Princípios de Projeto Orientado a Objetos

Marum Simão Filho

marumsimao@gmail.com

Agenda

- Introdução
- Desenvolvimento de Software Adaptativo
- Conceitos da Orientação a Objetos
- Princípios S.O.L.I.D.

Introdução

- Saber uma linguagem de programação orientada a objetos (OO) não é suficiente para criar bons sistemas OO.
- É preciso conhecer projeto (*design*) OO
  - É preciso saber como usar adequadamente os mecanismos de OO.
  - Só assim conseguimos o que OO promete
    - Mantenabilidade, reusabilidade, extensibilidade, adaptabilidade...
- Princípios e padrões de projeto nos mostram como usar bem os mecanismos de OO.

Desenvolvimento de Software Adaptativo

- *Manifesto do Desenvolvimento Ágil*
  agilemanifesto.org
  - Indivíduos e interações mais que processos e ferramentas.
  - Software Funcionando mais que Documentação Extensa.
  - Cliente Colaborativo mais que Negociação de Contratos.
  - **Resposta à Mudança** mais que Seguir um Plano.

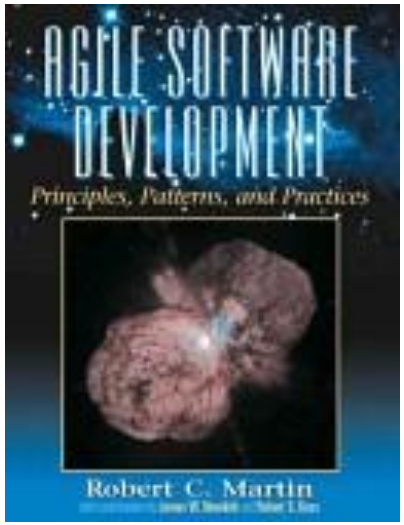
- *Princípios por trás do Manifesto Ágil*
  agilemanifesto.org/principles.html
  - **Aceite mudanças de requisito,** mesmo que tardiamente
  - Atenção contínua à **excelência técnica** e **boas práticas de design** melhoram a **agilidade**

Desenvolvimento de Software Adaptativo

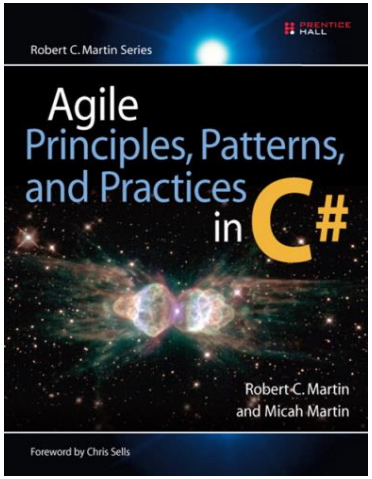
- *Scrum Flácido*,
  Martin Fowler, Janeiro, 2009
  martinfowler.com/bliki/FlaccidScrum.html
  - "O que aconteceu [com alguns projetos Scrum recentemente] é que não prestaram a devida atenção à Qualidade Interna do software produzido. Se você cometer esse erro, em breve você terá sua produtividade destruída, porque adicionar novas funcionalidades ao software será muito mais difícil do que você gostaria..."

"A comunidade Scrum precisa redobrar seus esforçoes para garantir que as pessoas entendam a importância de fortes **Práticas e Técnicas**..."

Desenvolvimento de Software Adaptativo



Robert C. Martin é autor de livros clássicos sobre design OO e desenvolvimento Ágil

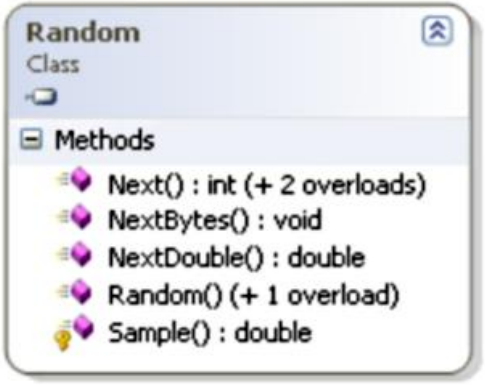


Conceitos de Orientação a Objetos

- Coesão
- Acoplamento
- Abstração
- Encapsulamento
- Herança
- Polimorfismo

Conceitos de Orientação a Objetos

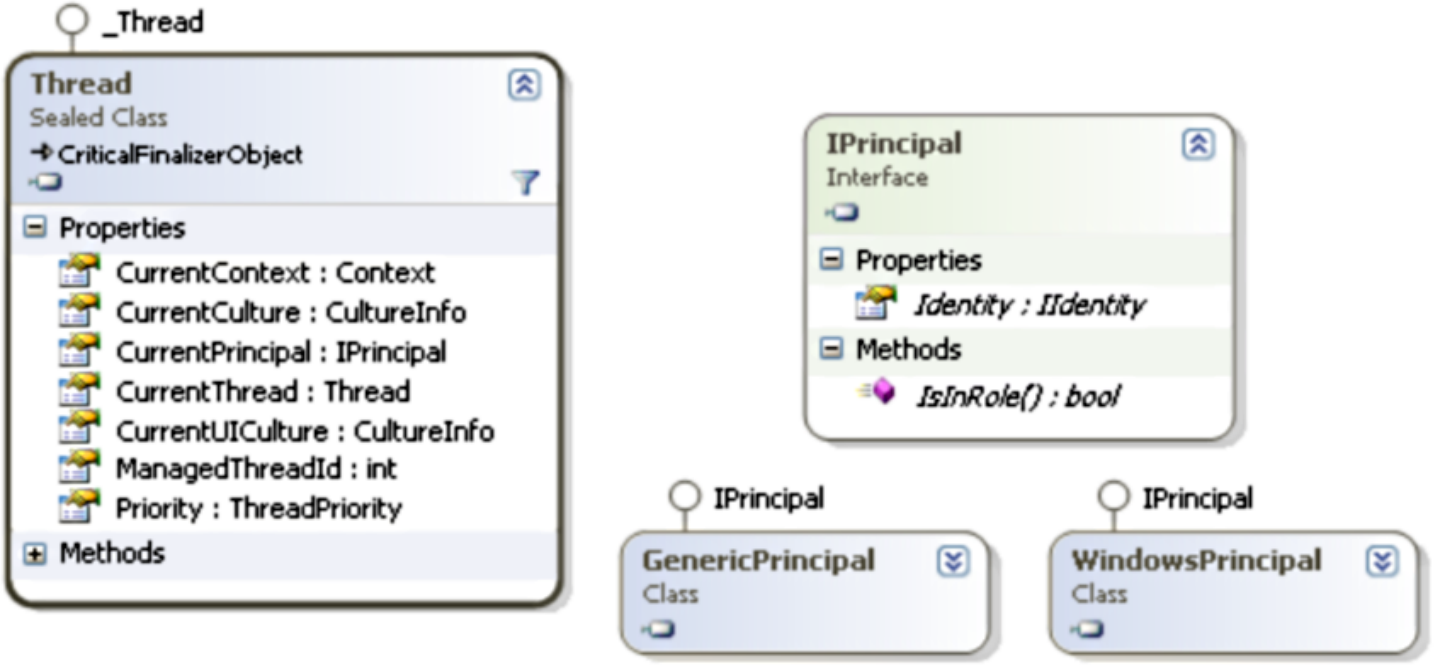
- Coesão
  - "É a medida de quão fortemente **relacionadas** e **focadas** estão as várias **responsabilidades** de um módulo."



  - "Módulos com alta coesão tendem a ser preferidos porque a alta coesão está associada a várias características desejáveis do software, incluindo robustês, confiabilidade, reusabilidade e compreensibilidade, ao passo que a baixa coesão é associada a caracteristicas indesejáveis, tais como a dificuldade de manter, testar, reutilizar e até mesmo de compreender."
  - en.wikipedia.org/wiki/Cohesion_(computer_science)

Conceitos de Orientação a Objetos

- Acoplamento
  - "É o grau em que cada módulo do programa **depende** de cada um dos outros módulos."



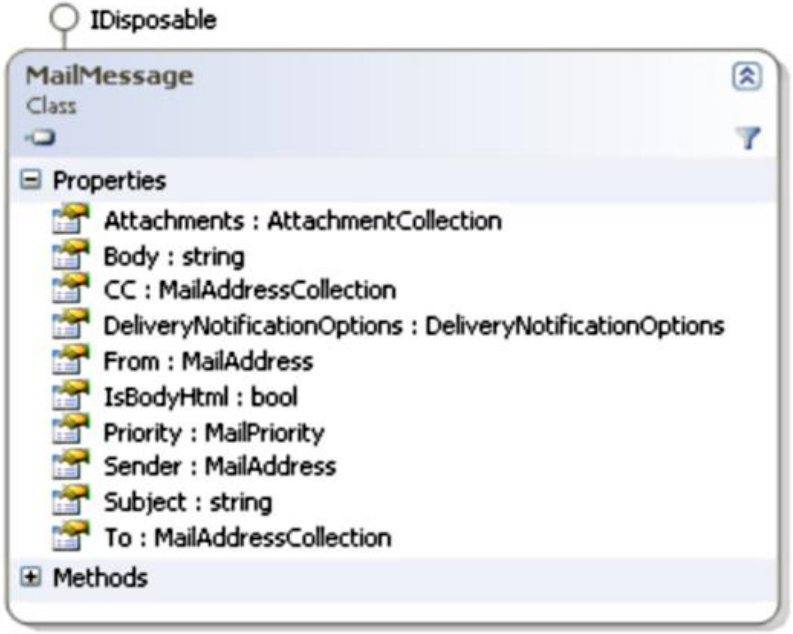
- en.wikipedia.org/wiki/Coupling_(computer_science)

Conceitos de Orientação a Objetos

- Acoplamento (cont.)
  - “Baixo acoplamento refere-se ao relacionamento no qual um módulo interage com outro módulo através de uma interface estável e não precisa se preocupar com outras implementações internas do módulo…”
  - “Com o baixo acoplamento, uma alteração em um módulo não exigirá a mudança na implementação de outro módulo.”
  - “O baixo acoplamento é geralmente um sinal de um sistema bem estruturado e, quando combinado com a alta coesão, suporta os objetivos globais de legibilidade e manutenibilidade.”

