UNIESP
Disciplina: Métodos Avançados de Programação
Professora: Alana Morais

Aluno: José Carlos Ribeiro Soares Junior

Princípios do SOLID

Introdução

Nos dias atuais com os sistemas cada vez mais complexos e com a participação intensa nas interações entre diversos times desenvolvedores que produzem os sistemas é importante que padrões sejam respeitados, primeiramente para uma melhor legibilidade de códigos, tendo em vista que outras pessoas vão interagir, participar mesmo que com uma rápida leitura do script, além de uma possível reaproveitamento de códigos, com arquivos mais simples e não menos eficientes.

São partes da construção do SOLID.

[S]ingle Responsibility Principle(Princípio da Responsabilidade Única),
[O]pen/Closed Principle (Princípio do Aberto/Fechado),
[L]iskov Substitution Principle (Princípio da Substituição de Liskov),
[I]nterface Segregation Principle (Princípio da Segregação de Interfaces),
[D]ependency Inversion Principle (Princípio da Inversão de Dependências).

Segundo Silva (2013) os princípios deste padrão significam. Propor uma maneira para desenvolvimento orientado a objetos visando à gestão de dependências e combate aos sintomas do apodrecimento do código-fonte, permeando a construção de melhores códigos do ponto de vista do Projeto e Construção de Software.

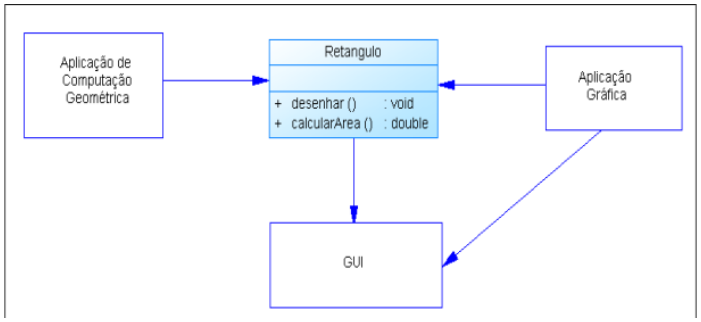
Os Princípios do SOLID.

1) Princípios da Responsabilidade Única (SRP)

Este Princípio descreve que uma classe deve ter uma boa coesão, o que facilita no entendimento dessa classe e o mantimento da mesma na utilização na construção do código. “Isso significa que você deve projetar suas classes de forma que cada uma tenha uma única finalidade. Isso não significa que cada classe deve ter apenas um método, mas que todos os membros da classe estão relacionados com a função principal da classe. Quando uma classe tem múltiplas responsabilidades, estes devem ser separados em classes novas.” (CARR, 2010).

É brilhante essa explicação, que deixa bem claro a usabilidade do princípio de responsabilidade única (SRP), mostrando claramente como este deve ser usado. Uma vantagem da aplicabilidade desse tipo de “regra” por assim dizer é a facilidade de entender o que quer dizer cada classe, porém ao escrever um código com este padrão os projetos sofrem forte acoplamento de suas classes, e podem ser quebrados de forma prejudicial se forem alterados inesperadamente.

> “A violação do SRP causa vários problemas desagradáveis. Primeiro, precisamos incluir GUI na Aplicação de Geometria Computacional. (...) Segundo, se uma alteração na Aplicação Gráfica fizer Retângulo mudar por algum motivo, essa mudança poderá nos obrigar a reconstruir, testar novamente e entregar a Aplicação de Geometria Computacional outra vez.” (MARTIN, 2011, p. 136).



Exemplo:
Neste exemplo é visto a quebra o uso da responsabilidade única, onde a classe “Retângulo” além de desenhar, também calcula a área, isso mostra a violação do princípio, onde a classe em destaque deveria ter apenas uma função.

