Refatorando com padrões de projeto

Um guia em Java

Casa do Código

MARCOS BRIZENO

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

[2018]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-94188-21-2

EPUB: 978-85-94188-22-9

MOBI: 978-85-94188-23-6

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

AGRADECIMENTOS

Gostaria de agradecer à minha esposa Ingrid, pela paciência que precisou ter enquanto passava meu tempo escrevendo, e à nossa gata Maggie, que nos ajudou a manter a motivação.

Também gostaria de agradecer aos meus pais, Marcos e Lúcia, que desde cedo incentivaram meu interesse em ler e escrever junto ao meu irmão Brunno.

Consegui realizar este objetivo graças ao esforço e dedicação da minha família.

SOBRE O AUTOR

Marcos Brizeno é Cientista da Computação pela Universidade Estadual do Ceará e Consultor na ThoughtWorks Brasil. Apaixonado por Engenharia de Software, em especial Metodologias Ágeis, gosta dos desafios de desenvolver software e construir soluções para os problemas dos usuários. Publica regularmente em seu blog (http://brizeno.wordpress.com), e gosta de jogar videogames para passar o tempo e ter novas ideias.

PREFÁCIO

Estudar os conceitos e a estrutura dos padrões de projeto lhe dará um maior entendimento sobre design orientado a objetos, permitindo implementar soluções flexíveis e fáceis de evoluir. Se você entende os conceitos básicos de Orientação a Objeto e quer dar o próximo passo, este livro foi escrito pensando em você.

Ao identificar, coletar e formalizar padrões de projetos, os autores buscam uma maneira mais fácil de catalogar e passar o conhecimento de soluções para outras pessoas. Ao longo dos capítulos, vamos explorar mais sobre como aplicar essas soluções de maneira mais efetiva no dia a dia através das técnicas de refatoração.

Este livro apresenta exemplos práticos de como identificar problemas e entender melhor o contexto da situação, para em seguida implementar um padrão de projeto que melhore o código. Ao modificar o código, veremos como aplicar técnicas de refatoração para que o processo seja feito de maneira responsável e garantindo o funcionamento da aplicação por meio de testes unitários.

O livro foi escrito para tornar a leitura fácil, com muitos exemplos de código, diagramas e explicações focadas no problema sendo explorado. Os capítulos descrevendo os padrões podem ser lidos individualmente, caso não queira seguir uma ordem linear. Sugiro também que tente fazer as modificações sugeridas no código e até mesmo as suas próprias melhorias.

O que é necessário saber

Padrões de projeto é um tópico avançado de Orientação a Objetos (OO), logo, é necessário ter um bom conhecimento sobre OO. É esperado que você entenda os relacionamentos entre objetos, interfaces, contratos etc. Se você já tem experiência com isso, então os problemas de design ficarão bem claros, o que facilita entender a motivação por trás da aplicação dos padrões.

Os exemplos e as discussões serão feitos considerando sua aplicação em Java, por isso é necessário ter conhecimento razoável sobre a linguagem. Não é preciso conhecer todas as suas funcionalidades; os exemplos são simples e curtos. Se você já conhece bem Java, talvez ache pontos em que o código poderia ser simplificado ou diminuído utilizando algumas funcionalidades da linguagem, mas esse não é o foco do livro.

Sumário

Versão: 22.7.7

Introdução

Capítulo 1

O QUE VOCÊ QUER APRENDER

Geralmente, os autores começam explicando o que será encontrado dentro dos seus livros. Entretanto, antes disso, quero lhe sugerir uma inversão desse pensamento: o que você quer aprender com este livro? O que o levou a escolhê-lo?

O objetivo de responder essas perguntas é dar mais clareza sobre o que é mais importante para você neste momento, aproveitando melhor as partes que mais lhe interessam. Ao colocar em prática os conceitos aqui apresentados, você terá uma nova visão sobre os seus projetos e seus padrões.

1.1 O QUE VOCÊ VAI ENCONTRAR

Este livro vai apresentar, de uma maneira bem prática, 9 dos 23 padrões de projeto catalogados no livro *Design Patterns: elements of reusable object-oriented software*, de Erich Gama, Ralph Johnson, Richard Helm e John Vlissides (1994).

Quadrados e retângulos para explicar herança e um carro que tem quatro rodas para demonstrar composição são exemplos simples e bastante usados para explicar conceitos básicos, mas que

ficam bem distante da realidade do dia a dia de um desenvolvedor. A ideia do livro não é ser uma referência para padrões de projeto, mas sim apresentar com exemplos práticos e próximos da realidade como eles podem melhorar sua vida. Vamos explorar bastante o uso dos padrões de projeto, mostrando não só o código da aplicação como também testes.

Cada capítulo começa com um problema e uma amostra de código que é uma solução rápida. Depois, serão propostas extensões do problema, que vão expor algumas falhas no design inicial. Isso vai nos forçar a refatorar o código, criar uma situação para aplicar um padrão de projeto e avaliar o resultado final.

Ao refatorar o código, mostraremos quais passos tomar para aplicar as mudanças e também quais modificações de testes serão necessárias. Afinal de contas, mesmo utilizando um padrão bem conhecido e usado em várias situações, é necessário ter segurança ao fazer alterações.

Os padrões estão divididos em dois grupos: comuns e situacionais. Padrões comuns são aqueles mais facilmente encontrados em projetos, para o qual talvez você ache uma aplicação assim que ler o capítulo. Já os padrões situacionais não são tão comuns, mas resolvem bem situações específicas, desde que o contexto seja levado em conta.

1.2 COMO APROVEITAR BEM O LIVRO

O primeiro passo é participar da lista de discussão do grupo! Participe do fórum da Casa do Código (http://forum.casadocodigo.com.br), onde poderemos conversar e

trocar ideias diretamente, além de falar com outras pessoas que também estão lendo o livro.

Além do Java, vamos utilizar bastante o *JUnit* (http://junit.org/), que é o *framework* de testes mais utilizado em Java. Se você não tem experiência com *JUnit*, não se preocupe! Os testes serão bem simples e foram escritos pensando em você.

Um típico teste escrito em *JUnit* começa com um método público, sem retorno e sem parâmetros, `public void testeDeExemplo()`. Além disso, usamos a *annotation* `@Test` para executar o método como parte da suíte de testes.

No exemplo a seguir, usamos o método `assertEquals` para avaliar que o retorno da chamada `criterio.getPorPagina()` é igual a 20.

```
@Test
public void retornaResultadosPorPaginaQuandoEspecificado() {
    Parametros parametros = new Parametros("Um produto", 20);

    Criterio criterio = Busca.criarCriterio(parametros);

    assertEquals(criterio.getResultadosPorPagina(), 20);
}
```

Cada teste é organizado em três partes, seguindo o padrão AAA, isto é, *Arrange, Act, Assert* (http://c2.com/cgi/wiki?ArrangeActAssert):

- **Arranjo:** onde os dados do teste são preparados;
- **Ação:** onde a ação-alvo do teste é chamada;
- **Asserção:** onde o resultado da ação sobre os dados é validado.

É comum ver testes escritos assim, pois a separação entre o que

deve ser chamado e o que é esperado fica visualmente mais clara.

Além disso, em alguns testes usamos dublês de testes, com o *Mockito* (http://site.mockito.org). A principal utilização de dublês ao longo do livro será para validar que um método foi chamado, ou para que ele retorne um valor específico, independente da sua lógica.

No teste a seguir, usamos um *mock* pois estamos interessados apenas em saber se o método `realizarBuscaCom` do objeto `servico` foi chamado com o parâmetro `criterio`. Além disso, através dos métodos `when` e `thenReturn` do *Mockito*, garantimos que, ao chamar `servico.realizarBuscaCom(criterio)`, a lista `resultado` será retornada:

```
// /src/test/java/factory/BuscaTest.java
public class BuscaTest {

    @Test
    public void realizarBusca() {
        ServicoDeBusca servico = mock(ServicoDeBusca.class);

        Busca busca = new Busca(servico);
        ParametrosDeBusca parametros =
            new ParametrosDeBusca(TipoDeBusca.NORMAL);
        CriterioDeBusca criterio =
            new FabricaDeCriterio(parametros).criarCriterio();

        List<int> resultado = Arrays.asList(1, 2, 3);
        when(servico.realizarBuscaCom(criterio))
            .thenReturn(resultado);

        busca.por(parametros);

        verify(servico).realizarBuscaCom(criterio);
    }
}
```

Todos os exemplos de código utilizados no livro estão disponíveis em um repositório no GitHub: https://github.com/MarcosX/rppj. Se você já conhece Git e GitHub, basta clonar o repositório. Se você não conhece essas ferramentas, é possível fazer o download do código na página do repositório.

No *branch* principal (`master`), você encontrará os códigos iniciais, antes da refatoração, e no *branch* `refatorado` estarão os exemplos após a refatoração. Assim você pode comparar os códigos sempre que precisar.

Capítulo 2

REFATORAÇÃO E PADRÕES DE PROJETO

Neste capítulo, serão brevemente apresentados os conceitos de refatoração e padrões de projeto, e como eles podem ajudar no seu dia a dia. Mesmo que você seja experiente e conheça os conceitos, recomendo a leitura deste capítulo, pois os dois já estão por aí há bastante tempo, e cada pessoa tem um entendimento próprio do que eles são ou não.

2.1 O QUE É REFATORAÇÃO?

Refatoração já não é mais novidade, e o livro *Refactoring: improving the design of existing code*, de Martin Fowler (1999), já possui mais de 15 anos de existência. Mesmo assim, o assunto continua sendo relevante e pertinente tanto para novos profissionais como para quem nunca parou para entender o que é e o que não é refatoração.

A definição de refatoração por Martin Fowler pode ser entendida em 3 partes:

- **Melhorar o design existente:**

Melhorar o design existente é uma definição muito importante, pois é bem comum pensar que qualquer coisa que não seja adição de funcionalidade é refatoração. Correções não devem fazer parte do processo de refatoração; lembre-se de que o foco é melhorar o design! Por isso é melhor reduzir a quantidade de mudanças, o que nos leva à segunda parte da definição.

- **Aplicar mudanças em pequenos passos:**

  O conceito de aplicar mudanças em pequenos passos não é muito objetivo, afinal, o que são "pequenos passos"? O objetivo principal é reduzir a quantidade de mudanças, pois menos mudanças é igual a menos problemas! Se você está alterando os parâmetros de um método e percebe que algo na lógica poderia ser melhor, tome nota e volte para fazer a mudança depois.

- **Evitar deixar o sistema quebrado:**

  Essa terceira parte da definição é sobre evitar deixar o sistema quebrado, garantindo que as melhorias não acabem afetando funcionalidades existentes. Para isso, é importante ter uma boa suíte de testes, tanto com relação à cobertura quanto à facilidade e velocidade de execução. Além de executar frequentemente os testes da classe em mudança, também é importante garantir que a integração com o resto do sistema continue funcionando.

Um exemplo bem simples de refatoração seria renomear o método para que o código fique mais expressivo. Note o comentário explicando o que o método faz para facilitar o

entendimento do código:

```
//método para calcular o preço final
public long calcula(long precoBase, long bonusPorcentagem) {
    if (bonusPorcentagem == null || bonusPorcentagem < 10) {
        return precoBase;
    } else {
        return precoBase * (bonusPorcentagem/100);
    }
}
```

Um exemplo de teste seria:

```
@Test
public void calculaPrecoFinalCom20DeBonus() {
    long precoBase = 100;
    long bonusPorcentagem = 20;

    long precoFinal = calcula(precoBase, bonusPocentagem);

    assertEquals(120, precoFinal);
}
```

No entanto, se mudarmos o nome do método de uma vez, todos os testes quebrariam, pois estariam usando o método antigo. Criamos, então, um novo método que chama o anterior:

```
public long precoFinal(long precoBase, long bonusPocentagem) {
    return calcula(precoBase, bonusPocentagem);
}
```

Agora, podemos atualizar os testes chamando o novo método:

```
@Test
public void calculaPrecoFinalCom20DeBonus() {
    long precoBase = 100;
    long bonusPorcentagem = 20;

    long precoFinal = precoFinal(precoBase, bonusPocentagem);

    assertEquals(120, precoFinal);
}
```

Por fim, atualizamos o nome do método e apagamos o antigo:

```
//método para calcular o preço final
public long precoFinal(long precoBase, long bonusPorcentagem) {
    if (bonusPorcentagem == null || bonusPorcentagem < 10) {
        return precoBase;
    } else {
        return precoBase * (bonusPorcentagem/100);
    }
}
```

Apesar dos vários passos para simplesmente renomear um método e deixar o código mais expressivo, ao longo de todas as mudanças, os testes estavam sempre passando. Uma outra mudança que poderia ser feita era extrair a lógica do `if` , mas ela não é necessária para renomear o método, então deixamos para depois. Refatorações simples como renomear ou até mesmo extrair métodos podem ser feitas automaticamente pelas IDEs, permitindo mais agilidade no desenvolvimento.

2.2 TÉCNICAS DE REFATORAÇÃO

Ao aplicar refatorações, existem várias técnicas que podem ser utilizadas para alcançar a melhoria desejada. O exemplo de refatoração descrita na seção anterior é chamado de *Renomear Método*. As técnicas descrevem passos que podem ser tomados, garantindo que, a cada mudança, todos os testes continuem passando.

Martin Fowler descreve no livro *Refactoring* um conjunto de técnicas que pode ser usado em várias situações. Ao longo do livro, vamos usar algumas delas para modificar o código e aplicar os padrões.

Extrair Método

Extrair Método é uma técnica bem simples e poderosa, usada quando precisamos quebrar um método que possui mais de uma responsabilidade. O método a seguir, apesar de se chamar `desativarUsuarios`, faz muito mais do que isso.

Primeiro, ele busca os usuários para desativar, para depois executar a ação de desativá-los e, por fim, notifica os usuários por e-mail.

```
public class DesativarUsuariosWorker {
  public void desativarUsuarios() {
    RepositorioUsuarios repositorio = new RepositorioUsuarios();
    List<Usuarios> usuarios = repositorio.all().stream()
      .filter(usuario ->
          usuario.semLoginRecente() && usuario.estaAtivo())
      .collect(Collectors.toList());

    usuarios.forEach(usuario -> repositorio.desativar(usuario));
    NotificadorViaEmail.usuariosDesativados(usuarios);
  }
}
```

Vamos usar Extrair Método para retirar a responsabilidade de buscar os usuários. O primeiro passo é criar o novo método para representar a ideia que estamos extraindo e, em seguida, copiar o código, duplicando-o por enquanto.

```
public class DesativarUsuariosWorker {
  private List<Usuarios> usuariosParaDesativar() {
    RepositorioUsuarios repositorio = new RepositorioUsuarios();
    return repositorio.all().stream()
      .filter(usuario ->
          usuario.semLoginRecente() && usuario.estaAtivo())
      .collect(Collectors.toList());
  }
}
```

Ao extrair um método, é preciso estar atento às variáveis locais, pois elas também precisam existir dentro do novo método. Podemos apenas copiar as variáveis locais, passá-las como argumento ou extraí-las para seu próprio método.

Outro ponto para manter em mente é mover testes que sejam específicos ao novo método. Nem sempre esse será o caso, mas, geralmente, quando um método possui mais de uma responsabilidade, ele também vai ter vários testes unitários. Ao dividi-lo, vale a pena atualizar os testes para garantir que o novo método se comporte da mesma forma que o anterior.

Uma vez que o novo método já está criado, todo o seu escopo está funcional e possui testes unitários, o próximo passo é substituir o código pelo novo método.

```
public class DesativarUsuariosWorker {
  public void desativarUsuarios() {
    RepositorioUsuarios repositorio = new RepositorioUsuarios();
    List<Usuarios> usuarios = usuariosParaDesativar();

    usuarios.forEach(usuario -> repositorio.desativar(usuario));
    NotificadorViaEmail.usuariosDesativados(usuarios);
  }
}
```

2.3 MOVER MÉTODO

Mover Método pode ser usado quando temos um método que utiliza mais informações de outra classe do que da sua própria. Assim, reduzimos a complexidade do código, pois ele vai ter acesso a todas as informações locais da nova classe em vez de ficar perguntando antes de tomar ações.

Voltando para o exemplo anterior, o método

`desativarUsuarios` da classe `DesativarUsuariosWorker`, apesar de ter sido quebrado em mais de um método, ainda possui muito código relacionado ao repositório de usuários. Quando o nome de outro objeto aparece muitas vezes, é um sinal de que talvez a responsabilidade esteja no lugar errado.

```
public class DesativarUsuariosWorker {
  public void desativarUsuarios() {
    RepositorioUsuarios repositorio = new RepositorioUsuarios();
    List<Usuarios> usuarios = usuariosParaDesativar();

    usuarios.forEach(usuario -> repositorio.desativar(usuario));
    NotificadorViaEmail.usuariosDesativados(usuarios);
  }

  private List<Usuarios> usuariosParaDesativar() {
    RepositorioUsuarios repositorio = new RepositorioUsuarios();
    return repositorio.all().stream()
      .filter(usuario ->
        usuario.semLoginRecente() && usuario.estaAtivo())
      .collect(Collectors.toList());
  }
}
```

Vamos usar Mover Método para que `usuariosParaDesativar` seja responsabilidade da classe `RepositorioUsuarios`. O primeiro passo é duplicar o código, sem alterar nenhuma funcionalidade, copiando a lógica de `DesativarUsuariosWorker` para a classe `RepositorioUsuarios`.

```
public class RepositorioUsuarios {

  public List<Usuarios> usuariosParaDesativar() {
    return all().stream()
      .filter(usuario ->
        usuario.semLoginRecente() && usuario.estaAtivo())
      .collect(Collectors.toList());
  }
}
```

Como o método vai fazer parte de outra classe, também é necessário duplicar os seus testes unitários para garantir que ele continue funcionando como esperado. Após duplicar o método, basta substituir sua chamada e executar todos os testes novamente.

```
public class DesativarUsuariosWorker {
  public void desativarUsuarios() {
    RepositorioUsuarios repositorio = new RepositorioUsuarios();
    List<Usuarios> usuarios = repositorio.usuariosParaDesativar()
;

    usuarios.forEach(usuario -> repositorio.desativar(usuario));
    NotificadorViaEmail.usuariosDesativados(usuarios);
  }
}
```

2.4 MOVER CAMPO

Semelhante ao **Mover Método**, Mover Campo é útil quando temos um atributo que é mais usado em outra classe do que em sua própria. O benefício principal é que, ao ser definido na nova classe, garantimos que ele fique protegido de modificações externas.

No exemplo a seguir, temos a classe `CalculadorDePreco`, que possui a responsabilidade de calcular o preço final de uma corrida, e a classe `Taxi`, que representa um táxi com suas informações. Apesar de as constantes `BANDEIRA_UM` e `BANDEIRA_DOIS` estarem definidas no `Taxi`, elas são realmente usadas no `CalculadorDePreco`.

```
public class CalculadorDePreco {
  private static final float VALOR_POR_KM = 0.48f;

  public float calcularCorrida(float kmRodados, float bandeira) {
    return bandeira * (kmRodados * VALOR_POR_KM);
  }
}
```

```
public class Taxi {
  private static final float BANDEIRA_UM = 1.2f;
  private static final float BANDEIRA_DOIS = 1.8f;

  public float calcularCorrida(kmRodados) {
    CalculadorDePreco calculador = new CalculadorDePreco();
    if (ehFinalDeSemana()) {
      return calculador.calcularCorrida(kmRodados, BANDEIRA_DOIS)
;
    } else {
      return calculador.calcularCorrida(kmRodados, BANDEIRA_UM);

    }
  }
}
```

O primeiro passo é copiar os atributos na nova classe e, em seguida, referenciá-los na classe antiga:

```
public class CalculadorDePreco {
  private static final float VALOR_POR_KM = 0.48f;
  public static final float BANDEIRA_UM = 1.2f;
  public static final float BANDEIRA_DOIS = 1.8f;

  public float calcularCorrida(float kmRodados, float bandeira) {
    return bandeira * (kmRodados * VALOR_POR_KM);
  }
}

public class Taxi {

  public float calcularCorrida(kmRodados) {
    CalculadorDePreco calculador = new CalculadorDePreco();
    if (ehFinalDeSemana()) {
      float bandeira = CalculadorDePreco.BANDEIRA_DOIS;
      return calculador.calcularCorrida(kmRodados, bandeira);
    } else {
      float bandeira - CalculadorDePreco.BANDEIRA_UM;
      return calculador.calcularCorrida(kmRodados, bandeira);
    }
  }
}
```

Uma vez que os atributos só são usados na nova classe, o próximo passo é encontrar uma maneira de removê-los da classe antiga. No nosso código, vamos tirar o parâmetro `bandeira` e passar um `boolean` que sinaliza se devemos usar bandeira dois ou não. Assim, a lógica de verificação fica completamente no método `CalculadorDePreco.calcularCorrida`.

```
public class CalculadorDePreco {
  private static final float VALOR_POR_KM = 0.48f;
  private static final float BANDEIRA_UM = 1.2f;
  private static final float BANDEIRA_DOIS = 1.8f;

  public float calcularCorrida(float kmRodados,
                               boolean bandeiraDois) {
    if (bandeiraDois) {
      return BANDEIRA_DOIS * (kmRodados * VALOR_POR_KM);
    } else {
      return BANDEIRA_UM * (kmRodados * VALOR_POR_KM);
    }
  }
}

public class Taxi {

  public float calcularCorrida(kmRodados) {
    CalculadorDePreco calculador = new CalculadorDePreco();
    return calculador.calcularCorrida(kmRodados, ehFinalDeSemana(
));
  }
}
```

Agora, a lógica que antes estava dividida entre as classes ficou concentrada apenas no `CalculadorDePreco`.

2.5 EXTRAIR CLASSE

Extrair Classe pode ser entendido como uma evolução de **Extrair Método**. Usamos essa técnica quando uma classe possui

mais de uma responsabilidade. Para construir a nova classe, vamos utilizar as técnicas **Mover Campo** e **Mover Método**.

