# Princípios de Projeto

Universidade Federal de Uberlândia
Programação Orientada a Objetos II
Prof. Fabiano Azevedo Dorça

# Princípios de Projeto

- **Princípios SOLID: O que são?**
- SOLID são cinco princípios básicos de programação e design orientados a objeto, introduzidos por Robert C. Martin no início dos anos 2000.
- A palavra **SOLID** é um acróstico onde cada letra significa a sigla de um princípio: **SRP, OCP, LSP, ISP e DIP**

# Princípios de Projeto

- Os princípios **SOLID** devem ser aplicados no desenvolvimento de software de forma que o ***software produzido tenha as seguintes características:***

- **Seja fácil de manter, adaptar e se ajustar às constantes mudanças de requisitos;**
- **Seja fácil de entender e testar;**
- **Seja construído de forma a estar preparado para ser facilmente alterado com o menor esforço possível;**
- **Seja possível de ser reaproveitado;**
- **Exista em produção o maior tempo possível;**
- **Que atenda realmente as necessidades dos clientes para o qual foi criado;**

# Princípios de Projeto

- A utilização dos princípios **SOLID** tem o objetivo de evitar :

- **Erros, Falhas e defeitos;**
- **Estrutura de código ruim;**
- **Código insustentável (difícil de manter) ;**
- **Desempenho sofrível;**
- **Código de difícil compreensão.**

# Princípios de Projeto

**S**RP: Single Responsibility Principle (Princípio da Responsabilidade Única). Noção de que ***um objeto deve possuir uma única responsabilidade.***

- Portanto uma classe deve ser implementada tendo apenas um único objetivo.
- Quando uma classe possui mais que um motivo para ser alterada é por que provavelmente ela esta fazendo mais coisas do que devia, ou seja, ela esta tendo mais de um objetivo.

# Princípios de Projeto

Podemos então inferir as seguintes premissas a partir da definição da responsabilidade única:

**– Baseado no princípio da coesão funcional, uma classe deve ter uma única responsabilidade;**

**– Se uma classe possuir mais de uma responsabilidade, deve-se considerar sua decomposição em duas ou mais classes;**

**– Cada responsabilidade é um "eixo de mudança" e as fontes de mudança devem ser isoladas;**

# Princípios de Projeto

- Exemplo:
-


- Métodos:
- **Area()** - Calcula a área do Retângulo;
- **Desenhar()** - Desenha o Retângulo;

# Princípios de Projeto

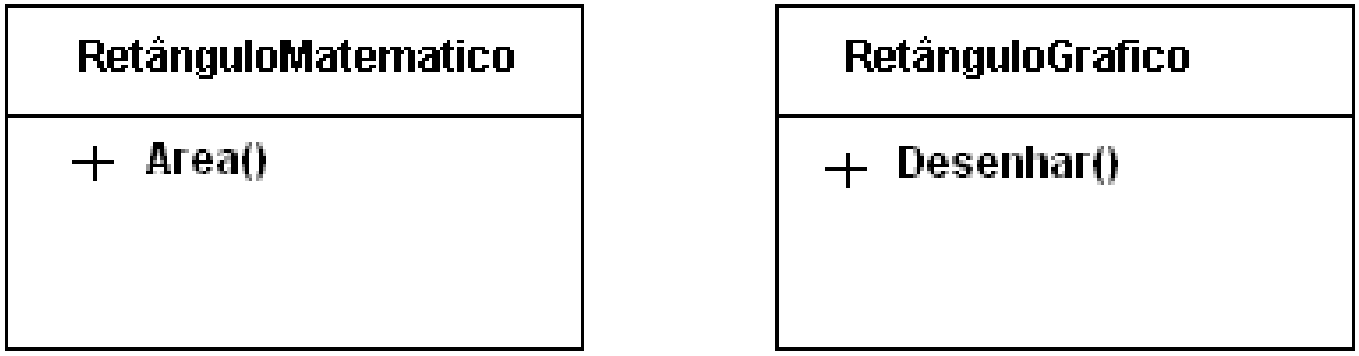
- Este desenho viola o princípio da responsabilidade única - **SRP** pois a classe **Retângulo** possui duas responsabilidades definidas:
- Calcular a área do retângulo usando um modelo matemático;
- Desenhar o retângulo usando uma interface gráfica;
- **PROBLEMA**: Qualquer alteração no modelo matemático implicará na modificação da classe, o que pode afetar a interação com a interface gráfica e qualquer alteração na interface gráfica também implicará em alterações na classe podendo afetar a utilização do modelo matemático usado para calcular a área do retângulo.

