Capítulo 3

Tópicos de Orientação a Objetos

Um bom design de software visa a uma arquitetura flexível que permita futuras alterações, facilite a produção de código organizado e legível, maximizando seu reaproveitamento. Todo o paradigma da orientação a objetos, seus princípios e boas práticas procuram trazer esses benefícios para o design.

Ao pensar no sistema como um todo, outras questões de mais alto nível surgem, em especial aquelas que tratam da forma como os objetos se relacionam e sua organização dentro e entre sistemas. Ao relacionar dois objetos distintos, deve-se levar em conta as boas práticas que serão discutidas nesse capítulo, como a diminuição do acoplamento entre objetos.

3.1. Programe voltado à interface, não à implementação

Ao trabalhar com coleções, escolher a implementação certa para cada caso é uma tarefa difícil. Cada uma delas, como `ArrayList`, `LinkedList` ou `HashSet`, é melhor para resolver determinas categorias de problemas. Pode ser muito arriscado

escrever todo o código da aplicação dependente de uma decisão antecipada. Apesar disso, grande parte dos desenvolvedores opta por sempre utilizar `ArrayList` desde o início sem critério algum.

Considere um DAO de funcionários que pode listar o nome de todos que trabalham em determinado turno, devolvendo um `ArrayList` com os nomes:

```
public class FuncionarioDao {
   public ArrayList<String> buscaPorTurno(Turno turno) { ... }
}
```

E um código que precisa saber se determinado funcionário esteve presente, efetuando uma busca simples na lista devolvida:

```
FuncionarioDao dao = new FuncionarioDao();
ArrayList<String> nomes = dao.buscaPorTurno(Turno.NOITE);

boolean presente = nomes.contains("Anton Tcheckov");
```

Mas a busca com `contains` em um `ArrayList` é, em termos computacionais, bastante custosa. Poderíamos utilizar outras alternativas de coleção, trocando o retorno de `ArrayList` para `HashSet`, por exemplo, pois sua operação `contains` é muito mais eficiente computacionalmente, usando internamente uma tabela de espalhamento (*hash table*). Para poucos funcionários, a diferença é imperceptível, porém, à medida que a lista aumenta, a diferença de desempenho entre um `ArrayList` e um `HashSet` se torna mais clara, e até mesmo um gargalo de performance.

O problema em realizar uma mudança como esta, de implementação, é que todo código que usava o retorno do método como `ArrayList` quebra, mesmo que só usássemos métodos que também existem definidos em `HashSet`. Seria preciso alterar todos os lugares que dependem de alguma forma desse método. Além de trabalhoso, tarefas do tipo *search/replace* são um forte sinal de código ruim.

Há esse acoplamento sintático com a assinatura do método, que conseguimos resolver olhando os erros do compilador. Mas sempre há também informações semânticas implícitas na utilização desse método, e que não são expostos através da assinatura. Um exemplo de acoplamento semântico está em depender da informação de que uma `List` permite dados duplicados, enquanto um `Set` garante unicidade dos elementos. Como problemas no acoplamento sintático são encontrados em tempo de compilação, os semânticos somente o são em

execução, daí um motivo da importância de testes que garantam o comportamento esperado.

O grande erro do método `buscaPorTurno` da classe `FuncionarioDao` foi atrelar todos os usuários do método a uma implementação específica de `Collection`. Desta forma, alterar a implementação torna-se sempre muito mais custoso, caracterizando o **alto acoplamento** que tanto se procura evitar.

Para minimizar esse problema, é possível usar um tipo de retorno de método mais genérico, que contemple diversas implementações possíveis, fazendo com que os usuários do método não dependam em nada de uma implementação específica. A interface `Collection` é uma boa candidata:

```
public class FuncionarioDao {
   public Collection<String> buscaPorTurno(Turno turno) { ... }
}
```

Com o método desta forma, podemos trocar a implementação retornada sem receio de quebrar nenhum código que esteja invocando `buscaPorTurno`, já que ninguém depende de uma implementação específica. Usar *interfaces* Java é um grande benefício nestes casos, pois ajuda a garantir que nenhum código dependa de uma implementação específica, pois interfaces não carregam nenhum detalhe de implementação.

Repare que é possível optar ainda por outras interfaces, como `List` (mais específica) e `Iterable` (menos específica). A escolha da interface ideal vai depender do que você quer permitir que o código invocador possa utilizar e realizar na referência retornada. Quanto menos específica, menor o acoplamento e mais possibilidades de diferentes implementações. Em contrapartida, o código cliente tem uma gama menor de métodos que podem ser invocados.

