6

Objetos e Estruturas de Dados

Há um motivo para declararmos nossas variáveis como privadas. Não queremos que ninguém dependa delas. Desejamos ter a liberdade para alterar o tipo ou a implementação, seja por capricho ou impulso. Por que, então, tantos programadores adicionam automaticamente métodos de acesso (escrita, ou *setters,* e leitura, ou *getters*) em seus objetos, expondo suas variáveis privadas como se fossem públicas?

Abstração de dados

Considere a diferença entre as listagens 6.1 e 6.2. Ambas representam os dados de um ponto no plano cartesiano. Um expõe sua implementação e o outro a esconde completamente.

Listagem 6-1
Caso concreto

```
public class Point {
  public double x;
  public double y;
}
```

Listagem 6-2
Caso abstrato

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

O belo da Listagem 6.2 é que não há como dizer se a implementação possui coordenadas retangulares ou polares. Pode não ser nenhuma! E ainda assim a interface representa de modo claro uma estrutura de dados.

Mas ela faz mais do que isso. Os métodos exigem uma regra de acesso. Você pode ler as coordenadas individuais independentemente, mas deve configurá-las juntas como uma operação atômica.

A Listagem 6.1, por outro lado, claramente está implementada em coordenadas retangulares, e nos obriga a manipulá-las independentemente. Isso expõe a implementação. De fato, ela seria exposta mesmo se as variáveis fossem privadas e estivéssemos usando métodos únicos de escrita e leitura de variáveis.

Ocultar a implementação não é só uma questão de colocar uma camada de funções entre as variáveis. É uma questão de ! Uma classe não passa suas variáveis simplesmente por meio de métodos de escrita e leitura. Em vez disso, ela expõe interfaces abstratas que permite aos usuários manipular a *essência* dos dados, sem precisar conhecer a implementação.

Considere as listagens 6.3 e 6.4. A primeira usa termos concretos para comunicar o nível de combustível de um veículo, enquanto a segunda faz o mesmo, só que usando . No caso concreto, você tem certeza de que ali estão apenas métodos acessores (escrita e leitura, ou *getter* e *setter*) de variáveis. No caso abstrato, não há como saber o tipo dos dados.

Listagem 6-3
Veículo concreto

```
public interface Vehicle {
  double getFuelTankCapacityInGallons();
  double getGallonsOfGasoline();
}
```

Listagem 6-4
Veículo abstrato

```
public interface Vehicle {
  double getPercentFuelRemaining();
}
```

Em ambos os casos acima, o segundo é preferível. Não queremos expor os detalhes de nossos dados. Queremos expressar nossos dados de forma abstrata. Isso não se consegue meramente através de interfaces e/ou métodos de escrita e leitura. É preciso pensar bastante na melhor maneira de representar os dados que um objeto contenha. A pior opção é adicionar levianamente métodos de escrita e leitura.

Anti-simetria data/objeto

Esses dois exemplos mostram a diferença entre objetos e estruturas de dados. Os objetos usam abstrações para esconder seus dados, e expõem as funções que operam em tais dados. As estruturas de dados expõem seus dados e não possuem funções significativas. Leia este parágrafo novamente.

Note a natureza complementar das duas definições. Elas são praticamente opostas. Essa diferença pode parecer trivial, mas possui grandes implicações.

Considere, por exemplo, a classe shape procedimental na Listagem 6.5. A classe Geometry opera em três classes shape que são simples estruturas de dados sem qualquer atividade. Todas as ações estão na classe Geometry.

Listagem 6-5
Classe shape procedimental

```
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}

public class Geometry {
  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuch
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
```

Listagem 6-5 (continuação)
Classe shape procedimental

```
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

Programadores de orientação a objeto talvez torçam o nariz e reclamem que isso é procedimental—e estão certos. Mas nem sempre. Imagine o que aconteceria se adicionássemos uma função `perimeter()` à `Geometry`. As classes shape não seriam afetadas! Assim como quaisquer outras classes que dependessem delas!

Por outro lado, se adicionarmos uma nova classe shape, teremos de alterar todas as funções em Geometry. Leia essa frase novamente. Note que as duas situações são completamente opostas.

Agora, considere uma solução orientada a objeto na Listagem 6.6. O método `área()` é *polifórmico*. Não é necessária a classe `Geometry`. Portanto, se eu adicionar uma nova forma, nenhuma das funções existentes serão afetadas, mas se eu adicionar uma nova função, todas as classes shape deverão ser alteradas[1].

Listagem 6-6
Classes shape polifórmicas

```
public class Square implements Shape {
  private Point topLeft;
  private double side;

  public double area() {
    return side*side;
  }
}

public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}
```

1. Desenvolvedores orientados a objeto experiente conhecem outras maneiras de se contornar isso. O padrão Visitor, ou dual-dispatch, por exemplo.

Listagem 6-6 (continuação)
Classes shape polifórmicas

