `Map<String, dynamic>`: dinâmico porque pode conter qualquer tipo de valor.

Tratamento de Erros

Três erros comuns ao trabalhar com *JSON*:

```
// Erro 1: JSON malformado
try {
  jsonDecode('{"nome": "Hero"'); // ← Falta fechar
} catch (e) {
  print('Parse error: $e'); // ← FormatException
}

// Erro 2: Tipo seguro (sempre faça cast)
final dados = jsonDecode('{"numero": 42}');
final resultado = dados['numero'] as int; // ← Safe cast (valida tipo)

// Erro 3: Chave inexistente
final valor = dados['chaveQueNaoExiste']; // ← null
// ← null coalescing
final valor2 = dados['chaveQueNaoExiste'] ?? 'padrão';
```

Saída esperada:

```
Parse error: FormatException: Unexpected end of input (at character
↪  15)
null
padrão
```

Sempre use `try/catch` ao fazer *parse* de *JSON*, pois dados corrompidos lançam `FormatException`.

Padrão toJson() / fromJson()

Este é o padrão de ouro em Dart para *serialização*: toda classe que precisa ser salva tem dois métodos: - `toJson()`: Converte a instância em `Map<String, dynamic>` (pronta para `jsonEncode()`) - `fromJson()`: Factory que reconstrói a instância de um `Map<String, dynamic>`

É simples, poderoso e reutilizável.

Jogador: Serialização Completa

Vamos serializar um `Jogador` completo: atributos básicos, inventário (lista de itens), e posição. Observe como `toJson()` também serializa objetos aninhados (`inventario`, `posicao`), criando uma estrutura *JSON* profunda que `jsonEncode()` consegue transformar em string.

```
// lib/modelos/jogador.dart
class Jogador {
  String nome;
  int hpMax;
  int hpAtual;
  int ataque;
  int nivel;
  int xp;
  List<Item> inventario;
  Offset posicao;

  Jogador({
    required this.nome,
    required this.hpMax,
    this.ataque = 5,
    this.nivel = 1,
    this.xp = 0,
    this.inventario = const [],
    this.posicao = const Offset(0, 0),
  }) {
    hpAtual = hpMax;
  }

  // ← Converter para JSON (estrutura para arquivo)
  Map<String, dynamic> toJson() {
    return {
      'nome': nome,
      'hpMax': hpMax,
```

```
      'hpAtual': hpAtual,
      'ataque': ataque,
      'nivel': nivel,
      'xp': xp,
      // ← Items também serializam
      'inventario': inventario.map((i) => i.toJson()).toList(),
      'posicao': {
        'x': posicao.dx,
        'y': posicao.dy,
      },
    };
  }

  // ← Converter de JSON (reconstruir do arquivo)
  factory Jogador.fromJson(Map<String, dynamic> map) {
    return Jogador(
      nome: map['nome'] as String,
      hpMax: map['hpMax'] as int,
      ataque: map['ataque'] as int,
      nivel: map['nivel'] as int,
      xp: map['xp'] as int,
      inventario: (map['inventario'] as List)
          .map((i) => Item.fromJson(i as Map<String, dynamic>))
          .toList(), // ← Items também desserializam
      posicao: Offset(
        (map['posicao']['x'] as num).toDouble(),
        (map['posicao']['y'] as num).toDouble(),
      ),
    );
  }
}
```

Saída esperada (após toJson()):

```
{
  "nome": "Aragorn",
  "hpMax": 100,
  "hpAtual": 100,
  "ataque": 8,
  "nivel": 5,
  "xp": 1250,
  "inventario": [
    {"nome": "Espada de Elendil", "quantidade": 1},
    {"nome": "Poção de Cura", "quantidade": 3}
  ],
  "posicao": {"x": 10, "y": 15}
}
```

Item: Serialização Simples

Items são mais simples que jogadores. Observe que `toJson()` pode ser uma arrow function se for curta.

```
// lib/modelos/item.dart
class Item {
  String nome;
  int quantidade;

  Item({required this.nome, required this.quantidade});

  // ← Serializar (arrow function)
  Map<String, dynamic> toJson() => {
    'nome': nome,
    'quantidade': quantidade,
  };

  // ← Desserializar
  factory Item.fromJson(Map<String, dynamic> map) {
    return Item(
```

```
      nome: map['nome'] as String,
      quantidade: map['quantidade'] as int,
    );
  }
}
```

Saída esperada (após toJson()):

```
{"nome": "Poção de Cura", "quantidade": 5}
```

Serializar Todo o Estado do Jogo

Para salvar um jogo inteiro, você precisa de uma classe que agregue todo o estado: jogador, mapa, inimigos, etc. Esta é a "foto" do jogo num momento específico.

```
// lib/jogo/estadoJogo.dart
class EstadoJogo {
  late Jogador jogador;
  late MapaMasmorra mapa;
  late List<Inimigo> entidades;
  int andarAtual = 0;
  DateTime ultimoSalva = DateTime.now();

  // ← Serializar tudo
  Map<String, dynamic> toJson() {
    return {
      'jogador': jogador.toJson(), // ← Jogador serializa a si mesmo
      'mapa': mapa.toJson(), // ← Mapa serializa a si mesmo
      // ← Lista de inimigos
      'entidades': entidades.map((e) => e.toJson()).toList(),
      'andarAtual': andarAtual,
      // ← DateTime como string ISO
      'ultimoSalva': ultimoSalva.toIso8601String(),
```

```
    };
  }

  // ← Desserializar tudo
  factory EstadoJogo.fromJson(Map<String, dynamic> map) {
    final estado = EstadoJogo();
    estado.jogador = Jogador.fromJson(
      map['jogador'] as Map<String, dynamic>,
    );
    estado.mapa = MapaMasmorra.fromJson(
      map['mapa'] as Map<String, dynamic>,
    );
    estado.entidades = (map['entidades'] as List)
        .map((e) => Inimigo.fromJson(e as Map<String, dynamic>))
        .toList();
    estado.andarAtual = map['andarAtual'] as int;
    estado.ultimoSalva = DateTime.parse(
      map['ultimoSalva'] as String, // ← Reconstrói DateTime de string
    );
    return estado;
  }
}
```

Este padrão funciona recursivamente: `EstadoJogo` chama `toJson()` de seus membros, que chamam `toJson()` de seus membros, e assim por diante. No final, você tem uma estrutura de *JSON* profunda.

MapaMasmorra: Serializar Tiles

Serializar um mapa é complexo: temos uma matriz 2D de tiles. Não podemos salvar objetos `Tile` diretamente; convertemos para strings (nomes dos tipos) e reconvertemos.

```
// lib/mundo/mapaMasmorra.dart
class MapaMasmorra {
  int largura;
```

```
  int altura;
  List<List<Tile>> tiles;

  MapaMasmorra(this.largura, this.altura)
    : tiles = List.generate(altura, (_) =>
        List.generate(largura, (_) => Tile.vazio())
      );

  // ← Serializar: converte tiles em strings
  Map<String, dynamic> toJson() {
    return {
      'largura': largura,
      'altura': altura,
      'tiles': tiles.map((linha) =>
          // ← TipoTile como string
          linha.map((tile) => tile.tipo.toString()).toList()
      ).toList(),
    };
  }

  // ← Desserializar: reconstrói tiles de strings
  factory MapaMasmorra.fromJson(Map<String, dynamic> map) {
    final largura = map['largura'] as int;
    final altura = map['altura'] as int;
    final mapa = MapaMasmorra(largura, altura);

    final tileStrings = map['tiles'] as List;
    for (int y = 0; y < altura; y++) {
      for (int x = 0; x < largura; x++) {
        final tipoStr = (tileStrings[y] as List)[x] as String;
        // ← Encontra enum pelo nome
        mapa.tiles[y][x] = Tile(tipo: TipoTile.values
          .firstWhere((t) => t.toString() == tipoStr));
      }
    }
    return mapa;
```

```
  }
}
```

Observe a estratégia: enums são serializados como strings, e ao desserializar, encontramos o enum novamente usando `.values.firstWhere()`.

GerenciadorSalve: Múltiplos Slots

Um jogo típico permite vários *save slots*: save 1, save 2, save 3. Cada um é um arquivo separado. `GerenciadorSalve` centraliza toda a lógica: salvar, carregar, listar. É uma camada de abstração que o resto do jogo usa sem conhecer detalhes do disco.

```dart
// lib/persistencia/gerenciadorSalve.dart
import 'dart:convert';
import 'dart:io';

class GerenciadorSalve {
  static const String dirSalves = 'salves'; // ← Diretório de saves
  static const int numSlots = 5; // ← 5 slots disponíveis

  static Future<void> inicializar() async {
    final dir = Directory(dirSalves);
    if (!dir.existsSync()) {
      // ← Cria diretório se não existir
      dir.createSync(recursive: true);
    }
  }

  static Future<void> salvar(
    EstadoJogo estado,
    int slot,
  ) async {
    if (slot < 0 || slot >= numSlots) {
      throw ArgumentError('Slot inválido: $slot');
```

```
    }

    final arquivo = File('$dirSalves/salve_$slot.json');
    final json = jsonEncode(estado.toJson()); // ← Serializa para
    ↪  string

    try {
      await arquivo.writeAsString(json); // ← Escreve em disco
      print('Jogo salvo no slot $slot');
    } catch (e) {
      print('Erro ao salvar: $e');
      rethrow;
    }
  }

  static Future<EstadoJogo?> carregar(int slot) async {
    if (slot < 0 || slot >= numSlots) {
      throw ArgumentError('Slot inválido: $slot');
    }

    final arquivo = File('$dirSalves/salve_$slot.json');

    if (!arquivo.existsSync()) {
      return null; // ← Nenhum salve neste slot
    }

    try {
      final json = await arquivo.readAsString(); // ← Lê de disco
      // ← Parse JSON
      final map = jsonDecode(json) as Map<String, dynamic>;
      return EstadoJogo.fromJson(map); // ← Reconstrói estado
    } catch (e) {
      print('Erro ao carregar: $e');
      return null; // ← Arquivo corrompido
    }
  }
```

```
  // ← Listar todos os saves com timestamps
  static Future<List<DateTime?>> listarSalves() async {
    final slots = <DateTime?>[];

    for (int i = 0; i < numSlots; i++) {
      final arquivo = File('$dirSalves/salve_$i.json');

      if (arquivo.existsSync()) {
        try {
          final json = await arquivo.readAsString();
          final map = jsonDecode(json) as Map<String, dynamic>;
          final timestamp = DateTime.parse(
            map['ultimoSalva'] as String,
          );
          slots.add(timestamp);
        } catch (_) {
          slots.add(null); // ← Arquivo corrompido
        }
      } else {
        slots.add(null); // ← Vazio
      }
    }

    return slots;
  }
}
```

Saída esperada (após salvar):

```
Jogo salvo no slot 0
Jogo salvo no slot 1
```

Saída esperada (listarSalves()):

```
[2026-04-04 10:30:45, 2026-04-03 18:15:20, null, null, null]
```

Auto-Save Após Cada Andar

Um *auto-save* garante que o progresso do jogador não é perdido. No slot 0, você mantém uma "foto" automática do jogo que atualiza a cada turno. Se o jogo crasha, quando o jogador reabre, pode recuperar de onde parou.

```
// lib/jogo/dungeonCrawl.dart
class DungeonCrawl {
  late EstadoJogo estado;
  static const int slotAutoSalve = 0; // ← Slot dedicado para
  ↪  auto-save

  void executar() async {
    await GerenciadorSalve.inicializar();

    while (estado.jogador.estaVivo) {
      renderizar();
      final cmd = processarInput();
      executarComando(cmd);

    // ← Auto-salve a cada turno (importante: jogo continua
    ↪  responsivo)

      await _autoSalvar();
    }
  }

  Future<void> _autoSalvar() async {
    estado.ultimoSalva = DateTime.now();
    // ← Salva assincronamente
    await GerenciadorSalve.salvar(estado, slotAutoSalve);
  }
}
```

Observe que `_autoSalvar()` é `async` e `await`. Assim, se o disco for lento, o jogo não congela esperando—continua rodando enquanto o save acontece em *background*.

Saída esperada (durante jogo):

```
Turno 1: Herói se move
[auto-save em background]
Turno 2: Herói ataca Goblin
[auto-save em background]
```

Carregar Save ao Iniciar

Este é o fluxo completo: menu inicial, escolha do jogador, carregamento ou novo jogo. Integra tudo que aprendemos: *async/await*, persistência, *JSON*, tratamento de erros.

```dart
// lib/main.dart
import 'dart:io';

void main() async {
  await GerenciadorSalve.inicializar(); // ← Prepara diretório

  // ← MENU
  print('Bem-vindo ao Masmorra!');
  print('1. Novo jogo');
  print('2. Carregar salve');

  stdout.write('> ');
  final opcao = stdin.readLineSync() ?? '1';

  EstadoJogo estado;

  if (opcao == '1') {
    // ← NOVO JOGO
    estado = criarNovoJogo();
```

```
  } else {
    // ← CARREGAR JOGO
    print('\nSlots disponíveis:');
    final salves = await GerenciadorSalve.listarSalves();

    for (int i = 0; i < salves.length; i++) {
      if (salves[i] != null) {
        print('  $i. ${salves[i]}');
      } else {
        print('  $i. [Vazio]');
      }
    }

    stdout.write('Qual slot? > ');
    final slot = int.parse(stdin.readLineSync() ?? '0');

    final carregado = await GerenciadorSalve.carregar(slot);
    if (carregado == null) {
      print('Erro ao carregar. Novo jogo...');
      estado = criarNovoJogo();
    } else {
      estado = carregado;
    }
  }

  // ← INICIAR JOGO
  final game = DungeonCrawl()..estado = estado;
  game.executar();
}
```

Saída esperada (novo jogo):

```
Bem-vindo ao Masmorra!
1. Novo jogo
2. Carregar salve
```

```
> 1
[jogo começa]
```

Saída esperada (carregar jogo):

```
Bem-vindo ao Masmorra!
1. Novo jogo
2. Carregar salve
> 2

Slots disponíveis:
  0. 2026-04-04 10:30:45.123456
  1. 2026-04-03 18:15:20.654321
  2. [Vazio]
  3. [Vazio]
  4. [Vazio]
Qual slot? > 0
[jogo continua do turno salvo]
```

Por que não usar banco de dados?

Você poderia usar SQLite ou Firebase em vez de *JSON* em arquivo. Cada abordagem tem trade-offs:

JSON em arquivo (escolha neste capítulo): - □ Simples, sem dependências externas - □ Arquivo legível, fácil debugar - □ Rápido para pequenos saves (<10MB) - □ Não escala para dados gigantes - □ Sem queries sofisticadas - □ Sem índices (busca é O(n))

SQLite: - □ Rápido para muitos dados - □ Queries sofisticadas - □ Índices para busca O(1) - □ Mais complexo - □ Requer biblioteca externa

Firebase: - □ Multiplayer sincronizado - □ Backup automático na nuvem - □ Requer conexão - □ Dados compartilhados (privacidade)

Para um *roguelike* offline, *JSON* em arquivo é perfeito. Quando você precisar de multiplayer ou dados massivos, migre para banco de dados.

Desafios da Masmorra

Desafio 31.1. Seu Primeiro Await. I/O é lento—disco, rede. Dart não congela esperando. Escreva função que simula carregamento lento: `Future<String> carregarHistoria() async { await Future.delayed(Duration(seconds: 1)); return 'Epopeia carregada'; }`. Chame do `main()` com `async`: `print(await carregarHistoria())`. Note que programa continua responsivo. Dica: `async` + `await` é a base de I/O moderno.

Desafio 31.2. Serializar e Reconstruir. Escolha `Item` ou `Arma`. Implemente `Map<String, dynamic> toJson()`: retorna mapa com todas propriedades. E `factory Item.fromJson(Map m)` que reconstrói. Teste: `var item = Item('Espada', 10); var map = item.toJson(); var item2 = Item.fromJson(map); expect(item2.nome, equals(item.nome));`. Agora Item pode viajar como JSON. Dica: toJson/fromJson é padrão em Dart.

Desafio 31.3. Salve em Disco. JSON em memória é inútil—precisa ir pro disco. Escreva `Item` para arquivo JSON: (1) crie Item, (2) chame `jsonEncode(item.toJson())`, (3) escreva em arquivo com `await File('item.json').writeAsString(json)`, (4) leia de volta, (5) valide que dados são iguais. Arquivo persiste após fechar programa. Dica: sempre use `await` em operações de arquivo.

Desafio 31.4. Múltiplos Saves. Implemente `GerenciadorSalve` com 3 slots: `salvar(estado, slot)` serializa para `save_$slot.json`, `carregar(slot)` desserializa. Trate arquivo faltando (retorna null). Teste: (1) salve estado em slot 1, (2) mude estado, (3) carregue slot 1, deve ser igual ao original. Dica: try/catch captura erros de disco.

Boss Final 31.5. Auto-Save Mágico. Você está explorando andar 3, de repente fecha o jogo. Quando reabre, está no mesmo lugar. Integre autosave: (1) no main, crie `GerenciadorSalve`, (2) em cada turno/comando do jogo, `await gerenciador.salvar(estadoJogo, 999)` (slot auto), (3) ao iniciar, pergunte "Recuperar save anterior?", (4) teste: jogue 10 turnos, fecha, reabre, deve estar no turno 10. Progresso é sagrado. Dica: `main()` deve ser `async`, salve após cada ação importante.

Pergaminho do Capítulo

Você aprendeu persistência completa:

- *Future* é uma promessa de um valor futuro
- *async* marca função como assíncrona (pode usar *await*)
- *await* aguarda uma *Future*

- *dart:io* para ler/escrever arquivos
- *dart:convert* para *JSON* encode/decode
- *toJson()/fromJson()* para *serialização*
- *GerenciadorSalve* gerencia múltiplos slots
- *Auto-save* garante progresso não é perdido
- Tratamento de erros para arquivos corrompidos

Um jogo sem persistência é um jogo que o jogador não pode realmente vencer: toda sessão é resetada. Com persistência, é uma campanha real que atravessa semanas.

Dica do Mestre: Sempre trate erros em I/O assíncrono. Arquivo pode estar corrompido, disco cheio, permissões insuficientes. Use `try/catch`:

```
Future<EstadoJogo?> carregar(int slot) async {
  try {
    final arquivo = File('salve_$slot.json');
    final json = await arquivo.readAsString();
    return EstadoJogo.fromJson(jsonDecode(json));
  } on FileSystemException catch (e) {
    print('Erro de disco: $e');
    return null;
  } on FormatException catch (e) {
    print('Arquivo corrompido: $e');
    return null;
  } catch (e) {
    print('Erro desconhecido: $e');
    return null;
  }
}
```

Próximo Capítulo

No Capítulo 32, organizaremos o projeto para escala profissional. Estrutura de pastas, imports consistentes, `pubspec.yaml` e `analysis_options.yaml` são a base de qualquer projeto Dart sério.

Capítulo 32 - Organização de Projeto: lib/, test/, pubspec.yaml

Você abre a gaveta de ferramentas. Está tudo lá: martelos, pregos, parafusos, serras. Mas está misturado. Gasta 10 minutos procurando a chave inglesa. Um carpinteiro experiente tem tudo organizado. Encontra em 2 segundos. Um projeto Dart bem organizado é assim.

Um aventureiro não guardaria todas as suas armas, poções, mapas e tesouro no mesmo bolso. Mistura tudo, tudo se quebra, nada se encontra. Um aventureiro profissional tem mochilas temáticas: armas em um compartimento; poções em outro; ouro guardado com cuidado; mapas enrolados separados.

Um projeto Dart precisa da mesma organização profissional. Código de modelo em `lib/modelos/`, código de interface em `lib/ui/`, o ponto de entrada em `lib/main.dart`, testes em `test/`, configuração em `pubspec.yaml`. Você não coloca tudo em um único arquivo na raiz do projeto. Use **dart test** para rodar testes e **dart create** para gerar novos projetos. A **analysis_options.yaml** define regras de linting. Estrutura profissional torna fácil encontrar código, adicionar funcionalidades novas, reutilizar código em outros projetos, trabalhar em equipe.

Estrutura de Projeto Dart

A estrutura de pastas de um projeto Dart profissional segue convenções que facilitam manutenção e escalabilidade. Você organiza código por domínio de responsabilidade: `model/` concentra dados e lógica de negócio, `ui/` tudo que renderiza ou interage com o usuário, `jogo/` a orquestração central, e assim por diante. Testes espelham essa mesma organização em `test/`. Isso não é aleatório; é uma convenção adotada pela comunidade Dart que torna projetos previsíveis e fáceis de navegar para qualquer desenvolvedor.

```
masmorra_ascii/
  lib/
    main.dart
    model/
      jogador.dart
      inimigo.dart
      item.dart
    ui/
      telaascii.dart
      renderizador.dart
    jogo/
      dungeonCrawl.dart
      estadoJogo.dart
      parseador.dart
    combate/
      combate.dart
    mundo/
      mapaMasmorra.dart
    config/
      constantes.dart
    persistencia/
      gerenciadorSalve.dart
  test/
    model/
    jogo/
    combate/
  pubspec.yaml
  analysis_options.yaml
  README.md
```

lib/: Código Reutilizável

`lib/` é o coração do seu projeto—contém todo código que pode ser importado por outros projetos Dart. A razão para isso é simples: você quer que sua lógica de jogo seja independente da interface. Um dia você pode querer usar a mesma lógica em Flutter, em um servidor backend, ou compartilhada

como um *package*. Ao isolar código reutilizável em `lib/`, você constrói para o futuro.

```
import 'package:masmorra_ascii/modelos/jogador.dart';
```

Organize por responsabilidade: `model/` para estruturas de dados e lógica de negócio, `ui/` para renderização, `jogo/` para o *loop* principal e orquestração, `combate/` para batalhas, `mundo/` para geração e manutenção do mapa. O ponto de entrada executável fica em `lib/main.dart`.

test/: Testes

Espelha `lib/`:

```
test/
  model/
    jogador_test.dart
  jogo/
    parseador_test.dart
```

pubspec.yaml: Configuração

```
name: masmorra_ascii
description: Roguelike ASCII em Dart
version: 1.0.0

environment:
  sdk: '>=3.11.0 <4.0.0'

dependencies:

dev_dependencies:
  test: ^1.25.0
  lints: ^3.0.0
```

lib/main.dart: Ponto de Entrada

O arquivo `main.dart` é a porta de entrada do programa executável. Sua responsabilidade é única: orquestração de alto nível. Ele inicializa sistemas, mostra menu, delega trabalho para classes de domínio. Nunca contém lógica complexa de jogo; apenas coordena. Isso torna fácil testar lógica de jogo isoladamente (sem passar por `main.dart`) e mantém o programa limpo e compreensível.

```
// lib/main.dart
import 'jogo/dungeonCrawl.dart';
import 'persistencia/gerenciadorSalve.dart';
import 'dart:io';

void main() async {
  // ← inicializa persistência antes de qualquer lógica
  await GerenciadorSalve.inicializar();

  print('');
  print('MASMORRA ASCII: INÍCIO');
  print('—' * 30);
  print('');

  print('1. Novo jogo');
  print('2. Carregar salve');
  print('3. Sair');
  print('');

  stdout.write('Escolha: > ');
  final opcao = stdin.readLineSync() ?? '3';

  // ← switch despacha para funções especializadas
  switch (opcao) {
    case '1':
      await _novoJogo();
    case '2':
      await _carregarJogo();
```

```
    default:
      return;
  }
}

// ← função privada: novo jogo desde zero
Future<void> _novoJogo() async {
  stdout.write('Seu nome: > ');
  final nome = stdin.readLineSync() ?? 'Herói';

  final game = DungeonCrawl();
  game.iniciar(nome);
  game.executar();
}

// ← função privada: carregar jogo salvo
Future<void> _carregarJogo() async {
  final salves = await GerenciadorSalve.listarSalves();

  for (int i = 0; i < salves.length; i++) {
    if (salves[i] != null) {
      print('  $i. ${salves[i]}');
    } else {
      print('  $i. [Vazio]');
    }
  }

  stdout.write('Slot: > ');
  final slot = int.parse(stdin.readLineSync() ?? '0');

  final estado = await GerenciadorSalve.carregar(slot);
  if (estado == null) {
    print('Erro ao carregar');
    return;
  }
```

```
  final game = DungeonCrawl()..estado = estado;
  game.executar();
}
```

Saída esperada:

```
MASMORRA ASCII: INÍCIO
──────────────────────────────────

1. Novo jogo
2. Carregar salve
3. Sair

Escolha: > 1
Seu nome: > Aventureiro
```

Imports: Relativos em lib/, package: em test/

Quando você espalha código em múltiplos arquivos, precisa importar de um para o outro. Dentro de `lib/`, use *imports* relativos—são simples, diretos e tornam refatoração de pastas mais fácil. Em `test/`, use `package:` *imports* (convenção Dart para acessar código de `lib/` a partir de `test/`). A razão para isso é clara: `lib/` é seu código reutilizável; `test/` é código que testa o pacote *como se fosse um consumidor externo*. Assim você garante que seu pacote pode de fato ser importado por outros projetos sem problemas.

```
// Em lib/ — imports relativos ✓
import 'model/jogador.dart';
import 'combate/combate.dart';

// Em test/ — package: imports ✓
import 'package:masmorra_ascii/model/jogador.dart';
```

Imports relativos em `lib/` são claros: você sabe exatamente onde está o arquivo em relação ao arquivo atual. Imports `package:` em `test/` espelham como um usuário externo importaria seu código.

pubspec.yaml: Dependências

O arquivo `pubspec.yaml` (abreviação de "pub spec") é o coração de metadados do seu projeto. Ele declara o nome, versão, dependências de produção, e dependências de desenvolvimento (dev_dependencies). Dependências de desenvolvimento como `test` e `lints` são usadas apenas durante desenvolvimento e testes; não são incluídas quando alguém usa seu código como *package*. Isso mantém seu *package* leve e sem poluição.

```
name: masmorra_ascii
description: Roguelike ASCII em Dart puro
version: 1.0.0
publish_to: none  # ← impede publicação acidental em pub.dev

environment:
  sdk: '>=3.11.0 <4.0.0'  # ← requer Dart 3.11 ou superior

dependencies:
  # Puro Dart (nenhuma por enquanto)

dev_dependencies:
  test: ^1.25.0  # ← para escrever e rodar testes
  lints: ^3.0.0  # ← para análise estática de código
```

Executar o jogo:

```
$ dart lib/main.dart
```

Adicionar dependência de produção:

```
$ dart pub add http
```

Adicionar dependência de desenvolvimento (apenas para testes e análise):

```
$ dart pub add --dev coverage
```

Saída esperada:

```
Added dependency 'http' to pubspec.yaml
Downloading http 1.1.0...
```

analysis_options.yaml: Qualidade

Análise estática é um aliado silencioso. Enquanto você programa, `dart analyze` verifica padrões que levam a bugs: variáveis não usadas, falta de null checks, nomes inconsistentes, código morto. O arquivo `analysis_options.yaml` define regras que seu projeto deve seguir. Você começa com `package:lints/recommended.yaml` (recomendações oficiais Dart) e adiciona regras extras conforme necessário. Isso não é burocracia; é guardrails que evitam centenas de horas debugando depois.

```
include: package:lints/recommended.yaml  # ← regras oficiais

linter:
  rules:
    - camel_case_types  # ← nomes de classes em PascalCase
    - constant_identifier_names  # ← constantes em UPPER_CASE
    - empty_constructor_bodies  # ← construtores vazios devem ser ;
    - prefer_const_constructors  # ← use const quando possível
    - prefer_const_declarations  # ← use const para constantes
    - prefer_final_fields  # ← campos não reatribuídos devem ser final
    - prefer_null_aware_operators  # ← use ?? em vez de if null
```

```
    - use_key_in_widget_constructors  # ← (para Flutter futuramente)
    - use_late_for_private_fields_and_variables  # ← lazy init
    - use_string_buffers  # ← concatenação em loop usa StringBuffer
    - use_to_close_over_close  # ← feche recursos que abrir
```

Execute análise:

```
$ dart analyze
```

Saída esperada (zero problemas):

```
Analyzing masmorra_ascii...
No issues found!
```

Se houver problemas, o analyze lista cada um com localização exata para corrigir.