Algo similar também ocorre para receber argumentos. Considere um método que grava diversos funcionários em lote no nosso DAO:

```
public class FuncionarioDao {
   public void gravaEmLote(ArrayList<Funcionario> funcionarios) { ... }
}
```

Receber precisamente `ArrayList` como argumento tem pouca utilidade; raramente necessitamos que uma coleção seja tão específica assim. Receber aqui um `List` provavelmente baste para o nosso método, e permite que o código invocador passe outras coleções como argumento. Podemos ir além e receber `Collection`

ou, ainda, `Iterable`, caso nossa necessidade seja apenas percorrer os elementos. A escolha de `Iterable`, neste caso, permitiria o maior desacoplamento possível, mas limitaria o uso dentro do método; não seria possível, por exemplo, acessar a quantidade de elementos que essa coleção possui, nem os elementos de maneira aleatória através de um índice. Devemos procurar um balanço entre o desacoplamento e a necessidade do nosso código. Esta é a ideia do *Princípio de Segregação de Interfaces*: **clientes não devem ser forçados a depender de interfaces que não usam**.[1]

O desenvolvedor deve ter em mente que acoplar uma classe, que possui menos chances de alterações em sua estrutura, com outra menos *estável* pode ser perigoso.[2] Considere a interface `List`, que possui muitas razões para não mudar, afinal, ela é implementada por várias outras classes; se sofresse alterações, todas as classes que a implementam teriam que ser alteradas também. Consideramos, então, que ela é altamente estável, o que significa que ela raramente obrigará uma mudança nas classes que a utilizam (Figura 3.1).

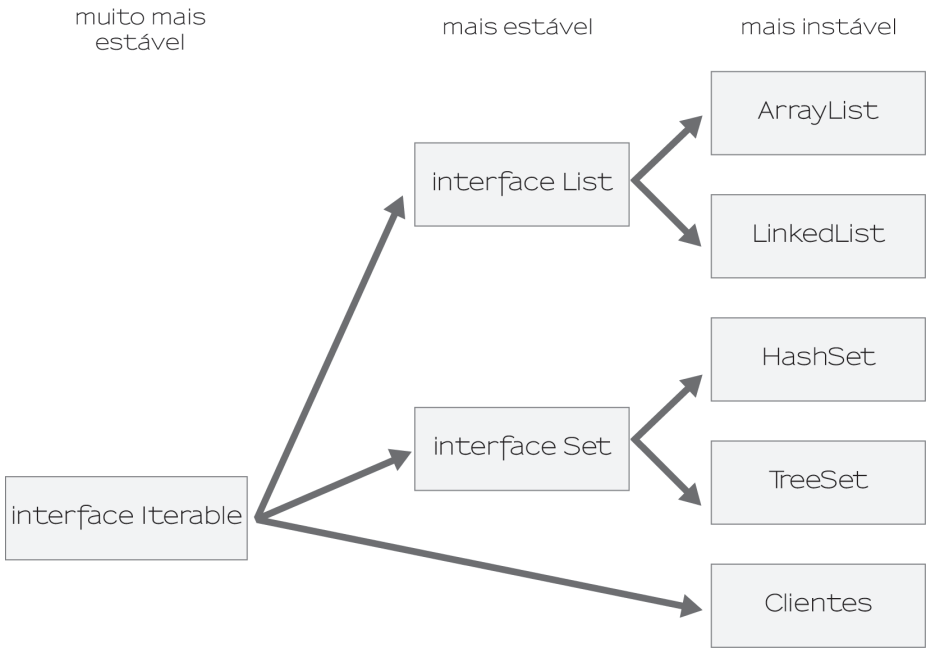

Figura 3.1 – Interfaces são mais estáveis por garantirem menores mudanças com quebra de compatibilidade.

Uma implementação de lista, `MeuProprioArrayList`, feita pelo desenvolvedor é provavelmente mais *instável* que a interface `List`, já que as forças que a im-

pedem de mudar são fracas (não há outras classes utilizando essa implementação). Ou seja, uma classe acoplada a essa implementação de lista eventualmente pode ser obrigada a mudar por causa de alguma alteração em `MeuProprioArrayList`.

Os frameworks e bibliotecas consagrados sempre tiram proveito do uso de interfaces, desacoplando-se o máximo possível de implementações. Tanto a `Session` do Hibernate quanto a `EntityManager` da JPA devolvem `List` nos métodos que envolvem listas de resultados. Ao analisar a fundo as implementações atuais de `Session` e `EntityManager` do Hibernate, elas não retornam nem `ArrayList`, nem `LinkedList`, nem nenhuma coleção do `java.util`, e, sim, implementações de listas persistentes de pacotes do próprio Hibernate.

Isto é possível, novamente, pelo desacoplamento provido pelo uso das interfaces. Além disso, o retorno das consultas com JPA e Hibernate são `List`, para deixar claro ao usuário que a ordem é importante. Manter a ordem de inserção e permitir acesso aleatório são características do contrato de `List` e são importantes para o resultado de consultas, pois podem definir uma ordenação (`order by`), uma ótima justificativa para optar por uma interface mais específica, e não usar `Iterable` ou `Collection`.