O código a seguir mostra o `BaixarRegistrosDeVendaFtpNoBancoWorker`, que tem como responsabilidades baixar arquivos de um servidor `ftp` e salvá-los no banco de dados. A classe mistura tanto informações sobre o servidor de `ftp`, como `host`, `usuario` etc., bem como informações sobre qual tabela do banco deve ser atualizada.

```
public class BaixarRegistrosDeVendaFtpNoBancoWorker {
  private String host;
  private String porta;
  private String usuario;
  private String senha;
  private RepositorioDeVendas repositorioDeVendas;

  public void requisitarFtp(String caminhoArquivo) {
    ClienteFtp cliente = new ClienteFtp(host, porta);
    cliente.login(usuario, senha);
    ArquivoFtp arquivo = cliente.buscarArquivo(caminhoArquivo);
    repositorioDeVendas.salvarDeArquivo(arquivo);
  }
}
```

Para reduzir a complexidade, vamos separar a responsabilidade de baixar arquivos de um servidor FTP da parte que os salva no banco. Para isso, o primeiro passo é criar a classe que conterá a nova responsabilidade e copiará os métodos para ela, bem como os testes específicos desse comportamento.

```
public class GerenciadorFtp {
  private String host;
  private String porta;
  private String usuario;
  private String senha;

  public ArquivoFtp requisitarFtp(String caminhoArquivo) {
    ClienteFtp cliente = new ClienteFtp(host, porta);
```

```
    cliente.login(usuario, senha);
    return cliente.buscarArquivo(caminhoArquivo);
  }
}
```

Em seguida, vamos substituir o código antigo pela nova classe. No nosso exemplo, vamos apenas chamar `GerenciadorFtp.requisitarFtp`, e passar o arquivo recebido para `salvarArquivoNoBanco`.

```
public class BaixarRegistrosDeVendaFtpNoBancoWorker {
  private RepositorioDeVendas repositorioDeVendas;

  public void requisitarFtp(String caminhoArquivo) {
    GerenciadorFtp genrenciador = new GerenciadorFtp();
    ArquivoFtp arquivo =
      genrenciador.requisitarFtp(caminhoArquivo);
    repositorioDeVendas.salvarDeArquivo(arquivo);
  }
}
```

Agora, com os métodos simplificados e as responsabilidades separadas, fica até mais fácil encontrar nomes mais sugestivos para as classes, deixando claro o que cada uma faz. A classe `GerenciadorFtp` é genérica o suficiente para ser usada por qualquer outro *worker*. Já a classe `BaixarRegistrosDeVendaFtpNoBancoWorker` pode ser nomeada para apenas `SalvarRegistroDeVendasWorker`.

```
public class SalvarRegistroDeVendasWorker {
  private RepositorioDeVendas repositorioDeVendas;

  public void requisitarFtp(String caminhoArquivo) {
    GerenciadorFtp genrenciador = new GerenciadorFtp();
    ArquivoFtp arquivo =
      genrenciador.requisitarFtp(caminhoArquivo);
    repositorioDeVendas.salvarDeArquivo(arquivo);
  }
}
```

```
public class GerenciadorFtp {
  private String host;
  private String porta;
  private String usuario;
  private String senha;

  public ArquivoFtp requisitarFtp(String caminhoArquivo) {
    ClienteFtp cliente = new ClienteFtp(host, porta);
    cliente.login(usuario, senha);
    return cliente.buscarArquivo(caminhoArquivo);
  }
}
```

2.6 O QUE SÃO PADRÕES DE PROJETO?

Assim como boa parte da área de computação, padrões de projeto também vieram da engenharia, neste caso mais especificamente da arquitetura. Portas geralmente são colocadas nos cantos da parede para que elas não ocupem muito espaço ao abrir, certo? Isso é um exemplo de padrão arquitetural.

Uma definição de padrão de projeto de que gosto bastante também é dividida em três partes: **uma solução comum** para **um problema** em **um determinado contexto**.

Voltando ao problema da porta, posicioná-la no canto é uma solução bem comum. Mas e o contexto?

Imagine que agora a porta é na entrada de uma agência bancária. Ela seria giratória e geralmente é posicionada no centro da parede. Aplicar o padrão de posicionamento da porta no canto não faz muito sentido aqui, embora ainda possa ser utilizado. É bastante comum ver aplicações de padrões de projeto que não fazem muito sentido, porque o contexto foi totalmente ignorado.

Dizer que um padrão é uma solução comum implica que

nenhum padrão é "criado", mas sim documentado. Um bom exemplo é o livro da Gangue dos Quatro (*Gang of Four*), *Design Patterns: elements of reusable object-oriented software* (comentado no capítulo anterior), no qual os 23 padrões documentados não são criações dos autores; eles vieram de observações e generalizações de situações comuns.

Para que exista uma solução, é preciso que primeiro se tenha o problema. Todos os padrões tentam resolver algum tipo específico de problema, e é assim que eles são classificados.

Voltando ao livro da Gangue dos Quatro, os padrões são categorizados como de:

- **Criação** — Problemas que envolvem criar objetos;
- **Estruturais** — Problemas com a arquitetura da aplicação;
- **Comportamentais** — Problemas com o estado interno e o comportamento de objetos.

Por fim, mas não menos importante, o contexto do problema é o fator principal ao decidir aplicar ou não um padrão! Ao longo do livro, vamos explorar exemplos e discutir bastante o contexto da situação.

Aplicar um padrão simplesmente por aplicar pode resultar em um design muito mais complexo do que o necessário e montes de código que ninguém consegue entender sem ter de pular entre várias classes.

2.7 REFATORANDO COM PADRÕES

Como uma solução comum para um problema existente,

dentro de um contexto, pode ajudar a melhorar o design em pequenos passos, sem deixar o sistema quebrado?

Uma vantagem bem óbvia de utilizar padrões de projeto é que eles facilitam a comunicação. Se você e uma colega entendem o conceito de tipos e classes, por exemplo, não é necessário ficar explicando-os, pois isso faz parte do vocabulário de vocês. Além disso, provavelmente vai ser mais fácil entender um código ao perceber que ele usa um padrão conhecido.

Por serem soluções que já foram usadas várias e várias vezes, os padrões de projeto tendem a ser genéricos o suficiente para facilitar a acomodação de mudanças. Além disso, já foram validados por várias pessoas em vários projetos.

Ao longo da discussão sobre os padrões, será fácil ver que as soluções seguem os princípios de design orientado a objetos, como ter uma responsabilidade por classe e facilitar a extensão do comportamento.

Quando estou fazendo uma refatoração, geralmente sigo estes passos:

- **Identificar uma oportunidades de melhoria:** antipadrões, ou "maus cheiros" (do inglês *code smells*, como são mais conhecidos), são sintomas de que o seu código não está tão bom quanto poderia. Um clássico exemplo são as linhas de comentários explicando o que o código está fazendo. O já mencionado livro do Martin Fowler (1999), *Refactoring: improving the design of existing code*, apresenta um bom catálogo de antipadrões.

  Além dos antipadrões, também é possível melhorar o

código pensando em facilitar futuras alterações. Ao desenvolver um sistema com vários tipos de usuários (usuários comuns, administradores, suporte, gerentes etc.), é mais fácil modelar uma classe `Usuario` considerando todos esses possíveis subtipos desde o começo.

- **Entender o contexto do código:** como já foi falado antes, é muito importante levar em consideração o contexto ao decidir aplicar um padrão de projeto. Apesar de ser uma solução amplamente usada e testada, adicionar a estrutura para suportar um padrão também tem seus custos. Geralmente, são criadas abstrações que deixam o código mais espalhado entre classes.

  Ao imaginar o design ideal, procure sempre soluções mais simples antes de aplicar um padrão. O termo *YAGNI*, que em inglês quer dizer "você não vai precisar disso" (*You ain't gonna need it*), explica exatamente a situação na qual o código é desnecessariamente genérico.

- **Aplicar pequenas mudanças nos testes e códigos:** um ponto interessante no processo de refatoração é seguir a mentalidade do TDD (*Test Driven Development*) e fazer as alterações primeiro no teste. Apesar de não introduzir uma nova funcionalidade, refatorar o código pode causar mudanças nas interfaces dos componentes; portanto, modifique o teste para ver como o componente será usado antes de tomar a decisão final de aplicar ou não o padrão.

  Após a refatoração, o sistema deve estar funcional, com todos os testes passando. Aplicar pequenas mudanças ajuda a reduzir o tempo em que o sistema fica quebrado,

aumentando a confiança. Se o tempo entre testes de sucesso for muito grande, talvez seja melhor refatorar partes menores. Mas lembre-se de manter a visão do design ideal, para que não acabe com uma refatoração pela metade.

Capítulo 3

JAVA E O PARADIGMA ORIENTADO A OBJETOS

Nas próximas seções do livro, vamos aplicar alguns dos padrões descritos pela Gangue dos Quatros. Porém, antes de mergulhar neles, vamos avaliar algumas características da linguagem Java que impactam diretamente na utilização dos padrões.

3.1 PENSANDO ORIENTADO A OBJETOS

O paradigma orientado a objetos nos faz pensar no design da aplicação em termos de objetos trocando mensagens. Mas o que isso quer dizer? Quais são as motivações por trás desse modo de pensar? Como isso impacta nas implementações dos padrões de projeto?

Mensagens e estado

Alan Kay, criador da programação orientada a objetos, descreve a linguagem usando a metáfora de células biológicas: objetos se comunicariam utilizando mensagens, visando proteger e esconder seu estado interno (http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en).

Uma célula não altera a estrutura interna de outra, elas apenas trocam mensagens por meio de estímulos. Uma vez recebido o estímulo externo, o estado interno da célula sofrerá modificações. A definição de Orientação a Objetos é fortemente baseada nesses dois conceitos, **mensagens** e **estado**.

O estado de um objeto é basicamente composto por seus atributos, mas eles representam mais do que apenas os dados que um objeto possui. Como um sistema é orquestrado ao redor de objetos, os dados representam uma parte do mundo virtual que foi modelado.

Se em algum ponto os dados estiverem em um estado inconsistente (por não existir ou possuir um valor inesperado), o sistema todo pode estar em risco. Esse é o principal motivo para que o estado interno de um objeto seja escondido, e um dos grandes motivadores da abstração de dados em objetos.

Ter o seu estado interno escondido e protegido não quer dizer que mudanças não ocorram, apenas que elas devem ser centralizadas dentro dos objetos. Para acionar essas mudanças, dois objetos trocam mensagens por meio de uma interface. Essa interface é formada pelo conjunto de métodos do objeto, e representa o comportamento dos objetos.

Padrões orientados a objetos

Os padrões apresentados no livro *Design Patterns: elements of reusable object-oriented software* são implementados em duas linguagens: C++ e Smalltalk, ambas orientadas a objetos. No entanto, estas possuem características bem diferentes, e uma que merece atenção especial é a tipagem.

C++ é estaticamente tipado e requer declaração de tipo. Ao criar uma variável, é necessário dizer explicitamente qual o seu tipo, e este não pode ser alterado durante a execução do programa. Para atribuir um valor de tipo diferente a uma variável, é preciso convertê-lo primeiro, mesmo que ocorra perda de informação.

```
float d = 1.97f;
int i = (int) d; //converte para 1
```

Em linguagens com tipagem estática, um mecanismo de herança e de polimorfismo é necessário para facilitar a troca dos tipos e evitar perda de informação. O principal impacto da tipagem estática no uso de padrões é a necessidade de criar mais classes para que a linguagem permita a troca dos tipos.

No exemplo a seguir, `LivroPromocional` precisa herdar da classe `Livro`, e apenas os métodos definidos em `Livro` estarão disponíveis, mesmo que `livro` aponte para um `LivroPromocional`. Então, o método `calcularPrecoFinal` está definido em `Livro`, mas pode ser sobrescrito em `LivroPromocional`.

```
Livro* livro = new LivroPromocional();
p->calcularPrecoFinal();
```

Por outro lado, Smalltalk é dinamicamente tipado e não requer declaração de tipo, ou seja, não é preciso dizer qual o tipo do objeto que a variável vai referenciar. Qualquer tipo passado será atribuído sem problemas e, ao executar um método, o único requisito é que ele esteja declarado no tipo referenciado pela variável.

As diferenças na tipagem tornam as duas linguagens bem singulares, mas ainda assim ambas são orientadas a objetos. Então,

ambas podem se beneficiar da aplicação de padrões, apenas a implementação será um pouco diferente.

3.2 CARACTERÍSTICAS DO JAVA

Assim como em C++, Java é estaticamente tipado, e o compilador faz as validações de tipos antes mesmo do programa executar. Isso dá mais segurança de que o programa não encontrará erros ao ser executado, mas exige mais código para obter uma boa flexibilidade.

Vamos avaliar agora algumas características da linguagem que serão importantes ao desenharmos soluções com padrões de projeto.

Nem tudo é um objeto

Uma característica curiosa de Java é que, apesar de ser uma linguagem orientada a objetos, nem tudo é representado como objeto. Um bom exemplo são os tipos primitivos, como o `int`, que, apesar de possuírem uma representação como objetos do tipo `Integer`, não possuem métodos que podem ser chamados.

```
1.toString() // erro de compilação pois 1 é um tipo primitivo
new Integer(1).toString() // objetos Integer possuem métodos
```

Na prática (e ao longo do livro), dificilmente isso causará problemas, mas vale a pena saber que precisamos lidar com tipos primitivos de maneira diferente de outros objetos.

Polimorfismo

Todo objeto em Java herda da classe `java.lang.Object`,

assim `Object` é considerado o tipo mais genérico. Ser "genérico" quer dizer, de maneira simplificada, que uma variável do tipo `Object` pode receber qualquer outro objeto.

```
Object variavelGenerica = new String();
variavelGenerica = new Integer();
variavelGenerica = new MinhaClasse();
```

Essa ideia de utilizar um objeto de um tipo genérico e poder mudar sua implementação por outros objetos é chamada de polimorfismo. Um bom exemplo é a maneira como o Java lida com lista de dados. A classe `List` é a classe base para diversas implementações, e nela estão definidas operações básicas como `get` e `add`. Cada implementação de `List`, por exemplo `ArrayList` e `LinkedList`, pode definir seus detalhes sem que uma variável do tipo `List` sofra alterações.

```
List lista = new ArrayList<String>();
lista.add("uma string");
lista.get(0);
lista = new LinkedList<Integer>();
lista.add("uma string");
lista.get(0);
```

Utilizamos polimorfismo em alguns padrões de projeto para garantir que implementações diferentes possam ser usadas sem que o código que as utiliza precise mudar.

Sobrecarga e sobrescrita de métodos

Uma das maneiras mais básicas de alcançar o polimorfismo é sobrescrever ou sobrecarregar métodos. Ao sobrescrever um método de uma classe, estamos eliminando completamente seu código; já ao sobrecarregá-lo, vamos adicionar mais regras à lógica que já existia. Um bom exemplo é quando implementamos novas

classes e podemos modificar alguns comportamentos de `Object` (lembre-se de que `Object` é a classe base para todas as outras).

```
public class Estudante {
    private String serie;

    private String nome;

    @Override
    public String toString() {
        return "Nome: " + nome + "\nSerie: " + serie;
    }
}
```

O método `toString` é definido em `Object` e é usado sempre que vamos converter um objeto em `String` (por exemplo, ao imprimi-lo no console), então podemos sobrescrevê-lo para exibir uma mensagem mais amigável. Utilizamos a *annotation* `@Override` para indicar que o método `toString` não está definido nessa classe, mas mesmo assim estamos redefinindo-o.

No código anterior, apagamos completamente o comportamento definido em `Object` . Porém, se precisarmos, podemos manter o antigo comportamento ainda presente usando o `super` :

```
public class Estudante {
    private String serie;

    private String nome;

    @Override
    public String toString() {
        String idDoObjeto = super.toString();
        return "Nome: " + nome + "\nSerie: " + serie +
            "\nId do Objeto: " + idDoObjeto;
    }
}
```

A chamada `super.toString()` vai executar a lógica original, que no caso retorna o *id* do objeto. Assim podemos manter a execução da lógica original e adicionar o que precisamos a ela.

Interfaces e classes abstratas

Outra maneira de polimorfismo que podemos usar é definir apenas um "contrato", sem nenhuma lógica, utilizando interfaces e classes abstratas. A principal diferença entre elas é que uma interface não possui lógica, enquanto uma classe abstrata pode implementar métodos.

Por exemplo, para desenvolver um jogo, podemos definir uma interface para representar o que uma arma precisa implementar:

```
public interface Arma {
    int getDano();

    int getBonusVelocidade();
}
```

A interface por si só não define nenhum comportamento, é preciso criar uma classe e fazê-la implementar a interface:

```
public class Adaga implements Arma {
    @Override
    public int getDano() {
        return 10;
    }

    @Override
    public int getBonusVelocidade() {
        return 3;
    }
}
```

Se tivermos métodos que são comuns a todas as classes que implementam `Arma` — por exemplo, um método que sempre

retorna metade do valor do dano —, podemos convertê-la em uma classe abstrata e evitar a repetição:

```
public abstract class Arma {
    public abstract int getDano();

    public abstract int getBonusVelocidade();

    public int danoComArmaQuebrada() {
        return getDano()/2;
    }
}
```

Agora qualquer classe que implemente `Arma` ganha o método `danoComArmaQuebrada` . Classes abstratas e interfaces serão bem úteis quando estivermos trabalhando com a implementação dos padrões, pois elas permitem que o código seja mais flexível.

Padrões comuns

Capítulo 4

FACTORY: GERENCIANDO A CRIAÇÃO DE OBJETOS

Rafael é um desenvolvedor com bastante experiência, não apenas com Java. No entanto, ele sabe ser pragmático e resolver problemas em vez de procurar a solução perfeita para cada situação. Além de ser um bom desenvolvedor, ele também tem o espírito empreendedor e está sempre em busca de uma ideia para seguir!

Em meio a tantos sites de compras com modelos iguais, Rafael e alguns amigos estão criando uma nova aplicação que promete ser diferente de todas as outras e revolucionar a maneira como compras são feitas. As expectativas estão altas, e a equipe quer buscar investidores para ajudar a fazer a ideia ficar popular.

Após uma pesquisa de mercado, Rafael nota que as características mais importantes para um site de compras é uma ferramenta de busca. Usuários querem encontrar os produtos o mais rápido possível. Esse é exatamente um dos pontos fortes da aplicação de Rafael que ele planeja demonstrar em uma reunião com potenciais investidores.

Durante a reunião, tudo vai muito bem e o grupo de

investidores fica maravilhado com a nova ferramenta, decidindo investir nela. Com toda a animação após a grande conquista, Rafael se junta à sua equipe, e decide que é hora de fazer algumas melhorias no produto para compensar as decisões tomadas na correria dos últimos dias. O primeiro alvo é a parte que prepara as buscas.

4.1 O CUSTO DA FLEXIBILIDADE

Como uma das principais funcionalidades são as várias buscas, o código que lida com elas precisa ser bem flexível para acomodar as necessidades de cada um dos usuários. Para essa primeira demonstração com os potenciais investidores, Rafael preparou três diferentes tipos de buscas: normal, por categoria e promocional.

É possível especificar em uma busca:

- A quantidade de produtos exibidos por página, sendo 15 por padrão;
- A categoria de produtos;
- A ordem de exibição dos produtos.

Na busca normal, apenas o nome do produto precisa ser especificado. Caso nenhuma opção de ordem de exibição seja usada, o resultado será ordenado por "relevância".

Já na busca por categoria, é necessário especificar qual categoria se está buscando; caso contrário, ela volta a ser uma busca normal. A ordem de exibição dos resultados deve ser por "mais recente", a menos que um valor seja especificado.

Por fim, na busca promocional, a categoria será sempre "em

promoção" e o resultado sempre será ordenado por "mais recente", não importando os outros valores.

Rafael ficou responsável por fazer a parte de busca da aplicação. O requisito básico é receber um objeto `ParametrosDeBusca` com informações sobre a busca a ser realizada e, então, executá-la. O objeto `ParametrosDeBusca` possui os seguintes atributos:

- `tipoDeBusca` — Indica qual tipo de busca foi selecionado pelo usuário;
- `resultadosPorPagina` — Indica quantos produtos por página devem ser exibidos;
- `categoria` — Define que tipo de produto deve ser pesquisado;
- `ordernarPor` — Diz qual o critério de ordenação escolhido pelo usuário;

A classe `ParametrosDeBusca` define alguns valores padrões para esses atributos, que serão utilizados para criar buscas normais.

```
// /src/main/java/factory/ParametrosDeBusca.java
public class ParametrosDeBusca {
    private int resultadosPorPagina = 15;
    private Categoria categoria = Categoria.TUDO;
    private TipoDeBusca tipoDeBusca = TipoDeBusca.NORMAL;
    private OrdenarPor ordernarPor = OrdenarPor.RELEVANCIA;
}
```

Para representar a categoria, o tipo de busca e a forma de ordenação, foram criados `Enum` s, por exemplo:

```
// /src/main/java/factory/OrdenarPor.java
public enum OrdenarPor {
    RECENTE, PRECO, RELEVANCIA
}
```

Rafael decidiu por criar uma classe `Busca` como ponto de entrada, recebendo o objeto `ParametrosDeBusca` e, a partir dele, criar um objeto `CriterioDeBusca`, que vai basicamente conter as informações sobre a busca, como: ordem, quantidade de produtos por página etc. Com o critério de busca definido, a classe `ServicoDeBusca` será usada para fazer a busca de fato, retornando os `ids` dos itens correspondentes. O método `por` vai realizar a busca:

```
// /src/main/java/factory/Busca.java
public class Busca {
    private ServicoDeBusca servicoDeBusca;

    public Busca(ServicoDeBusca servicoDeBusca) {
        this.servicoDeBusca = servicoDeBusca;
    }

    public void por(ParametrosDeBusca parametros) {
        CriterioDeBusca criterio = criarCriterio(parametros);
        List<String> idsDeResultado =
            servicoDeBusca.realizarBuscaCom(criterio);
        encontrarProdutosPorIds(idsDeResultado);
    }
}
```

Ao olhar para o código novamente, Rafael nota que esse método de `Busca` ficou bem simples, curto e com nomes legíveis. No entanto, o real problema está no método que cria o `CriterioDeBusca` a partir dos parâmetros.

Devido à pressa para a grande reunião, seu código ficou bem "feio", com vários `ifs` e bem difícil de manter. Rafael olha para o código e imagina o quão difícil seria mudar um tipo de busca já existente, ou até mesmo incluir um novo.

```
// /src/main/java/factory/Busca.java
public class Busca {
```

```
  CriterioDeBusca criarCriterio(ParametrosDeBusca parametros) {
    CriterioDeBusca criterio = new CriterioDeBusca();
    criterio.setPaginacao(parametros.getResultadosPorPagina());
    criterio.setCategoria(parametros.getCategoria());

    TipoDeBusca busca = parametros.getTipoDeBusca();
    if (busca.equals(TipoDeBusca.PROMOCIONAL)) {
      // Busca promocional ignora parâmetros de busca
      criterio.setCategoria(Categoria.EM_PROMOCAO);
      criterio.setOrdernarPor(OrdenarPor.RECENTE);
    } else if (busca.equals(TipoDeBusca.POR_CATEGORIA)) {
      criterio.setCategoria(parametros.getCategoria());
      if (busca.equals(Categoria.TUDO)) {
        // Se Categoria não for especificada volta para busca nor
mal
        criterio.setOrdernarPor(OrdenarPor.RELEVANCIA);
      } else {
        // Se tiver categoria, ordena por mais recente
        criterio.setOrdernarPor(parametros.getOrdernarPor());
      }
    } else { //Busca normal
      criterio.setOrdernarPor(parametros.getOrdernarPor());
    }

    return criterio;
  }
}
```

Apesar de perceber que o código ficou feio, Rafael lembra de que também desenvolveu testes para garantir que tudo está funcionando como esperado, e fica mais aliviado. Um exemplo de teste é criar um conjunto específico de parâmetros e validar o critério que foi construído a partir deles. Quando buscamos apenas pelo nome do produto, os outros atributos devem assumir valores padrão:

```
// /src/test/java/factory/BuscaTest.java
public class BuscaTest {

    @Test
    public void criaCriterioDeBuscaNormal() {
        ServicoDeBusca servico = mock(ServicoDeBusca.class);
```

```
        Busca busca = new Busca(servico);
        ParametrosDeBusca parametros = new ParametrosDeBusca();
        CriterioDeBusca criterio = busca.criarCriterio(parametros
);

        assertEquals(criterio.getPaginacao(), 15);
        assertEquals(criterio.getOrdernarPor(), OrdenarPor.RELEVA
NCIA);
        assertEquals(criterio.getCategoria(), Categoria.TUDO);
    }
}
```

Para cada um dos casos de busca descritos anteriormente, foram criados testes apropriados, garantindo que todos eles estejam cobertos e funcionando. O teste a seguir garante que, ao especificarmos uma quantidade de resultados por página, o critério seja construído apropriadamente:

```
// /src/test/java/factory/BuscaTest.java
public class BuscaTest {

    @Test
    public void criaCriterioDeBuscaNormalComPaginacao() {
        ServicoDeBusca servico = new ServicoDeBusca();

        Busca busca = new Busca(servico);
        ParametrosDeBusca parametros = new ParametrosDeBusca();
        parametros.setResultadosPorPagina(20);
        CriterioDeBusca criterio = busca.criarCriterio(parametros
);

        assertEquals(criterio.getPaginacao(), 20);
    }
}
```

Ao pensar em refatorar o código, Rafael tem a ideia de extrair a lógica de criação de buscas para outra classe, deixando para `Busca` apenas a responsabilidade de chamar o serviço de busca para mapear os `ids` dos produtos. Além de criar outra classe, ele

também gostaria que cada tipo de busca pudesse ser criado em seu método próprio, separando melhor os vários `ifs` do código.