Conceitos de Orientação a Objetos

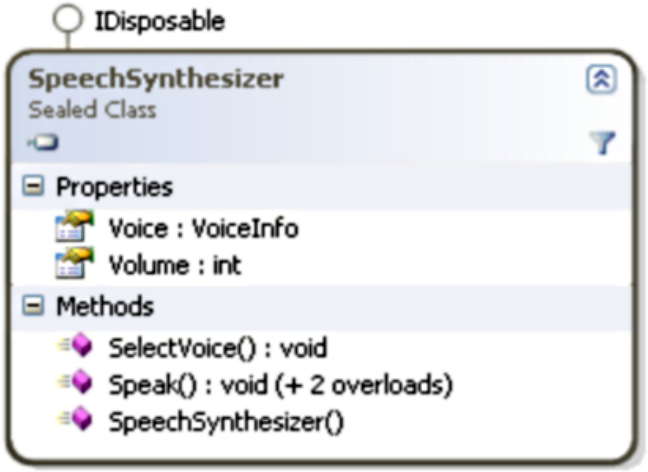
- Abstração
  - "É um mecanismo e prática para reduzir e ocultar detalhes, de tal forma que possamos nos concentrar em poucos conceitos de cada vez."



- en.wikipedia.org/wiki/Abstraction_(computer_science)

Conceitos de Orientação a Objetos

- Encapsulamento
  - “É o ato de ocultar os mecanismos internos e estruturas de dados de um componente de software através do uso de uma interface definida, de tal forma que os usuários do componente (outras partes do *software*) precisem saber somente o que o componente faz, sem se tornarem dependentes dos detalhes sobre como o componente faz seu trabalho.”



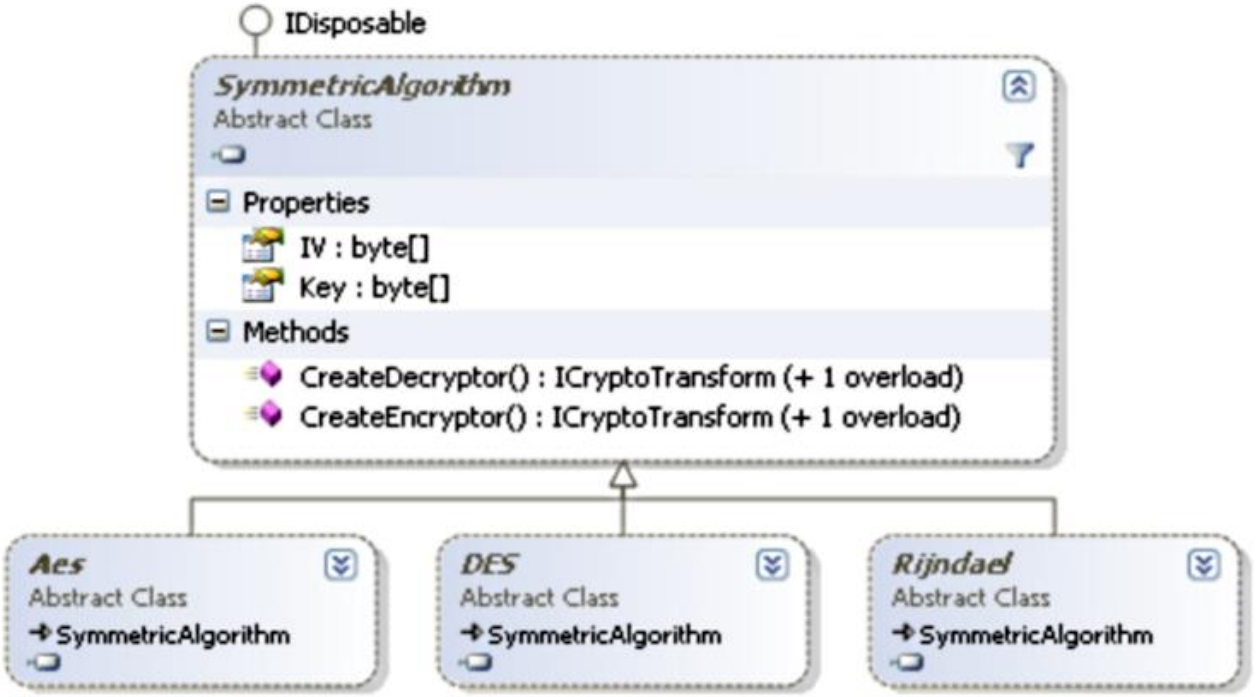

```
// Reference System.Speech

new SpeechSynthesizer()
 .Speak(
  “Olá, pessoal!!!”
 );
```

- en.wikipedia.org/wiki/Encapsulation_(computer_science)

Conceitos de Orientação a Objetos

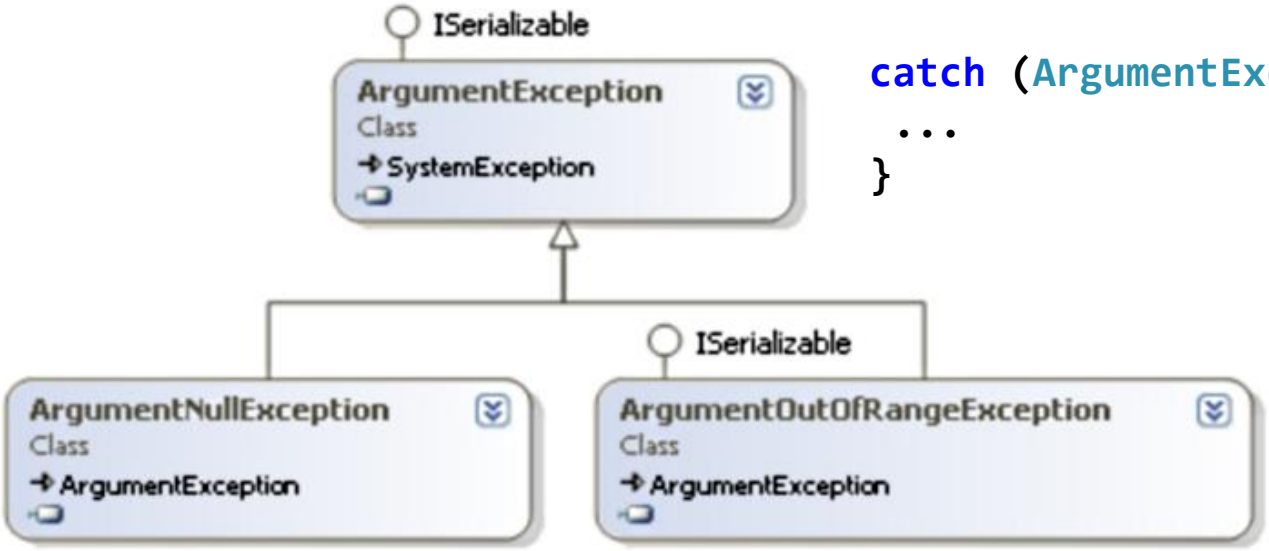
- Herança
  - "É uma forma de criar novas classes (cujas instâncias são chamadas de objetos) usando classes que já foram definidas."