Desafios da Masmorra

Desafio 32.1. Arquitetura Profissional. Todo projeto grande precisa de estrutura. Crie: `lib/model/` (dados), `lib/ui/` (renderização), `lib/jogo/` (loop principal), `lib/combate/` (batalha), `lib/mundo/` (mapa/geração), `lib/config/` (constantes), `lib/persistencia/` (save/load), `test/` (testes). Use `mkdir -p lib/model lib/ui lib/jogo lib/combate lib/mundo lib/config lib/persistencia test`. O ponto de entrada fica em `lib/main.dart`. Estrutura é como anatomia de um ser vivo: cada órgão em seu lugar. Dica: organize por domínio de negócio, não por tipo de código.

Desafio 32.2. Reorganizar com Cuidado. Mova cada arquivo para sua pasta: `Jogador` → `lib/model/jogador.dart`, `TelaAscii` → `lib/ui/tela.dart`, `MapaMasmorra` → `lib/mundo/mapa.dart`. Depois, atualize importações de `'jogador.dart'` para `'model/jogador.dart'`. Teste: `dart analyze` deve passar com zero erros. Se errar um import, código quebraria. Execute `dart lib/main.dart` para confirmar—jogo funciona igual. Dica: refatore arquivo por arquivo, não tudo de uma vez.

Desafio 32.3. Metadados do Projeto. Crie `pubspec.yaml` (coração do projeto): nome, versão, ambiente Dart, dev_dependencies. YAML é sensível a espaços (2 espaços). Exemplo mínimo: nome `masmorra_ascii`, versão `0.1.0`, sdk `>=3.11.0`, dev_dependencies: `test` e `lints`. Execute `dart pub get`. Dica: pubspec.yaml é o contrato do projeto.

Desafio 32.4. Ponto de Entrada Limpo. Revise `lib/main.dart` (arquivo executável). Deve ser fino: imports relativos e orquestração. Exemplo: `import 'jogo/jogo_principal.dart'; void main() async { await rodaJogo(); }`. Execute `dart lib/main.dart`. Lógica complexa fica nas subpastas de lib, main.dart é só porta de entrada. Dica: main.dart orquestra, não implementa.

Boss Final 32.5. Pronto para Produção. Integre tudo: (1) Reorg arquivos em pastas, (2) Update imports, (3) Configure pubspec.yaml e analysis_options.yaml, (4) Execute `dart analyze` → zero avisos, (5) Execute `dart test` → todos verdes, (6) Execute `dart lib/main.dart` → jogo funciona. Projeto é agora profissional, pronto para crescer, documentado e mantido. Dica: cada passo é um commit git: "refactor: move Jogador para lib/model".

Por Que Não…?

Por que não colocar tudo em um arquivo? Você *poderia* colocar todo código em `lib/main.dart`, e funcionaria. Mas seu projeto viraria impossível de navegar em poucas semanas. Refatorar qualquer coisa quebraria tudo. Integrar com alguém seria caos. Reutilizar código em outro projeto? Impossível. Organização não é luxo; é fundação.

Por que não usar imports relativos em `test/`? Se você usar `import 'model/jogador.dart'` em teste, assume uma posição relativa específica de `test/` para `model/`. Se depois você move `model/` para outra pasta, todos testes quebram. Imports `package:` não se importam com posição; sempre funcionam enquanto o pacote existe. É por isso que é convenção em `test/`.

Por que não omitir `analysis_options.yaml`? Porque sem regras explícitas, cada desenvolvedor segue seu próprio padrão. Um escreve `camelCase`, outro `snake_case`. Um usa `??`, outro `if null`. Código fica inconsistente e confuso. `analysis_options.yaml` é acordado silencioso que torna código legível para todos.

Pergaminho do Capítulo

Estrutura profissional estabelece fundações: - `lib/` para código reutilizável, independente de interface - `test/` espelhando `lib/`, testando pacote como

consumidor externo - `pubspec.yaml` declarando dependências, versão e metadados - `analysis_options.yaml` impondo qualidade e consistência - *Imports* relativos em `lib/`, `package:` em `test/` - `main.dart` como orquestrador fino, não implementador

Um projeto bem organizado é dez vezes mais fácil de manter, e cem vezes mais fácil de estender.

Dica do Mestre: Use `.gitignore`:

```
.dart_tool/
pubspec.lock
.DS_Store
*.swp
```

Crie `README.md`:

```
# Masmorra ASCII

Roguelike ASCII em Dart puro.

## Como Jogar

```bash
dart lib/main.dart
```

Testes

```bash
dart test
```
```

Próximo Capítulo

No Capítulo 33, adicionaremos a última camada de qualidade: testes *golden* que validam a aparência visual da HUD e um renderizador ASCII polido com barras de progresso e painéis informativos. Você começará a integrar

a estrutura que construiu aqui com renderização visual, garantindo que a interface nunca quebra acidentalmente.

PARTE VI

A MENTE DOS MONSTROS

Padrões dão nome ao instinto bruto. IA dá propósito deliberado à ameaça. Aqui, os inimigos deixam de ser pedaços de máquina e ganham vontade aparente, comportamento visível, o fantasma da inteligência. E quando o jogo todo se ergue — código, arte, lógica, intriga — as máquinas e os jogadores se olham face a face. Abrem-se os olhos. Vence quem melhor compreende o outro.

Capítulo 33 - Testes Golden e HUD ASCII Polido

Você testou comportamento—HP sobe, XP conta. Mas há uma dimensão que os testes nunca capturaram: o desenho na tela. Uma golden test captura a imagem de hoje e compara com amanhã. Se mudou sem permissão, o teste grita aviso.

Você testou comportamento. HP sobe quando bebe poção, XP conta quando mata inimigo, combate funciona. Mas há uma dimensão que ninguém testava: o desenho. Como sabe se a HUD fica alinhada? Se as caixas estão desalinhadas? Se uma refatoração invisível quebrou a aparência?

Golden tests são screenshots testados. Você captura a saída ASCII exatamente como deveria parecer, salva esse "golden" (padrão de ouro), e depois, em cada mudança futura, compara. Se o desenho mudou, o teste falha. Você fica sabendo: foi intencional ou acidental?

Antes da batalha final, todo herói polida sua armadura. HUD polida e testes golden são esse polimento final. Não é apenas funcional; é profissional.

Por que Testes Golden Importam?

Quando você está desenvolvendo um *roguelike*, a saída visual é tão importante quanto a lógica. Um herói com HP renderizado errado, estatísticas desalinhadas ou um mapa truncado pode parecer um bug crítico para o jogador. Golden tests são o seu escudo contra esses problemas invisíveis. Eles funcionam como patrulheiros noturnos da masmorra: capturam exatamente o que o jogador vê a cada frame, e se algo mudou (mesmo que acidentalmente), o alarme toca.

Imagine refatorar o sistema de renderização para otimizar performance. Você muda como barras de HP são desenhadas, reorganiza linhas da HUD, ajusta larguras. Sem Golden tests, você só descobre o problema quando começa a jogar e nota que tudo está estranho. Com Golden tests, o teste falha imediatamente, avisando que algo visual mudou—intencional ou não.

Golden Tests: Snapshots de Saída

Um *golden test* (teste padrão de ouro) captura a saída visual exatamente como deveria ser e a valida em futuras execuções. O fluxo é simples:

1. **Executa código**, coleta a saída textual (neste caso, o ASCII renderizado da HUD)
2. **Primeira execução**: cria arquivo "golden" (baseline) com a saída esperada
3. **Próximas execuções**: compara saída atual com baseline; se mudou, o teste falha
4. **Mudança intencional**: você revisa a diferença, confirma que é desejada, e atualiza o golden

Por que isso importa? Refatorar código de renderização é perigoso. Você muda um detalhe—espaçamento, caractere de barra, alinhamento—e acidentalmente quebra a aparência para o jogador. Sem *golden tests*, você só descobre ao jogar. Com eles, o teste grita: "Ei, algo visual mudou!" Você revisa, confirma se foi intencional, e segue.

Esse padrão é padrão-ouro em teste visual. Engines gráficas (Unity, Godot) comparam pixels. Em um *roguelike* ASCII, comparamos strings. A ideia é idêntica: capturar e validar saída visual através de regressão.

O código abaixo implementa um *golden test* básico. Ele renderiza o status do jogador, e se o arquivo golden não existe, cria um. Se existe, valida que a saída atual bate com o padrão salvo:

```
// test/ui/hud_golden_test.dart
import 'package:test/test.dart';
import 'package:masmorra_ascii/ui/renderizador.dart';
import 'dart:io';

void main() {
  group('Golden Tests', () {
    test('renderizar status jogador', () {
      final render = Renderizador();
      final jogador = Jogador(
        nome: 'Herói',
        hpMax: 50,
        ataque: 10,
```

```
    );
    jogador.hpAtual = 35;  // ← 70% de HP
    jogador.xp = 120;
    jogador.nivel = 5;

    final output = render.renderizarStatus(jogador);

    final goldenFile = File('test/golden/status.txt');
    if (goldenFile.existsSync()) {
      // ← segunda e próximas execuções: compara com padrão
      final golden = goldenFile.readAsStringSync();
      expect(output, equals(golden),
        reason: 'HUD diferente do padrão. Verifique alinhamento.');
    } else {
      // ← primeira execução: cria o arquivo golden
      goldenFile.parent.createSync(recursive: true);
      goldenFile.writeAsStringSync(output);
      print('Golden criado em: ${goldenFile.path}');
    }
  });

  test('renderizar mapa inteiro', () {
    final render = Renderizador();
    final mapa = MapaMasmorra(largura: 20, altura: 10);
    final jogador = Jogador(nome: 'Herói')..pos = Offset(5, 5);
    final inimigos = [
      // ← E (inimigo)
      Inimigo(tipo: TipoInimigo.goblin)..pos = Offset(8, 7),
      // ← E (inimigo)
      Inimigo(tipo: TipoInimigo.orc)..pos = Offset(12, 3),
    ];

    final output = render.renderizarMapa(mapa, jogador, inimigos);
    final goldenFile = File('test/golden/mapa.txt');

    if (goldenFile.existsSync()) {
```

```
        // ← compara com padrão salvo
        expect(output, equals(goldenFile.readAsStringSync()));
      } else {
        // ← primeira execução: cria padrão
        goldenFile.parent.createSync(recursive: true);
        goldenFile.writeAsStringSync(output);
      }
    });
  });
}
```

Quando usar Golden tests: - Quando a saída visual é crítica (HUD, mapa, log de combate) - Quando você refatora código de renderização e quer garantir que nada mudou visualmente - Quando colabora com outras pessoas e precisa rastrear mudanças de UI no git - Quando você quer testar casos complexos (barra de HP com 7%, posição de inimigos específica, alinhamento com nomes longos)

HUD Polida: Renderização Profissional

Uma HUD profissional não é apenas texto amontoado. Ela alinha itens visualmente, usa linhas simples para organizar informações, mostra barras visuais em vez de números crus (uma barra de HP preenchida é mais intuitiva que "45/50"). A qualidade da interface comunica ao jogador: "este jogo foi feito com esmero."

Por que isso importa: Em um *roguelike* ASCII, a interface é tudo que o jogador vê. Não há gráficos 3D para compensar um layout ruim. Informações bem alinhadas, barras bem preenchidas, números bem espaçados—tudo isso comunica profissionalismo e torna o jogo legível em combate intenso. Um mapa desalinhado causa confusão; uma barra truncada esconde informação crítica.

Técnicas chave que vamos usar: - ***StringBuffer***: Construir strings linha por linha é eficiente. Em vez de concatenar com + a cada linha ($O(n^2)$ complexidade), você acumula tudo em um buffer e chama `toString()` ao final ($O(n)$). Para HUD com 20+ linhas, a diferença é significativa. - **Métodos *helpers* privados**: `_centralizar()`, `_barra()`, etc. Reutilizáveis, testáveis

isoladamente, e reduzem repetição. - **Caracteres de desenho**: ─, ═, ▓, ▒ criam separadores e barras visualmente claros sem ASCII elaborado.

Aqui está um `Renderizador` completo que encapsula essas técnicas:

```
// lib/ui/renderizador.dart
class Renderizador {
  // ← padrão em terminais (mantém compatibilidade)
  static const int largura = 80;

  /// Renderiza o painel de status do jogador com barras visuais.
  /// Mostra: nome, HP com barra, nível, ataque e XP acumulado.
  /// Usa StringBuffer para eficiência; não concatena com `+` em loop.
  String renderizarStatus(Jogador j) {
    final buffer = StringBuffer();

    // Nome centralizado para destaque visual
    buffer.writeln(_centralizar(j.nome, largura));

    // Separador de topo
    buffer.writeln('─' * largura);

    // HP: barra visual + percentual (mais intuitivo que números crus)
    final barraHp = _barra(j.hpAtual, j.hpMax, 20);
    final niv = j.nivel.toString().padRight(2);
    buffer.writeln('HP: [$barraHp] | Nível: $niv');

    // Ataque (modificador) e XP acumulado
    final atk = j.ataque.toString().padRight(2);
    final xp = j.xp.toString().padRight(5);
    buffer.writeln('Ataque: $atk | XP: $xp');

    // Separador final (delimita painel)
    buffer.writeln('─' * largura);

    return buffer.toString();
  }
```

```
/// Centraliza texto. Se maior que `largura`, retorna intacto.
/// Usado para nomes de personagens e títulos que devem destacar.
String _centralizar(String texto, int largura) {
  if (texto.length >= largura) return texto;
  final padding = (largura - texto.length) ~/ 2;
  return texto.padRight(padding + texto.length).padLeft(largura);
}

/// Desenha barra visual (▓ preenchido, ░ vazio) com percentual.
/// Ex.: _barra(35, 50, 20) dá 14 blocos cheios, 6 vazios, "70%".
/// Mais intuitivo que "35/50": você lê visual em combate rápido.
String _barra(int atual, int maximo, int largura) {
  if (maximo == 0) maximo = 1; // ← evita divisão por zero (edge
  ↪  case)

  final preenchido = (atual / maximo * largura).toInt();
  final vazio = largura - preenchido;
  final pct = (atual / maximo * 100).toInt();

  final p = pct.toString().padLeft(3);
  return '▓' * preenchido + '░' * vazio + ' $p%';
}

/// Renderiza o mapa da masmorra com posição do jogador e inimigos.
/// @ = jogador, E = inimigo, . = vazio
/// Permite jogador "ler" o mapa inteiro com visão tática.
String renderizarMapa(
    MapaMasmorra m, Jogador j, List<Inimigo> inimigos) {
  final buffer = StringBuffer();
  buffer.writeln('— Mapa ' + '—' * (largura - 7));

  for (int y = 0; y < m.altura; y++) {
    // ← indentação para não colar na borda esquerda
    buffer.write('  ');
    for (int x = 0; x < m.largura; x++) {
```

```
        final pos = Offset(x.toDouble(), y.toDouble());
        if (pos == j.pos) {
          buffer.write('@');  // ← Posição do jogador
        } else if (inimigos.any((e) => e.pos == pos)) {
          buffer.write('E');  // ← Inimigo
        } else {
          buffer.write('.');  // ← Vazio
        }
      }
      buffer.writeln();
    }

    buffer.writeln('─' * largura);
    return buffer.toString();
  }

/// Renderiza o log de combate (últimas ações: ataques, poções,
↪  danos).

  /// Mostra apenas os 5 últimos eventos para não poluir a tela.
  /// Útil para o jogador entender o que aconteceu na masmorra.
  String renderizarLog(List<String> eventos) {
    final buffer = StringBuffer();
    buffer.writeln('─ Log de Combate ' + '─' * (largura - 17));

    // Mostra apenas os 5 últimos eventos para não poluir tela
    final mostrados = eventos.length > 5
      ? eventos.sublist(eventos.length - 5)
      : eventos;

    for (final evento in mostrados) {
      // ← trunca eventos muito longos para caber na largura
      final truncado = evento.length > largura - 4
        ? evento.substring(0, largura - 7) + '...'
        : evento;
      buffer.writeln('  $truncado');
```

```
    }

    buffer.writeln('─' * largura);
    return buffer.toString();
  }
}
```

Notas de design: - A constante `largura = 80` é padrão em terminais desde os anos 80 (terminais VT100). Respeitar isso torna o jogo compatível em qualquer terminal, em qualquer máquina. É escolha pragmática, não estética. - *StringBuffer* é mais eficiente que concatenação com `+` em loops ou múltiplas strings. Concatenação cria cópia a cada `+`; *StringBuffer* acumula internamente. Para HUD com 20+ linhas, a diferença é tangível em performance. - Métodos privados (`_barra`, `_centralizar`) agrupam lógica reutilizável e testável. Você pode testar `_barra()` isoladamente sem dependência de `Jogador` ou `MapaMasmorra`. Isso é composição em ação.

Integração com Capítulo 32: Você organizou código em `lib/ui/` no capítulo anterior; agora você implementa o conteúdo. A estrutura habilita a especialização. Renderizador vive isolado em `lib/ui/renderizador.dart`, testado com *golden tests*, e pode ser reutilizado em múltiplos contextos (terminal, Flutter, web, etc.).

O Jogo Até Aqui

Ao final desta parte, seu jogo com HUD polido no terminal se parece com isto:

```
MASMORRA - Andar 3 (Normal)     Turno: 87

  ################
  #..............#
  #..@...........#
  #.........G....#
  #..............#
  ########..#######
         #..#
```

Cada parte adiciona novas camadas ao jogo. Compare com o início e veja o quanto você evoluiu!

Desafio 33.1. Padrão de Ouro. Golden tests são screenshots de texto: você captura a HUD "perfeita", salva em arquivo, e valida que futuros testes batem. Crie um: Jogador "Aventureiro" com HP 30/40, nível 3, ataque 7. Chame `render.renderizarStatus(jogador)`. Primeira execução cria arquivo golden `test/golden/status.txt`. Execute novamente: deve passar (saída confere). É seu seguro contra regressões visuais. Dica: golden test é padrão em testes visuais.

Desafio 33.2. Artesão de Barras. Barra de progresso é arte. Implemente `_barra(atual, max, largura)` que retorna string com blocos: ▓ cheio, ⁘ vazio. Teste 4 casos: (1) barra cheia (50/50), (2) metade (25/50), (3) quase vazia (5/50), (4) vazia (0/50). Valide percentuais: `50/50` → "100%", `25/50` → "50%". Trate `max == 0` sem dividir por zero. Barras visuais contam história do progresso. Dica: `(current * width) ~/ max` calcula quantos blocos.

Desafio 33.3. HUD que Respira. Renderizar HUD bonita é combinar múltiplas caixas alinhadas. Teste Golden que renderiza: (1) status do jogador (nome, HP com barra, nível, XP), (2) mapa 20x10 (@=jogador, E=inimigo, .=chão, #=parede), (3) lado a lado em layout profissional. Salve em `test/golden/hud_completo.txt`. Todas caixas mesma largura (80), alinhadas. Veja: HUD é janela para estado do jogo. Dica: construa com `StringBuffer`, linha por linha.

Desafio 33.4. Galeria de Cenários. Não existe um único estado "correto"—*roguelike* tem 100 situações. Crie 4 goldens para extremos: (1) `status_novo.txt` (nível 1, HP cheio, XP 0), (2) `status_critico.txt` (1 HP / 50, nível 9, XP máximo), (3) `mapa_vazio.txt` (você sozinho), (4) `mapa_cercado.txt` (você rodeado por 4 inimigos). Cada captura um estado. Execute testes: todos criam goldens. Agora refatore renderizador—testes vão falhar (desejado). Dica: cenários extremos expõem bugs que casos normais ocultam.

Desafio 33.5. Refatoração Auditada. Você quer melhorar renderizador. Implemente 4 testes golden acima, execute para criar goldens. Depois refatore: mude largura de 80 para 100, adicione timestamp, altere barra de `█` para `#`. Execute testes—falharão (golden velho vs novo). Revise os `.txt`, confirme mudanças são intencionais. Se sim: delete goldens antigos, execute testes, criem novos. Git mostra exatamente que mudou. Regressão visual é impossível agora. Dica: `git diff test/golden/` mostra antes/depois visualmente.

Boss Final 33.6. Progressão Cinematográfica. Golden tests + progressão. Setup: Jogador nível 1, 0 XP, HP cheio. (1) Teste Golden `prog_nivel1.txt`: renderize estado inicial. (2) `prog_nivel2.txt`: ganhe XP para 30%, nível ainda 1 mas barra mudou. (3) `prog_sobe.txt`: ganhe o XP faltante, nível sobe para 2, barra reseta, HP restaura, ataque aumenta—veja em tela. (4) Repita até nível 3. Crie 5+ arquivos golden que contam a jornada. Cada golden é um frame de filme. Seu teste de integração valida lógica + visual simultaneamente. Dica: isso é teste de integração real—lógica (XP) + renderização juntas.

Por Que Não...?

Por que não usar *mocks* ou snapshots fotográficos? *Mocks* testam comportamento, não aparência. Snapshots fotográficos são específicos de plataforma (PNG em Windows ≠ PNG em Mac por diferenças de rendering). Strings de saída são portáveis e legíveis em diff. *Golden tests* com strings ganham.

Por que não concatenar strings com +? Você *poderia*, e em pequenas escalas funciona. Mas concatenação com + é $O(n^2)$ porque cada + cria cópia. Para 20 linhas de HUD, são 20 cópias. Para 100 linhas, são 10.000 operações. *StringBuffer* é O(n) e invisívelmente mais rápido.

Por que não usar uma biblioteca de renderização como ncurses? ncurses adiciona dependência pesada para um jogo ASCII. Seu *Renderizador* simples é 100 linhas de Dart puro que você controla completamente.

Adicionar ncurses seria sobre-engenharia. YAGNI: “You Aren’t Gonna Need It.”

Pergaminho do Capítulo

Golden tests (*snapshots* de saída textual) capturam HUD exatamente como deveria parecer e a validam em testes futuros. Se algo visual muda, o teste falha, avisando se foi acidental ou intencional. Esse padrão é padrão-ouro em teste visual.

HUD polida usa *StringBuffer* para construir strings eficientemente (O(n) em vez de $O(n^2)$), métodos *helpers* privados para evitar repetição, e caracteres de desenho (–, ▒, ⣿) para parecer profissional. Tudo alinhado e testável.

Quando usar *golden tests*: - Refatorar código de renderização → testes garantem que nada quebrou visualmente - Colaborar em time → rastreie mudanças de *UI* no git (diffs são claros) - Testar casos complexos → barras em 7%, inimigos em posições específicas, nomes longos

Workflow típico: 1. Primeira execução: teste cria arquivo *golden* (baseline) 2. Próximas execuções: teste valida que saída atual confere 3. Se refatorou propositalmente: revisa diferença e atualiza *golden* após confirmar intenção

Golden tests são seu seguro contra regressão visual. Em um *roguelike* ASCII, a interface é tudo que o jogador vê.

Dica do Mestre: Commit goldens com código:

```
git add test/golden/
git add lib/ui/
```

Rastreie mudanças de UI no histórico. Assim, quando você volta no git log, vê exatamente como a HUD era em cada versão. É como um screenshotting automático.

Bonus: Use `git diff test/golden/status.txt` para ver antes/depois visualmente quando você refatora. Muito útil para validar mudanças.

Saída esperada (primeira execução de golden test):

```
Golden criado em: test/golden/status.txt
```

Depois, o arquivo `test/golden/status.txt` contém:

Próximas execuções: teste passa silenciosamente (saída atual = golden salvo). Se refatorar renderização e mudar a saída, o teste falha com diferença clara.

Próximo Capítulo

No Capítulo 34, entramos no território dos padrões de projeto. *Strategy* e *Command* darão inteligência aos inimigos — cada um com comportamento próprio que pode ser trocado em tempo de execução. A interface polida que construiu aqui vai exibir uma IA sofisticada funcionando por trás.

Capítulo 34 - Strategy e Command: Inimigos que Pensam

Nos andares mais profundos da masmorra, há inscrições nas paredes. São padrões de design, técnicas que programadores experientes usam há décadas para resolver problemas recorrentes. Strategy ensina que há muitas formas de atacar o mesmo problema, e o código pode trocar entre elas em tempo de execução. Command transforma ações em objetos, permitindo desfazer jogadas e gravar histórico. Factory cria inimigos e itens sem que o resto do código precise saber dos detalhes. Observer conecta sistemas que não se conhecem. State dá aos inimigos comportamentos que mudam conforme a situação.

Nesta parte final, você vai aplicar cada um desses padrões no jogo que já construiu. A IA dos inimigos ficará mais inteligente, o código mais flexível, e a arquitetura mais elegante. Quando terminar, não terá apenas um jogo completo. Terá um projeto que demonstra domínio de Dart e de engenharia de software. A ascensão final não é derrotar o Dragão. É perceber que você se tornou o tipo de programador que sabe construir qualquer coisa.

> *Um zumbi se arrasta lentamente em padrão aleatório. Um lobo te persegue ferozmente. Um esqueleto patrulha sua rota, ignorando você até você cruzar seu caminho. Cada um tem uma mente própria, não escravizada em if/else aninhados, mas livre para mudar de estratégia como as circunstâncias exigem.*

Neste capítulo você vai implementar dois padrões de design essenciais: Strategy para comportamentos intercambiáveis e Command para ações rastreáveis. Juntos, eles transformam inimigos estáticos em adversários inteligentes.

O Problema: Comportamento Rígido

Até agora, em um combate típico, todos os inimigos agem da mesma forma:

```
void atacarInimigo(Inimigo inimigo) {
  int dano = calcularDano(heroi.arma, inimigo.defesa);
  inimigo.hp -= dano;

  // Inimigo sempre ataca de volta do mesmo jeito
  int danoRetorno = calcularDano(inimigo.arma, heroi.defesa);
  heroi.hp -= danoRetorno;
}
```

Todos fazem a mesma coisa. Um zumbi deveria andar aleatoriamente. Um lobo deveria perseguir você. Um dragão deveria mudar de tática conforme a luta avança. Sem um padrão, você acaba com centenas de if/else aninhados, e cada novo tipo de inimigo quebra a lógica anterior.

Strategy: Uma Mente para Cada Inimigo

O padrão **Strategy** encapsula um conjunto de algoritmos e os torna intercambiáveis. É como em Final Fantasy VII: muda de matéria, muda de poder. A classe do personagem fica a mesma, mas o comportamento muda.

Defina uma interface abstrata:

```
abstract class EstrategiaIa {
  Acao decidir(Inimigo self, Jogador alvo, MapaMasmorra mapa);
}
```

Uma estratégia recebe o inimigo, o alvo e o mapa, e retorna uma **Acao** (que veremos em breve). Agora implemente estratégias concretas.

IAAgressiva: Ataque Direto

Um lobo não hesita. Vê você, vai atrás.

A estratégia agressiva é simples mas efetiva: se o alvo está longe, ande em sua direção. Se está perto (1 tile), ataque. Se você não conseguir traçar uma linha reta até o alvo (há paredes no caminho), apenas ande aleatoriamente. Observe como a decisão retorna uma `Acao` (que veremos em breve); o padrão Command encapsula o que fazer.

```
class IAAgressiva implements EstrategiaIa {
  @override
  Acao decidir(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    int distancia = mapa.distancia(self.pos, alvo.pos);

    if (distancia <= 1) {
      return AcaoAtacar(self, alvo);
    } else if (self.temLinhaDeVisao(alvo, mapa)) {
      var proxPasso = mapa.caminhoParaPos(self.pos, alvo.pos);
      return AcaoMover(self, proxPasso);
    } else {
      return AcaoMoverAleatorio(self, mapa);
    }
  }
}
```

> **Nota:** O método `mapa.distancia()` calcula a distância Manhattan entre duas posições no mapa, utilizada para determinar se um inimigo está próximo o suficiente para atacar (implementado em `mapa.dart`, Capítulo 12). O método `self.temLinhaDeVisao()` verifica se há linha de visão direta entre o inimigo e o alvo (sem paredes bloqueando), usando o algoritmo de Bresenham (implementado em `campo_visao.dart`, Capítulo 19). O método `mapa.caminhoParaPos()` retorna o próximo passo do caminho mais curto entre duas posições, usando busca em largura ou A* (implementado em `pathing.dart`, Capítulo 20).