Extrair a lógica de criação de objetos para classes e métodos específicos é o objetivo dos padrões *Factory*: Simple Factory, Factory Method e Abstract Factory.

4.2 OS PADRÕES FACTORY

O problema que esses padrões resolvem é criar instâncias de objetos separadas do resto da lógica; por isso são classificados como padrões de criação. Entretanto, Simple Factory, Factory Method e Abstract Factory diferem bastante na solução para o problema, o que torna a escolha de qual usar muito dependente do contexto em que serão usados.

Uma nomenclatura comum ao falar sobre padrões *Factory* é chamar os objetos criados de **produtos**, e as classes que os criam são as **fábricas** — utilizando a metáfora de que fábricas fazem produtos.

Vamos começar com o Simple Factory, que é o mais simples e parece ser exatamente a refatoração de que Rafael precisa!

Simple Factory

O contexto de utilização do Simple Factory é quando você possui apenas um tipo produto para ser criado e existe apenas uma maneira de fazê-lo, ou seja, apenas uma fábrica. A solução é criar uma classe para extrair o comportamento de criação de objetos.

Para o problema de Rafael, no qual ele quer separar a lógica

que cria os objetos de busca da qual os utiliza, esse é exatamente o contexto. O único objeto que precisa ser criado é o `CriterioDeBusca` e, apesar das diferentes configurações, todos são criados da mesma maneira. A solução parece que ficará bem simples e ideal.

Vamos criar a classe `FabricaDeCriterio`, na qual serão instanciados objetos `CriterioDeBusca` com os valores apropriados. Para cada um dos tipos de busca, será criado um método que vai conter as regras específicas deles. Essa refatoração é conhecida por **Extrair Classe**, que tem como principal objetivo separar responsabilidades em duas classes diferentes.

No nosso caso, a classe `Busca` configura um critério e realiza a busca, assim, vamos separar a responsabilidade de criar/configurar um critério em `FabricaDeCriterio`. Como um exemplo, veja como ficaria a criação de buscas promocionais:

```
// /src/main/java/factory/FabricaDeCriterio.java
public class FabricaDeCriterio {
    private final ParametrosDeBusca parametros;

    public FabricaDeCriterio(ParametrosDeBusca parametros) {
        this.parametros = parametros;
    }

    public CriterioDeBusca criterioNormal() {
        CriterioDeBusca criterio = new CriterioDeBusca();
        criterio.setPaginacao(parametros.getResultadosPorPagina()
);
        criterio.setCategoria(parametros.getCategoria());
        criterio.setOrdernarPor(parametros.getOrdernarPor());
        return criterio;
    }
}
```

Agora, na classe `Busca`, basta acionar o método apropriado

da `FabricaDeCriterio` para criar o `CriterioDeBusca`.

```
// /src/main/java/factory/Busca.java
public class Busca {

    CriterioDeBusca criarCriterio(ParametrosDeBusca parametros) {
        CriterioDeBusca criterio;
        TipoDeBusca busca = parametros.getTipoDeBusca();
        if (busca.equals(TipoDeBusca.PROMOCIONAL)) {
            criterio = new FabricaDeCriterio(parametros)
                .criterioPromocional();
        } else if (busca.equals(TipoDeBusca.POR_CATEGORIA)) {
            criterio = new FabricaDeCriterio(parametros)
                .criterioPorCategoria();
        } else {
            criterio = new FabricaDeCriterio(parametros)
                .criterioNormal();
        }

        return criterio;
    }
}
```

A próxima refatoração é mover o método `criarCriterio` para a `FabricaDeCriterio`, utilizando **Mover Método**, para separar completamente as responsabilidades.

```
// /src/main/java/factory/FabricaDeCriterio.java
public class FabricaDeCriterio {
    public CriterioDeBusca criarCriterio() {
        CriterioDeBusca criterio;
        TipoDeBusca busca = parametros.getTipoDeBusca();
        if (busca.equals(TipoDeBusca.PROMOCIONAL)) {
            criterio = new FabricaDeCriterio(parametros)
                .criterioPromocional();
        } else if (busca.equals(TipoDeBusca.POR_CATEGORIA)) {
            criterio = new FabricaDeCriterio(parametros)
                .criterioPorCategoria();
        } else {
            criterio = new FabricaDeCriterio(parametros)
                .criterioNormal();
        }
```

```
        return criterio;
    }
}
```

Agora que a lógica para criar critérios está definida na classe `FabricaDeCriterio`, podemos apenas mover os testes da classe `Busca` e usar a nova estrutura.

```
// /src/test/java/factory/FabricaDeCriterioTest.java
public class FabricaDeCriterioTest {

    @Test
    public void criaCriterioDeBuscaNormal() {
        ParametrosDeBusca parametros =
            new ParametrosDeBusca(TipoDeBusca.NORMAL);
        CriterioDeBusca criterio =
            new FabricaDeCriterio(parametros).criarCriterio();

        assertEquals(criterio.getPaginacao(), 15);
        assertEquals(criterio.getOrdernarPor(),
            OrdenarPor.NAO_ESPECIFICADO);
        assertEquals(criterio.getCategoria(), Categoria.TUDO);
    }
}
```

Agora, nos testes da classe `Busca`, vamos apenas garantir que, ao executar o método `por`, o `ServicoDeBusca.realizarBuscaCom` será executado com o `CriterioDeBusca` esperado. Para isso, criaremos um *mock* do serviço e validaremos que o método foi chamado com o parâmetro esperado.

```
// /src/test/java/factory/BuscaTest.java
public class BuscaTest {

    @Test
    public void realizarBusca() {
        ServicoDeBusca servico = mock(ServicoDeBusca.class);

        Busca busca = new Busca(servico);
        ParametrosDeBusca parametros =
```

```
            new ParametrosDeBusca(TipoDeBusca.NORMAL);
        busca.por(parametros);

        CriterioDeBusca criterio =
            new FabricaDeCriterio(parametros).criarCriterio();

        verify(servico).realizarBuscaCom(criterio);
    }
}
```

Por fim, podemos usar a classe `FabricaDeCriterio` diretamente no método `Busca.por`:

```
// /src/main/java/factory/Busca.java
public class Busca {

    public void por(ParametrosDeBusca parametros) {
        CriterioDeBusca criterio =
            new FabricaDeCriterio(parametros).criarCriterio();
        List<String> idsDeResultado =
            servicoDeBusca.realizarBuscaCom(criterio);
        encontrarProdutosPorIds(idsDeResultado);
    }
}
```

No final, a classe `Busca` ficou bem mais simples e com menos responsabilidades. O Simple Factory não é tão complicado, é apenas uma nova classe com alguns métodos especializados em criar outros objetos.

Na verdade, o Simple Factory nem é considerado um padrão por alguns autores, mas sim um idioma de linguagem. Veja o quão simples é o diagrama de uso:

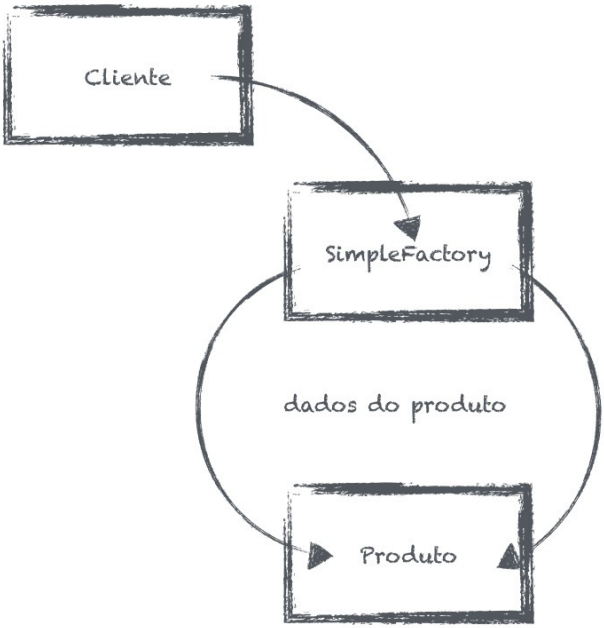

Figura 4.1: Funcionamento do Simple Factory

Apesar de simples, ele resolve o problema sem criar uma estrutura de classes muito grande, o que agrada muito um desenvolvedor pragmático como Rafael. Porém, caso a quantidade de métodos implementados na classe fábrica comece a aumentar demais, talvez seja um indicador de que é melhor criar novas fábricas, ou até mesmo avaliar a utilização de outro padrão.

Factory Method

O contexto do Factory Method é quando queremos criar um mesmo tipo de produto, mas com diversas fábricas, cada uma funcionando de uma maneira própria. Assim, além das configurações passadas, cada fábrica possui uma lógica específica

que afeta o produto final.

No exemplo de Rafael, não é necessário se preocupar com maneiras de criar os objetos, pois todos serão criados do mesmo jeito. Portanto, usar o Factory Method nesse caso seria mais complicado que o Simple Factory, e não traria nenhum outro benefício.

No entanto, imagine que, para oferecer mais tipos diferentes de busca, ele deseja utilizar múltiplas *engines*, como ElasticSearch, Solr ou uma busca comum no banco de dados. Além das configurações do objeto de busca `CriterioDeBusca`, cada uma das *engines* precisaria de configurações próprias, como o nome do `host`, a consulta formatada, configurações de limite de tempo etc.

O código simplificado a seguir mostra a implementação para a situação anterior. O objeto `ParametrosDeBusca` contém qual ferramenta de busca deve ser usada para a pesquisa e, a partir dela, decidimos qual *engine* usar:

```
if (parametros.getEngine().equals(Engine.ELASTIC_SEARCH)) {
  TipoDeBusca busca = parametros.getTipoDeBusca();
  if (busca.equals(TipoDeBusca.PROMOCIONAL)) {
    // código para tratar busca promocional
  } else if (busca.equals(TipoDeBusca.POR_CATEGORIA)) {
    // código para tratar busca por categoria
  } else {
    // código para tratar busca normal
  }
  // configuração elasticsearch
} else if (parametros.getEngine().equals(Engine.SOLR)) {
  // mesmos ifs anteriores
  // configuração Solr
} else if (parametros.getEngine().equals(Engine.BANCO)) {
  // mesmos ifs anteriores
  // configuração banco de dados
}
```

Aplicar o Factory Method ajuda da seguinte maneira: para cada *engine*, é criada uma classe fábrica que sabe apenas sobre suas próprias configurações, e todas as classes fábricas criam os mesmos objetos de busca. Assim, o mesmo código que usa a busca no banco de dados pode também utilizar o ElasticSearch, sem grandes alterações.

A classe `FabricaElasticsearch` terá apenas a configuração relacionada à *engine* `ElasticSearch` e à lógica de como os objetos `CriterioDeBusca` devem ser criados para que a busca seja feita corretamente.

Ao aplicar o Factory Method, cada um dos `ifs` externos que verifica a ferramenta de busca seria extraído para uma classe fábrica, e cada um dos `if` internos seria extraído para um método. Por exemplo, para criar buscas com ElasticSearch, basta utilizar a `FabricaElasticsearch`:

```
class FabricaElasticsearch {
    FabricaElasticsearch(ParametrosDeBusca parametros) {
        this.host = "elasticsearch.myhost.com"
        this.porta = "4004"
        this.parametros = parametros
    }

    private CriterioDeBusca criterioPorCategoria() { }

    private CriterioDeBusca criterioPromocional() { }

    private CriterioDeBusca criterioNormal() { }
}
```

O diagrama de utilização do Factory Method é simples, a única diferença em relação ao Simple Factory é que vamos ver mais de uma classe fábrica criando os produtos:

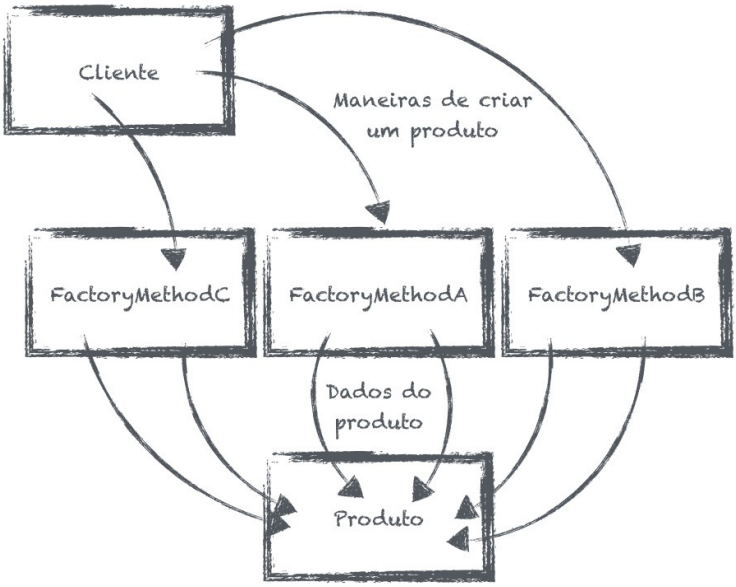

Figura 4.2: Funcionamento do Factory Method

A classe que continha a lógica maluca para criar buscas agora vai usar os seguintes passos:

1. Instanciar uma nova fábrica, ou recebê-la como parâmetro;
2. Chamar o método específico para criar o produto;
3. A fábrica vai executar sua lógica e retornar o produto.

Inversão de dependências

Ao utilizar o Factory Method, podemos receber a fábrica diretamente em vez de decidir, a partir de um parâmetro, qual deve ser instanciada. Dessa forma, vamos apenas usar a fábrica que foi passada e chamar o método para criar o produto diretamente. Para isso, precisamos criar uma interface comum para que todas as fábricas implementem o método `criarCriterio` .

```
public interface FabricaDeCriterio {
  public CriterioDeBusca criarCriterio();
}
```

Agora basta utilizar essa interface em vez das classes concretas, para que o código fique mais simples ainda.

```
public void por(FabricaDeCriterio fabrica) {
  CriterioDeBusca criterio = fabrica.criarCriterio()
}
```

Essa inversão de pensamento é conhecida como o **Princípio da Inversão de Dependência**, que sugere depender de classes abstratas, e não de classes concretas. Assim, a lógica de negócio não depende de qual classe será passada como parâmetro.

Abstract Factory

Para o Abstract Factory, o contexto de utilização é bem mais complexo do que o Factory Method. Seguindo a analogia das fábricas, existirão múltiplas fábricas que criam múltiplos tipos de

produtos.

Como um exemplo mais concreto para usar o Abstract Factory, imagine que você está trabalhando em um framework no qual escrevemos um aplicativo em Java e ele vai ser convertido para múltiplas plataformas, como Android, iOS e HTML.

Ao desenvolver uma aplicação nesse framework, não precisamos especificar qual plataforma será usada, então, podemos usar uma fábrica para isso. Precisamos que as fábricas criem vários elementos de interface, como botões, *inputs*, *dropdowns* etc.

Logo, uma `FabricaHTML` definiria a lógica específica para criar um botão em HTML, enquanto a `FabricaAndroid` vai definir a lógica para criar o botão em Java etc. Podemos definir uma família de produtos (por exemplo, botões e *dropdowns*) que será criada pelas fábricas (por exemplo, Android e HTML).

O diagrama de uso do Abstract Factory é bem mais complexo do que os anteriores:

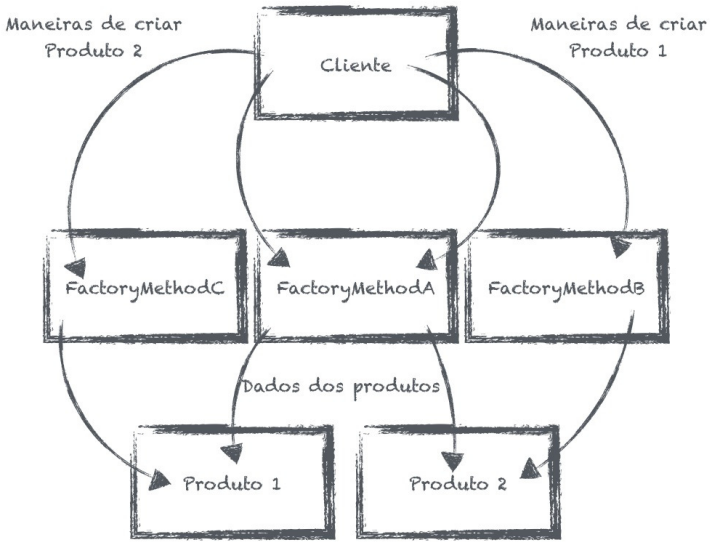

Figura 4.3: Funcionamento do Abstract Factory

A não ser que seu projeto tenha múltiplas dependências, como nesse exemplo, utilizar o Simple Factory vai resolver o problema sem a necessidade de criar várias camadas.

Capítulo 5

STRATEGY: DIVIDIR PARA SIMPLIFICAR

Paula sempre foi fã de redes sociais e possui uma conta em praticamente todas que você possa imaginar. Então, ao desenvolver o **EuS2Livros**, ela juntou suas grandes paixões: livros, redes sociais e programação!

O EuS2Livros é uma nova rede social para juntar escritores e pessoas apaixonadas por livros, aproximando mais os autores de seu público. O projeto parece bem promissor, e ela e seu grupo estão animados com as possibilidades de crescimento do negócio. Como última etapa do desenvolvimento, eles querem facilitar a entrada de novos usuários, permitindo-lhes utilizarem suas contas já existentes em outras redes sociais.

A arquitetura e o design do aplicativo foram muito bem desenvolvidos, e o módulo de login pode ser facilmente acoplado ao resto da aplicação. Tudo o que eles precisam fazer é validar o usuário, usando a rede social de sua escolha, e devolver o status (autenticado ou não) e uma mensagem que será exibida em caso de erros.

5.1 UM LOGIN COM VÁRIOS PROVEDORES

Suponha que você tem uma conta na rede social FaceNote e, com ela, você pode fazer o login no EuS2Livros sem precisar criar um usuário e uma senha. A aplicação pede permissão para o FaceNote enviando seus dados de usuário, assim, o FaceNote valida as suas informações e diz se você permitiu ou não o acesso ao EuS2Livros.

Paula ficou responsável por essa parte do sistema e decidiu criar classes de serviço para cada rede social, enviando os dados do usuário por meio de sua API específica. Os serviços recebem o ID do usuário (por exemplo, `"paulaS2livros"`) que será usado para autenticar e retornam um `int` que contém o status da chamada. Um exemplo é o serviço que faz login via FaceNote, que foi implementado assim:

```
// /src/main/java/strategy/ServicoFaceNoteLogin.java
public class ServicoFaceNoteLogin {

    public int autenticar(String idUsuario) {
        try {
            return autenticarViaPost(idUsuario);
        } catch (TimeoutException e) {
            logger.log(Level.SEVERE, e.getMessage());
        }
        return 500;
    }
}
```

Inicialmente, Paula criou o `ServicoFaceNoteLogin` para fazer login via FaceNote, e `ServicoZuiterLogin` para fazer login via Zuiter, outra famosa rede social. Sua ideia ao criar as classes de serviço é garantir que elas possam evoluir separadamente, conforme necessário, já que cada um dos serviços possui suas particularidades:

- **API do FaceNote:**

    - Login com sucesso — 200
    - Aplicação com acesso revocado — 403
    - Aplicação bloqueada — 408

- **API do Zuiter:**

    - Login com sucesso — 202
    - Autorização pendente — 400

Estes códigos de resposta serão utilizados pela classe `Login` para identificar o status da autorização. Paula adicionou várias constantes dentro da classe para evitar os números mágicos no meio do código, e deixá-lo mais legível.

```
// /src/main/java/strategy/Login.java
public class Login {
    private static int FACE_NOTE_SUCESSO = 200;
    private static int FACE_NOTE_REVOCADO = 403;
    private static int FACE_NOTE_BLOQUEADO = 408;

    private static int ZUITER_SUCESSO = 202;
    private static int ZUITER_PENDENTE = 400;
}
```

O código da classe `Login` recebe um objeto `DadosDeLogin`, que contém o método de login que deve ser usado junto aos dados do usuário. Depois, chama os serviços passando os dados e confere o código de retorno para saber se o usuário conseguiu ou não fazer o login. Ao final, retorna um objeto `RespostaLogin` que contém o status da autenticação e uma mensagem com mais detalhes.

```
// /src/main/java/strategy/Login.java
public class Login {

    public RespostaLogin com(DadosDeLogin dadosDeLogin) {
        int resposta = METODO_INVALIDO;
        Autenticacao metodo = dadosDeLogin.getMetodo();
        String usuario = dadosDeLogin.getUsuario();
```

```
        if (metodo.equals(Autenticacao.VIA_FACENOTE)) {
            resposta = servicoFaceNote.autenticar(usuario);
        } else if (metodo.equals(Autenticacao.VIA_ZUITER)) {
            resposta = servicoZuiter.autenticar(usuario);
        }

        String mensagem = "não foi possível autenticar";
        boolean status = false;

        if (resposta == FACE_NOTE_SUCESSO ||
                resposta == ZUITER_SUCESSO) {
            status = true;
            mensagem = "login com sucesso";
        } else if (resposta == FACE_NOTE_REVOCADO) {
            mensagem = "acesso revocado";
        } else if (resposta == FACE_NOTE_BLOQUEADO) {
            mensagem = "acesso bloqueado";
        } else if (resposta == ZUITER_PENDENTE) {
            mensagem = "acesso pendente";
        }

        return new RespostaLogin(mensagem, status);
    }
}
```

No final, essa é a maneira como a aplicação vai se comunicar com a API do FaceNote para permitir ou não o acesso de usuários:

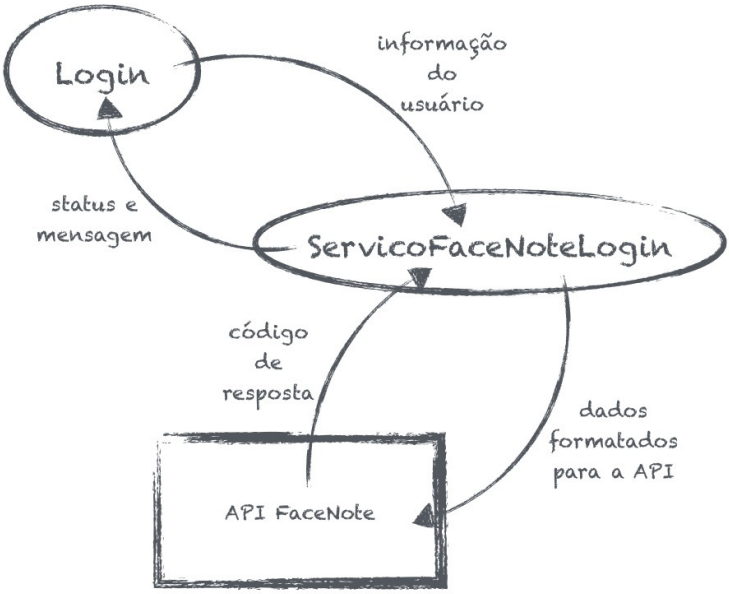

Figura 5.1: Fluxo de chamada para API do FaceNote

Como os testes precisam ser executados frequentemente, Paula também implementou dublês de teste do tipo *fake*. Dublês de teste ajudam a reduzir o custo dos testes, ignorando comportamentos desnecessários. Um *fake*, por exemplo, sobrecarrega o comportamento original da classe, e possibilita controlar diretamente o retorno dos métodos. Assim, Paula consegue evitar a chamada ao serviço externo durante os testes.