Vejamos um exemplo descrito em código java, a maneira correta de aplicabilidade do princípio SRP:

```java
package io.github.mariazevedo88.solid.srp;

public class Funcionario {

	private Integer id;
	private String nome;
	private double salario;
	private Cargo cargo;

	public Funcionario() {}

	public Funcionario(Integer id, String nome, double salario, Cargo cargo) {
		this.id = id;
		this.nome = nome;
		this.salario = salario;
		this.cargo = cargo;
	}

	public Integer getId() {
		return id;
	}

	public void setId(Integer id) {
		this.id = id;
	}

	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}

	public double getSalario() {
		return salario;
	}

```

```
39          public void setSalario(double salario) {
40                  this.salario = salario;
41          }
42
43          public Cargo getCargo() {
44                  return cargo;
45          }
46
47          public void setCargo(Cargo cargo) {
48                  this.cargo = cargo;
49          }
50
51          public double calculaSalario() {
52                  return cargo.getRegra().calcula(this);
53          }
54
55          @Override
56          public String toString() {
57                  return "Funcionario [id=" + id + ", nome=" + nome + ", salario=" + salario + ",
58          }
59  }
```

2) Princípio Aberto/Fechado (OCP)

Este princípio tem por objetivo permitir que ao escrever o código, nas suas alterações ele não seja prejudicado por uma edição, ou seja, aquilo que já esta em funcionamento permanecerá em funcionamento e o que vier a ser implementado posteriormente, este será adicionado e não afetara a estrutura nem as funcionalidades uma vez implementadas. Isto na orientação a objeto é fantástico, essa habilidade de construir, adicionando nova funções sem alterar as preexistentes.

> *"as entidades de software (classes, módulos, funções etc.) devem ser abertas para ampliação, mas fechadas para modificação".* De forma mais detalhada, diz que podemos estender o comportamento de uma classe, quando for necessário, por meio de **herança**, **interface** e **composição**, mas não podemos permitir a abertura dessa classe para fazer pequenas modificações. (Azevedo, 20418)

Este principio é muito importante e considerado um dos mais importantes, devido a sua funcionalidade e modo de aplicação que dar uma liberdade para quem desenvolve inclusive para o time de desenvolvimento, permitindo a adição de mais funções sem necessitar modificar o que já existe.

Em um projeto complexo se faz necessário usar muito da abstração, com isso, entra em ação o principio OCP, com o uso de interfaces podemos organizar códigos que são bem acoplados e deixá-los mais enxutos.

Exemplo do código em Java.

```java
public class CalculadoraDePrecos {

    public double calcula(Produto produto) {

        Frete frete = new Frete();
        double desconto = 0d;

        int regra = produto.getMeioPagamento();

        switch(regra) {
                case 1:
                        System.out.println("Venda à vista");
                        TabelaDePrecoAVista tabela1 = new TabelaDePrecoAVista();
                        desconto = tabela1.calculaDesconto(produto.getValor());
                        break;
                case 2:
                        System.out.println("Venda à prazo");
                        TabelaDePrecoAPrazo tabela2 = new TabelaDePrecoAPrazo();
                        desconto = tabela2.calculaDesconto(produto.getValor());
                        break;
        }

        double valorFrete = frete.calculaFrete(produto.getEstado());
        return produto.getValor() * (1 - desconto) + valorFrete;
    }
}
```

CalculadoraDePrecos.java hosted with ❤ by GitHub view raw

Esse código é construído a parti de um sistema de e-commerce fictício. Nota-se que na construção das classes da pra perceber que várias regras são criadas torando o código cada vez mais amarrado

```java
public class TabelaDePrecoAVista {

        public double calculaDesconto(double valor) {
                if(valor > 100.0) {
                        return 0.05;
                }else if(valor > 500.0) {
                        return 0.07;
                }else if(valor > 1000.0) {
                        return 0.10;
                }else {
                        return 0d;
                }
        }

}
```

Quanto mais regras forem criadas, mas complexidade, por isso importante o uso de interfaces, para poder da liberdade na elaboração das soluções das funções que serão aplicadas.

```java
public interface TabelaDePreco {

        public double calculaDesconto(double valor);

}
```

```java
public interface ServicoDeFrete {

        public double calculaFrete(String estado);

}
```

Sendo assim, a implementação do código através das interfaces usando o principio OCP, principio do aberto/fechado é uma ajuda para evitar catástrofes nos códigos.

```java
public class Frete implements ServicoDeFrete{

        @Override
        public double calculaFrete(String estado) {
                if("SAO PAULO".equals(estado.toUpperCase())) {
                        return 7.5;
            }else if("MINAS GERAIS".equals(estado.toUpperCase())){
                return 12.5;
            }else if("RIO DE JANEIRO".equals(estado.toUpperCase())) {
                return 10.5;
            }else {
                return 10.0;
            }
        }

}
```