# Princípios de Projeto

- No caso da classe **Retângulo** um melhor desenho será separar as duas responsabilidades em duas classes diferentes:

MODELO

VISÃO



# Princípios de Projeto

- **A base do desenvolvimento em camadas é o SRP,** visto que o objetivo de criar camadas é separar a apresentação do negócio estamos indiretamente aplicando o **SRP** de forma a termos somente um motivo para que cada camada seja alterada.

# Princípios de Projeto

**O**CP: Open/Closed Principle (Princípio Aberto / Fechado).

Noção de que o software deve ser aberto para extensão, mas fechado para modificação.

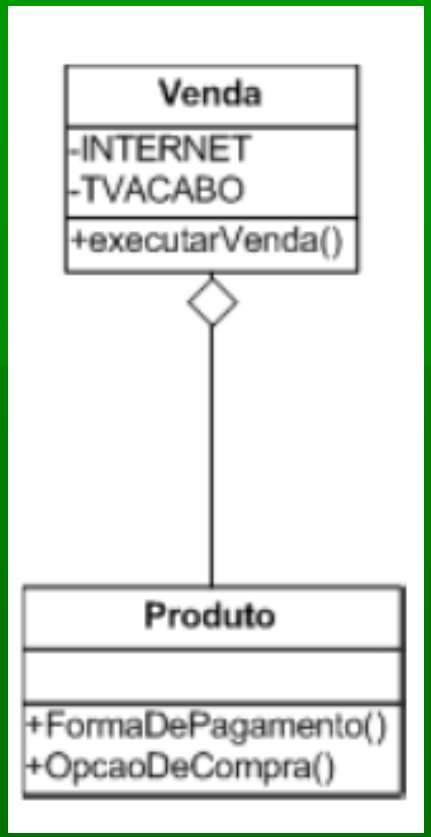
# Princípios de Projeto

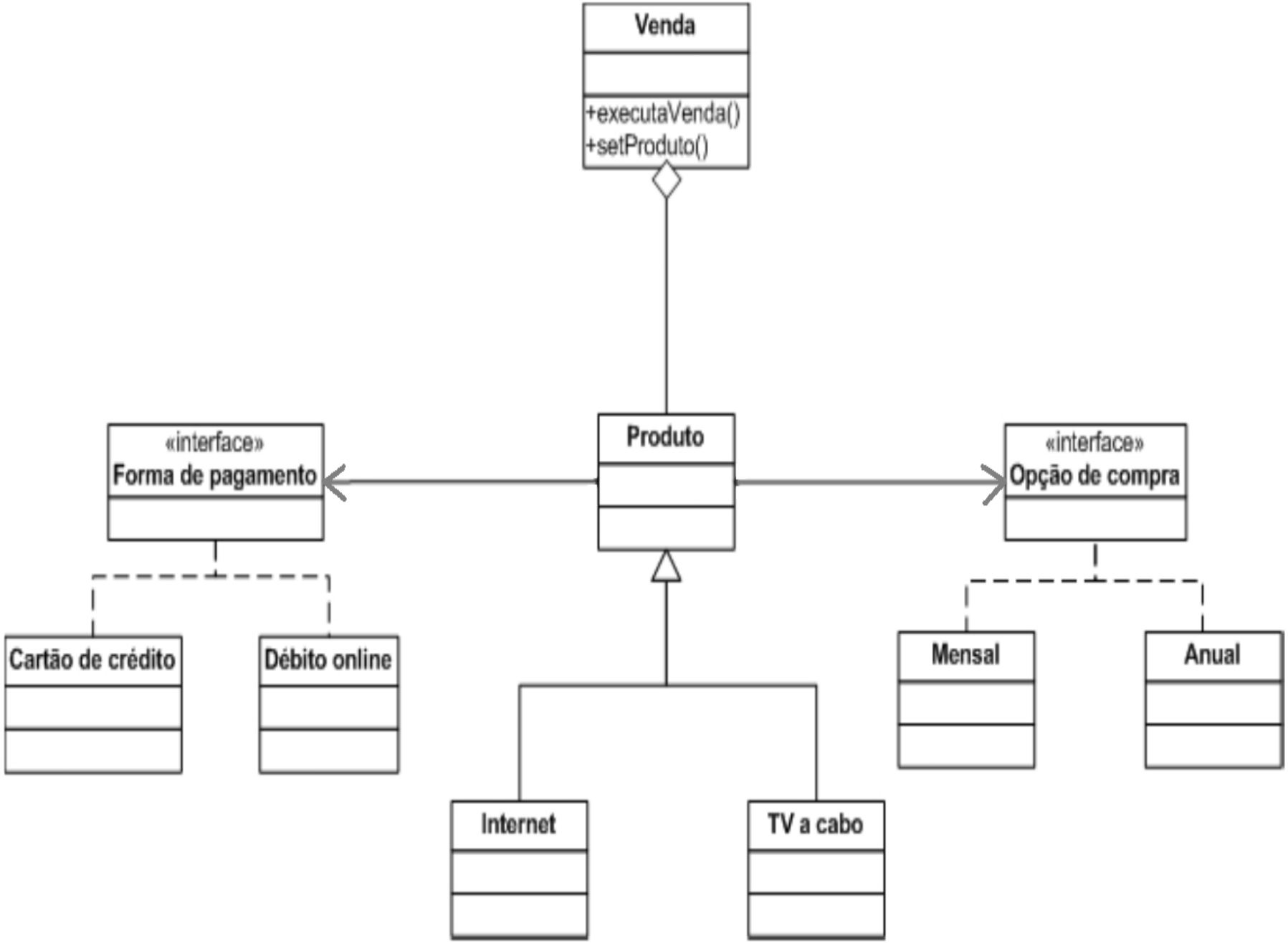
- *O princípio **Open/Closed:***

*uma nova funcionalidade deve ser adicionada com mudanças mínimas no código existente e que o projeto deve ser feito de forma a permitir a adição de novas funcionalidades como as novas classes, mantendo tanto quanto possível, o código existente inalterado.*

# Princípios de Projeto

- **Exemplo**:
- Problemas: Forma de Pagamento: como incluir novas formas? Opção de Compra: como incluir novas opções?




Venda
+executaVenda()
+setProduto()
«interface»
Forma de pagamento
Produto
«interface»
Opção de compra
Cartão de crédito
Débito online
Mensal
Anual
Internet
TV a cabo

# Princípios de Projeto

- **L**SP: Liskov Substitution Principle (Princípio da Substituição de Liskov).
- Noção de que objetos devem ser substituídos por instâncias de seus subtipos, sem afetar a correção do programa.

# Princípios de Projeto

- O princípio **LSP** foi definido por **Barbara Liskov** da seguinte forma:

  - **"Se *q*(*x*) é uma propriedade demonstrável dos objetos *x* de tipo *T*. Então *q*(*y*) deve ser verdadeiro para objetos *y* de tipo *S* onde *S* é um subtipo de *T*."**

# Princípios de Projeto

- *Traduzindo:*

***"Se você pode invocar um método q() de uma superclasse T , deve poder também invocar o método q() de uma subclasse T' que é derivada com herança de T ."***

- De forma bem simples o princípio de **Liskov** sugere que :

