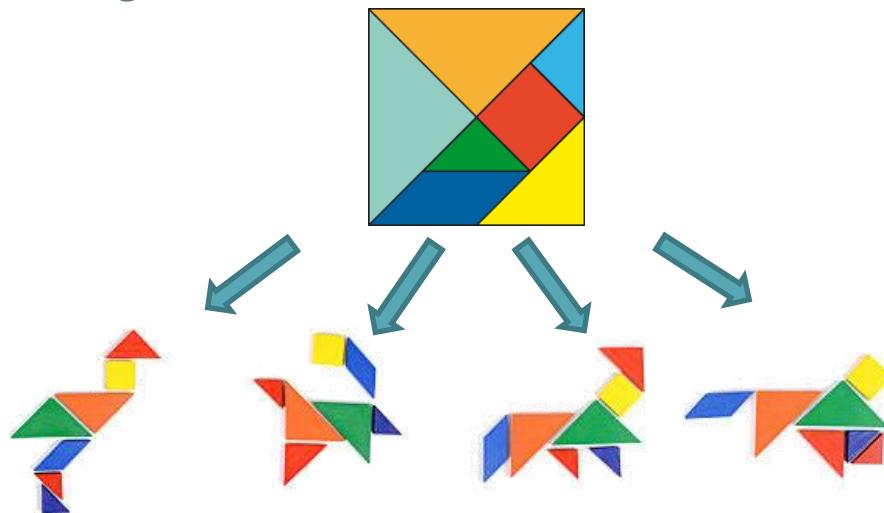


# Engenharia de Software

## Introdução a Padrão de Projeto e Padrões GOF



Baseado em Notas de Aula de: Eduardo Figueiredo (UFMG) e Carlos Lucena (PUC-RIO)

# Técnicas de Reuso



# Técnicas de Reuso

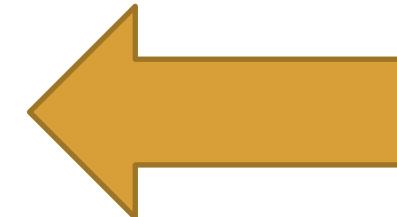
Algumas técnicas que propiciam o reuso de projeto e implementação em sistemas são:

- ◎ Padrões de Projeto (Design Patterns)
- ◎ Frameworks
- ◎ Linhas de Produto
- ◎ Bibliotecas de Software

# Técnicas de Reuso

Algumas técnicas que propiciam o reuso de projeto e implementação em sistemas são:

- ◎ Padrões de Projeto (Design Patterns)
- ◎ Frameworks
- ◎ Linhas de Produto
- ◎ Bibliotecas de Software



# Padrões de Projeto

## O Cenário



Manoel trabalha em uma empresa que criou um jogo de simulação de um lago com patos. O jogo pode mostrar uma grande variedade de espécies de patos **nadando e produzindo sons**.



*Duck*



*MallardDuck*



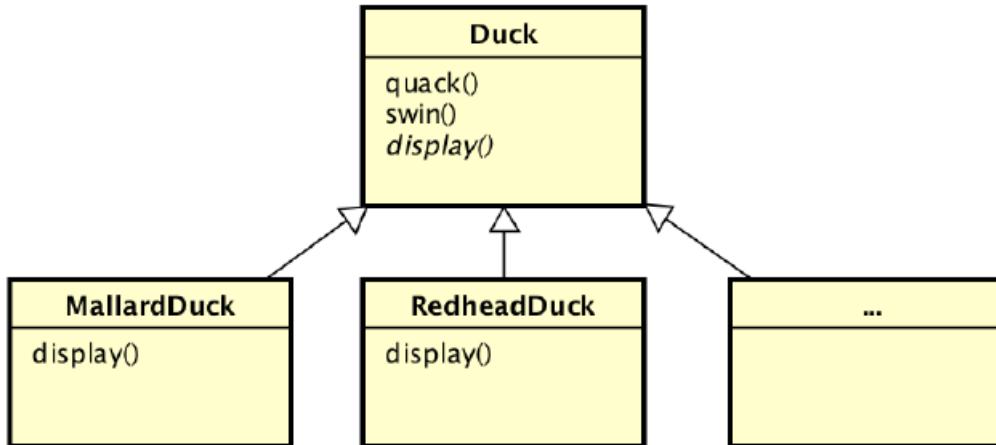
*RedheadDuck*

# Padrões de Projeto

## O Cenário



Manoel trabalha em uma empresa que criou um jogo de simulação de um lago com patos. O jogo pode mostrar uma grande variedade de espécies de patos **nadando e produzindo sons**. Os arquitetos do sistema usaram técnicas OO e chegaram ao seguinte modelo de classes.

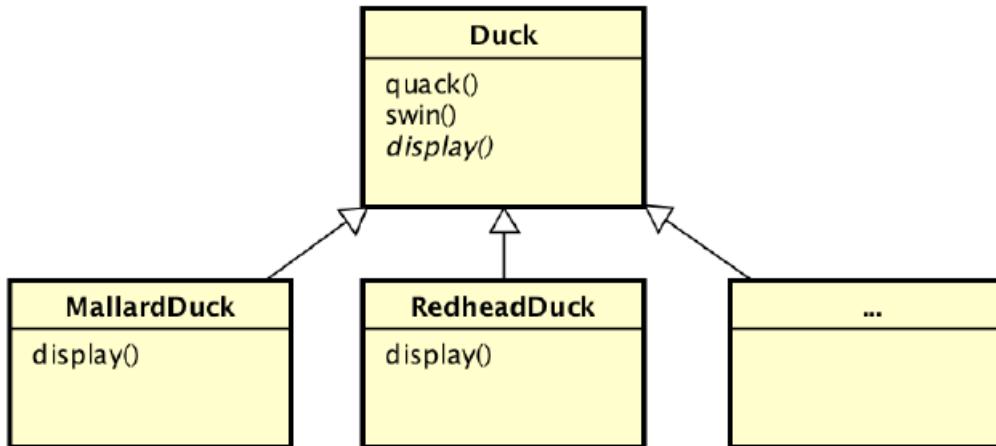


# Padrões de Projeto

## O Cenário



Manoel trabalha em uma empresa que criou um jogo de simulação de um lago com patos. O jogo pode mostrar uma grande variedade de espécies de patos **nadando e produzindo sons**. Os arquitetos do sistema usaram técnicas OO e chegaram ao seguinte modelo de classes.

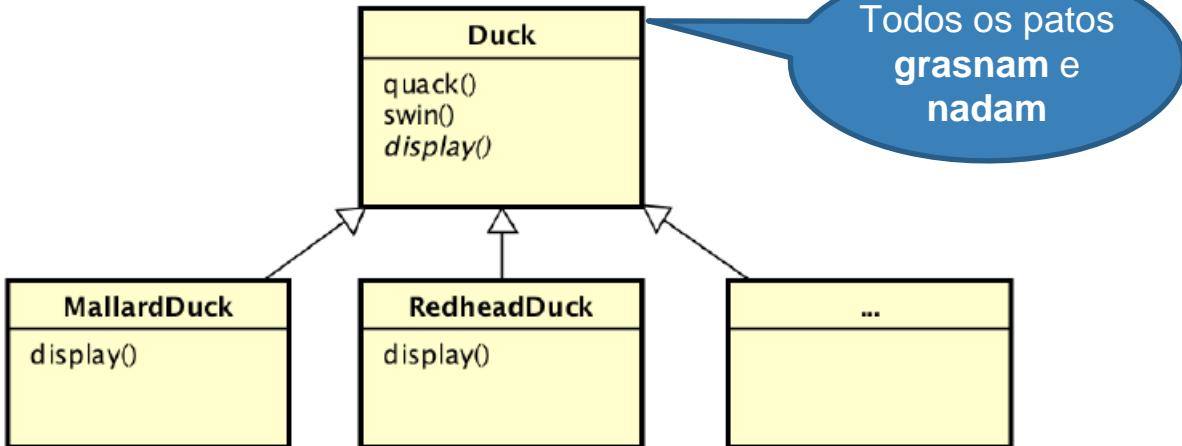


# Padrões de Projeto

## O Cenário



Manoel trabalha em uma empresa que criou um jogo de simulação de um lago com patos. O jogo pode mostrar uma grande variedade de espécies de patos **nadando** e **produzindo sons**. Os arquitetos do sistema usaram técnicas OO e chegaram ao seguinte modelo de classes.



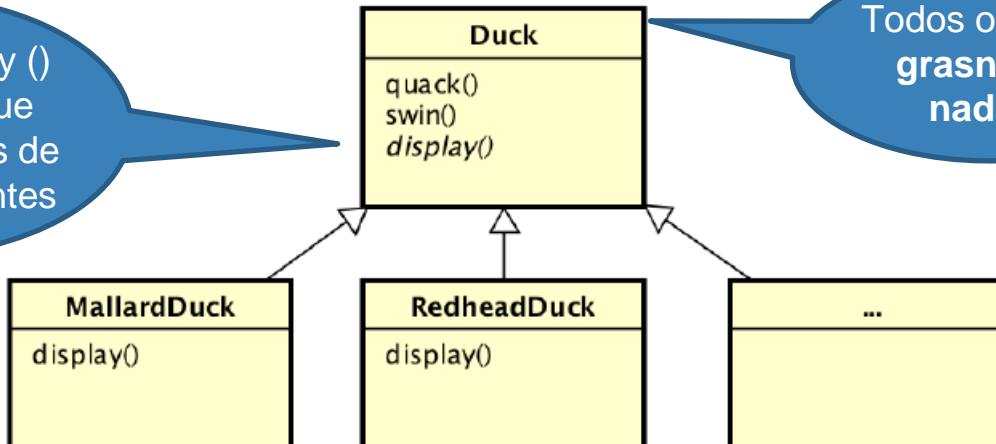
# Padrões de Projeto

## O Cenário



Manoel trabalha em uma empresa que criou um jogo de simulação de um lago com patos. O jogo pode mostrar uma grande variedade de espécies de patos **nadando** e **produzindo sons**. Os arquitetos do sistema usaram técnicas OO e chegaram ao seguinte modelo de classes.

O método `display()` é **abstrato** já que todos os subtipos de patos são diferentes



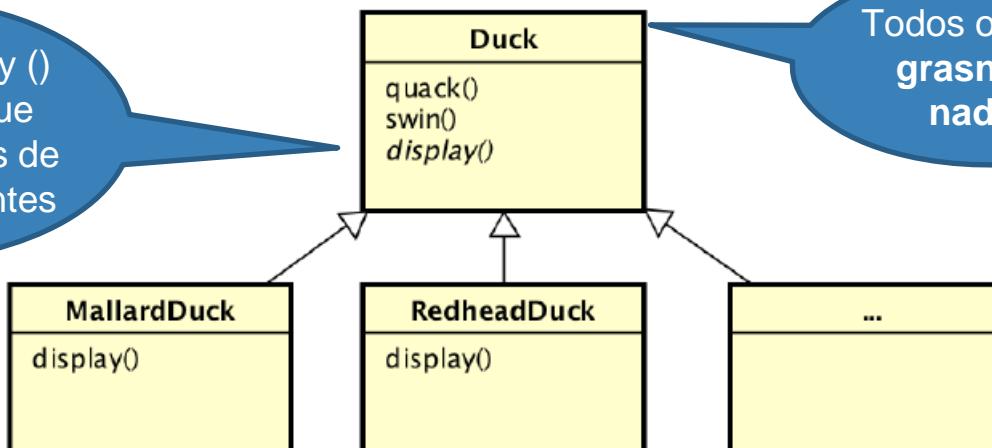
# Padrões de Projeto

## O Cenário



Manoel trabalha em uma empresa que criou um jogo de um imétodo abstrato obriga a classe em que ele se encontra ser **abstrata**. Dessa forma, podemos assumir que a classe **Duck** é abstrata. Segue o seguinte modelo de classes.

O método `display()` é **abstrato** já que todos os subtipos de patos são diferentes

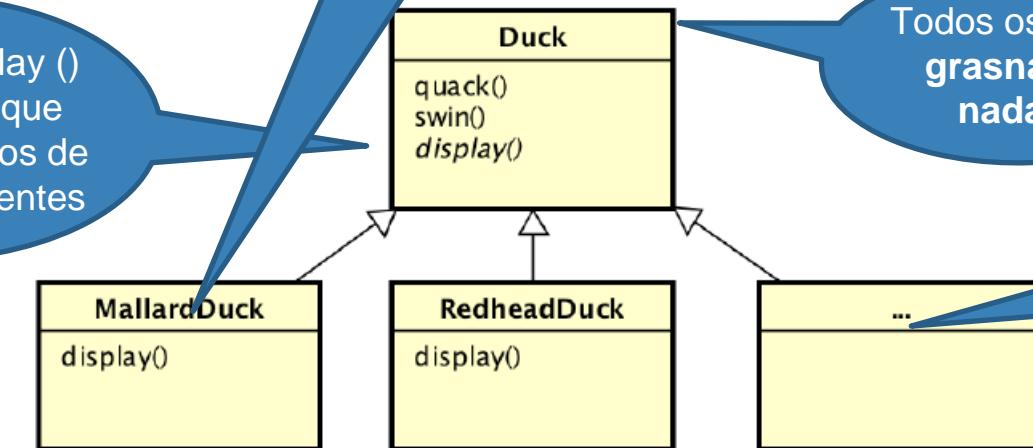


# Padrões de Projeto

## O Cenário



O método `display()` é **abstrato** já que todos os subtipos de patos são diferentes



Cada subtipo de pato implementa seu próprio comportamento de como ele será **exibido** na tela

Manoel trouxe a simulação com grande variedade de sons. Os arquitetos do sistema usaram técnicas OO e chegaram ao seguinte modelo

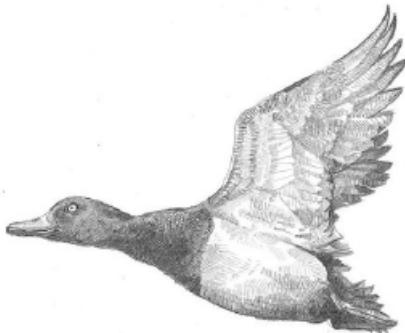
que criou um jogo de patos nadando e produzindo sons. O jogo pode mostrar uma variedade de patos **nadando e produzindo sons**.

Todos os patos **grasnam e nadam**

Muitos outros tipos de patos são subclasses de `Duck`

# Padrões de Projeto

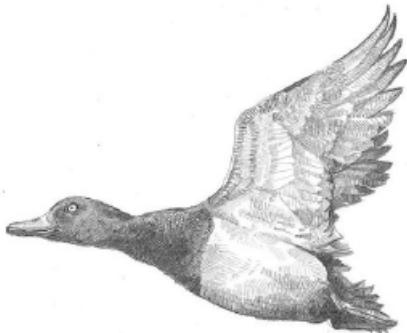
## Surge uma manutenção no sistema...



Os executivos da empresa decidiram que fazer os patos **voarem** é o que o simulador precisa para acabar com a concorrência.

# Padrões de Projeto

## Surge uma manutenção no sistema...

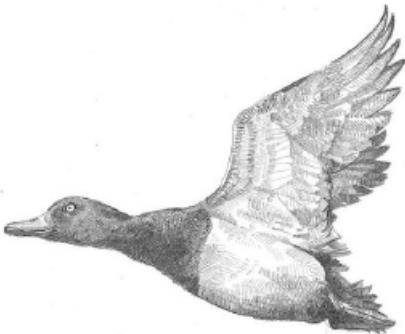


Os executivos da empresa decidiram que fazer os patos **voarem** é o que o simulador precisa para acabar com a concorrência.

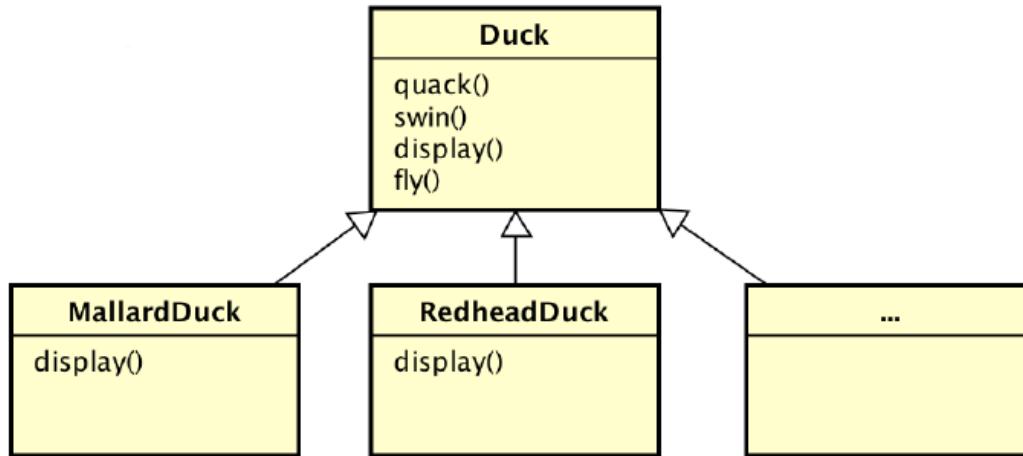
O que precisamos fazer para os patos **voarem**?