IACovardia: Retirada Estratégica

Um goblin covarde foge quando ferido:

Observe que `IACovardia` recebe um `limiteHP` (padrão 30%). Se o HP atual cai abaixo desse percentual, muda para fuga. Caso contrário, atua como agressivo. Isso permite criar variações: um goblin que foge em 30%, um orc que só foge em 10%, um dragão que nunca foge. Tudo com a mesma classe, apenas parâmetros diferentes.

```
class IACovardia implements EstrategiaIa {
  final int limiteHP;

  IACovardia({this.limiteHP = 30});

  @override
  Acao decidir(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    if (self.hp < (self.hpMax * limiteHP / 100)) {
      var fuga = mapa.caminhoParaPos(
        self.pos,
        alvo.pos,
        inverso: true
      );
      return AcaoMover(self, fuga);
    }

    int distancia = mapa.distancia(self.pos, alvo.pos);
    if (distancia <= 1) {
      return AcaoAtacar(self, alvo);
    }
    return AcaoMover(self, mapa.caminhoParaPos(self.pos, alvo.pos));
  }
}
```

> **Nota:** O método `mapa.caminhoParaPos()` aceita um parâmetro nomeado `inverso: true` que inverte o algoritmo de *pathfinding*, retornando um passo na direção *oposta* ao alvo, útil para implementar comportamento de fuga (implementado em `pathing.dart`, Capítulo 20).

IAPatrulha: Vigilância Constante

Um esqueleto segue uma rota. Se você aparecer no seu campo de visão, passa a atacar:

Patrulha é mais sofisticada. O inimigo caminha por uma rota predefinida. Se detecta o alvo, passa para combate direto. Note o `emCombate`: uma vez em combate, o inimigo permanece assim até a morte ou vitória. Sem isso, um

inimigo poderia ficar alternando entre patrulha e perseguição infinitamente. Esse flag garante coerência no comportamento.

```
class IAPatrulha implements EstrategiaIa {
  final List<Pos> rota;
  int indiceRota = 0;
  bool emCombate = false;

  IAPatrulha(this.rota);

  @override
  Acao decidir(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    if (self.temLinhaDeVisao(alvo, mapa)) {
      emCombate = true;
    }

    if (emCombate) {
      int distancia = mapa.distancia(self.pos, alvo.pos);
      if (distancia <= 1) {
        return AcaoAtacar(self, alvo);
      }
      return AcaoMover(self, mapa.caminhoParaPos(self.pos, alvo.pos));
    }

    var proxAlvo = rota[indiceRota];
    if (self.pos == proxAlvo) {
      indiceRota = (indiceRota + 1) % rota.length;
      proxAlvo = rota[indiceRota];
    }

    return AcaoMover(self, mapa.caminhoParaPos(self.pos, proxAlvo));
  }
}
```

Nota: O método `self.temLinhaDeVisao()` é usado aqui para detectar automaticamente quando um inimigo em

patrulha avista o jogador, disparando o início do combate (implementado em `campo_visao.dart`, Capítulo 19). Uma vez que `emCombate` é ativado, permanece assim até a morte, garantindo coerência no comportamento sem alternância errática entre patrulha e perseguição.

IAPassiva: Defesa Apenas

Um zumbi anda aleatoriamente e só ataca se for atacado primeiro:

Passiva é o oposto de agressiva. O inimigo ignora você até ser atacado. Depois, reage. Isso simula zumbis que estão dormindo ou distraídos, ou animais selvagens que fogem de humanos mas atacam se provocados. Note como `foiAtacada` é um flag que nunca volta a falso: uma vez despertado, o zumbi permanece hostil.

```
class IAPassiva implements EstrategiaIa {
  bool foiAtacada = false;

  @override
  Acao decidir(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    if (!foiAtacada) {
      return AcaoMoverAleatorio(self, mapa);
    }

    int distancia = mapa.distancia(self.pos, alvo.pos);
    if (distancia <= 1) {
      return AcaoAtacar(self, alvo);
    }
    return AcaoMover(self, mapa.caminhoParaPos(self.pos, alvo.pos));
  }
}
```

Nota: A estratégia passiva depende do seu código de combate (não mostrado) para atualizar o flag `foiAtacada = true` quando este inimigo sofre dano. Depois disso, funciona como a estratégia agressiva. Este é um exemplo de

comunicação entre a IA e o sistema de combate através de estado mutável.

Integrando Strategy no Inimigo

Modifique a classe `Inimigo` para usar uma estratégia.

Quando você integra Strategy:

```
class Inimigo {
  late Pos pos;
  late String nome;
  late int hp;
  late int hpMax;
  late Arma arma;
  late Defesa defesa;
  final EstrategiaIa estrategia;

  Inimigo({
    required this.nome,
    required this.hp,
    required this.arma,
    required this.defesa,
    required this.estrategia,
  }) : hpMax = hp;

  Acao obterProximaAcao(Jogador alvo, MapaMasmorra mapa) {
    return estrategia.decidir(this, alvo, mapa);
  }
}
```

Command: Ações Reversíveis

Strategy diz *o quê fazer*. Command diz *como fazer* de forma reversível e histórica. Command encapsula uma solicitação como um objeto, permitindo desfazer, refazer e manter histórico.

Defina a interface:

Em vez de métodos que executam diretamente, cada ação é um objeto que sabe como se executar, desfazer e descrever a si mesma. Isso permite construir um histórico de ações (útil para replay de combates, undo e logging). A interface é simples: `executar()` faz a coisa, `desfazer()` desfaz, e `descricao` descreve.

```
abstract class Acao {
  void executar();
  void desfazer();
  String get descricao;
}
```

AcaoAtacar

A ação de ataque captura o estado antes (HP anterior) e depois (dano aplicado). Isso permite desfazer: basta restaurar `hp` para o valor anterior. Observe como `dano` é calculado e armazenado em `executar()`; é necessário porque em `desfazer()` você precisa saber quanto dano foi aplicado para poder reverter.

```
class AcaoAtacar implements Acao {
  final Inimigo atacante;
  final Character alvo;
  late int dano;
  late int hpAnterior;

  AcaoAtacar(this.atacante, this.alvo);

  @override
  void executar() {
    hpAnterior = alvo.hp;
    dano = calcularDano(atacante.arma, alvo.defesa);
    alvo.hp -= dano;
  }

  @override
```

```
  void desfazer() {
    alvo.hp = hpAnterior;
  }

  @override
  String get descricao =>
      "${atacante.nome} ataca ${alvo.nome} por $dano!";
}
```

AcaoMover

Movimento também é uma ação. Captura a posição original, move, permite desfazer restaurando a posição. Note que `ePassavel()` garante que você não anda através de paredes; se o destino é inválido, a ação não muda a posição. Importante: sempre validar antes de modificar estado.

```
class AcaoMover implements Acao {
  final Inimigo self;
  final Pos destino;
  late Pos origem;
  final MapaMasmorra mapa;

  AcaoMover(this.self, this.destino, this.mapa);

  @override
  void executar() {
    origem = self.pos;
    if (mapa.ePassavel(destino)) {
      self.pos = destino;
    }
  }

  @override
  void desfazer() {
    self.pos = origem;
```

```
  }

  @override
  String get descricao => "${self.nome} se move";
}
```

AcaoAguardar

Uma ação que não faz nada. Parece inútil, mas é essencial. Um inimigo pode decidir que a melhor ação este turno é aguardar: recarregar, regenerar, ou simplesmente deixar o alvo fazer a próxima ação. Sem `AcaoAguardar`, você precisaria de `if (acao == null)` em todo o código. Com ela, tudo é uniforme: sempre execute uma ação, mesmo que não faça nada.

```
class AcaoAguardar implements Acao {
  final Character self;

  AcaoAguardar(this.self);

  @override
  void executar() {}

  @override
  void desfazer() {}

  @override
  String get descricao => "${self.nome} aguarda";
}
```

Histórico de Ações e Undo

Uma das grandes vantagens de Command é manter um histórico completo:

O `GerenciadorAcoes` mantém uma lista de todas as ações já executadas. Quando você quer desfazer, volta um índice e chama `desfazer()` do comando anterior. Quer refazer? Avança o índice. Quer replay do combate inteiro?

Itere o histórico. Quer ver o log? Obtenha as descrições. Tudo vem grátis dessa abstração simples.

```
class GerenciadorAcoes {
  final List<Acao> historico = [];
  int indiceAtual = -1;

  void executar(Acao cmd) {
    cmd.executar();
    historico.removeRange(indiceAtual + 1, historico.length);
    historico.add(cmd);
    indiceAtual = historico.length - 1;
  }

  void desfazer() {
    if (indiceAtual >= 0) {
      historico[indiceAtual].desfazer();
      indiceAtual--;
    }
  }

  void refazer() {
    if (indiceAtual < historico.length - 1) {
      indiceAtual++;
      historico[indiceAtual].executar();
    }
  }

  List<String> obterHistorico() => historico
      .sublist(0, indiceAtual + 1)
      .map((cmd) => cmd.descricao)
      .toList();
}
```

Turno de Combate Integrado

Agora um turno é limpo e legível:

Veja como `executarTurnoInimigo` é agora uma função simples e linear. O inimigo decide uma ação (via sua estratégia), a ação se executa, o log registra. Sem if/else aninhados, sem estado implícito, sem surpresas. A IA vem da estratégia, a reversibilidade vem do comando.

```
void executarTurnoInimigo(
  Inimigo inimigo,
  Jogador heroi,
  MapaMasmorra mapa,
  GerenciadorAcoes gerenciador,
) {
  var acao = inimigo.obterProximaAcao(heroi, mapa);
  gerenciador.executar(acao);
  log.escrever(acao.descricao);

  if (heroi.hp <= 0) {
    log.escrever("${heroi.nome} caiu!");
  }
}
```

Boss com Fases

Um padrão avançado: um chefe que muda de estratégia conforme seu HP cai:

Um boss não é apenas um inimigo forte. É um combate progredindo. Conforme o herói inflige dano, o chefe muda de tática. Assim como em Dark Souls, onde o boss fica desesperado quando está perto de morrer. Aqui, `BossComFases` muda a estratégia interna baseado no HP. O resto do código não precisa saber disso; para o jogo principal, é apenas mais uma `EstrategiaIa`.

```
class BossComFases implements EstrategiaIa {
  late EstrategiaIa estrategiaAtual = IAAgressiva();

  @override
  Acao decidir(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
```

```
    if (self.hp < (self.hpMax * 50 / 100)) {
      estrategiaAtual = IACovardia(limiteHP: 20);
    } else if (self.hp < (self.hpMax * 25 / 100)) {
      estrategiaAtual = IAAgressiva(); // Desesperado
    }

    return estrategiaAtual.decidir(self, alvo, mapa);
  }
}
```

É como em Dragon Ball: conforme Goku fica mais ferido, ele muda de abordagem. Cada fase tem uma mente própria.

Antes vs. Depois

Antes: Indistinção

```
Zumbi: sempre anda aleatoriamente, sempre ataca se próximo
Lobo: sempre persegue, sempre ataca
Dragão: sempre ataca com força bruta
```

Depois: Singularidade

```
Zumbi (IAPassiva): anda lentamente até ser provocado
Lobo (IAAgressiva): persegue agressivamente, não desiste
Esqueleto (IAPatrulha): patrulha, mas quando vê você muda para combate
Dragão (BossComFases): adapta tática a cada fase, inteligente
```

Pergaminho do Capítulo

Neste capítulo, você aprendeu dois padrões de design que transformam inimigos estáticos em adversários inteligentes e comportamentos previsíveis. O padrão Strategy permite que cada inimigo tenha sua própria “mente” (uma agressiva persegue você ferozmente, outra patrulha e dispara quando

detectada, outra foge quando ferida), tudo sem modificar a classe `Inimigo`. Você implementou cinco estratégias diferentes (`IAAgressiva`, `IACovardia`, `IAPatrulha`, `IAPassiva` e `BossComFases`), cada uma definindo como um inimigo decide agir em um turno. O padrão Command encapsula cada ação (ataque, movimento, espera) como um objeto reversível, permitindo que você construa um histórico completo, desfaça ações e implemente replay de combates. Juntos, Strategy e Command eliminam if/else aninhados, criam inimigos que "pensam" e proveem a base para sistemas de IA sofisticados que respeitam a elegância do código.

Dica do Mestre: Strategy e Command são padrões que vão muito além de jogos. Em aplicações reais, use Strategy sempre que tiver múltiplas formas de executar um algoritmo que pode mudar em runtime, e Command sempre que precisar de histórico, undo/redo, ou logging de operações. Um exemplo: um sistema de pagamentos que pode usar Visa, Mastercard, ou Pix; cada é uma Strategy. Um editor de documentos que permite desfazer múltiplas edições; cada edição é um Command. O investimento em aprender esses padrões numa masmorra digital te torna um desenvolvedor melhor em qualquer contexto.

Desafios da Masmorra

**Desafio 34.1. Implemente uma estratégia `IASuicida` que sempre avança em sua direção até ficar ao seu lado e então "explode", causando dano em raio de 3 tiles em volta (afeta você e outros inimigos).

**Desafio 34.2. Implemente um comando `AcaoLancarMagia` que custa mana (novo atributo do Inimigo) e pode ser desfeito. Crie uma estratégia que lança magia se tiver mana suficiente.

**Desafio 34.3. Modifique `IAPatrulha` para suportar múltiplas rotas e escolha uma aleatoriamente quando criada ou quando termina a rota atual.

**Desafio 34.4. Implemente `AcaoFuga` que move o inimigo para uma posição segura; se não encontrar após 5 turnos, o inimigo muda para `IAAgressiva`.

**Boss Final 34.5. Crie um sistema de "Comportamento Adaptativo" onde um inimigo começa com `IAPatrulha` e, após sofrer 3 ataques consecutivos sem conseguir contra-atacar, muda para `IAAgressiva`. Use um contador interno que reseta quando consegue atacar.

**Desafio 34.6. Implemente uma estratégia `IAMago` para um inimigo mago que: - Mantém distância mínima de 3 tiles do herói - Lança um ataque mágico (`AcaoLancarMagia`) a cada 2 turnos se o herói está em linha de visão - Se o herói

se aproxima a menos de 3 tiles, recua (como `IACovardia` mas com limiteHP = 100%) - A magia custará um novo atributo `mana`, que regenera 5 pontos por turno quando não está em combate

**Desafio 34.7. Implemente um comando `AcaoDesfazerMultiplo` que desfaz as últimas N ações de uma vez. Modifique `GerenciadorAcoes` para suportar `desfazerN(int n)` e `refazerN(int n)`, permitindo replay rápido de segmentos de combate durante debug.

**Desafio 34.8. Crie uma estratégia `IACompostaAgressiva` que combina múltiplas estratégias em sequência: - Primeiros 3 turnos: `IAPatrulha` (patrulha até detectar) - Próximos 5 turnos após detectar: `IAAgressiva` (ataque direto) - Se HP < 50%: `IACovardia` (foge) - Mantém um state machine interno que controla qual sub-estratégia usar em cada fase

---

O padrão Strategy transformou inimigos passivos em adversários inteligentes. Command permitiu que cada ação fosse registrada e revertida. Juntos, eles formam a base de IA sofisticada. A próxima fronteira é multiplicar esses inimigos inteligentes de forma eficiente e fazer o mundo todo reagir aos eventos do combate.

> *"A inteligência sem ação é mera reflexão. A ação sem inteligência é mera sorte. Um rei verdadeiro domina ambas."*

Próximo Capítulo

No Capítulo 35, você verá como usar **Factory** para criar centenas de inimigos variados de forma escalável (centralizada, orientada a dados) e **Observer** para fazer o mundo inteiro reagir aos eventos do combate sem acoplamento direto. Factory define que tipo de inimigo é criado e com que parâmetros; Observer faz cada sistema (log, XP, som, UI) reagir a eventos sem modificar o código de combate.

Capítulo 35 - Factory e Observer: O Mundo Reage

Você mata um inimigo. Instantaneamente: som toca, tela pisca, XP sobe, conquista desbloqueia. O mundo não estava esperando sua ordem—estava observando. Quando algo muda, tudo que precisa saber é notificado. Cacos que se comunicam sem se acoplarem.

Neste capítulo você vai implementar **Factory (padrão de design)** para criar inimigos e itens de forma escalável e **data-driven** e **Observer (padrão de design)** para fazer múltiplos sistemas reagirem aos eventos sem acoplamento direto. Aprenderá também sobre **Singleton**, um padrão para garantir que apenas uma instância de uma classe exista no programa.

Factory: Construtor Central

O padrão Factory centraliza a criação de objetos. Em vez de espalhar `new Inimigo(...)` por todo o código, temos um único lugar que sabe como construir cada tipo.

O Problema: Criação Descentralizada

Sem Factory, você acaba com espaguete:

O código abaixo mostra exatamente o problema: toda vez que você precisa gerar um inimigo, você cria manualmente com todos os parâmetros. Se quer mudar HP, defesa ou estratégia de um zumbi, você precisa encontrar todos os lugares onde escreve `Inimigo(nome: "Zumbi", hp: 20, ...)` e atualizar. Esqueça de um lugar e o jogo fica inconsistente. Um zumbi que deveria ter 20 HP tem 15 em um andar e 20 em outro.

```
void gerarInimigos() {
  if (Random().nextInt(100) < 30) {
    var goblin = Inimigo(
```

```
      nome: "Goblin", hp: 20, arma: Arma(dano: 3),
      defesa: Defesa(1), estrategia: IACovardia(),
    );
    inimigos.add(goblin);
  } else if (Random().nextInt(100) < 60) {
    var lobo = Inimigo(
      nome: "Lobo", hp: 35, arma: Arma(dano: 5),
      defesa: Defesa(2), estrategia: IAAgressiva(),
    );
    inimigos.add(lobo);
  }
}
```

Problemas: - Balanço espalhado por todo o código. - Adicionar novo tipo é tedioso e propenso a erros. - Sem consistência entre andares. - Testes exigem saber detalhes de construção.

FabricaInimigo: Solução Centralizada

Agora toda definição está em um único lugar. Um `Map` de catálogo mapeia tipo (string) para definição. Métodos estáticos criam inimigos: `criar('zumbi', 3)` cria um zumbi no andar 3, com HP e dano escalonados por andar. Mudar HP de um zumbi? Mude em um lugar. Quer um novo tipo? Adicione ao catálogo. O resto do código nunca vê os detalhes construtivos; sempre vai through a factory. Isso é elegância.

```
// lib/fabrica_inimigo.dart
class FabricaInimigo {
  static final Map<String, DefinicaoInimigo> catalogo = {
    'zumbi': DefinicaoInimigo(
      nome: 'Zumbi',
      hpBase: 15,
      danoBase: 2,
      defesaBase: 0,
      raridade: 0.4,
      estrategia: IAPassiva(),
```

```
  ),
  'lobo': DefinicaoInimigo(
    nome: 'Lobo',
    hpBase: 25,
    danoBase: 5,
    defesaBase: 1,
    raridade: 0.35,
    estrategia: IAAgressiva(),
  ),
  'esqueleto': DefinicaoInimigo(
    nome: 'Esqueleto',
    hpBase: 30,
    danoBase: 4,
    defesaBase: 2,
    raridade: 0.25,
    estrategia: IAPatrulha([]),
  ),
};

static Inimigo criar(String tipo, int andar) {
  var def = catalogo[tipo];
  if (def == null) throw ArgumentError('Tipo desconhecido: $tipo');

  int hpFinal = def.hpBase + (andar * 3);
  int danoFinal = def.danoBase + (andar * 1);
  int defesaFinal = def.defesaBase + (andar ~/ 3);

  return Inimigo(
    nome: def.nome,
    hp: hpFinal,
    arma: Arma(dano: danoFinal),
    defesa: Defesa(defesaFinal),
    estrategia: def.estrategia,
  );
}
```

```
  static Inimigo criarAleatorio(int andar) {
    double sorteio = Random().nextDouble();
    double acumulado = 0.0;

    for (var entrada in catalogo.entries) {
      acumulado += entrada.value.raridade;
      if (sorteio < acumulado) {
        return criar(entrada.key, andar);
      }
    }

    return criar('zumbi', andar);
  }

  static void carregarDoJSON(String json) {
    var mapa = jsonDecode(json) as Map<String, dynamic>;
    for (var tipo in mapa.keys) {
      var dados = mapa[tipo] as Map<String, dynamic>;
      catalogo[tipo] = DefinicaoInimigo(
        nome: dados['nome'] as String,
        hpBase: dados['hp'] as int,
        danoBase: dados['dano'] as int,
        defesaBase: dados['defesa'] as int,
        raridade: dados['raridade'] as double,
        estrategia: _criarEstrategia(dados['estrategia'] as String),
      );
    }
  }

  static EstrategiaIa _criarEstrategia(String tipo) {
    return switch (tipo) {
      'agressiva' => IAAgressiva(),
      'covardia' => IACovardia(),
      'patrulha' => IAPatrulha([]),
      'passiva' => IAPassiva(),
      'boss' => BossComFases(),
```

```
      _ => IAAgressiva(),
    };
  }
}

class DefinicaoInimigo {
  final String nome;
  final int hpBase;
  final int danoBase;
  final int defesaBase;
  final double raridade;
  final EstrategiaIa estrategia;

  DefinicaoInimigo({
    required this.nome,
    required this.hpBase,
    required this.danoBase,
    required this.defesaBase,
    required this.raridade,
    required this.estrategia,
  });
}
```

Agora criar é simples e centralizado:

```
var zumbi = FabricaInimigo.criar('zumbi', 3);
var aleatorio = FabricaInimigo.criarAleatorio(5);
```

É como o *spawner* de monstros em Minecraft: tudo vem de um único lugar, balanceado por andar.

FabricaItem: Similarmente

Itens também precisam de factory. Aqui, cada item tem nome, descrição, valor, raridade e um `criador` (uma função que constrói o item real). Isso

permite criar diferentes tipos de poções, armas e armaduras sem espalhar lógica de construção por todo o código.

```
// lib/fabrica_item.dart
class FabricaItem {
  static final Map<String, DefinicaoItem> catalogo = {
    'pocao_vida': DefinicaoItem(
      nome: 'Poção de Vida',
      descricao: 'Restaura 20 HP',
      valor: 50,
      raridade: 0.5,
      criador: () => Pocao(cura: 20),
    ),
    'espada_ferrea': DefinicaoItem(
      nome: 'Espada de Ferro',
      descricao: 'Dano: 5',
      valor: 150,
      raridade: 0.2,
      criador: () => Arma(dano: 5, nome: 'Espada'),
    ),
  };

  static Item criar(String tipo) {
    var def = catalogo[tipo];
    if (def == null) throw ArgumentError('Item desconhecido: $tipo');
    return def.criador();
  }

  static Item criarAleatorio() {
    var tipos = catalogo.keys.toList();
    return criar(tipos[Random().nextInt(tipos.length)]);
  }
}

class DefinicaoItem {
  final String nome;
```

```
  final String descricao;
  final int valor;
  final double raridade;
  final Item Function() criador;

  DefinicaoItem({
    required this.nome,
    required this.descricao,
    required this.valor,
    required this.raridade,
    required this.criador,
  });
}
```

Observer: Sistema de Reações

O padrão Observer permite que múltiplos observadores se inscrevam em eventos sem que o disparador conheça os observadores. Usa `Stream` Dart e `StreamController` para desacoplar totalmente.

Por que Factory e Observer Juntos?

Factory e Observer são complementares. Factory **cria** os objetos (inimigos, itens) de forma consistente. Observer **reage** ao que esses objetos fazem (morte, coleta, dano). Factory diz "aqui está um novo zumbi"; Observer diz "algo importante aconteceu, vou reagir". Juntos, eles separam **criação** (centralizada, orientada a dados) de **comportamento** (reativo, desacoplado). É uma arquitetura poderosa: o que é criado é controlado em um lugar, mas como o sistema reage é extensível sem modificação.

O Problema: Acoplamento

Sem Observer, matar um inimigo seria acoplado demais:

O código abaixo é a raiz do caos. Uma função `matarInimigo` que faz tudo: escreve log, adiciona XP, adiciona ouro, toca som, pisca tela, verifica conquistas. Quer adicionar animação? Edita aqui. Quer remover som? Edita aqui. Cada adição é um risco de quebrar algo que já funciona. Pior,

`matarInimigo` precisa conhecer todos os sistemas: log, UI, som, conquistas, banco de dados. Altíssimo acoplamento.

```
void matarInimigo(Inimigo inimigo) {
  inimigo.hp = 0;
  log.escrever("${inimigo.nome} morreu!");
  heroi.xp += inimigo.xpRecompensa;
  heroi.ouro += inimigo.ouroRecompensa;
  ui.piscar(cor: Color.red);
  som.tocar('morte');
  conquistas.verificar('matador');
}
```

Tudo em uma função. Novo observador? Edita aqui. Ruim.

BarramentoEventos: Solução

Em vez de `matarInimigo` saber de tudo, ele apenas emite um evento: "um inimigo morreu". Quem se importa? Log, UI, som, conquistas, estatísticas. Todos escutam. Nenhum conhece o outro. `matarInimigo` não precisa saber de nada além do evento básico. Quer adicionar um novo sistema que reage a mortes? Cria um novo observador e o registra. Zero mudança no código de combate.

```
// lib/barramento_eventos.dart
abstract class EventoJogo {
  final DateTime timestamp = DateTime.now();
}

class EventoMorteInimigo extends EventoJogo {
  final Inimigo inimigo;
  final Jogador matador;

  EventoMorteInimigo({required this.inimigo, required this.matador});
}
```

```
class EventoColheitaItem extends EventoJogo {
  final Item item;
  final Character personagem;

  EventoColheitaItem({required this.item, required this.personagem});
}

class EventoDanoAplicado extends EventoJogo {
  final Character atacante;
  final Character alvo;
  final int dano;

  EventoDanoAplicado({
    required this.atacante,
    required this.alvo,
    required this.dano,
  });
}

class BarramentoEventos {
  static final BarramentoEventos _instancia = BarramentoEventos._();

  final _controlador = StreamController<EventoJogo>.broadcast();

  BarramentoEventos._();

  factory BarramentoEventos() => _instancia;

  void emitir(EventoJogo evento) {
    _controlador.add(evento);
  }

  Stream<T> on<T extends EventoJogo>() {
    return _controlador.stream.whereType<T>();
  }
```

```
  void fechar() {
    _controlador.close();
  }
}
```

Observadores Concretos

Cada observador é uma classe simples que escuta um tipo de evento e reage. `ObservadorLog` escreve no log. `ObservadorUI` pisca a tela. `ObservadorSom` toca um efeito sonoro. Cada um é isolado e testável. Se o log está quebrado, não afeta som. Se o som está quebrado, não afeta conquistas.

```
// lib/observadores.dart
class ObservadorLog {
  final BarramentoEventos bus;
  final Log log;
  late StreamSubscription subscription;

  ObservadorLog(this.bus, this.log) {
    subscription = bus.on<EventoJogo>().listen((evento) {
      if (evento is EventoMorteInimigo) {
        log.escrever("${evento.inimigo.nome} foi derrotado!");
      } else if (evento is EventoDanoAplicado) {
        log.escrever(
          "${evento.atacante.nome} -> "
          "${evento.alvo.nome}: ${evento.dano}!",
        );
      }
    });
  }

  void cancelar() => subscription.cancel();
}

class ObservadorEstatisticas {
```

```
  final BarramentoEventos bus;
  late StreamSubscription subscription;

  int totalMatos = 0;
  int ouroColetado = 0;
  int danoTotal = 0;

  ObservadorEstatisticas(this.bus) {
    subscription = bus.on<EventoJogo>().listen((evento) {
      if (evento is EventoMorteInimigo) {
        totalMatos++;
      } else if (evento is EventoDanoAplicado) {
        danoTotal += evento.dano;
      }
    });
  }

  void cancelar() => subscription.cancel();
}

class ObservadorUI {
  final BarramentoEventos bus;
  final GerenciadorUI ui;
  late StreamSubscription subscription;

  ObservadorUI(this.bus, this.ui) {
    subscription = bus.on<EventoJogo>().listen((evento) {
      if (evento is EventoDanoAplicado) {
        ui.piscar(
            cor: Cor.vermelho, duracao: Duration(milliseconds: 150));
      } else if (evento is EventoMorteInimigo) {
        ui.mostrarAnimacaoMorte(evento.inimigo.pos);
      }
    });
  }
```

```
  void cancelar() => subscription.cancel();
}

class ObservadorSom {
  final BarramentoEventos bus;
  final GerenciadorSom som;
  late StreamSubscription subscription;

  ObservadorSom(this.bus, this.som) {
    subscription = bus.on<EventoJogo>().listen((evento) {
      if (evento is EventoDanoAplicado) {
        som.tocar('acerto');
      } else if (evento is EventoMorteInimigo) {
        som.tocar('morte');
      }
    });
  }

  void cancelar() => subscription.cancel();
}
```

É como o sistema de notificações do dispositivo móvel: quando algo acontece, múltiplos apps reagem. Nenhum conhece o outro.