- en.wikipedia.org/wiki/Inheritance_(computer_science)

Conceitos de Orientação a Objetos

- Polimorfismo
  - "É a habilidade de um tipo B, de se parecer e se comportar como um outro tipo A. Em linguagens fortemente tipadas, isto geralmente significa que o tipo B é de alguma forma derivado do tipo A, ou que o tipo B implementa a mesma interface que o tipo A."




```
catch (ArgumentException) {
  ...
}
```

- en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming

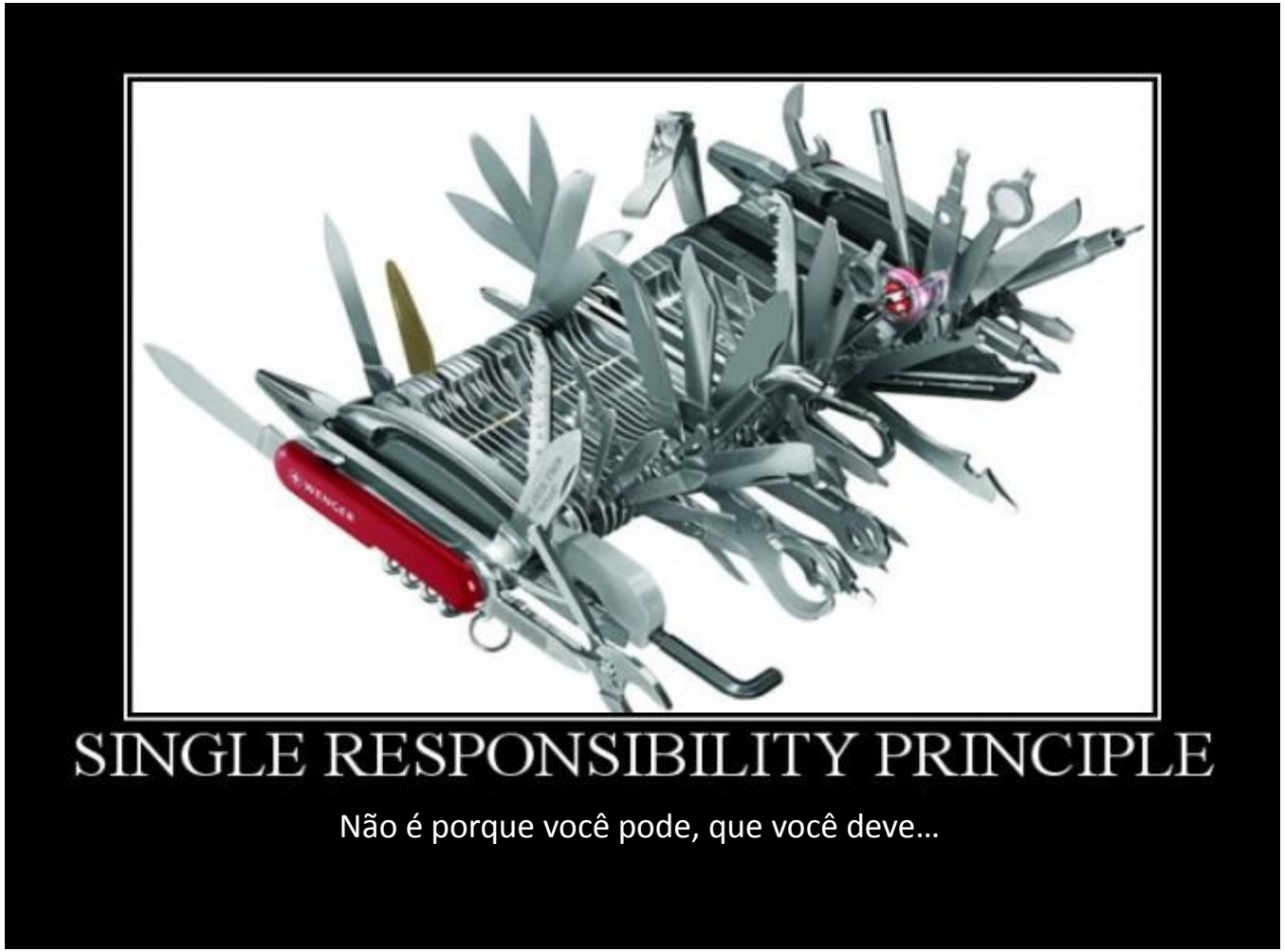
Princípios SOLID

- Alguns princípios de projeto OO bem conhecidos
- Bem explicados por Robert Martin
  - Série de artigos
  - Livro: Agile Software Development: Principle, Patterns and Practices.
- O que esses princípios tem a ver com desenvolvimento ágil?
  - Ajudam a “abraçar” as mudanças.

Princípios SOLID

- **S**RP: The Single Responsibility Principle
  - en.wikipedia.org/wiki/Single_responsibility_principle
- **O**CP: The Open/Closed Principle
  - en.wikipedia.org/wiki/Open/closed_principle
- **L**SP: The Liskov Substitution Principle
  - en.wikipedia.org/wiki/Liskov_substitution_principle
- **I**SP: The Interface Segregation Principle
  - www.oodesign.com/interface-segregation-principle.html
- **D**IP: The Dependency Inversion Principle
  - en.wikipedia.org/wiki/Dependency_inversion_principle

O Princípio da Responsabilidade Única

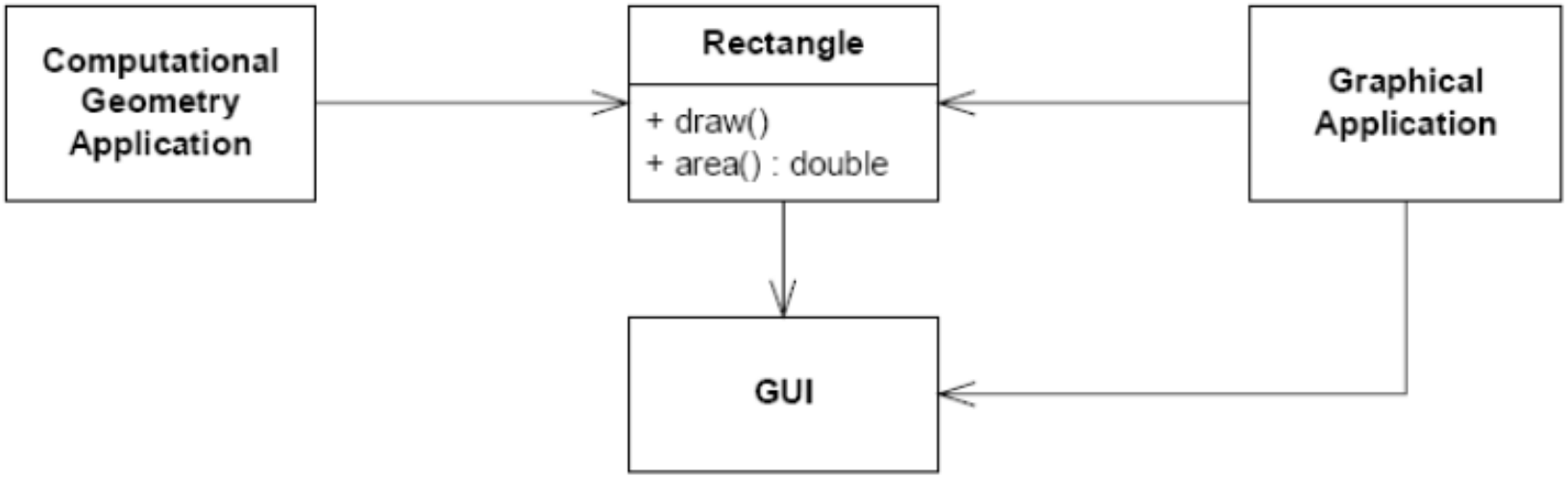


O Princípio da Responsabilidade Única

- **S**RP: The Single Responsibility Principle
- Uma classe deve ter somente uma razão para ser mudada.
- Quando um requisito muda, tal mudança se manifestará como mudança em responsabilidades das classes.
- Se uma classe assume mais de uma responsabilidade, então:
  - haverá mais de uma razão para essa classe mudar;
  - mudanças em uma responsabilidade podem impedir ou diminuir a habilidade da classe para realizar as outras.
- Fragilidade!

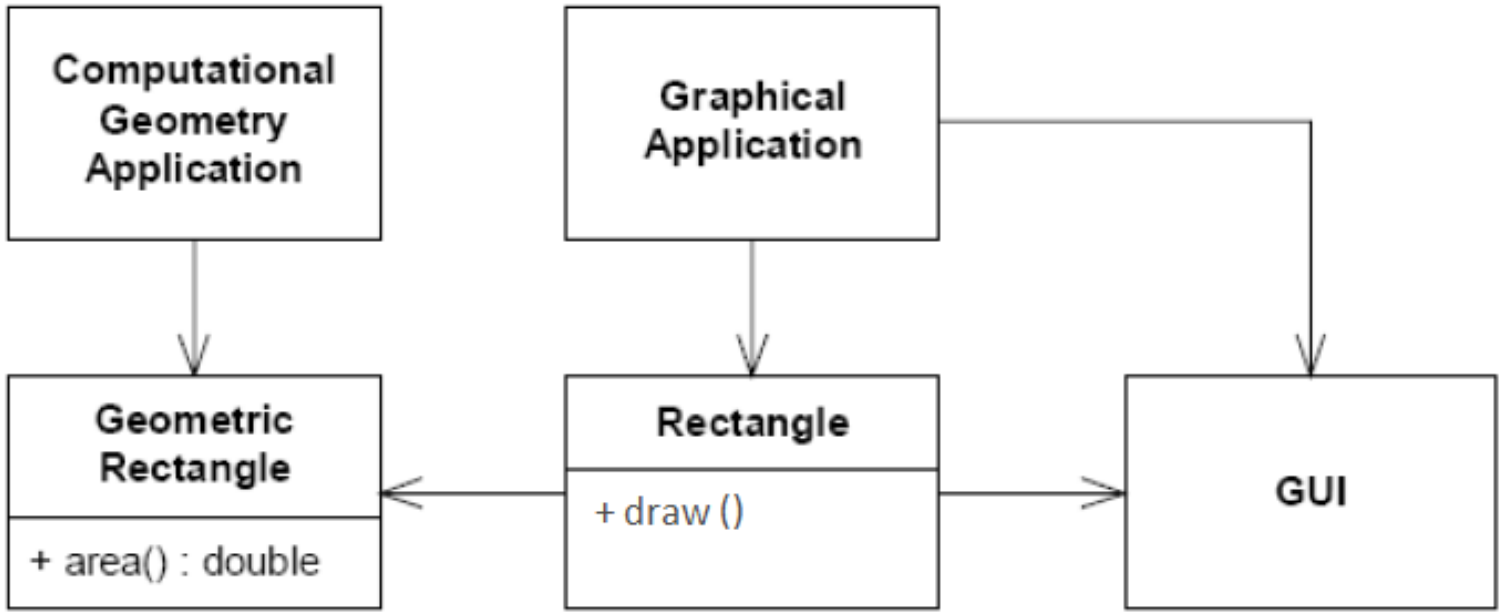
O Princípio da Responsabilidade Única

- Mais de uma responsabilidade na classe Rectangle.