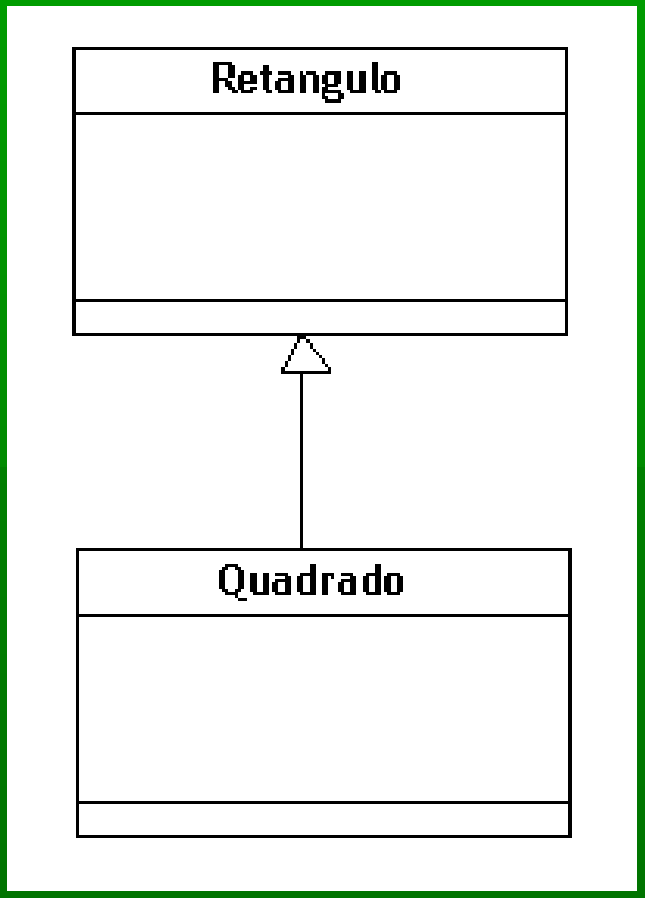
***"Sempre que uma classe cliente esperar uma instância de uma classe base X, uma instância de uma subclasse Y de X deve poder ser usada no seu lugar, sem alterar o comportamento esperado ou efeitos colaterais."***

# Princípios de Projeto

- Em outras palavras: **"*Uma classe base deve poder ser substituída pela sua classe derivada.*"**
- O objetivo é ter certeza de que novas classes derivadas estão estendendo das classes base mas sem alterar o seu comportamento.
- **Por que o princípio de LisKov é importante ?**
- **Porque sem aplicar o princípio de Liskov a hierarquia de classes seria uma bagunça e os testes de unidade para a superclasse nunca teriam sucesso para a subclasse.**

# Princípios de Projeto

## Exemplo:




A classe **Quadrado** *é um* **Retângulo** que tem uma característica especial: a largura e a altura são iguais.

# Princípios de Projeto

```
class Retangulo
    {
        protected int m_largura;
        protected int m_altura;

        public void setLargura(int largura)
        { m_largura = largura;}

        public void setAltura(int altura)
        {  m_altura = altura;  }

        public int getLargura()
        { return m_largura;   }

        public int getAltura()
        { return m_altura;    }

        public int getArea()
        { return m_largura * m_altura; }
    }
```

# Princípios de Projeto

```
class Quadrado Retangulo
    {
        public void setLargura(int largura)
        {
            m_largura = largura;
            m_altura = largura;
        }

        public void setAltura(int altura)
        {
            m_largura = altura;
            m_altura = altura;
        }
    }
```

# Princípios de Projeto

- Se as definições do nosso projeto estiverem de acordo com o princípio LSP a utilização da classe **Quadrado**() deverá poder ser usada sem problema algum no lugar da classe base **Retângulo**.

