Código Limpo:

Habilidades Práticas do Agile Software

•••

Capítulo 6: Objetos e Estruturas de Dados

Este material foi desenvolvido pelo grupo PET/ADS do IFSP São Carlos

Objetos e Estruturas de Dados: introdução

"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 tantos programadores optam por adicionar métodos de acesso automaticamente e expor suas variáveis privadas como públicas?

Abstração de dados: explicação

Ocultar a implementação é uma questão de **abstração!**

Uma classe deve expor uma interface abstrata que permite a manipulação da essência dos dados sem expor sua implementação.

É preciso pensar na melhor maneira de representar os dados de um objeto, adicionar métodos de escrita e leitura irrefletidamente é a pior opção.

Abstração de dados: exemplo 1

Os códigos abaixo representam os dados de um ponto no plano cartesiano:

Caso concreto

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

Expõe por completo sua implementação.

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);
}
```

Representa uma estrutura de dados clara, mesmo sem expôr a implementação de seus dados.

Abstração de dados: exemplo 2

Os códigos abaixo comunicam o nível de combustível de um veículo:

Veículo concreto

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

Veículo abstrato

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

A segunda abordagem é preferível nos dois exemplos, pois representa os dados de forma mais abstrata.

Antissimetria data/objeto: expondo a diferença

Qual a diferença entre objetos e estruturas de dados?

- ➔ Objetos usam abstrações para ocultar dados e expõe funções que operam sobre esses dados
- ➔ Estruturas de dados expõem seus dados e não possuem funções significativas

Apesar da diferença parecer trivial, ela carrega grandes implicações.

Antissimetria data/objeto: classe Shape procedimental

A seguir, é apresentado um exemplo de implementação de estruturas de dados:

```
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;
}
```

continua ...

Antissimetria data/objeto: classe Shape procedimental

A classe Geometry opera sobre as estruturas de dados, que não têm comportamentos:

```
public class Geometry {
   public final double PI = 3.141592653589793;

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

Antissimetria data/objeto: classe Shape polimórfica

O mesmo exemplo pode ser implementado com orientação a objetos:

```
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;
   }
}
```

continua ...

Antissimetria data/objeto: classe Shape polimórfica

O mesmo exemplo pode ser implementado com orientação a objetos (continuação):

```
                                                        ... continuação

public class Circle implements Shape {
   private Point center;
   private double radius;
   public final double PI = 3.141592653589793;
   public double area() {
       return PI * radius * radius;
   }
}
```

Antissimetria data/objeto: diferenças das classes Shape

Na abordagem **procedimental,** a adição da função perimeter() à classe Geometry não alteraria as classes shape.

Se for adicionada uma nova classe Shape, no entanto, todas as funções de Geometry serão alteradas.

Já na abordagem **polimórfica**, não é necessária a classe Geometry e, caso seja adicionada uma nova forma, nenhuma função será afetada.

No entanto, se forem adicionadas novas funções, todas as classes shapes serão alteradas

Antissimetria data/objeto: explicação

O código procedimental facilita a adição de novas funções, pois não demanda alteração das estruturas de dados já existentes.

O código orientado a objetos (abordagem polimórfica), por outro lado, facilita a adição de novas classes, sem precisar alterar as funções já existentes.

As necessidades do sistema ditarão qual abordagem é mais adequada.

A Lei de Demeter: explicação

"*Um módulo não deve enxergar o interior dos objetos que ele manipula*"

A Lei de Demeter diz que um método f de uma classe C só deve chamar métodos de:

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

Em outras palavras: *“Fale apenas com conhecidos, não com estranhos.”*

A Lei de Demeter: explicação

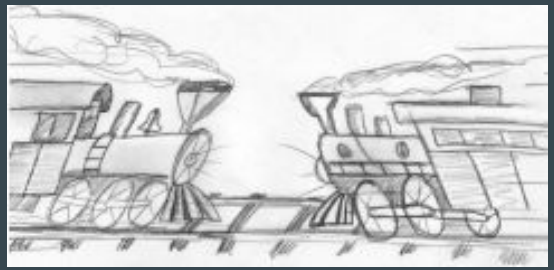
Pela Lei de Demeter, não se deve chamar métodos de objetos retornados por chamadas a outros métodos.

O código a seguir viola essa premissa duas vezes, consegue perceber?

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

Train Wrecks: explicação

Códigos como o anterior são chamados de Train Wrecks (acidente ferroviário), por parecerem com carrinhos de trem acoplados.



Train Wrecks: como evitar?