Não está de acordo com o princípio!

O Princípio da Responsabilidade Única



Está de acordo com o princípio!

O Princípio da Responsabilidade Única

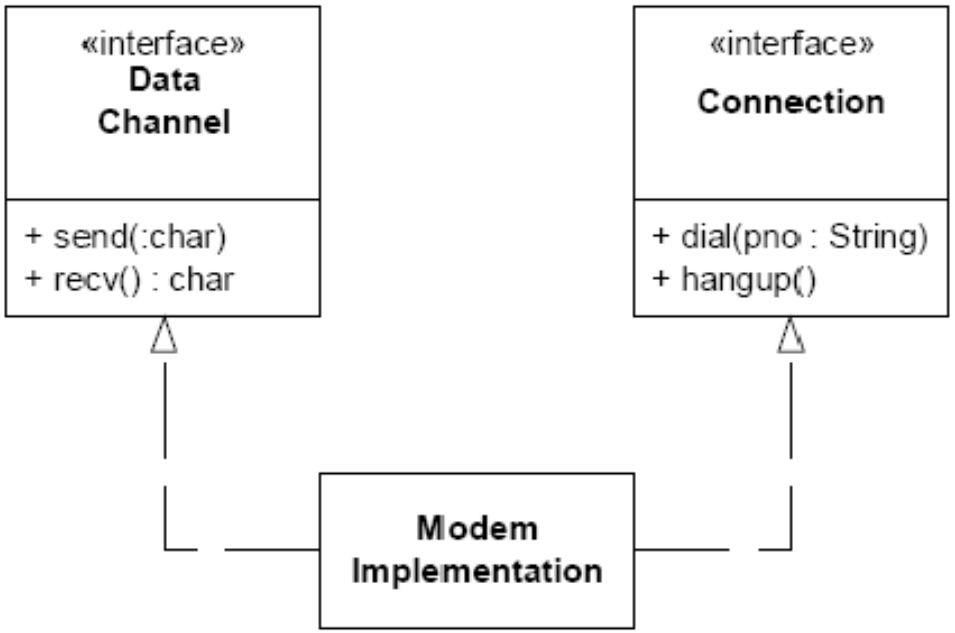
- No contexto do princípio SRP, responsabilidade é definida como sendo “razão para mudar”.
- Muitas vezes é difícil de se distinguir diferentes responsabilidades.

```
interface Modem
{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```

- Quantas responsabilidades?

O Princípio da Responsabilidade Única

- As responsabilidades devem ser separadas?
  - Depende da forma como a aplicação muda.



O Princípio da Responsabilidade Única

- Como fazer?
  - Reconheça a classe como um bloco de construção de software
  - Crie classes que
    - façam uma única coisa ou mudem por um única razão
    - sejam pequenas
    - sejam fáceis de ler e entender
    - sejam fáceis de manter
    - sejam potencialmente fáceis de testar
    - sejam fáceis de se substituir
  - Garanta a alta coesão
    - Cada membro de classe deve estar diretamente relacionado com o nome da classe
  - Lembre-se que se trata de um princípio, não uma regra. Use sua experiência, conhecimento e aplique o bom senso.

O Princípio da Responsabilidade Única

- Benefícios
  - O código é menor
    - Mais fácil de ler
    - Mais fácil de entender
    - Mais fácil de manter
  - O código é mais fácil de testar
  - Alterações são mais fáceis de gerenciar
    - O código é mais fácil de se substituir
  - A contenção de código por múltiplos desenvolvedores é reduzida.

O Princípio Aberto/Fechado

O Princípio Aberto/Fechado

- **O**CP: The Open/Closed Principle
- Entidades de software (classes, módulos, funções, etc.) devem ser “abertas” para extensão e “fechadas” para modificação.
- Aberto para extensão
  - Quando requisitos mudam, somos capazes de estender o módulo com novos “comportamentos” que satisfazem as mudanças.
- Fechado para modificação
  - Estender o comportamento de um módulo não consiste em alterar seu código fonte.

O Princípio Aberto/Fechado

- As classes devem estar **abertas** para extensão, mas **fechadas** para modificação.

- Adicione novos comportamentos através da extensão.
- Não altere o código existente!

O Princípio Aberto/Fechado

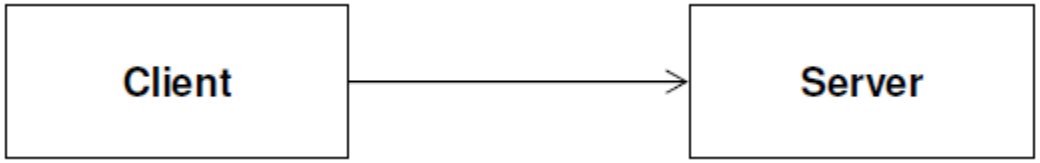
- Permite que a classe seja facilmente estendida sem alterar seu comportamento.
- **Abstração** e **Polimorfismo** são os mecanismos primários por trás desse princípio.
- Abstrações são fixas (classes abstratas/interfaces) ...

  ... e representam um grupo ilimitado de possíveis comportamentos (classes derivadas).

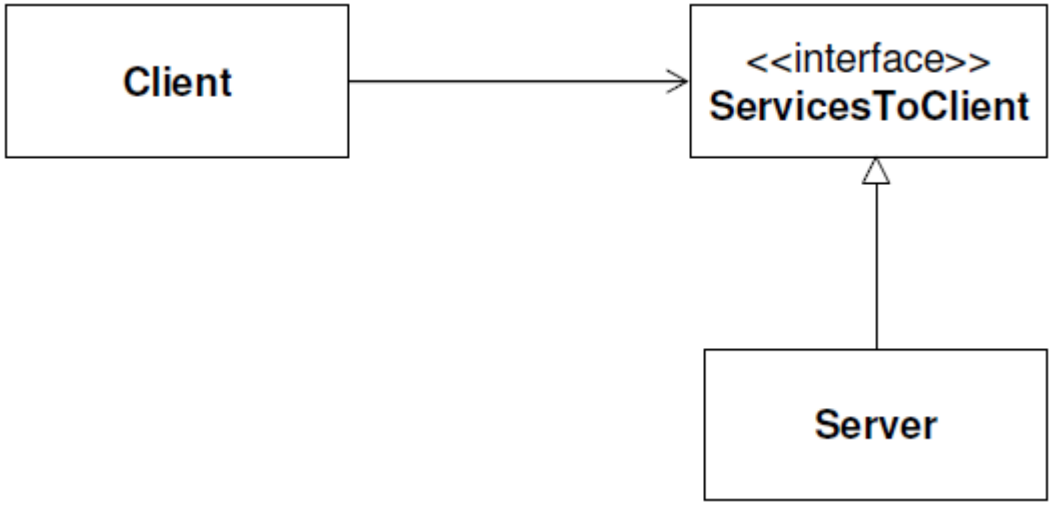
O Princípio Aberto/Fechado

- Pedido do cliente:
  - ✓ “Por favor, escreva o *log* de todos os erros para um arquivo.”
- Lá vem o cliente:
  - ✓ “Agora, estou precisando fazer o *log* de erros para o *Event Log*.”
- Lá vem o cliente de novo:
  - ✓ “Na verdade, eu gostaria que todos os erros fossem logados para...”

O Princípio Aberto/Fechado



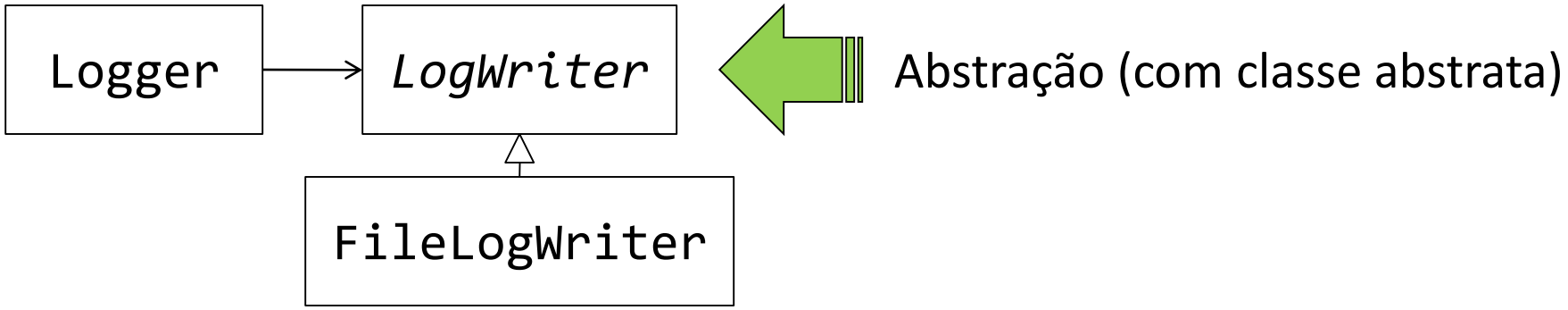
Client não é aberta e nem fechada



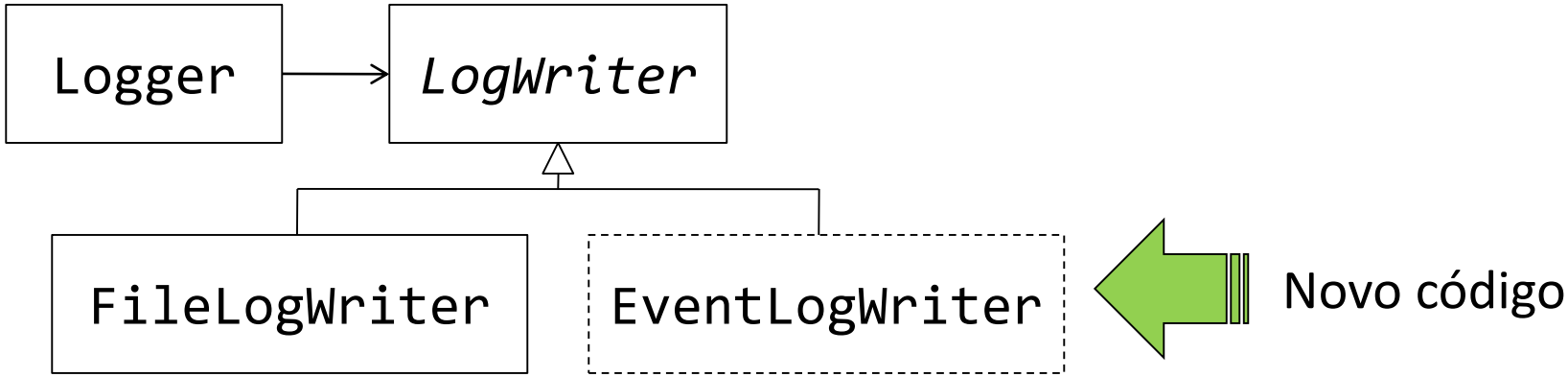
Client é aberta e fechada

O Princípio Aberto/Fechado

- A classe Logger

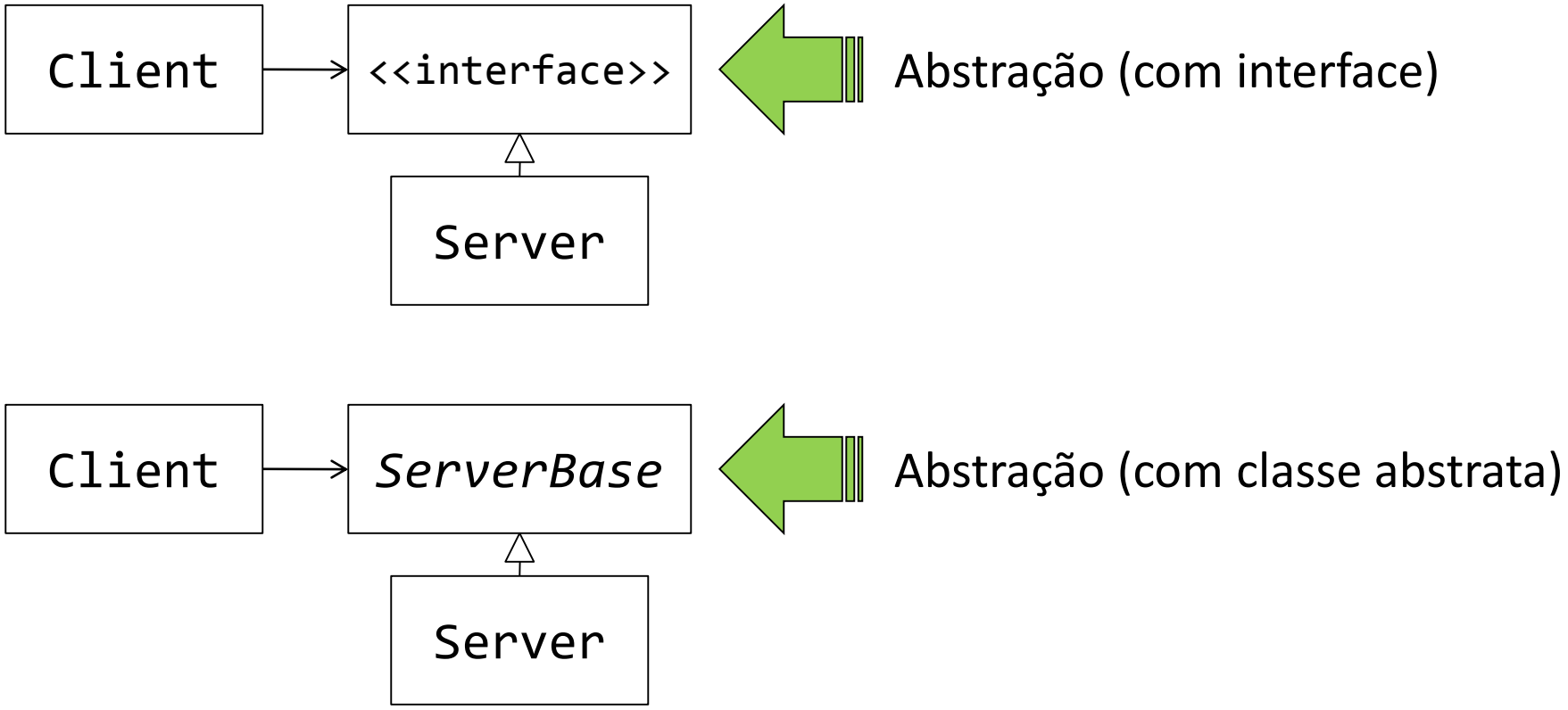


- Alterações são feitas pela adição e não pela modificação de código

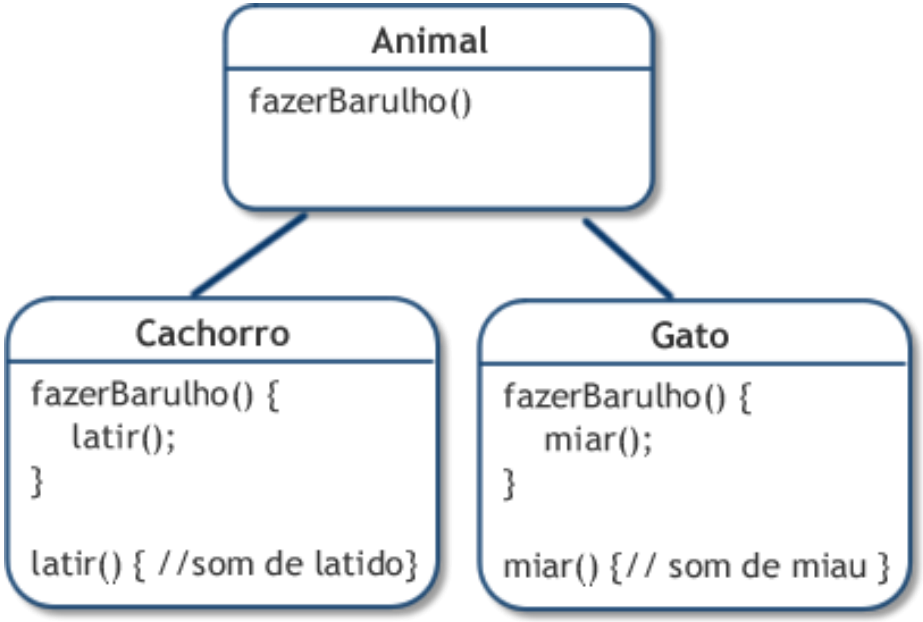


O Princípio Aberto/Fechado

- Abstrações



Como fazer?

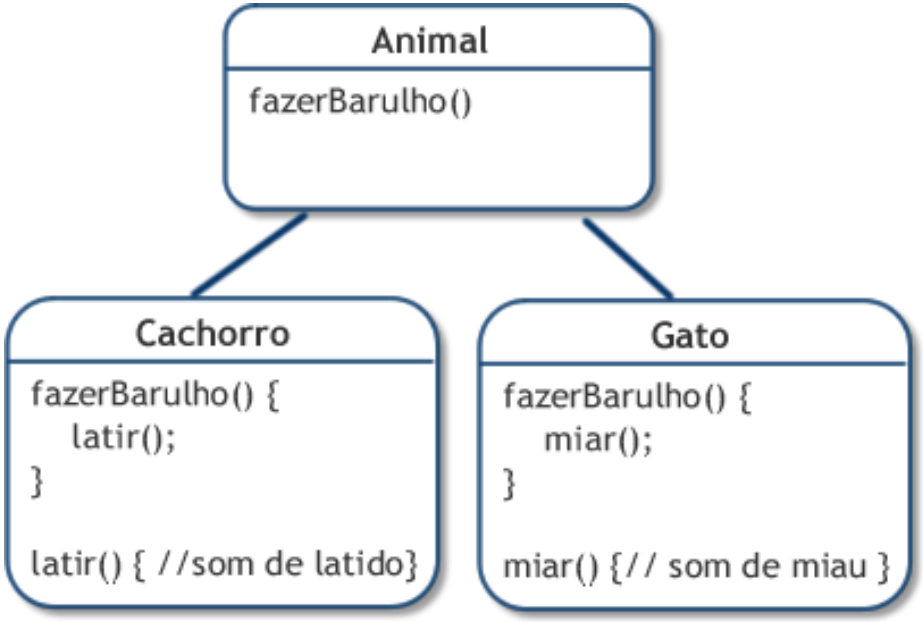


Como fazer um animal miar a partir do código abaixo?