Integrando Factory e Observer

Factory cria, Observer reage. Veja como se combinam:

Factory gera inimigos de forma consistente. Durante um turno, o inimigo age, e a ação é executada. Se a ação é um ataque que mata o alvo, um evento é emitido. Todos os observadores registrados escutam e reagem. Simples, elegante, extensível.

```
void gerarMasmorra(int andar) {
  var inimigos = <Inimigo>[];
  for (int i = 0; i < 5; i++) {
```

```
    inimigos.add(FabricaInimigo.criarAleatorio(andar));
  }
  return inimigos;
}

void executarTurnoInimigo(
  Inimigo inimigo,
  Jogador heroi,
  BarramentoEventos bus,
) {
  var acao = inimigo.obterProximaAcao(heroi, mapa);
  acao.executar();

  if (acao is AcaoAtacar) {
    int dano = calcularDano(inimigo.arma, heroi.defesa);
    bus.emitir(EventoDanoAplicado(
      atacante: inimigo,
      alvo: heroi,
      dano: dano,
    ));

    if (heroi.hp <= 0) {
      bus.emitir(EventoMorteInimigo(inimigo: heroi, matador:
↪  inimigo));
    }
  }
}
```

Inicialização Completa

O fluxo de inicialização estabelece o barramento, registra todos os observadores, executa o combate, e depois limpa. Isso garante que observadores vivos escutam eventos e que recursos são liberados quando termina.

```
void main() {
  final bus = BarramentoEventos();

  final obsLog = ObservadorLog(bus, log);
  final obsEstat = ObservadorEstatisticas(bus);
  final obsUI = ObservadorUI(bus, ui);
  final obsSom = ObservadorSom(bus, som);

  var inimigos = gerarMasmorra(andar: 3);
  executarCombate(inimigos, heroi, bus);

  obsLog.cancelar();
  obsEstat.cancelar();
  obsUI.cancelar();
  obsSom.cancelar();
  bus.fechar();
}
```

Vantagens do Design

Antes, adicionar novo efeito exigia editar código de combate. Depois, você apenas cria um novo observador:

Isso é Open/Closed Principle: o código é aberto para extensão (novos observadores) mas fechado para modificação (código de combate não muda). Um novo observador que desbloqueia conquistas é adicionado em um arquivo novo, sem tocar em nada existente. Quer remover? Apaga o arquivo e remove uma linha de registro. Manutenção simples.

```
class ObservadorConquistas {
  final BarramentoEventos bus;
  late StreamSubscription subscription;

  ObservadorConquistas(this.bus) {
    subscription = bus.on<EventoMorteInimigo>().listen((evento) {
      if (evento.inimigo.nome == "Dragão") {
```

```
        conquistas.desbloquear('matador_de_dragoes');
      }
    });
  }

  void cancelar() => subscription.cancel();
}
```

Feito. Nenhuma alteração em código de combate.

Pergaminho do Capítulo

Neste capítulo você aprendeu como Factory centraliza a criação de objetos, removendo lógica de construção espalhada por todo o código. Implementou `FabricaInimigo` que define balanceamento em um único lugar e `FabricaItem` para itens, ambas permitindo fácil extensão e carregamento de dados via JSON. O padrão Observer permitiu que múltiplos sistemas (log, UI, som, estatísticas, conquistas) reajam a eventos do jogo (morte, dano, colheita) sem conhecerem um ao outro, eliminando acoplamento e simplificando a adição de novos comportamentos. Juntos, Factory e Observer transformam um jogo de um sistema monolítico em um ecossistema modular e extensível, onde novos inimigos e novos observadores se adicionam sem modificar código existente.

Dica do Mestre: Factory Pattern é essencial em desenvolvimento real. Qualquer sistema que cria múltiplos objetos de tipos variados deve centralizar essa criação. Em aplicações web, factories criam modelos de banco de dados. Em sistemas de configuração, factories parseiam dados e constroem objetos. Em testes, factories criam fixtures. Observer é igualmente crucial: é a base de qualquer sistema event-driven profissional. Desde sistemas de notificação em apps até pipelines de data processing, Observer permite que sistemas desacoplados se comuniquem. O investimento em aprender esses padrões agora te preparará para código profissional em qualquer contexto.

Desafios da Masmorra

**Desafio 35.1. Implemente uma `FabricaSala` que lê definição JSON (tipo de sala, inimigos, loot) e gera uma sala completa com todos os inimigos criados via Factory.

**Desafio 35.2. Crie um `EventoSubirNivel` e um `ObservadorSubidaNivel` que toca som especial, mostra animação e escreve no log quando o herói sobe de nível.

**Desafio 35.3. Implemente um `ObservadorRegistroCombate` que armazena todos os eventos de combate em uma lista, permitindo "replay" de combates para debug (refaz cada ação em sequência).

**Desafio 35.4. Crie um `ObservadorPersistencia` que escuta `EventoMorteInimigo` e atualiza um JSON com estatísticas globais (inimigos mais perigosos, itens mais valiosos encontrados).

**Boss Final 35.5. Implemente um sistema de "Reações em Cadeia" onde um evento dispara eventos posteriores (morte -> loot -> colheita -> XP -> subida de nível -> conquista). Use `Future` e `Timer` para simular delays entre reações.

**Desafio 35.6. Implemente um `ObservadorFiltro` que permite observadores se inscreverem apenas em eventos que atendem certos critérios: - Exemplo: `bus.onComFiltro<EventoDanoAplicado>((evento) => evento.dano > 10)` só reage a dano acima de 10 - Implemente `Stream<T> onComFiltro<T extends EventoJogo>(bool Function(T) filtro)` no `BarramentoEventos` - Crie um `ObservadorDanoCrítico` que só reage quando o dano em um ataque é maior que 50% do HP máximo do alvo

**Desafio 35.7. Crie um `RegistroEventos` que persiste todos os eventos em um arquivo JSON de log: - Cada evento tem timestamp, tipo, e dados relevantes - Implemente `salvarJSON(String caminho)` e `carregarJSON(String caminho)` - Permita replay: leia o JSON, re-emita todos os eventos em sequência com os mesmos timings - Útil para debug e para mostrar "replay" de combates para jogador

**Desafio 35.8. Implemente uma `FabricaInimigoEvoluída` que carrega definições não apenas de um JSON estático, mas de múltiplos JSONs baseado em "temas" de andar (tema "undead" para andares 1-5, tema "demonios" para 6-10). Cada tema tem sua própria lista de inimigos com variações de poder. A Factory `carregarTema(String nomeArquivo)` carrega todas as definições de um arquivo JSON estruturado.

Factory transformou criação de inimigos em um processo escalável e orientado a dados. Observer transformou sistemas isolados em um ecossistema de reações elegante. Juntos, eles permitem crescimento sem acoplamento: novos observadores se adicionam sem modificar código anterior, balanço muda via JSON.

> *"A verdadeira elegância de um sistema reside não no que ele faz hoje, mas em quanto pode crescer amanhã sem quebrar o que já funciona."*

Próximo Capítulo

No Capítulo 36, você verá o último padrão crucial: o padrão **State** para **máquinas de estado finito** (FSM) que permitem IA verdadeiramente inteligente. Enquanto Factory e Observer resolvem criação (de forma centralizada) e reações (de forma desacoplada), State resolve comportamento complexo com transições claras e visuais — a inteligência que torna a IA adaptativa e interessante. Inimigos terão estados discretos (Patrulhando, Alerta, Perseguindo, Atacando, Fugindo) com transições explícitas baseadas em condições, tornando comportamento previsível mas estratégico para o jogador.

Capítulo 36 - Máquinas de Estado: Patrulha, Alerta e Perseguição

Um inimigo não ataca você instantaneamente quando te vê. Para um momento, alerta. Olha em volta. Se você sair de seu campo de visão, recua lentamente para patrulha. Se você se aproximar, começa a persegui-lo. Se você o ferir, persegue com fúria. Se você for muito forte, foge. Isto é uma máquina de estados: estados discretos, claramente nomeados, com transições explícitas. Cada estado é um objeto que sabe quando e como mudar para outro. Isso é mais organizado do que dizer "se X e se Y e se Z", porque os estados são visualizáveis, testáveis e extensíveis.

Neste capítulo você vai aprender o padrão State, transformando comportamento complexo em máquinas de estado finito (FSM). Cada inimigo terá estados como Patrulhando, Alerta, Perseguindo, Atacando e Fugindo.

O Problema: Comportamento Confuso

Sem máquinas de estado, o código fica confuso assim:

```
class Inimigo {
  bool viu_jogador = false;
  bool perto_jogador = false;
  int turnos_alerta = 0;
  bool fugindo = false;

  void executarTurno(Jogador alvo, MapaMasmorra mapa) {
    if (hp < 10 && viu_jogador) {
      fugindo = true;
      moverAoContrario(alvo);
    } else if (perto_jogador) {
```

```
      atacar(alvo);
    } else if (viu_jogador) {
      moverEm(alvo);
      turnos_alerta++;
      if (turnos_alerta > 5) {
        viu_jogador = false;
      }
    } else {
      patrulhar();
    }
  }
}
```

Problemas: - Estados implícitos (várias bools não formam um estado claro). - Transições obscuras (quando exatamente mudar de "alerta" para "patrulha"?). - Difícil de estender (novo estado quebra tudo). - Difícil de debugar (qual é o estado real?).

A Solução: State Pattern

Defina uma interface abstrata para estados:

Cada estado sabe como atualizar-se (transicionar para outro) e como agir (executar ação). Isso torna cada estado independente e testável. Um teste de "Patrulhando" não precisa saber de "Atacando". Cada um é uma máquina simples com regras claras.

```
abstract class EstadoIA {
  EstadoIA? atualizar(Inimigo self, Jogador alvo, MapaMasmorra mapa);
  Acao agir(Inimigo self, Jogador alvo, MapaMasmorra mapa);
  String get nome;
}
```

Cada estado retorna um novo estado (ou null se continua). A classe Inimigo muda de estado automaticamente:

Veja como `executarTurno` é simples: atualizar o estado, agir, pronto. O estado decide transições, a classe apenas obedece. Se um novo estado

retorna null, o inimigo permanece no estado atual; isso previne mudanças erráticas.

```
class Inimigo {
  late EstadoIA estado = Patrulhando([]);
  late Pos pos;
  late String nome;
  late int hp;

  void executarTurno(Jogador alvo, MapaMasmorra mapa) {
    var novoEstado = estado.atualizar(this, alvo, mapa);
    if (novoEstado != null) {
      print('$nome muda para ${novoEstado.nome}');
      estado = novoEstado;
    }

    var acao = estado.agir(this, alvo, mapa);
    acao.executar();
  }
}
```

Implementando Estados Concretos

Patrulhando

Patrulhando é o estado de repouso. O inimigo segue uma rota. Se detecta o alvo, passa para "Alerta". Caso contrário, continua patrulhando. Simples e previsível.

```
class Patrulhando implements EstadoIA {
  final List<Pos> rota;
  int indiceRota = 0;

  Patrulhando(this.rota);
```

```
  @override
  EstadoIA? atualizar(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    if (self.temLinhaDeVisao(alvo, mapa)) {
      return Alerta();
    }
    return null;
  }

  @override
  Acao agir(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    var proxPosicao = rota[indiceRota];
    if (self.pos == proxPosicao) {
      indiceRota = (indiceRota + 1) % rota.length;
      proxPosicao = rota[indiceRota];
    }

    var proxPasso = mapa.caminhoParaPos(self.pos, proxPosicao);
    return AcaoMover(self, proxPasso, mapa);
  }

  @override
  String get nome => "Patrulhando";
}
```

> **Nota:** O método `self.temLinhaDeVisao()` verifica se há linha de visão direta entre o inimigo e o alvo usando o algoritmo de Bresenham, sem paredes ou obstáculos bloqueando (implementado em `campo_visao.dart`, Capítulo 19). O método `mapa.caminhoParaPos()` retorna o próximo passo do caminho mais curto entre duas posições (implementado em `pathing.dart`, Capítulo 20).

Alerta

Alerta é um estado intermediário. O inimigo viu você, mas não tem certeza. Aguarda 3 turnos. Se você sair de visão, volta a patrulhar. Se se aproximar,

passa para perseguição ou ataque. É como em Zelda quando um inimigo te vê, pisca e fica em guarda antes de atacar.

```
class Alerta implements EstadoIA {
  int turnosAlerta = 0;

  @override
  EstadoIA? atualizar(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    if (!self.temLinhaDeVisao(alvo, mapa)) {
      turnosAlerta++;
      if (turnosAlerta > 3) {
        // Retorna ao patrulhamento com rota vazia (simplificado).
        // Num sistema real, poderia gerar rota aleatória ou
        // retomar a patrulha anterior. Com rota vazia, o inimigo
        // fica imóvel até detectar o jogador novamente.
        return Patrulhando([]);
      }
      return null;
    }

    turnosAlerta = 0;
    int distancia = mapa.distancia(self.pos, alvo.pos);
    if (distancia <= 1) {
      return Atacando();
    }

    return Perseguindo();
  }

  @override
  Acao agir(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    return AcaoAguardar(self);
  }

  @override
  String get nome => "Alerta";
```

```
}
```

> **Nota:** O método `mapa.distancia()` calcula a distância Manhattan entre duas posições, usada para determinar se o inimigo está próximo o suficiente para atacar (implementado em `mapa.dart`, Capítulo 12).

Perseguindo

Perseguindo é comprometido. O inimigo está atrás de você. Se você sair de visão, volta para alerta. Se ficar perto demais, passa para ataque. Se ficar muito ferido, foge. É o "combate em movimento": nem está descansando, nem atacando diretamente.

```
class Perseguindo implements EstadoIA {
  @override
  EstadoIA? atualizar(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    if (!self.temLinhaDeVisao(alvo, mapa)) {
      return Alerta();
    }

    int distancia = mapa.distancia(self.pos, alvo.pos);
    if (distancia <= 1) {
      return Atacando();
    }

    if (self.hp < (self.hpMax * 30 / 100)) {
      return Fugindo();
    }

    return null;
  }

  @override
  Acao agir(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
```

```
    var proxPasso = mapa.caminhoParaPos(self.pos, alvo.pos);
    return AcaoMover(self, proxPasso, mapa);
  }

  @override
  String get nome => "Perseguindo";
}
```

> **Nota:** O método `mapa.caminhoParaPos()` retorna o próximo passo do caminho mais curto, essencial para fazer o inimigo se mover inteligentemente em direção ao alvo sem atravessar paredes (implementado em `pathing.dart`, Capítulo 20).

Atacando

Atacando é o engajamento total. O inimigo está ao seu lado (1 tile) e batendo. Se você se afasta, volta a perseguir. Se fica muito ferido, foge. Caso contrário, continua atacando.

```
class Atacando implements EstadoIA {
  @override
  EstadoIA? atualizar(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    int distancia = mapa.distancia(self.pos, alvo.pos);

    if (distancia > 1) {
      return Perseguindo();
    }

    if (self.hp < (self.hpMax * 25 / 100)) {
      return Fugindo();
    }

    return null;
  }
```

```
  @override
  Acao agir(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    return AcaoAtacar(self, alvo);
  }

  @override
  String get nome => "Atacando";
}
```

Fugindo

Fugindo é retirada. O inimigo anda longe de você, tentando se regenerar. Se HP regenera, volta a perseguir. Se passa muito tempo fugindo (turnos_fuga > 10), desiste e volta a patrulhar. Nenhum inimigo foge eternamente.

```
class Fugindo implements EstadoIA {
  int turnosFuga = 0;

  @override
  EstadoIA? atualizar(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    turnosFuga++;

    if (self.hp > (self.hpMax * 60 / 100)) {
      return Perseguindo();
    }

    if (turnosFuga > 10) {
      // Retorna ao patrulhamento com rota vazia (simplificado).
      // Num sistema real, poderia gerar rota aleatória ou retomar
      // a patrulha anterior. Com rota vazia, o inimigo fica imóvel
      // até detectar o jogador novamente.
      return Patrulhando([]);
    }
```

```
    return null;
  }

  @override
  Acao agir(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    var fuga = mapa.caminhoParaPos(
      self.pos,
      alvo.pos,
      inverso: true,
    );
    return AcaoMover(self, fuga, mapa);
  }

  @override
  String get nome => "Fugindo";
}
```

> **Nota:** O método `mapa.caminhoParaPos()` com `inverso: true` retorna o próximo passo na direção *oposta* ao alvo, implementando assim um comportamento inteligente de fuga (em vez de apenas mover aleatoriamente). Esse mecanismo evita que inimigos fuja eternamente: após 10 turnos de fuga, retorna ao patrulhamento (implementado em `pathing.dart`, Capítulo 20).

Diagrama de Transições

Ciclo de vida de um inimigo (FSM). A fonte editável do diagrama está em `assets/diagrams/capitulo-036-fsm-transicoes.mmd`; o PNG é gerado em `./scripts/build.sh` com Node.js/npx (`@mermaid-js/mermaid-cli`).

Fases de Boss com FSM

Um chefe inteligente tem fases que são estados:

Um boss não é estático. Conforme você o machuca, ele muda. Primeira fase: caminha em sua direção. Segunda fase (quando perde 50% de HP):

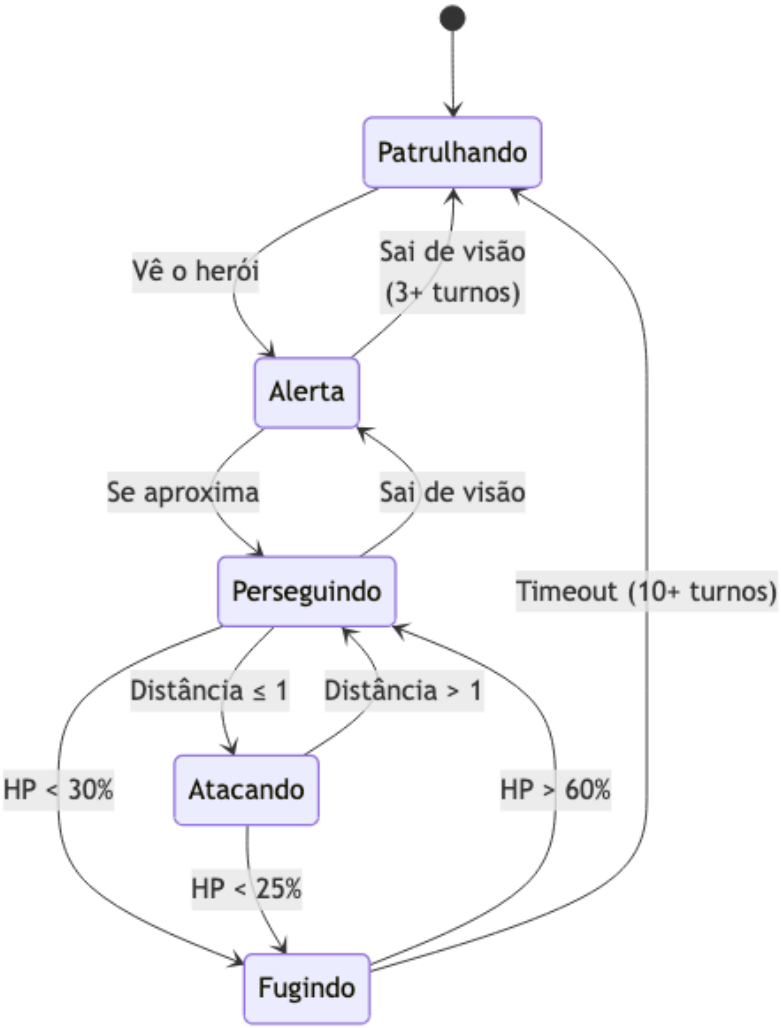


Figura 2: Diagrama de transições da FSM do inimigo

ataca com força dobrada. Isso é exatamente o padrão State: cada fase é um estado com comportamento diferente. Transição automática baseada em HP.

```
class BossFaseUm implements EstadoIA {
  @override
  EstadoIA? atualizar(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    if (self.hp < (self.hpMax * 50 / 100)) {
      print('${self.nome} entra em fúria! Fase 2!');
      return BossFaseDois();
    }
    return null;
  }

  @override
  Acao agir(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    var proxPasso = mapa.caminhoParaPos(self.pos, alvo.pos);
    return AcaoMover(self, proxPasso, mapa);
  }

  @override
  String get nome => "BossFaseUm";
}

class BossFaseDois implements EstadoIA {
  @override
  EstadoIA? atualizar(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    return null;
  }

  @override
  Acao agir(Inimigo self, Jogador alvo, MapaMasmorra mapa) {
    int dano = calcularDano(self.arma, alvo.defesa) * 2;
    return AcaoAtacarEspecial(self, alvo, dano);
  }

  @override
```

```
  String get nome => "BossFaseDois";
}
```

É como em Dark Souls: cada padrão de ataque é um estado. O chefe muda conforme você o danifica.

Feedback Visual: Símbolo Muda por Estado

Para o jogador entender em que estado o inimigo está:

O símbolo que renderiza no mapa muda dinamicamente baseado no estado. Um 'z' patrulhando, um 'z!' em alerta, 'Z!!' atacando, 'z...' fugindo. O jogador lê o mapa e instantaneamente entende o estado de cada inimigo. Feedback visual é essencial em jogos; o jogador não consegue ler seu código, mas consegue entender um símbolo.

```
class Inimigo {
  String get simbolo {
    return switch (estado) {
      Patrulhando() => 'Z',
      Alerta() => 'z',
      Perseguindo() => 'z!',
      Atacando() => 'Z!!',
      Fugindo() => 'z...',
      _ => '?',
    };
  }
}
```

Agora o jogador vê um "z" (patrulhando) virar "Z!!" (atacando) ao ser descoberto.

Comparação: Antes vs. Depois

Antes

O código com if/else aninhados é impossível de seguir. Quantos estados há? Não está claro. Como vai de um para outro? Você tem que ler cada condição, manter na cabeça, rastrear lógica. É mental exaustão.

```
if (viu && perto) atacar();
else if (viu && longe) perseguir();
else if (viu && !perto) aguardar();
else if (!viu && turnos > 5) patrulhar();
```

Depois

Aqui, a máquina de estados é explícita. Cada estado é uma classe independente. Transições são claras (o método `atualizar` retorna um novo estado ou null). O código de execução não muda; é sempre "atualizar, depois agir". Elegante, testável, extensível.

```
var novoEstado = estado.atualizar(this, alvo, mapa);
if (novoEstado != null) estado = novoEstado;
var acao = estado.agir(this, alvo, mapa);
```

Claro, lógico, testável.

Pergaminho do Capítulo

Neste capítulo você aprendeu o padrão State, transformando comportamento complexo e difícil de manter em máquinas de estado finito explícitas e testáveis. Em vez de múltiplos booleanos e if/else aninhados, cada inimigo possui um único `estado` que define seu comportamento e transições. Implementou cinco estados (Patrulhando, Alerta, Perseguindo, Atacando, Fugindo) onde cada um sabe quando transicionar para o próximo baseado em condições claras (distância, linha de visão, HP). Viu como o padrão State torna comportamento visual; o símbolo do inimigo no mapa muda conforme

muda o estado, dando feedback instantâneo ao jogador. Finalmente, aplicou State a chefes multi-fases para criar adversários adaptativos que mudam tática conforme você os danifica, como em Dark Souls.

Dica do Mestre: O padrão State é fundamental em qualquer sistema que tem múltiplos modos de operação. Máquinas de vendas, compiladores, games, interfaces de aplicação; todos usam State internamente. A vantagem principal é que cada estado é isolado e testável: você consegue testar o comportamento de "Patrulhando" sem saber de "Atacando". Máquinas de estado são também visualmente debugáveis; você consegue desenhar um diagrama e comparar com o código. Em desenvolvimento profissional, quando você se deparar com uma classe cheia de booleans e condicionais, pense em State. Seu código futuro (e seus colegas de time) vão agradecer a clareza.

Desafios da Masmorra

**Desafio 36.1. Implemente um estado `Confuso` onde o inimigo anda aleatoriamente durante 5 turnos. Adicione uma estratégia que pode fazer inimigo entrar em pânico quando toma dano crítico.

**Desafio 36.2. Crie um estado `Regenerando` para um inimigo especial que, quando sua HP fica baixa, entra em um ciclo de regeneração onde sua HP sobe 2 por turno durante 8 turnos. Se tomar dano, sai deste estado.

**Desafio 36.3. Implemente um sistema de "Agressão Escalada" onde cada hit que você acerta incrementa um contador que faz o chefe passar por fases mais cedo (fase 2 em 40% de HP em vez de 50%).

**Desafio 36.4. Crie um estado `SaltoEspecial` onde um inimigo pula para uma posição aleatória e depois volta. Integre com a FSM de forma que o inimigo escolhe esse estado aleatoriamente durante perseguição.

**Desafio 36.5. (Desafio). Implemente uma máquina de estado de "Comportamento Imprevisível" onde o boss tem 5 estados (Atacando, Fugindo, Invulnerável, ChamandoMinions, Regenerando), cada um com 30% de chance de ser escolhido quando o anterior termina. Adicione debug logging de cada mudança de estado.

**Desafio 36.7. Implemente um estado `Confuso` com o seguinte comportamento: - O inimigo anda aleatoriamente durante exatamente 5 turnos - A cada turno, há 10% de chance de se recuperar (transicionar de volta para `Alerta`) - Se o inimigo é atacado enquanto confuso, sai do estado imediatamente e vai para `Atacando` - Adicione uma estratégia que faz o inimigo

entrar em pânico quando toma dano crítico (> 50% do HP em uma ação), transitando para `Confuso` por 3 turnos como reflexo defensivo

**Desafio 36.8. Crie um estado `VigiaEspecial` que transiciona automaticamente para `Confuso`: - Este estado representa um inimigo em alta alerta que vai ficar confuso se não conseguir atacar por muito tempo - Se o inimigo não consegue atingir o herói em linha reta por 4 turnos consecutivos (está bloqueado), fica confuso - Use um contador interno `turnosBloquados` que incrementa cada turno se o inimigo não tem linha reta para o herói - Implemente transição: `VigiaEspecial` -> `Confuso` (quando contador atinge 4) -> `Alerta` (após 5 turnos confuso)

**Desafio 36.9. Implemente um estado `BossEmFuria` que: - Só é acessível quando o boss tem menos de 20% de HP (estado enraivecido) - Neste estado, o boss ignora linha de visão e persegue o herói por toda a masmorra - Ataque a cada turno (não aguarda) com dano aumentado em 50% - Após 10 turnos de fúria, o boss explode e morre (mecanismo de limite de tempo para evitar combates infinitos) - Transição visual: símbolo muda para `D!!!` (Dragão em fúria) quando entra neste estado

**Boss Final 36.10. Volte ao Capítulo 34 (Strategy) e substitua os `if/else` aninhados que controlam a IA dos inimigos por uma máquina de estados completa. Implementa estados para o Lobo: Patrulhando (começa aqui), Alerta (quando você está a 5 tiles de distância e tem linha de visão), Perseguindo (quando está a 3 tiles), Atacando (quando está ao lado), Fugindo (quando HP < 25%). O símbolo do Lobo muda a cada estado no mapa: `L` patrulhando, `L?` alerta, `L!` perseguindo, `L!!` atacando, `L..` fugindo. Rode o jogo e observe como a máquina de estados torna o comportamento do inimigo previsível mas estratégico: você consegue "ler" o mapa pelo símbolo e saber exatamente em que estado cada Lobo está.