# Padrões de Projeto

## Surge uma manutenção no sistema...



Os executivos da empresa decidiram que fazer os patos **voarem** é o que o simulador precisa para acabar com a concorrência.

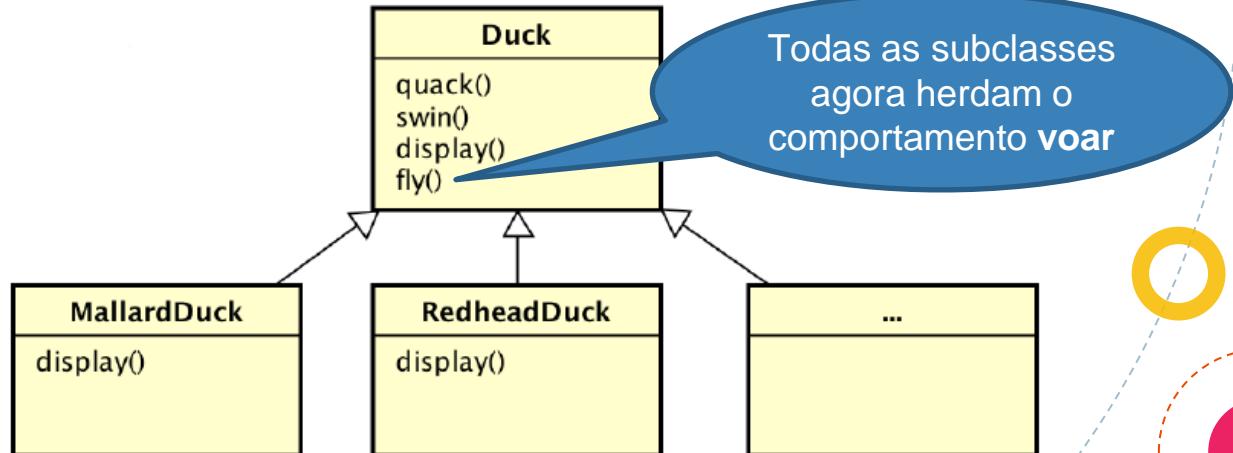


# Padrões de Projeto

## Surge uma manutenção no sistema...



Os executivos da empresa decidiram que fazer os patos **voarem** é o que o simulador precisa para acabar com a concorrência.

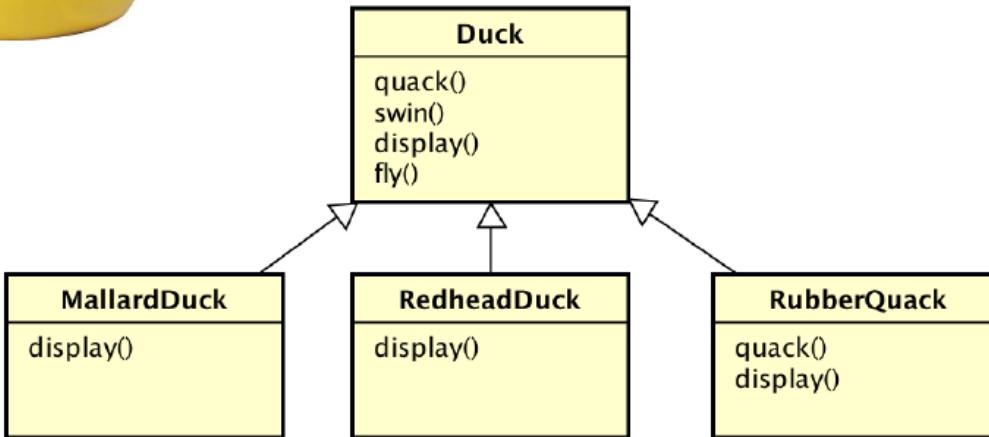


# Padrões de Projeto

## Surge uma manutenção no sistema...



Após a atualização do sistema, em plena demonstração do simulador para os acionistas, **patos de borracha** começaram a voar pela tela.

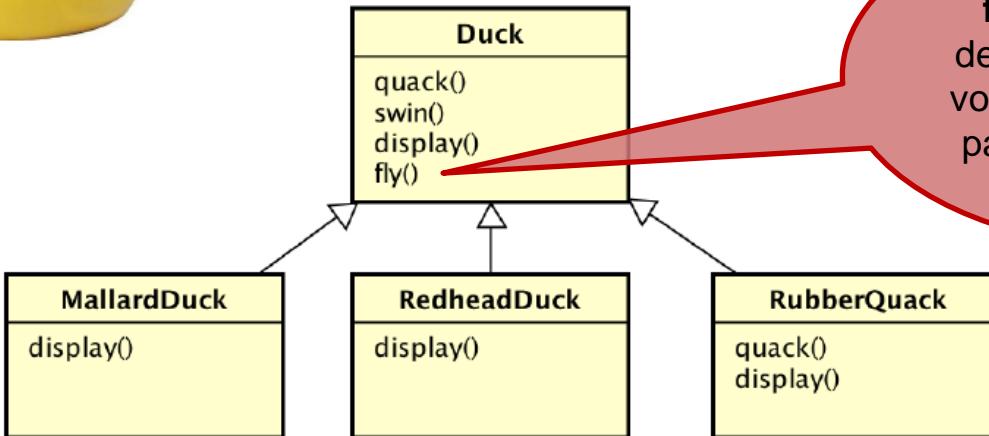


# Padrões de Projeto

## Surge uma manutenção no sistema...



Após a atualização do sistema, em plena demonstração do simulador para os acionistas, **patos de borracha** começaram a voar pela tela.



Ao incluirmos o método **fly()** na superclasse, demos a capacidade de voar a **todos** os tipos de patos, incluindo os que **não** deveriam voar

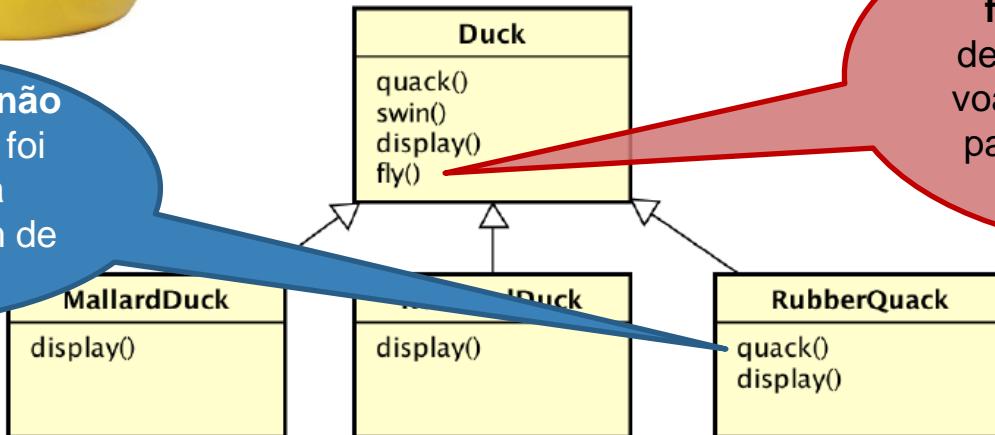
# Padrões de Projeto

## Surge uma manutenção no sistema...



Patos de borracha **não** grasan, quack () foi sobreescrito para implementar o som de “Squeak”

Após a atualização do sistema, em plena demonstração do simulador para os acionistas, **patos de borracha** começaram a voar pela tela.



Ao incluirmos o método **fly()** na superclasse, demos a capacidade de voar a **todos** os tipos de patos, incluindo os que **não** deveriam voar

# Padrões de Projeto

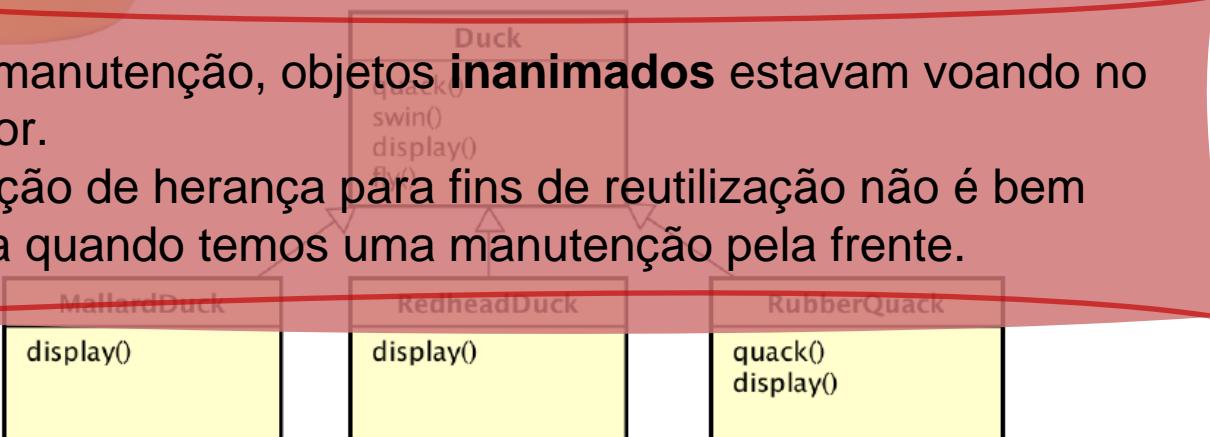
## Surge uma manutenção no sistema...



Após a atualização do sistema, em plena demonstração do simulador para os acionistas, **patos de borracha** começaram a voar pela tela.

Após a manutenção, objetos **inanimados** estavam voando no simulador.

A utilização de herança para fins de reutilização não é bem recebida quando temos uma manutenção pela frente.

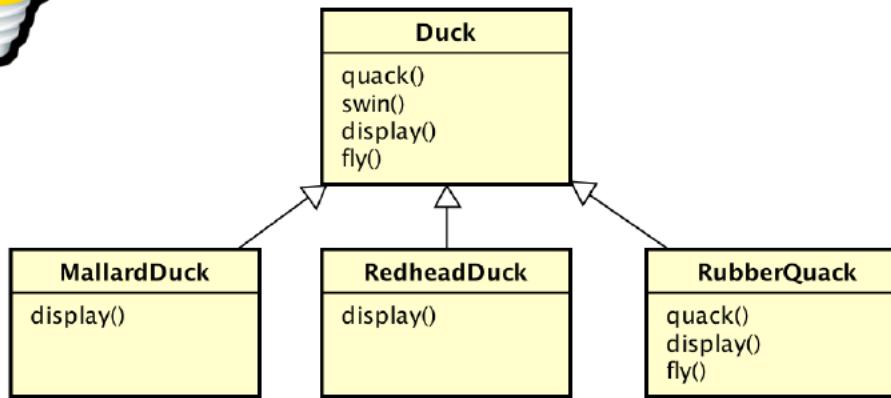


# Padrões de Projeto

## Pensando sobre herança...



Poderíamos *sobrescrever* o método **fly()** na classe *RubberQuack*, similar ao que fizemos com o método **quack()**.

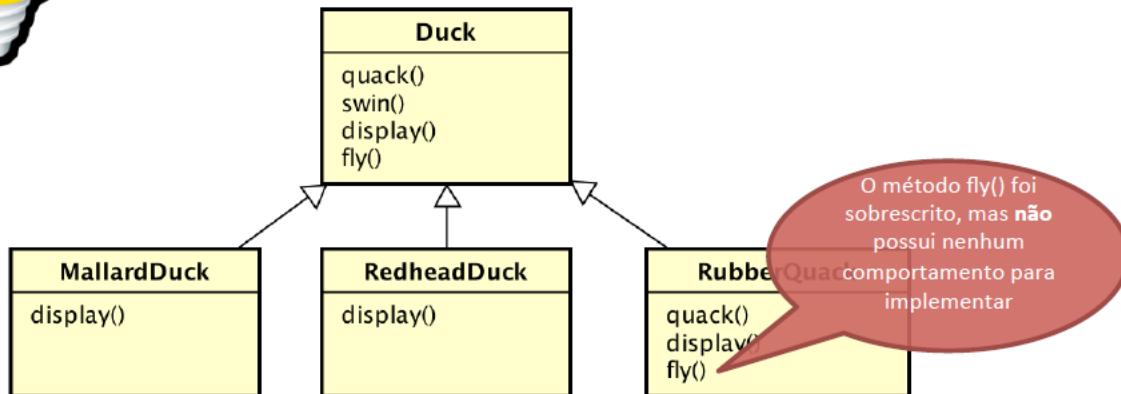


# Padrões de Projeto

## Pensando sobre herança...



Poderíamos *sobrescrever* o método `fly()` na classe `RubberQuack`, similar ao que fizemos com o método `quack()`.

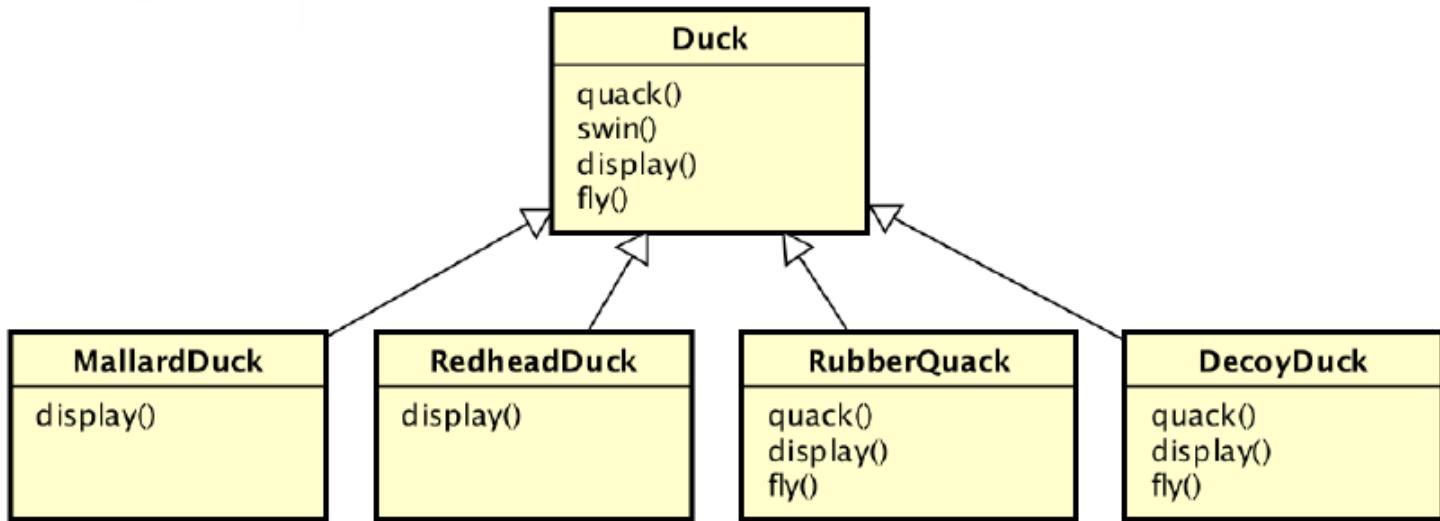


# Padrões de Projeto

## Pensando sobre herança...



E se surgissem novo tipos de patos a serem tratados pelo simulador? Como um **pato de madeira**. Ele **não** deve **voar** nem **grasnar**...