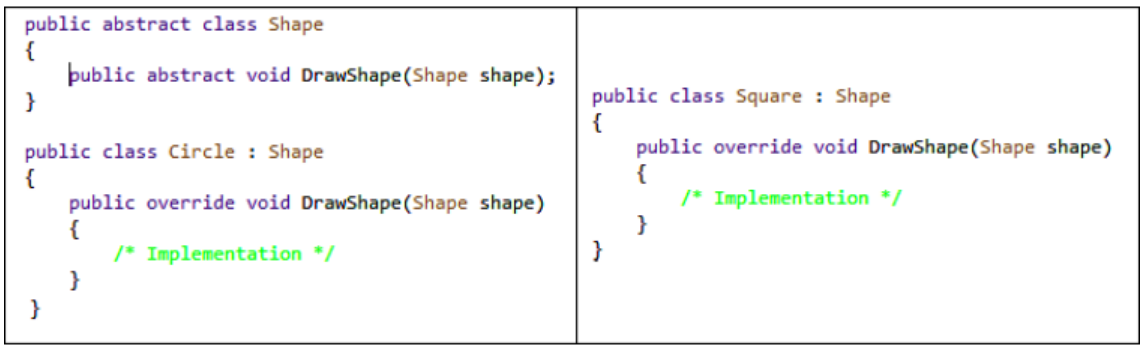
3) Principio da Substituição de Liskov

Princípio da substituição de Liskov, ou Liskov Substitution Principle (LSP) [Kk2009] aponta o princípio da substituição de Liskov, ou Liskov Substitution Principle (LSP), como sendo uma extensão do OCP, de forma que é difícil diferenciá-los, mas apontando uma sutil diferença entre eles. Para [Kk2009], OCP é centrado no acoplamento de abstrações. LSP, apesar de fortemente ligado no acoplamento de abstrações, também é fortemente ligado a pré-condições e póscondições, que ligam o LSP ao Projeto por Contrato, de Bertrand Meyer, onde os conceitos de pré-condições e pós-condições foram formalizados. (Azevedo, 20418)

Esse principio é interessante por apresentar um comportamento em que lembra o principio do OCP, mas se difere pela capacidade de assumir um

comportamento diferente, neste caso a aplicabilidade do princípio da substituição afirma que qualquer subclasse pode ser utilizada como classe pai sem que haja qualquer modificação, respeitando o que diz o principio OCP e que além de tudo não deve interferir ou modificar as classes herdadas, esse tipo de substituição segue os conceitos básicos da abstração e do polimorfismo, conceitos que são ligados a orientação a objeto.

Exemplo de aplicabilidade:



```
public abstract class Shape
{
    public abstract void DrawShape(Shape shape);
}

public class Circle : Shape
{
    public override void DrawShape(Shape shape)
    {
        /* Implementation */
    }
}
```

```
public class Square : Shape
{
    public override void DrawShape(Shape shape)
    {
        /* Implementation */
    }
}
```

Essa figura é explicada pela seguinte afirmação.

Por fim temos que o Princípio de Substituição de Liskov é um dos principais fatores concedentes do Princípio do Aberto / Fechado. “A possibilidade de substituição de subtipos permite que um módulo, expresso em termos de um tipo base, seja extensível sem modificação. (...) Assim, o contrato do tipo base precisa ser bem compreendido, se não explicitamente imposto, pelo código.” (MARTIN, 2011, p. 169)

Esses princípios assim como o da substituição de Liskov, são prontos apenas teóricos, são práticas a serem seguidas, aplicando esse principio ligado ao OCP é possível de um código bem escrito e independente.

4) Princípio da Segregação de Interfaces (ISP)

Make fine grained interfaces that are client specific

O Princípio da Segregação de Interfaces possui como lema “Clientes não devem ser forçados a depender de interfaces que eles não irão usar.” (MARTIN, 2011, p. 181). Como o próprio nome diz, define-se pela separação das interfaces que podem ser divididas. “Ou seja, as interfaces podem ser divididas em grupos de métodos. Cada grupo atende a um conjunto diferente de clientes. Assim, alguns clientes usam um grupo de métodos e outros clientes usam outros grupos.” (MARTIN, 2011, p. 181).