**TESTANDO...**

```
class Program
{
        private static Retangulo getNovoRetangulo()
        {
            //um factory
            return new Quadrado();
        }

        public static void main(string[] args)
        {
                Retangulo r = Program.getNovoRetangulo();

               r.setLargura(5);
               r.setAltura(10);

              // o usuário sabe que r é um retângulo
               // e assume que ele pode definir largura e altura
               // como para a classe base(Retangulo)
              System.out.println(r.getArea());

              // O valor retornado é 100 e não 50 como era esperado
        }
}
```

# Princípios de Projeto

- No exemplo temos que a uma instância da classe **Quadrado** é devolvida por uma *factory* com base em algumas condições e nós não sabemos exatamente que tipo de objeto será retornada.
- Se o princípio LSP estivesse sendo corretamente aplicado o fato de usar a instância de Quadrado não implicaria em mudança de comportamento como veremos que irá ocorrer.
- Estamos criando um novo retângulo definindo a altura de 5 e a largura de 10 para obter a área.
- O resultado deveria ser **50 mas obtemos 100.**

# Princípios de Projeto

- **Neste caso um quadrado não é um retângulo, e ao aplicarmos o princípio "É Um" da herança de forma automática vimos que ele não funciona para todos os casos.**

- A instância da classe **Quadrado** quando usada quebra o código produzindo um resultado errado e isso viola o principio de LisKov onde *uma classe filha (Quadrado) deve poder substituir uma classe base(Retangulo).*

# Princípios de Projeto

ISP: Interface Segregation Principle (Princípio de Segregação da Interface).

Muitas interfaces específicas são melhores do que uma interface de uso geral.

# Princípios de Projeto

O princípio da segregação de interfaces ajuda a resolver problemas de interface poluída, disponibilizando métodos desnecessários e consequentemente, criando dependencias desnecessárias.

O Interface Segregation Principle (ISP) afirma que as classes não devem implementar métodos que não precisam.

# Princípios de Projeto

Interface Poluída:

Quando se tem uma classe que implementa métodos que não utiliza e tem-se outra(s) classe(s) que o utiliza, a primeira classe será afetada pelas mudanças que a segunda classe necessitar.

# Princípios de Projeto

**Exemplo:**

```
public interface Animal {
    void fly();
    void  run();
    void bark();
}
```

# Princípios de Projeto

```
public class Bird implements Animal {
    public void bark() { }
    public void  run() {
        // Implementação
    }
    public void fly() {
        // Implementação
    }
}
```

# Princípios de Projeto

```
public class Cat implements Animal {
    public void fly() {  }
    public void bark() {  }
    public void  run() {
        // Implementação
    }
}
```

# Princípios de Projeto

```
public class Dog implements Animal {
    public void fly() { }
    public void bark() {
        // implementação
    }
    public void  run() {
        // implementação
    }
}
```

# Princípios de Projeto

Uma classe deve depender apenas dos métodos que realmente serão utilizados.

Este objetivo pode ser alcançado quebrando as interfaces da classe poluída em interfaces específicas, eliminando a dependência dos métodos que não utilizam, criando uma independência entre eles.

# Princípios de Projeto

**Aplicando o ISP:**

```
public interface Flyable {
    void fly();
}
public interface Runnable {
    void  run();
}
public interface Barkable {
    void bark();
}
```

# Princípios de Projeto

```
public class Bird implements Flyable, Runnable {
    public void  run() {
        // implementação
    }
    public void fly() {
        // implementação
     }
}
```

# Princípios de Projeto

```
public class Cat implements Runnable{
    public void  run() {
        // implementação
    }
}
```

# Princípios de Projeto

```
public class Dog implements Runnable, Barkable {
    public void bark() {
        // implementação
    }
    public void  run() {
        // implementação
    }
}
```

# Princípios de Projeto

Clientes não devem ser forçados a depender de métodos que eles não usam.

Esse princípio trata dos problemas de termos interfaces "gordas".

Uma interface "gorda" geralmente tem baixa coesão, e poderia ser quebradas em mais de uma interface.