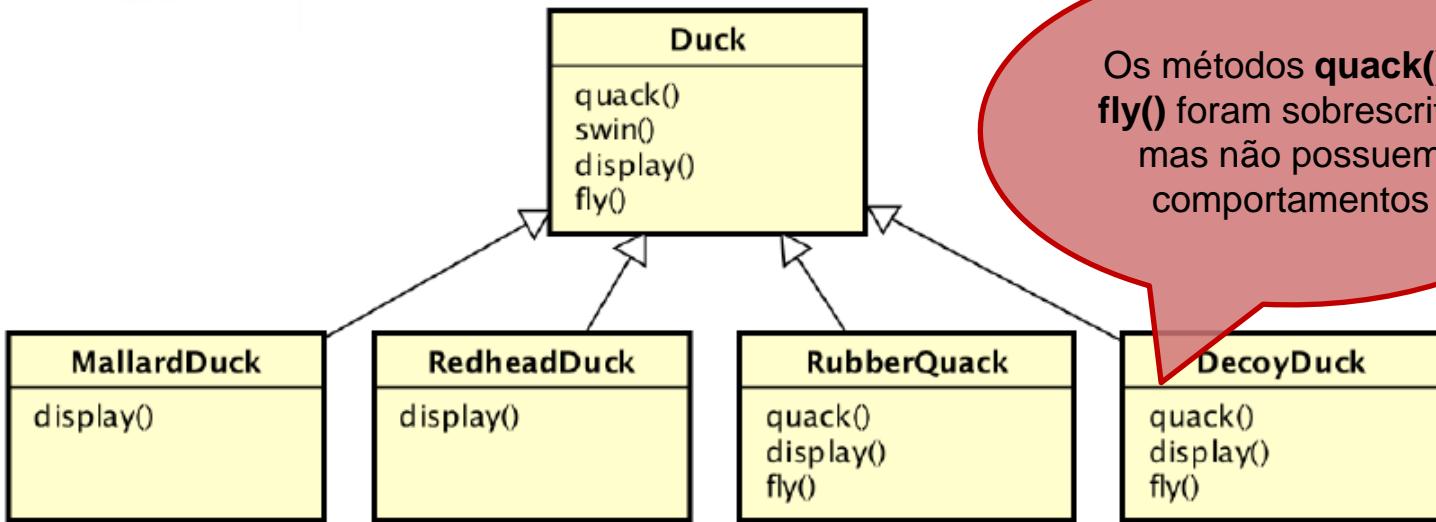


# Padrões de Projeto

## Pensando sobre herança...



E se surgissem novo tipos de patos a serem tratados pelo simulador? Como um **pato de madeira**. Ele **não** deve **voar** nem **grasnar**...



# Padrões de Projeto

## Pensando sobre herança...



Com o que vimos até agora, quais opções a seguir podem ser consideradas **desvantagens** ao se utilizar herança para produzir o comportamento de *Duck*?

# Padrões de Projeto

## Pensando sobre herança...



Com o que vimos até agora, quais opções a seguir podem ser consideradas **desvantagens** ao se utilizar herança para produzir o comportamento de *Duck*?

- a) O código é duplicado entre subclasses.
- b) Não podemos fazer patos dançar.
- c) É difícil conhecer o comportamento de todos os patos.
- d) Os patos não conseguem voar e gransnar ao mesmo tempo.
- e) As alterações podem afetar sem querer outros patos.

# Pontos a repensar...

- E se o produto precisasse ser atualizado a cada 3 meses ou menos, devido a uma decisão estratégica de negócio?
- A cada atualização as especificações do produto continuarão mudando e seremos forçados a *sobrescrever* os métodos `fly()` e `quack()` para cada nova *subclasse* de `Duck`.

# Pontos a repensar...

- E se o produto precisasse ser atualizado a cada 3 meses ou menos, devido a uma decisão estratégica de negócio?
- A cada atualização as especificações do produto continuarão mudando e seremos forçados a *sobrescrever* os métodos `fly()` e `quack()` para cada nova *subclasse* de `Duck`.

Quando utilizamos herança a manutenção do código se torna mais complexa e propícia a erros

# Pontos a repensar...

- E se o produto precisasse ser atualizado a cada 3 meses ou menos, devido a uma decisão estratégica de negócio?
- A cada atualização as especificações do produto continuarão mudando e seremos forçados a *sobrescrever* os métodos `fly()` e `quack()` para cada nova *subclasse* de `Duck`.

Precisamos de uma maneira mais simples para que apenas **alguns** (mas não todos) tipos de patos **voem** ou **grasnem**.

# Padrões de Projeto

## Que tal uma Interface?



- Poderíamos tirar o método `fly()` da superclasse **Duck** e criar uma interface **Flyable** com um método `fly()`.

# Padrões de Projeto

## Que tal uma Interface?



- Poderíamos tirar o método `fly()` da superclasse **Duck** e criar uma interface **Flyable** com um método `fly()`.
- Dessa forma, somente os patos que devem voar irão implementar esta interface e ter um método `fly()`.
-

# Padrões de Projeto

## Que tal uma Interface?



- Poderíamos tirar o método `fly()` da superclasse **Duck** e criar uma interface **Flyable** com um método `fly()`.
- Dessa forma, somente os patos que devem voar irão implementar esta interface e ter um método `fly()`.
- Além disso, poderíamos criar o método **Quackable** pois nem todos os patos podem **grasnar**.

# Padrões de Projeto

## Que tal uma Interface?

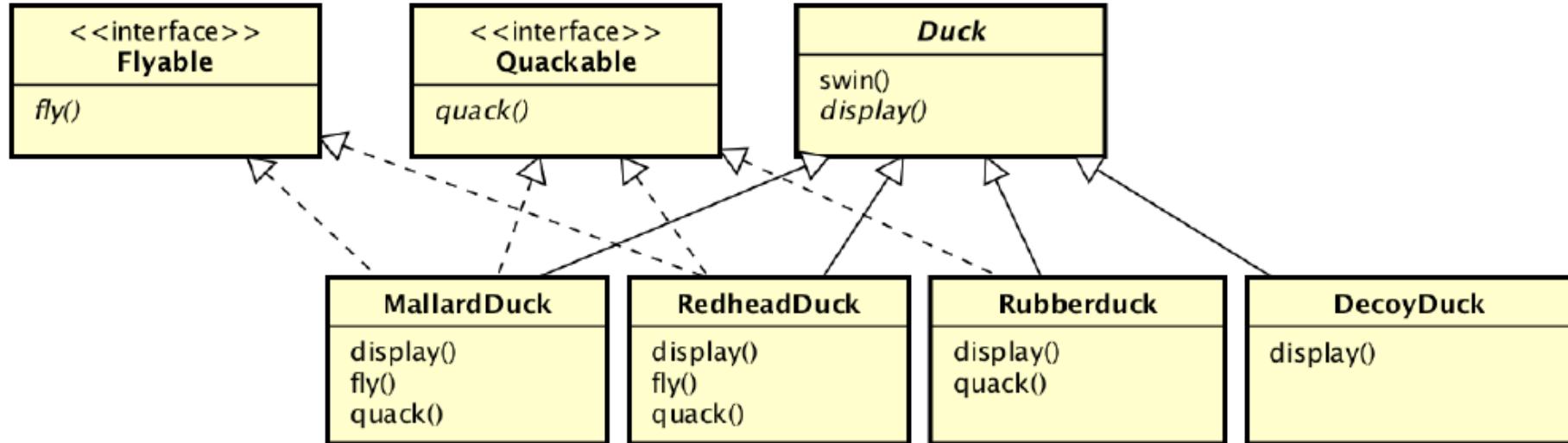


- Poderíamos tirar o método `fly()` da superclasse **Duck** e criar uma interface **Flyable** com um método `fly()`.
- Dessa forma, somente os patos que devem voar irão implementar esta interface e ter um método `fly()`.
- Além disso, poderíamos criar o método **Quackable** pois nem todos os patos podem **grasnar**.

Uma **Interface** é um contrato onde quem assina se responsabiliza por implementar seus métodos. Ela expõe o que o objeto deve fazer e **não** como ele faz.

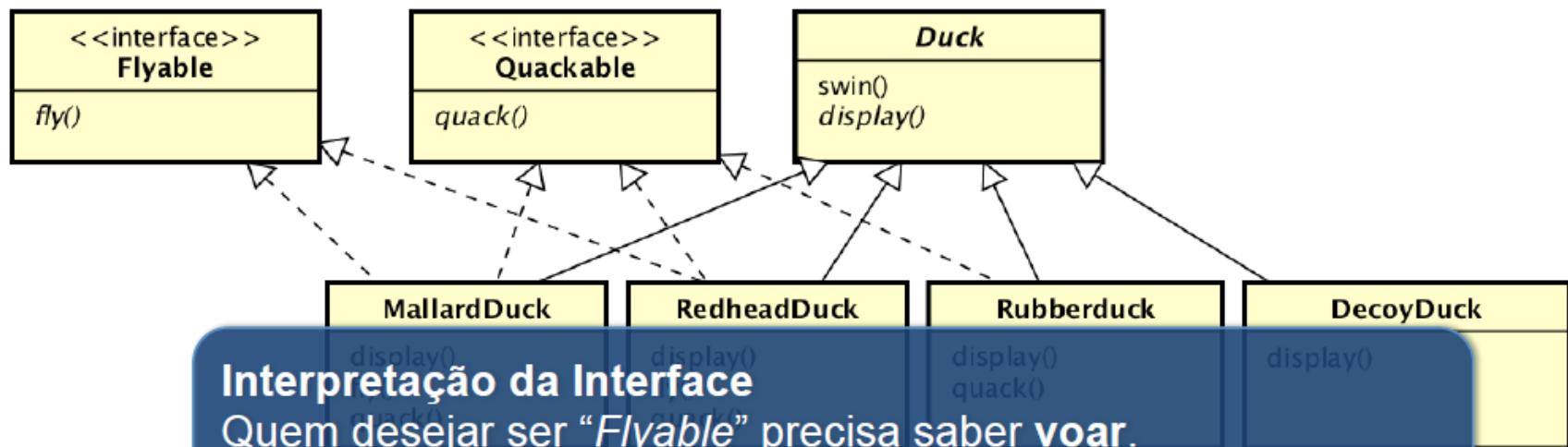
# Padrões de Projeto

## Que tal uma Interface?



# Padrões de Projeto

## Que tal uma Interface?



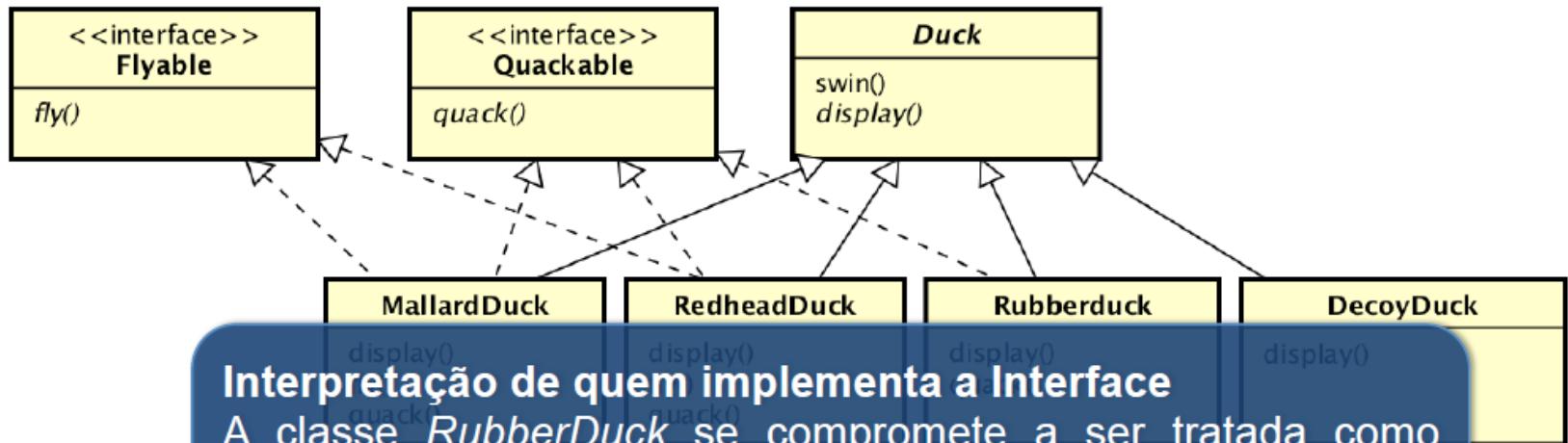
### Interpretação da Interface

Quem desejar ser “*Flyable*” precisa saber **voar**.

Quem desejar ser “*Quackable*” precisa saber **grasnar**.

# Padrões de Projeto

## Que tal uma Interface?

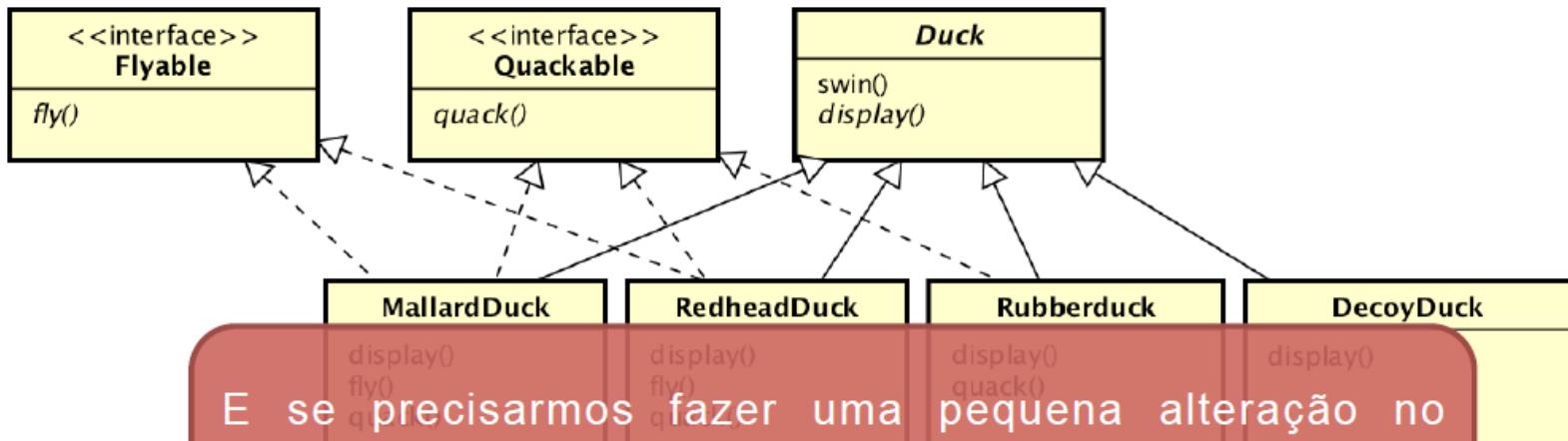


### Interpretação de quem implementa a Interface

A classe `RubberDuck` se compromete a ser tratada como `Quackable`, sendo obrigada a ter os métodos necessários, definidos neste contrato.

# Padrões de Projeto

## Que tal uma Interface?



E se precisarmos fazer uma pequena alteração no comportamento de vôo... de todas as inúmeras subclasses de *Duck que voam*?

# Padrões de Projeto

## Repensando a Interface...

- Vimos que nem todas as subclasses devem ter o comportamento de **voar** ou **grasnar**, então o uso de herança **não** foi uma solução adequada.
- Implementar *Flyable* e *Quackable* nas subclasses resolveu parte do problema (sem fazer patos de borracha voarem inadequadamente), porém **degradou** completamente a **reutilização** de código para esses comportamentos, criando apenas um pesadelo de **manutenção** diferente.

É notório que pode haver mais de um tipo de comportamento de vôo até entre os patos que voam.

# Padrões de Projeto

## Repensando a Interface...

- Vimos que nem todas as subclasses devem ter o comportamento de **voar** ou **grasnar**, então o uso de herança **não** foi uma solução adequada.
- Implementar *Flyable* e *Quackable* nas subclasses resolveu parte do problema (sem fazer patos de borracha voarem inadequadamente), porém **degradou** completamente a **reutilização** de código para esses comportamentos, criando apenas um pesadelo de **manutenção** diferente.

# Padrões de Projeto

## A Constante no Desenvolvimento de Software



- Qual a única coisa com a qual podemos contar sempre no desenvolvimento de software?

# Padrões de Projeto

## A Constante no Desenvolvimento de Software



- Qual a única coisa com a qual podemos contar sempre no desenvolvimento de software?
- Não importa onde você trabalha, o que está criando ou em que linguagem está programando, qual a constante que estará sempre com você?

ALTERAÇÃO

# Padrões de Projeto

## A Constante no Desenvolvimento de Software



- Qual a única coisa com a qual podemos contar sempre no desenvolvimento de software?
- Não importa onde você trabalha, o que está criando ou em que linguagem está programando, qual a constante que estará sempre com você?

**Independentemente** de como você desenvolva uma aplicação, com o tempo ela precisará **crescer** e **mudar** para não morrer.

# A Constante no Desenvolvimento de Software



- Qual a única coisa com a qual podemos contar sempre no desenvolvimento de software?
- Não importa onde você trabalha, o que está criando ou em que linguagem está programando, qual a constante que estará sempre com você?