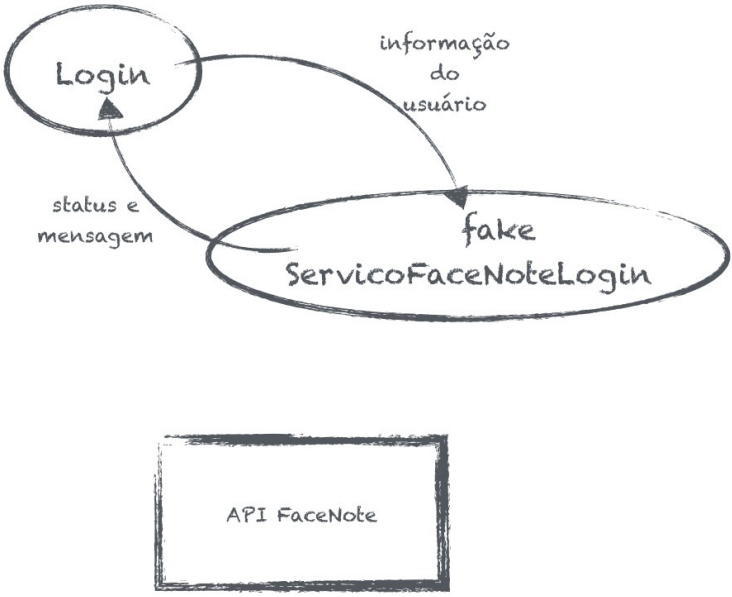

Figura 5.2: Fluxo de chamada para API do FaceNote utilizando fake

Por exemplo, será retornado 202 sempre que o nome do usuário passado como parâmetro for `"Gil"`, e `404` para qualquer outro usuário.

```
// /src/test/java/strategy/ServicoFaceNoteLoginFake.java
class ServicoFaceNoteLoginFake extends ServicoFaceNoteLogin {
    public int autenticar(String idUsuario) {
        if (idUsuario.equals("Gil")) {
            return 200;
        }
        return 404;
    }
}
```

Um exemplo de teste que Paula escreveu para essa classe é: criar o objeto `DadosDeLogin` com dados falsos, chamar `Login.com` passando esses dados, e verificar o status e a

mensagem que são retornados. Usando o `ServicoFaceNoteLoginFake`, basta configurar o ID do usuário como `"Gil"`, que o serviço vai realizar a autenticação com sucesso sempre.

```
// /src/test/java/strategy/LoginTest.java
public class LoginTest {
    @Test
    public void autenticaViaFaceNote() throws Exception {
        ServicoFaceNoteLogin servicoFaceNote =
            new ServicoFaceNoteLoginFake();
        ServicoZuiterLogin servicoZuiter =
            new ServicoZuiterLoginFake();
        Login login = new Login(servicoFaceNote, servicoZuiter);

        String usuario = "Gil";
        DadosDeLogin dados =
            new DadosDeLogin(usuario, Autenticacao.VIA_FACENOTE);

        RespostaLogin respostaLogin = login.com(dados);
        String mensagemSucesso = "login com sucesso";

        assertEquals(true, respostaLogin.getStatus());
        assertEquals(mensagemSucesso, respostaLogin.getMessage())
;
    }
}
```

Para o caso de teste quando a autenticação é revocada pelo FaceNote, Paula configura o *fake* para que, quando o ID do usuário for `"Ana"`, seja retornado `403`.

```
// /src/test/java/strategy/LoginTest.java
public class LoginTest {

    @Test
    public void falhaComAcessoRevocado() throws Exception {
        ServicoFaceNoteLogin servicoFaceNote =
            new ServicoFaceNoteLoginFake();
        ServicoZuiterLogin servicoZuiter =
            new ServicoZuiterLoginFake();
        Login login = new Login(servicoFaceNote, servicoZuiter);
```

```
        String usuario = "Ana";
        DadosDeLogin dados =
            new DadosDeLogin(usuario, Autenticacao.VIA_FACENOTE);

        RespostaLogin respostaLogin = login.com(dados);
        String mensagemErro = "acesso revocado";

        assertEquals(false, respostaLogin.getStatus());
        assertEquals(mensagemErro, respostaLogin.getMessage());
    }
}
```

Agora basta alterar o `ServicoFaceNoteLoginFake` para responder com o código de acesso revocado quando o ID do usuário usado for `"Ana"` .

```
// /src/test/java/strategy/ServicoFaceNoteLoginFake.java
class ServicoFaceNoteLoginFake extends ServicoFaceNoteLogin {
    public int autenticar(String idUsuario) {
        if (idUsuario.equals("Gil")) {
            return 200;
        } else if(idUsuario.equals("Ana")) {
            return 403;
        }
        return 404;
    }
}
```

Com essa primeira versão terminada e bem testada, o time agora vai procurar por oportunidades de refatoração. O primeiro mau cheiro levantado é que o método `Login.com` já está bem grande e vai crescer mais ainda, conforme novas redes sociais forem adicionadas.

Paula olha o código mais um pouco e sugere separar as lógicas de login específicas de cada provedor, extraindo-as para classes próprias e criando diferentes estratégias de login. Para isso, ela sugere utilizar o padrão Strategy.

5.2 O PADRÃO STRATEGY

Por ser classificado como um padrão de comportamento, o padrão Strategy resolve problemas de distribuição de responsabilidades. Para sua aplicação, é necessário que exista uma maneira clara de separar essas responsabilidades em algoritmos próprios. A solução proposta é encapsular esses algoritmos nas estratégias, de maneira que trocá-las seja bem fácil.

Cada uma das maneiras de fazer login pode ser considerada uma estratégia e, em vez da classe `Login` conter todas as lógicas, ela vai apenas delegar a chamada de acordo com os parâmetros recebidos. Dessa forma, a lógica específica de uma API fica autocontida na sua classe, centralizando mudanças.

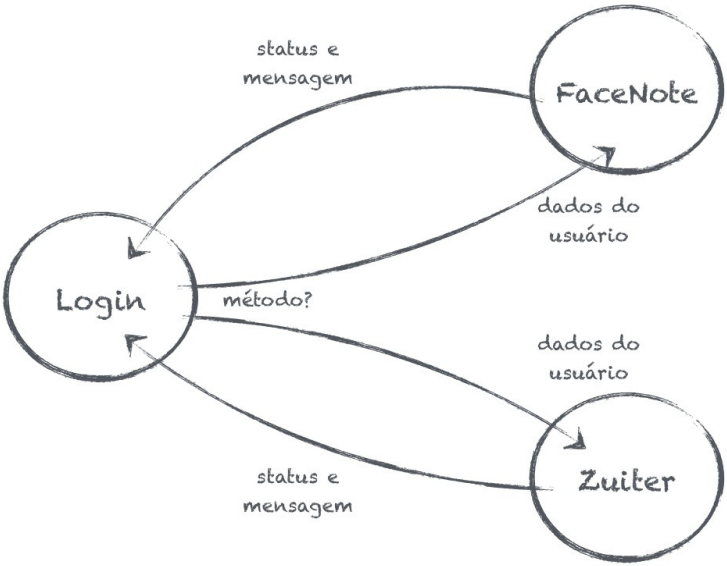

Figura 5.3: Comunicação com diferentes estratégias

Caso a API do Zuiter sofra uma alteração, em vez de alterar o método gigante da classe `Login`, basta fazer a mudança na classe de estratégia específica do Zuiter. O resto da lógica de validação continua funcionando da mesma maneira.

O primeiro passo para aplicar o Strategy é separar as lógicas ainda dentro da classe `Login`. Em vez de o método `Login.com` possuir lógica para todos os serviços, ele vai apenas delegar a chamada para outros métodos. Como exemplo, vamos usar **Extrair Método** para separar a lógica de autenticação via FaceNote, de forma que o método já retorne a `RespostaLogin` com `status` e `mensagem`:

```
// /src/main/java/strategy/Login.java
public class Login {
    private RespostaLogin autenticarViaFaceNote(String idUsuario)
{
        int resposta = servicoFaceNote.autenticar(idUsuario);
        String mensagem = "não foi possível autenticar";
        boolean status = false;

        if (resposta == FACE_NOTE_SUCESSO) {
            status = true;
            mensagem = "login com sucesso";
        } else if (resposta == FACE_NOTE_REVOCADO) {
            mensagem = "acesso revocado";
        } else if (resposta == FACE_NOTE_BLOQUEADO) {
            mensagem = "acesso bloqueado";
        }

        return new RespostaLogin(mensagem, status);
    }
```

Após extrair todas as formas de login para seus próprios métodos, Paula nota que o método principal fica responsável apenas por decidir qual forma de autenticação usar. Isso deixa o código bem menor e mais simples de se entender.

```
// /src/main/java/strategy/Login.java
public class Login {

    public RespostaLogin com(DadosDeLogin dadosDeLogin) {
        Autenticacao metodo = dadosDeLogin.getMetodo();
        String usuario = dadosDeLogin.getUsuario();
        if (metodo.equals(Autenticacao.VIA_FACENOTE)) {
            return autenticarViaFaceNote(usuario);
        } else if (metodo.equals(Autenticacao.VIA_ZUITER)) {
            return autenticarViaZuiter(usuario);
        }

        String mensagem =
            "método de autenticação não especificado";
        boolean status = false;

        return new RespostaLogin(mensagem, status);
    }
}
```

O próximo passo é extrair as estratégias. Paula cria a classe `LoginViaFaceNote`, onde ficará contida a lógica para chamar o `ServicoFaceNoteLogin` e para tratar o código de resposta da API. Com os métodos já separados na classe `Login`, basta utilizar **Extrair Classe** e mover os métodos e constantes relacionados à autenticação via FaceNote para a classe específica.

```
// /src/main/java/strategy/LoginViaFaceNote.java
public class LoginViaFaceNote {

    static int FACE_NOTE_SUCESSO = 200;
    static int FACE_NOTE_REVOCADO = 403;
    static int FACE_NOTE_BLOQUEADO = 408;

    private ServicoFaceNoteLogin servicoFaceNote;

    public LoginViaFaceNote(ServicoFaceNoteLogin servicoFaceNote)
{
        this.servicoFaceNote = servicoFaceNote;
    }

    public RespostaLogin autenticar(String idUsuario) {
```

```
        int resposta = servicoFaceNote.autenticar(idUsuario);
        String mensagem = "não foi possível autenticar";
        boolean status = false;

        if (resposta == FACE_NOTE_SUCESSO) {
            status = true;
            mensagem = "login com sucesso";
        } else if (resposta == FACE_NOTE_REVOCADO) {
            mensagem = "acesso revocado";
        } else if (resposta == FACE_NOTE_BLOQUEADO) {
            mensagem = "acesso bloqueado";
        }

        return new RespostaLogin(mensagem, status);
    }
}
```

Um bom exemplo da transparência do padrão Strategy é que, ao extrair estratégias, os testes da classe `Login` não precisam ser alterados. O contrato, o recebimento de um objeto `DadosDeLogin` com parâmetros e a devolução de `RespostaLogin` com status e mensagem continuam os mesmos.

Mover os testes da classe `Login` para as classes de estratégia é uma boa ideia, pois deixa claro e bem documentado qual o comportamento esperado das estratégias, além de ser uma mudança bem simples.

Veja como ficariam os testes para `LoginViaFaceNote`:

```
// /src/test/java/strategy/LoginViaFaceNoteTest.java
public class LoginViaFaceNoteTest {
  @Test
  public void autenticaViaFaceNote() throws Exception {
    ServicoFaceNoteLogin servicoFaceNote =
      new ServicoFaceNoteLoginFake();

    LoginViaFaceNote login =
      new LoginViaFaceNote(servicoFaceNote);
```

```
    String usuario = "Gil";

    RespostaLogin respostaLogin = login.autenticar(usuario);
    String mensagemSucesso = "login com sucesso";

    assertEquals(true, respostaLogin.getStatus());
    assertEquals(mensagemSucesso, respostaLogin.getMessage());
  }
}
```

A classe `Login` agora nem precisa mais acessar os serviços diretamente, basta utilizar as estratégias e ficar completamente livre de informações que são específicas do método de autenticação.

```
// /src/main/java/strategy/Login.java
public class Login {

    private LoginViaFaceNote loginViaFaceNote;
    private LoginViaZuiter loginViaZuiter;

    public Login(LoginViaFaceNote loginViaFaceNote,
                 LoginViaZuiter loginViaZuiter) {
        this.loginViaFaceNote = loginViaFaceNote;
        this.loginViaZuiter = loginViaZuiter;
    }

    public RespostaLogin com(DadosDeLogin dadosDeLogin) {
        Autenticacao metodo = dadosDeLogin.getMetodo();
        String usuario = dadosDeLogin.getUsuario();
        if (metodo.equals(Autenticacao.VIA_FACENOTE)) {
            return loginViaFaceNote.autenticar(usuario);
        } else if (metodo.equals(Autenticacao.VIA_ZUITER)) {
            return loginViaZuiter.autenticar(usuario);
        }

        String mensagem =
            "método de autenticação não especificado";
        boolean status = false;

        return new RespostaLogin(mensagem, status);
    }
}
```

Para os testes da classe `Login`, Paula precisa apenas garantir que, ao tentar uma autenticação, a estratégia correta será aplicada. Testar que `LoginViaFaceNote` é usado ao tentar fazer login via FaceNote, por exemplo, requer apenas uma verificação de que o método `LoginViaFaceNote.autenticar` será chamado ao executar `Login.com` utilizando um *mock*:

```
// /src/test/java/strategy/LoginTest.java

public class LoginTest {

  @Test
  public void fazLoginViaFaceNote() throws Exception {
    LoginViaFaceNote loginViaFaceNote =
      mock(LoginViaFaceNote.class);
    LoginViaZuiter loginViaZuiter =
      mock(LoginViaZuiter.class);

    Login login = new Login(loginViaFaceNote, loginViaZuiter);

    String usuario = "Paula";
    DadosDeLogin dadosDeLogin =
      new DadosDeLogin(usuario, Autenticacao.VIA_FACENOTE);

    login.com(dadosDeLogin);

    verify(loginViaFaceNote).autenticar(usuario);
  }
}
```

Responsabilidades foram distribuídas e as lógicas não estão mais misturadas em um só lugar. Além disso, adicionar novas maneiras de login requer poucas mudanças nas classes já existentes, basta a classe `Login` chamar a nova estratégia, que precisa apenas obedecer ao contrato já definido.

Paula olha de novo para o código e gosta do que vê. As lógicas das várias redes sociais estão bem separadas nas estratégias e

podem crescer sem nenhum impacto maior na aplicação. Na verdade, ela percebe que a sua ideia de criar diferentes classes de serviço nem é mais necessária, dado que o código dos serviços `ServicoFaceNoteLogin` e `ServicoZuiterLogin` é praticamente o mesmo. Nesse momento, ela já começa a pensar na sua próxima refatoração.

Capítulo 6

TEMPLATE METHOD: DEFININDO ALGORITMOS EXTENSÍVEIS

Em sistemas web, é muito comum existirem tarefas que precisam ser realizadas de maneira assíncrona, para que o servidor não fique travado apenas esperando e consumindo recursos desnecessários. Os processos que executam essas tarefas paralelas ao sistema principal são conhecidos como *workers*.

No seu projeto atual, Débora está trabalhando em algoritmos para alguns *workers* de um sistema. Ela já desenvolveu funcionalidades parecidas antes para outros, mas esse em especial requer um trabalho a mais. Esse projeto precisará de vários tipos de *workers*, por exemplo, enviar e-mails, importar informações a partir de um arquivo, ou até mesmo realizar buscas periódicas no banco de dados.

6.1 NEM TÃO DIFERENTES ASSIM

Inicialmente, Débora implementou o *worker* que envia e-mails. O teste a seguir mostra o seu funcionamento básico: dado um usuário, uma lista com destinatários e o tipo de conteúdo da

mensagem (convite, promocional etc.), criamos um objeto do tipo `EnviarEmail` e verificamos que foi enviado um e-mail para cada um dos destinatários com os dados corretos:

```
// /src/test/java/template/EmailWorkerTest.java
public class EmailWorkerTest {

  @Test
  public void mandaEmailParaDestinatarios() throws Exception {
    int idUsuario = 1;
    List<String> destinatarios = Arrays.asList(
      "email@email.com", "outro@email.com", "um@email.com");
    String assunto = "Convite enviado por Usuario1";

    ServicoEmail servicoEmail = new ServicoEmail();
    EmailWorker emailWorker = new EmailWorker(servicoEmail);
    EnviarEmail enviarEmail = new EnviarEmail(
      idUsuario, Email.CONVITE, destinatarios);
    EmailEnviado email = emailWorker.enviar(enviarEmail);

    assertEquals(destinatarios, email.getDestinatarios());
    assertEquals(assunto, email.getAssunto());
    assertEquals(3, email.getEmailsEnviados());
  }
}
```

A implementação também é bem simples: primeiro, buscamos as informações do usuário, para então gerar o conteúdo do e-mail e o assunto. Em seguida, chamamos o `servicoEmail.enviarEmail` que envia o e-mail e retorna informações em um objeto `EmailEnviado`.

```
// /src/main/java/template/EmailWorker.java
public class EmailWorker {
    public EmailEnviado enviar(EnviarEmail enviarEmail) {
        Usuario usuario =
            buscarUsuario(enviarEmail.getIdUsuario());
        String corpoEmail =
            gerarCorpoEmail(enviarEmail.getTipoEmail(), usuario);
        String assunto =
            gerarAssunto(enviarEmail.getTipoEmail(), usuario);
```

```
        EmailEnviado emailEnviado =
            new EmailEnviado("", Collections.emptyList());

        try {
            List<String> destinatarios =
                enviarEmail.getDestinatarios();
            emailEnviado = servicoEmail
                .enviarEmail(assunto, corpoEmail, destinatarios);
        } catch (TimeoutException e) {
            logger.log(Level.SEVERE, e.getMessage());
        }
        return emailEnviado;
    }
}
```

Os outros métodos não precisam ser detalhados aqui, pois não influenciam no entendimento do problema.

Ao planejar a implementação dos próximos *workers*, Débora percebe que será necessário adicionar comportamentos comuns, como *logs*, gerenciamento de exceções, estabelecer um tempo limite para que o processo não fique executando indefinidamente, entre outros.

Levando esses novos requisitos em consideração, a implementação final do método `enviar` lida com a exceção `TimeoutException` seguindo uma estratégia de tentar novamente, baseada no valor de `limiteTentativas`:

```
// /src/main/java/template/EmailWorker.java
public class EmailWorker {

    private int limiteTentativas;

    public EmailEnviado enviar(EnviarEmail enviarEmail) {
        Usuario usuario =
            buscarUsuario(enviarEmail.getIdUsuario());
        String corpoEmail =
            gerarCorpoEmail(enviarEmail.getTipoEmail(), usuario);
        String assunto =
```

```
            gerarAssunto(enviarEmail.getTipoEmail(), usuario);
        EmailEnviado emailEnviado =
            new EmailEnviado("", Collections.emptyList());

        int contadorTentativas = 0;
        while (contadorTentativas < limiteTentativas) {
            try {
                List<String> destinatarios =
                    enviarEmail.getDestinatarios();
                emailEnviado = servicoEmail.enviarEmail(
                    assunto, corpoEmail, destinatarios);
                contadorTentativas = limiteTentativas;
            } catch (TimeoutException e) {
                logger.log(Level.SEVERE, e.getMessage());
                contadorTentativas++;
            }
        }
        return emailEnviado;
    }
}
```

Após avaliar o problema com uma visão mais geral, Débora percebe que, apesar de as tarefas serem diferentes, o fluxo de execução dos *workers* pode seguir um mesmo padrão. Ela acaba rascunhando algo assim:

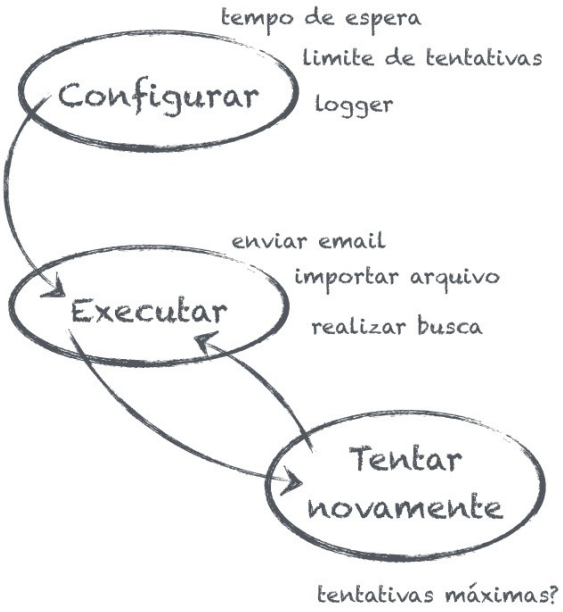

Figura 6.1: Fluxo de execução dos workers

Pensando em uma maneira de fazer com que esse algoritmo fique simples de ser reutilizado, ela avalia criar apenas um *worker*, e definir o que ele executa usando o padrão Strategy para trocar as implementações. O problema é que cada *worker* precisa fazer mais do que apenas o método de execução, o que pode tornar as estratégias mais complicadas.

