

Visão e análise de projetos

UC: Modelos, Métodos e Técnicas em Engenharia de Software

Prof. Eliane Faveron Maciel

UNIFACS
ecossistema ânima

8 de abril de 2024

Overview

1. Princípios de Projeto

- 1.1 Abstração
- 1.2 Coesão e Acoplamento
- 1.3 Integridade Conceitual
- 1.4 Decomposição e Modularização

2. SOLID

3. Referências

Princípios de Projeto

Tópicos

- **Abstração**
- **Coesão e acoplamento**
- **Decomposição e modularização**
- Encapsulamento
- Separação de interface e implementação
- Suficiência e Completude
- Simplicidade
- Separação por interesses

Abstração

Segundo [1], uma abstração é uma representação simplificada de uma entidade.

Funções, classes, interfaces, pacotes e bibliotecas são os instrumentos clássicos oferecidos por linguagens de programação para criação de abstrações.

Exemplo:

Na autoescola, você aprende como os principais componentes do carro funcionam. Não precisamos entender cada componente em um nível técnico para aprender a dirigir.

- Freios
- Transmissão
- Sistema de suspensão
- Bateria

Exemplo de abstração em Python

Por exemplo, vamos dizer que você queira usar o módulo de estatísticas do Python, que é um módulo integrado do Python.

```
1 from statistics import mean
2
3 randomList = [-1.0, 2.5, 3.25, 5.75]
4 print(mean(randomList))
```

Coesão e Acoplamento

Toda classe deve implementar uma única funcionalidade ou serviço. Especificamente, todos os métodos e atributos de uma classe devem estar voltados para a implementação do mesmo serviço [1].

Acoplamento é a **força** da conexão entre duas classes. Dizemos que existem dois tipos de acoplamento entre classes: **acoplamento aceitável** e **acoplamento ruim**.¹

¹cap. 5 Princípios de Projeto. [1]

Integridade Conceitual

"Integridade conceitual é a consideração mais importante no projeto de sistemas. É melhor um sistema omitir algumas funcionalidades e melhorias anômalas, de forma a oferecer um conjunto coerente de ideias, do que oferecer diversas ideias interessantes, mas independentes e descoordenadas."[?]

Exemplos de falta de integridade conceitual em nível de código

- Quando uma parte do sistema usa um padrão de nomes para variáveis (por exemplo, *camelCase*, como em **notaTotal** e *snake_case* **nota_total**.
- Utilização de diferentes tipos de *frameworks* para manipulação de páginas Web.
- Quando em uma parte do sistema resolve-se um problema usando-se uma estrutura de dados X, e outra parte é resolvido por meio de uma estrutura Y.
- Quando funções de uma parte do sistema que precisam de uma determinada informação a obtém diretamente de um arquivo de configuração. E em outras funções a mesma informação deve ser passada como parâmetro.²

²cap. 5 Princípios de Projeto. [1]

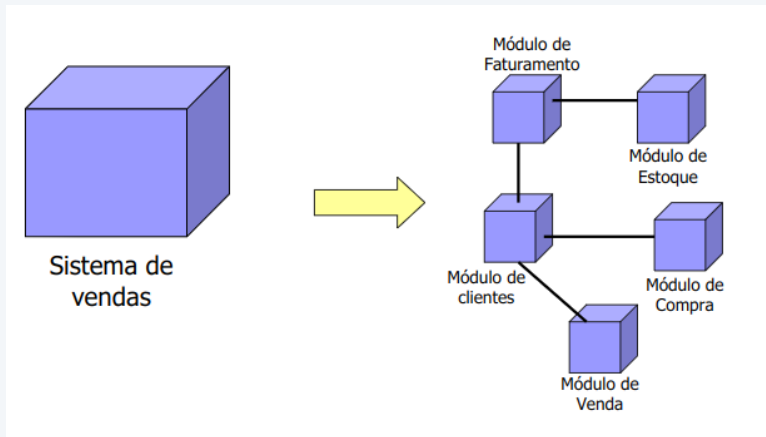
Ocultamento de Informação

Em 1972 David Parnas, introduz o termo **módulos** ou **modularização**. Hoje, muito utilizado com a divisão em **classes**.³

- **Desenvolvimento em paralelo.** Suponha que um sistema X foi implementado por meio de classes C1, C2, ..., Cn.
- **Flexibilidade a mudanças.** Por exemplo, suponha que descobrimos que a classe Ci é responsável pelos problemas de desempenho do sistema.
- **Facilidade de entendimento.** Por exemplo, um novo desenvolvedor contratado pela empresa pode ser alocado para trabalhar em algumas classes apenas.

