Princípios Básicos para um Bom Desenho OO

Rodrigo Bonifácio

3 de dezembro de 2015

Existem várias heurísticas associadas ao desenho OO

- todas os atributos devem ser privados
- o uso de variáveis globais deve ser evitado
- o uso de RTTI deve ser evitado (o mesmo para reflection)
- programe para uma interface, em vez de uma implementação

Existem várias heurísticas associadas ao desenho OO

- todas os atributos devem ser privados
- o uso de variáveis globais deve ser evitado
- o uso de RTTI deve ser evitado (o mesmo para reflection)
- programe para uma interface, em vez de uma implementação

Qual a origem dessas heurísticas?

Existem várias heurísticas associadas ao desenho OO

- todas os atributos devem ser privados
- o uso de variáveis globais deve ser evitado
- o uso de RTTI deve ser evitado (o mesmo para reflection)
- programe para uma interface, em vez de uma implementação

Qual a origem dessas heurísticas? A questão chave é a natureza evolutiva do software.

Existem várias heurísticas associadas ao desenho OO

- todas os atributos devem ser privados
- o uso de variáveis globais deve ser evitado
- o uso de RTTI deve ser evitado (o mesmo para reflection)
- programe para uma interface, em vez de uma implementação

Qual a origem dessas heurísticas? A questão chave é a natureza evolutiva do software.Precisamos escrever programas que são estáveis para atender às necessidades de mudança.

The Open Closed Principle (Bertand Meyer)

Software entities (classes, Modules, Methods, etc.) should be open for extension, but closed for modification.

The Open Closed Principle (Bertand Meyer)

Software entities (classes, Modules, Methods, etc.) should be open for extension, but closed for modification.

Como esse princípio se relaciona com a nossa implementação da linguagem MiniHaskell?

Módulos em conformidade com esse princípio são *abertos* a extensões, mas *fechados* a alterações. Ou seja, o design deve permitir a alteração do comportamento de um módulo sem que seja necessário alterar o código fonte do módulo.

Aspectos essenciais

- abstração
- composição de objetos via interfaces bem definidas

Ok . . . , vamos discutir um pouco de refactoring para entender melhor o problema (ler o primeiro capítulo do livro *Refactoring: Improving the Design of Existing Code*. Marting Fowler).

abstração e polimorfismo por subtipo são os mecanismos essenciais relacionados ao *Open-Closed Principle*. Esses mecanismos fazem uso extensivo de herança (herança de implementação? herança de tipo?)

abstração e polimorfismo por subtipo são os mecanismos essenciais relacionados ao *Open-Closed Principle*. Esses mecanismos fazem uso extensivo de herança (herança de implementação? herança de tipo?), e isso leva as seguintes questões:

- o que caracteriza uma boa hierarquia de classes?
- quais as armadilhas nos levam a construir hierarquias de classes não aderentes ao *Open-Closed Principle*?

Substitution Principle (B. Liskov)

Functions that use pointers or references to base classes must be able to use objects from derived classes without knowing it.

Substitution Principle (B. Liskov)

Functions that use pointers or references to base classes must be able to use objects from derived classes without knowing it.

- caso o corpo de um método possua uma referência a uma superclasse mas verifique se o objeto é instância das subclasses, o princípio da substituição é violado.

Exemplo

```
public double area(Shape s) {
    double area = 0.0;

    if(s instanceof Circle) {
        // ...
    }
    else if(s instance Rectangle) {
        // ...
    }
    //...
    return area;
}
```

Considere a seguinte implementação de um Retângulo

```
public class Rectangle {
  private double height;
  private double width;

  public void setHeight(double h) {
     this.height = h; }
  public void setWidth(double w) { this.width
     = w; }

  public double getHeight() { return height; }
  public double getWidth() { return width; }
}
```

Considere a seguinte implementação de um Retângulo

```
public class Rectangle {
  private double height;
  private double width;

  public void setHeight(double h) {
     this.height = h; }
  public void setWidth(double w) { this.width
     = w; }

  public double getHeight() { return height; }
  public double getWidth() { return width; }
}
```

1. como implementar uma classe Square?

Considere a seguinte implementação de um Retângulo

```
public class Rectangle {
  private double height;
  private double width;

  public void setHeight(double h) {
     this.height = h; }
  public void setWidth(double w) { this.width
     = w; }

  public double getHeight() { return height; }
  public double getWidth() { return width; }
}
```

1. como implementar uma classe Square?
2. quais cuidados caso optemos por herança?

O que aconteceria com o seguinte teste:

```
public void testArea(Rectangle r) {
  r.setWidth(5);
  r.setWidth(4);
  assertEquals(20, r.area());
}
```

A model, viewed in isolation, can not be meaningfully validated. The validity of a model can only be expressed in terms of its clients

- um quadrado pode ser um retângulo (na geometria)

A model, viewed in isolation, can not be meaningfully validated. The validity of a model can only be expressed in terms of its clients

- um quadrado pode ser um retângulo (na geometria) mas um objeto que representa um quadrado não corresponde a um objeto que representa um retângulo (os comportamentos são distintos)

A model, viewed in isolation, can not be meaningfully validated. The validity of a model can only be expressed in terms of its clients

- um quadrado pode ser um retângulo (na geometria) mas um objeto que representa um quadrado não corresponde a um objeto que representa um retângulo (os comportamentos são distintos), estão no mesmo nível de hierarquia de *figuras geométricas*.
- essa discussão tem relação com *design-by-contract* descrito por Bertand Meyer, onde objetos possuem `invariantes` e as operações possuem `pre-condicoes` e `pos-condicoes`. Precisamos refletir sobre tais construções em relação a retângulos e quadrados.

Pos-Condicao para setWidth em Rectangle

```
/**
 * ensure width == w && height = old.height
 */
public void setWidth(double w) { ... }
```

Pos-Condicao para setWidth em Square

```
/**
 * ensure width == w && height = w
 */
public void setWidth(double w) { ... }
```

Por outro lado, Betrand Meyer estabeleceu a seguinte regra:

...when redefining a routine [in a derivative], you may only replace its precondition by a weaker one, and its postcondition by a stronger one.

Por outro lado, Betrand Meyer estabeleceu a seguinte regra:

> ...when redefining a routine [in a derivative], you may only replace its precondition by a weaker one, and its postcondition by a stronger one.

- Por outro lado, a `pos-condicao` de `setWidth` em `Rectangle` não é preservada em `Square`. Isso também é uma violação do *Substitution Principle*.

Vários padrões e princípios estão descritos no site Object Mentor[1]. Essa é uma excelente referência para melhorar as habilidades necessárias para conceber um bom desenho OO.

Outros princípios

- The Dependency Inversion Principle (Layering)
- The Interface Segragation Principle (Template Method)

[1] http://www.objectmentor.com/resources/publishedArticles.html

Princípios Básicos para um Bom Desenho OO

Rodrigo Bonifácio

3 de dezembro de 2015