O que ela precisa é definir um modelo para todos os *workers* seguirem e, assim, reduzir as duplicações entre eles. E é então que ela descobre o Template Method!

6.2 O PADRÃO TEMPLATE METHOD

Assim como no Strategy, o Template Method também é um padrão comportamental que busca simplificar as responsabilidades dos objetos. O problema a ser resolvido é que temos um conjunto de algoritmos que, apesar de seguir um mesmo fluxo, precisa de alguma flexibilidade.

O contexto para utilização do padrão é quando podemos separar o comportamento base, comum a todos os algoritmos, criando um *template* com pontos de extensão.

A estrutura base define os métodos que são chamados, a ordem de execução e o que é retornado por eles, em uma única classe que será usada pelo cliente para executar os algoritmos. Cada algoritmo específico herda dessa classe base e sobrescreve os métodos para implementar sua própria lógica.

Os métodos que são sobrescritos em cada algoritmo são chamados de **métodos gancho**. Eles fornecem um alto nível de flexibilidade para a solução, em que parte fica compartilhada e parte é específica. Além disso, por existir apenas uma única classe para chamar os algoritmos, o código de utilização fica o mesmo para qualquer que seja a implementação.

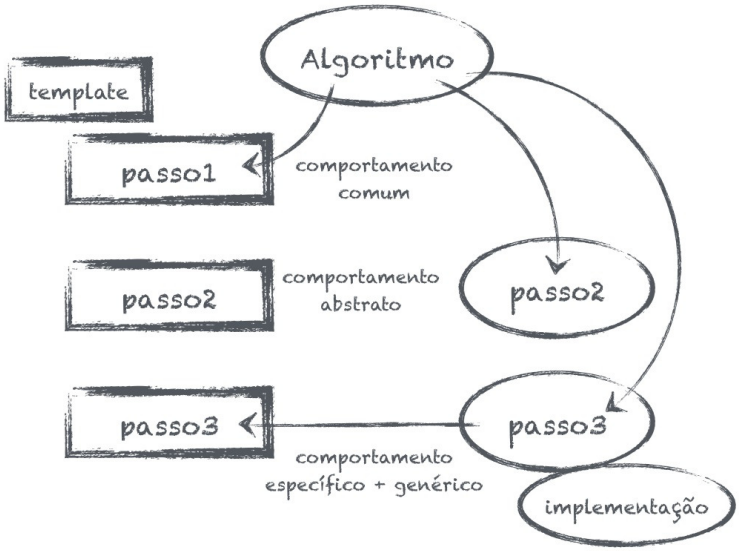

Figura 6.2: Template e sua implementação

Pela figura, podemos perceber a flexibilidade da utilização do padrão:

- O `passo1` mostra um método comum a todas as implementações, portanto, é definido apenas no lado do *template*.
- Já o `passo2` é específico de cada algoritmo, sendo definido apenas na implementação.
- E o `passo3` mostra que é possível um comportamento misto, com uma parte comum a todos (definida no *template* e acessada via `super`), e outra específica de cada algoritmo (definida na implementação).

Para resolver o seu problema, Débora cria a classe base `TemplateWorker` definindo os passos do algoritmo. No método

`executar`, vamos chamar os ganchos que serão definidos pelas subclasses. Como cada *worker* precisa de dados diferentes para funcionar, a implementação do método `executar` retorna um tipo genérico `T`, que será definido durante a compilação, e recebe como parâmetro um `Object`, que também é um objeto genérico e pode ser convertido para qualquer outro objeto nas subclasses.

```
// /src/main/java/template/TemplateWorker.java
public abstract class TemplateWorker {

    public <T>T executar(Object parametros) {
        antesExecucao(parametros);
        T resultado = valorPadraoDeRetorno();
        do {
            try {
                resultado = trabalhar(parametros);
            } catch (TimeoutException e) {
                trataExcecao(e);
            }
        } while (deveContinuarTentando());
        return resultado;
    }
}
```

Os métodos gancho de `TemplateWorker` são definidos na classe base como `abstract`, para forçar que os *workers* os definam, ou com uma implementação simples (ou até mesmo vazia), tornado sua implementação opcional na subclasse.

```
// /src/main/java/template/TemplateWorker.java
public abstract class TemplateWorker {

    protected abstract <T> T valorPadraoDeRetorno();

    protected abstract void trataExcecao(TimeoutException e);

    protected boolean deveContinuarTentando() {
        return false;
    };
```

```
    protected abstract <T>T trabalhar(Object parametros)
      throws TimeoutException;

    protected void antesExecucao(Object parametros) { };
}
```

Os testes da classe `EmailWorker` podem continuar os mesmos de antes. A única diferença é que agora eles devem chamar o método `executar`, definido na classe base `TemplateWorker`:

```
// /src/test/java/template/EmailWorkerTest.java
public class EmailWorkerTest {

    @Test
    public void mandaEmailParaDestinatarios() throws Exception {
        int idUsuario = 1;
        List<String> destinatarios = Arrays.asList(
          "email@email.com", "outro@email.com", "um@email.com");
        String assunto = "Convite enviado por Usuario1";

        ServicoEmail servicoEmail = new ServicoEmail();
        EmailWorker emailWorker = new EmailWorker(servicoEmail);
        EnviarEmail enviarEmail = new EnviarEmail(
              idUsuario, Email.CONVITE, destinatarios);
        EmailEnviado email = emailWorker.executar(enviarEmail);

        assertEquals(destinatarios, email.getDestinatarios());
        assertEquals(assunto, email.getAssunto());
        assertEquals(3, email.getEmailsEnviados());
    }
}
```

Com a utilização dos métodos ganchos, a lógica do antigo método `enviar` será dividida, usando vários **Extrair Método**. O método `trabalhar` vai conter a execução principal do *worker*, agrupando informações e chamando o método `enviar_email`:

```
// /src/main/java/template/EmailWorker.java
public class EmailWorker extends TemplateWorker {
    @Override
    protected <T> T trabalhar(Object parametros)
        throws TimeoutException {
```

```
        EnviarEmail enviarEmail = (EnviarEmail) parametros;
        Usuario usuario =
            buscarUsuario(enviarEmail.getIdUsuario());
        String corpoEmail =
            gerarCorpoEmail(enviarEmail.getTipoEmail(), usuario);
        String assunto =
            gerarAssunto(enviarEmail.getTipoEmail(), usuario);

        EmailEnviado emailEnviado = servicoEmail.enviarEmail(
            assunto, corpoEmail, enviarEmail.getDestinatarios());
        contadorTentativas = limiteTentativas;
        return (T) emailEnviado;
    }
}
```

No método gancho `antesExecucao`, será feita a configuração específica do *worker* para tentar novamente em caso de falha.

```
// /src/main/java/template/EmailWorker.java
public class EmailWorker extends TemplateWorker {
    @Override
    protected void antesExecucao(Object parametros) {
        limiteTentativas = 5;
        contadorTentativas = 0;
    }
}
```

Já no método `deveContinuarTentando`, vamos adicionar a lógica para executar o método novamente, caso o `contadorTentativas` ainda não tenha ultrapassado o `limiteTentativas`.

```
// /src/main/java/template/EmailWorker.java
public class EmailWorker extends TemplateWorker {
    @Override
    protected boolean deveContinuarTentando() {
        return limiteTentativas > contadorTentativas;
    }
}
```

Com o algoritmo bem definido, adicionar novos *workers* será

apenas uma questão de encaixar sua lógica nos ganchos, pois a execução principal se mantém a mesma. Caso o novo *worker* precise de um novo gancho, basta definir um método vazio no `TemplateWorker` para que os *workers* já existentes não precisem ser modificados. Assim, além de ser fácil adicionar novos *workers*, também é fácil realizar mudanças no algoritmo base e incorporar novos requisitos à solução.

Capítulo 7

ADAPTER: SEJA COMO A ÁGUA

Quando tomamos a decisão de aposentar um sistema, ou um conjunto de sistemas, os dados antigos são uma grande preocupação. Achar uma maneira de utilizá-los é muitas vezes essencial, mas é preciso evitar que a nova aplicação fique presa ao design da anterior.

Essa é a situação na qual Celso se encontra. Ele está à frente de uma equipe, criando uma aplicação que vai mapear informações sobre **fornecedores**, **estoque** e **clientes** de uma grande loja online. A ideia principal é substituir um conjunto de aplicações por uma única, centralizando as informações e facilitando a vida dos atendentes que precisam abrir múltiplas janelas para buscar partes da informação.

7.1 CAOS E ORDEM

Celso olha mais uma vez para a documentação dos sistemas antigos pensando em como orquestrar todas essas informações. A aplicação que controla o **estoque** guarda todas as informações em uma base de dados não relacional. Já a que contém informações sobre os **fornecedores** utiliza um serviço de mensagens em fila.

Por fim, as informações sobre **clientes** são expostas em uma API *soap* trocando arquivos XML (http://pt.wikipedia.org/wiki/SOAP).

Para a primeira grande entrega do novo sistema, será permitido acessar e editar as informações dos clientes que estão acessíveis utilizando a API *soap* do sistema legado. A classe `Cliente` foi criada para obter essas informações, com um atributo chamado `idUniversal`, que é um código usado para identificar clientes em todos os sistemas, e um `ClienteSoap` que é usado para acessar o serviço *SOAP*.

```
// /src/main/java/adapter/Cliente.java
public class Cliente {

    private String idUniversal;

    private ClienteSoap clienteSoap;

    public Cliente(String idUniversal, ClienteSoap clienteSoap) {
        this.idUniversal = idUniversal;
        this.clienteSoap = clienteSoap;
    }
}
```

No sistema antigo, as preferências eram divididas de acordo com seu propósito, por exemplo, preferências de notificações por e-mail ou quais os endereços para entregas. Porém, nessa nova versão, todas elas serão exibidas em uma mesma tela, logo, Celso decide obter todas as preferências e guardá-las no mesmo objeto.

O método que obtém as preferências de um cliente vai executar uma série de chamadas ao serviço externo, coletar cada pedaço da informação e depois vai agrupá-las em um único objeto `PreferenciasCliente`. Por fim, o método converte `PreferenciasCliente` para *JSON*, pois o novo sistema oferecerá uma API para acessar os dados.

```
// /src/main/java/adapter/Cliente.java
public class Cliente {

    public String getPreferencias() {
        String emailsXml =
            clienteSoap.obterPreferenciasEmail(idUniversal);
        List<String> emails = emailsXml == null ?
            Collections.emptyList() :
            Arrays.asList(emailsXml.split(","));

        String cartao =
            clienteSoap.obterPreferenciasCartao(idUniversal);
        cartao = cartao == null ? "" : cartao;

        String telefonesXml =
            clienteSoap.obterPreferenciasTelefone(idUniversal);
        List<String> telefones = telefonesXml == null ?
            Collections.emptyList() :
            Arrays.asList(telefonesXml.split(","));

        String endereco =
            clienteSoap.obterPreferenciasEndereco(idUniversal);
        endereco = endereco == null ? "" : endereco;

        PreferenciasCliente preferenciasCliente =
            new PreferenciasCliente(
                emails, endereco, telefones, cartao);

        return new Gson().toJson(preferenciasCliente);
    }
}
```

Os métodos de `clienteSoap` vão apenas executar a chamada ao serviço *SOAP* externo, e não precisam ser detalhados aqui. A questão maior do problema é a definição do contrato entre o sistema legado e o novo.

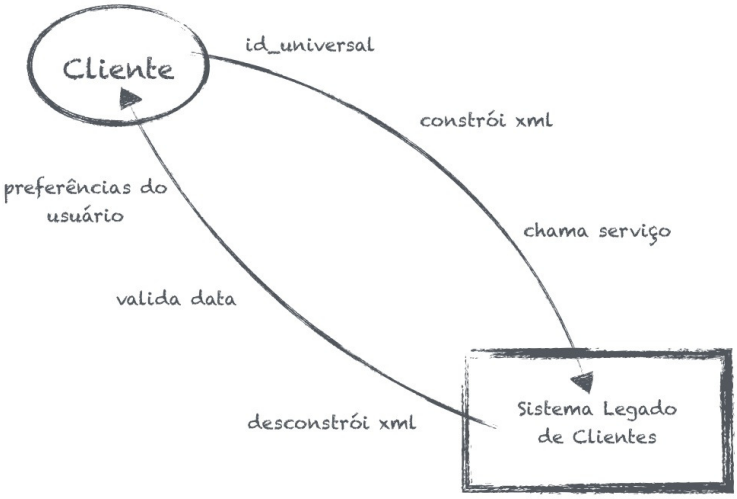

Figura 7.1: Contrato entre o cliente e o serviço legado

Nos testes para o método `preferencias`, Celso utilizou um *mock* para que as chamadas externas não influenciem na execução.

```
// /src/test/java/adapter/ClienteTest.java
public class ClienteTest {

    @Test
    public void retornaListaDeEmailsDoCliente() {
        String idUniversal = "FG123";
        String emailPrincipal = "cliente@cliente.com";
        String emailSecundario = "cliente@email.com";
        List<String> emailsCliente =
            Arrays.asList(emailPrincipal, emailSecundario);

        ClienteSoap clienteSoap = mock(ClienteSoap.class);
        when(clienteSoap.obterPreferenciasEmail(idUniversal))
            .thenReturn(emailsCliente);
        Cliente cliente = new Cliente(idUniversal, clienteSoap);

        String preferenciasJson = cliente.getPreferencias();
        PreferenciasCliente preferencias = new Gson().fromJson(
            preferenciasJson, PreferenciasCliente.class);
```

```
        assertEquals(2, preferencias.getEmails().size());
        assertEquals(emailPrincipal,
            preferencias.getEmails().get(0));
        assertEquals(emailSecundario,
            preferencias.getEmails().get(1));
    }
}
```

Apesar de o método `preferencias` delegar ações para vários métodos, ele ainda concentra muita lógica de maneira bem sutil. Note as pequenas verificações de valor nulo, como `endereco = endereco == null ? "" : endereco;` , e as chamadas de `split(",")` para separar *strings* em um *array* e depois converter esses *arrays* em listas. Mesmo que simples, ainda são lógicas para transformar os dados recebidos do sistema legado.

Celso não consegue parar de pensar em quão doloroso será quando o sistema antigo for aposentado e todo o código já feito tiver de ser alterado. Ele sabe que o código vai ser alterado, mas não como e nem quando! Pode ser que, no futuro, os dados sejam lidos de preferência em um banco de dados local, ou talvez seja necessário utilizar tanto o serviço externo quanto o banco local ao mesmo tempo!

A ideia que Celso vai tentar aplicar é definir uma interface intermediária para obtenção de dados. Dessa forma, obtê-los será feito independentemente de como serão utilizados, e eles estarão em classes separadas e compartilhadas. Essa camada de flexibilidade intermediária pode ser implementada seguindo o padrão Adapter.

7.2 O PADRÃO ADAPTER

O problema resolvido pelo Adapter é quando temos duas

classes com interfaces diferentes, mas que precisam trabalhar em conjunto. A solução é isolar as conversões de uma interface para a outra, do resto da lógica de negócio. Para aplicar bem a solução, é preciso que o contexto de uso do padrão permita separar bem a transformação dos dados do resto da lógica.

Utilizar o Adapter vai definir um contrato com o que é esperado pela classe cliente e com o que o adaptador precisa definir. Criar novos adaptadores não requer integração com o cliente, basta que o dado seja retornado na maneira esperada.

A figura a seguir mostra como o Adapter introduz uma camada intermediária para deixar a classe `Cliente` mais leve e com menos responsabilidades:

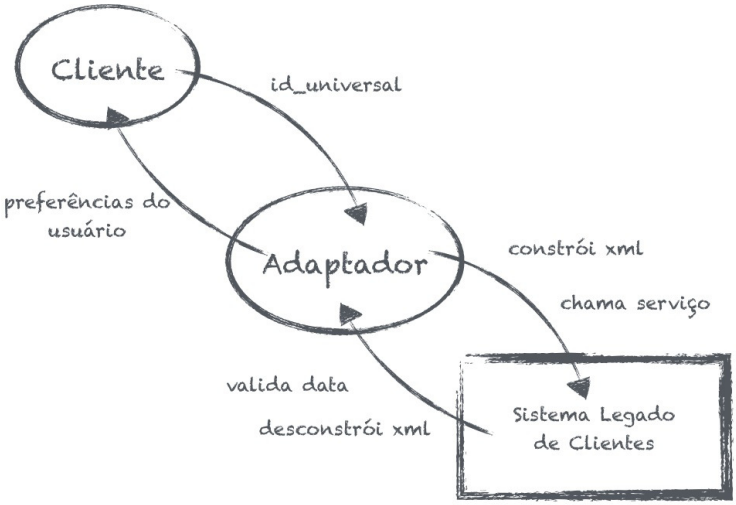

Figura 7.2: O Adaptador é quem define o contrato com Cliente

O contrato é definido no método `preferencia` , que utiliza o `idUniversal` e retorna as informações de e-mail, endereço,

pagamento e telefone em um único objeto `PreferenciasCliente`. O primeiro passo de Celso é usar **Extrair Método** para separar a responsabilidade de fazer a requisição *SOAP*.

```
// /src/main/java/adapter/Cliente.java
public class Cliente {
    public String getPreferencias() {
        PreferenciasCliente preferenciasCliente =
            getPreferenciasCliente();
        return new Gson().toJson(preferenciasCliente);
    }

    private PreferenciasCliente getPreferenciasCliente() {
        String emailsXml =
            clienteSoap.obterPreferenciasEmail(idUniversal);
        List<String> emails = emailsXml == null ?
            Collections.emptyList() :
            Arrays.asList(emailsXml.split(","));

        String cartao =
            clienteSoap.obterPreferenciasCartao(idUniversal);
        cartao = cartao == null ? "" : cartao;

        String telefonesXml =
            clienteSoap.obterPreferenciasTelefone(idUniversal);
        List<String> telefones = telefonesXml == null ?
            Collections.emptyList() :
            Arrays.asList(telefonesXml.split(","));

        String endereco =
            clienteSoap.obterPreferenciasEndereco(idUniversal);
        endereco = endereco == null ? "" : endereco;

        return new PreferenciasCliente(
            emails, endereco, telefones, cartao);
    }
}
```

Com a separação feita, fica fácil saber o que deve ser movido para o adaptador *SOAP*. Utilizamos **Extrair Classe** em `Cliente`, e criamos a classe `AdaptadorSoap`, que precisa apenas do

`idUniversal` de um `Usuario` para realizar as requisições.

```
// /src/main/java/adapter/AdaptadorSoap.java
public class AdaptadorSoap {

    private ClienteSoap clienteSoap;

    public AdaptadorSoap(ClienteSoap clienteSoap) {
        this.clienteSoap = clienteSoap;
    }

    private PreferenciasCliente
                getPreferenciasCliente(String idUniversal) {
        String emailsXml =
            clienteSoap.obterPreferenciasEmail(idUniversal);
        List<String> emails = emailsXml == null ?
            Collections.emptyList() :
            Arrays.asList(emailsXml.split(","));

        String cartao =
            clienteSoap.obterPreferenciasCartao(idUniversal);
        cartao = cartao == null ? "" : cartao;

        String telefonesXml =
            clienteSoap.obterPreferenciasTelefone(idUniversal);
        List<String> telefones = telefonesXml == null ?
            Collections.emptyList() :
            Arrays.asList(telefonesXml.split(","));

        String endereco =
            clienteSoap.obterPreferenciasEndereco(idUniversal);
        endereco = endereco == null ? "" : endereco;

        return new PreferenciasCliente(
            emails, endereco, telefones, cartao);
    }
}
```

Uma boa ideia seria copiar os testes da classe `Cliente` para `AdaptadorSoap`, já que agora a lógica se encontra na classe `adaptador`:

```
// /src/test/java/adapter/AdaptadorSoapTest.java
```

```
public class AdaptadorSoapTest {

    @Test
    public void retornaListaDeEmailsDoCliente() {
        String idUniversal = "FG123";
        String emails = "cliente@cliente.com,cliente@email.com";

        ClienteSoap clienteSoap = mock(ClienteSoap.class);
        when(clienteSoap.obterPreferenciasEmail(idUniversal))
            .thenReturn(emails);
        AdaptadorSoap adaptadorSoap =
            new AdaptadorSoap(clienteSoap);

        PreferenciasCliente preferencias =
            adaptadorSoap.getPreferenciasCliente(idUniversal);

        assertEquals(2, preferencias.getEmails().size());
        assertEquals("cliente@cliente.com",
            preferencias.getEmails().get(0));
        assertEquals("cliente@email.com",
            preferencias.getEmails().get(1));
    }
```

O código final da classe `Cliente` fica bem mais simples agora que a única preocupação é delegar a obtenção dos dados para o adaptador, e depois transformar o objeto `PreferenciasCliente` em uma `String` contendo o *JSON*:

```
// /src/main/java/adapter/Cliente.java
public class Cliente {

    private String idUniversal;

    private AdaptadorSoap adaptadorSoap;

    public Cliente(
        String idUniversal, AdaptadorSoap adaptadorSoap) {

        this.idUniversal = idUniversal;
        this.adaptadorSoap = adaptadorSoap;
    }

    public String getPreferencias() {
```

```
        PreferenciasCliente preferenciasCliente =
            adaptadorSoap.getPreferenciasCliente(idUniversal);
        return new Gson().toJson(preferenciasCliente);
    }
}
```

Feito isso, Celso pode remover o código antigo da classe cliente, bem como seus testes. A responsabilidade da classe `Cliente` é apenas a transformação para *JSON*. Celso deixa um teste simples, que valida a criação do objeto, mas ele nem seria necessário, já que a conversão para *JSON* é feita por outra biblioteca.

```
// /src/test/java/adapter/ClienteTest.java
public class ClienteTest {

  @Test
  public void retornaPreferenciasDoCliente() {
    String idUniversal = "FG123";
    String email = "cliente@cliente.com";
    String telefone = "+1234567890";
    String endereco = "Rua ABC n 123, Cidade, Estado";
    String cartao = "1234 5678";

    ClienteSoap clienteSoap = mock(ClienteSoap.class);
    when(clienteSoap.obterPreferenciasEmail(idUniversal))
        .thenReturn(email);
    when(clienteSoap.obterPreferenciasTelefone(idUniversal))
        .thenReturn(telefone);
    when(clienteSoap.obterPreferenciasEndereco(idUniversal))
        .thenReturn(endereco);
    when(clienteSoap.obterPreferenciasCartao(idUniversal))
        .thenReturn(cartao);
    AdaptadorSoap adaptadorSoap =
        new AdaptadorSoap(clienteSoap);

    Cliente cliente = new Cliente(idUniversal, adaptadorSoap);
    String preferenciasJson = cliente.getPreferencias();
    PreferenciasCliente preferencias = new Gson()
      .fromJson(preferenciasJson, PreferenciasCliente.class);

    assertEquals(telefone,preferencias.getTelefones().get(0));
```

```
    assertEquals(email, preferencias.getEmails().get(0));
    assertEquals(cartao, preferencias.getCartaoPagamento());
    assertEquals(endereco, preferencias.getEndereco());
  }
}
```

Agora, caso seja necessário ler os dados de qualquer outra forma, o contrato entre a classe `Cliente` e os adaptadores está bem claro: basta que o adaptador receba o `idUniversal` e retorne as preferências de e-mail, endereço, pagamento e telefone em um objeto `PreferenciasCliente`, e tudo funcionará sem problemas.