```
Cachorro c = new Cachorro();

c.latir();
```

Como fazer?

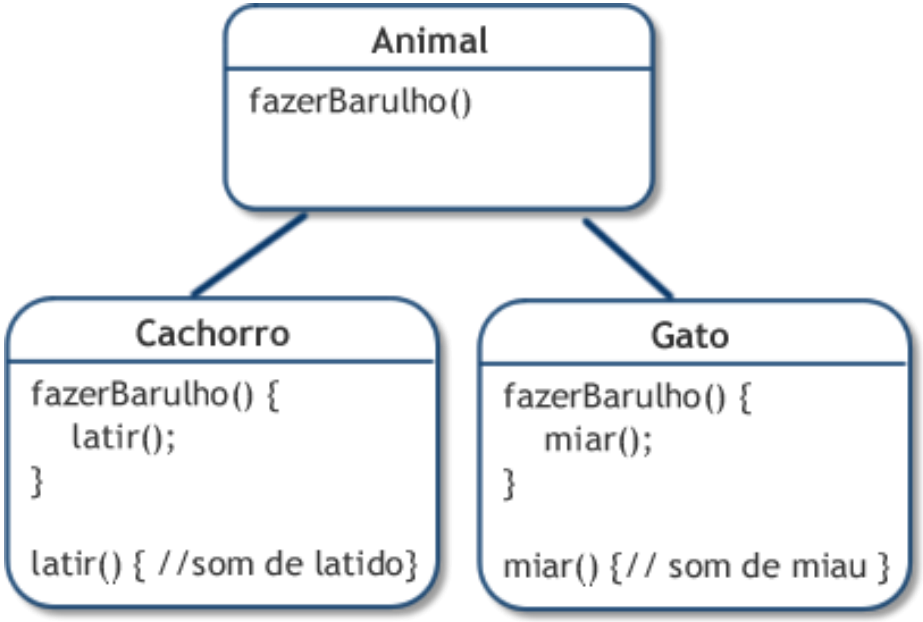


E se fizermos assim?

```
Animal c = new Cachorro();

c.fazerBarulho();
```

Como fazer?



Melhorando ainda mais...

```
Animal c = getAnimal();

c.fazerBarulho();
```

O Princípio Aberto/Fechado

- Como fazer?
  - Implemente códigos voláteis por meio de abstrações
    - Use classes abstratas/interfaces como supertipo para implementar códigos que são propensos à mudanças.
  - Lembre-se que trata-se de um princípio, não uma regra. Use sua experiência, conhecimento e aplique o bom senso.

O Princípio Aberto/Fechado

- Benefícios
  - O design é mais estável
    - Código existente (e funcionando) não é alterado
    - Mudanças são feitas pela adição e não pela modificação de código existente
    - As alterações são isoladas e o impacto da mudança não é propagado por toda a aplicação
  - O código é mais fácil de testar
  - As alterações são mais fáceis de gerenciar
    - É mais simples substituir código
    - O design é extensível
  - O código é potencialmente reutilizável.

O Princípio Aberto/Fechado

- Observações
  - Nunca conseguiremos projetar um módulo fechado para todo tipo de mudança.
  - Como não pode ser completo, o fechamento deve ser estratégico
    - o *designer* deve “fechar” o módulo para os tipos de mudanças mais prováveis de acontecer
  - Não devemos exagerar no uso de abstrações, caso contrário, o design pode ficar **desnecessariamente complexo**!

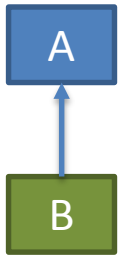
O Princípio da Substituição de Liskov

O Princípio da Substituição de Liskov

- **L**SP: The Liskov Substitution Principle
- Subtipos devem poder substituir seus tipos base.
- Vimos que os mecanismos primários do OCP são abstração e polimorfismo, que são apoiados pelo uso de herança.
- O LSP se preocupa com o uso adequado da herança
  - Como melhor usar herança?
  - Quais as armadilhas que podem nos levar a usar herança de forma a violar o OCP?

O Princípio da Substituição de Liskov

- Considere que temos uma função *f.*
- *f* recebe como argumento um objeto de uma classe base A: *f* (A)
- Suponha que B é uma subclasse de A...



... e que, quando um objeto de B é passado para *f* no lugar de A, ocorre um mau funcionamento de *f*:

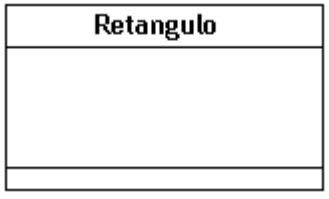
f (B) → XXX

- Então, B viola o LSP.
- B torna o *design* frágil.

O Princípio da Substituição de Liskov

- Exemplo (http://www.macoratti.net/11/05/oop_lsp1.htm)
  - Definindo uma classe **Retangulo** que tem duas propriedades: largura e altura.

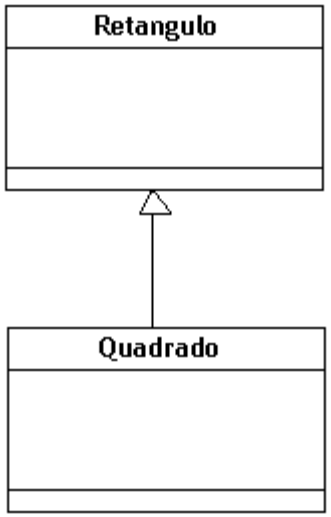
```
class Retangulo
{
      protected int m_largura;
      protected int m_altura;
      public void setLargura(int largura)
      { m_largura = largura;}
      public void setAltura(int altura)
      {  m_altura = altura;  }
      public int getLargura()
      { return m_largura;   }
      public int getAltura()
      { return m_altura;    }
      public int getArea()
      { return m_largura * m_altura; }
}
```



O Princípio da Substituição de Liskov

- Exemplo (cont.)
  - A classe **Quadrado** *é um* **Retangulo** que tem uma característica especial: a largura e a altura são iguais.

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



O Princípio da Substituição de Liskov

- Exemplo (cont.)
  - Ao usar a classe Quadrado, podemos ter uma inconsistência.