Esse principio sugere que classes que possam se tornar mais robustas, através de interfaces façam a comunicação entre funções que deveriam ser usadas de maneira especifica, a criação de mais interfaces de forma correta através deste padrão evita a poluição do código e faz com que a aplicabilidade de fato exista, o principio da

segregação faz com que as interfaces comuniquem-se com o usuário através das classes, mas elas que fazem a implementação do objeto.

Para nos livrarmos dos acoplamentos desnecessários o **ISP** alerta-nos para que não haja prejuízo as regras de negócios. Porém devemos ficar alerta para o exagero na criação de interfaces desnecessárias ou muitas exageradas.

Exemplo de aplicabilidade:

```java
package io.github.mariazevedo88.solid.isp;

public class Vendedor extends Funcionario implements Comissionavel{

	private double salario;
	private int totalVendas;

	public Vendedor(double salario, int totalVendas) {
		this.salario = salario;
		this.totalVendas = totalVendas;
	}

	@Override
	public double getSalario() {
		return this.salario + this.getComissao();
	}

	@Override
	public double getComissao() {
		return this.totalVendas * 0.2;
	}

	@Override
	public String toString() {
		return "Vendedor [salario=" + salario + ", totalVendas=" + totalVendas + "]";
	}
}
```

Neste trecho, podemos observar que a classe vendedor estende para a classe funcionário após isso, usa-se a interface de comissionavel, para garantir que o funcionário vendedor posso se comunicar com suas funções, e tudo isso funciona devido a abstração e a flexibilidade que as interfaces proporcionam.

Esse exemplo poderia ser escrito também para outro tipo de funcionário que não tivesse a interface comissionavel, sendo assim usaríamos a abstração para implementar um outro tipo de interface, contudo a classe abstrata vendedor e a classe pai funcionário não seriam modificadas e seus métodos e atributos seriam usados sem interferência no código.

5) Princípio da inversão de Dependências (DIP)

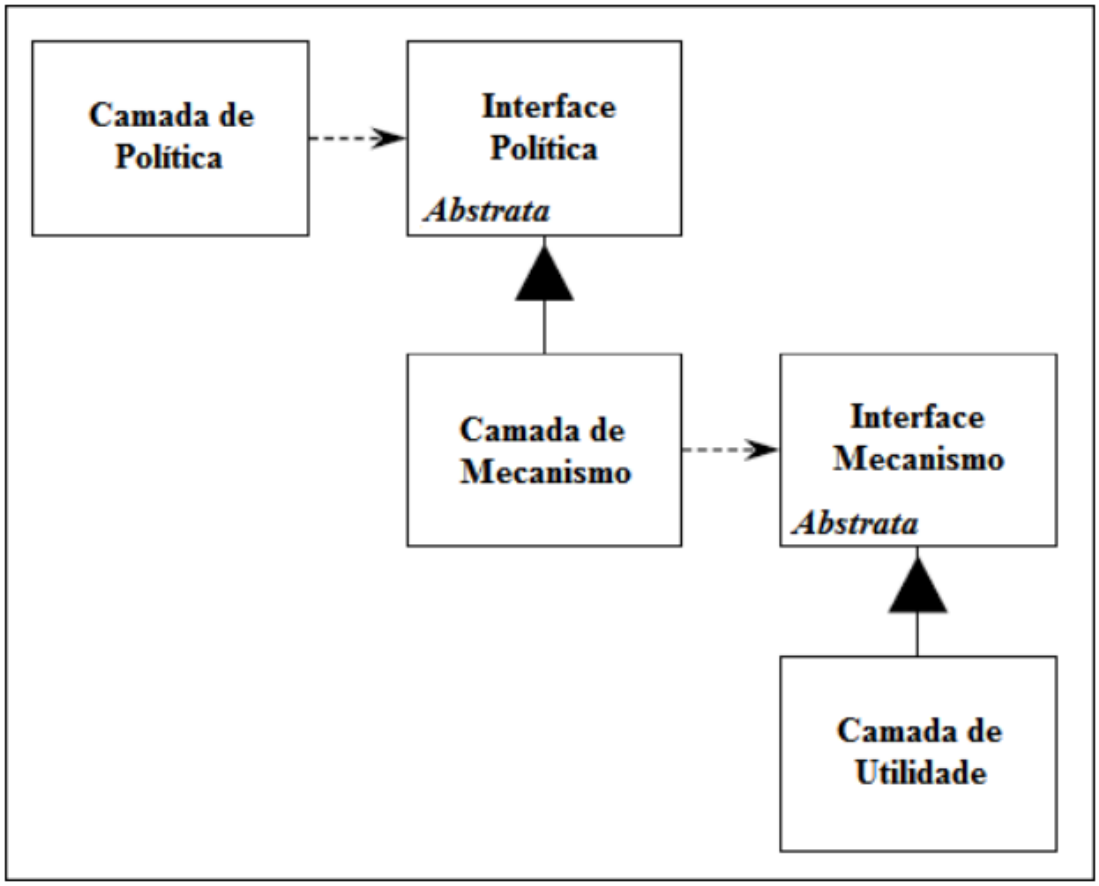
a. “Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.” (MARTIN, 2011, p. 171).