Celso fica feliz com o resultado final da refatoração e mostra para o resto do time o quão fácil será manter o código da aplicação, já que a comunicação com o serviço externo está isolada. Caso essa comunicação precise mudar, também será algo fácil de fazer; basta seguir o contrato.

Padrões situacionais

Capítulo 8

STATE: 11 ESTADOS E 1 OBJETO

Guilherme é um desenvolvedor que conseguiu o emprego dos seus sonhos: desenvolver jogos! Seu primeiro projeto é um jogo de plataforma no qual Maria, a personagem principal, é uma encanadora que foi parar em um mundo surreal para salvar o príncipe do reino.

Ao longo de sua jornada, Maria vai utilizar vários itens que lhe darão novos poderes para enfrentar os desafios. No entanto, o desafio real será Guilherme conseguir modelar o código de uma maneira que facilite as várias mudanças no comportamento da heroína.

8.1 MARIA E SEUS PODERES

Segundo as especificações do jogo, Maria começa-o em um formato pequeno, sem poderes especiais, e pode ir pegando itens que lhe darão novos poderes ao longo dele. Ao colidir com um inimigo, ela leva dano e perde seus poderes; caso não tenha nenhum, morre e perde o jogo. A figura a seguir mostra as transições da personagem de acordo com os acontecimentos do jogo:

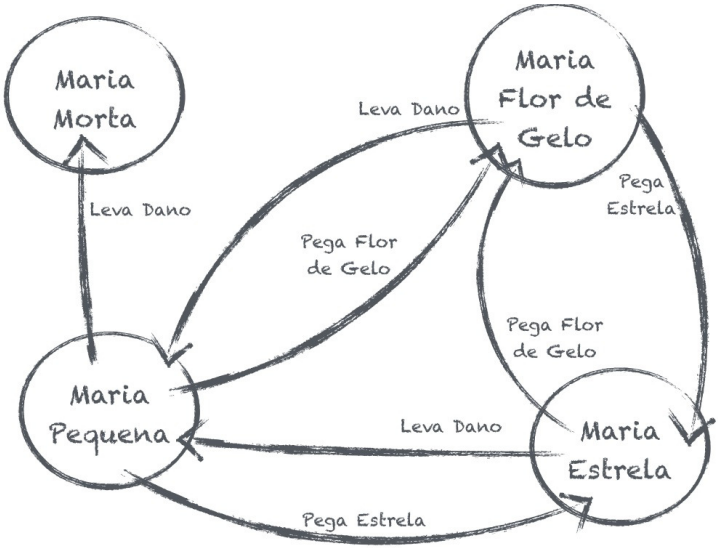

Figura 8.1: Ações e itens de Maria

Com base nesses requisitos iniciais, Guilherme começou uma implementação básica do comportamento da personagem na classe `Maria`:

```
// /src/main/java/strategy/Maria.java
public class Maria {

    private EstadoMaria estadoAtual;

    public Maria() {
        this.estadoAtual = EstadoMaria.PEQUENA;
    }
}
```

E para armazenar os possíveis estados, ele criou o `enum` `EstadoMaria`:

```
// /src/main/java/strategy/EstadoMaria.java
public enum EstadoMaria {
    PEQUENA, FLOR_DE_GELO, ESTRELA, MORTA
```

```
}
```

Ainda na classe `Maria`, Guilherme definiu as ações da personagem: pegar a flor de gelo, levar dano de um inimigo e pegar uma estrela.

```
// /src/main/java/strategy/Maria.java
public class Maria {
    public void pegarFlorDeGelo() {
        if (estadoAtual.equals(EstadoMaria.ESTRELA)) {
            return;
        }
        estadoAtual = EstadoMaria.FLOR_DE_GELO;
    }

    public void pegarEstrela() {
        estadoAtual = EstadoMaria.ESTRELA;
    }

    public void levarDano() {
        if (estadoAtual.equals(EstadoMaria.ESTRELA)) {
            return;
        }
        if (estadoAtual.equals(EstadoMaria.PEQUENA)) {
            estadoAtual = EstadoMaria.MORTA;
        } else {
            estadoAtual = EstadoMaria.PEQUENA;
        }
    }
}
```

Um exemplo dos testes que Guilherme escreveu é criar um novo objeto `Maria`, chamar o método `pegarFlorDeGelo` e garantir que o estado atual seja `EstadoMaria.FLOR_DE_GELO`. Veja o código:

```
// /src/test/java/strategy/MariaTest.java
public class MariaTest {

    @Test
    public void mariaPequenaPegaFlorDeGelo() throws Exception {
        Maria maria = new Maria();
```

```
        maria.pegarFlorDeGelo();
        EstadoMaria estadoAtual = maria.getEstadoAtual();
        assertEquals(EstadoMaria.FLOR_DE_GELO, estadoAtual);
    }
}
```

Com os requisitos atuais, a classe `Maria` está relativamente simples. Tudo bem que existem vários `ifs`, principalmente no método `levarDano`, mas nada que doa muito.

Após uma demonstração para um pequeno grupo de usuários, a equipe do jogo percebe que os itens da personagem Maria tornaram o jogo bem mais divertido. Assim, eles decidem adicionar novos itens que Maria pode utilizar ao longo de sua jornada. Ao ouvir isso, Guilherme já começa a pensar na classe `Maria`, e nas novas constantes e `ifs` que precisam mudar.

Depois de alguma discussão, o time decidiu incrementar o jogo da seguinte forma: Maria agora pode pegar o item flor de fogo, que tem comportamento bem semelhante ao item flor de gelo. A figura mostra como seriam as transições de estados considerando apenas o novo item:

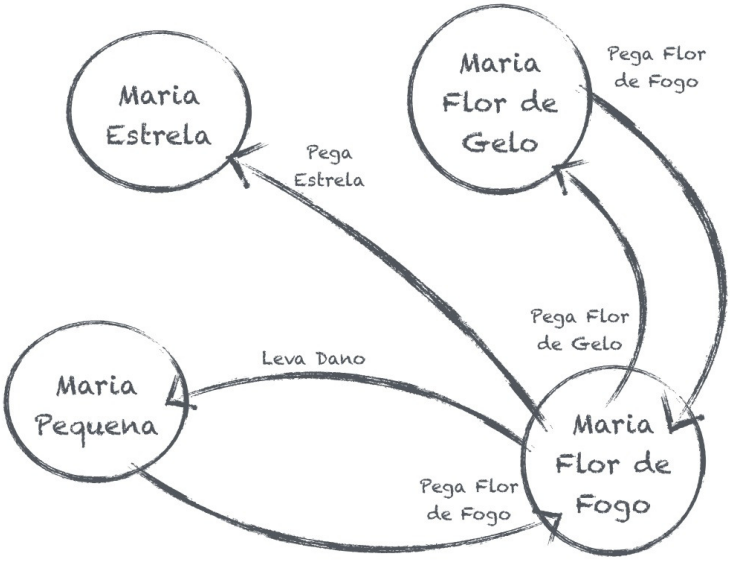

Figura 8.2: Adicionando novo item e ações para pegar flor de fogo

Codificar mais essas ações faria com que aquele grupo pequeno de `ifs` se tornasse um inferno de manutenção. Guilherme sugeriu para a equipe que, em vez de apenas uma classe fazer tudo, tentassem dividir o código criando uma classe para lidar com cada estado que Maria pode assumir.

Outra vantagem que Guilherme apresenta é que as regras de transição ficam definidas nos próprios estados, assim, ao executar uma ação, cada estado sabe qual deve ser o próximo. Isso evitaria que a lógica ficasse concentrada apenas em um lugar, acumulando vários `ifs` . A solução proposta é o padrão State.

8.2 O PADRÃO STATE

Pense no problema como sendo uma máquina de estados, em

que cada habilidade (estrela, flor de gelo etc.) é um estado, e cada ação (pegar uma flor de gelo, levar dano etc.) é uma transição entre eles. Ao utilizar este padrão, buscamos uma maneira de compartilhar a responsabilidade de trocar estados, tratando cada um como uma classe e cada transição como um método.

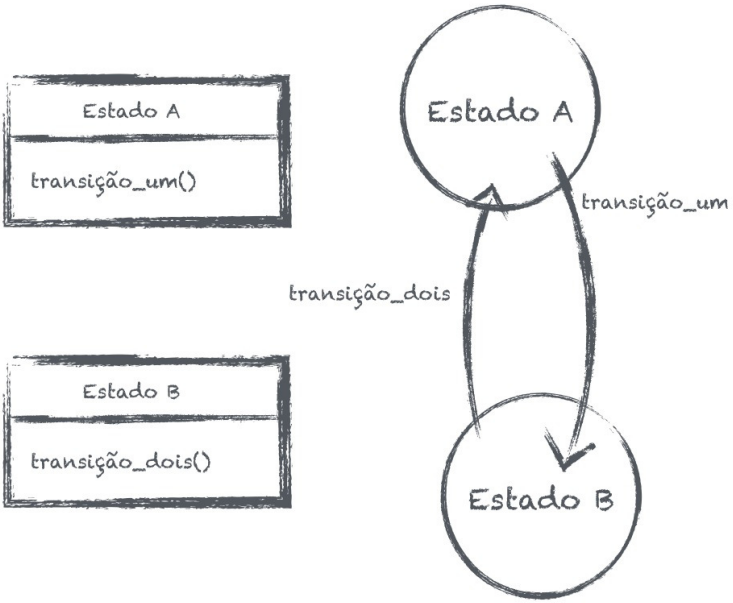

Figura 8.3: Como estados são modelados em classes

Como os outros padrões de comportamento, o State também busca simplificar o relacionamento entre classes. O contexto de utilização é quando existe um conjunto de comportamentos que precisa mudar constantemente, não sendo necessárias ações externas. A solução é separar as funcionalidades em classes diferentes e deixar que, em vez de o cliente fazer mudanças, cada estado saiba para qual estado deve mudar.

Cada estado precisa implementar os mesmos métodos de `Maria`, mas apenas com sua lógica específica. Para isso, Guilherme começa com a implementação de uma *interface* que define os métodos comuns a todos os estados de Maria:

```
// /src/main/java/state/Estado.java
public interface Estado {
    Estado pegarFlorDeGelo();

    Estado pegarEstrela();

    Estado levarDano();

    EstadoMaria getEstado();
}
```

Agora cada estado pode ser implementado seguindo essa interface e criando novas instâncias dos estados apropriados. Por exemplo, o estado `MariaPequena` ficaria assim:

```
// /src/main/java/state/MariaPequena.java
public class MariaPequena implements Estado {
    @Override
    public Estado pegarFlorDeGelo() {
        return new MariaFlorDeGelo();
    }

    @Override
    public Estado pegarEstrela() {
        return new MariaEstrela();
    }

    @Override
    public Estado levarDano() {
        return new MariaMorta();
    }

    @Override
    public EstadoMaria getEstado() {
        return EstadoMaria.PEQUENA;
    }
}
```

Para inserir o novo estado, em vez de utilizar as constantes definidas no `enum EstadoMaria`, a classe `Maria` terá uma instância da interface `Estado` recém-criada:

```
// /src/main/java/state/Maria.java
public class Maria {

    private Estado estadoAtual;

    public Maria() {
        this.estadoAtual = new MariaPequena();
    }
}
```

Como cada método retorna o próximo estado, após Guilherme implementar todos os estados, a classe `Maria` não precisa mais fazer verificações como no código anterior `estadoAtual.equals(EstadoMaria.PEQUENA)`. Ela vai apenas delegar as chamadas das ações para o seu `estadoAtual`, e atualizá-lo com o resultado da ação.

```
// /src/main/java/state/Maria.java
public class Maria {
    public void pegarFlorDeGelo() {
        estadoAtual = estadoAtual.pegarFlorDeGelo();
    }

    public void pegarEstrela() {
        estadoAtual = estadoAtual.pegarEstrela();
    }

    public void levarDano() {
        estadoAtual = estadoAtual.levarDano();
    }

    public EstadoMaria getEstadoAtual() {
        return estadoAtual.getEstado();
    }
}
```

A ideia é que, a cada transição, Maria atualize seu estado atual. O novo estado, por sua vez, saberá o que fazer quando outra transição ocorrer – bem semelhante a uma máquina de estados.

Como a classe `Maria` continua seguindo a mesma interface, os testes continuam instanciando os mesmos objetos e chamando os mesmos métodos. A única diferença é que, para obter o `EstadoMaria`, basta chamar `getEstadoAtual`.

```
// /src/test/java/state/MariaTest.java
public class MariaTest {
    @Test
    public void mariaPequenaPegaFlorDeGelo() throws Exception {
        Maria maria = new Maria();
        maria.pegarFlorDeGelo();
        EstadoMaria estadoAtual = maria.getEstadoAtual();
        assertEquals(EstadoMaria.FLOR_DE_GELO, estadoAtual);
    }
}
```

Para implementar o novo item flor de fogo, é preciso criar a ação `pegarFlorDeFogo` em todos os estados. A implementação segue o mesmo raciocínio das já existentes: muda-se o `estadoAtual` para o novo estado `FlorDeFogo` e ele lidará com as novas mudanças. Veja como ficaria o teste:

```
// /src/test/java/state/MariaTest.java
public class MariaTest {
    @Test
    public void mariaPequenaPegaFlorDeFogo() throws Exception {
        Maria maria = new Maria();
        maria.pegarFlorDeFogo();
        EstadoMaria estadoAtual = maria.getEstadoAtual();
        assertEquals(EstadoMaria.FLOR_DE_FOGO, estadoAtual);
    }
}
```

O novo método `pegarFlorDeFogo` será definido como parte da interface dos estados já existentes:

```
// /src/test/java/state/MariaTest.java
public interface Estado {
    Estado pegarFlorDeFogo();
}
```

A implementação do novo estado `FlorDeFogo` será tão simples quanto as anteriores. Na ação `pegar_flor_de_fogo`, como já estamos no estado `FlorDeFogo`, nada acontece e não precisamos instanciar nenhum novo objeto.

```
// /src/main/java/state/MariaFlorDeFogo.java
public class MariaFlorDeFogo implements Estado {
    @Override
    public Estado pegarFlorDeGelo() {
        return new MariaFlorDeGelo();
    }

    @Override
    public Estado pegarEstrela() {
        return new MariaEstrela();
    }

    @Override
    public Estado levarDano() {
        return new MariaPequena();
    }

    @Override
    public Estado pegarFlorDeFogo() {
        return this;
    }

    @Override
    public EstadoMaria getEstado() {
        return EstadoMaria.FLOR_DE_FOGO;
    }
}
```

Por fim, como Guilherme precisou adicionar uma nova transição (pegar a flor de fogo), precisamos defini-la nos estados já existentes de acordo com as novas regras:

```
// /src/main/java/state/MariaPequena.java
public class MariaPequena implements Estado {
    @Override
    public Estado pegarFlorDeFogo() {
        return new MariaFlorDeFogo();
    }
}
```

Strategy e State

Se você reparar bem, o padrão Strategy é bem parecido com o State: ambos vão dividir uma lógica complexa dentro de objetos próprios. Ao avaliar o contexto para decidir qual padrão pode ajudá-lo melhor, note as seguintes características de cada um:

- No **Strategy**, uma vez definida a estratégia, ela não vai mudar naquela execução.
- No **State**, cada estado sabe suas transições, então, o cliente não garante o estado atual.

Para avaliar o uso dos padrões em um problema, basta se perguntar o seguinte:

- **Strategy:** a estratégia precisa mudar uma vez que ela foi definida? Se sim, os vários `ifs` provavelmente vão voltar a se espalhar pelo código para avaliarem um resultado e definirem uma nova estratégia.
- **State:** o cliente precisa ficar verificando qual o estado atual o tempo todo? Se sim, a vantagem de não ter os `ifs` espalhados pelo código desaparece completamente.

Após a sessão de refatoração, Guilherme olha para o código com bastante orgulho por sua sugestão ter ajudado a descomplicar o problema. No final das contas, o time ficou com bem mais classes do que na visão inicial da solução, mas cada uma ficou extremamente simples, com métodos de apenas uma linha e sem nenhum `if` ! Até mesmo aquela lista de constantes no começo da classe `Maria` desapareceu.

Capítulo 9

BUILDER: CONSTRUIR COM CLASSE

Ana é uma experiente desenvolvedora Ruby que já trabalhou em vários projetos, principalmente com *startups*. As aplicações, na maior parte, eram pequenas e simples, mas mesmo assim seguiam boas práticas, como testes automatizados e integração contínua. No entanto, ao se mudar para uma nova cidade, ela começou a trabalhar em uma aplicação de uma grande empresa de venda de carros, um cenário bem diferente para ela.

Falando em aplicações corporativas, classes grandes são um problema muito difícil de ser atacado. Geralmente, elas têm muitos atributos e possuem muita lógica associada. Contudo, em alguns casos, é realmente necessário que a classe possua uma quantidade razoável de atributos. Ana acabou de entrar em uma equipe que está construindo uma aplicação para negociação de carros que sofre desse problema.

Existe uma grande quantidade de informação associada aos carros: ano de fabricação, cor, modelo, fabricante etc. Como era de se esperar, existe uma classe `Carro` para representar essas informações, que contém também toda a lógica relacionada a elas.

9.1 MUITA INFORMAÇÃO EM UM SÓ LUGAR

Por ter entrado há pouco tempo, Ana consegue analisar o código sem os "costumes" dos outros integrantes da equipe. Ao olhar para a famosa classe `Carro`, ela não gosta muito da quantidade de atributos ali acumulados. Criar um objeto `Carro` é bem trabalhoso e demanda muitos parâmetros.

```
// /src/main/java/builder/Carro.java
public class Carro {
    private String modelo;
    private String fabricante;
    private String anoFabricacao;
    private String placa;
    private String cor;
    private String kmRodados;
    private String anoModelo;
    private String precoMinimo;
    private String precoAnunciado;

    public Carro(String modelo, String fabricante,
        String anoFabricacao, String placa, String cor,
        String kmRodados, String anoModelo,
        String precoMinimo, String precoAnunciado) {

        this.modelo = modelo;
        this.fabricante = fabricante;
        this.anoFabricacao = anoFabricacao;
        this.placa = placa;
        this.cor = cor;
        this.kmRodados = kmRodados;
        this.anoModelo = anoModelo;
        this.precoMinimo = precoMinimo;
        this.precoAnunciado = precoAnunciado;
    }
}
```

Olhando apenas para os nomes dos atributos, Ana começa a pensar em possibilidades de extraí-los em classes próprias; por exemplo, `ano_fabricacao` e `ano_modelo` poderiam ter sua

própria classe. Apesar de isso simplificar a criação, diminuindo a quantidade de parâmetros no construtor de `Carro`, ainda serão necessários os mesmos dados para criar instâncias.

Outro grande sintoma desse problema no código existente são os testes para essa classe. Qualquer teste que precise utilizar objetos do tipo `Carro` vai precisar passar muita informação, ou passar vários dados "lixo" (`null`, `""` etc.). Veja, por exemplo, esse teste que verifica a validação de que o ano do modelo não pode ser inferior ao de fabricação:

```
// /src/test/java/builder/CarroTest.java
public class CarroTest {

  @Test
  public void validaAnoModeloNaoPodeSerAnteriorAoAnoFabicacao() {
    int anoFabricacao = 2000;
    int anoModelo = 1999;

    Carro carroInvalido =
      new Carro("modelo a", "fabricante a",
          anoFabricacao, "abc1234", null,
          0l, anoModelo, 0l, 0l);

    String mensagemDeErro =
      "ano do modelo nao pode ser anterior ao ano de fabricacao";
    List<String> erros = carroInvalido.getErros();

    assertFalse(carroInvalido.validar());
    assertEquals(1, erros.size());
    assertEquals(mensagemDeErro, erros.get(0));
  }
}
```

O problema é que alguns daqueles atributos são de fato necessários para criar um novo objeto, e até existem validações para eles. O efeito colateral aqui é que vários testes precisam passar informações que não têm a menor utilidade para o caso de teste

(como `"ABC1234"`), dificultando entender o seu real objetivo.

Observe a quantidade de informação presente no teste a seguir, que valida se a placa está presente. Tente descobrir o que é passado como placa no construtor do `Carro`:

```
// /src/test/java/builder/CarroTest.java
@Test
public void validaPlacaExiste() {
  Carro carroInvalido =
    new Carro("modelo a", "fabricante a", 2000,
      null, null, 0l, 2000, 0l, 0l);

  String mensagemDeErro = "placa nao pode ser nulo";
  assertFalse(carroInvalido.validar());
  assertEquals(1, carroInvalido.getErros().size());
  assertEquals(mensagemDeErro, carroInvalido.getErros().get(0));
}
```

Ana pensa um pouco e avalia a utilização do padrão Simple Factory para separar a responsabilidade de criação, mas logo muda de ideia, pois as possibilidades de combinações de dados são muito grandes. A classe fábrica precisaria definir muitos métodos ou receber muitos parâmetros, o que seria uma lógica semelhante à do construtor atual, apenas escondida em um método.

O que ela realmente gostaria de fazer é definir um conjunto de valores padrão para os campos obrigatórios, mas ainda assim permitir que eles fossem sobrescritos. Assim, seria simples adicionar apenas os dados necessários para a situação em mãos, como as informações de ano no teste anterior. É aí que ela se depara com o padrão *Builder*, que pode ser exatamente o que ela procura!

9.2 O PADRÃO BUILDER

Por ser classificado como um padrão de criação, o Builder resolve o mesmo problema dos padrões *Factory*: separar a criação de objetos da lógica de negócio. No entanto, o contexto de sua utilização é quando o processo de criação precisa de muitos atributos diferentes, ou precisa ser mais flexível do que apenas chamar um método.

A solução é definir uma classe para criar o objeto por partes, assumindo um valor padrão para atributos não definidos pelo cliente. Dessa forma, garantimos que, ao final do processo, criamos um objeto válido e com as informações necessárias.