# Princípios de Projeto

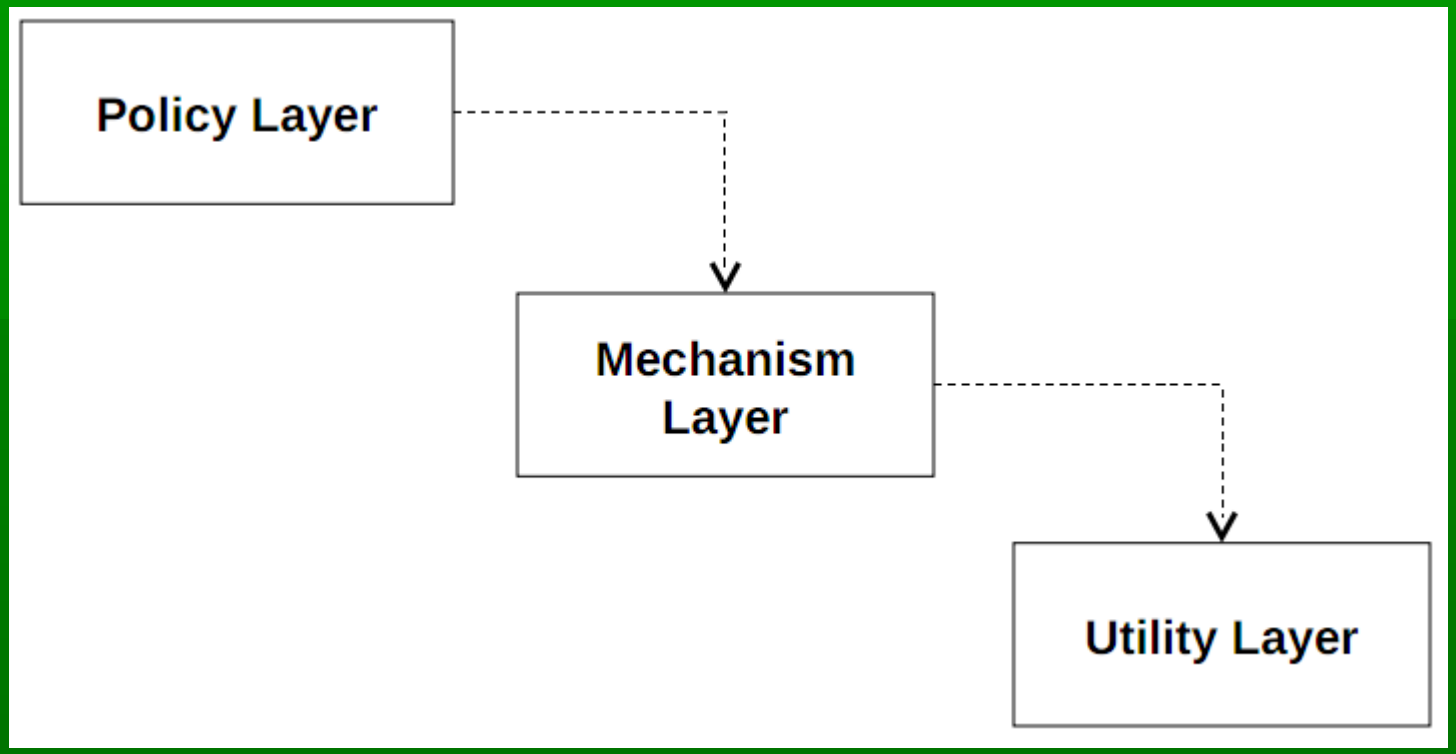
**D**IP: Dependency Inversion Principle (Princípio de Inversão da Dependência).

Módulos de “alto nível” não devem depender de módulos de “baixo nível”. Os dois devem depender de abstrações.

Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

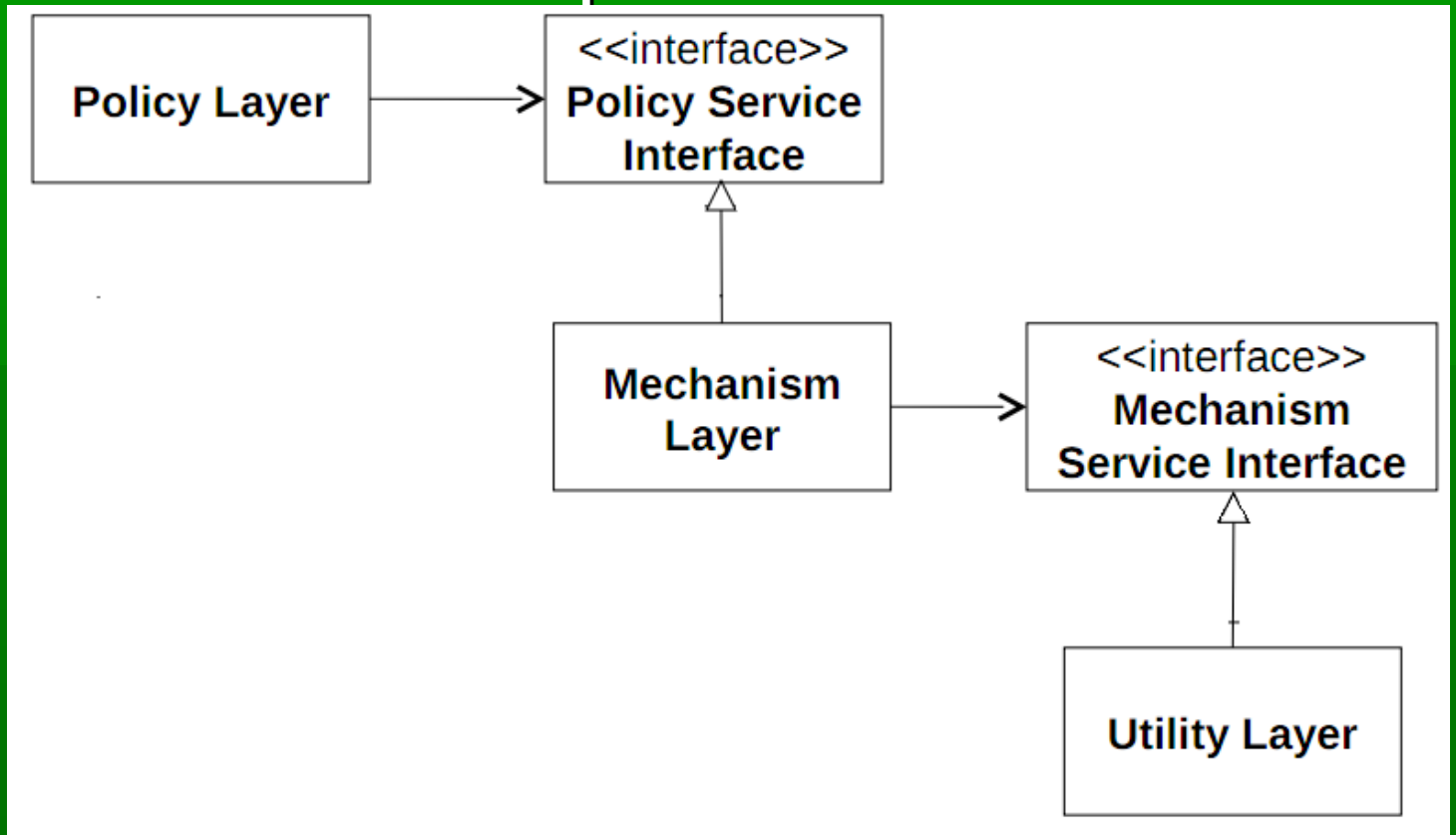
# Princípios de Projeto

- Considere uma aplicação estruturada em camadas



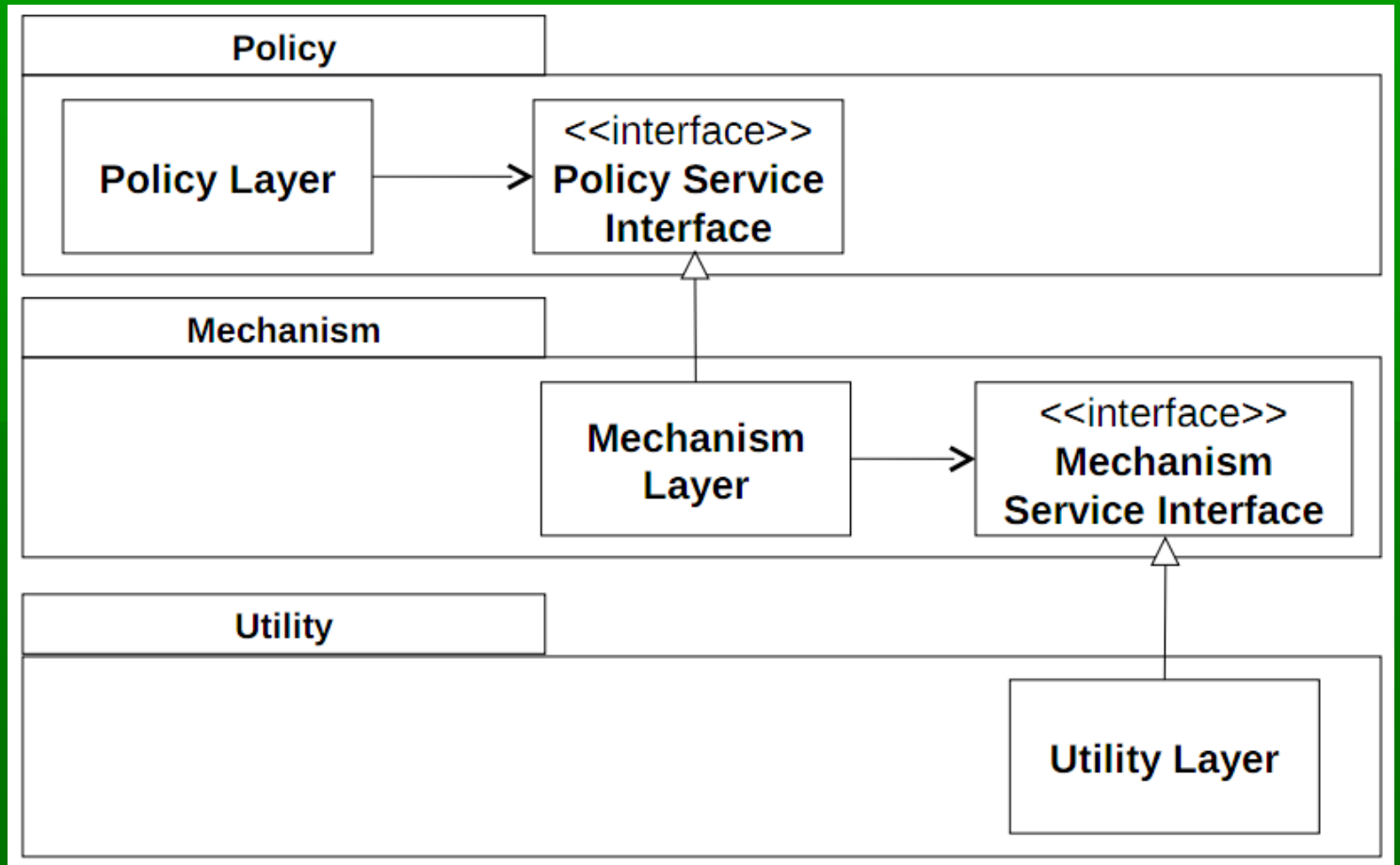
# Princípios de Projeto

- A mesma estrutura em camadas, com inversão de dependências:



# Princípios de Projeto

Dividindo em pacotes:



# Princípios de Projeto

- Conclusão:
- Quando os princípios são aplicados em conjunto, aumentam bastante as chances de que o sistema criado seja simples de manter e estender.