Independentemente de como você desenvolva uma aplicação, com o tempo ela precisará **crescer** e **mudar** para não morrer.

*“Sistemas devem ser continuamente adaptados ou eles se tornam progressivamente menos satisfatórios.”*

[Lehman]

# Padrões de Projeto

## Repensando o Problema

- Herança não funcionou muito bem:
  - O comportamento do pato continua mudando entre as subclasses;
  - Não é apropriado que todas as subclasses tenham estes comportamentos.
- Foi promissor usarmos os conceitos de **Interface**, mas:
  - Não tivemos reaproveitamento de código;
  - Sempre que um comportamento for modificado todas as subclasses que o implementam devem ser alteradas.

# PRINCÍPIOS DE PROJETO

# #1 Princípios de Projeto

**Identifique os aspectos de sua aplicação que variam e separe-os do que permanece igual.**

- Em outras palavras: Pegue as partes que variam e encapsule-as para depois poder alterar ou estender estas partes sem afetar as que não variam.

# #1 Princípios de Projeto

**Identifique os aspectos de sua aplicação que variam e separe-os do que permanece igual.**

- Em outras palavras: Pegue as partes que variam e encapsule-as para depois poder alterar ou estender estas partes sem afetar as que não variam.

Isso traz menos consequências indesejadas nas alterações realizadas no código e maior flexibilidade.

# Padrões de Projeto

**Separando as partes que variam das que ficam iguais**

- Vamos criar dois conjuntos de classes, um para voar e e outro para grasar.
- Cada conjunto de classes irá conter todas as implementações de seu comportamento.

# Padrões de Projeto

Separando as partes que variam das que ficam iguais

- Vamos criar dois conjuntos de classes, um para voar e e outro para grasar.
- Cada conjunto de classes irá conter todas as implementações de seu comportamento.

Sabemos que **fly()** e **quack()** são as partes da classe *Duck* que variam entre os patos.

# Padrões de Projeto

Separando as partes que variam das que ficam iguais

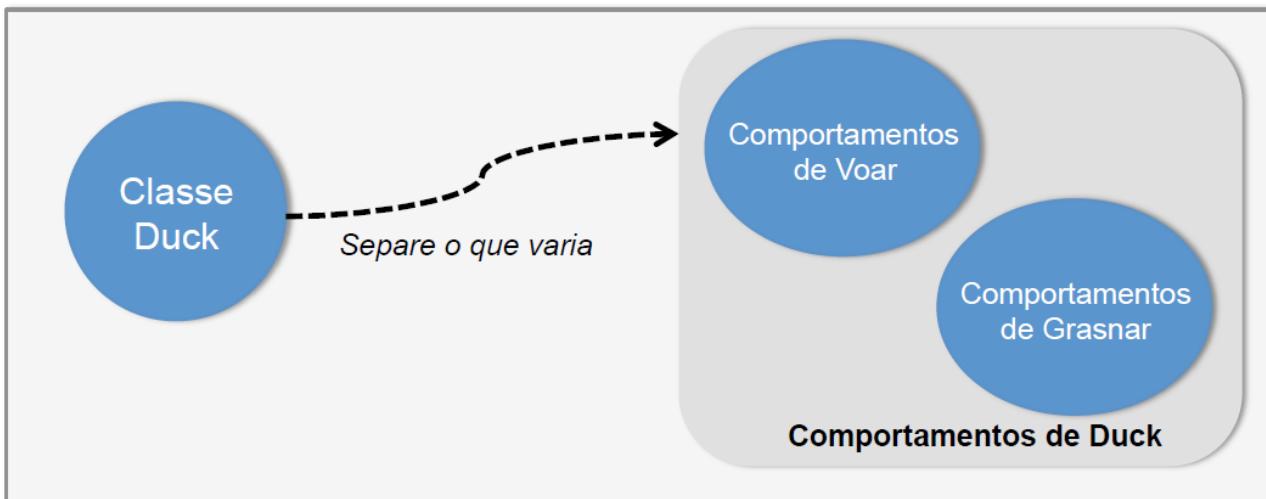
- Vamos criar dois conjuntos de classes, um para voar e e outro para grasar.
- Cada conjunto de classes irá conter todas as implementações de seu comportamento.



# Padrões de Projeto

Separando as partes que variam das que ficam iguais

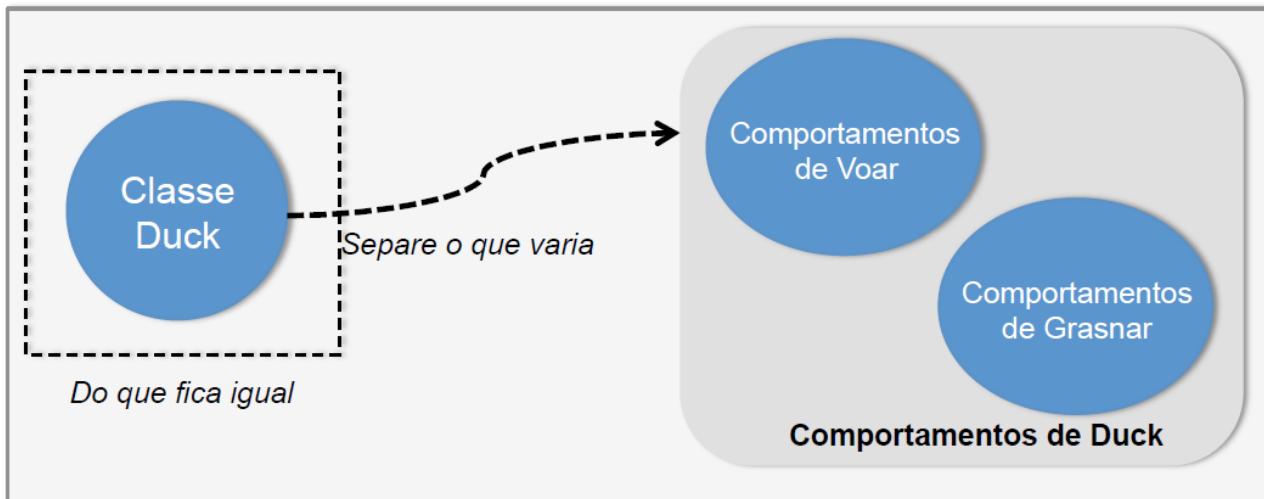
- Vamos criar dois conjuntos de classes, um para voar e e outro para grasnar.
- Cada conjunto de classes irá conter todas as implementações de seu comportamento.



# Padrões de Projeto

Separando as partes que variam das que ficam iguais

- Vamos criar dois conjuntos de classes, um para voar e e outro para grasnar.
- Cada conjunto de classes irá conter todas as implementações de seu comportamento.



# Padrões de Projeto

## Pensando na Solução...



Como vamos desenvolver o conjunto de classes que implementam os comportamentos de voar e grasnar?

# Padrões de Projeto

## Pensando na Solução...



**Como vamos desenvolver o conjunto de classes que implementam os comportamentos de voar e grasnar?**

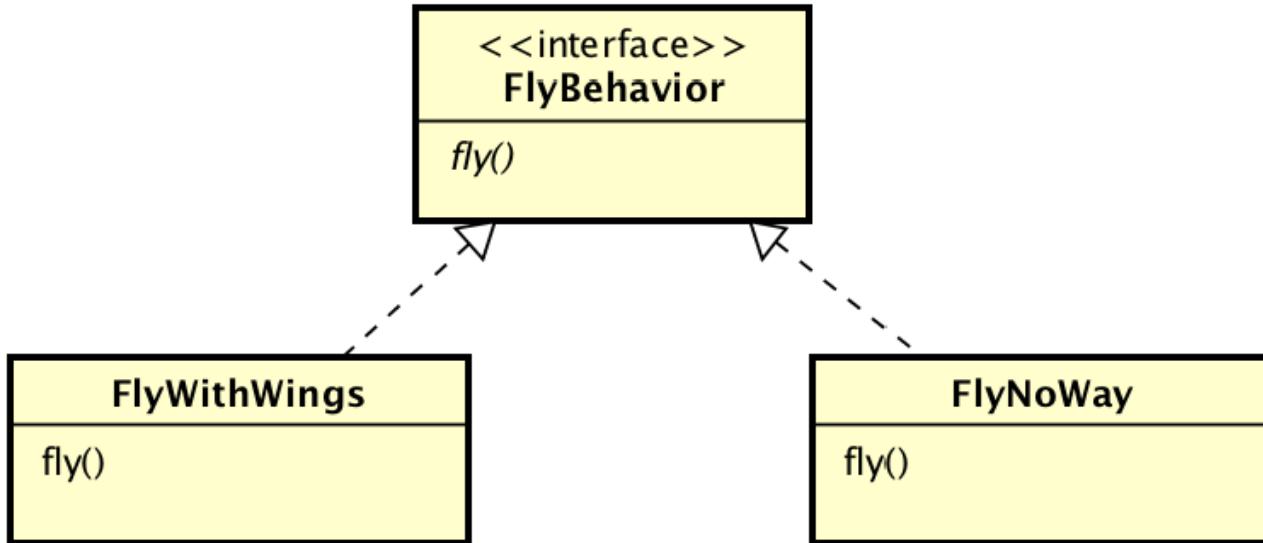
- Gostaríamos de manter a arquitetura flexível.
- Queremos atribuir comportamentos às instâncias da classe *Duck*;
- Poderíamos alterar o comportamento de uma instância de *Duck* em tempo de execução.

## #2 Princípios de Projeto

Programa para uma Interface e não para uma implementação.

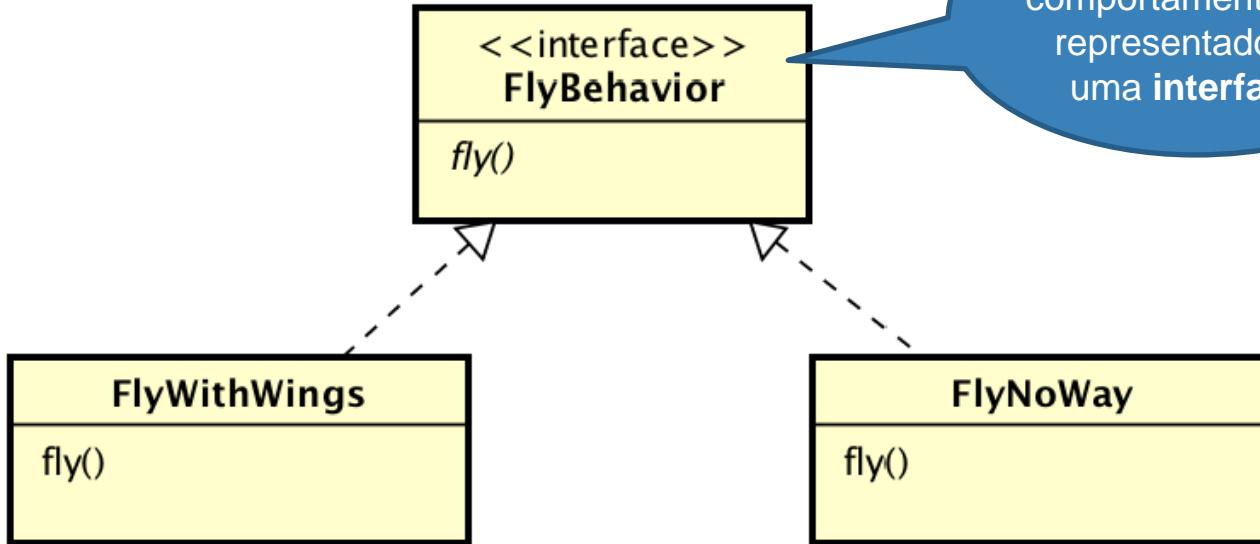
# #2 Princípios de Projeto

Programa para uma Interface e não para uma implementação.



# #2 Princípios de Projeto

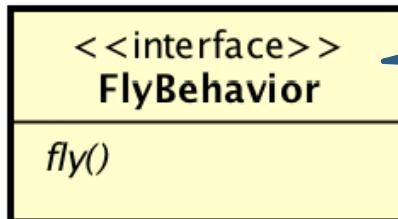
Programa para uma Interface e não para uma implementação.



# #2 Princípios de Projeto

Programa para uma Interface e não para uma implementação.

Implementa o vôo do pato

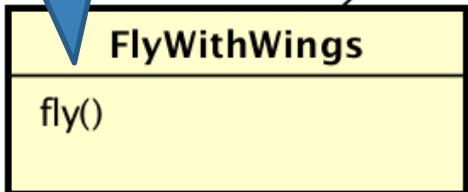


Cada comportamento será representado por uma interface

# #2 Princípios de Projeto

Programa para uma Interface e não para uma implementação.

Implementa o vôo do pato



Cada comportamento será representado por uma interface

Não possui implementação, pois é um comportamento para quem não sabe voar

# #2 Princípios de Projeto

Programa para uma Interface e não para uma implementação.

Implementa o vôo do pato



Cada comportamento será representado por uma interface

Dessa forma, as classes derivadas de *Duck* não precisam conhecer nenhum detalhe de implementação dos seus comportamentos

Não possui implementação, pois é um comportamento para quem não sabe voar

## #2 Princípios de Projeto

Programa para uma Interface e não para uma implementação.

Cada comportamento será

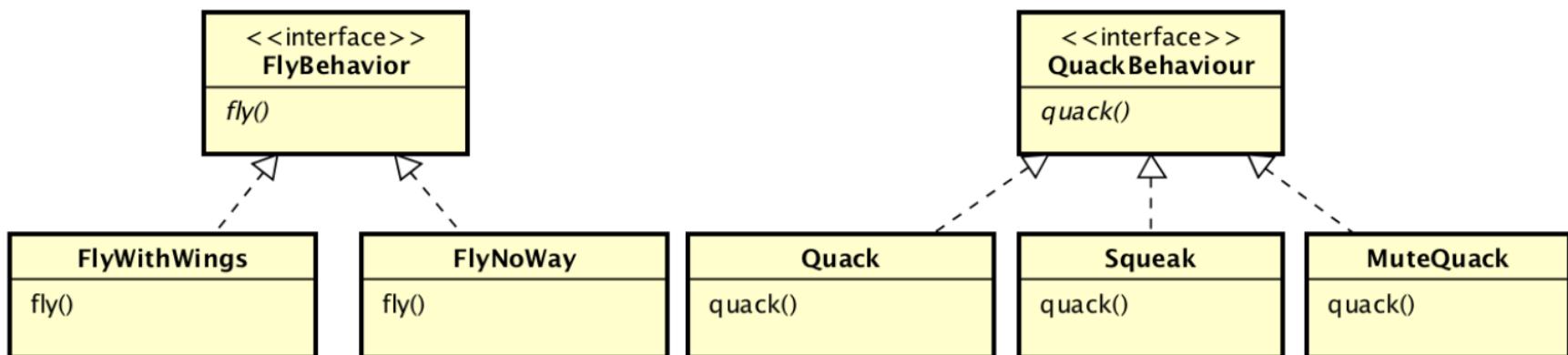
Com o novo *design*, as classes derivadas de *Duck* irão usar um comportamento representado por uma **Interface**.

A implementação real do comportamento não fica **acoplada** às classes derivadas de *Duck*.