O padrão State transformou comportamento complexo em máquinas de estado explícitas. Agora inimigos têm "vidas" que você consegue visualizar e entender. Cada transição é clara. Cada estado é testável isoladamente.

> *"Uma máquina de estado bem feita é como um roteiro bem escrito: cada cena conhece a próxima, cada personagem conhece seu papel nesta cena, e a audiência acompanha cada passo."*

Próximo Capítulo

No Capítulo 37, você consolidará tudo que aprendeu sobre design patterns e assincronismo numa aplicação completa. Verá como State, Factory, Observer e Strategy trabalham juntos para criar um ecossistema de jogo coeso. Polirá a interface, salvará o jogo com persistência robusta, e estará pronto para mostrar seu projeto ao mundo como um exemplo de engenharia de software profissional em Dart.

Capítulo 37 - Síntese: O Jogo Completo, Polido e Pronto

> *Você começou com um `print('Olá')`. Agora tem um roguelike funcional com IA inteligente, máquinas de estado, padrões de design profissionais, save/load, e uma economia de jogo. Isso não é um exercício. É um artefato real. Um produto que você construiu do zero e que pode mostrar com orgulho. E mais ainda, é uma porta aberta. Dart conhece. Design patterns você domina. O mundo inteiro está esperando.*

Neste capítulo final você vai ver a visão geral completa do que construiu, polirá a interface visual, revisitará todos os padrões de design em contexto, e aprenderá os próximos passos.

O Que Você Construiu

Começou simples:

```
Capítulo 1-5:      Fundamentos Dart (variáveis, operadores, listas)
Capítulo 6-10:     Controle (if/else, loops, funções, closures)
Capítulo 11-14:    OOP (classes, herança, mixins, enums)
Capítulo 15-21:    2D, ASCII, geração procedural, *dungeon crawl*
Capítulo 22-27:    Economia, loja, progressão, chefe, jogo completo
Capítulo 28-33:    Refatoração, testes, async, save/load, organização
Capítulo 34-36:    Padrões (Strategy, Command, Factory, Observer)
Capítulo 37:       Síntese, polimento, próximos passos
```

Resultado: um *roguelike* em terminal, jogável, com:

- Dungeon procedural infinita
- Inimigos com IA inteligente (patrulha, alerta, perseguição, fuga)
- Chefe multi-fases
- Combate tático (ataque, defesa, magia)

- Economia (loja, ouro, itens)
- Progressão (XP, níveis, habilidades)
- Save/load persistente
- Testes automatizados
- Código profissional e documentado

Isto é um produto real.

Saída Esperada

Aqui está como o jogo final parece em execução, do menu até a jornada:

```
                    M A S M O R R A  A S C I I

                   Um Roguelike em Dart, por Você


                   [ Pressione ENTER para começar ]


MASMORRA ASCII - Menu Principal
──────────────────────────────────────────────

  [1] Novo Jogo
  [2] Continuar
  [3] Créditos
  [4] Sair

──────────────────────────────────────────────

> 1

 ╔═══════════════════════════════════════╗
 ║    ESCOLHA SEU NÍVEL DE DIFICULDADE   ║
 ╠═══════════════════════════════════════╣
 ║ [1]  RECRUTA (recomendado iniciante) ║
 ║     +50% XP, inimigos -20% saúde      ║
```

```
╠                                          ╣
╠ [2]   NORMAL (balanço perfeito)      ╣
╠     1x XP, dificuldade média           ╣
╠                                          ╣
╠ [3]  VETERANO (para desafiadores)    ╣
╠     -50% XP, inimigos +30% saúde      ╣
╚═══════════════════════════════════════════╝

Escolha (1-3): 2

Dificuldade: NORMAL

╔═══════════════════════════════════════════╗
╠        CRIE SEU PERSONAGEM             ╣
╚═══════════════════════════════════════════╝

Qual é o nome do seu herói? Guerreiro

Bem-vindo, Guerreiro!
Sua jornada na masmorra começa agora...

Andar 1/5: A Descida Inicial

┌──────────────────────────────────────┐
│ @ . . . . . . . . . . . . . . . . . │
│ . . Z . . . . . . . . . . . . . . . │
│ . . . . . . . . . . . . . . . . . . │
│ . . . . . . . . . . . . . . . . . > │
└──────────────────────────────────────┘

┌─ GUERREIRO ──────────────────────────┐
│ HP: 50/50  |████████████████████████████| │
│ XP: 0/100  |▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒| │
│ Nível: 1   Ouro: 0                   │
│ Inventário: 0/10                     │
└──────────────────────────────────────┘
```

```
> d
> d
> d
> a

[Combate com zumbi]

Você ataca o Zumbi por 8 dano!
O Zumbi ataca você por 3 dano! (HP: 47/50)

Você ataca o Zumbi por 7 dano!
O Zumbi está derrotado!

Você ganhou 25 XP e 15 ouro!

[Após descida]

Andar 5: A Câmara do Rei da Masmorra

┌──────────────────────────────────────┐
│ . . . . . . . . . . . . . . . . . . . │
│ . . . . . . . . . . . . . . . . . . . │
│ . . . . . . . . @ . . . . . K . . . . │
│ . . . . . . . . . . . . . . . . . . . │
└──────────────────────────────────────┘

┌─ GUERREIRO ──────────────────────────┐
│ HP: 35/50  |████████████▒▒▒▒▒▒▒▒▒▒▒▒| │
│ XP: 287/300|████████████████████▒▒▒▒| │
│ Nível: 8   Ouro: 2.350               │
│ Inventário: 6/10                     │
└──────────────────────────────────────┘

[Combate épico com o chefe]
```

```
REI DA MASMORRA - Fase 1
HP: 60/100 |████████▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒|

Você ataca por 12 dano!
O Rei contra-ataca por 8 dano! (HP: 27/50)

[...]

VITÓRIA!

Parabéns, aventureiro! Você derrotou
o Chefão da Masmorra!

Estatísticas da Jornada:
  Andares: 5/5
  Inimigos derrotados: 42
  Ouro coletado: 2.350
  Turnos totais: 287
  Nível final: 8

Conquistas desbloqueadas:
  ★ Primeira Vitória
  ★ Caçador de Tesouros
  ★ Exterminador

Deseja jogar novamente? (s/n)
> s
```

Este fluxo mostra: - **Splash screen** inicial com título - **Menu principal** com 4 opções claras - **Seleção de dificuldade** com descriptions - **Criação de personagem** personalizada - **Loop de exploração** com renderização de mapa ASCII, HUD dinâmico, combate por turnos - **Progressão** (XP, níveis, ouro) - **Tela de vitória** com estatísticas e conquistas - **Retorno ao menu** para nova partida

A arquitetura permite todos esses elementos funcionarem de forma harmônica através do coordenador central `MasmorraAscii`.

Polimento Visual: Menu Aprimorado

O polimento visual é a diferença entre um jogo "feito" e um jogo "profissional." Não é só código funcionando; é experiência coerente do começo ao fim. Um menu bem organizado, uma splash screen clara, créditos que reconhecem o trabalho—tudo comunica que você não apenas codificou, você *designou*.

Splash Screen ASCII

Uma *splash screen* é a primeira coisa que o jogador vê. É a porta de entrada do seu jogo. Pode ser simples (texto e bordas ASCII), mas deve ser bem feita. Limpe a tela, desenhe arte ASCII criativa, explique o que é o jogo. Faz diferença psicológica real na experiência.

```
// lib/tela_titulo.dart
void mostrarSplash() {
  limparTela();
  print('');
  print('');
  // ← espaçamento visual para destaque
  print('          M A S M O R R A  A S C I I');
  print('');
  print('       Um Roguelike em Dart, por Você');
  print('');
  print('');
  print('       [ Pressione ENTER para começar ]');
  print('');
}
```

Menu Principal

O menu é onde o jogador controla o fluxo do jogo. Novo jogo, continuar, créditos, sair. Cada opção executa uma ação diferente. O menu deve ser simples de navegar, claro visualmente, e robusto contra entrada inválida. Um bom menu comunica: "você tem controle, e este é um jogo profissional."

```
// lib/menu_principal.dart
void mostrarMenu() {
  while (true) {
    limparTela();
    print('');
    print('MASMORRA ASCII - Menu Principal');
    print('=' * 47);
    print('');
    // ← número entre chaves para destaque
    print('  [1] Novo Jogo');
    print('  [2] Continuar');       // ← fácil de ler, acionável
    print('  [3] Créditos');
    print('  [4] Sair');
    print('');
    print('=' * 47);
    print('');

    var opcao = stdin.readLineSync();

    switch (opcao) {
      case '1':
        iniciarNovoJogo();
        break;
      case '2':
        carregarJogo();
        break;
      case '3':
        mostrarCreditos();
        break;
      case '4':
        exit(0);
      default:
        print('Opção inválida');  // ← feedback para entrada ruim
    }
  }
}
```

```dart
// lib/creditos.dart
void mostrarCreditos() {
  limparTela();

  // ← Créditos não são só cortesia. Documentam a jornada.
  // "Programação e Design: Você" não é modéstia — é verdade.
  // Você construiu isso do zero. Agradeça influências. Créditos
  // bem feitos fazem o jogo parecer profissional e reflexivo.

  print('');
  print('CRÉDITOS - Masmorra ASCII');
  print('=' * 45);
  print('');
  print('Programação e Design: Você');  // ← reconheça seu trabalho
  print('');
  print('Agradecimentos especiais a:');
  print('  - Dart, por ser incrível');
  print('  - Design Patterns, por nos tornar melhores');
  print('  - Você, por persistir até aqui');
  print('');
  print('Esta jornada de 36 capítulos te ensinou:');
  print('  - Fundamentos de Dart');
  print('  - Pensamento orientado a objetos');
  print('  - Padrões de design profissionais');
  print('  - Desenvolvimento de jogos');
  print('  - Como fazer código que dura');
  print('');
  print('Parabéns. Você não é mais iniciante.');
  print('');
  print('[ Pressione ENTER para voltar ]');
  print('');

  stdin.readLineSync();
}
```

Arquitetura Completa: MVC em Ação

Você construiu um *roguelike* profissional seguindo padrão *MVC* (Model-View-Controller), mesmo que de forma natural. O **Model** é toda lógica (EstadoJogo, MapaMasmorra, Jogador, Inimigo, Item, IA, combate). A **View** é TelaAscii renderizando. O **Controller** é LoopJogo orquestrando entradas e fluxo. Essa separação é poderosa: você *pode* trocar a View (substituir ASCII por Flutter sem mudar Model), ou adicionar Controllers alternativos (API HTTP para multiplayer) sem tocar em nada mais.

Aqui está como tudo se conecta:

Observe como cada componente tem responsabilidade clara e comunica-se via barramento de eventos. `LoopJogo` é o orquestrador central (**Controller**). Ele gerencia `EstadoJogo` (dados), `MapaMasmorra` (dungeon), `Jogador` (você), lista de `Inimigo` (adversários), lista de `Item` (loot), `BarramentoEventos` (reações), `GerenciadorSalve` (persistência) como **Model**. E `TelaAscii` é a **View** (renderização). Cada componente é independente e reutilizável. `LoopJogo` conecta tudo. A fonte editável do diagrama está em `assets/diagrams/capitulo-037-arquitetura-mvc.mmd`; o PNG é gerado em `./scripts/build.sh` com Node.js/npx (`@mermaid-js/mermaid-cli`).

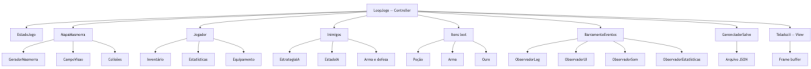


Figura 3: Arquitetura MVC: LoopJogo, modelo e TelaAscii

Revisita aos Padrões de Design: Sinfonia Harmônica

Você aprendeu 5 padrões que trabalham em harmonia. Cada um resolve um problema específico. Juntos, tornam código flexível, testável e extensível. Este é o coração da engenharia profissional.

1. Strategy: Comportamento Plugável

Cada inimigo tem uma `EstrategiaIA` que pode ser trocada em tempo de execução. Um lobo agressivo pode virar covarde se ferido—sem tocar no código de `Inimigo`.

Strategy permite que comportamento seja desacoplado da classe. Você não modifica `Inimigo` para mudar IA; você apenas troca sua estratégia. É

como trocar um cartucho em um videogame: o console fica igual, o comportamento muda. Elegante. Extensível. Testável.

```
var lobo = Inimigo(
  nome: "Lobo",
  estrategia: IAAgressiva(),  // ← comportamento plugável
);

if (lobo.hp < lobo.hpMax * 30 / 100) {
  lobo.estrategia = IACovardia();  // ← muda sem modificar Inimigo
}
```

Vantagem: Comportamento desacoplado da classe. Novo comportamento? Cria nova estratégia. Código existente não muda.

2. Command: Ações Reversíveis

Cada ação é um objeto que pode ser executado e desfeito. Permite replay, undo e histórico completo.

Command encapsula uma solicitação como um objeto. Permite manter histórico completo do jogo, desfazer movimentos (útil para testes e debug), e replay de combates. Sem *Command*, desfazer seria impossível; você teria que guardar snapshots do estado inteiro (explosão de memória). Com *Command*, cada ação sabe como reverter a si mesma.

```
var acao = AcaoAtacar(inimigo, heroi);  // ← ação encapsulada
acao.executar();                         // ← "faça isso"
gerenciador.executar(acao);              // ← registra para histórico
// ← volta atrás (se necessário)
acao.desfazer();
```

Vantagem: Histórico completo e capacidade de undo sem explosão de memória.

3. Factory: Criação Centralizada

Todos os inimigos e itens são criados por *factories*, não espalhados no código.

Factory centraliza conhecimento de como criar objetos. Balanço, parâmetros, configuração: tudo em um lugar. Se mudar o HP de um zumbi, muda em um lugar. Quer suportar carregar config de JSON? Muda em um lugar. *Factory* também permite testes: você pode facilmente criar inimigos de teste sem saber todos os detalhes de construção. É encapsulamento em escala de criação de objetos.

```
// ← criação centralizada
var inimigo = FabricaInimigo.criarAleatorio(andar: 3);
// ← sem duplicação
var item = FabricaItem.criar('pocao_vida');
```

Vantagem: Balanço e configuração centralizados. Fácil de modificar, testar, carregar de config externa.

4. Observer: Reações Desacopladas

Quando algo acontece, múltiplos observadores reagem independentemente.

Observer permite que sistemas isolados se comuniquem sem conhecer um ao outro. Um inimigo morre: log registra, *UI* pisca, som toca, conquistas verificam. Nenhum deles conhece os outros; todos ouvem o barramento de eventos. Quer adicionar um novo observador (por exemplo, transmissão ao servidor)? Cria e registra. Zero mudança no código de combate. Isso é profissionalismo.

```
// ← evento
bus.emitir(EventoMorteInimigo(inimigo: goblin, matador: heroi));

obsLog.ouve(evento);            // ← observador 1: registra log
obsUI.ouve(evento);             // ← observador 2: atualiza HUD
obsSom.ouve(evento);            // ← observador 3: toca som
obsEstatisticas.ouve(evento);   // ← observador 4: atualiza stats
```

Vantagem: Novos observadores sem modificar código existente. Escalabilidade real.

5. State: Máquinas de Estado

Comportamento complexo modelado como estados discretos com transições explícitas.

State torna IA inteligível. Cada estado é uma classe com regras claras. Transições são explícitas e visuais. O comportamento é testável. Sem *State*, você tem if/else aninhados incompreensíveis (código espaguete). Com *State*, você tem um diagrama visual que o jogador *lê na tela* via símbolos que mudam (`z` patrulhando, `z!` alerta, `Z!!` atacando).

```
// ← avalia transição
var novoEstado = estado.atualizar(this, alvo, mapa);
if (novoEstado != null) {
  this.estado = novoEstado;  // ← muda de estado
}
// ← executa ação do estado
var acao = this.estado.agir(this, alvo, mapa);
```

Vantagem: IA clara, visual, testável, extensível, e debugável (você vê o estado no mapa).

O que Você Aprendeu em Dart

Você agora domina a linguagem em profundidade:

- **Fundamentos**: Variáveis, tipos, *null safety*, operadores e expressões
- **Coleções**: Strings, listas, mapas, e manipulação eficiente
- **Controle**: if/else, loops, pattern matching, switch avançado
- **Funções**: Funções de primeira classe, *closures*, currying, callbacks
- **OOP**: Classes, construtores, getters/setters, herança, *mixins*, enums, *sealed classes*
- **Generics**: Parâmetros de tipo, restrições, variância
- **Async**: *Async/await*, *Futures*, *Streams*, processamento assíncrono real
- **Exceções**: Try/catch/finally, tipos customizados, propagação
- **JSON**: Serialização e desserialização, conversão de tipos
- **Testes**: *Test package*, asserções, mocks, *golden tests*
- **Padrões**: Toda a engenharia de software (5 design patterns, *MVC*, arquitetura)

Isto é Dart profissional. Você não é mais novato; é desenvolvedor.

Próximos Passos

Agora que você domina Dart e design patterns, três caminhos se abrem:

Flutter: Do Terminal para Mobile/Desktop

Seu *roguelike* é terminal. Mas Dart também alimenta Flutter, um *framework* para criar apps mobile e desktop com qualidade de produção.

Toda a lógica de jogo que você construiu roda **independente de interface**. Isso é o poder de arquitetura desacoplada. Migrar para Flutter é mover apenas a camada de apresentação (View)—o motor inteiro (`LoopJogo`, IA, economia, save/load) permanece intacto e reutilizável. Você apenas substitui renderização ASCII por *widgets* Flutter.

```
// Exemplo: migração para Flutter (mesmo Model, nova View)
void main() {
  runApp(MasmorraApp());
}

class MasmorraApp extends StatefulWidget {
  @override
  _MasmorraAppState createState() => _MasmorraAppState();
}

class _MasmorraAppState extends State<MasmorraApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Masmorra ASCII',
      home: MasmorraScreen(),  // ← nova View, mesmo Model
    );
  }
}
```

Backend & Multiplayer

Seu jogo é single-player. Mas Dart pode fazer backend robusto com `Shelf` ou `Serverpod`. Multiplayer? Sincronização de estado? Leaderboards? Tudo possível.

```
// Exemplo: servidor com rotas (*endpoints*)
final handler = Router()
  ..get('/api/leaderboard', _leaderboardHandler)  // ← GET leaderboard
  // ← POST para salvar no servidor
  ..post('/api/save', _saveHandler);
```

Seu Model funciona tal qual; é apenas mais um Controller consumindo-o.

pub.dev: Partilhar com a Comunidade

Publique seu código como *package*. Compartilhe com a comunidade. Seus padrões, sua IA, sua renderização—tudo reutilizável.

```
dart pub publish
```

Seu jogo vira uma biblioteca. Alguém o usa como base para seu próprio roguelike. Assim funciona engenharia colaborativa.

Por Que Esses 5 Padrões?

Você aprendeu 5 padrões porque eles resolvem 80% dos problemas reais em software. *Strategy* aparece toda vez que você tem algoritmos trocáveis (IA, preços, recomendações). *Command* aparece em undo/redo, *macros*, fila de tarefas. *Factory* aparece em criação de múltiplos objetos correlatos (banco de dados, serialização, injeção de dependência). *Observer* aparece em notificações, log, *webhooks*. *State* aparece em máquinas de estado por toda parte (fluxo de pedidos, workflows, gaming). Você não está aprendendo abstração; está aprendendo linguagem universal de engenharia.

Por que não mais padrões? Cada padrão adicional é complexidade. Os 5 que você aprendeu escalam para 99% de projetos. Tentar usar todos os 23 padrões Gang of Four é como ter 23 chaves para um problema. Você

usa a certa. Profissionais dominam uns 10 bem, e sabem quando alcançar por eles.

Pergaminho do Capítulo

Neste capítulo final você revisitou tudo que construiu ao longo de 36 capítulos: de um simples `print('Olá')` até um *roguelike* funcional com IA inteligente, padrões de design profissionais, save/load, economia e mais. Viu a arquitetura completa seguindo *MVC*, como `LoopJogo` orquestra `MapaMasmorra`, `Jogador`, `Inimigos`, `Items`, `BarramentoEventos`, `GerenciadorSalve` e `TelaAscii`. Revisitou os cinco padrões de design aprendidos (*Strategy* para comportamentos plugáveis, *Command* para ações reversíveis, *Factory* para criação centralizada, *Observer* para reações desacopladas, *State* para máquinas de estado) e viu como trabalham juntos em sinfonia. Entendeu que aprendeu Dart profissional e está pronto para os próximos passos: Flutter, networking, publicação em pub.dev.

O Jogo Até Aqui

Ao final desta parte, seu jogo *roguelike* completo e polido no terminal se parece com isto:

```
VITÓRIA!

Parabéns, aventureiro! Você derrotou
o Chefão da Masmorra!

Estatísticas da Jornada:
  Andares: 5/5
  Inimigos derrotados: 42
  Ouro coletado: 2.350
  Turnos totais: 287
  Nível final: 8

Conquistas desbloqueadas:
  Primeira Vitória
```

```
  Caçador de Tesouros
  Exterminador

Deseja jogar novamente? (s/n)
>
```

Cada parte adiciona novas camadas ao jogo. Compare com o início e veja o quanto você evoluiu!

Desafios da Masmorra

**Desafio 37.1. Crie uma tela de "Game Over" que mostra estatísticas do jogo (total de mortes infligidas, ouro coletado, maior profundidade alcançada, tempo jogado). Salve em JSON.

**Desafio 37.2. Implemente um sistema de "Conquistas" que desbloqueiam marcos (primeira morte, 10 mortes, 100 mortes, derrotar o chefe). Mostre na tela.

**Desafio 37.3. Adicione um "Modo Desafio" onde o jogo é mais difícil (inimigos mais fortes, menos poções encontradas). Acompanhe recordes em um JSON de leaderboard.

**Desafio 37.4. Crie uma "Enciclopédia de Inimigos" acessível no menu que mostra cada tipo encontrado, com estatísticas (HP, dano, estratégia).

**Boss Final 37.5. Refatore todo o jogo para usar um padrão MVC (Model-View-Controller) limpo. O Model é EstadoJogo e lógica. O View é TelaAscii. O Controller é LoopJogo. Adicione um segundo "backend" Controller que permite jogar via API HTTP sem a tela.

Reflexão Final

Você chegou aqui. Trinta e seis capítulos, de "print('Olá')" até um *roguelike* profissional. Isso não é pouco. Isso é uma jornada real de aprendizado em:

- Linguagem (Dart)
- Engenharia (arquitetura, padrões)
- Design (jogos, UX, UI)
- Persistência (você continuou mesmo quando ficou difícil)

Você não é mais iniciante. Você é um desenvolvedor. Seu código é profissional. Seus projetos têm fundações sólidas.

Daqui em diante, tudo que você construir será melhor porque você entende os princípios. Novos padrões serão fáceis. Novos desafios serão degraus, não paredes.

Bem-vindo ao outro lado.

Dica Profissional - O Valor do Que Você Construiu:

Você fez algo real. Não um tutorial copiado, não um exemplo genérico. Um jogo completo com IA inteligente, economia funcional, save/load persistente, padrões de design aplicados, e código testável. No mercado, isso é um portfolio profissional. Isso é prova de competência.

Como Continuar Aprendendo:

Agora que domina Dart e design patterns, o próximo passo é especialização. Escolha um caminho: Flutter para mobile/desktop, Shelf/Serverpod para backend, ou aprofunde em algoritmos de IA (árvores de decisão, redes neurais). Cada novo domínio usará os fundamentos que você construiu aqui. Design patterns não mudam. Arquitetura desacoplada não muda. O que muda é o contexto de aplicação.

Aplicações Real-World:

Tudo que você aprendeu existe em produção: - **Factory**: Todo sistema que cria múltiplos objetos a partir de configuração (CMS, e-commerce, SaaS). - **Observer**: Toda notificação em tempo real (chat, alertas de mercado, analytics). - **Strategy**: Toda IA ou algoritmo que pode mudar em runtime (recomendações, preços dinâmicos). - **Command**: Todo sistema que precisa de undo/redo (editores, controle de versão). - **State**: Toda máquina de estado (fluxo de pedidos, workflow, gaming).

Você não está aprendendo abstração. Você está aprendendo linguagem universal de engenharia profissional.

Recursos para Continuar

Documentação e Comunidade: - Documentação oficial Dart — referência authorizada - Flutter — framework para mobile/desktop - Pub.dev (packages) — repositório de bibliotecas - Comunidade Dart: Discord (Dart Community), GitHub (dart-lang), Stack Overflow (tag: dart)

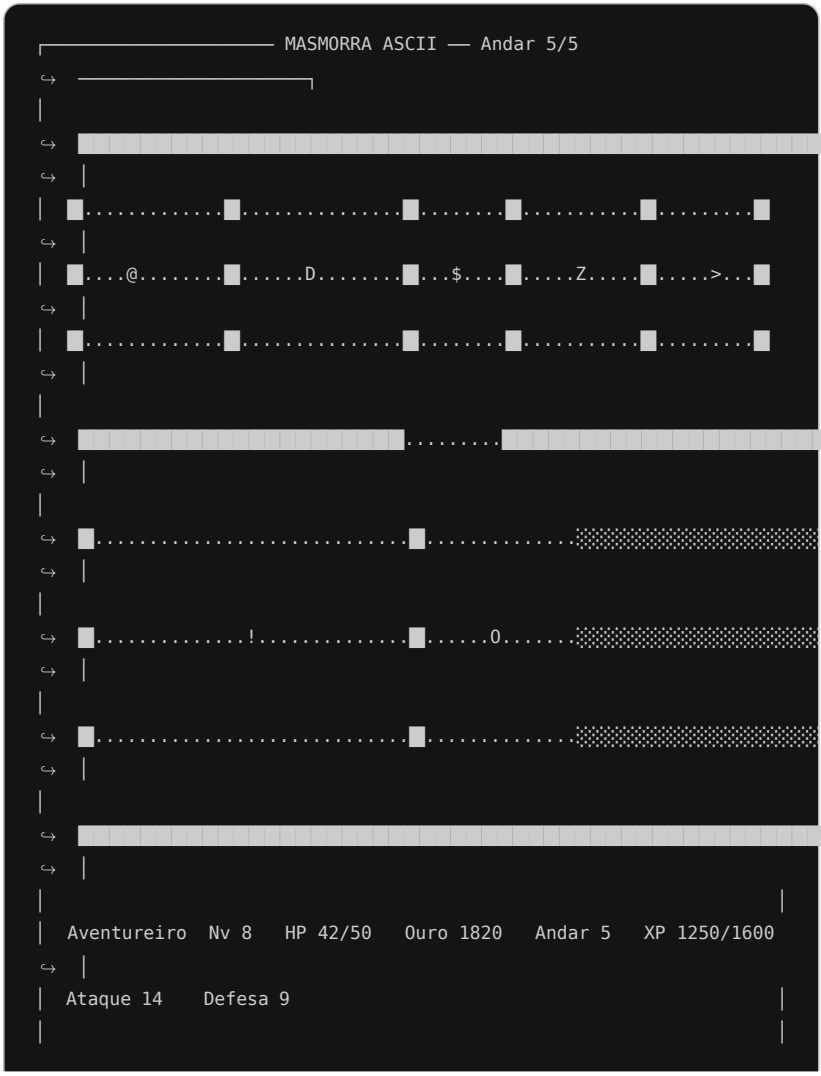
Livros Recomendados: - **"Design Patterns: Elements of Reusable Object-Oriented Software"** (Gang of Four) — clássico profundo, fundamental - **"Game Programming Patterns"** (Bob Nystrom, online gratuito) — padrões específicos de jogos, leitura obrigatória - **"Clean Code"** (Robert

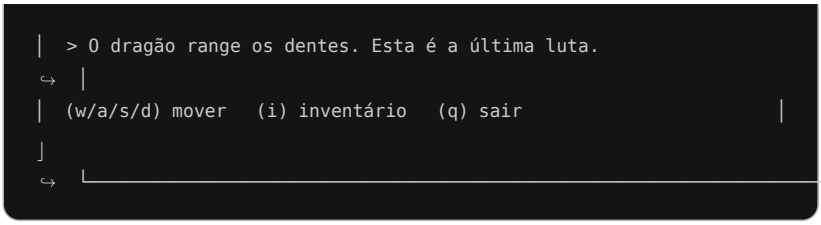
Martin) — engenharia de software profissional além de padrões - **“Refactoring”** (Martin Fowler) — como melhorar código existente sem quebrar funcionalidade

Próximos Domínios: - **Algoritmos**: Estruturas de dados avançadas (B-trees, heaps, grafos), busca A*, programação dinâmica - **IA**: Árvores de decisão, redes neurais, aprendizado por reforço (para IA de jogo melhor) - **Backend**: Express.js/Node (ou Shelf em Dart), bancos de dados, autenticação, APIs RESTful - **DevOps**: Docker, CI/CD (GitHub Actions), deploy em produção

Agora você tem ferramentas. Use-as bem. E lembre-se: todo especialista começou como você, com “print(‘Olá’)”. A diferença é persistência.