```
class Program
{
   private static Retangulo getNovoRetangulo() {
      return new Quadrado();
   }
   public static void main(string[] args)  {
      //vamos criar um novo retangulo
      Retangulo r = Program.getNovoRetangulo();
      //definindo a largura e altura do retangulo
      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
   }
}
```

O Princípio da Substituição de Liskov

- Entendendo o problema…
  - Ao definir a largura para 10, acabamos definindo também a altura, deixando a área com valor de 100 e não 50 como era esperado.
  - Nesse 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!
  - Isso **viola** o principio de Liskov no qual uma classe filha (Quadrado) deve poder substituir uma classe base (Retangulo).

O Princípio da Substituição de Liskov

- Tirando nossas conclusões...
  - Este exemplo mostra uma violação clara do princípio **LSP** onde a inclusão da classe **Quadrado** herdando de **Retangulo** mudou o comportamento da classe base, violando, assim, o **Princípio Open-Closed**.
  - O principio **LSP** é uma extensão do **Princípio Open-Closed**.
  - Isso significa que precisamos ter certeza de que as novas classes derivadas estão estendendo as classes bases sem alterar seu comportamento.

O Princípio da Substituição de Liskov

- Como fazer?
  - Certifique-se de que
    - o tipo base pode ser substituído por um tipo derivado
    - tipos derivados sejam substituíveis por seus tipos base
  - Evite checagem de tipos em runtime (*instanceof*, p. ex.).
  - Lembre-se que se trata de um princípio, não uma regra. Use sua experiência, conhecimento e aplique o bom senso.

O Princípio da Substituição de Liskov

- Benefícios
  - O design é mais flexível
    - O tipo base pode ser confiavelmente substituído por um tipo derivado.
    - É potencialmente mais fácil para testar.
  - É requerido para suportar o princípio Fechado/Aberto.

O Princípio da Segregação de Interfaces

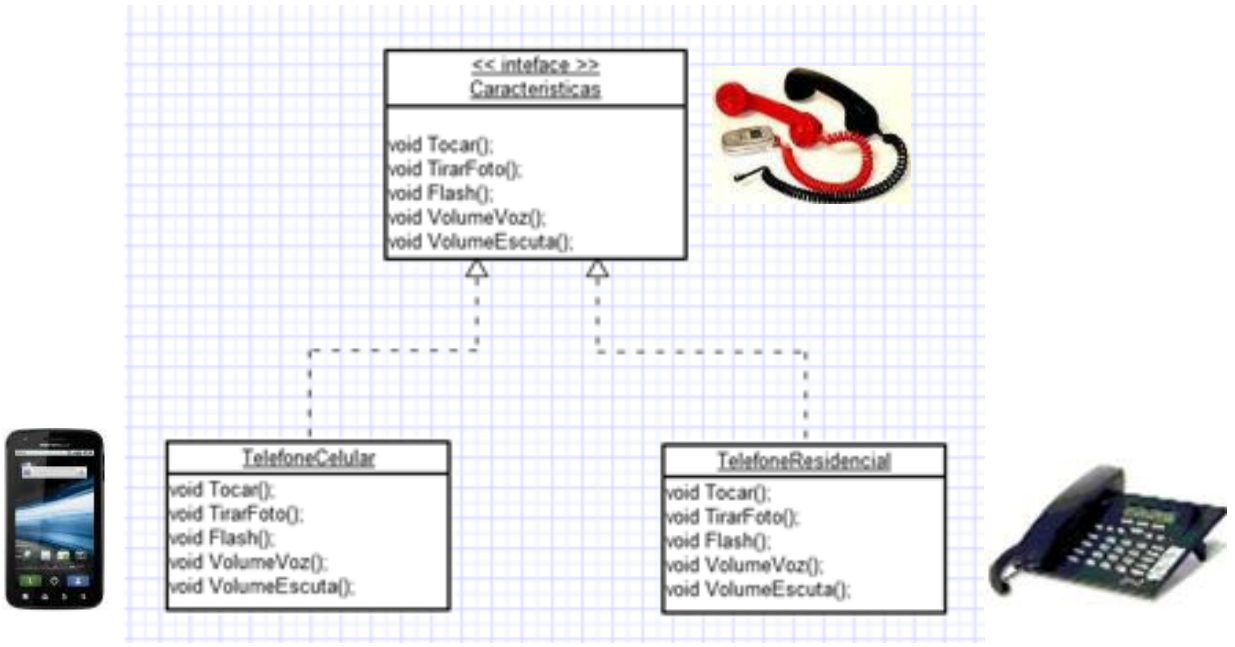
Adeque suas interfaces às necessidades dos clientes individuais.

O Princípio da Segregação de Interfaces

- ISP: The Interface Segregation Principle
- Clientes de interfaces não devem ser forçados a implementar contratos de interfaces que não requeiram.
- Em outras palavras, clientes não devem ser forçados a dependerem 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 quebrada em mais de uma interface.

O Princípio da Segregação de Interfaces

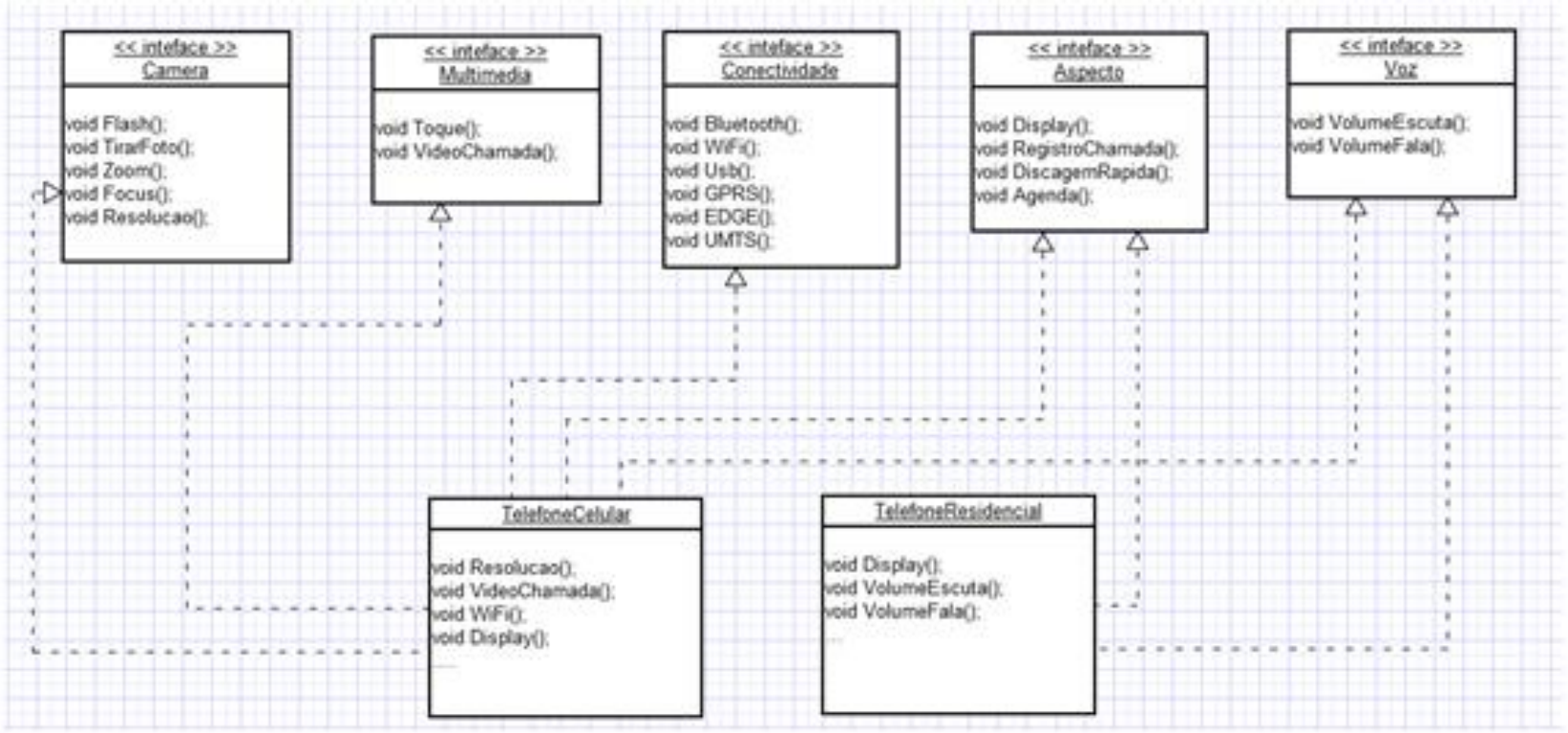
- Exemplo com interface “gorda” (poluída)



http://engenhariadesoftwareagil.com/

O Princípio da Segregação de Interfaces

- Separando as interfaces...



http://engenhariadesoftwareagil.com/

O Princípio da Segregação de Interfaces

- Classes com interfaces “gordas” podem causar acoplamentos estranhos e problemáticos entre seus clientes.
  - Quando um cliente força uma mudança na interface, os outros clientes são afetados.