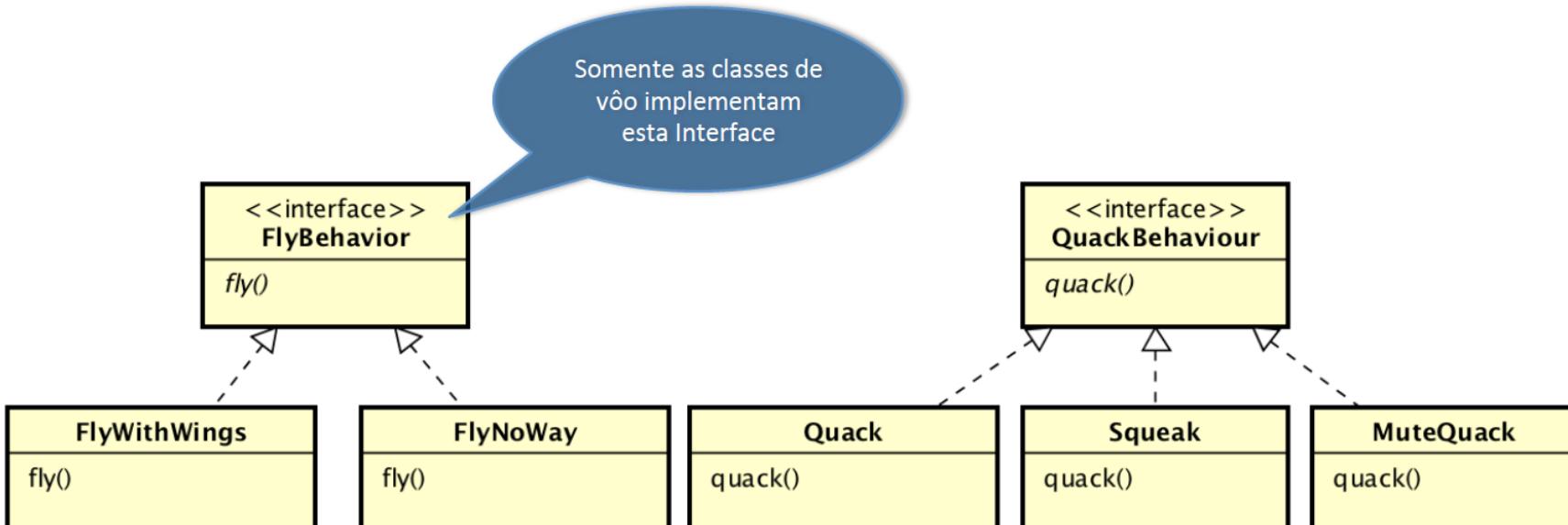
# Padrões de Projeto

## Implementando os Comportamentos



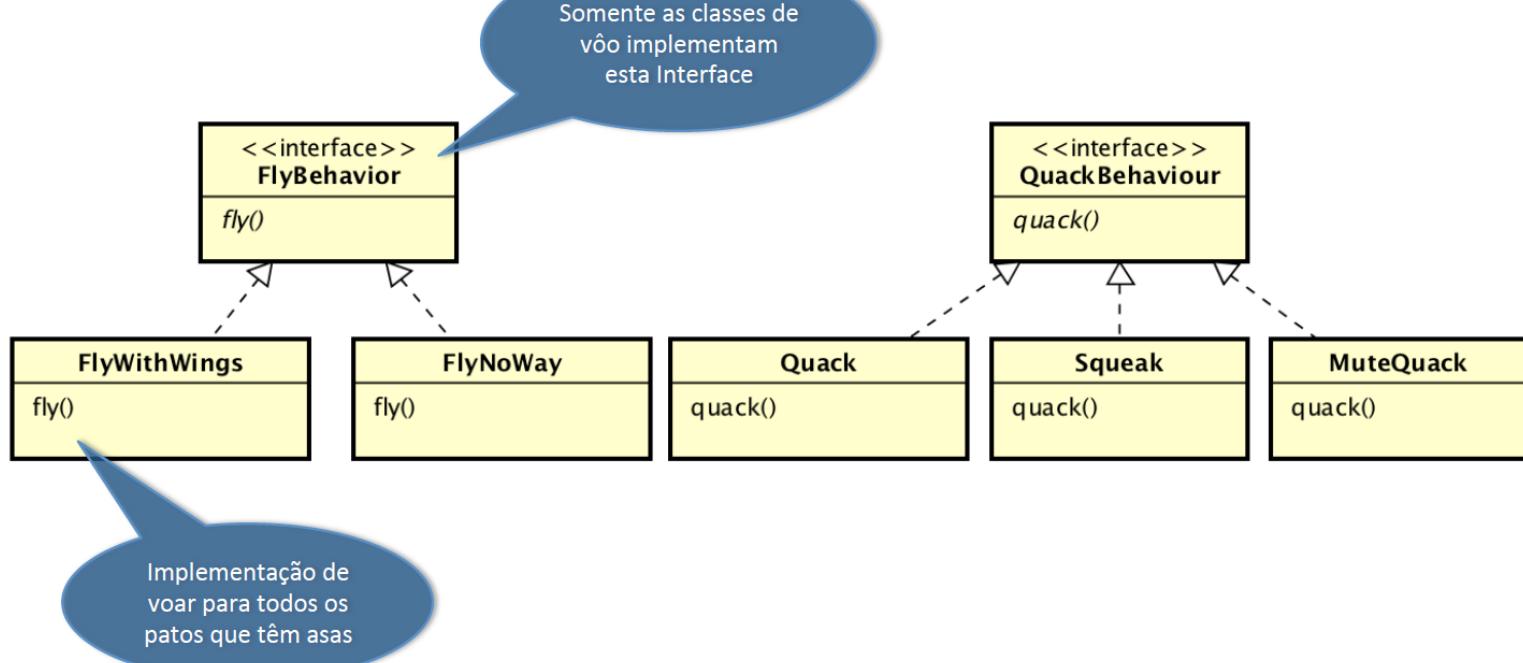
# Padrões de Projeto

## Implementando os Comportamentos

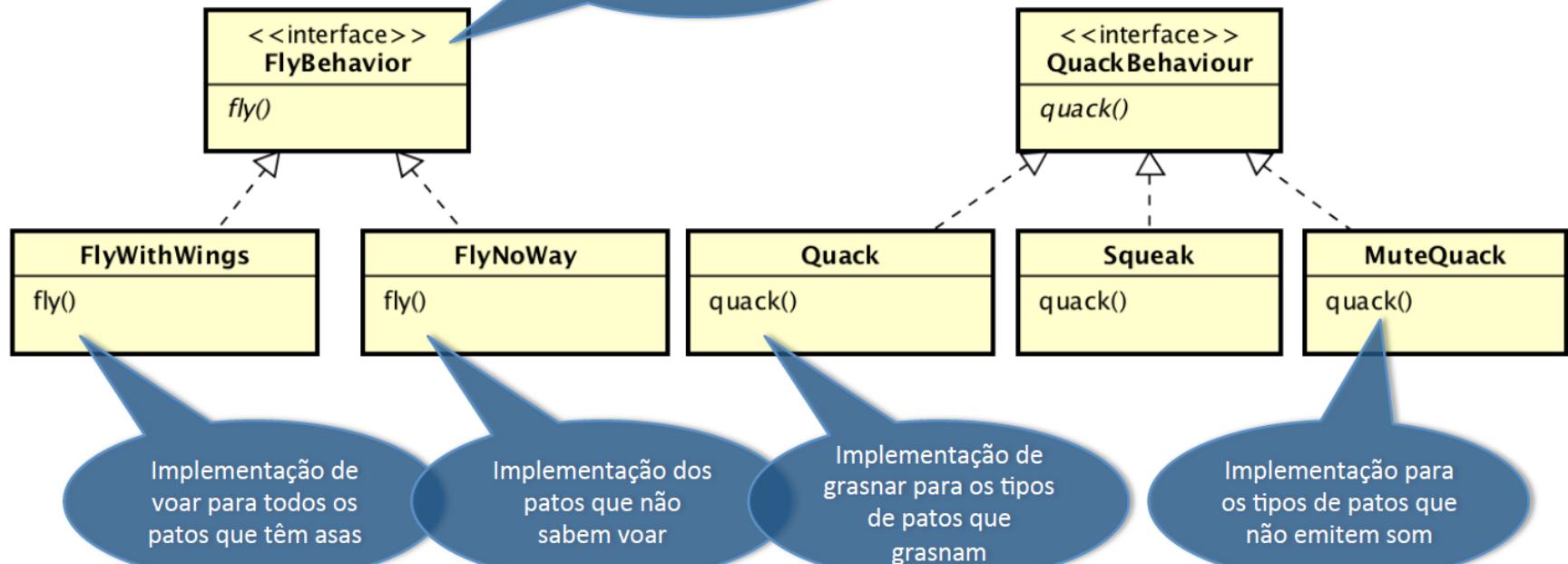


# Padrões de Projeto

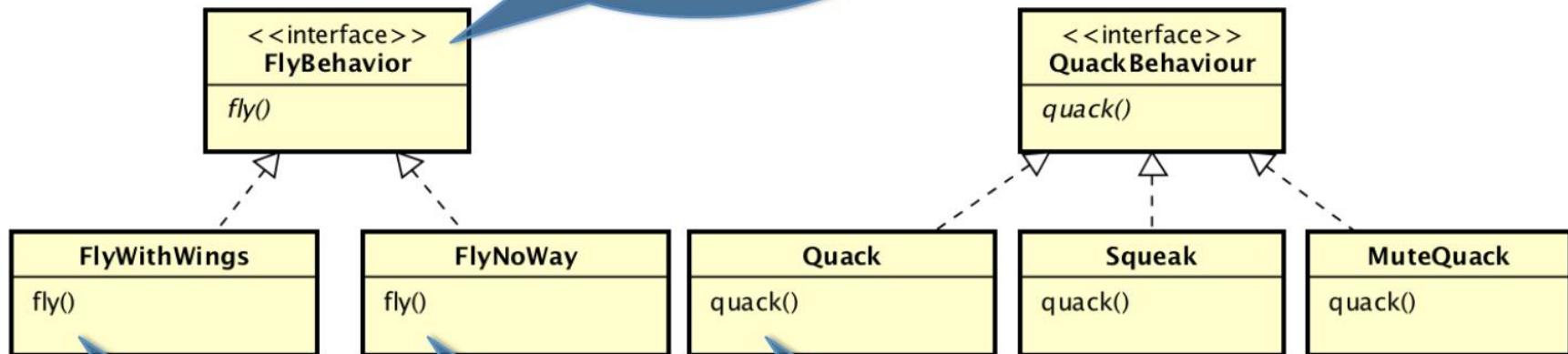
## Implementando os Comportamentos



# Implementando os Comportamentos



# Implementando os Comportamentos



Somente as classes de vôo implementam esta Interface

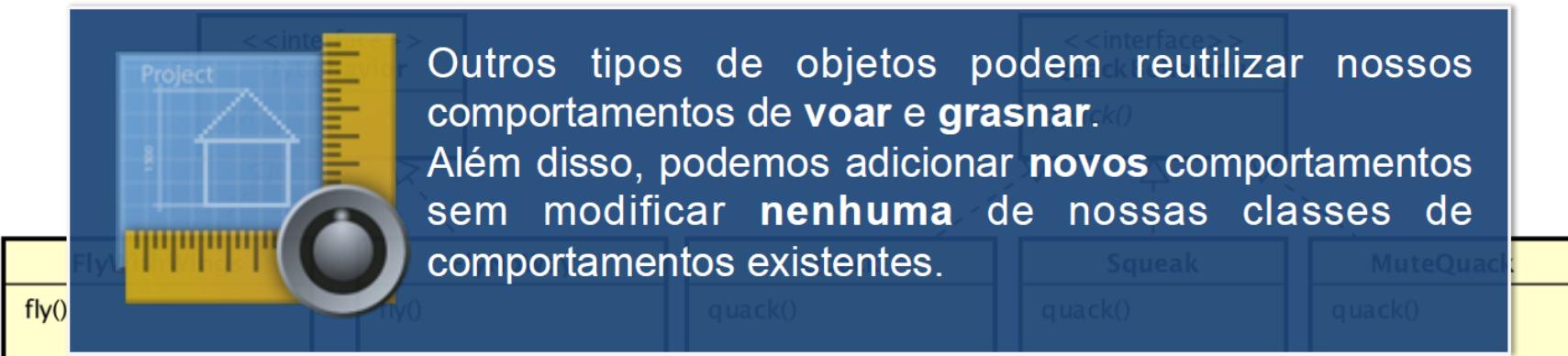
Implementação de voar para todos os patos que têm asas

Implementação dos patos que não sabem voar

Implementação de grunhar para os tipos de patos que grunham

# Padrões de Projeto

## Implementando os Comportamentos



# Padrões de Projeto

## Pensando em reuso...



Usando o nosso novo projeto, o que você faria se precisasse adicionar um vôo de foguete à aplicação?

Você consegue pensar numa classe que poderia usar o comportamento de *Quack* que não seja um pato?

# Padrões de Projeto

## Pensando em reuso...



Usando o nosso novo projeto, o que você faria se precisasse adicionar um vôo de foguete à aplicação?

**Poderíamos criar uma classe FlyRocket que implementasse a interface FlyBehaviour.**

Você consegue pensar numa classe que poderia usar o comportamento de Quack que não seja um pato?

**Uma classe que representasse um dispositivo eletrônico que simula sons de pato.**

# Padrões de Projeto

## Integrando o Comportamento

Delegaremos o comportamento de voar e grasar em vez de usar os métodos definidos nas subclasses de *Duck*.

- Vamos adicionar duas variáveis de instância à classe *Duck* chamadas *flyBehaviour* e *quackBehaviour*.
- Iremos remover os métodos *fly()* e *quack()* da classe *Duck* e de suas subclasses pelo fato de estarmos delegando estes comportamentos para as classes *FlyBehaviour* e *QuackBehaviour*.
- Incluiremos dois novos métodos à classe *Duck*, chamados *performFly* e *performQuack*.

# Padrões de Projeto

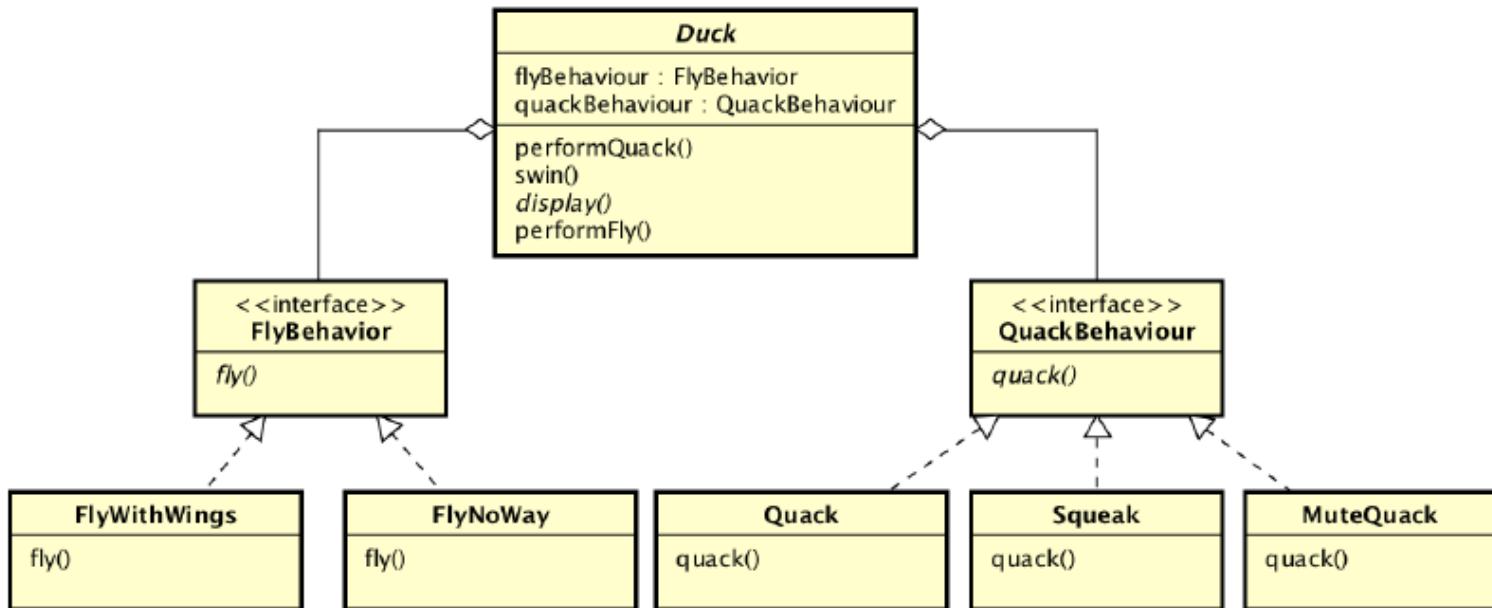
## Modelo x Implementação

Duck
flyBehaviour : FlyBehavior quackBehaviour : QuackBehaviour
performQuack() swim() display() performFly()

```
public abstract class Duck {  
  
    FlyBehavior flyBehavior;  
    QuackBehavior quackBehavior;  
  
    public Duck() {  
    }  
  
    public void performFly() {  
        flyBehavior.fly();  
    }  
  
    public void performQuack() {  
        quackBehavior.quack();  
    }  
    // outros métodos aqui...
```

# Padrões de Projeto

## Modelo x Implementação



# Padrões de Projeto

## Mais integração...

- Como agora estamos delegando o comportamento de *voar* e *grasnar*, temos que definir as variáveis de instância *flyBehaviour* e *quackBehaviour* nas subclasses de *Duck*.

```
public class MallardDuck extends Duck {  
  
    public MallardDuck() {  
  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
  
    public void display() {  
        System.out.println("I'm a real Mallard duck");  
    }  
}
```

# Padrões de Projeto

## Mais integração...

- Como agora estamos delegando o comportamento de *voar* e *grasnar*, temos que definir as variáveis de instância *flyBehaviour* e *quackBehaviour* nas subclasses de *Duck*.

```
public class MallardDuck extends Duck {  
    public MallardDuck() {  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
  
    public void display() {  
        System.out.println("I'm a real Mallard duck");  
    }  
}
```

A classe MallardDuck herda a variáveis de instância de Quack

# Padrões de Projeto

## Mais integração...

- Como agora estamos delegando o comportamento de *voar* e *grasnar*, temos que definir as variáveis de instância *flyBehaviour* e *quackBehaviour* nas subclasses de *Duck*.

```
public class MallardDuck extends Duck {  
    public MallardDuck() {  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
  
    public void display() {  
        System.out.println("I'm a real Mallard duck");  
    }  
}
```

A responsabilidade de  
grasnar é delegada ao  
objeto Quack

# Padrões de Projeto

- Como agora estamos delegando o comportamento de *voar* e *grasnar*, temos que definir as variáveis de instância *flyBehaviour* e *quackBehaviour* nas subclasses de *Duck*.

```
public class MallardDuck extends Duck {  
    public MallardDuck() {  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
    public void display() {  
        System.out.println("I'm a real Mallard duck");  
    }  
}
```

Como **MallardDuck** sabe como voar este comportamento é utilizado

# Padrões de Projeto

## #3 Princípios de Projeto

Dar prioridade à composição

- Cada pato tem um *FlyBehaviour* e um *QuackBehaviour* aos quais delega as capacidades de voar e grasar.