³On the criteria to be used in decomposing systems into modules [2]

Exemplo de modularização



SOLID

UNIFACS

O que é SOLID?

- Princípios sugeridos por Robert C. Martin, em seu livro. Agile Principles, Patterns and Practices in C⁴;
- Listam práticas de design de software que aprimoram o código desenvolvido, facilitando seu reuso e compreensão;

⁴<https://www.amazon.com/Agile-Principles-Patterns-Practices-C/dp/0131857258>

Princípios SOLID

Vamos estudar os princípios propostos por Robert Martin e Michael Feathers [3].

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

Single Responsibility Principle

Princípio da Responsabilidade Única: *"Uma classe não pode ter mais de um motivo para ser alterada."*

- Uma classe deve fazer somente uma coisa.
 - Um método deve fazer somente uma coisa.
1. **Coesão** é o quanto os elementos de uma classe fazem sentido em estar juntos. (Não se mistura laranjas com cebolas)
 2. **Acoplamento** diz respeito a quanto uma classe depende da outra. A alteração em uma causará efeitos na outra.

Medida básica de qualidade

- Maior coesão → Melhor
- Maior acoplamento → Pior

```
1 class Quadrado:
2
3     def __init__(self, comprimento_lado=0):
4         self.comprimento_lado = comprimento_lado
5     def calcula_area(self):
6         return self.comprimento_lado ** 2
7     def desenhar(self):
8         # Desenha um quadrado
9         pass
```

Exemplo: Correto

```
1 class CalculaQuadrado:
2     def __init__(self, comprimento_lado=0):
3         self.comprimento_lado = comprimento_lado
4     def calcula_area(self):
5         return self.comprimento_lado * 2
6 class DesenhaQuadrado:
7
8     def desenha(self):
9         # Desenha um quadrado
10        pass
```

OPEN/CLOSED PRINCIPLE (OCP)

Princípio do aberto/fechado: *"O comportamento de uma classe deve estar aberta para extensão porém fechada para alterações."*

O projeto da classe prevê a possibilidade de extensões e customizações. Para isso, o projetista pode se valer de recursos como herança, funções de mais alta ordem e padrões de projeto (Abstract Factory, Template Method e Strategy)⁵.

⁵Veremos os padrões na próxima aula

Exemplo em Python:

Em Python, uma classe é mutável e um método é apenas um atributo de uma classe. Dessa forma, é possível sobrescrever um método dinamicamente em tempo de execução.

```
1 area = CalculaQuadrado(10)
2 def forma_geometrica():
3     return 'Sou um quadrado.'
4 area.forma_geometrica = forma_geometrica
5 print(area.forma_geometrica()) # Imprime: Sou um quadrado.
```

Exemplo em Java

```
1 public class Cachorro {  
2     protected String nome;  
3  
4     public Cachorro(String nome) {  
5         this.nome = nome;  
6     }  
7     public void andar() {  
8         // Implementação do método andar  
9     }  
10    public void latir() {  
11        // Implementação do método latir  
12    }  
13 }
```

Exemplo em Java

```
1 public class CachorroMachucado extends Cachorro {  
2  
3     ...  
4  
5     public void andar() {  
6         // Implementação da modificação do método  
7     }  
8 }
```

OCP: Resumo

- Quando é necessário adicionar funcionalidades a uma classe pré-existente, fazemos um herança da mesma.
- Nunca alteramos a classe-pai (superclasse) depois que a mesma já está sendo utilizada.

LISKOV SUBSTITUTION PRINCIPLE (LSP)

Princípio da substituição de Liskov: *"Uma classe filha deve poder ser substituída pela sua classe pai."*

Quando orientação a objetos se tornou comum, na década de 80, houve um incentivo ao uso de herança. Argumentava-se que hierarquias de classes profundas, com vários níveis, seriam um indicativo de um bom projeto, no qual foi possível atingir elevados índices de reuso [1].

- **Herança de classes** (exemplo: `class A extends B`), que é aquela que envolve reuso de código.
- **Herança de interfaces** (exemplo: `interface I extends J`), que não envolve reuso de código. Essa forma de herança é mais simples e não suscita preocupações.

LSP: Prefira Composição a Herança

Herança expõe para subclasses detalhes de implementação das classes pai. Logo, frequentemente diz-se que herança viola o encapsulamento das classes pai. A implementação das subclasses se torna tão acoplada à implementação da classe pai que qualquer mudança nessas últimas pode forçar modificações nas subclasses. [4]

Recomendação

Se existirem duas soluções de projeto, uma baseada em herança e outra em composição, a solução por meio de composição, normalmente, é a melhor. Uma relação de **composição** entre duas classes **A e B** quando a classe A possui um atributo do tipo B.