- Clientes só devem depender de métodos que eles chamam.

O Princípio da Segregação de Interfaces

- Como fazer?
  - Considere as necessidades do cliente quando for desenvolver interfaces.
  - Crie interfaces específicas para o cliente.

O Princípio da Segregação de Interfaces

- Benefícios
  - A coesão é aumentada
    - Os clientes podem exigir interfaces coesas.
  - O *design* é mais estável
    - As alterações são isoladas e o impacto da mudança não é propagado por toda a aplicação.
  - Suporta o princípio da Substituição de Liskov.

O Princípio da Inversão de Dependência

Você soldaria uma lâmpada diretamente na fiação elétrica na parede?

O Princípio da Inversão de Dependência

- **D**IP: The Dependency Inversion Principle

1. Módulos de nível mais alto não devem depender de módulos de nível mais baixo. Ambos devem depender de abstrações.
2. Abstrações não devem depender dos detalhes de implementação. Detalhes é que devem depender de abstrações.

O Princípio da Inversão de Dependência

- Em uma aplicação temos
  - classes de baixo nível que programam operações básicas; e
  - classes de alto nível que encapsulam a lógica complexa e depende das classes de baixo nível.
- Uma maneira natural de programar esta aplicação seria escrever as classes de baixo nível primeiramente e depois escrever as classes de alto nível mais complexa.
- Como as classes de alto nível são definidas em função das outras previamente escritas, este parece ser o caminho lógico para fazer esta programação.

O Princípio da Inversão de Dependência

- Porém, este não é um bom design, deixa o código rígido, frágil e imóvel.
- Para evitar tais problemas, podemos introduzir uma camada de abstração entre as classes de alto nível e classes de baixo nível.
- Os módulos de alto nível contêm a lógica complexa que não deve depender dos módulos de baixo nível.
- A camada de abstração não deve ser criada com base em módulos de baixo nível. Os módulos é que devem ser criados com base na camada de abstração.

O Princípio da Inversão de Dependência

- De acordo com este princípio, a maneira de projetar uma estrutura de classe é começar a partir de módulos de alto nível para os módulos de baixo nível:

Classes de Alto Nível ->

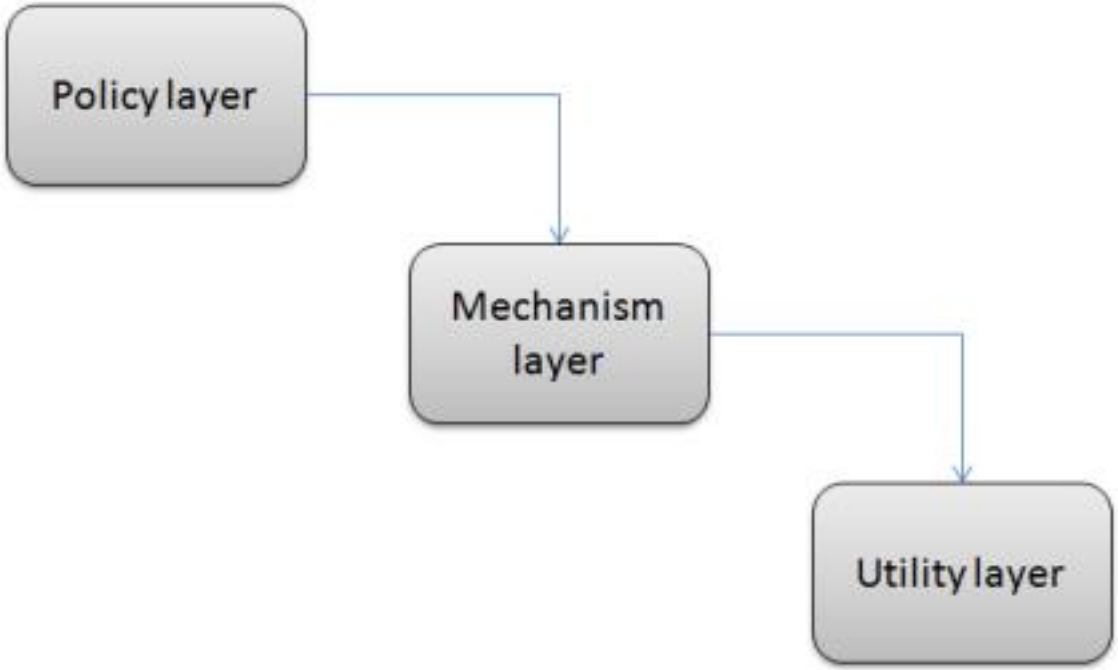
Camada de Abstração ->

Classes de Baixo Nível

- Robert Martin afirma que, se as dependências são invertidas, temos um design OO, senão, temos um design procedural.

O Princípio da Inversão de Dependência

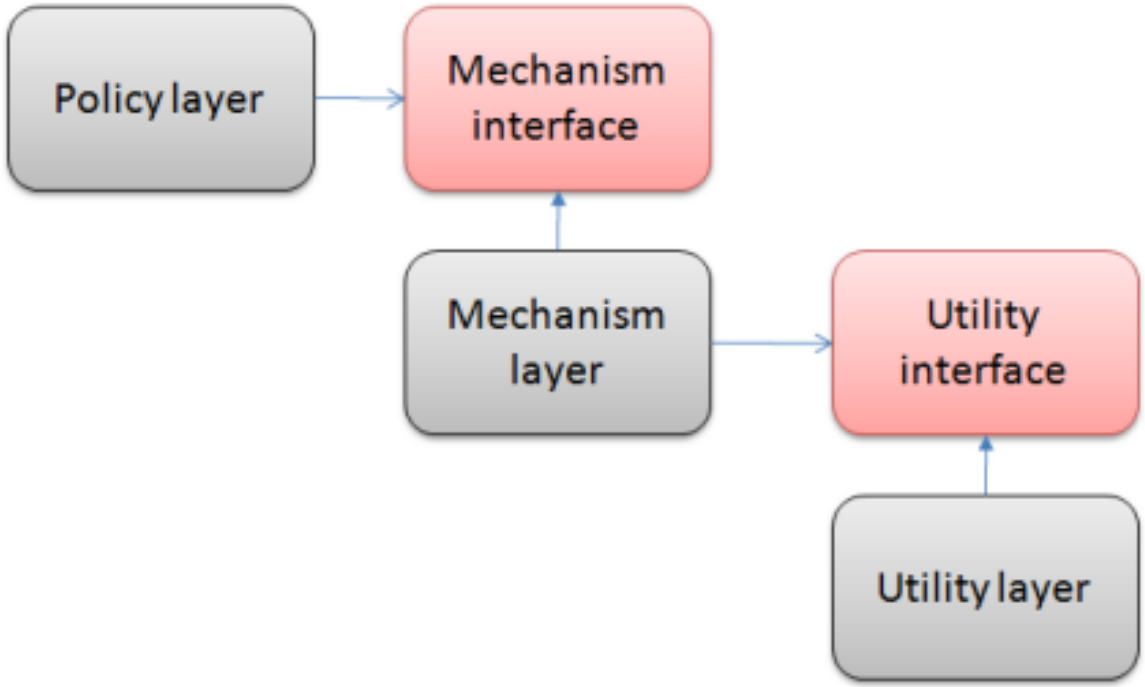
- Considere uma aplicação estruturada em camadas.



- Observe as dependências!

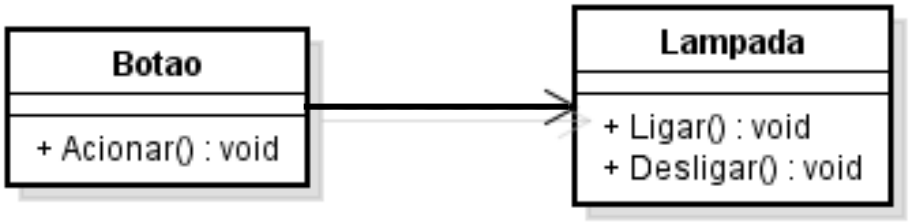
O Princípio da Inversão de Dependência

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



O Princípio da Inversão de Dependência

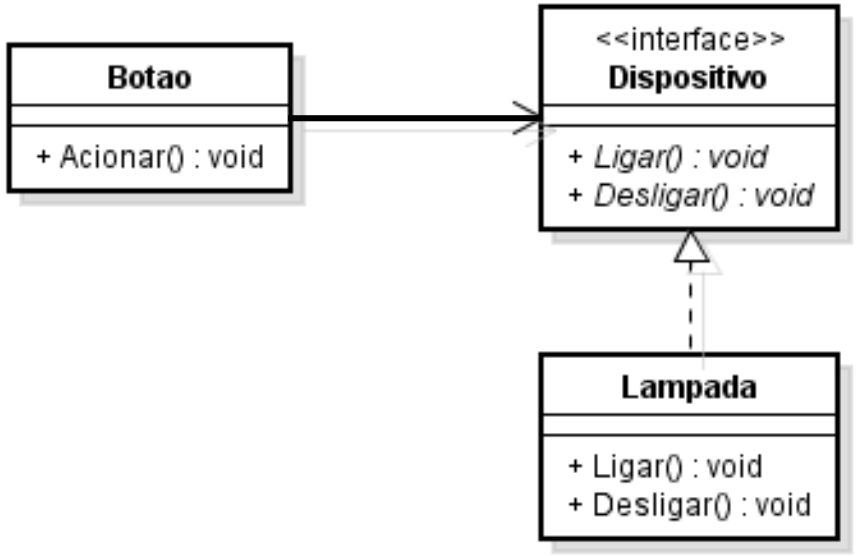
- Um exemplo mais concreto