**Criar sistemas usando composições nos permite:**

- Uma maior flexibilidade;
- Encapsular uma família de algoritmos em seu próprio conjunto de classes;
- Alterar o comportamento de objetos em tempo de execução.

# Padrões de Projeto

## Padrão Strategy

- Acabamos de aplicar o nosso primeiro padrão de projetos, o padrão **Strategy**.
- O padrão **Strategy** define uma família de objetos, encapsula cada um deles e o torna intercambiáveis. O **Strategy** deixa o algoritmo variar independentemente dos clientes que o utilizam.



“

Os padrões de projeto são descrições de objetos que se comunicam e classes que são customizadas para resolver um problema de projeto genérico em um contexto específico.

*Gamma, Helm, Vlissides & Johnson, sobre padrões de projeto em software*

# Padrões de Projeto

O que é?

- ◎ Uma nova categoria de conhecimento
- ◎ Conhecimento não é novo, mas falar sobre ele é
- ◎ O objetivo é conhecer o que você já conhece

Como?

- ◎ Partindo de problemas e soluções recorrentes em diferentes áreas do conhecimento

# Padrões de Projeto

Escritores (livros, HQs, roteiros) raramente inventam novas histórias  
Ideias frequentemente reusadas!



Projetistas também reutilizam soluções  
De preferência as boas!  
Problemas são tratados de modo a não reinventar soluções  
Experiência é o que torna uma pessoa um expert  
    Prever o que pode dar errado  
    Prever potenciais mudanças  
    Prever dependências indesejadas  
Porém, muitas vezes transmitidas de forma tácita, sem apoio de documentação

# Padrões de Projeto

## Por que aprender padrões?

- Aprender com a experiência dos outros
  - Identificar problemas comuns em engenharia de software
  - Utilizar soluções testadas e bem documentadas
- Aprender boas práticas de programação
  - Padrões utilizam eficientemente herança, composição, modularidade, polimorfismo e abstração para construir código reutilizável, eficiente, de alta coesão e baixo acoplamento
- Ter um caminho e um alvo para refatorações
- Facilitar a comunicação, compreensão e documentação
  - Vocabulário comum
  - Ajuda a entender o papel das classes do sistema

# Padrões de Projeto

## Por que aprender padrões?

- Vocabulário comum
  - Faz o sistema ficar menos complexo ao permitir que se fale em um nível mais alto de abstração
- Ajuda na documentação e na aprendizagem
  - Conhecendo os padrões de projeto torna mais fácil a compreensão de sistemas existentes
  - "As pessoas que estão aprendendo POO freqüentemente reclamam que os sistemas com os quais trabalham usam herança de forma complexa e que é difícil de seguir o fluxo de controle. Geralmente a causa disto é que eles não entendem os padrões do sistema" [GoF]
  - Aprender os padrões ajudam um novato a agir mais como um especialista

# Padrões de Projeto

## Padrões de projeto

Iniciaram a “febre” de padrões, mas existem outros:

Padrões de gerenciamento

Ex. Métodos ágeis como SCRUM e XP

Padrões organizacionais

Padrões de análise

Padrões arquiteturais

Padrões de implementação

# Padrões de Projeto

## Elementos do padrão de projeto

### Nome

Apelido para descrever o padrão

### Problema

Descreve quando o padrão pode ser aplicado

### Contexto

Características da situação em que o padrão se aplica

### Solução

Determina o que fazer para resolver o problema

### Consequências

Descreve resultados positivos (vantagens) e negativos (desvantagens) de aplicação do padrão

### Usos conhecidos

Ocorrência do padrão e de suas aplicações em sistemas conhecidos

# Metáforas no mundo real

## *State*

Pessoas reagem de formas diferentes de acordo com o humor.



## *Observer*

Jornais são entregues aos seus assinantes .



## *Chain of Responsibility*

Chefe delega responsabilidade a um subordinado, que delega a outro, que ...



"Johnson gave it to Wilson to give to Adams to give to O'Connor to give to Anderson to give to me to give to you to get it done right away."

Padrões são formados por vários objetos.  
Cada objeto tem um papel (**responsabilidade**) na solução.

# Padrões de Projeto

## Padrões clássicos ou Padrões GoF (*Gang of Four*)

- Descreve 23 padrões de projeto “clássicos”
- Soluções genéricas para os problemas mais comuns do desenvolvimento de software orientado a objetos
- Obtidas através de experiências de sucesso na indústria de software
- Sobre o livro
  - Seus quatro autores são conhecidos como “The Gang of Four” (GoF).
  - O livro tornou-se um clássico na literatura orientada a objetos e continua atual.

### Design Patterns

Elements of Reusable  
Object-Oriented Software

Erich Gamma  
Richard Helm  
Ralph Johnson  
John Vlissides



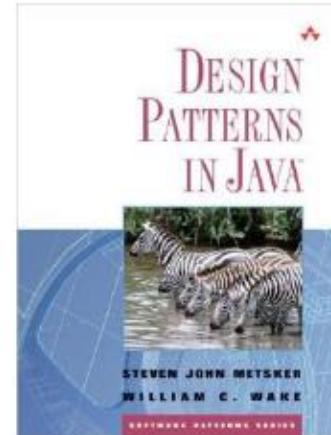
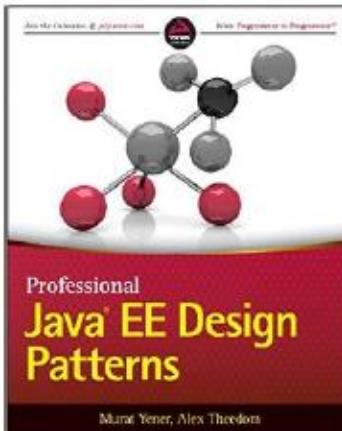
Foreword by Grady Booch

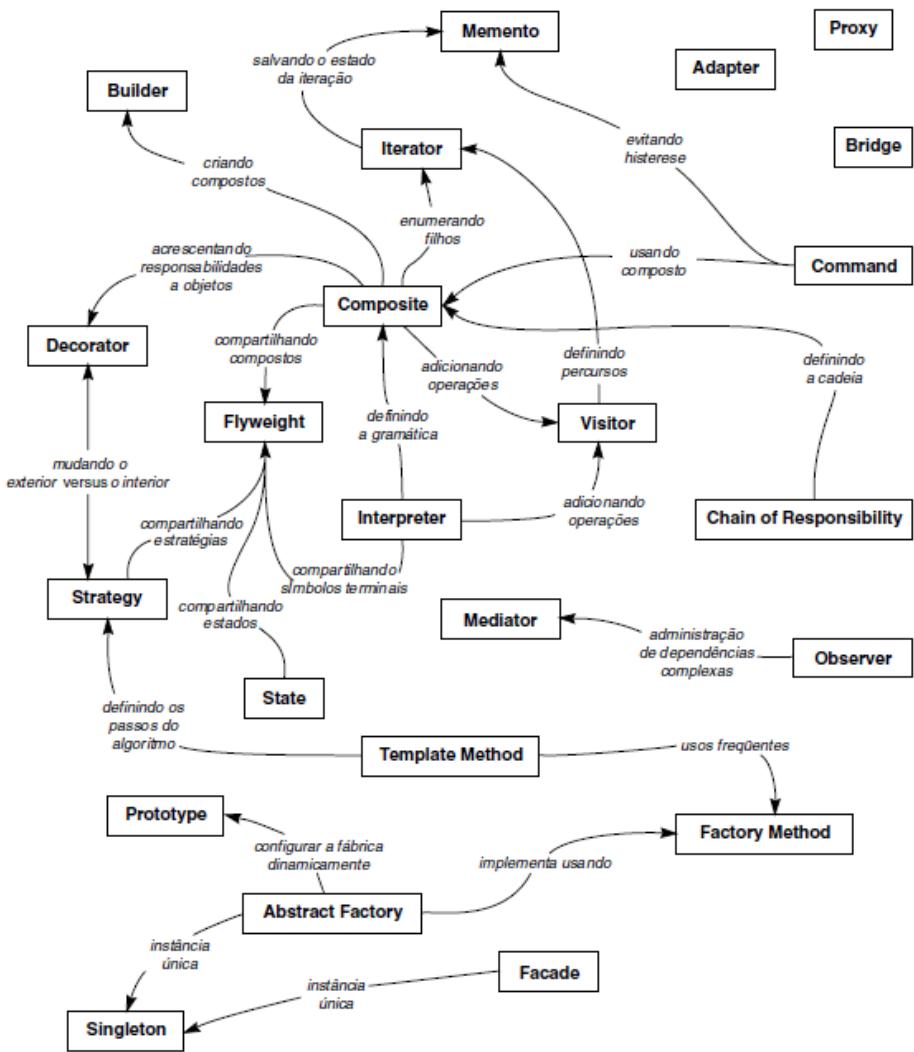
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

# Padrões de Projeto

## Mais Padrões?

- Há vários catálogos de padrões para software
  - Muitos são específicos a uma determinada área
  - Padrões Java EE
  - Padrões de implementação Java ou C#





# Padrões de Projeto

## Formas de Classificação dos Padrões

Há várias formas de classificar os padrões

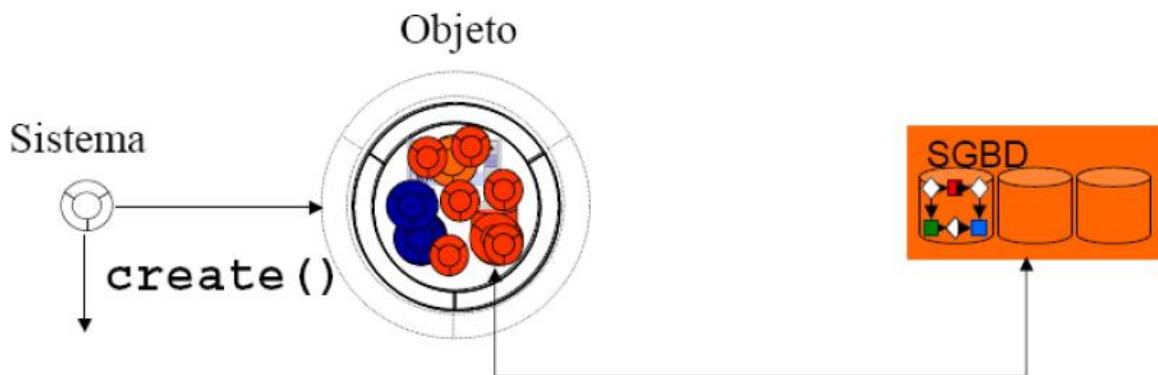
- Os padrões *GoF* são tradicionalmente classificados pelo propósito:
  - Criação de classes e objetos
  - Alteração da estrutura de um programa
  - Controle do seu comportamento

# Padrões de Projeto

## Padrões de Criação

Abstraem o processo de instanciação

Tornam um sistema independente da forma como os objetos são criados, compostos e representados

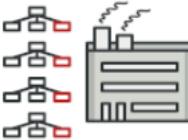


# Padrões de Projeto



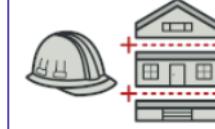
## Factory Method

Fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados.



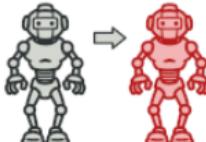
## Abstract Factory

Permite que você produza famílias de objetos relacionados sem ter que especificar suas classes concretas.



## Builder

Permite construir objetos complexos passo a passo. O padrão permite produzir diferentes tipos e representações de um objeto usando o mesmo código de construção.



## Prototype

Permite que você copie objetos existentes sem fazer seu código ficar dependente de suas classes.



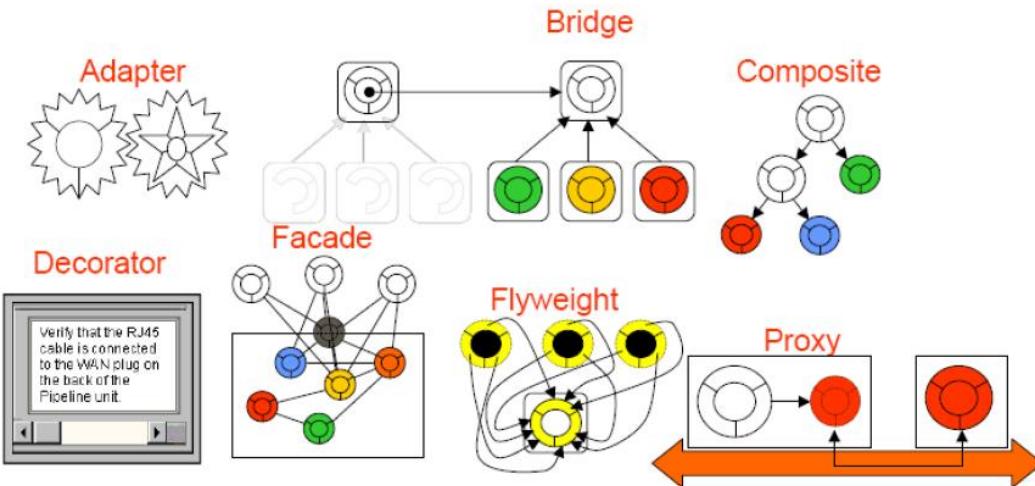
## Singleton

Permite a você garantir que uma classe tem apenas uma instância, enquanto provê um ponto de acesso global para esta instância.

# Padrões de Projeto

## Padrões Estruturais

Lidam com a composição de classes (ou objetos) para formar grandes estruturas no sistema



# Padrões de Projeto



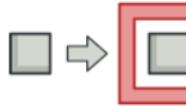
## Bridge

Permite que você divida uma classe grande ou um conjunto de classes intimamente ligadas em duas hierarquias separadas—abstração e implementação—que podem ser desenvolvidas independentemente umas das outras.



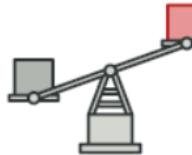
## Facade

Fornece uma interface simplificada para uma biblioteca, um framework, ou qualquer outro conjunto complexo de classes.



## Proxy

Permite que você forneça um substituto ou um espaço reservado para outro objeto. Um proxy controla o acesso ao objeto original, permitindo que você faça algo ou antes ou depois do pedido chegar ao objeto original.



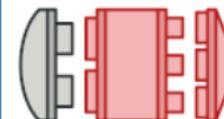
## Flyweight

Permite que você coloque mais objetos na quantidade disponível de RAM ao compartilhar partes do estado entre múltiplos objetos ao invés de manter todos os dados em cada objeto.



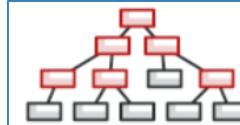
## Decorator

Permite que você adicione novos comportamentos a objetos colocando eles dentro de um envoltório (wrapper) de objetos que contém os comportamentos.



## Adapter

Permite a colaboração de objetos de interfaces incompatíveis.



## Composite

Permite que você componha objetos em estrutura de árvores e então trabalhe com essas estruturas como se fossem objetos individuais.