Cadeias de chamadas são consideradas descuidadas e devem ser evitadas. Ao invés de utilizá-las, prefira dividi-las desta forma:

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

Train Wrecks: violação da Lei de Demeter

O trecho anterior viola a Lei de Demeter? A resposta para a esta questão depende:

- ➔ Se ctxt, options e scratchDir forem objetos, suas estruturas internas deveriam estar ocultas, portanto o conhecimento das mesmas é uma clara violação da Lei.
- ➔ Mas se forem estruturas de dados, então suas estruturas internas já são naturalmente expostas e a Lei não se aplica nestes casos.

O uso de funções de acesso torna a questão confusa.

Híbridos

Em alguns casos, esta confusão leva a utilização de estruturas híbridas, metade objetos e metade estrutura de dados.

Estruturas deste tipo contém tanto funções que fazem algo significativo quanto variáveis e métodos de acesso e alteração públicos.

Híbridos dificultam tanto a adição de novas funções quanto de novas estruturas de dados e devem ser evitados

Estruturas Ocultas: código antes

Se ctxt, options e scratchDir forem objetos com ações reais, como conseguir o caminho absoluto de scratchDir?

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

ou

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

Nenhuma parece satisfatória!

Estruturas ocultas

Se ctxt for um objeto, devemos dizê-lo para fazer algo, não perguntá-lo sobre sua estrutura interna. Por que queremos o caminho absoluto de scratchDir?

Considerando o código abaixo, do mesmo módulo, descobre-se que a intenção é criar um diretório de rascunho para a criação de um arquivo de rascunho:

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

...

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

Estruturas ocultas: código depois

Então, devemos dizer para o objeto ctxt realizar esta tarefa:

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

Esta alteração permite ao ctxt esconder suas estruturas internas e evita que a função viole a Lei de Demeter.

Objetos de transferência de dados

Objeto de Transferência de Dados (DTO) é uma classe com variáveis públicas e nenhuma função

Estruturas muito úteis na comunicação com banco de dados, por exemplo

Os primeiros em estágios de tradução, que convertem dados brutos em objetos.

O formato mais comum é o formato de “bean”

Objetos de Transferência de Dados: exemplo

O código a seguir ilustra uma implementação de DTO:

```
public class Address {
   private String street;
   private String street Extra;
   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 = street Extra;
       this.city = city;
       this.state = state;
       this.zip = zip;
   }
```

continua ...

Objetos de Transferência de Dados: exemplo

O código a seguir ilustra uma implementação de DTO (continuação):

```
                                                                    continuação ...

    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

São formas especiais de DTOs que contém variáveis públicas, mas também possuem métodos de navegação como save(salvar) e find(buscar) por exemplo.

São traduções diretas de fontes de dados, como bancos de dados.

Não devemos inserir regras de negócios no Active Records, isso gera híbridos.

Deve-se tratar os Active Records como estrutura de dados e criar objetos separados para as regras de negócio.

Conclusão

Objetos expõem ações e ocultam os dados, facilitando a adição de novos tipos de objetos.

Estruturas de dados expõem os dados e não possuem ações significativas, facilitando a adição de novas ações.

Em partes do sistema, podemos desejar flexibilidade para adicionar novos tipos de dados, e, portanto, optamos por objetos.

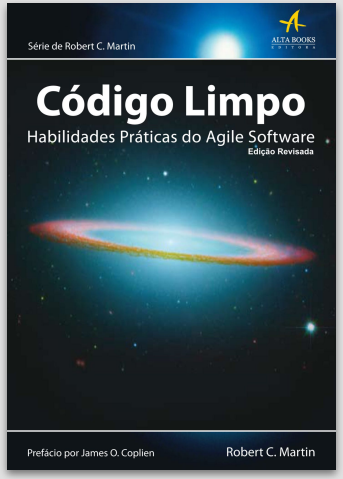
Em outros casos, podemos desejar flexibilidade para adicionar novas ações, e, portanto, optamos por tipos de dados e procedimentos.

Sobre o autor

Robert C. “Uncle Bob” Martin é desenvolvedor e consultor de software desde 1990. Ele é o fundador e o presidente da Object Mentor, Inc., uma equipe de consultores experientes que orientam seus clientes no mundo todo em C++, Java, C#, Ruby, OO, Padrões de Projeto, UML, Metodologias Agile e eXtreme Programming.

Este conjunto de slides foi elaborado a partir da obra:

MARTIN, Robert. **Código Limpo:**
Habilidades Práticas do Agile Software. 1. ed.
Rio de Janeiro: Alta Books, 2009.



A equipe

Lucas Sigoli, Autor

Aluno e bolsista do PET/ADS desde abril de 2022
LinkedIn

Lucas Oliveira, Revisor

Professor de Computação, é tutor do PET/ADS desde janeiro de 2023.
LinkedIn