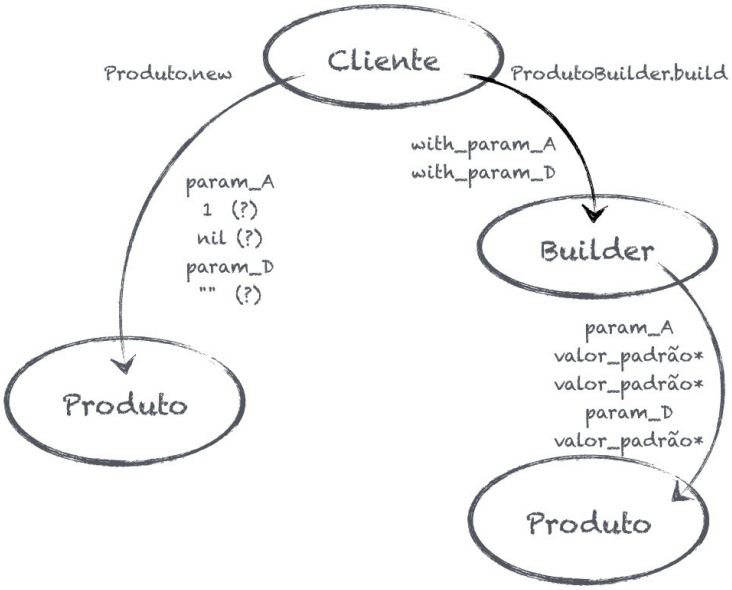

Figura 9.1: Chamada do construtor vs. Builder

De acordo com a imagem, precisamos de parâmetros que não são importantes se quisermos chamar o construtor. Em alguns casos, será necessário até mesmo criar objetos apenas para que o

método a ser chamado funcione. Usando o padrão, toda essa preocupação fica por conta do Builder, que apenas retorna o objeto com os valores de interesse do cliente.

Builder e Factory

Os padrões *Builder* e *Factory* são bem parecidos. Para decidir quando utilizar um ou o outro, pense apenas no processo de criação:

- Se o objeto a ser criado tem poucos atributos ou não precisa de muitas opções, use o Factory Method ou mesmo Simple Factory;
- Se existem muitos atributos a serem utilizados, ou é necessário dar muitas opções para criar o objeto, use o Builder.

Os padrões Simple Factory e Factory Method são os básicos para criação de objetos, pois apenas definem um método para instanciar objetos. Geralmente, eles serão a primeira opção, pois resolvem o problema de maneira bem simples. É conforme o projeto evolui que podemos ver a necessidade de deixar o processo de criação mais flexível; logo, o Builder pode ser uma boa opção.

Ana cria a classe `CarroValidoBuilder` e define nela um conjunto padrão de valores que um carro deve ter para ser válido. O teste para especificar esse comportamento seria instanciar o Builder — sem definir nenhum valor para os atributos do carro —,

criar a instância de `Carro` e verificar se ela é válida:

```
// /src/test/java/builder/CarroValidoBuilderTest.java
public class CarroValidoBuilderTest {

    @Test
    public void criaCarroValido() throws Exception {
        CarroValidoBuilder builder = new CarroValidoBuilder();
        Carro carroValido = builder.build();

        assertTrue(carroValido.validar());
    }
}
```

Os atributos padrões do carro serão definidos no construtor do `CarroValidoBuilder`, para que eles sejam atribuídos a uma nova instância de `Carro` no método `construir_carro`.

```
// /src/main/java/builder/CarroValidoBuilder.java
public class CarroValidoBuilder {
    private String modelo;
    private String fabricante;
    private int anoFabricacao;
    private String placa;
    private String cor;
    private long kmRodados;
    private int anoModelo;
    private long precoMinimo;
    private long precoAnunciado;

    public CarroValidoBuilder() {
        this.modelo = "Modelo A";
        this.fabricante = "Fabricante A";
        this.anoFabricacao = 2000;
        this.anoModelo = 2001;
        this.placa = "ABC1234";
    }

    public Carro build() {
        Carro carro =
            new Carro(modelo, fabricante, anoFabricacao,
                placa, cor, kmRodados, anoModelo,
                precoMinimo, precoAnunciado);
```

```
        return carro;
    }
}
```

Para deixar o `CarroValidoBuilder` mais flexível, serão adicionados métodos para configurar valores para os outros atributos. Um exemplo de teste seria criar um carro com o `CarroValidoBuilder`, modificar sua cor e quilometragem, e verificar os valores:

```
// /src/test/java/builder/CarroValidoBuilderTest.java
@Test
public void criaCarroComCorEKmRodados() {
    CarroValidoBuilder builder =
        new CarroValidoBuilder()
            .comCor("azul")
            .comKmRodados(12341);
    Carro carroValido = builder.build();

    assertTrue(carroValido.validar());
    assertEquals("azul", carroValido.getCor());
    assertEquals(12341, carroValido.getKmRodados());
}
```

Para encadear as chamadas dos métodos do Builder, é preciso retornar o próprio Builder ao final de cada método de configuração. Ou seja, os métodos `com_*` vão atribuir o valor para o atributo e retornar `this` para que novos métodos possam ser encadeados.

```
// /src/main/java/builder/CarroValidoBuilder.java
public class CarroValidoBuilder {
    public CarroValidoBuilder comCor(String cor) {
        this.cor = cor;
        return this;
    }

    public CarroValidoBuilder comKmRodados(long kmRodados) {
        this.kmRodados = kmRodados;
        return this;
```

```
    }
}
```

Essa técnica de encadear chamadas de métodos de um mesmo objeto se chama Interface Fluente, ou *Fluent Interface*, em inglês (descrição por Martin Fowler, em http://martinfowler.com/bliki/FluentInterface.html). Com ela, favorecemos a legibilidade do código, deixando as chamadas mais expressivas.

Veja como seria o código que configura um `Carro` sem a interface fluente, e compare com o código do teste visto anteriormente:

```
// sem interface fluente
CarroValidoBuilder builder = new CarroValidoBuilder();
builder.comCor("azul");
builder.comKmRodados(1234l);

// com interface fluente
CarroValidoBuilder builder =
    new CarroValidoBuilder()
        .comCor("azul")
        .comKmRodados(1234l);
```

Com a classe `CarroValidoBuilder` definida, podemos partir para a refatoração dos códigos na qual carros são instanciados. O exemplo de teste da classe `Carro` ficaria assim:

```
// /src/test/java/builder/CarroTest.java
public class CarroTest {
    @Test
    public void validaPlacaExiste() throws Exception {
        CarroValidoBuilder builder =
            new CarroValidoBuilder().comPlaca(null);

        Carro carroInvalido = builder.build();

        String mensagemDeErro = "placa nao pode ser nulo";
        List<String> erros = carroInvalido.getErros();
```

```
        assertFalse(carroInvalido.validar());
        assertEquals(1, erros.size());
        assertEquals(mensagemDeErro, erros.get(0));
    }
}
```

Facilitar a instanciação de objetos da classe `Carro` com o padrão Builder também permite que outros tipos de Builder sejam criados e usados, basta manter a mesma interface. Também é possível utilizá-lo no código de produção, aproveitando ainda mais da semântica das interfaces fluentes para deixar o código de criação de carros mais legível.

O padrão Builder permite mais do que apenas melhorar a legibilidade na construção de objetos. Os métodos de configuração dão uma maior flexibilidade, permitindo adicionar lógica específica em algum ponto.

Ao completar a refatoração, Ana mostra para seus colegas de equipe o quão mais legível ficou o código com a utilização do padrão Builder. Além disso, nenhum código de produção foi alterado por essa refatoração e os testes ficaram bem mais legíveis, o que facilitou a aceitação pelo resto da equipe.

Capítulo 10

DECORATOR: ADICIONANDO CARACTERÍSTICAS

Em um dos seus projetos de final de semana, Tarso está desenvolvendo um jogo no qual as personagens terão acesso a um grande número de armas. Além disso, cada arma poderá ser incrementada ao longo do jogo, melhorando as habilidades da personagem que a utiliza. Esses incrementos também poderão se acumular, permitindo que cada jogador customize ao máximo sua personagem e suas habilidades.

10.1 ESPADA MÁGICA FLAMEJANTE DA VELOCIDADE

Todo jogador começa com uma arma básica, cada uma com características diferentes. A partir daí, o jogador pode usar encantamentos na sua arma de acordo com seu estilo de jogo. Esse é o fator que torna o jogo atraente: um bom nível de customização para os jogadores permite que eles tenham muitos caminhos diferentes para seguir.

Para exemplificar, o seguinte caso de teste mostra como uma

adaga simples modificaria os atributos de uma personagem. O primeiro passo é criar uma `Personagem` e uma `Adaga`. Depois a adaga será equipada à personagem e o teste verifica que a sua `forca_de_ataque` recebeu o bônus da adaga.

```
// /src/test/java/decorator/PersonagemTest.java
public class PersonagemTest {

    @Test
    public void equiparAdagaAumentaForcaDeAtaqueEm10() {
        Personagem personagem = new Personagem(5);
        personagem.equiparArma(new Adaga());

        assertEquals(15, personagem.getForcaDeAtaque());
    }
}
```

A implementação da `Personagem` é bem simples: ela possui seus atributos básicos e uma arma. No método `getForcaDeAtaque`, o bônus da arma que ela está usando é adicionado ao dano básico da personagem. O mesmo é válido para o método `getVelocidade`:

```
// /src/main/java/decorator/Personagem.java
public class Personagem {

    private final int danoBase;
    private final int velocidade;
    private Arma armaEquipada;

    public Personagem(int danoBase, int velocidade) {
        this.danoBase = danoBase;
        this.velocidade = velocidade;
    }

    public void equiparArma(Arma arma) {
        this.armaEquipada = arma;
    }

    public int getForcaDeAtaque() {
        return danoBase + armaEquipada.getDano();
```

```
    }

    public int getVelocidade() {
        return velocidade + armaEquipada.getBonusVelocidade();
    }
}
```

A `Adaga` também possui uma implementação bem simples: ela apenas define os métodos e retorna os números dos modificadores da adaga.

```
// /src/main/java/decorator/Adaga.java
public class Adaga implements Arma {
    @Override
    public int getDano() {
        return 10;
    }

    @Override
    public int getBonusVelocidade() {
        return 3;
    }
}
```

A interface `Arma` define os métodos `getDano` e `getBonusVelocidade` que devem ser implementados por suas classes.

```
// /src/main/java/decorator/Arma.java
public interface Arma {
    int getDano();

    int getBonusVelocidade();
}
```

A implementação para outros tipos de armas seria bem parecida: basta seguir a mesma interface `Arma`, que os objetos `Personagem` poderão utilizá-las. Tarso se inspirou em outros jogos nos quais as armas das personagens evoluem junto com elas, e quer muito adicionar várias possibilidades de melhorias nas

armas.

A `Adaga`, por exemplo, poderia ser encantada, virando uma `AdagaMagica` e oferecendo mais bônus de velocidade e dado:

```
// /src/main/java/decorator/AdagaMagica.java
public class AdagaMagica implements Arma {
    @Override
    public int getDano() {
        return 15;
    }

    @Override
    public int getBonusVelocidade() {
        return 7;
    }
}
```

O real problema vem quando as melhorias nas armas entram em cena. Conforme novas melhorias para elas vão aparecendo, fica impraticável criar novas classes para cada uma das possibilidades. Se quisermos adicionar a propriedade "flamejante" para as armas, precisaríamos de uma `AdagaFlamejante` e de uma `AdagaMagicaFlamejante`.

O mesmo problema seria válido para novas armas. Uma `Espada`, por exemplo, também precisaria ser `EspadaFlamejante`, `EspadaMagicaFlamejante` ou `EspadaMagica`.

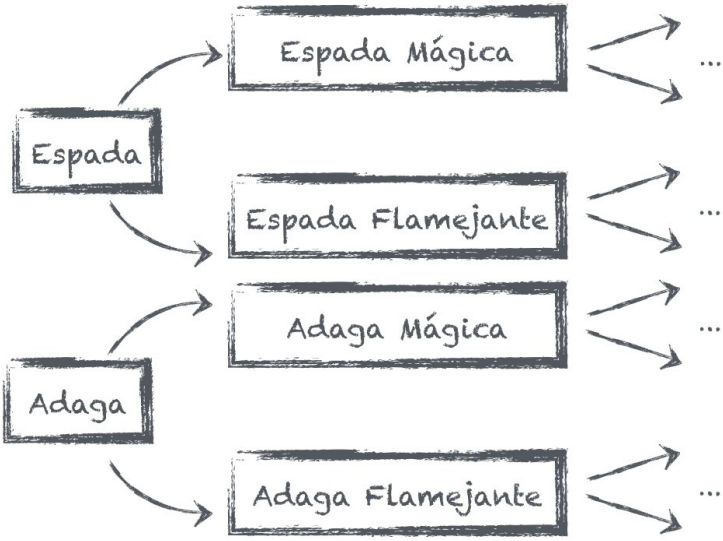

Figura 10.1: Possibilidades de evolução das armas

Para adicionar uma melhoria que aumenta a velocidade de ataque, seriam criadas mais quatro novas classes: `AdagaDaVelocidade`, `AdagaMagicaDaVelocidade`, `AdagaFlamejanteDaVelocidade` e `AdagaMagicaFlamejanteDaVelocidade`.

Criar um grande número de classes não é o único problema, mantê-las também vai ser bem complicado. Imagine que o bônus de dano para armas mágicas precise ser reajustado, e todas as classes que implementam armas mágicas precisarão ser alteradas. Mesmo para um projeto de fim de semana, essa não é a melhor maneira de evoluir o código.

Tarso pensa em usar módulos para resolver o problema, definindo uma arma mágica como um módulo e apenas incluindo-o nas classes de armas básicas. Entretanto, devido à natureza do

problema, é preciso um nível de interação muito grande entre as armas básicas e os encantos. É preciso que o módulo saiba sobre informações da arma básica e de outros módulos que possam estar incluídos também.

O ideal aqui seria alguma maneira de separar as armas básicas das suas melhorias, para que as bonificações sejam calculadas e aplicadas ao valor base das armas. Depois de algumas conversas com seus colegas, Tarso estuda um pouco sobre o padrão *Decorator* e descobre que é exatamente a solução que ele procura.

10.2 O PADRÃO DECORATOR

O problema que o padrão Decorator resolve é estruturar o incremento de funcionalidades de um objeto dinamicamente, por isso é classificado como um padrão estrutural. O contexto é bem importante, pois é preciso um conjunto bem definido de: objetos componentes, que possuem o comportamento básico; e objetos decoradores, que recebem um componente e incrementam seu comportamento.

Para resolver o problema, o padrão sugere que decoradores e componentes usem uma mesma interface, assim o cliente pode utilizar qualquer um. Além disso, os decoradores recebem um componente básico para que, ao executar um método, sejam chamados tanto o método do componente básico quanto a modificação do decorador.

A flexibilidade do uso de decoradores é maior ainda, pois, como eles possuem a mesma interface, é possível empilhar decoradores em um único componente. Dessa forma, o decorador

mais externo vai repassar a chamada para seu decorador interno, até finalmente chegar ao componente.

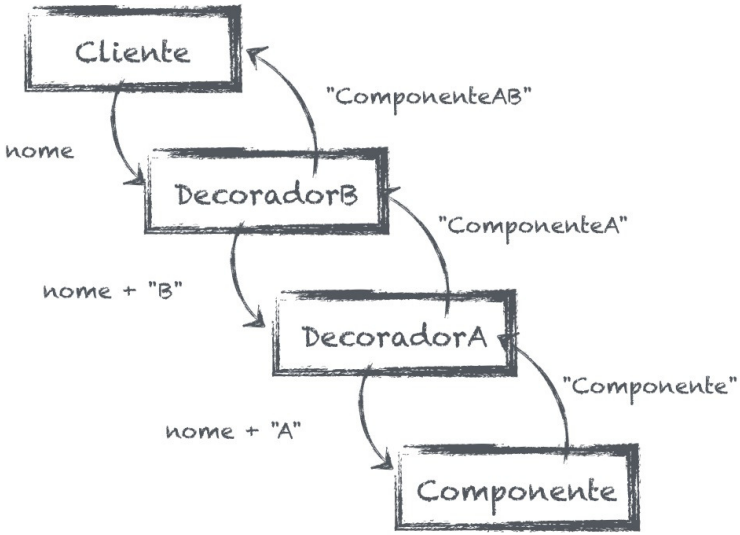

Figura 10.2: Encadeamento de chamadas

Semelhante a uma chamada recursiva, o componente retorna a chamada para um decorador, que retorna para o próximo decorador da pilha, até atingir o decorador final, no qual o cliente obterá o resultado agregado de todas as chamadas.

No problema de Tarso, nós temos as armas como componentes básicos (adagas, espadas etc.) e os encantamentos como decoradores (armas mágicas, flamejantes etc.). A nova implementação do decorador `ArmaMagica` é tão simples quanto o código já existente. A única adição é o componente básico `arma`, que será utilizado para calcular os valores:

```
// /src/main/java/decorator/ArmaMagica.java
public class ArmaMagica implements Arma {
```

```
    private Arma arma;

    public ArmaMagica(Arma arma) {
        this.arma = arma;
    }

    @Override
    public int getDano() {
        return arma.getDano() + 5;
    }

    @Override
    public int getBonusVelocidade() {
        return arma.getBonusVelocidade() + 4;
    }
}
```

Será necessário modificar um pouco os testes da classe `personagem` para criar uma adaga mágica. Ao instanciar uma `ArmaMagica`, passamos uma `Adaga`, criando a estrutura de decorador e componente básico. Agora, ao calcular a força de ataque da personagem, a `ArmaMagica` vai repassar a chamada à `Adaga`, que retorna o valor base do bônus de dano (+10). Em seguida, é aplicado o bônus da `ArmaMagica` (+5), que será somado ao dano base da personagem (5):

```
// /src/test/java/decorator/PersonagemTest.java
public class PersonagemTest {
    @Test
    public void equiparAdagaMagicaAumentaForcaDeAtaqueEm15() {
        Personagem personagem = new Personagem(5, 10);
        personagem.equiparArma(new ArmaMagica(new Adaga()));

        assertEquals(20, personagem.getForcaDeAtaque());
    }
}
```

Para definir um novo decorador – por exemplo, `Flamejante` –, basta seguir a mesma ideia de `ArmaMagica`, implementar a

interface `Arma` e definir seus próprios valores de bonificação. Instanciar uma adaga mágica flamejante da velocidade requer apenas englobar uma `Adaga` nos respectivos decoradores:

```
// Adaga Mágica Flamejante
Arma adagaMagicaFlamejante = new Flamejante(new ArmaMagica(new Ad
aga()));
```

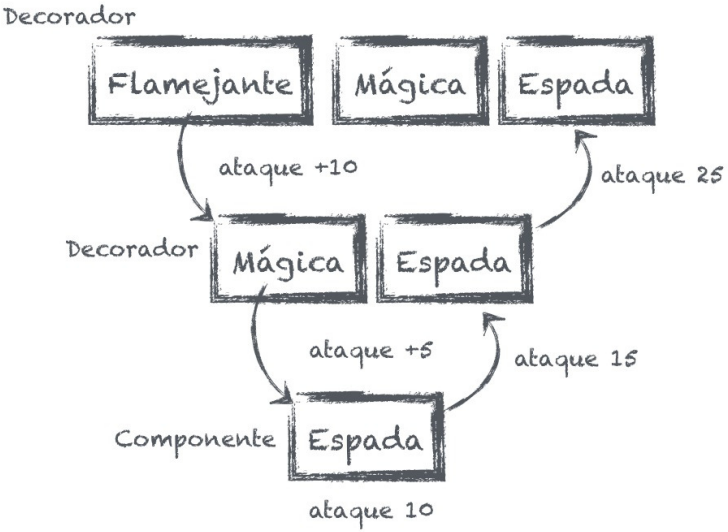

Figura 10.3: Encadeamento de chamadas

No final, Tarso precisa de apenas três classes para criar todos os diferentes tipos de adagas. Além disso, se ele quiser modificar as regras de como os encantos funcionam, a mudança ocorre apenas no respectivo decorador.

Adicionar novas armas é igualmente simples, basta seguir a mesma interface e será possível agregar todos os decoradores já existentes. Criar uma `Espada` seria apenas definir os seus valores:

```
// /src/main/java/decorator/EspadaLonga.java
```

```
public class Espada implements Arma {
    @Override
    public int getDano() {
        return 15;
    }

    @Override
    public int getBonusVelocidade() {
        return 1;
    }
}
```

Agora podemos passar um objeto `Espada` da mesma forma que um objeto `Adaga`, reutilizando os mesmos decoradores.

```
// Adaga Mágica Flamejante
Arma adagaMagicaFlamejante =
    new Flamejante(new ArmaMagica(new Espada()));
```

No final da refatoração, Tarso fica contente com o ganho na flexibilidade, mesmo sendo apenas seu projeto de final de semana. Além de ter colocado o padrão em prática, ele consegue evoluir bem melhor os sistemas de arma do jogo. Agora, resta fazer mais "pesquisas" para conseguir novas ideias para o seu jogo.

Capítulo 11

MEDIATOR: NOTIFICAÇÕES INTELIGENTES

Gil é a atual líder técnica de uma equipe que está construindo uma aplicação que monitora produtos em lojas virtuais. Uma das grandes funcionalidades do sistema é emitir alertas quando determinados eventos acontecem.

Por exemplo, uma pessoa apaixonada por jogos possui uma lista de produtos que gostaria de comprar, então, sempre que for detectada uma mudança no preço desses produtos, um e-mail será enviado para esse usuário. Ou se houver uma grande procura por determinado produto, um alerta será emitido para outro sistema, caso os estoques estejam baixos.

Enquanto pareava com um dos seus colegas de equipe, Gil notou um problema que começava a aumentar. A grande variedade de fontes de alertas e os possíveis destinos que eles teriam tornariam o código de notificações cada vez mais complexo, além da mistura com regras de negócio importantes.

11.1 O ESPAGUETE DE NOTIFICAÇÕES

O problema que Gil notou pode ser visto mais claramente no código do `BuscaPromocaoWorker`, que busca no banco de dados por uma lista de produtos e verifica se houve alguma mudança nos seus preços. Caso alguma mudança seja detectada, serão enviadas notificações para os usuários que têm esse produto na sua lista de favoritos (por meio de e-mail) e também para o fornecedor do produto (uma chamada REST), avisando sobre uma possível alta na demanda pelo produto.

A implementação de `BuscaPromocaoWorker.executar` busca no banco de dados a lista de produtos com preços baixos (`getProdutosPromocionais`), e faz uma interseção com a lista de produtos de interesse de um determinado usuário (`usuario.getProdutosDeInteresse`). Assim, ele obtém a lista de produtos para emitir alertas, utilizando o `NotificadorCliente` e o `NotificadorFornecedor`.

```
// /src/main/java/mediator/BuscaPromocaoWorker.java
public class BuscaPromocaoWorker {

    public void executar(Usuario usuario,
        List<Produto> produtosPromocionais) {

        List<Produto> produtosDeInteresse =
            usuario.getProdutosDeInteresse();

        List<Produto> produtos = produtosDeInteresse.stream()
                .filter(produtosPromocionais::contains)
                .collect(Collectors.toList());

        notificadorCliente.produtosEmPromocao(usuario, produtos);
        notificadorFornecedor.produtosEmPromocao(produtos);
        atualizarNotificacaoDeUsuario();
    }
}
```

Para separar as responsabilidades, foram criadas classes bem

específicas de notificação. Dessa forma, os testes unitários não precisam se preocupar com o que cada notificador vai fazer.

```
// /src/test/java/mediator/BuscaPromocaoWorkerTest.java
public class BuscaPromocaoWorkerTest {

    @Test
    public void deveNotificarUsuarioProdutosEmPromocao() {
        List<Produto> produtos = Arrays.asList(
            new Produto("Super Mario Brothers"),
            new Produto("USB Controller"));
        Usuario usuario = new Usuario();
        usuario.setProdutosDeInteresse(produtos);

        NotificadorCliente notificadorCliente =
            mock(NotificadorCliente.class);
        NotificadorFornecedor notificadorFornecedor =
            mock(NotificadorFornecedor.class);
        BuscaPromocaoWorker worker =
            new BuscaPromocaoWorker(
                notificadorCliente, notificadorFornecedor);

        worker.executar(usuario, produtos);

        verify(notificadorCliente)
            .produtosEmPromocao(usuario, produtos);
        verify(notificadorFornecedor)
            .produtosEmPromocao(produtos);
    }
}
```

Esse *worker* é apenas um dos locais onde os notificadores são usados, as fontes de onde as notificações podem ser transmitidas são bem variadas. Sem falar que as notificações possuem vários fins, desde e-mail para usuários até chamadas REST para outras aplicações.