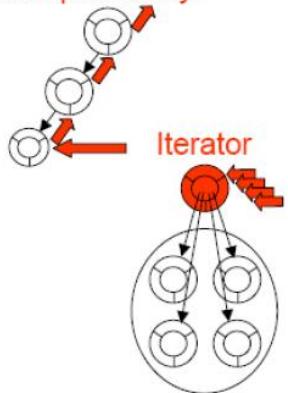
# Padrões de Projeto

## Padrões Comportamentais

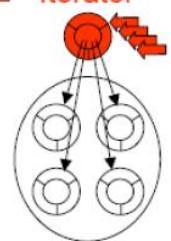
Caracterizam a forma como classes (ou objetos) interagem

Distribuem responsabilidade

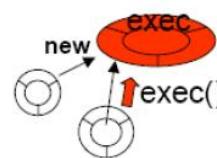
ChainOf  
Responsibility



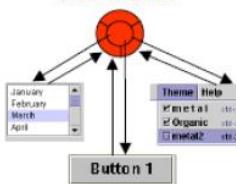
Iterator



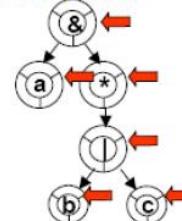
Command



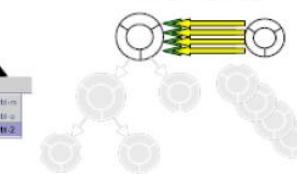
Mediator



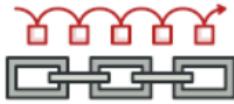
Interpreter



Memento



# Padrões de Projeto



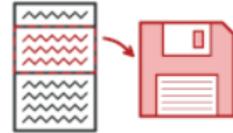
## Chain of Responsibility

Permite que você passe pedidos por uma corrente de handlers. Ao receber um pedido, cada handler decide se processa o pedido ou passa para o próximo handler da corrente.



## Iterator

Permite que você percorra elementos de uma coleção sem expor as representações estruturais deles (lista, pilha, árvore, etc.)



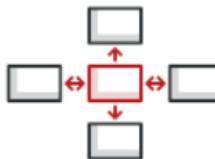
## Memento

Permite que você salve e restaure o estado anterior de um objeto sem revelar os detalhes de sua implementação.



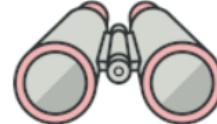
## Command

Transforma o pedido em um objeto independente que contém toda a informação sobre o pedido. Essa transformação permite que você parametrize métodos com diferentes pedidos, atraso ou coloque a execução do pedido em uma fila, e suporte operações que não podem ser feitas.



## Mediator

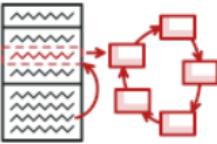
Permite que você reduza as dependências caóticas entre objetos. O padrão restringe comunicações diretas entre objetos e os força a colaborar apenas através do objeto mediador.



## Observer

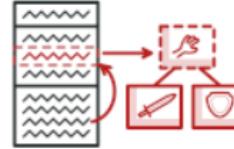
Permite que você defina um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.

# Padrões de Projeto



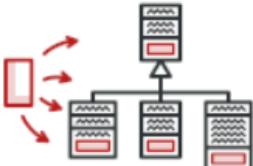
**State**

Permite que um objeto altere seu comportamento quando seu estado interno muda. Parece como se o objeto mudasse de classe.



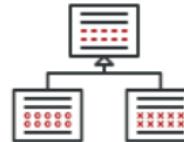
**Strategy**

Permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis.



**Visitor**

Permite que você separe algoritmos dos objetos nos quais eles operam.



**Template Method**

Define o esqueleto de um algoritmo na superclasse mas deixa as subclasses sobrepor etapas específicas do algoritmo sem modificar sua estrutura.

# Padrões de Projeto

- 1. Abstract Factory
- 2. Builder
- 3. Factory Method
- 4. Prototype
- 5. Singleton
- 6. Adapter
- 7. Bridge
- 8. Composite
- 9. Decorator
- 10. Facade
- 11. Flyweight
- 12. Proxy
- 13. Chain of Responsibility
- 14. Command
- 15. Interpreter
- 16. Iterator
- 17. Mediator
- 18. Memento
- 19. Observer
- 20. State
- 21. Strategy
- 22. Template Method
- 23. Visitor

- Padrões de Criação
- Padrões Estruturais
- Padrões de Comportamento

# Padrões de Projeto

## Padrões GoF - Classificação por Propósito

		Propósito		
		Criação	Estrutura	Comportamento
Classe	Escopo	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Builder Abstract Factory Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

*Classificação segundo GoF*

# Padrões de Projeto

## Formas de Classificação dos Padrões

Há várias formas de classificar os padrões

- Os padrões *GoF* são tradicionalmente classificados pelo **propósito**:
  - Criação de classes e objetos
  - Alteração da **estrutura** de um programa
  - Controle do seu **comportamento**
- *Metsker* os classifica em 5 grupos, por **intenção**:
  - Oferecer uma **interface**
  - Atribuir uma **responsabilidade**
  - Realizar a **construção** de classes ou objetos
  - Controlar formas de **operação**
  - Implementar uma **extensão** para a aplicação

# Padrões de Projeto

Intenção	Padrões
<b>1. Interfaces</b>	<i>Adapter, Facade, Composite, Bridge</i>
<b>2. Responsabilidade</b>	<i>Singleton, Observer, Mediator, Proxy, Chain of Responsibility, Flyweight</i>
<b>3. Construção</b>	<i>Builder, Factory Method, Abstract Factory, Prototype, Memento</i>
<b>4. Operações</b>	<i>Template Method, State, Strategy, Command, Interpreter</i>
<b>5. Extensões</b>	<i>Decorator, Iterator, Visitor</i>

*Classificação segundo Steven Metsker*

# Padrões de Projeto

Por onde começar?

# Padrões de Projeto

## ◎ Ordem Sugerida

- <http://mahemoff.com/paper/software/learningGoFPatterns/>
- Fácil
- Intermediário
- Avançado

# Padrões de Projeto

## Fácil

1. Facade (185)
2. Singleton (127)
3. Mediator (273)
4. Iterator (257)
5. Strategy (315)
6. Command (233)
7. Builder (97)
8. State (305)
9. Template Method (325)
10. Factory Method (107)
11. Memento (283)
12. Prototype (117)

# Padrões de Projeto

## Intermediários

1. Proxy (207)
2. Decorator (175)
3. Adapter (139)
4. Bridge (151)
5. Observer (293)

# Padrões de Projeto

## Avançados

1. Composite (163)
2. Interpreter (243)
3. Chain Of Responsibility (223)
4. Abstract Factory (87)
5. Visitor (331)

# Vamos Aprender sobre os Padrões de Projetos GOFs?



# Exercício

- Trabalho em equipe
- Estilo: Seminário (apresentação de slides com os resultados para toda turma)
- Tempo: até 25 minutos por equipe
- Entrega: 26/10/2023 (08h30)
- Apresentação: 26/10; 31/10 e 07/11 - ordem de sorteio
- **Observação: todas as equipes e membros da equipe DEVEM apresentar**
- Enviar pelo SIGAA

# Exercício

- Apresentar **DOIS PADRÕES CLÁSSICOS** (catálogo do 'GangofFour') descrevendo:
  - ❖ Objetivo/propósito do Padrão
  - ❖ Estrutura do Padrão
  - ❖ O problema que solucionam
  - ❖ A solução
  - ❖ Diagramas UML do padrão
  - ❖ Exemplos em Java do padrão (Pseudocódigo)
  - ❖ Aplicações típicas (Quando usar o padrão?)
  - ❖ Relacionamento entre os padrões (qual a relação desse padrão com os outros?)
  - ❖ Prós e contras
  - ❖ Como implementar?

# Exercício

- Para fazer essa atividade podem consultar os livros:
  - ◎ Use a Cabeça ! Padrões de Projetos (design Patterns) - 2<sup>a</sup> Ed.  
Elisabeth Freeman e Eric Freeman. Editora: Alta Books
  - ◎ Padroes de Projeto – Soluções reutilizáveis de software orientado a objetos. Erich Gamma, Richard Helm, Ralph Johnson. Editora Bookman
  - ◎ Fazer pesquisar na internet
  - ◎ <https://brizeno.wordpress.com/padroes/>
  - ◎ <https://refactoring.guru/pt-br/design-patterns>

## ORDEM DE APRESENTAÇÃO

01/03/2021

<u>Equipe</u>	<u>Tempo (*)</u>
<u>Equipe (et al.)</u>	<b>10h05 – 10h30</b>
<u>Equipe ( et al.)</u>	<b>10h35 – 11h00</b>
<u>Equipe (et al.)</u>	<b>11h05 – 11h30</b>
<u>Equipe (et al.)</u>	<b>11h35 – 12h00</b>

19/10/2023	Introdução aos Padrões de Projeto <i>Definição do Trabalho sobre Padrões de Projeto</i>	2h
24/10/2023	Atividade prática – padrão de projeto	2h
26/10/2023	Entrega e Seminário - apresentação sobre Padrões de Projeto ( <b>SEM1</b> )	2h
31/10/2023	Seminário - apresentação sobre Padrões de Projeto ( <b>SEM1</b> )	2h
02/11/2023	Feriado Finados	0h
07/11/2023	Seminário - apresentação sobre Padrões de Projeto ( <b>SEM1</b> )	2h

09/11/2023	Encontros Universitários	2h
14/11/2023	Padrões GRASP	2h
16/11/2023	Feedback parte 2 do trabalho ( <b>TP2</b> ) e entrega de nota do Seminário Padrões de Projeto ( <b>SEM1</b> )	2h
21/11/2023	Atividade prática – trabalho Final ( <b>TP3</b> )	2h
23/11/2023	Entrega e Seminário - Apresentação do projeto final ( <b>TP3</b> )	2h
28/11/2023	Apresentação do projeto final ( <b>TP3</b> )	2h
30/11/2023	Apresentação do projeto final ( <b>TP3</b> )	2h

05/12/2023	Prova Parcial 02 ( <b>PP2</b> )	2h
07/12/2023	2a chamada prova parcial 2 ( <b>PP2</b> )	2h
08/12/2023	Entrega de provas e notas	2h
14/12/2023	Avaliação Final	0h
18/12/2023	Consolidação das sínteses de notas e frequências 2023.2 das disciplinas semestrais pelos professores - Prazo Final	0h

# Referências

- ◎ Use a Cabeça ! Padrões de Projetos (design Patterns) - 2<sup>a</sup> Ed. Elisabeth Freeman e Eric Freeman. Editora: Alta Books
- ◎ Padroes de Projeto – Soluções reutilizáveis de software orientado a objetos. Erich Gamma, Richard Helm, Ralph Johnson. Editora Bookman
- ◎ Alexander, C. (1977). A pattern language: towns, buildings, construction. Oxford University Press.
- ◎ Neil, T. (2014). Mobile design pattern gallery: UI patterns for smartphone apps. " O'Reilly Media, Inc.".
- ◎ Peters, J. F., & Pedrycz, W. (2001). Engenharia de software: teoria e prática. Rio de Janeiro: Campus, 681(519.683), 2.

# Referências

- ◎ Pressman, R. S. (2006) Engenharia de software. 6<sup>a</sup>. ed. São Paulo: McGraw-Hill, xxxi, 720p.
- ◎ Reutilização de Software - Engenharia de Software Magazine 39: Disponível em: <http://www.devmedia.com.br/reutilizacao-de-software-revista-engenharia-de-software-magazine-39/21956>
- ◎ Sommerville, I. (2007) Engenharia de Software-8<sup>a</sup> Edição. Ed Person Education.
- ◎ Trigaux, J. C., & Heymans, P. (2003). Modelling variability requirements in software product lines: a comparative survey. EPH3310300R0462/215315, FUNDP–Equipe LIEL, Namur.

# Referências

- ◎ Pressman, R. S. (2006) Engenharia de software. 6<sup>a</sup>. ed. São Paulo: McGraw-Hill, xxxi, 720p.
- ◎ Reutilização de Software - Engenharia de Software Magazine 39: Disponível em: <http://www.devmedia.com.br/reutilizacao-de-software-revista-engenharia-de-software-magazine-39/21956>
- ◎ Sommerville, I. (2007) Engenharia de Software-8<sup>a</sup> Edição. Ed Person Education.
- ◎ Trigaux, J. C., & Heymans, P. (2003). Modelling variability requirements in software product lines: a comparative survey. EPH3310300R0462/215315, FUNDP–Equipe LIEL, Namur.



muito obrigada.



Perguntas?

# EXEMPLOS

# 1

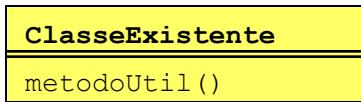
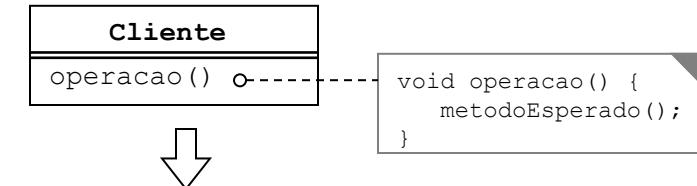
## Adapter

*"Objetivo: converter a interface de uma classe em outra interface esperada pelos clientes. Adapter permite a comunicação entre classes que não poderiam trabalhar juntas devido à incompatibilidade de suas interfaces."*

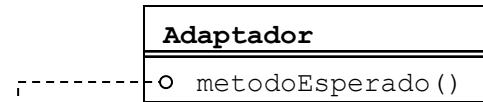
[GoF]

# Problema e Solução

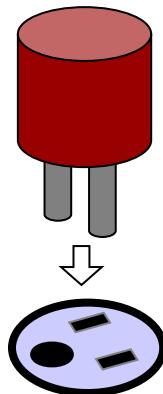
## Problema



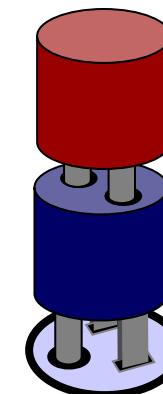
## Solução



```
void metodoEsperado()  
{  
    metodoUtil();  
}
```

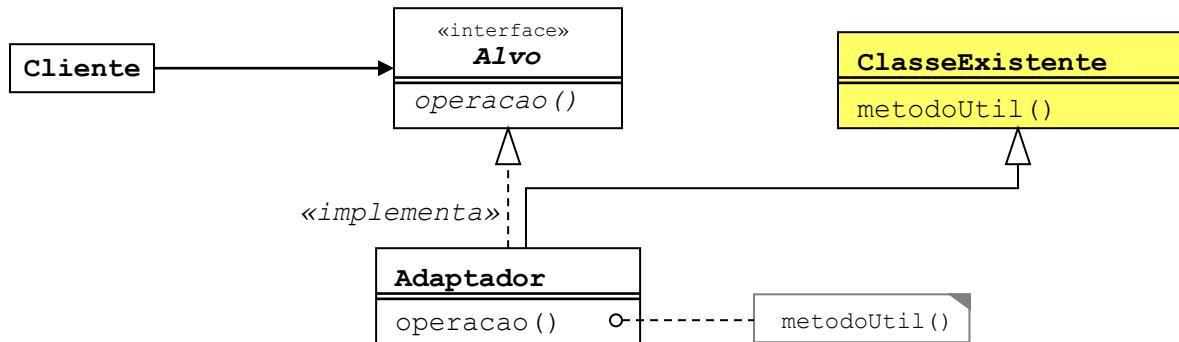


**Adaptador**



# Duas formas de Adapter

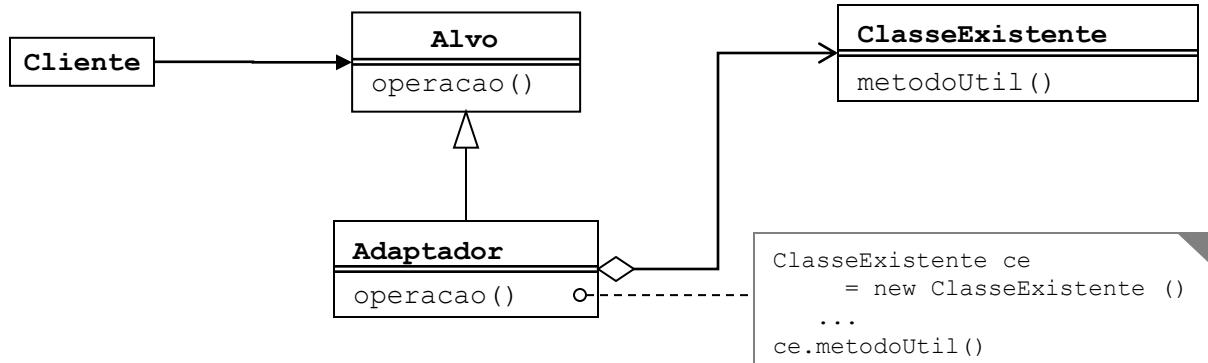
- *Class Adapter*: usa herança múltipla



- *Cliente*: aplicação que colabora com objetos aderentes à interface *Alvo*
- *Alvo*: define a interface requerida pelo *Cliente*
- *ClasseExistente*: interface que requer adaptação
- *Adaptador (Adapter)*: adapta a interface do Recurso à interface *Alvo*