```
public class Circle implements Shape {
  private Point center;
  private double radius;
  public final double PI = 3.141592653589793;

  public double area() {
    return PI * radius * radius;
  }
}
```

Novamente, vemos que a natureza complementar dessas duas definições: elas são praticamente opostas! Isso expõe a dicotomia fundamental entre objetos e estruturas de dados:

O código procedimental (usado em estruturas de dados) facilita a adição de novas funções sem precisar alterar as estruturas de dados existentes. O código orientado a objeto (OO), por outro lado, facilita a adição de novas classes sem precisar alterar as funções existentes.

O inverso também é verdade:

O código procedimental dificulta a adição de novas estruturas de dados, pois todas as funções teriam de ser alteradas. O código OO dificulta a adição de novas funções, pois todas as classes teriam de ser alteradas.

Portanto, o que é difícil para a OO é fácil para o procedimental, e o que é difícil para o procedimental é fácil para a OO!

Em qualquer sistema complexo haverá vezes nas quais desejaremos adicionar novos tipos de dados em vez de novas funções. Para esses casos, objetos e OO são mais apropriados. Por ouro lado, também haverá vezes nas quais desejaremos adicionar novas funções em vez de tipos de dados. Neste caso, estruturas de dados e código procedimental são mais adequados.

Programadores experientes sabem que a ideia de que tudo é um objeto *é um mito*. Às vezes, você realmente *deseja* estruturas de dados simples com procedimentos operando nelas.

A lei de Demeter

Há uma nova heurística muito conhecida chamada Lei de Demeter[2]: um módulo não deve enxergar o interior dos objetos que ele manipula. Como vimos na seção anterior, os objetos escondem seus dados e expõem as operações. Isso significa que um objeto não deve expor sua estrutura interna por meio dos métodos acessores, pois isso seria expor, e não ocultar, sua estrutura interna.

Mais precisamente, a Lei de Demeter diz que um método f de uma classe C só deve chamar os métodos de:

- *C*
- Um objeto criado por *f*
- Um objeto passado como parâmetro para *f*
- Um objeto dentro de uma instância da variável *C*

2 http://en.wikipedia.org/wiki/Law_of_Demeter

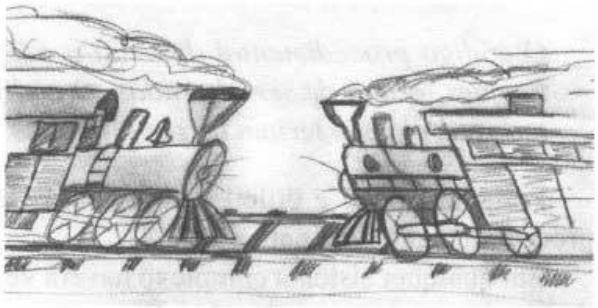
O método *não* deve chamar os métodos em objetos retornados por qualquer outra das funções permitidas. Em outras palavras, fale apenas com conhecidos, não com estranhos.
O código[3] seguinte parece violar a Lei de Demeter (dentre outras coisas), pois ele chama a função getScratchDir() no valor retornado de getOptions() e, então, chama getAbsolutePath() no valor retornado de getScratchDir().

```
final String outputDir = ctxt.getOptions().getScratchDir().
getAbsolutePath();
```

Carrinhos de trem

Esse tipo de código costuma ser chamador de *carrinho de trem*, pois parece com um monte de carrinhos de trem acoplados. Cadeias de chamadas como essa geralmente são consideradas descuidadas e devem ser evitadas [G36]. Na maioria das vezes é melhor dividi-las assim:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```



Esses dois pedaços de código violam a Lei de Demeter? Certamente módulo que os contém sabe que o objeto `ctxt` possui opções (*options*), que contêm um diretório de rascunho (*scratchDir*), que tem um caminho absoluto (*AbsolutePath*). É muito conhecimento para uma função saber. A função de chamada sabe como navegar por muitos objetos diferentes.

Se isso é uma violação da Lei de Demeter depende se `ctxt`, `Options` e `ScracthDir` são ou não objetos ou estruturas de dados. Se forem objetos, então sua estrutura interna deveria estar oculta ao invés de exposta, portanto o conhecimento de seu interior é uma violação clara da lei. Por outro lado, se forem apenas estruturas de dados sem atividades, então eles naturalmente expõem suas estruturas internas, portanto aqui não se aplica a lei.

O uso de funções de acesso confunde essas questões. Se o código tiver sido escrito como abaixo, então provavelmente não estaríamos perguntando sobre cumprimento ou não da lei.

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

Essa questão seria bem menos confusa se as estruturas de dados simplesmente tivessem variáveis públicas e nenhuma função, enquanto os objetos tivessem apenas variáveis privadas e funções públicas. Entretanto, há frameworks e padrões (*e.g., "beans"*) que exigem que mesmo estruturas de dados simples tenham métodos acessores e de alteração.

3. Está em algum lugar no framework do Apache.

Híbridos

De vez em quando, essa confusão leva a estruturas híbridas ruins que são metade objeto e metade estrutura de dados. Elas possuem funções que fazem algo significativo, e também variáveis ou métodos de acesso e de alteração públicos que, para todos os efeitos, tornam públicas as variáveis privadas, incitando outras funções externas a usarem tais variáveis da forma como um programa procedimental usaria uma estrutura de dados[4].

Esses híbridos dificultam tanto a adição de novas funções como de novas estruturas de dados. Eles são a pior coisa em ambas as condições. Evite criá-los. Eles indicam um modelo confuso cujos autores não tinham certeza – ou pior, não sabiam – se precisavam se proteger de funções ou tipos.

Estruturas ocultas

E se `ctxt`, `options` e `scratchDir` forem objetos com ações reais? Então, como os objetos devem ocultar suas estruturas internas, não deveríamos ser capazes de navegar por eles. Então, como conseguiríamos o caminho absoluto de scratchDir ('diretório de rascunho')?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

ou

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

A primeira opção poderia levar a uma abundância de métodos no objeto ctxt. A segunda presume que `getScratchDirectoryOption()` retorna uma estrutura de dados, e não um objeto. Nenhuma das opções parece boa.

Se ctxt for um objeto, devemos dizê-lo para fazer algo; não devemos perguntá-lo sobre sua estrutura interna. Por que queremos o caminho absoluto de scratchDir? O que faremos com ele? Considere o código, muitas linhas abaixo, do mesmo módulo:

```
String outFile = outputDir + "/" + className.replace('.', '/') +
".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

A mistura adicionada de diferentes níveis de detalhes [G34][G6] é um pouco confusa. Pontos, barras, extensão de arquivos e objetos `File` não devem ser misturados entre si e nem com o código que os circunda. Ignorando isso, entretanto, vimos que a intenção de obter o caminho absoluto do diretório de rascunho era para criar um arquivo de rascunho de um determinado nome.
Então, e se disséssemos ao objeto ctxt para fazer isso?

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

That seems like a reasonable thing for an object to do! Isso permite ao ctxt esconder sua estrutura interna e evitar que a função atual viole a Lei de Demeter ao navegar por objetos os quais ela não deveria enxergar.

4. Às vezes chama-se de *Feature Envy* em [Refatoração].

Objetos de transferência de dados

A forma perfeita de uma estrutura de dados é uma classe com variáveis públicas e nenhuma função.

Às vezes, chama-se isso de objeto de transferência de dados, ou DTO (sigla em inglês). Os DTOs são estruturas muitos úteis, especialmente para se comunicar com bancos de dados ou analisar sintaticamente de mensagens provenientes de sockets e assim por diante. Eles costumam se tornar os primeiros numa série de estágios de tradução que convertem dados brutos num banco de dados em objetos no código do aplicativo.

De alguma forma mais comum é o formulário "bean" exibido na Listagem 6.7. Os beans têm variáveis privadas manipuladas por métodos de escrita e leitura. O aparente encapsulamento dos beans parece fazer alguns puristas da OO sentirem-se melhores, mas geralmente não oferece vantagem alguma.

Listagem 6-7
address.java

```
public class Address {
  private String street;
  private String streetExtra;
  private String city;
  private String state;
  private String zip;

  public Address(String street, String streetExtra,
                 String city, String state, String zip) {
    this.street = street;
    this.streetExtra = streetExtra;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  public String getStreet() {
    return street;
  }

  public String getStreetExtra() {
    return streetExtra;
  }

  public String getCity() {
    return city;
  }

  public String getState() {
    return state;
  }

  public String getZip() {
    return zip;
  }
}
```

O Active Record

Os Active Records são formas especiais de DTOs. Eles são estruturas de dados com variáveis públicas (ou acessadas por Beans); mas eles tipicamente possuem métodos de navegação, como `save` (salvar) e `find` (buscar). Esses Active Records são traduções diretas das tabelas de bancos de dados ou de outras fontes de dados.

Infelizmente, costumamos encontrar desenvolvedores tentando tratar essas estruturas de dados como se fossem objetos, colocando métodos de regras de negócios neles. Isso é complicado, pois cria um híbrido entre uma estrutura de dados e um objeto.

A solução, é claro, é tratar o Record Active como uma estrutura de dados e criar objetos separados que contenham as regras de negócio e que ocultem seus dados internos (que provavelmente são apenas instâncias do Active Record).

Conclusão

Os objetos expõem as ações e ocultam os dados. Isso facilita a adição de novos tipos de objetos sem precisar modificar as ações existentes e dificulta a inclusão de novas atividades em objetos existentes. As estruturas de dados expõem os dados e não possuem ações significativas. Isso facilita a adição de novas ações às estruturas de dados existentes e dificulta a inclusão de novas estruturas de dados em funções existentes.

Em um dado sistema, às vezes, desejaremos flexibilidade para adicionar novos tipos de dados, e, portanto, optaremos por objetos. Em outras ocasiões, desejaremos querer flexibilidade para adicionar novas ações, e, portanto, optaremos tipos de dados e procedimentos.

Bons desenvolvedores de software entendem essas questões sem preconceito e selecionam a abordagem que melhor se aplica no momento.

Bibliografia

[Refatoração] *Refatoração - Aperfeiçoando o Projeto de Código Existente, Martin Fowler et al.*, Addison-Wesley, 1999.