Epílogo


MASMORRA ASCII — Andar 5/5
Aventureiro Nv 8 HP 42/50 Ouro 1820 Andar 5 XP 1250/1600
Ataque 14 Defesa 9



A última tela antes do confronto final. O herói (`@`) diante do Dragão (`D`), do zumbi (`Z`), do orc (`O`), de ouro (`$`) e da escada (`>`). Um item (`!`) aguarda coleta. A névoa (`░`) ainda esconde parte do andar.

Você pressiona Enter pela última vez. O compilador chora, aceita, compila. A masmorra pisca na tela uma última vez e desaparece. Volta ao vazio. O terminal volta ao normal, aquele terminal que era apenas um retângulo de texto há trinta e sete capítulos, quando você começava sem saber nem o nome da primeira classe que iria escrever. Mas você saiu. Emergiu. A tela está preta outra vez, e você, sentado na cadeira, não é mais quem era quando a descida começou.

Trinta e sete capítulos. Milhares de linhas de código. Você não pode contar quantas vezes refatorou aquele laço que não fazia sentido. Quantas vezes viu o compilador reclamar daquele ponto e vírgula que não deveria estar ali. Quantas vezes você e a máquina tiveram essa conversa silenciosa, onde o erro apontava e você corrigia, corrigia, corrigia até que a lógica virou verdade e funcionou. Era frustrante. Era mágico. Era tudo de uma vez.

Mas o jogo funciona. A masmorra respira. Os inimigos patrulham com inteligência que você deu a eles: máquinas de estado que aprendeu construindo. O ouro cai quando você destrói uma múmia, não porque o código foi cópia de um exemplo, mas porque você entendeu a lógica de loot e a escreveu. O mapa muda a cada jogo, gerado por algoritmos que você escreveu e testou. O combate segue turnos que você definiu. O arquivo salvo sobrevive quando você fecha tudo e volta amanhã. Tudo está lá, palpitante, vivo, feito de decisões suas, de padrões que você escolheu, de abstrações que você construiu com as próprias mãos.

Você olha para a pasta do projeto. `src/`. `pubspec.yaml`. `lib/`. `test/`. Aquela pasta que começou vazia agora pulsa com vida que você criou. Você não apenas aprendeu Dart. Você aprendeu a criar. A ver além da sintaxe. A entender que um `class` não é só uma palavra-chave: é um conceito que ganha carne quando você a escreve com intenção. Que um `loop` não é só um mecanismo: é o pulso de um mundo que responde. Que um `Future` não é

abstração distante, mas o caminho que seus saves percorrem, de disco para memória, para tela, para você.

Os monstros foram burros no começo. Zumbis que andavam ao acaso. Mas você os fez pensar. Máquinas de estado deram intenção aos zumbis. Padrões de movimento deram propósito aos lobos. O dragão final veio com múltiplas fases porque você merecia um adversário à altura do caminho que percorreu. Você não criou apenas uma masmorra. Você criou seres. Seres de ASCII, feitos de caracteres simples, mas que respiram com a lógica que você soprou neles. Nada ali é acaso. Tudo é escolha sua.

E quando a próxima pessoa rodar o seu `dart run`, ou quando você mesmo rodar em uma semana, naquele dia que quer reviver a sensação da primeira descida, a masmorra estará lá. Intacta. Viva. Esperando. Porque você não apenas consumiu um tutorial e o descartou. Você desceu, explorou, aprendeu, e subiu. E quando subiu, trouxe consigo não um jogo, mas a compreensão fundamental de que é você quem cria os mundos. Um arquivo por vez. Uma função por vez. Um `main` que espera ser chamado.

A masmorra não termina. Ela continua a cada `dart run`. A cada bug que você conserta, porque você aprendeu a ler a pilha de erros como um mapa. A cada feature que você acrescenta porque você pensou nela, e não porque um capítulo mandou. Porque você é agora o que sempre foi desde o começo, quando pressionou aquele comando pela primeira vez: um criador. Um programador. Alguém que olha para o vazio do terminal e diz, com todos os dedos pousados no teclado, pronto para construir o que vier a seguir.

Há quem diga que roguelikes são sobre morte permanente. Sobre permadeath e reset e recomeçar do zero. Talvez. Mas o que você leva daqui não morre. O que você aprendeu (não só de Dart, mas de pensar estruturalmente, de desenhar abstrações, de ver um problema e decompô-lo em padrões) fica. E quando você entra em uma nova masmorra (um novo projeto, uma nova linguagem, um novo desafio), você não entra desarmado. Você entra como quem já desceu e subiu uma vez.

Boa sorte na próxima descida. Mas desta vez, você já sabe: descidas assim têm volta. E quando você volta, volta diferente. Volta construtor.

Apêndice A: Referência Rápida de Dart

Este apêndice reúne os principais conceitos de Dart usados ao longo do livro. Use como consulta rápida quando precisar lembrar de uma sintaxe ou padrão.

Tipos básicos

```
int vida = 100;
double dano = 7.5;
String nome = 'Guerreiro';
bool vivo = true;
```

Null safety

```
String? alvo;            // pode ser nulo
String nome = 'Herói';   // nunca nulo
alvo?.length;            // acesso seguro
alvo ?? 'ninguém';       // valor padrão se nulo
alvo ??= 'padrão';      // atribui se nulo
```

Coleções

```
// Lista
List<String> itens = ['espada', 'poção', 'escudo'];
itens.add('anel');
itens.removeAt(0);

// Map
Map<String, int> precos = {'espada': 50, 'poção': 20};
```

```
precos['escudo'] = 80;

// Set
Set<String> visitados = {'sala1', 'sala2'};
visitados.add('sala3');
```

Funções

```
// Função com retorno
int calcularDano(int forca, int nivel) {
  return forca * nivel;
}

// Arrow function
int dobro(int x) => x * 2;

// Função com parâmetros nomeados
void criar({required String nome, int vida = 100}) {
  print('$nome tem $vida HP');
}
```

Classes

```
class Jogador {
  final String nome;
  int _vida;

  Jogador(this.nome, this._vida);

  // Named constructor
  Jogador.iniciante(this.nome) : _vida = 100;

  // Getter
```

```
  int get vida => _vida;

  // Método
  void receberDano(int dano) {
    _vida = (_vida - dano).clamp(0, _vida);
  }

  @override
  String toString() => '$nome (HP: $_vida)';
}
```

Herança e classes abstratas

```
abstract class Inimigo {
  String get nome;
  int get vida;
  void agir(Jogador alvo);
}

class Zumbi extends Inimigo {
  @override
  String get nome => 'Zumbi';

  @override
  int get vida => 30;

  @override
  void agir(Jogador alvo) {
    // anda aleatoriamente
  }
}
```

Mixins

```
mixin Combatente {
  int atacar(int forca) => forca + Random().nextInt(6);
}

mixin Curavel {
  void curar(int quantidade) { /* ... */ }
}

class Guerreiro extends Jogador with Combatente, Curavel {
  Guerreiro(String nome) : super(nome, 100);
}
```

Enums (Dart 3)

```
enum Direcao {
  norte(0, -1),
  sul(0, 1),
  leste(1, 0),
  oeste(-1, 0);

  final int dx;
  final int dy;
  const Direcao(this.dx, this.dy);
}
```

Sealed classes e pattern matching

```
sealed class ComandoJogo {}

class CmdMover extends ComandoJogo {
  final Direcao dir;
```

```
  CmdMover(this.dir);
}

class CmdAtacar extends ComandoJogo {}

class CmdUsarItem extends ComandoJogo {
  final Item item;

  CmdUsarItem(this.item);
}

// Switch exaustivo
switch (comando) {
  case CmdMover(:final dir):
    jogador.mover(dir);
  case CmdAtacar():
    iniciarCombate();
  case CmdUsarItem(:final item):
    jogador.usar(item);
}
```

Async e Await: Programação Assíncrona

Sintaxe Básica

```
// Future<T> retorna um valor do tipo T no futuro
Future<String> carregarSave(String caminho) async {
  final arquivo = File(caminho);
  if (await arquivo.exists()) {
    return await arquivo.readAsString();
  }
  return '{}';
}
```

```
// Usar com await (bloqueia até completar)
void main() async {
  final dados = await carregarSave('save.json');
  print(dados);
}

// Usar sem await (não bloqueia)
void executarEmBackground() {
  carregarSave('save.json').then((dados) {
    print('Carregado: $dados');
  });
}
```

Tratamento de Erros

```
// Try/catch em async
Future<void> salvarProgresso(String arquivo, String dados) async {
  try {
    final file = File(arquivo);
    await file.writeAsString(dados);
    print('Salvo com sucesso!');
  } catch (e) {
    print('Erro ao salvar: $e');
  } finally {
    print('Operação finalizada');
  }
}

// Capturar erros sem try/catch
Future<String> carregarComFallback(String caminho) async {
  try {
    return await File(caminho).readAsString();
  } on FileSystemException {
    return 'dados padrão';
  }
```

```
}
```

Future: Computação Assíncrona

```
// Future completado imediatamente
Future<int> damoDB() => Future.value(42);

// Future que completa depois de delay
Future<int> damoComEspera() async {
  await Future.delayed(Duration(seconds: 2));
  return 42;
}

// Múltiplas futures em paralelo
Future<void> carregarTodos() async {
  final dados1 = carregarSave('save1.json');
  final dados2 = carregarSave('save2.json');

  // Aguardar todas
  final resultados = await Future.wait([dados1, dados2]);
  print('Todos carregados: $resultados');
}
```

Streams: Fluxos de Dados Assíncronos

```
// Generator async retorna Stream
Stream<int> contarAte5() async* {
  for (int i = 1; i <= 5; i++) {
    yield i;
    await Future.delayed(Duration(seconds: 1));
  }
}
```

```
// Consumir stream
void main() async {
  await for (final numero in contarAte5()) {
    print('Número: $numero');
  }
}

// Transformar stream
Stream<String> nomesEmMaiusculas(Stream<String> nomes) async* {
  await for (final nome in nomes) {
    yield nome.toUpperCase();
  }
}
```

Leitura e Escrita de Arquivos (dart:io)

```
import 'dart:io';
import 'dart:convert';

// Ler arquivo completo
Future<String> lerArquivo(String caminho) async {
  return await File(caminho).readAsString();
}

// Escrever arquivo
Future<void> escreverArquivo(String caminho, String conteudo) async {
  await File(caminho).writeAsString(conteudo);
}

// Ler linha por linha
Future<void> processarLinhas(String caminho) async {
  final arquivo = File(caminho);
  final linhas = arquivo.openRead()
    .transform(utf8.decoder)
    .transform(const LineSplitter());
```

```
  await for (final linha in linhas) {
    print('Linha: $linha');
  }
}

// Checar existência e criar diretório
Future<void> garantirDiretorio(String caminho) async {
  final dir = Directory(caminho);
  if (!await dir.exists()) {
    await dir.create(recursive: true);
  }
}
```

JSON

```
import 'dart:convert';

// Serializar
String json = jsonEncode({'nome': 'Herói', 'vida': 100});

// Deserializar
Map<String, dynamic> mapa = jsonDecode(json);
```

Testes

```
import 'package:test/test.dart';

void main() {
  group('Jogador', () {
    test('receber dano reduz vida', () {
      final jogador = Jogador('Teste', 100);
      jogador.receberDano(30);
```

```
      expect(jogador.vida, equals(70));
    });

    test('vida não fica negativa', () {
      final jogador = Jogador('Teste', 10);
      jogador.receberDano(50);
      expect(jogador.vida, equals(0));
    });
  });
}
```

Padrões de Projeto Usados no Livro

Strategy: cada inimigo tem uma estratégia de IA que decide seu comportamento. Permite trocar o comportamento em tempo de execução.

Command: ações do jogo (mover, atacar, usar item) são objetos com `executar()` e `desfazer()`. Permite histórico e desfazer.

Factory: criação centralizada de inimigos e itens por tipo e andar. Facilita balanceamento e extensão.

Observer: sistema de eventos com `Stream`. Quando algo acontece no jogo, vários sistemas são notificados (log, UI, estatísticas).

State: máquinas de estado para comportamento de inimigos (patrulha, alerta, perseguição, ataque, fuga) e fases de boss.

Para Explorar Depois

O calabouço vai mais fundo. Aqui estão os skills para a próxima aventura: recursos avançados que transformam seus programas Dart de "funcional" para "obra-prima".

Streams: Fluxos de Dados Assíncronos

Streams são tubos por onde dados fluem continuamente. Perfeito para eventos em tempo real, atualizações de sensores, ou qualquer coisa que acontece ao longo do tempo. Use `Stream`, `StreamController`, ou `async*` com `yield` para criar seus próprios.

```
Stream<int> contadorStream() async* {
  for (int i = 0; i < 5; i++) {
    yield i;
    await Future.delayed(Duration(seconds: 1));
  }
}
```

Isolates: Concorrência de Verdade

Isolates são "threads" do Dart: mundos paralelos que executam código pesado sem travar a UI. Diferente de threads convencionais, eles não compartilham memória, o que evita deadlocks e race conditions. Use para cálculos pesados ou processamento de dados.

```
void computarFatorialPesado() async {
  final resultado = await compute(fatorial, 1000000);
}

int fatorial(int n) => n <= 1 ? 1 : n * fatorial(n - 1);
```

typedef: Apelidos para Tipos

Typedefs criam aliases para funções e tipos complexos, melhorando legibilidade do código. Essencial para callbacks complicados.

```
typedef Comparador = int Function(dynamic a, dynamic b);
typedef DadosJogo = ({String nome, int vida});

Comparador minhaCmp =
    (a, b) => a.toString().compareTo(b.toString());
```

Extensions: Superpoderes para Tipos Existentes

Extensions adicionam métodos a classes já existentes—String, List, num—sem herança ou modificação. Transforme a forma como você trabalha com tipos padrão.

```
extension on String {
  String gritarEmMaisculas() => toUpperCase();
}

print('herói'.gritarEmMaisculas()); // HERÓI
```

Mixins Avançados: O Poder da Composição

Mixins com restrições (`on`) garantem que suas misturas só funcionam em classes específicas. Prefira mixins a herança múltipla—são mais seguros e flexíveis.

```
mixin Curioso on Jogador {
  void investigar() => print('$nome investigou a sala');
}

class Paladino extends Jogador with Curioso {
  Paladino(String nome) : super(nome, 150);
}
```

Recursos Úteis

- Documentação oficial: `dart.dev`
- Pacotes: `pub.dev`
- Guia de estilo: `dart.dev/effective-dart`
- Flutter: `flutter.dev`

Apêndice B: MUD em Rede (Opcional)

Este apêndice não faz parte do percurso obrigatório do livro. O núcleo permanece um jogo offline no terminal. Mas se você quiser dar o próximo passo e transformar a Masmorra ASCII em um MUD (Multi-User Dungeon) multiplayer que funciona em rede, aqui estão as direções (e não são tão longas quanto pode parecer).

A Ideia Fundamental

Um MUD é, essencialmente, o mesmo jogo que você já construiu, mas com vários jogadores conectados ao mesmo tempo via rede. O modelo de domínio (salas, jogador, combate, itens) permanece intacto. O que muda é a camada de transporte e estado:

- **Localmente:** você lê do `stdin` e escreve em `stdout`. O jogo é uma sequência de iterações no seu terminal.
- **Em rede:** o servidor recebe comandos via WebSocket de múltiplos clientes. Mantém um mundo compartilhado na memória. Envia eventos para todos os jogadores afetados por uma ação.

Não é magia. É arquitetura: separar o modelo de domínio (que não muda) da apresentação (que agora é remota) e da comunicação (que agora é de muitos para um servidor).

Arquitetura em Camadas

Pensar em rede significa pensar em separação:

Camadas da arquitetura em rede. A fonte editável do diagrama está em `assets/diagrams/apendice-b-camadas-rede.mmd`; o PNG é gerado em `./scripts/build.sh` com Node.js/npx (`@mermaid-js/mermaid-cli`).

A beleza disso: quase nada do que você escreveu no livro muda. Você reutiliza suas classes `Jogador`, `Sala`, `Inimigo`, etc. O que muda é o **loop principal**: deixa de ser single-player local e vira um servidor que coordena múltiplos clientes.

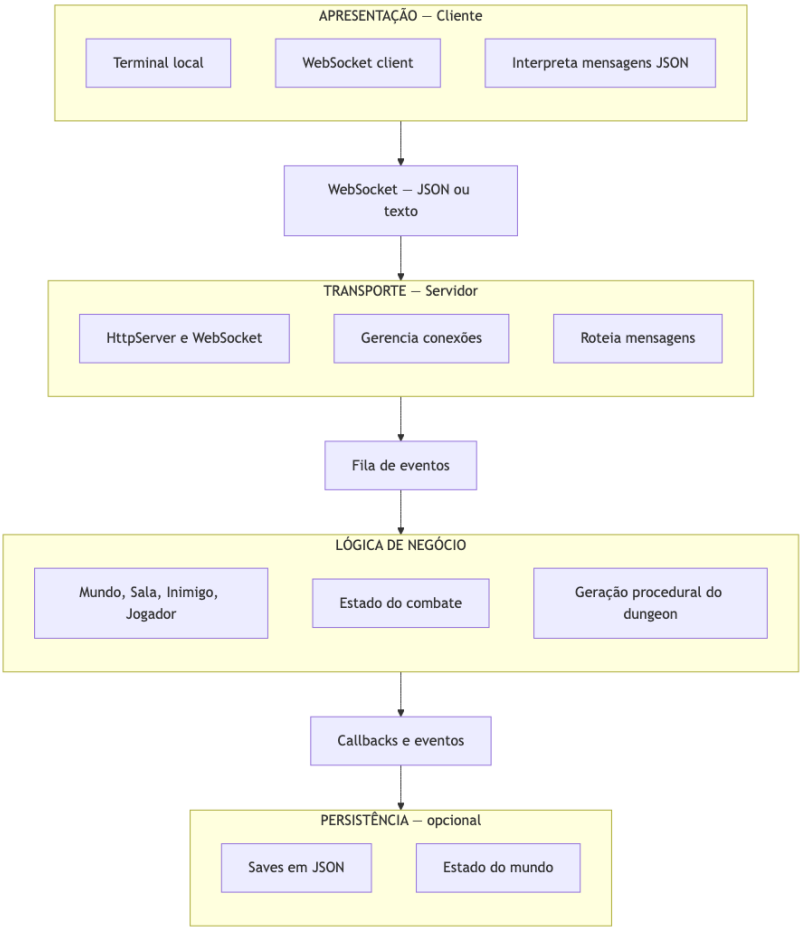


Figura 4: Arquitetura em camadas: cliente, transporte, lógica e persistência

Implementação: Servidor WebSocket Básico

O pacote `dart:io` já traz tudo que você precisa. Não é necessário adicionar dependências externas (embora `package:shelf` seja excelente para APIs mais complexas).

```
import 'dart:io';
import 'dart:convert';

// Mapa global: cada jogador ativo está aqui
final jogadoresAtivos = <String, JogadorConectado>{};

// O mundo compartilhado — sua masmorra vive aqui
late Mundo mundoCompartilhado;

void main() async {
  // Inicializa o mundo (mesmo código do livro)
  mundoCompartilhado = Mundo(semente: 12345);

  // Cria o servidor HTTP
  final servidor = await HttpServer.bind('localhost', 8080);
  print('Servidor MUD rodando em localhost:8080');

  // Loop infinito aguardando conexões
  await for (final requisicao in servidor) {
    if (WebSocketTransformer.isUpgradeRequest(requisicao)) {
      // Uma nova pessoa entrou na masmorra
      final ws = await WebSocketTransformer.upgrade(requisicao);
      tratarNovaConexao(ws);
    }
  }
}

// Representa um jogador conectado
class JogadorConectado {
  final String id;
  final Jogador jogador; // Sua classe do livro
```

```
  final WebSocket ws;
  bool ativo = true;

  JogadorConectado(this.id, this.jogador, this.ws);

  void enviar(String mensagem) {
    if (ativo && ws.closeCode == null) {
      ws.add(mensagem);
    }
  }

  void fechar() {
    ativo = false;
    ws.close();
  }
}

void tratarNovaConexao(WebSocket ws) {
  // Cria um jogador novo
  final idJogador = _gerarID();
  final novoJogador = Jogador(
    nome: 'Aventureiro#$idJogador',
    vida: 100,
  );

  final conectado = JogadorConectado(idJogador, novoJogador, ws);
  jogadoresAtivos[idJogador] = conectado;

  // Envio inicial
  conectado.enviar('Bem-vindo à Masmorra ASCII Online!');
  conectado.enviar('Seu nome é: ${novoJogador.nome}');
  _descreverSala(conectado);

  // Escuta mensagens do cliente
  ws.listen(
    (mensagem) => _processarComando(idJogador, mensagem.toString()),
```

```
    onDone: () => _desconectar(idJogador),
    onError: (erro) {
      print('Erro no WebSocket: $erro');
      _desconectar(idJogador);
    },
  );
}

void _processarComando(String idJogador, String comando) {
  final conectado = jogadoresAtivos[idJogador];
  if (conectado == null) return;

  // Parse: "mover norte", "atacar", "usar poção", etc.
  final partes = comando.trim().toLowerCase().split(' ');
  if (partes.isEmpty) return;

  final acao = partes[0];

  switch (acao) {
    case 'mover':
      if (partes.length < 2) {
        conectado.enviar('Sintaxe: mover [norte|sul|leste|oeste]');
        return;
      }
      _moverJogador(idJogador, partes[1]);
      break;

    case 'atacar':
      _iniciarCombate(idJogador);
      break;

    case 'inventario':
      final inv = conectado.jogador.inventario;
      conectado.enviar('Seu inventário: $inv');
      break;
```

```
    case 'sair':
      conectado.enviar('Você saiu da masmorra. Até logo!');
      _desconectar(idJogador);
      break;

    default:
      conectado.enviar('Comando desconhecido: $acao');
  }
}

void _moverJogador(String idJogador, String direcao) {
  final conectado = jogadoresAtivos[idJogador];
  if (conectado == null) return;

  // Aqui você chama a lógica de movimento do seu Jogador
  // Por exemplo: conectado.jogador.mover(direcao);
  // E verifica colisões, inimigos, etc.

  conectado.enviar('Você se moveu para $direcao');
  _descreverSala(conectado);

  // Notifica outros jogadores na mesma sala
  _notificarSala(conectado, '${conectado.jogador.nome} entrou');
}

void _descreverSala(JogadorConectado conectado) {
  // Sua classe Sala tem descrição?
  // conectado.enviar(sala.descreverPara(jogador));
  // Isso renderiza a sala, lista inimigos visíveis, saídas, etc.
}

void _notificarSala(JogadorConectado origem, String mensagem) {
  // Encontra todos os jogadores na mesma sala
  for (final outro in jogadoresAtivos.values) {
    if (outro.ativo && outro.id != origem.id) {
      // Verifica se estão na mesma sala
```

```
      if (_mesmasSalas(origem.jogador, outro.jogador)) {
        outro.enviar(mensagem);
      }
    }
  }
}

void _desconectar(String idJogador) {
  final conectado = jogadoresAtivos.remove(idJogador);
  if (conectado != null) {
    conectado.fechar();
    print('${conectado.jogador.nome} desconectou');
    // Notifica outros jogadores
    final n = conectado.jogador.nome;
    _notificarSala(conectado, '$n saiu da masmorra');
  }
}

String _gerarID() =>
↪  DateTime.now().millisecondsSinceEpoch.toString();
bool _mesmasSalas(Jogador a, Jogador b) {
  // Implementar conforme sua estrutura de Sala
  return a.salaAtual == b.salaAtual;
}
```

Padrões de Mensagem: Contrato Client-Servidor

Defina um formato claro para trocas entre cliente e servidor. JSON é limpo e extensível:

```
// Cliente ENVIA
{
  "tipo": "comando",
  "acao": "mover",
  "parametro": "norte"
```

```
}

// Servidor RESPONDE
{
  "tipo": "descricao",
  "sala": "Corredor Escuro e Úmido",
  "saidas": ["norte", "sul"],
  "inimigos": ["Zumbi", "Múmia"],
  "jogadores": ["Aventureiro#123", "Aventureiro#456"]
}

// Evento de broadcast (servidor notifica TODOS afetados)
{
  "tipo": "evento",
  "acao": "morte",
  "jogador": "Aventureiro#123",
  "mensagem": "Aventureiro#123 foi derrotado!"
}
```

Parse robusto em Dart:

```
void _processarMensagem(String json) {
  final dados = jsonDecode(json) as Map<String, dynamic>;
  final tipo = dados['tipo'] as String;

  switch (tipo) {
    case 'comando':
      _executarComando(dados);
    case 'chat':
      _broadcast(dados);
    // etc.
  }
}
```

Gerenciamento de Estado Compartilhado

A chave: **uma única instância do mundo, protegida por sincronização**.

Se dois jogadores atacarem o mesmo inimigo simultaneamente, você precisa garantir que: 1. Ambas as ações sejam registradas 2. O HP do inimigo não seja decrementado duas vezes por engano 3. Se o inimigo morre, ambos veem a morte

Use `Mutex` ou `Lock` do Dart para seções críticas:

```
import 'dart:async';

class MundoCritico {
  final mundo = Mundo();
  final _lock = Lock(); // De package:async

  Future<void> executarAcao(String idJogador, String acao) async {
    await _lock.synchronized(() {
      // Apenas uma ação por vez nesta seção
      final jogador = mundo.obterJogador(idJogador);
      if (acao == 'atacar') {
        final inimigo = mundo.obterInimigoProximo(jogador);
        if (inimigo != null) {
          jogador.atacar(inimigo);
          if (inimigo.vida <= 0) {
            mundo.removerInimigo(inimigo);
          }
        }
      }
    });
  }
}
```

Broadcast: Notificando Múltiplos Clientes

Quando algo acontece (inimigo morre, tesouro aparece, outro jogador chega), **todos na sala precisam saber**.

```
void _broadcast(String mensagem) {
  for (final conectado in jogadoresAtivos.values) {
    if (conectado.ativo) {
      conectado.enviar(mensagem);
    }
  }
}

void _broadcastParaSala(String nomeSala, String mensagem) {
  for (final conectado in jogadoresAtivos.values) {
    if (conectado.ativo && conectado.jogador.salaAtual == nomeSala) {
      conectado.enviar(mensagem);
    }
  }
}
```

Sessões e Persistência

Cada jogador merece uma sessão:

```
class Sessao {
  final String id;
  late Jogador jogador;
  late DateTime conectadoEm;
  int ultimaAtividadeEm = DateTime.now().millisecondsSinceEpoch;

  Sessao(this.id);

  bool estaInativa(int tempoMaximoEmMs) {
    return DateTime.now().millisecondsSinceEpoch - ultimaAtividadeEm >
        tempoMaximoEmMs;
  }
}
```

E salvar periodicamente:

```
void _salvarProgresso(Sessao sessao) {
  final json = jsonEncode({
    'nome': sessao.jogador.nome,
    'vida': sessao.jogador.vida,
    'ouro': sessao.jogador.ouro,
    'sala': sessao.jogador.salaAtual,
    'inventario': sessao.jogador.inventario.map((i) =>
    ↪  i.nome).toList(),
  });

  // Salva em arquivo ou banco de dados
  File('saves/${sessao.id}.json').writeAsStringSync(json);
}
```

Escalabilidade: Próximos Passos

O servidor acima funciona para **dezenas de jogadores**. Se você quiser milhares:

1. **Distribuir o estado:** Não guarde tudo na memória de um servidor. Use Redis para cache, PostgreSQL para persistência.
2. **Sharding:** Cada servidor gerencia um "andar" diferente da masmorra.
3. **Message queues:** Use RabbitMQ ou Kafka para fila de mensagens assíncronas entre servidores.
4. **Logging e monitoramento:** Adicione observabilidade com Sentry ou Datadog.

