Princípios de projetos SOLID'os'

CI1163
Departamento de Informática
Universidade Federal do Paraná
Curitiba – PR

2019

SOLID

- Princípios de projeto orientado a objetos (mas que podem ser generalizados)
  - Tem como objetivo auxiliar no desenvolvimento de software
    - Código “limpo”
    - Responsabilidades bem definidas
    - Facilidade de refatoração e manutenção
- Sub-conjunto de boas práticas
  - Não são ideias novas, mas permite categorização útil
- Não são GRASP’s, mas possui elementos relacionados
- Autor: Robert Martin (uncle Bob)
  - referência sobre o assunto: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

SOLID

- **S** ingle Responsibility Principle
- **O**pen-Closed Principle
- **L** iskov Substitution Principle
- **I** nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle (SRP)

- **Uma classe deve ter UM, e somente um, MOTIVO para mudar**
- Alterações causam “incerteza”
  - Cada linha modificada pode introduzir BUG novo
  - Diminui a coesão e aumenta acoplamento
- Mais restrito que o padrão especialista (*cf. Larman*)
- É o padrão “base”
- Anti-padrões: *Classe Deus/Grande bola de lama/etc.*

Este código segue o SRP ?

```
class Turma {
    float calculaMedia(){/*...*/}
    List getAlunos(){/*...*/}
    int getTotalAlunos(){/*...*/}
    aluno add(Aluno aluno){/*...*/}
    void delete(Aluno aluno){/*...*/}

    void imprimeDados(){/*...*/}
    void mostraDados(){/*...*/}

    void load(){/*...*/}
    void save(){/*...*/}
    void update(){/*...*/}
    void delete(){/*...*/}
}
```

Pontos chave:

Os dados são relativos às turmas, mas são separados em subconjuntos de funcionalidades.

Na manutenção, as modificações são localizadas.

Adaptando para o SRP

```
class Turma {
    float calculaMedia(){/*...*/}
    List getAlunos(){/*...*/}
    int getTotalAlunos(){/*...*/}
    aluno add(Aluno aluno){/*...*/}
    void delete(Aluno aluno){/*...*/}
}
class TurmaViewer {
    void imprimeDados(){/*...*/}
    void mostraDados(){/*...*/}
}
class Repositorio {
    void load(){/*...*/}
    void save(){/*...*/}
    void update(){/*...*/}
    void delete(){/*...*/}
}
```

Solução:

- separação em classes diferentes

- válido para outros elementos de software: métodos, atributos, pacotes, etc.

Open-Closed Principle (OCP)

- **Objetos e/ou classes devem estar abertos para extensão, mas fechados para modificação**
  - É possível incluir novas funcionalidades
- Alterações causam “incerteza” (de novo)
- Abstração é a chave
- Utilização de bom encapsulamento
  - Atributos sempre privados
  - Uso de polimorfismo: criação de interfaces/classes abstratas
  - **Jamais** usar variáveis globais ou “similares”
- Anti-padrões: código espaguete

Este código segue o OCP ?

```
class Contrato {
    float salario()  {}
}
class Estagio {
    float bolsa() {}
}
class Folha {
    protected int saldo;
    public void calcular(){
      if ( f instanceof Contrato )
        this.saldo = f.salario();
      else
         if (f instanceof Estagio)
           this.saldo = f.bolsa();

    }
}
```

Ponto chave:

Há um problema de acoplamento de controle.

Difícil de alterar a classe Folha.

Adaptando para o OCP

```
interface Pagamento {
   public float getSaldo();
}
class Contrato implements Pagamento {
    float getSaldo() {...}
}
class Estagio implements Pagamento {
    float getSaldo() {...}
}
class Folha {
    protected int saldo;
    public float calcular(Pagamento p){
       saldo = getSaldo();
    }
}
```

Solução:

- separação em classes diferentes usando polimorfismo: objeto passado como parâmetro

- privilegiar atributos privados

- NUNCA ter atributos estáticos "globais"

Liskov* Substitution Principle(LSP)

- **Uma classe derivada deve ser substituível pela sua classe base**
  - *“Se para cada objeto **o1** do tipo **S** há um objeto **o2** do tipo **T** de forma que, para todos os programas **P** definidos em termos de **T**, o comportamento de **P** é inalterado quando **o1** é substituído por **o2** então **S** é um subtipo de **T***
- Utilização de poliformismo adequado
  - A validade do modelo depende de seus filhos
  - Relacionamento IS-A ligado ao comportamento
    - Problemas em *CASTs*
- Pode ser relacionado com “Design por contrato” (*Bertrand Meyer*)
  - Pré condições e pós condições na execução

**Barbara Liskov - Turing Award em 2008,* trabalhos sobre TAD

Este código segue o LSP ?

```
class Retangulo {
    private int base, altura;
    void calculaArea(){...}
}
class Quadrado extends Retangulo {
    void calculaArea(){...}
}
```

Ponto chave:

Há no relacionamento IS-A - É-UM. O quadrado não é um retângulo.

O que acontece com a base E altura ? Podem impactar no resultado

if it looks like a duck, quacks like a duck but needs batteries for that purpose - it's probably a violation of LSP

Adaptando para o LSP

```
class FormaGeometrica extends FormaGeometrica {
   abstract void calculaArea();
}

class Retangulo extends FormaGeometrica {
    private int base, altura;
    void calculaArea(){...}
}
class Quadrado extends FormaGeometrica {
    private int lado;
    void calculaArea(){...}
}
```

Solução:

- criar classe abstrata e não ter novos métodos nas sub-classes → próximo à criação de interfaces

*- **ou** realizar uma modelagem diferente, pode haver problema conceitual*

Interface Segregation Principle (ISP)

- Classe não implementa interface com métodos que não vai usar
- Evitar poluição da interface → baixo acoplamento
- Anti-padrão: interface “Deus”/martelo de ouro

Este código segue o ISP ?

```
interface Persistencia {
   void save();
   void insert(Object o);
   void delete();
   void next();
   void previous();
   void append();
   void write();
   void read();
}
class BancoDados implements Persistencia{
}
class Arquivo implements Persistencia{
}
```

Ponto chave:

Os métodos são específicos a seus tipos, não tentar fazer interfaces genéricas demais.

Adaptando para o ISP

```
interface PersistenciaBd {
   void save();
   void insert(Object o);
   void delete();
}
interface PersistenciaAquivo{
   void append();
   void write();
   void read();
   void next();
   void previous();
}
class BancoDados implements PersistenciaBD{
}
class Arquivo implements PersitenciaArquivo{
}
```

Solução:

*- a interface define apenas um conjunto de operações **coesas** → SRP*

Dependency Inversion Principle (DIP)

- Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações
- Abstrações não devem depender de detalhes. Detalhes (implementações) devem depender de abstrações
- Utilização constante de polimorfismo
  - Facilita a reutilização
- Anti-padrão: complexidade “acidental”/gambiarra/copiar-colar

Este código segue o DIP ?

```
class LembraSenha {
    private Connection dbConn;

    LembraSenha() {
       dbConnection = new MySQLConnection();
    }

}
```

Ponto chave:

Há um problema de forte acoplamento

Instanciação de módulos de baixo nível no lugar errado

Adaptando para o DIP

```
class LembraSenha {
    private Connection dbConn;

    LembraSenha(Connection c) {
       dbConnection = c;
    }

}

// definir a interface Connection e
// implementar diferentes subclasses
```

solução:

- Implementação baseada em interface

- Identificar qual módulo é alto ou baixo nível

- Passagem dos objetos com a implementação como parâmetros