Nas principais APIs do Java, é fundamental programar voltado à interface. Ao usar `java.io`, evitamos ao máximo nos referenciar a `FileInputStream`, `SocketInputStream`, entre outras. O código a seguir aceita apenas arquivos como streams:

```
public class ImportadoraDeDados {
   public void carrega(FileInputStream stream) { ... }
}
```

Desta forma, não é possível passar qualquer tipo de `InputStream` para a `ImportadoraDeDados`. Adotar esta limitação depende do código dentro do método `carrega`. Ao utilizar algum método específico de `FileInputStream` que não esteja definido em `InputStream`, não há o que fazer para desacoplar o código. Caso contrário, esse método poderia, e deveria, receber uma referência a `InputStream`, ficando mais flexível e podendo receber os mais diferentes tipos de streams, como argumento, que provavelmente não foram previamente imaginados. **Utilize sempre o tipo menos específico possível.**

Repare que, muitas vezes, classes abstratas trabalham como interfaces, no sentido conceitual de orientação a objetos.[3] Classes abstratas possuem a vantagem de se poder adicionar-lhes um novo método não abstrato, sem quebrar o código

já existente. Já com o uso de *interfaces* (aqui, pensando na palavra-chave do Java), a adição de qualquer método acarretará a quebra das classes que a implementam. Como interfaces nunca definem nenhuma implementação, a vantagem é que o código está sempre desacoplado de qualquer outro que as utilize. Isso muda com os polêmicos *extension methods* do Java 8, permitindo escrever uma implementação padrão nas interfaces, possibilitando suas evoluções, ao mesmo tempo que minimiza a quebra de compatibilidade.

Os métodos `load(InputStream)`, da classe `Properties`, e `fromXML(InputStream)`, do `XStream`, são ótimos exemplos de código que não dependem de implementação. Podem receber arquivos dos mais diferentes streams: rede (`SocketInputStream`), upload HTTP (`ServletInputStream`), arquivos (`FileInputStream`), de dentro de JARs (`JarInputStream`) e arrays de byte genéricos (`ByteArrayInputStream`).

A *Java Database Conectivity* (JDBC) é outra API firmemente fundada no uso de interfaces. O pacote `java.sql` possui pouquíssimas classes concretas. Sempre que encontramos um código trabalhando com conexões de banco de dados, vemos referências à interface `Connection`, e nunca diretamente a `MySQLConnection`, `OracleConnection`, `PostGreSQLConnection`, ou qualquer outra implementação de um driver específico, apesar de esta possibilidade existir.

Referindo-se sempre a `Connection`, deixamos a escolha da implementação centralizada em um ou poucos locais. Desta forma, fica muito fácil trocar a implementação sem modificar todo o restante do código. Isso ocorre graças ao desacoplamento provido pelo uso de interfaces.

No caso do JDBC, essa escolha por uma implementação está centralizada na classe concreta `DriverManager`, que aqui age como uma factory. Ela decide por instanciar uma implementação específica de `Connection` de acordo com os parâmetros passados como argumentos ao método `getConnection` e conforme os possíveis drivers previamente carregados.

```
Connection connection =
   DriverManager.getConnection("jdbc:mysql://192.168.0.33/banco");
```

Pode ser fácil enxergar as vantagens do uso das interfaces, mas é bem mais difícil começar a utilizá-las extensivamente no seu próprio domínio. O uso exagerado de reflection para invocar métodos dependendo de algumas condições pode ser muitas vezes substituído por interfaces. Assim, a decisão de qual método invocar é deixada para a invocação virtual de método que o polimorfismo promove,

diminuindo bastante a complexidade e aumentando a manutenibilidade, além de algum ganho de performance.[4]

Isto é ainda mais gritante com o uso da instrução `switch`, ou mesmo em um excessivo número de `ifs` encadeados; essa abordagem pode acoplar totalmente seu modelo, tornando necessárias mudanças frequentes nele toda vez que uma nova entidade for adicionada ao domínio.[5]

Programe voltado à interface, não à implementação é outro dos príncipios de orientação a objetos do livro *Design Patterns*,[6,3] abordado por meio de outros exemplos no seminal *Dependency Inversion Principle*, de Bob Martin.[7]

3.2. Componha comportamentos

Um código com poucas possibilidades de fluxos lógicos (*branches* de execução), ou seja, poucos caminhos de execução, é mais fácil de entender e manter. O exemplo a seguir mostra um processo de pagamento:

```
public void processa(Pagamento aPagar) {

   if (aPagar.isServico() && aPagar.getValor() > 300) {
      impostos.retem(aPagar.getValor() * TAXA_A_RETER);
   } else if(aPagar.isProduto()) {
      estoque.diminui(aPagar.getItem());
   }

   conta.executa(aPagar);

   if (aPagar.desejaReceberConfirmacao()) {
      emails.enviaConfirmacao(aPagar);
   }
}
```

Apesar de simples, existem seis possibilidades diferentes de execução do método, além de ele misturar diversos comportamentos que não possuem relação, isto é, responsabilidades diferentes para uma única classe: impostos, estoques, transferências e e-mails.

Tal comportamento pode ser composto por diversas partes menores e, para tanto, refatorações pequenas podem ser executadas. A mais simples seria a extração de quatro métodos, uma solução que simplifica o código atual, mas não aumenta sua coesão.