Mas para aprender, o acima é suficiente. Uma masmorra funcionando em rede é um projeto denso, e você já tem toda a lógica do livro. Agora é apenas orquestração.

Recursos Complementares

- `package:shelf_web_socket` para uma camada de transporte mais robusta
- `package:async` (incluída com Dart) para `Lock`, `Mutex` e outros primitivos de concorrência
- `package:shelf` se quiser adicionar endpoints REST (status do servidor, rankings, etc.)

- Dart docs: `dart.dev/guides/libraries/library-tour#dartio` para WebSocket detalhes

O calabouço não termina aqui. Ele só fica mais profundo. E agora, mais multiplayer.

Apêndice C: Achievements do Aventureiro

Que tal tornar sua jornada ainda mais épica? Cada seção do livro traz uma série de conquistas que você pode desbloquear conforme avança. Marque o checkbox ao lado assim que completar o desafio. Alguns achievements são bem diretos: código escrito, teste rodando, feature funcionando. Outros exigem um toque extra de criatividade e pensamento além do esperado. A diversão está em conseguir todos antes de cravar a melhor pontuação possível na masmorra final.

Prepare-se: há alguns achievements secretos escondidos pela floresta. Se você os encontrar, você merece um prêmio especial (spoiler: é a satisfação intelectual de saber que subiu mais fundo do que a maioria).

Primeiros Passos

Capítulos 1–7: A jornada começa aqui. Você está aprendendo os fundamentos de Dart e ganhando confiança com cada linha de código. Critério: código compila sem erros.

[ ] **Hello, World!** – Compilou e rodou seu primeiro programa Dart no terminal. Critério: `print('Hello, World!')` aparece na tela.

[ ] **Que Comece o Jogo!** – Criou a estrutura básica do jogo com input e output funcionando. Critério: pode digitar um comando e receber resposta.

[ ] **Guerreiro Iniciante** – Definiu a classe `Jogador` com propriedades básicas (nome, vida, ataque). Critério: classe compila e pode criar uma instância com `Jogador('teste', 100, 10)`.

[ ] **Conhecedor de Tipos** – Usou corretamente pelo menos 5 tipos primitivos diferentes de Dart (int, double, String, bool, List). Critério: todas as variáveis typadas corretamente, sem erros de análise estática.

[ ] **Organizador de Código** – Criou sua primeira classe e instanciou um objeto com sucesso. Critério: classe instantiada e método chamado nela sem erro.

[ ] **Mestre das Variáveis** – Declarou corretamente variáveis nulas e não-nulas usando null safety. Critério: sem avisos de análise estática sobre null safety; código permite `?` onde apropriado.

[ ] **Desafio Extra: Personagem Único** – Criou uma classe `Jogador` com 3 atributos customizados além dos básicos (nome, vida, ataque). Critério: novos atributos aparecem em `toString()` ou getter.

[ ] **Konami Code** — Implementou uma sequência secreta de inputs no jogo (↑ ↑ ↓ ↓ ←→←→BA). Não dá vida extra, mas dá respeito.

Orientação a Objetos

Capítulos 8–14: O código ganha profundidade. Herança, polimorfismo e abstrações tornam seu jogo muito mais elegante. Critério: padrão OOP implementado e testado.

[ ] **Construtor Primordial** – Implementou um construtor nomeado em uma classe. Critério: `Jogador.iniciante(String nome)` existe e funciona sem `new`.

[ ] **Getter Dos Deuses** – Usou seu primeiro getter para encapsular uma propriedade privada. Critério: propriedade prefixada com _, acesso via getter público.

[ ] **Herança Domada** – Criou uma classe que estende outra (ex: `Guerreiro extends Jogador`). Critério: subclasse compila, herda propriedades, pode fazer `super()`.

[ ] **Polimorfismo em Ação** – Override de um método da classe pai na classe filha com `@override`. Critério: subclasse sobrescreve método; comportamento diferencia conforme tipo.

[ ] **Classes Abstratas Decifradas** – Criou uma classe abstrata e implementou suas funções obrigatórias. Critério: `abstract class Inimigo` existe; subclasses implementam todos os métodos abstratos.

[ ] **Inimigo Forjado** – Criou pelo menos 3 tipos de inimigos diferentes com comportamentos distintos. Critério: `Zumbi`, `Esqueleto`, `Lobo` existem e cada um age diferente em combate.

[ ] **Mixin do Poder** – Usou um mixin para adicionar funcionalidade a uma classe. Critério: `mixin Combatente { ... }` ou similar existe; classe usa `with Combatente`.

[ ] **Desafio Extra: Hierarquia Completa** – Criou uma árvore de herança com 4+ classes relacionadas. Critério: `Inimigo` → `ZumbiComum`, `Esqueleto`, `Boss`, `Dragão` (ou similar).

A Masmorra Ganha Vida

Capítulos 15–21: A masmorra deixa de ser teoria e vira um mundo real. Salas, itens, monstros e sistema de combate entram em cena.

[ ] **Cartógrafo da Masmorra** – Criou um sistema de salas conectadas. Critério: mapa com 10+ salas navináveis via comando `mover norte/sul/leste/oeste`.

[ ] **Gerador de Mundos** – Implementou a geração procedural de masmorras. Critério: novo `dart run` gera mapa diferente; usando algoritmo ou `Random`.

[ ] **Primeira Espada** – Criou o sistema de itens com uma arma funcionando. Critério: classe `Arma` ou `Item` existe; pode ser equipada e usada em combate.

[ ] **Poção Curativa** – Implementou um item que cura dano. Critério: poção aumenta HP do jogador quando usada; `receberCura(50)` funciona.

[ ] **Combate Realizado** – Fez seu primeiro combate entre jogador e inimigo. Critério: combate por turnos ocorre; jogador e inimigo trocam ataques.

[ ] **Vitória Épica** – Derrotou um inimigo no combate (vida chegou a 0). Critério: inimigo morre; seu HP exibe `0` ou sai do jogo.

[ ] **Sobrevivente** – Tomou dano, se recuperou com uma poção e continuou vivo. Critério: `jogador.vida < max`, usou poção, `jogador.vida > 0` após.

[ ] **Desafio Extra: Boss de Andar** – Criou um inimigo especial mais forte que aparece a cada 5 andares. Critério: boss tem 3x+ HP e ataque; aparece em andar 5, 10, 15, etc.

Economia e Combate

Capítulos 22–27: Moedas de ouro, loot, balanceamento e tática entram na equação. Seu jogo fica viciante.

[ ] **Primeira Moeda de Ouro** – Ganhou sua primeira moeda derrotando um inimigo. Critério: `jogador.ouro` aumenta após matar inimigo.

[ ] **Caçador de Tesouro** – Coletou ao menos 100 moedas de ouro em uma sessão. Critério: `jogador.ouro >= 100` antes de salvar/sair.

[ ] **Loot Épico** – Um inimigo dropou um item raro (chance < 25%). Critério: inimigo morre, item raro aparece (ex: anel mágico, poção rara).

[ ] **Comerciante Experiente** – Comprou um item em uma loja ou vendeu algo. Critério: transação completa; ouro transferido, inventário modificado.

[ ] **Armadura Forjada** – Equipou uma peça de armadura que reduz dano recebido. Critério: defesa aumenta; dano sofrido é menor após equipar.

[ ] **Estratégia Vencedora** – Usou tática no combate escolhendo ataque vs defesa. Critério: combate tem opções (não apenas ataque automático); suas escolhas afetam o resultado.

[ ] **Derrotado e Revivido** – Morreu, carregou save anterior e venceu a masmorra. Critério: game over ocorreu; recarregou save; continuou e venceu.

[ ] **Desafio Extra: Fortuna de um Rei** – Reuniu 1000+ moedas de ouro sem gastar nada. Critério: `jogador.ouro >= 1000` em uma partida contínua.

Código Profissional

Capítulos 28–32: Testes, tratamento de erro, I/O de arquivos e code smell desaparecem. Seu jogo é mantível.

[ ] **Testador Devotado** – Escreveu seu primeiro teste unitário usando `package:test`. Critério: arquivo `.test.dart` ou em `test/`, teste passa.

[ ] **Cobertura Completa** – Escreveu testes para 3+ métodos diferentes. Critério: 3+ funções testadas; cada teste valida comportamento específico.

[ ] **Exceção Tratada** – Implementou `try/catch` para lidar com um erro de forma elegante. Critério: exceção capturada, mensagem amigável exibida, jogo continua.

[ ] **Save and Load** – Salvou um progresso em arquivo e carregou de volta com sucesso. Critério: `salvar()` cria arquivo; `carregar()` reconstrói estado idêntico.

[ ] **JSON Dominado** – Serializou uma classe complexa para JSON e desserializou. Critério: `Jogador.toJson()` e `Jogador.fromJson()` funcionam corretamente.

[ ] **Debug Eficiente** – Usou `print` ou debugger para encontrar e corrigir um bug. Critério: bug identificado via logs; correção aplicada; teste passa.

[ ] **I Am Error** — Encontrou e corrigiu um bug em código alheio antes de ser avisado. Zelda II manda lembranças.

[ ] **Desafio Extra: Teste de Integração** – Escreveu um teste que valida combate completo. Critério: teste simula combate início→fim; verifica HP, morte, loot.

Mestre dos Padrões

Capítulos 33–36: Strategy, Factory, Observer, Command, State. Você agora pensa em padrões. Seu código respira elegância.

[ ] **Factory Fundamental** – Implementou uma Factory para criar inimigos. Critério: `InimigoFactory.criar(tipo, andar)` existe; retorna tipo correto conforme parâmetros.

[ ] **Strategy Sutil** – Diferentes inimigos usam estratégias de IA diferentes. Critério: `enum Estrategia` ou `abstract class Estrategia`; `Zumbi`, `Lobo` comportam-se diferente.

[ ] **Observer Atento** – Usou `Stream` para notificar múltiplos listeners de um evento. Critério: `StreamController` emite; múltiplos `.listen()` recebem.

[ ] **Command Armazenado** – Implementou Command pattern com `execute()` e `desfazer()`. Critério: comando executável e revertível; histórico mantido.

[ ] **Estado Máquina** – Usou State pattern para estado de combate ou comportamento de inimigo. Critério: `enum EstadoCombate` ou sealed class; estados mudam corretamente.

[ ] **Sealed Class Power** – Usou `sealed class` e pattern matching. Critério: `sealed class Comando { ... }` com subclasses; `switch` exaustivo funciona.

[ ] **Desafio Extra: Todos os Padrões** – Integrou todos os 6 padrões em um jogo funcionando. Critério: jogo roda; cada padrão aparece no código; nenhum é dummy.

Achievements Secretos

Desafios especiais que requerem um pouco de criatividade e pensamento lateral. Se você encontrar um, você merece bragging rights; uma história para contar.

[ ] **A Verdade está Aqui** – Encontrou uma mensagem de easter egg escondida no código. Critério: mensagem de humor ou referência dentro de classe, comentário ou saída.

[ ] **Programador Filosófico** – Refletiu sobre sua própria arquitetura e refatorou por vontade própria. Critério: refator não pedido no livro; melhora legibilidade/performance.

[ ] **Mentorado** – Ajudou alguém a entender um conceito do livro ou resolveu uma dúvida deles. Critério: conversou com alguém; explicou padrão ou conceito; pessoa aprendeu.

[ ] **Open Source Spirit** – Compartilhou seu código em repositório público (GitHub, GitLab). Critério: repositório acessível, código visível, pelo menos 1 commit.

[ ] **Extensão Criativa** – Adicionou uma feature completamente original não mencionada no livro. Critério: feature funciona; não é apenas copy-paste; você a desenhou.

[ ] **Performance Optimizado** – Identificou um gargalo de performance e otimizou. Critério: benchmark antes/depois; melhora mensurável (geração mais rápida, render mais suave).

[ ] **Documentação Impecável** – Documentou suas classes com docstrings explicativas. Critério: `///` comments em todas as classes públicas e métodos públicos.

[ ] **Hackathon Pessoal** – Completou o livro inteiro em menos de 30 dias. Critério: timestamp de criação para capitulo final antes de 30 dias do primeiro.

[ ] **Desafio Ultra: MUD Multiplayer** – Seguiu o Apêndice B e implementou versão em rede. Critério: servidor WebSocket roda; 2+ clientes conectam simultaneamente; sincronização funciona.

[ ] **Void Spirit** – Entendeu completamente `null safety` sem pensar mais uma vez. Critério: Seu código não tem avisos de null safety; você usa `?`, `??`, `late` corretamente por reflexo.

[ ] **Do a Barrel Roll** — Implementou rotação ou transformação de mapa. Peppy Hare aprovaria.

[ ] **The Cake Is a Lie** — Descobriu que um save corrompido não é o que parece. GLaDOS ficaria orgulhosa.

[ ] **It's Dangerous to Go Alone** — Completou o MUD multiplayer do Apêndice B. Agora leve um amigo.

[ ] **There Is No Spoon** — Dominou null safety a ponto de nunca precisar do operador `!`. Neo aprovaria.

Parabéns! Se você chegou aqui e marcou tudo (ou mesmo a maioria), você não apenas leu um livro. Você aprendeu Dart de verdade, engenharia de software em contexto, e criou um jogo que respira inteligência. Você não é apenas um programador agora. Você é um verdadeiro construtor de mundos. E a próxima masmorra que você explorar (seja em código ou em vida) você entra diferente.

Apêndice D: Glossário

Este glossário reúne os termos técnicos mais importantes usados ao longo do livro e na jornada roguelike. Consulte-o sempre que encontrar uma palavra desconhecida: aqui você encontrará tanto a explicação técnica quanto o contexto do jogo.

Abstract Class: classe que não pode ser instanciada diretamente e serve como base para outras classes. Define uma interface que subclasses devem implementar, força um contrato de comportamento.

Ahead-of-Time (AOT): compilação que converte código Dart em código de máquina nativo antes da execução. Resulta em inicialização rápida e performance previsível, ideal para aplicações que precisam iniciar rapidamente.

ANSI Escape Codes: sequências de caracteres que controlam formatação, cor e posição do cursor no terminal. O jogo usa códigos como `\x1B[2J` para limpar a tela e `\x1B[31m` para texto vermelho.

ASCII: American Standard Code for Information Interchange. Padrão de codificação que representa caracteres usando números de 0 a 127. Essencial para jogos baseados em texto que utilizam caracteres para desenhar o mundo.

Async/Await: padrão em Dart para operações assíncronas não-bloqueantes. `async` marca uma função como assíncrona, `await` pausa a execução até o `Future` resolver. Essencial para I/O (save/load) sem congelar o game loop.

BFS (Breadth-First Search): algoritmo de busca em largura que explora todos os nós de uma distância antes de explorar nós mais afastados. Usado em pathfinding para encontrar o caminho mais curto entre dois pontos.

Boss: inimigo poderoso, geralmente único ou raro, que representa um desafio significativo. Frequentemente marca o final de uma seção ou dungeon e oferece recompensas maiores ao ser derrotado.

Buffer: área de memória que armazena dados temporariamente. Em renderização, o buffer guarda o estado de cada tile antes de exibir na tela.

Command Pattern: padrão de design que encapsula uma solicitação como um objeto, permitindo parametrizar clientes com diferentes requisições e implementar filas, desfazer e logging.

Distância de Manhattan: método de calcular distância entre dois pontos em grid contando passos horizontais e verticais: `|x1 - x2| + |y1 - y2|`. Mais rápido que distância euclidiana em jogos baseados em grid. Usada em cálculos de FOV, spawn seguro e IA.

Death Spiral: situação no jogo em que uma derrota parcial (HP baixo, inventário esvaziado, recursos acabando) torna mais provável a próxima derrota, que por sua vez agrava ainda mais a situação. Roguelikes bem balanceados oferecem saídas do death spiral; roguelikes cruéis não.

Dungeon Crawler: tipo de jogo onde o jogador explora um calabouço ou série de andares subterrâneos, enfrentando monstros e coletando tesouro. Exemplos clássicos incluem Rogue e NetHack.

DRY (Don't Repeat Yourself): princípio que evita duplicação de código. Código duplicado é mais difícil de manter e mais propenso a bugs.

Effective Dart: conjunto de diretrizes e melhores práticas para escrever código Dart de alta qualidade, mantido pela equipe oficial do Dart.

Enum: tipo que define um conjunto fixo de constantes nomeadas. Útil para representar estados, direções ou tipos de entidades no jogo.

Factory Constructor: construtor especial em Dart que não necessariamente cria uma nova instância da classe, podendo retornar uma instância existente ou de uma subclasse.

Field of View (FOV): área que o jogador pode enxergar do seu ponto de vista atual. Implementada com algoritmos como shadowcasting para criar exploração realista. Diferente de Line of Sight (LOS), que verifica visibilidade entre dois pontos específicos.

Fog of War (Névoa de Guerra): técnica que oculta áreas ainda não exploradas pelo jogador ou que saíram do seu campo de visão. No nosso jogo, tiles descobertos ficam "lembrados" em cinza, enquanto áreas nunca visitadas permanecem invisíveis.

Frame Rate: número de quadros (frames) renderizados por segundo. Em jogos baseados em turnos como roguelikes, o conceito se aplica apenas à renderização da interface; o loop de jogo em si é pausado à espera do input do jogador.

Future: tipo Dart que representa um valor que será disponível no futuro. Retornado por operações assíncronas como leitura de arquivos. Aguardado com `await` ou gerenciado com `.then()`.

Game Loop: estrutura fundamental de um jogo que continuamente executa: processamento de entrada, atualização de estado e renderização. Garante comportamento consistente e responsivo.

Golden Test: teste que compara a saída de uma função com um resultado pré-gravado. Útil para testes de renderização ou geração procedural onde a saída é complexa.

HUD (Heads-Up Display): interface visual que exibe informações do jogo como vida, mana, inventário e mapa sem interromper o gameplay. Geralmente posicionado nas bordas da tela.

JIT (Just-In-Time): compilação que converte código durante a execução. Mais lento na inicialização mas pode otimizar código que está sendo executado frequentemente.

JSON: JavaScript Object Notation. Formato padrão para representar dados estruturados em texto. Usado para serializar estado do jogo em arquivos de save.

Late: palavra-chave Dart que permite declarar uma variável que será inicializada após a construção do objeto. Útil para valores que dependem de outros parâmetros mas são garantidos antes do primeiro uso.

Linha de Visão (Line of Sight / LOS): algoritmo que determina se um ponto é visível a partir de outro sem obstáculos bloqueando. Diferente de FOV, que calcula múltiplos pontos, LOS verifica uma linha reta entre dois objetos.

Loot: itens valiosos encontrados após derrotar inimigos ou explorar áreas. Inclui equipamentos, poções, ouro e outros objetos que melhoram o personagem.

Loot Table: tabela de probabilidades que define o que cada inimigo ou baú pode dropar, com pesos por raridade. Permite balancear economia do jogo sem recodificar: basta ajustar números.

Level Up: processo pelo qual o personagem do jogador ganha experiência e aumenta de nível, geralmente resultando em aumento de atributos e novas habilidades.

Mixin: mecanismo em Dart que permite reutilizar código de uma classe em múltiplas classes sem usar herança. Implementado com a palavra-chave `with`.

Mob: abreviação de mobile, refere-se a inimigos comuns e repetidos. Diferente de boss, que são únicos ou raros.

MUD (Multi-User Dungeon): jogo de texto multiplayer baseado em exploração de calabouço. Precursor dos roguelikes modernos, ainda popular na comunidade de jogadores.

MST (Minimum Spanning Tree / Árvore Geradora Mínima): estrutura de grafo que conecta todos os nós com peso total mínimo, sem

ciclos. Usada em geração procedural para conectar salas ou áreas sem sobreposição desnecessária.

Null Safety: recurso do Dart que torna seguro trabalhar com valores nulos. O compilador garante que variáveis não nulas nunca recebam `null`, prevenindo uma classe inteira de bugs.

NPC (Non-Player Character): personagem controlado pelo jogo, não pelo jogador. Pode ser comerciante, quest-giver, aliado ou simples decoração.

Observer Pattern: padrão de design que estabelece uma relação um-para-muitos entre objetos, onde mudanças em um objeto notificam automaticamente seus observadores.

Package: unidade de código reutilizável em Dart. Publicado no `pub.dev` e pode ser incluído em outros projetos via arquivo `pubspec.yaml`.

Pathfinding: algoritmo que encontra o caminho mais curto ou viável entre dois pontos. Exemplos incluem BFS, A* e Dijkstra.

Permadeath (Morte Permanente): característica central de roguelikes onde a morte do personagem é irreversível. Não há salvação automática, continuar ou desfazer. O personagem morre e a partida acaba, criando tensão e significado em cada decisão.

Pattern Matching: recurso de linguagem que permite comparar valores contra padrões complexos. Em Dart, usado em switch expressions e destructuring.

Procedural Generation: técnica que cria conteúdo do jogo algoritmicamente ao invés de manualmente. Gera dungeons, itens e obstáculos de forma aleatória mas controlada.

Pub: gerenciador de pacotes oficial do Dart. Permite publicar e consumir bibliotecas da comunidade.

Ray Casting: técnica que lança raios para determinar visibilidade. Mais simples que shadowcasting mas menos preciso em FOV.

Refatoração: processo de reestruturar código sem alterar seu comportamento externo. Melhora qualidade, legibilidade e mantenibilidade.

Render: processo de converter estado do jogo em saída visual que o jogador pode ver. No jogo ASCII, significa colocar caracteres corretos nas posições corretas.

Roguelike: gênero de jogo caracterizado por exploração de dungeon, combate por turnos, morte permanente (permadeath) e progressão de personagem. Inspirado no clássico Rogue de 1980.

Rooms and Corridors: algoritmo de geração procedural que cria dungeons colocando salas retangulares e conectando-as com corredores. Simples e eficiente.

Random Walk: técnica de geração onde um ponto se move aleatoriamente, deixando um rastro. Cria paisagens naturais e cavernas.

Sealed Class: classe cuja herança é restrita a classes específicas definidas no mesmo arquivo. Garante que todos os subtipos são conhecidos e gerenciáveis.

Save Scumming: prática de salvar o jogo imediatamente antes de momentos críticos e recarregar se algo der errado, efetivamente anulando o risco. Roguelikes clássicos combatem essa prática escrevendo o save apenas ao sair do jogo e apagando-o ao carregar — isso preserva o contrato do permadeath.

Seed (Semente): valor inicial que alimenta um gerador de números aleatórios. Mesma seed produz mesma sequência de números, permitindo reproduzir mapas e combates idênticos para debug e testes.

Spawn Rate: taxa com que inimigos ou itens aparecem num andar ou sala. Um spawn rate alto lota a masmorra de monstros; baixo demais deixa o jogo vazio. Parâmetro crítico de balanceamento.

Serialização: processo de converter objetos em um formato que pode ser armazenado ou transmitido, como JSON. Essencial para sistema de save/load.

Shadowcasting: algoritmo avançado que calcula campo de visão de forma rápida e realista, tratando linha de visão e obstáculos corretamente.

Sprite: imagem 2D que representa um objeto no jogo. Em jogos ASCII, um sprite é um ou alguns caracteres que representam uma entidade.

SDK (Software Development Kit): conjunto de ferramentas para desenvolver aplicações. O Dart SDK inclui compilador, runtime e bibliotecas padrão.

SOLID: acrônimo para cinco princípios de design: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion. Promove código flexível e sustentável.

Spawn: ato de criar uma nova entidade no jogo, como inimigo aparecendo em uma localização específica.

State Pattern: padrão de design que permite um objeto alterar seu comportamento quando seu estado interno muda. Implementado naturalmente com sealed classes em Dart.

Strategy Pattern: padrão de design que define uma família de algoritmos, encapsula cada um e os torna intercambiáveis. Permite selecionar algoritmo em tempo de execução.

Stream: sequência de eventos assíncronos em Dart. `StreamController` emite eventos, `listen()` os observa. Usado com Observer Pattern para reação a eventos sem acoplamento direto.

StringBuffer: estrutura em Dart para construir strings eficientemente, acumulando caracteres sem recriação de objetos. Essencial em renderização ASCII para montar cada frame completo antes de exibir.

Teste Unitário: teste automatizado que verifica comportamento de uma unidade de código isoladamente. Fundamental para garantir qualidade e evitar regressões.

Test-Driven Development (TDD): metodologia que escreve testes antes do código. Garante cobertura de teste e design orientado a testes.

Tile: unidade de grid no mapa, geralmente representado por um caractere ASCII. Cada tile ocupa uma posição [x, y] no dungeon.

Tactical Combat: estilo de combate em que cada decisão importa — posicionamento, ordem dos ataques, uso de itens, escolha do alvo. Contrasta com combate puramente mecânico. Roguelikes tendem a forçar combate tático porque permadeath penaliza pressa.

Tile-based Movement: movimentação restrita a uma grade de tiles (células). Cada passo leva exatamente de um tile ao adjacente. Simplifica colisão, pathfinding e FOV — e é a base da maioria dos roguelikes clássicos.

Turn-Based: sistema de jogo onde ações ocorrem em turnos sequenciais. O jogador faz uma ação, inimigos respondem, e o ciclo continua. Comum em roguelikes.

UI (User Interface): interface visual através da qual o jogador interage com o jogo. Inclui menus, HUD, diálogos e controles.

XP (Experience Points): pontos ganhos ao derrotar inimigos ou completar objetivos. Acumulam para permitir level up do personagem.

Do Mundo Real à Masmorra

Além das entradas alfabéticas acima, esta lista traduz o vocabulário da masmorra para conceitos de programação (e vice-versa). Use-a para ver como mecânicas de roguelike se expressam em Dart e na arquitetura do jogo. É um mapa bidirecional: comece no RPG e encontre o código, ou comece no código e encontre a mecânica do jogo.

Masmorra → *Loop principal (game loop)* O ciclo que mantém o jogo vivo: entrada → processamento → saída. A cada turno, o jogo lê ações, atualiza o estado e desenha na tela.

Turno → *Iteração do loop* Um ciclo completo da masmorra: o jogador age, os inimigos reagem, o mapa é desenhado.

Personagem → *Classe, Objeto* O herói é uma instância de uma classe `Jogador`, com atributos (nome, HP, inventário) e métodos (sofrer dano, usar item).

Atributos (HP, força, agilidade) → *Campos/Propriedades* Dados que definem o estado do personagem. HP é um int, força é um int, nome é uma String.

Inventário → *Lista (List)* Coleção ordenada de itens. Você pega itens em sequência (índice 0, 1, 2...).

Loja do Mercador → *Mapa (Map)* Cada item tem um nome (chave) e um preço (valor). "Espada" → 50 ouro, "Poção" → 20 ouro.

Salas visitadas (exploração) → *Conjunto (Set)* Você marca quais salas já visitou. Não importa a ordem, só se foi lá ou não. Muito rápido para verificar.

Morte permanente (permadeath) → *Sem try/catch, sem undo* Erro no combate? Game over. Sem salvação, sem continuar. É a verdade dos roguelikes.

Magia/Habilidades → *Polimorfismo (@override)* Zumbi sofre dano diferente de Esqueleto. Cada inimigo tem seu próprio `descreverAcao()`.

Herança de classes de inimigos → *Herança (extends)* Zumbi, Esqueleto, Lobo são todos Inimigos. Compartilham HP, método `sofrerDano()`, mas cada um age diferente.

Poder compartilhado (todos sangram) → *Mixin* Toda criatura que respira tem um método `sofrerDano()`. Em vez de copiar em cada classe, usamos mixin `Combatente`.

Escudo mágico (null safety) → *Null Safety* Dart garante que uma variável nunca será null sem sua permissão. Nenhuma surpresa de NullPointerException no meio do combate.