Exemplo em Java

```
1  class Stack extends
    ArrayList {
2      ...
3  }
4
5  class Stack {
6      private ArrayList
        elementos;
7      ...
8  }
```

1. um **Stack**, não é um ArrayList, mas sim uma estrutura que pode usar um ArrayList na sua implementação interna;
2. Quando se força uma solução via herança, a class Stack irá herdar métodos como get e set, que não fazem parte da especificação de pilhas.

```
1 class Retangulo:
2     def __init__(self, largura=0, altura=0):
3         self.largura = largura
4         self.altura = altura
5     def calcula_area(self):
6         return (self.largura * self.altura)
7
8 class Quadrado(Retangulo):
9     def __init__(self, largura=0, altura=0):
10         super(Quadrado, self).__init__(largura,
11                                         altura)
12         self.comprimento_largura = largura
13         self.comprimento_altura = altura
14     def calcula_area(self):
15         return (self.comprimento_largura * self.
16                 comprimento_altura)
```

Não faz sentido algum estender a classe **Retangulo** para a classe **Quadrado**. Os atributos são diferentes. Utilizar os atributos *comprimento_largura* e *comprimento_altura*, são uma gambiarra para resolvemos o problema.

INTERFACE SEGREGATION PRINCIPLE (ISP):

Princípio da segregação de interfaces: *"Várias interfaces específicas são melhores do que uma interface genérica."* Esse princípio também é uma aplicação da ideia de coesão.

- Uma classe não deve ser obrigada a implementar métodos que não serão utilizados.
- Evita-se esse problema dividindo (segregando) as operações definidas por cada interface.
- O número de interfaces aumenta, bem como a organização do código.

```
1 interface Funcionario {
2     double getSalario();
3     double getFGTS();// apenas funcionários CLT
4     int getSIAPE();// apenas funcionários públicos
5
6 }
```

```
1 interface Funcionario {
2     double getSalario();
3 }
4 interface FuncionarioCLT extends Funcionario {
5     double getFGTS();
6 }
7 interface FuncionarioPublico extends Funcionario {
8     int getSIAPE();
9 }
```

```
1 class CalculaArea:
2     def __init__(self, numero_lados=0):
3         self.numero_lados = numero_lados
4     def calcula_area(self):
5         return 'Calcula área para determinado polígono de N lados'
6 class CalculaVolume:
7     def __init__(self, numero_lados=0):
8         self.numero_lados = numero_lados
9     def calcula_volume(self):
10        return 'Calcula volume para determinado polígono de N lados'
11 class Quadrado(CalculaArea, CalculaVolume):
12     def __init__(self, numero_lados=None):
13         super(Quadrado, self).__init__(numero_lados)
14         self.numero_lados=numero_lados
15     @property
16     def area(self):
17         return self.calcula_area()
18     @property
19     def volume(self):
20 UNIFACS    return self.calcula_volume()
```

Dependency Inversion Principle

Princípio da inversão de dependências: *"Devemos depender de classes abstratas e não de classes concretas."*

- Módulos de alto nível não deveriam depender de módulos de baixo nível. Ambos deveriam depender de abstrações.
- Abstrações não deveriam depender de detalhes. Detalhes devem depender de abstrações.

Frame Title

```
1  interface I { ... }
2
3  class C1 implements I {
4      ...
5  }
6  class C2 implements I {
7      ...
8  }
9
10 class Cliente {
11
12     I i;
13
14     Cliente (I i) {
15         this.i = i;
16     }
```

Referências

Referências I



Marco Tulio Valente.

Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade.

Editora: Independente, 2020.



D. L. Parnas.

On the criteria to be used in decomposing systems into modules.

Commun. ACM, 15(12):1053–1058, dec 1972.





Robert C. Martin.


Clean Architecture: A Craftsman's Guide to Software Structure and Design.

Prentice Hall Press, USA, 1st edition, 2017.

Referências II

 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design patterns: elements of reusable object-oriented software.
Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

 Frederick P. Brooks.
The Mythical Man-Month: Essays on Softw.
Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1978.

 Ian Sommerville.
Software Engineering.
Addison-Wesley, Harlow, England, 9 edition, 2010.

Referências III



Roger S. Pressman.

Engenharia de software: Uma abordagem Profissional.

AMGH, Porto Alegre, 9 edition, 2021.



Viniacutecius Chan.

Solid com python: Entendendo os 5 princípios na prática, Feb 2019.



Obrigada

Prof. Eliane Faveron Maciel

UNIFACS
ecossistema ânima

8 de abril de 2024