b. “As abstrações não devem depender de detalhes. Os detalhes devem depender das abstrações.” (MARTIN, 2011, p. 171).

Baseado no OCP este princípio tem por objetivo Este princípio existe porque “os métodos de desenvolvimento de software mais tradicionais, como o projeto e análise estruturados, tendem a criar estruturas de software nas quais os módulos de alto nível dependem de módulos de módulos de baixo nível.” (MARTIN, 2011, p. 171)

O DIP foca no nível de abstração do acoplamento das classes, foca na questão de que classes devem ser acopladas a outras classes por meio de abstração do código, podemos ver esse padrão sendo aplicado quando falamos sobre o OCP, mas não o citamos como fazemos agora, a abstração envolvida durante o desenvolvimento o torna perceptível apenas quando o queremos enxergar.

Neste exemplo podemos observar camadas invertidas.



Nesta figura é possível perceber que as classes de nível superior são concretizadas a parti de interfaces abstratas.

Martin (2011) diz que uma interpretação do DIP, um tanto mais simplista, apesar de ainda muito poderosa, é a heurística “depender de abstrações”. Dito de forma simples,

essa heurística recomenda que todos os relacionamentos em um programa devessem terminar em uma classe ou interface abstrata

Conclusão

O SOLID é um padrão de projeto complexo de se entender a primeira leitura, mas que se torna interessante ao se verificar seus pontos, suas peculiaridades, se faz necessário o conhecimento desse modo de organização, aplicando suas regras nos projetos construídos.

Trabalhar com padrões que permitem legibilidade a códigos, organização e facilidade de manutenção deve ser levado a sério por que constrói software, o SOLID permite que tenhamos uma regra de organização, facilitando no dia a dia a implementação de ideias e planos de negócios aplicados a padrões de projetos, o uso correto traz muitos benefícios e destaca o profissional que o executa.

A boa pratica do uso correto da orientação a objetos nos ajuda a evitar problemas como a falta de organização e dificuldade na manutenção de códigos, por isso destaco a importância do SOLID para manter, testar e reaproveitar sistemas.

Bibliografia

http://romeirao.quixada.ufc.br/portal/wp-content/uploads/2014/04/GarantPrincipLiskov.226.pdf

https://medium.com/@mari_azevedo/princ%C3%ADpios-s-o-l-i-d-o-que-s%C3%A3o-e-porque-projetos-devem-utiliz%C3%A1-los-bf496b82b299

SILVA, Álvaro César Pereira da. **PRINCÍPIOS S.O.L.I.D. DE DESIGN APLICADOS NA MELHORIA DE CÓDIGO-FONTE EM SISTEMAS ORIENTADOS A OBJETOS**. Lavras – Mg: Universidade Federal de Lavras, 2013.

https://gist.githubusercontent.com/mariazevedo88/21383894538bb75f3210711cc5db96e3/raw/db8a05497c75b78f88883c910355da1324d66544/Funcionario.java

https://www.unifacvest.edu.br/assets/uploads/files/arquivos/5f08f-padilha-junior,-e.-principios-solid-e-boas-praticas-de...-unifacvest,-2012..pdf