```
public class Botao
{
        private Lampada lampada;
        public void Acionar();
        {
                if (/*some condition*/)
                    lampada.ligar();
        }
}
```

O Princípio da Inversão de Dependência

- Invertendo a dependência
  - **Botao** passa a depender de uma abstração (interface).




```
public interface Dispositivo
{
   void Ligar();
   void Desligar();
}
public class Botao
{
        private Dispositivo dispositivo;
        public void Acionar();
        {
                if (/*some condition*/)
                    dispositivo.ligar();
        }
}
```

Botão pode acionar diversos tipos de dispositivos, além de lâmpadas: motores, eletrodomésticos, etc.

O Princípio da Inversão de Dependência

- Como fazer?
  - Crie abstrações.
  - Dependa de abstrações
    - Declare parâmetros e variáveis em forma de interfaces ou classes abstratas.
    - Derive somente de classes abstratas/interfaces.
    - Sobrescreva somente membros abstratos.

O Princípio da Inversão de Dependência

- Benefícios
  - É fácil gerenciar a mudança
    - É mais fácil substituir o código
    - O design é extensível
  - O código é potencialmente mais fácil de testar.
  - O princípio Fechado/Aberto depende dele.






Revisão

1. Qual a motivação para o surgimento dos princípios de design OO?
2. Qual a diferença entre coesão e acoplamento?
3. Pelo princípio do Mínimo Conhecimento, quais elementos podem ser acessados a partir de um método?
4. Em que consiste o princípio de Hollywood?
5. Por que devemos preferir composição à herança?
6. O "aberto" do princípio open/closed está relacionado a quê?
7. Por que é melhor trabalhar com interfaces mais específicas do que mais genéricas?
8. Qual a motivação para as classes terem somente uma responsabilidade?
9. Em linhas gerais, como devemos implementar o princípio de Substituição de Liskov?
10. Explique o Princípio da Inversão de Dependência.

Referências

- Agile Software Development, Principles, Patterns, and Practices. Robert C. Martin.
- Use a Cabeça! Padrões de Projetos. Elisabeth Freeman.
- Use a Cabeça! Análise e Projeto Orientado ao Objeto. Gary Pollice.
- The Principles of OOD. Robert C. Martin. http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- Treinamento em Orientação a Objetos. http://www.mindworks.com.br/
- http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Princípios de Projeto Orientado a Objetos

Obrigado!!!

Marum Simão Filho
marumsimao@gmail.com