Contudo, o problema que Gil vê com esse código é que a lógica de negócio precisa saber quais notificadores acionar. Com a evolução dessa aplicação, os objetos se relacionarão de uma

maneira "muitos para muitos", com várias fontes de notificação acionando vários notificadores.

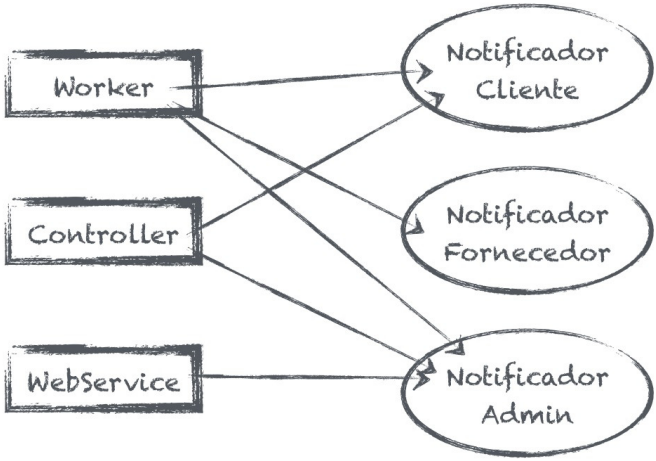

Figura 11.1: Fontes de notificação e notificadores

Ao modelar bancos de dados relacionais, quando existe uma relação "muitos para muitos" entre duas tabelas, um padrão que Gil conhece é criar uma tabela de associação. Essa tabela mantém referências para as outras duas, e cada uma referencia apenas a tabela de associação em vez de todas as entradas da outra.

Para modelagem orientada a objetos, podemos usar o mesmo padrão e criar uma classe para intermediar um relacionamento complicado. Esse padrão é conhecido como Mediator.

11.2 O PADRÃO MEDIATOR

O problema que o padrão busca resolver é simplificar a forma como um conjunto de objetos interage, por isso é classificado

como um padrão de comportamento. Assim, evitamos um acoplamento forte com objetos referenciando diretamente uns aos outros.

Para utilizar bem o padrão, é necessário que o relacionamento entre os grupos de objetos seja bem claro e possa ser extraído. A solução proposta é criar um novo objeto específico para intermediar a comunicação, o mediador. Assim, em vez de referenciar vários objetos ao mesmo tempo, basta usar o mediador.

Uma metáfora para explicar bem o Mediator é pensar em uma central de atendimento telefônico. Ao ligar para uma empresa, você não precisa saber exatamente qual o telefone para o setor de vendas ou atendimento ao consumidor. Você só precisa de um único número e, ao ligar para ele, será redirecionado para o setor de seu interesse.

Para resolver o problema de Gil, precisamos encapsular a maneira como as notificações são enviadas. Assim, as fontes das notificações não precisam se preocupar com quais notificadores chamar, apenas que uma notificação precisa ser enviada.

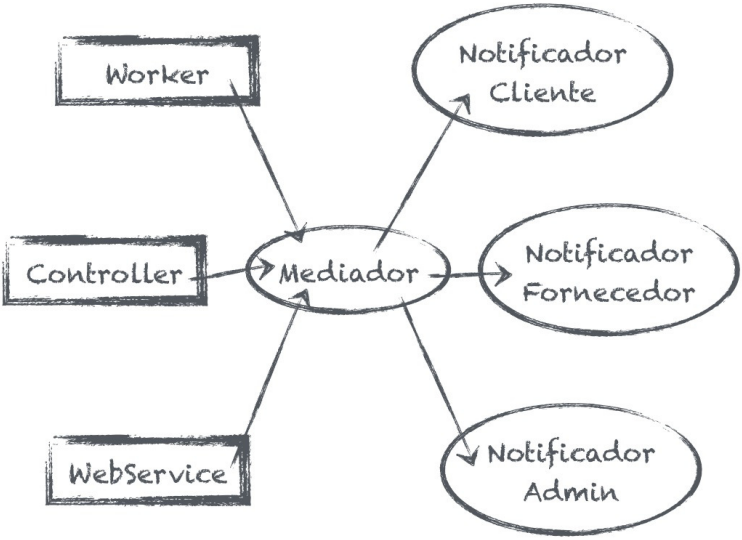

Figura 11.2: Mediador simplificando o relacionamento

O primeiro passo é aplicar **Extrair Método** na classe `BuscaPromocaoWorker` para separar a lógica de notificação da lógica de negócio. Para isso, Gil cria o novo método `notificarProdutosEmPromocao`, e move a chamada do `notificadorCliente` e `notificadorFornecedor` para lá.

```
// /src/main/java/mediator/BuscaPromocaoWorker.java
public class BuscaPromocaoWorker {

    private void notificarProdutosEmPromocao(
        Usuario usuario, List<Produto> produtos) {

        notificadorCliente.produtosEmPromocao(usuario, produtos);
        notificadorFornecedor.produtosEmPromocao(produtos);
    }

}
```

Após essa refatoração, fica bem mais claro o que é necessário

na classe `NotificadorMediator`. Todos os eventos de notificação devem estar implementados na classe mediadora, e dentro desses métodos é que vamos chamar os notificadores específicos.

Então, com o comportamento esperado da classe mediadora, podemos escrever testes verificando que o `NotificadorCliente` e o `NotificadorFornecedor` sejam chamados ao chamar `NotificadorMediator.produtosEmPromocao`:

```
// /src/test/java/mediator/NotificadorMediatorTest.java
public class NotificadorMediatorTest {

    @Test
    public void notificaProdutosEmPromocao() throws Exception {
        List<Produto> produtos = Arrays.asList(
                new Produto("Super Maria Sisters"),
                new Produto("USB Controller"));
        Usuario usuario = new Usuario();
        usuario.setProdutosDeInteresse(produtos);

        NotificadorCliente notificadorCliente =
            mock(NotificadorCliente.class);
        NotificadorFornecedor notificadorFornecedor =
            mock(NotificadorFornecedor.class);
        NotificadorMediator mediator =
            new NotificadorMediator(
                notificadorCliente, notificadorFornecedor);

        mediator.produtosEmPromocao(usuario, produtos);

        verify(notificadorCliente)
            .produtosEmPromocao(usuario, produtos);
        verify(notificadorFornecedor)
            .produtosEmPromocao(produtos);
    }
}
```

A implementação do mediador é bem simples, basta delegar a chamada dos métodos aos notificadores a quem ela interessar. Para isso, vamos usar **Extrair Classe** em `BuscaPromocaoWorker` e

mover a responsabilidade de notificação para o novo `NotificadorMediator`:

```
// /src/main/java/mediator/NotificadorMediator.java
public class NotificadorMediator {
    private NotificadorCliente notificadorCliente;
    private NotificadorFornecedor notificadorFornecedor;

    public void produtosEmPromocao(
        Usuario usuario, List<Produto> produtos) {

        notificadorCliente.produtosEmPromocao(usuario, produtos);
        notificadorFornecedor.produtosEmPromocao(produtos);
    }
}
```

Agora, na classe `BuscaPromocaoWorker`, Gil precisa apenas usar o mediador. A melhora na simplicidade do método já fica visível nos testes da classe, que agora só precisam verificar que o método `produtosEmPromocao` no `NotificadorMediator` foi chamado em vez de verificar cada um dos possíveis notificadores.

```
// /src/test/java/mediator/BuscaPromocaoWorkerTest.java
public class BuscaPromocaoWorkerTest {

  @Test
  public void deveNotificarUsuarioProdutosEmPromocao() {
    List<Produto> produtos = Arrays.asList(
                      new Produto("Super Maria Sisters"),
                      new Produto("USB Controller"));

    Usuario usuario = new Usuario();
    usuario.setProdutosDeInteresse(produtos);

    NotificadorMediator notificador =
      mock(NotificadorMediator.class);
    BuscaPromocaoWorker worker =
      new BuscaPromocaoWorker(notificador);

    worker.executar(usuario, produtos);

    verify(notificador).produtosEmPromocao(usuario, produtos);
```

```
  }
}
```

A implementação final do `BuscaPromocaoWorker` fica simplificada e apenas delega o envio de notificações para `NotificadorMediator`.

```
// /src/main/java/mediator/BuscaPromocaoWorker.java
public class BuscaPromocaoWorker {

    private NotificadorMediator notificador;

    public void executar(
        Usuario usuario, List<Produto> produtosPromocionais) {

        List<Produto> produtosDeInteresse =
            usuario.getProdutosDeInteresse();

        List<Produto> produtos = produtosDeInteresse.stream()
                .filter(produtosPromocionais::contains)
                .collect(Collectors.toList());

        notificador.produtosEmPromocao(usuario, produtos);
        atualizarNotificacaoDeUsuario();
    }
}
```

Gil olha para o código e também percebe o quão fácil será adicionar novos notificadores a essa estrutura. Caso a notificação já exista, basta adicionar uma nova classe que implemente o mesmo método, e adicioná-la à lista de classes do mediador. Por exemplo, para alertar o time de uma possível alta no tráfego do site quando um produto entra em promoção:

```
public class NotificadorSite {
    public void produtosEmPromocao(List<Produto> produtos) { }
}
```

Agora basta chamar o novo notificador no Mediator:

```
// /src/main/java/mediator/NotificadorMediator.java
```

```
public class NotificadorMediator {

    public void produtosEmPromocao(
        Usuario usuario, List<Produto> produtos) {

        notificadorCliente.produtosEmPromocao(usuario, produtos);
        notificadorFornecedor.produtosEmPromocao(produtos);
        notificadorSite.produtosEmPromocao(produtos);
    }
}
```

Se for preciso adicionar uma notificação completamente nova, basta criar o novo método no mediador e usá-lo nas classes que devem disparar a notificação. Um exemplo seria adicionar um alerta para fornecedores sobre produtos com pouco estoque. O primeiro passo é definir o novo alerta `produtosEmBaixa` em `NotificadorFornecedor`.

```
// /src/main/java/mediator/NotificadorFornecedor.java
public class NotificadorFornecedor {
    public void produtosEmBaixa(List<Produto> produtos) { }
}
```

Em seguida, basta criar o novo método `produtosEmBaixa` em `NotificadorMediator`, que vai apenas delegar a chamada para os notificadores.

```
// /src/main/java/mediator/NotificadorMediator.java
public class NotificadorMediator {

    public void produtosEmBaixa(List<Produto> produtos) {
        notificadorFornecedor.produtosEmBaixa(produtos);
    }
}
```

Por fim, na fonte de alertas, basta chamar o `NotificadorMediator`, que ele cuida do resto.

```
public class EstoqueBaixoWorker {
  public void executar() {
```

```
    List<Produto> produtosprodutos = produtosComBaixoEstoque();
    notificadorMediator.produtosEmBaixa(produtos);
  }
}
```

Gil apresenta o código para a sua equipe, e todos gostam da facilidade de extensão e de como as responsabilidades ficaram bem separadas. Agora, com essa dívida técnica a menos, a equipe consegue implementar notificações de maneira bem mais rápida.

Conclusão

Capítulo 12

OS OUTROS PADRÕES

Ao longo do livro, exploramos apenas nove padrões da lista de 23 catalogados pela Gangue dos Quatro (Erich Gama, Ralph Johnson, Richard Helm e John Vlissides). Isso não quer dizer que os outros devam ser ignorados, apenas que, como o foco deste livro é a utilização de padrões no processo de refatoração, decidi reduzir o conjunto de padrões a serem explorados e focar em suas aplicações.

Alguns dos padrões não entraram no livro, pois seu uso é muito específico, como o *Chain of Responsibility*, e até mesmo pensar em um exemplo real não é tão simples. Outros padrões, como o *Facade*, não são tão comuns, mas são geralmente mal interpretados e utilizados. E, finalmente, padrões como *Singleton* são usualmente contestados devido aos problemas que eles resolvem.

12.1 PADRÕES POUCO UTILIZADOS

Vimos que alguns padrões podem ser simplificados, mas existe um outro conjunto de padrões que foi deixado de lado e não foi apresentado no livro pois sua aplicação é muito rara.

Um bom exemplo, como dito, é o padrão *Chain of*

Responsibility, que propõe desacoplar a chamada do objeto do código do cliente para que mais de um objeto consiga tratar a solicitação. A solução proposta é que os objetos receptores sejam encadeados para que as chamadas sejam passadas ao longo da cadeia.

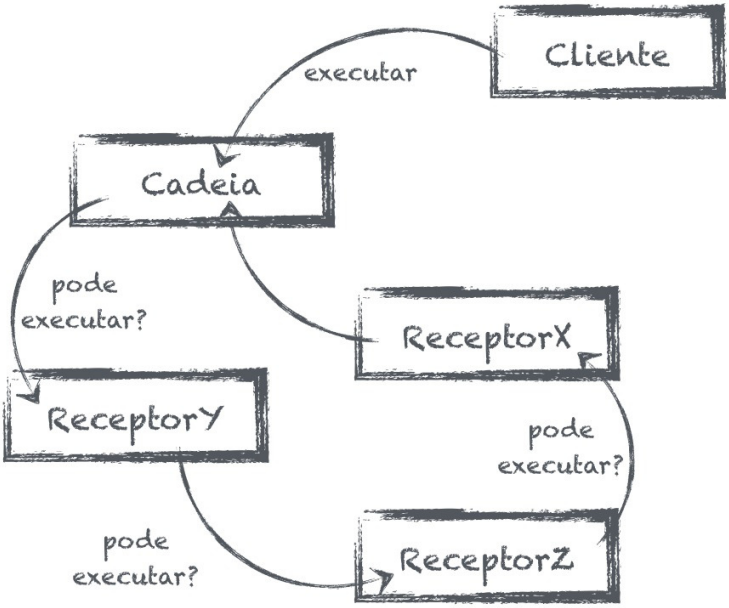

Figura 12.1: Utilização do padrão Chain of Responsibility

Ter múltiplos objetos que devem responder a uma mesma solicitação talvez seja um problema real que você tem, mas as chances são pequenas. Mesmo que esse seja o caso, será que o problema não pode ser resolvido de uma maneira mais simples? Lembre-se de que, no começo do livro, conversamos sobre procurar soluções mais simples antes de aplicar qualquer padrão, devido à complexidade e aos níveis de abstrações que os padrões trazem.

Até mesmo os padrões detalhados no livro devem ser cuidadosamente aplicados. Uma simplificação no design é sempre melhor do que qualquer aplicação de padrão de projeto.

12.2 PADRÕES MAL UTILIZADOS

Existem alguns padrões que são considerados por muitos como antipadrões, ou seja, soluções que foram usadas e que não melhoram o design da aplicação. O padrão *Facade* é um bom exemplo para demonstrar isso.

O problema que ele busca resolver é quando existe um conjunto de componentes distintos que precisam funcionar juntos. A solução é definir uma interface, em um nível mais alto, que facilite a utilização de componentes de um nível menor.

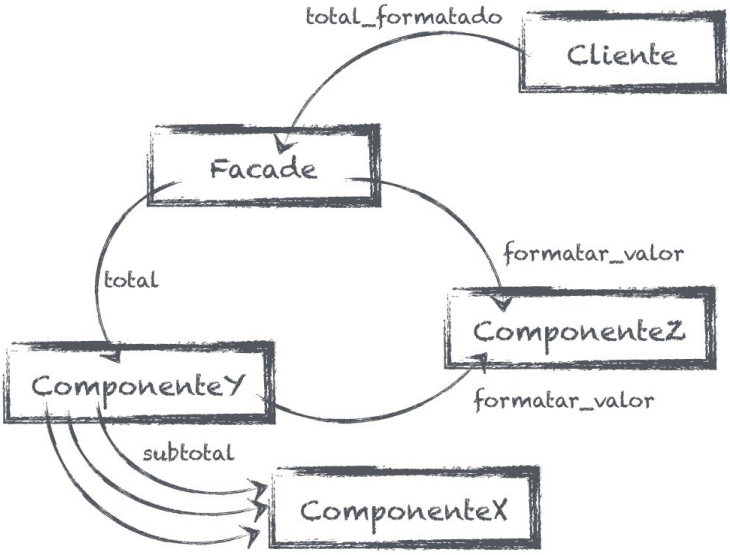

Figura 12.2: Utilização do padrão Facade

Esse parece um problema bem comum. Então, por que não usar o padrão Facade?

O padrão cria uma maneira mais fácil de usar um código que seria uma bagunça completa. Mas facilitar o acesso não resolve a bagunça completa!

É bem fácil olhar para o código do cliente e, vendo apenas uma chamada sendo feita, achar o código bonito, simples e claro. Mas, quando for preciso alterar esse código, a bagunça continuará lá.

Jogar a sujeira para debaixo do Facade não vai trazer nenhuma melhoria para sua aplicação. O problema não é com o padrão em si, mas com a aplicação fora de contexto.

Dado um conjunto de componentes que devem funcionar juntos e estão bem organizados, então o Facade facilita a utilização, provendo pontos mais simples. Outra vantagem dele é que alterações nos componentes internos não se propagam até o código do cliente, facilitando a manutenção.

12.3 PADRÕES QUE NINGUÉM DEVERIA UTILIZAR

Ainda existe outro conjunto de padrões também considerado antipadrões, pois o problema que eles resolvem não deveria ser resolvido! Vejamos como exemplo o padrão Singleton, cuja ideia é garantir que uma classe tenha apenas uma instância em toda a aplicação. No entanto, ter um acesso global a uma variável tende a ser um mau sinal. Em vez de criar uma boa maneira de acessá-la, seria melhor procurar uma maneira de não ter uma variável global.

O padrão Singleton resolve o problema fornecendo um único ponto de acesso a uma variável global, deixando seu acesso um pouco mais controlado.

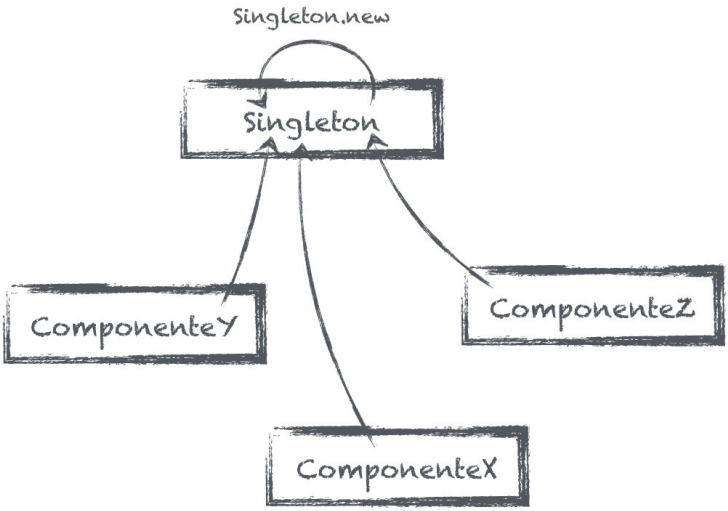

Figura 12.3: Utilização do padrão Singleton

O problema é que fornecer um ponto de acesso global para uma única instância é basicamente criar uma instância global. Criar variáveis globais é um grande transtorno pois, dado um momento específico da aplicação, é muito difícil ter certeza do estado dessa variável global (já que ela pode ser alterada em qualquer lugar). O código que usa uma instância global precisa sempre ficar se perguntando o estado dessa instância antes de tomar ações.

Isso não quer dizer que usar o Singleton seja ruim. Caso você realmente precise de uma instância global, o padrão propõe uma boa solução. Mas ter e acessar uma instância global não deveriam

ser um problema no seu código.

Capítulo 13
CONCLUSÃO

Parabéns por ter chegado até o final do livro! Espero que você tenha gostado de acompanhar a jornada dos nossos heróis enquanto utilizavam padrões de projeto para refatorar seus códigos. Mas, principalmente, espero que você tenha conseguido usar as técnicas apresentadas no seu código atual.

Agora que você já conhece vários padrões e sabe como aplicá-los, é preciso ter cuidado para não sair aplicando-os apenas por aplicar. Como já falado no começo do livro, é preciso considerar o contexto antes de aplicar um padrão para garantir que o código vai realmente melhorar.

13.1 DESIGN EVOLUCIONÁRIO

A discussão sobre planejar o design no começo do projeto, ou deixar que ele evolua a partir das necessidades de crescimento da aplicação, não é nova. No artigo *Is Design Dead* (http://martinfowler.com/articles/designDead.html), Martin Fowler mostra como as práticas do *Extreme Programming* (XP) permitem que a evolução do design seja viável. O ponto principal do artigo é a importância de deixar o código o mais simples possível.

Segundo Kent Beck (*Extreme Programming Explained: embrace change*, 1999), simplicidade de código pode ser definida a partir das seguintes características:

- Todos os testes passam;
- Sem duplicação;
- Mostra as intenções;
- Possui o menor número de classes ou métodos.

Ao aplicar padrões de projeto, não necessariamente estamos simplificando o código. Com certeza diminuímos duplicações ao refatorá-lo e ao separar as responsabilidades. Ao separar responsabilidades, também deixamos as intenções de cada parte dele mais claras e fáceis de serem usadas. No entanto, aumentamos a quantidade de classes e métodos.

Em teoria, ao tentar deixar o código simples seguindo as ideias apresentadas por Kent Beck, o design da aplicação vai evoluir naturalmente para o uso de padrões. No entanto, Martin Fowler sugere que a utilização de padrões seja mais consciente, aprendendo sobre os padrões de projetos e tendo uma ideia de para onde o design está indo.

Ainda no mesmo artigo, Martin Fowler sugere os seguintes pontos para aproveitar ao máximo padrões de projeto, que resumem perfeitamente o que eu acho que devemos ter em mente ao aprender sobre eles:

- Invista tempo aprendendo sobre padrões;
- Concentre-se em aprender quando aplicá-los (não muito cedo);
- Concentre-se em como implementar padrões na sua forma

mais simples, e adicionar complexidade depois;

- Se você adicionar um padrão e depois perceber que ele não está ajudando, não tenha medo de removê-lo.