Conhecimento tardio (só sabe quando pegar) → *late* Você declara uma variável mas só a inicializa depois, quando precisar. `late String nomeMasmorra;`

Contrato da criatura → *Classe abstrata (abstract class)* Uma classe `Inimigo` define o contrato: toda criatura viva tem HP, pode sofrer dano, descreve sua ação.

FOV (campo de visão) → *Algoritmo, Set* O que o herói enxerga. Calcula-se com raycasting ou shadowcasting. Resulta em um Set de tiles visíveis.

Névoa de guerra → *Rastreamento de estado* Tiles que você já visitou (explorados) vs. nunca visitou (nunca visto). Booleano ou enum.

Pathfinding (caminho ao inimigo) → *Algoritmo A* ou BFS* Dado dois pontos (herói e inimigo), encontra o caminho mais curto. BFS é simples, A* é otimizado.

Geração procedural → *Random + Algoritmo* MapaMasmorra gera salas e corredores aleatoriamente a cada partida. Mesma seed = mesmo mapa.

Colisão (parede, inimigo) → *Verificação booleana* Posição (x, y) está ocupada? if (tilemap[x][y].temParede) bloqueia movimento.

Tile (célula do mapa) → *Posição (x, y)* Cada quadrado do grid. Uma sala é um conjunto de tiles. ASCII é um sprite simples.

Sprite ASCII → *Caractere único (char)* Um personagem, um inimigo, uma parede. 'Z' para zumbi, '@' para herói, '**#**' para parede.

Roguelike → *Gênero com regras* Exploração de dungeon, combate por turnos, morte permanente, progressão de personagem, geração procedural.

Save (salvar partida) → *Serialização + JSON* Objeto Jogador → Map → String JSON → Arquivo em disco. Persistência.

Load (carregar partida) → *Desserialização + JSON* String JSON → Map → Objeto Jogador. Reconstruir tudo da memória persistente.

Boss final → *Padrão especial* Inimigo com mais HP, mais dano, comportamento único. Marca o fim de um andar.

Experience points (XP) → *Contador inteiro* int xp acumula. Ao atingir threshold, level up. Base para crescimento de personagem.

Level up → *Aumento de atributos* Força sobe, HP máximo sobe. Resultado de acumular XP.

Factory construtor (criar inimigo de dados) → *Factory Constructor* `Inimigo.deJSON(Map dados)` retorna Zumbi, Esqueleto, ou Lobo conforme dados. Lógica centralizada.

Padrão Strategy → *Strategy Pattern* Cada inimigo tem uma IA: `patrulhar()`, `perseguir()`, `atacar()`. Algoritmo intercambiável por tipo.

Padrão Command → *Command Pattern* Ação do jogador: move norte, ataca, pega item. Cada uma é um Command que pode ser desfeita, registrada, ou executada depois.

Padrão State → *State Pattern* Inimigo em estado Patrulha, Alerta, Perseguição. Muda de estado quando vê jogador. Sealed class implementa isso bem.

Padrão Observer → *Observer Pattern* Quando algo morre, log ouve, UI ouve, som ouve. Event bus + Stream. Baixo acoplamento.

Máquina de estados (FSM) → *Enum + Switch ou Sealed Class* Estado da entidade: vivo, morrendo, morto. Switch no estado, executa ação apropriada.

Testabilidade → *Testes Unitários* `test('jogador sofre dano', () { ... })`. Garante que sofrerDano(10) diminui HP de 10.

Análise estática → *dart analyze* Verifica erros antes de rodar. Variáveis não usadas, tipos errados, imports desnecessários.

Gerenciador de pacotes → *pub, pubspec.yaml* Declare dependências (package:test, etc). pub.dev é o repositório.

Sandbox seguro → *DartPad* Experimente Dart no navegador sem instalar nada. Perfeito para aprender.

Estrutura do projeto → *lib/, bin/, test/* lib/ tem código reutilizável. bin/ tem main(). test/ tem testes.

Configuração de análise → *analysis_options.yaml* Regras estritas: evita patterns perigosos, força estilos. Lint é customizável.

Distribuição do jogo → *dart compile exe* Compila para executável nativo. pub global install para ferramentas.

Geração de eventos → *Enum + Pattern Matching* Comandos do jogador como enum. Switch expression extrai dados: `switch(cmd) { ... }`.

Registros heterogêneos → *Records* Retornar múltiplos valores: `(bool sucesso, String mensagem)`. Melhor que Tuple ou classe auxiliar.

Extensão de linguagem → *Extension* `extension IntUtils on int { ... }` adiciona método a int sem herança. `5.vezes(() { ... })`.

Cascata de operações → *Cascade Operator (..)* `jogador..hp = 100..ouro = 0..moverPara('inicio')`. Encadeamento.

Condicional em coleção → *Collection if* `[item1, if (condicao) item2, item3]`. Construir lista com lógica inline.

Loop em coleção → *Collection for* `[...lista1, for (item in lista2) item.upper()]`. Flatten/map inline.

Type alias → *typedef* `typedef Acao = void Function()`. Nome legível para assinatura complexa de função.

Genérico → *Generic* `List<Item>`, `Map<String, Inimigo>`. Reutiliza estrutura, força tipo.

Processamento paralelo → *Isolate* Cada Isolate é uma thread Dart. Fork para computação pesada sem bloquear UI.

Contexto de execução → *Zone* Intercepta erros, logs, timers num escopo. Útil para testes e debugging.

Padrão Factory → *Factory Pattern* `DefinicaoItem` tem factory para criar Item concreto. Centraliza criação e validação.

Abstração de dados → *Interface (implements)* Contrato de métodos que uma classe deve ter. Em Dart, `class X implements Y` força cumprir interface de Y.

Getters e setters → *Propriedades computadas* `int get hp => _hp;` expõe leitura segura. `set hp(int v) { if (v >= 0) _hp = v; }` valida escrita.

Sobrecarga de operadores → *operator overload* `class Ponto { Ponto operator+(Ponto other) { ... } }`. Usar `+`, `-`, `==` em tipos custom.

Tipo abstrato / contrato → *Sealed class* `sealed class Entidade { }` garante que só subclasses conhecidas existem. Exaustividade em switch.

Como Usar Este Mapa Conceitual

- **Conceitos de programação:** comece na coluna "Termo de Programação". Se não souber o que é um `Stream`, procure na tabela e encontre a linha correspondente ao jogo.
- **Mecânicas da masmorra:** comece na coluna "Termo RPG". Para saber como "morte permanente" se relaciona com código, veja a coluna do meio e a descrição.
- **Padrões de design:** use as linhas de Strategy, Command, State e Observer para ligar teoria ao que o livro implementa no combate e na IA.
- **Dúvidas durante o desenvolvimento:** se você está implementando um feature e não sabe qual padrão usar, procure na coluna "Termo RPG" para encontrar sugestões de código.

Apêndice E - Terminal, ANSI e Solução de Problemas

Metade dos bugs de um roguelike ASCII não estão no código: estão no terminal que o executa. Este apêndice é o seu kit de primeiros socorros quando o jogo roda mas aparece torto, colorido demais, colorido de menos, ou quando caracteres viram losangos misteriosos.

Você já tem o jogo funcionando. Ele compila, os testes passam, o `dart run` não reclama. Mesmo assim, uma hora alguém (você ou um amigo que você convidou para jogar) vai abrir o terminal e ver: códigos estranhos tipo `\x1B[31m` em vez de cor vermelha, caixas quebradas em vez de molduras (`â•”â•â•—` em vez de ╔═╗), ou emojis e ícones que viraram quadrados vazios. Este apêndice reúne os problemas mais comuns e suas soluções.

Entendendo códigos ANSI

ANSI escape codes são sequências especiais começando com `\x1B[` (o caractere ESC) que o terminal interpreta como comandos: mudar cor, mover cursor, limpar tela. Quando você vê `\x1B[31mTexto\x1B[0m` literalmente na tela, significa que o terminal **não está interpretando** ANSI — está imprimindo os códigos como texto cru.

```
const vermelho = '\x1B[31m';
const reset = '\x1B[0m';
print('${vermelho}HP crítico!$reset');
```

Em um terminal que suporta ANSI, a frase aparece vermelha. Em um que não suporta, você vê literalmente `←[31mHP crítico!←[0m`.

Problemas comuns e soluções

Problema 1: códigos ANSI aparecem como texto

Sintoma: Você vê `\x1B[31m` ou `←[31m` no terminal em vez das cores.

Causa: Terminal sem suporte ANSI (`cmd.exe` antigo, PowerShell 5.x sem configuração).

Soluções:

- **Windows:** Instale Windows Terminal da Microsoft Store (gratuito). Suporta ANSI nativamente e é rápido.
- **PowerShell:** Use versão 7+. A versão 5 que vem com o Windows tem suporte ANSI limitado.
- **Fallback em código:** Detecte o terminal e desabilite cores quando necessário.

```
import 'dart:io';

bool get suportaAnsi {
  if (Platform.isWindows) {
    // Windows Terminal define WT_SESSION; cmd.exe antigo não
    return Platform.environment.containsKey('WT_SESSION') ||
           Platform.environment['TERM'] != null;
  }
  return stdout.hasTerminal;
}

String colorir(String texto, String codigoAnsi) {
  if (!suportaAnsi) return texto;
  return '$codigoAnsi$texto\x1B[0m';
}
```

Problema 2: caracteres box-drawing quebrados

Sintoma: Você escreveu `╔═══╗` no código e vê `â•”â•â•â•â•—` no terminal, ou aparece `???`, ou quadrados vazios.

Causa: Encoding errado. O terminal não está lendo UTF-8.

Soluções:

- **Windows:** Antes de rodar `dart run`, execute `chcp 65001` no mesmo terminal. Isso muda a página de código para UTF-8.
- **Verifique o arquivo:** Confirme que seus `.dart` estão salvos em UTF-8 (sem BOM). Quase todos os editores modernos fazem isso por padrão.
- **Fonte do terminal:** Algumas fontes não têm todos os caracteres Unicode. Troque para JetBrains Mono, Fira Code, DejaVu Sans Mono ou Consolas.

Problema 3: alinhamento quebrado

Sintoma: A arte ASCII aparece torta, colunas desalinhadas, bordas que não fecham.

Causa: Fonte proporcional (não monoespaçada).

Solução: Troque para fonte monoespaçada nas configurações do terminal:

- Windows Terminal: Settings → Profile → Appearance → Font face.
- macOS Terminal.app: Preferences → Profiles → Text → Font.
- iTerm2: Preferences → Profiles → Text → Font.
- Linux GNOME Terminal: Preferences → Profile → Text → Custom font.

Problema 4: emojis e ícones viram quadrados

Sintoma: Você usa um emoji colorido no código (por exemplo, símbolo de moeda), mas aparece □ ou `?`.

Causa: A fonte do terminal não tem os glifos dos emojis.

Soluções:

- Use fontes com suporte a emojis: JetBrains Mono com Nerd Fonts, Cascadia Code, ou similar.
- Evite emojis e use caracteres ASCII simples: `$` para ouro, `*` para item, `@` para jogador.
- Crie uma camada de abstração que troca emojis por ASCII quando detectar falta de suporte.

```
// Prefira ASCII no terminal (evita quadrados se a fonte não tiver
↪  emoji):
String get iconeOuro => '\$';
String get iconeInimigo => 'X';
```

Problema 5: cores muito fracas ou invertidas

Sintoma: Vermelho parece rosa, azul invisível sobre fundo escuro, texto ilegível.

Causa: Tema do terminal com paleta ruim, ou contraste insuficiente.

Soluções:

- Troque para um tema com alto contraste: Solarized Dark, Dracula, One Dark, Gruvbox.
- Use cores **brilhantes** (códigos 90-97) em vez das normais (30-37) para fundo escuro:
  - `\x1B[91m` (vermelho brilhante) em vez de `\x1B[31m` (vermelho normal).
- Evite combinações ruins: azul escuro em fundo preto é quase invisível.

Problema 6: tela pisca ou cursor desaparece

Sintoma: Ao redesenhar o mapa a cada turno, a tela pisca visivelmente, ou o cursor some.

Causa: Você está limpando a tela com `print` de muitas linhas em branco.

Solução: Use ANSI para limpar tela e reposicionar cursor:

```
void limparTela() {
  stdout.write('\x1B[2J\x1B[H'); // limpa tela + cursor no topo
}

void esconderCursor() => stdout.write('\x1B[?25l');
void mostrarCursor()  => stdout.write('\x1B[?25h');
```

Chame `esconderCursor()` no início do jogo e `mostrarCursor()` ao sair. Redesenhe com `limparTela()` antes de imprimir o novo frame.

Problema 7: `stdin.readLineSync()` bloqueia tudo

Sintoma: Você quer input sem Enter (pressionar W e já mover), mas `readLineSync` exige Enter.

Causa: `stdin` está em modo **line mode** por padrão.

Solução: Coloque o terminal em modo raw:

```
import 'dart:io';

stdin.echoMode = false;
stdin.lineMode = false;

int byte = stdin.readByteSync(); // lê uma tecla só, sem Enter
```

Lembre-se de **restaurar** os modos antes de sair, ou o terminal do usuário fica estranho depois:

```
void sairLimpo() {
  stdin.echoMode = true;
  stdin.lineMode = true;
  mostrarCursor();
  exit(0);
}
```

Teste rápido do terminal

Rode este snippet para diagnosticar seu terminal em 5 segundos:

```
void main() {
  print('\x1B[31mVermelho\x1B[0m \x1B[32mVerde\x1B[0m
↪  \x1B[34mAzul\x1B[0m');
  print('Box: ╔═══╗');
  print('     ║ OK ║');
  print('     ╚═══╝');
  print('Emoji: ❤');
  print('UTF: ção — áéíóú');
}
```

Se você vê cores, caixa alinhada com cantos arredondados bonitos e acentos corretos, seu terminal está pronto. Se algum desses falhar, volte à seção correspondente acima.

Recomendações finais por sistema

Windows: Windows Terminal + PowerShell 7 + fonte Cascadia Code ou JetBrains Mono.

macOS: iTerm2 + zsh + fonte JetBrains Mono.

Linux: Alacritty ou Kitty (performance) ou GNOME Terminal (padrão do Ubuntu) + fonte JetBrains Mono.

Qualquer sistema: se o terminal não quiser cooperar, ative um modo "plain" no seu jogo que desliga cores ANSI e troca caracteres Unicode por ASCII puro. Assim o jogo funciona em qualquer canto, mesmo em um servidor SSH precário ou num CI que não sabe o que é cor.

Mais fundo do que isso

Se quiser se aprofundar, existem bibliotecas Dart específicas para manipulação de terminal que abstraem muito do que foi mostrado aqui: `dart_console`, `ansi_styles`, `tint`. Elas resolvem detecção de suporte ANSI, paletas de cores e input raw de forma portável. Mas, para este livro, trabalhamos com o que a biblioteca padrão `dart:io` oferece — porque entender o que acontece em baixo nível deixa você pronto para usar qualquer biblioteca depois.

Apêndice F - Records e Extensions: recursos Dart 3 que merecem mais atenção

Um bom tesouro raramente aparece na primeira sala. Dois recursos do Dart 3 ficaram discretos ao longo do livro, usados en passant quando conveniente, mas jamais apresentados com o respeito que merecem. Este apêndice corrige isso.

Ao longo dos 37 capítulos, usamos dezenas de construções do Dart 3 — sealed classes, pattern matching, enhanced enums, null safety. Dois recursos apareceram de raspão, sempre que um atalho elegante se tornava útil, mas nunca ganharam um capítulo próprio: **Records** e **Extensions**. Este apêndice os apresenta com calma. Depois de ler, você provavelmente vai querer voltar ao próprio código e refatorar vários lugares.

F.1 Records: tuplas com nome e tipo

Um **record** é uma forma leve de agrupar valores sem precisar criar uma classe. Pense nele como uma tupla tipada: você junta algumas variáveis numa estrutura, cada uma com seu tipo, e passa adiante.

```
// Record anônimo: dois campos posicionais
(int, String) parUsuario = (42, 'Kleber');
print(parUsuario.$1); // 42
print(parUsuario.$2); // Kleber

// Record nomeado: campos com rótulos
({int id, String nome}) usuario = (id: 42, nome: 'Kleber');
print(usuario.id);   // 42
print(usuario.nome); // Kleber
```

Quando usar records em vez de classes

Records brilham quando você precisa retornar múltiplos valores de uma função e criar uma classe seria excessivo. No contexto da masmorra:

```
// Antes: retornar dois valores exigia uma classe auxiliar ou lista
↪  não-tipada
(int dano, bool critico) calcularAtaque(int forca, int destreza) {
  final dano = forca + 3;
  final critico = destreza > 15;
  return (dano, critico);
}

void main() {
  final (dano, critico) = calcularAtaque(10, 18);
  print('Dano: $dano (crítico? $critico)');
}
```

Repare: o destructuring `final (dano, critico) = ...` vem de graça. Não precisa acessar `.$1` e `.$2`.

Records com nomes

Para APIs mais legíveis, prefira nomes:

```
({int hp, int maxHp, int ouro}) lerEstado(Jogador j) {
  return (hp: j.hp, maxHp: j.maxHp, ouro: j.ouro);
}

final estado = lerEstado(meuJogador);
print('HP: ${estado.hp}/${estado.maxHp} - Ouro: ${estado.ouro}');
```

Records em pattern matching

Eles combinam naturalmente com `switch`:

```
String descreverAtaque((int, bool) resultado) {
  return switch (resultado) {
    (0, _) => 'Errou!',
    (final d, true) => 'CRÍTICO! Dano: $d',
    (final d, false) => 'Acerto normal. Dano: $d',
  };
}
```

Quando NÃO usar records

Records são imutáveis e não têm métodos próprios. Se o conjunto de dados tem comportamento (métodos), identidade (mais do que um par de valores iguais) ou vai crescer com o tempo, use uma classe. Regra prática: record para *dados de passagem*, classe para *entidades*.

F.2 Extensions: adicionando métodos a tipos existentes

Uma **extension** permite acrescentar métodos a tipos que você não pode (ou não quer) modificar — tipos da biblioteca padrão, pacotes de terceiros, ou até suas próprias classes que você prefere não inchar.

```
extension StringMasmorra on String {
  String get emCaixa => '[ $this ]';
  String repetirTres() => '$this$this$this';
  bool get pareceComando => startsWith('/') && length > 1;
}

void main() {
  print('entrar'.emCaixa);      // [ entrar ]
  print('.'.repetirTres());     // ...
  print('/norte'.pareceComando); // true
}
```

Extensions no contexto da masmorra

Suponha que você queira formatar qualquer `int` como barra de HP visual:

```
extension BarraVida on int {
  String barra(int maximo, {int largura = 10}) {
    final preenchido = (this / maximo * largura).round();
    return '[${'' * preenchido}${'' * (largura - preenchido)}]';
  }
}

void main() {
  print(7.barra(10));  // []
  print(3.barra(10));  // []
}
```

Ou data/hora amigável para logs de combate:

```
extension DataLog on DateTime {
  String get hhmm {
    final h = hour.toString().padLeft(2, '0');
    final m = minute.toString().padLeft(2, '0');
    return '$h:$m';
  }
}

void main() {
  print('[${DateTime.now().hhmm}] Ataque iniciado.');
}
```

Extensions com genéricos

Você pode estender tipos genéricos:

```
extension ListaMasmorra<T> on List<T> {
  T? aleatorio(Random rng) => isEmpty ? null :
↪  this[rng.nextInt(length)];
  T? get primeiroOuNulo => isEmpty ? null : first;
```

```
}
```

Cuidados com extensions

- Extensions **não podem** ser chamadas dinamicamente: o Dart precisa conhecer o tipo estático em tempo de compilação.
- Evite criar extensions muito genéricas com nomes curtos (ex: `.x` em `String`) — o código fica enigmático.
- Se várias extensions definem o mesmo método para o mesmo tipo, o Dart pede desambiguação.

F.3 Combinando records e extensions

Os dois recursos se complementam. Você pode ter uma função que retorna record, e extensions que formatam esse record:

```
typedef Resultado = ({int dano, bool critico, bool esquivou});

Resultado atacar(int forca, Random rng) {
  if (rng.nextDouble() < 0.1) return (dano: 0, critico: false,
  ↪  esquivou: true);
  final crit = rng.nextDouble() < 0.15;
  final dano = forca * (crit ? 2 : 1);
  return (dano: dano, critico: crit, esquivou: false);
}

extension FormatarResultado on Resultado {
  String descrever() {
    if (esquivou) return 'Esquiva!';
    if (critico) return 'CRÍTICO! ($dano de dano)';
    return 'Acerto: $dano de dano';
  }
}

void main() {
```

```
  final r = atacar(10, Random());
  print(r.descrever());
}
```

Repare que `typedef Resultado` dá um nome amigável ao record para usar em assinaturas.

F.4 Quando voltar e refatorar

Após ler este apêndice, você provavelmente vai enxergar pelo menos três lugares no seu código que poderiam ser mais claros com records (funções que retornam `Map<String, dynamic>` ou listas) ou extensions (helpers espalhados em funções top-level). Faça isso aos poucos: refatorações pequenas, uma de cada vez, com os testes do capítulo 32 te protegendo.

E, quando tiver tempo, releia a documentação oficial em https://dart.dev/language/records e https://dart.dev/language/extension-methods. Há mais detalhes sobre igualdade de records, cópias imutáveis e extensions estáticas que valem a leitura.

Bibliografia e Referências

Nenhum código é escrito em terreno virgem. Este livro existe porque muitas outras pessoas escreveram, documentaram, debateram e compartilharam o que sabiam. As fontes a seguir moldaram, direta ou indiretamente, as decisões técnicas e pedagógicas desta obra.

Documentação oficial Dart

A documentação oficial da linguagem Dart foi a principal referência técnica ao longo de todos os capítulos. Ela é atualizada com frequência e continuará relevante muito depois deste livro envelhecer.

- **Dart Language Tour**. Documentação oficial. Disponível em: https://dart.dev/language
- **Effective Dart: Style, Documentation, Usage, Design**. Guia de estilo oficial. Disponível em: https://dart.dev/effective-dart
- **Dart 3: Records, Patterns, Class Modifiers**. Documentação de recursos introduzidos em Dart 3. Disponível em: https://dart.dev/language/records e https://dart.dev/language/patterns
- **Dart 3 Release Notes**. Histórico oficial de mudanças da linguagem. Disponível em: https://dart.dev/guides/language/evolution
- **Introducing Dart 3**. Anúncio oficial da equipe Dart sobre a versão 3, cobrindo sealed classes, records e patterns. Medium, Dart Developer, 2023.
- **Sound Null Safety**. Documentação sobre o sistema de null safety do Dart. Disponível em: https://dart.dev/null-safety
- **Dart Extension Methods**. Referência oficial sobre métodos de extensão. Disponível em: https://dart.dev/language/extension-methods
- **Dart Core Libraries Reference**. Referência das bibliotecas `dart:core`, `dart:async`, `dart:io` e `dart:convert`. Disponível em: https://api.dart.dev

Game Design e Roguelikes

O entendimento sobre roguelikes, permadeath, loot tables e progressão de personagem deriva de uma tradição viva na comunidade de desenvolvimento de jogos.

- **Berlin Interpretation** (2008). Conjunto de critérios que define o gênero roguelike. Roguelike Celebration Wiki.
- *Articles on Roguelike Development*. **RogueBasin**. Compêndio colaborativo sobre técnicas de geração procedural, combate por turnos e design de masmorras. Disponível em: http://roguebasin.com
- CRAWFORD, Chris. *The Art of Computer Game Design*. McGraw-Hill, 1984 (reedição digital disponível gratuitamente).
- SALEN, Katie; ZIMMERMAN, Eric. *Rules of Play: Game Design Fundamentals*. MIT Press, 2003.

Engenharia de Software e Boas Práticas

Os capítulos sobre refatoração, testes, design patterns e organização de código foram fortemente influenciados por clássicos da engenharia de software.

- FOWLER, Martin. *Refactoring: Improving the Design of Existing Code*. 2. ed. Addison-Wesley, 2018.
- MARTIN, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.
- GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- HUNT, Andrew; THOMAS, David. *The Pragmatic Programmer*. 20th Anniversary Edition. Addison-Wesley, 2019.
- BECK, Kent. *Test-Driven Development: By Example*. Addison-Wesley, 2003.

Pedagogia de Programação

A estrutura incremental deste livro — cada capítulo expande o jogo anterior — dialoga com uma tradição de ensino de programação por projetos.

- NYSTROM, Robert. *Crafting Interpreters*. 2021. Disponível em: https://craftinginterpreters.com

- NYSTROM, Robert. *Game Programming Patterns*. 2014. Disponível em: https://gameprogrammingpatterns.com
- PAPERT, Seymour. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, 1980.
- GUZDIAL, Mark. *Learner-Centered Design of Computing Education*. Morgan & Claypool, 2015.

Comunidades e recursos online

Ao longo da escrita deste livro, dúvidas específicas foram resolvidas consultando discussões em comunidades ativas de desenvolvedores Dart.

- **Dart Discord Server** e **r/dartlang** no Reddit.
- **GitHub dart-lang/sdk issues and discussions**.
- **Stack Overflow**, tag `dart`.
- **Pub.dev**, repositório oficial de pacotes Dart, utilizado para inspeção de APIs reais em produção.

Nota do autor

Esta bibliografia é deliberadamente enxuta. Há centenas de outras fontes que teriam merecido citação, mas preferi listar apenas aquelas a que recorri com frequência durante a escrita. Para quem deseja aprofundar, cada obra acima aponta para dezenas de outras — siga as trilhas que fizerem sentido para seu próprio caminho.

Sobre o Autor

Kleber de Oliveira Andrade é doutor em Engenharia pela Escola de Engenharia de São Carlos (EESC/USP, 2016), mestre pela mesma instituição (2011) e bacharel em Ciência da Computação pela Escola de Engenharia de Piracicaba (2008).

A Origem

A relação com tecnologia começou cedo. Videogames desde os três anos, começando por um Atari 2600 que nunca mais saiu de vista, e as primeiras linhas de código aos dez, em Basic, numa máquina que parecia mágica. Essa curiosidade nunca desapareceu. Pelo contrário: se aprofundou ao longo de mais de vinte e cinco anos como programador e cerca de doze anos como professor universitário, período em que publicou pesquisas em reabilitação robótica, serious games, inteligência artificial aplicada e robótica móvel.

A Trajetória

A trajetória combina rigor acadêmico e pragmatismo de mercado de forma que raramente se separam. Trabalhou em consultoria de sistemas, liderou times de desenvolvimento em startups, e atualmente é CEO da **Koativa** (empresa focada em consultoria e desenvolvimento de soluções) e CTO da **Pagstar**. Essa combinação informa profundamente a forma como pensa engenharia de software: como lidar com o que se pode medir, o que não se pode, e como equilibrar ambos em contexto real.

Por Que Este Livro?

Masmorra ASCII nasce do cruzamento entre duas convicções que nunca o deixaram quieto: a paixão por ensinar programação e a fascinação por design de jogos. A ideia central é simples, mas poderosa: um roguelike no terminal (puro ASCII, puro Dart, puro código) pode ensinar orientação a objetos, padrões de projeto e inteligência artificial de forma mais viva do que qualquer exercício de lista encadeada jamais conseguiria.

É a mesma convicção que guia suas aulas e seus projetos: **software se aprende construindo algo que importa**. Não se aprende em vácuo. Aprende-se quando há algo palpável no final do caminho: um jogo que funciona, um mundo que respira, uma masmorra que você criou com suas próprias mãos.

Este livro é, no fundo, uma reflexão sobre como ensinar programação de forma que os aprendizes saiam diferentes, não apenas com técnica, mas com o entendimento profundo de que são eles, os programadores, que criam mundos.

Contato

Se quiser conversar sobre código, design de jogos, educação em programação ou qualquer das obsessões acima:

- **E-mail:** pdjkleber@gmail.com
- **Site:** https://kleberandrade.dev
- **GitHub:** https://github.com/kleberandrade
- **LinkedIn:** https://www.linkedin.com/in/kleberandrade/

Quando você terminar a leitura e construir sua masmorra, você é bem-vindo para compartilhar, especialmente se encontrou um easter egg que eu esqueci de documentar.