Programe voltado a interface, não a implementação

O que você usa para armazenar um punhado de objetos? Um `ArrayList`? Uma `LinkedList`? Um `HashSet`? Ou você nunca parou para pensar nisso? Já se perguntou por que existem tantas opções? Já escolheu alguma dessas e teve que voltar atrás?

Escolher a implementação certa para cada caso é uma tarefa difícil, já que cada uma delas é melhor para resolver um tipo diferente de problema. Pode ser muito arriscado fazer o resto do código inteiro depender de uma decisão antecipada. Apesar disso, grande parte dos desenvolvedores opta por `ArrayList` diretamente sem nenhum critério.

Considere um DAO de funcionários que pode listar o nome de todos que trabalham em determinado turno:

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

E agora um código que precisa saber se determinado funcionário esteve presente:

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

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

Alguém poderia resolver tentar outras alternativas de coleção, trocando o retorno de `ArrayList` para `HashSet`, pois sua operação `contains()` é mais eficiente computacionalmente.

Uma analogia simples para quem já trabalhou no mundo de banco de dados relacionais e queries `SQL`, um *full table scan* que varre uma tabela inteira para encontrar determinados elementos é inferior a uma busca indexada se analisarmos a velocidade do retorno de tal pesquisa.

Para poucos funcionários, a diferença é imperceptível, porém a medida que a lista aumenta, a diferença de desempenho entre um `ArrayList` e um `HashSet` se torna mais clara até se tornar um gargalo.

O problema em realizar uma mudança como essa, de implementação, é que todo o código que usava o retorno do método como `ArrayList` quebra, mesmo se só usassemos métodos que também existem definidos em `HashSet`. Você precisaria 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 mal escrito anteriormente.

O grande erro foi atrelar todos aqueles que usam `buscaPorTurno` a uma das implementações específicas de `Collection`. Desta forma, alterar a implementação é quase sempre custoso, caracterizando o **alto acoplamento** que tanto queremos evitar.

Para evitar esse problema, você pode usar um retorno mais genérico para o método, que contemple diversas implementações possíveis e que force os usuários do método não dependerem em nada de uma implementação específica. A interface `Collection` é uma boa candidata aqui:

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

Com o método dessa 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 de certa forma garante que **nenhum código depende de uma implementação específica**, pois interfaces não carregam implementação alguma.

Repare que poderíamos 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. Quando 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 `ArrayList` como argumento faz sentido em pouquíssimos casos: raramente precisamos que uma coleção seja tão específica assim. Receber aqui um `List` provavelmente basta 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 a lista. A escolha de `Iterable` permitiria o maior desacoplamento possível nesse caso, mas limitaria nosso uso dentro do método: não poderíamos, por exemplo, remover um elemento, acessar a quantidade de elementos que essa coleção possui, nem acessar elementos de maneira aleatória através de um índice. Devemos procurar um balanço entre o desacomplamento e a necessidade do nosso código.

Os frameworks e bibliotecas consagrados sempre tiram proveito do uso de interfaces, desacomplando-se o máximo possível de implementações. Tanto a `Session` do Hibernate e a `EntityManager` da JPA devolvem `List` nos métodos que envolvem listas de resultados. Se você for ver a fundo as implementações atuais de `Session` e `EntityManager` do Hibernate, eles não retornam nem `ArrayList`, nem `LinkedList` nem nenhuma coleção do `java.util`, e sim implementações de listas persistentes próprias do Hibernate. Isso é possível novamente pelo desacomplamento provido pelo uso das interfaces.

Além disso, o resultado das consultas com JPA e Hibernate vêm em objetos `List`, para deixar claro ao usuário da biblioteca que a ordem é importante. Manter a ordem de inserção é uma das características do contrato de `List` e é importante para o resultado de consultas, já que podem definir algum tipo de 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 programarmos voltado a interface. No `java.io` evitamos ao máximo nos referenciar a `FileInputStream`, `SocketInputStream`, entre outras. O código abaixo aceita apenas arquivos como streams:

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

Dessa forma não podemos passar qualquer tipo de `InputStream` para a `ImportadoraDeDados`. Precisamos desse limitante? Isso vai depender do código dentro do método `carrega`. Se ele usa algum método específico de `FileInputStream` que não esteja definido em `InputStream`, não há o que fazer para desacoplar esse 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 podem aparecer como **interfaces** (no sentido conceitual de orientação a objetos) [63]. Classes abstratas possuem a vantagem de você poder adicionar um novo método **não abstrato** nelas, sem quebrar código já existente que herda dela. Já com o uso de `interface`s (aqui, pensando na palavra chave do Java), a adição de qualquer método acarretará na quebra das classes que a implementam. Como interfaces nunca definem nenhuma implementação, a vantagem é que o código está sempre desacoplado a qualquer outro código usando a interface.

Os métodos `load(InputStream)`, da classe `Properties`, e `fromXML(InputStream)`, do `XStream`, são ótimos exemplos de código não dependente 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`), por exemplo.

A JavaDatabase Conectivity (JDBC) é um outro excelente exemplo de API totalmente fundada em interfaces. O pacote `java.sql` possui pouquíssimas classes concretas. Sempre que encontramos um código trabalhando com conexões de banco de dados, vemos referências a interface `Connection`, e nunca diretamente a `MySQLConnection`, `OracleConnection`, `PostGreSQLConnection` ou qualquer outra implementação de um driver específico, apesar dessa possibilidade existir.

Referindo-se sempre por `Connection`, deixamos a escolha da implementação centralizada em um ou poucos locais. Desta forma fica muito fácil trocar a implementação, sem trocar 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 conhecida classe concreta `DriverManager`, que é 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");
```

O título deste tópico é o outro dos príncipios fundamentais de orientação a objetos citados no livro *Design Patterns* [14, 63].