## Duas formas de Adapter

- *Object Adapter*: usa composição



- *Única solução se Alvo não for uma interface Java*
- *Adaptador possui referência para objeto que terá sua interface adaptada (instância de ClasseExistente).*
- *Cada método de Alvo chama o(s) método(s) correspondente(s) na interface adaptada.*

# Class Adapter em Java

```
public class ClienteExemplo {
    Alvo[] alvos = new Alvo[10];
    public void inicializaAlvos() {
        alvos[0] = new AlvoExistente();
        alvos[1] = new Adaptador();
        // ...
    }
    public void executaAlvos() {
        for (int i = 0; i < alvos.length; i++) {
            alvo.operacao();
        }
    }
}
```

```
public interface Alvo
{
    void operacao();
}
```

```
public class Adaptador extends ClasseExistente implements Alvo
{
    public void operacao() {
        String texto = metodoUtilDois("Operação Realizada.");
        metodoUtilUm(texto);
    }
}
```

```
public class ClasseExistente {
    public void metodoUtilUm(String texto) {
        System.out.println(texto);
    }
    public String metodoUtilDois(String texto) {
        return texto.toUpperCase();
    }
}
```

# Object Adapter em Java

```
public class ClienteExemplo {  
    Alvo[] alvos = new Alvo[10];  
    public void inicializaAlvos() {  
        alvos[0] = new AlvoExistente();  
        alvos[1] = new Adaptador();  
        // ...  
    }  
    public void executaAlvos() {  
        for (int i = 0; i < alvos.length; i++) {  
            alvos[i].operacao();  
        }  
    }  
}
```

```
public abstract class Alvo {  
    public abstract void operacao();  
    // ... resto da classe  
}
```

```
public class Adaptador extends Alvo {  
    ClasseExistente existente = new ClasseExistente();  
    public void operacao() {  
        String texto = existente.metodoUtilDois("Operação Realizada.");  
        existente.metodoUtilUm(texto);  
    }  
}
```

```
public class ClasseExistente {  
    public void metodoUtilUm(String texto) {  
        System.out.println(texto);  
    }  
    public String metodoUtilDois(String texto) {  
        return texto.toUpperCase();  
    }  
}
```

## *Quando usar?*

- *Sempre que for necessário adaptar uma interface para um cliente*
- *Class Adapter*
  - *Quando houver uma interface que permita a implementação estática*
- *Object Adapter*
  - *Quando menor acoplamento for desejado*
  - *Quando o cliente não usa uma interface Java ou classe abstrata que possa ser estendida*

## *Onde estão os adapters?*

- A API do J2SDK tem vários adapters
- Você consegue identificá-los?

# Padrões de Projeto

## *Introdução: interfaces*

- **Interface**: coleção de métodos e dados que uma classe permite que objetos de outras classes acessem
- **Implementação**: código dentro dos métodos
- **Interface Java**: componente da linguagem que representa apenas a interface de um objeto
  - Exigem que classe que implementa a interface ofereça implementação para seus métodos
  - Não garante que métodos terão implementação que faça efetivamente alguma coisa (chaves vazias): stubs.

# Padrões de Projeto

## Além das interfaces

- Padrões de design que concentram-se principalmente na adaptação de interfaces
  - *Adapter*: para adaptar a interface de uma classe para outra que o cliente espera
  - *Façade*: oferecer uma interface simples para uma coleção de classes
  - *Composite*: definir uma interface comum para objetos individuais e composições de objetos
  - *Bridge*: desacoplar uma abstração de sua implementação para que ambos possam variar independentemente

# Padrões de Projeto

## *Distribuição de responsabilidades*

- Os padrões a seguir estão principalmente associados a atribuições especiais de responsabilidade
  - *Singleton*: centraliza a responsabilidade em uma única instância de uma classe
  - *Observer*: desacopla um objeto do conhecimento de que outros objetos dependem dele
  - *Mediator*: centraliza a responsabilidade em uma classe que determina como outros objetos interagem
  - *Proxy*: assume a responsabilidade de outro objeto (intercepta)
  - *Chain of Responsibility*: permite que uma requisição passe por uma corrente de objetos até encontrar um que a processe
  - *Flyweight*: centraliza a responsabilidade em objetos compartilhados de alta granularidade (blocos de montagem)

# Padrões de Projeto

- Construtores em Java definem maneiras padrão de construir objetos. Sobrecarga permite ampla flexibilidade
- Alguns problemas em depender de construtores
  - Cliente pode não ter *todos os dados necessários para instanciar um objeto*
  - Cliente fica *acoplado* a uma implementação concreta (*precisa saber a classe concreta para usar new com o construtor*)
  - Cliente de herança *pode criar construtor* que chama métodos que dependem de valores ainda não inicializados (*vide processo de construção*)
  - *Objeto complexo* pode necessitar da criação de objetos menores previamente, com certo controle difícil de implementar com construtores
  - *Não há como limitar o número de instâncias criadas*

# Padrões de Projeto

## Além dos construtores

- Padrões que oferecem alternativas à construção de objetos
  - *Builder*: obtém informação necessária em passos antes de requisitar a construção de um objeto
  - *Factory Method*: adia a decisão sobre qual classe concreta instanciar
  - *Abstract Factory*: construir uma família de objetos que compartilham um "tema" em comum
  - *Prototype*: especificar a criação de um objeto a partir de um exemplo fornecido
  - *Memento*: reconstruir um objeto a partir de uma versão que contém apenas seu estado interno

# Padrões de Projeto

## Introdução: operações

- Definições essenciais
  - **Operação**: especificação de um serviço que pode ser requisitado por uma instância de uma classe. Exemplo: operação `toString()` é implementada em todas as classes.
  - **Método**: implementação de uma operação. Um método tem uma assinatura. Exemplo: cada classe implementa `toString()` diferentemente
  - **Assinatura**: descreve uma operação com um nome, parâmetros e tipo de retorno. Exemplo: `public String toString()`
  - **Algoritmo**: uma seqüência de instruções que aceita entradas e produz saída. Pode ser um método, parte de um método ou pode consistir de vários métodos.

# Padrões de Projeto

*Além das operações comuns*

- Vários padrões lidam com diferentes formas de implementar operações e algoritmos
  - *Template Method*: implementa um algoritmo em um método adiando a definição de alguns passos do algoritmo para que subclasses possam definir-los
  - *State*: distribui uma operação para que cada classe represente um estado diferente; encapsula **um estado**
  - *Strategy*: encapsula **um algoritmo** fazendo com que as implementações sejam intercambiáveis
  - *Command*: encapsula **uma instrução** em um objeto
  - *Interpreter*: distribui uma operação de tal forma que cada implementação se aplique a um tipo de composição diferente

# Padrões de Projeto

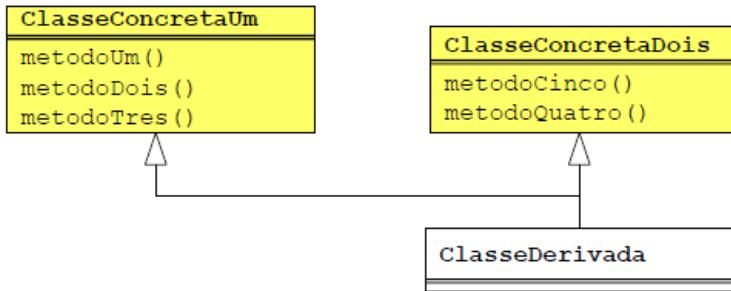
## *Introdução: Extensão*

- Extensão é a adição de uma classe, interface ou método a uma base de código existente [2]
- Formas de extensão
  - Herança (criação de novas classes)
  - Delegação (para herdar de duas classes, pode-se estender uma classe e usar delegação para "herdar" o comportamento da outra classe)
- Desenvolvimento em Java é sempre uma forma de extensão
  - Extensão começa onde o reuso termina

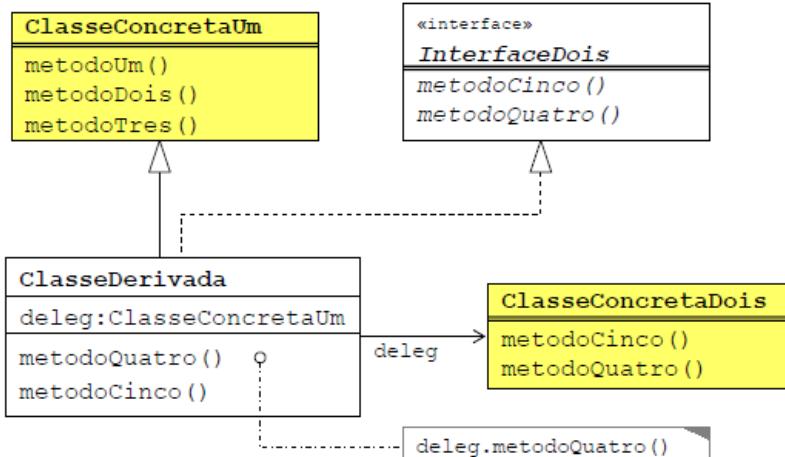
# Padrões de Projeto

*Exemplo de extensão  
por delegação*

*Efeito  
Desejado*



*Efeito Possível  
em Java*



■ Classes existentes  
□ Classes novas

# Padrões de Projeto

## Além da extensão

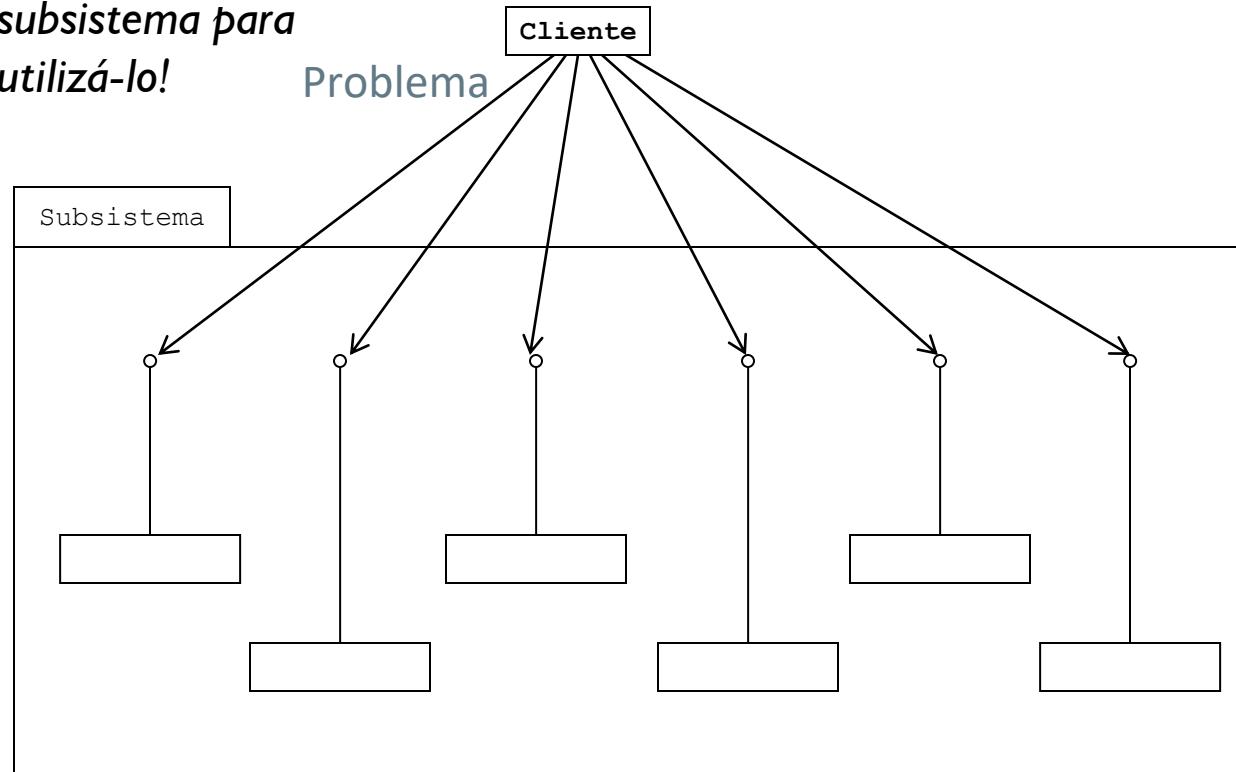
- Tanto herança como delegação exigem que se saiba, em tempo de compilação, que comportamentos são desejados. Os patterns permitem acrescentar comportamentos em um objeto sem mudar sua classe
- Principais classes
  - *Command* (capítulo anterior)
  - *Template Method* (capítulo anterior)
  - *Decorator*: adiciona responsabilidades a um objeto dinamicamente.
  - *Iterator*: oferece uma maneira de acessar uma coleção de instâncias de uma classe carregada.
  - *Visitor*: permite a adição de novas operações a uma classe sem mudar a classe.

# 2

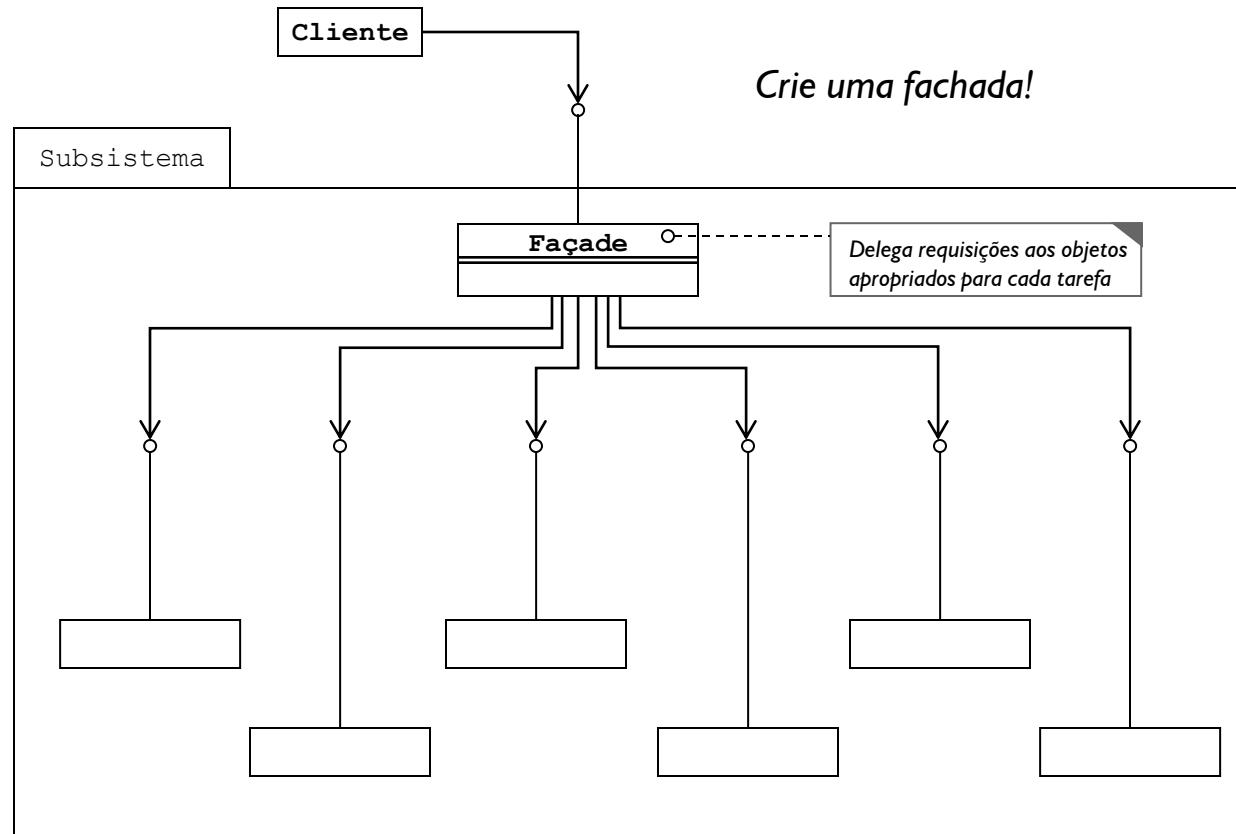
## Façade

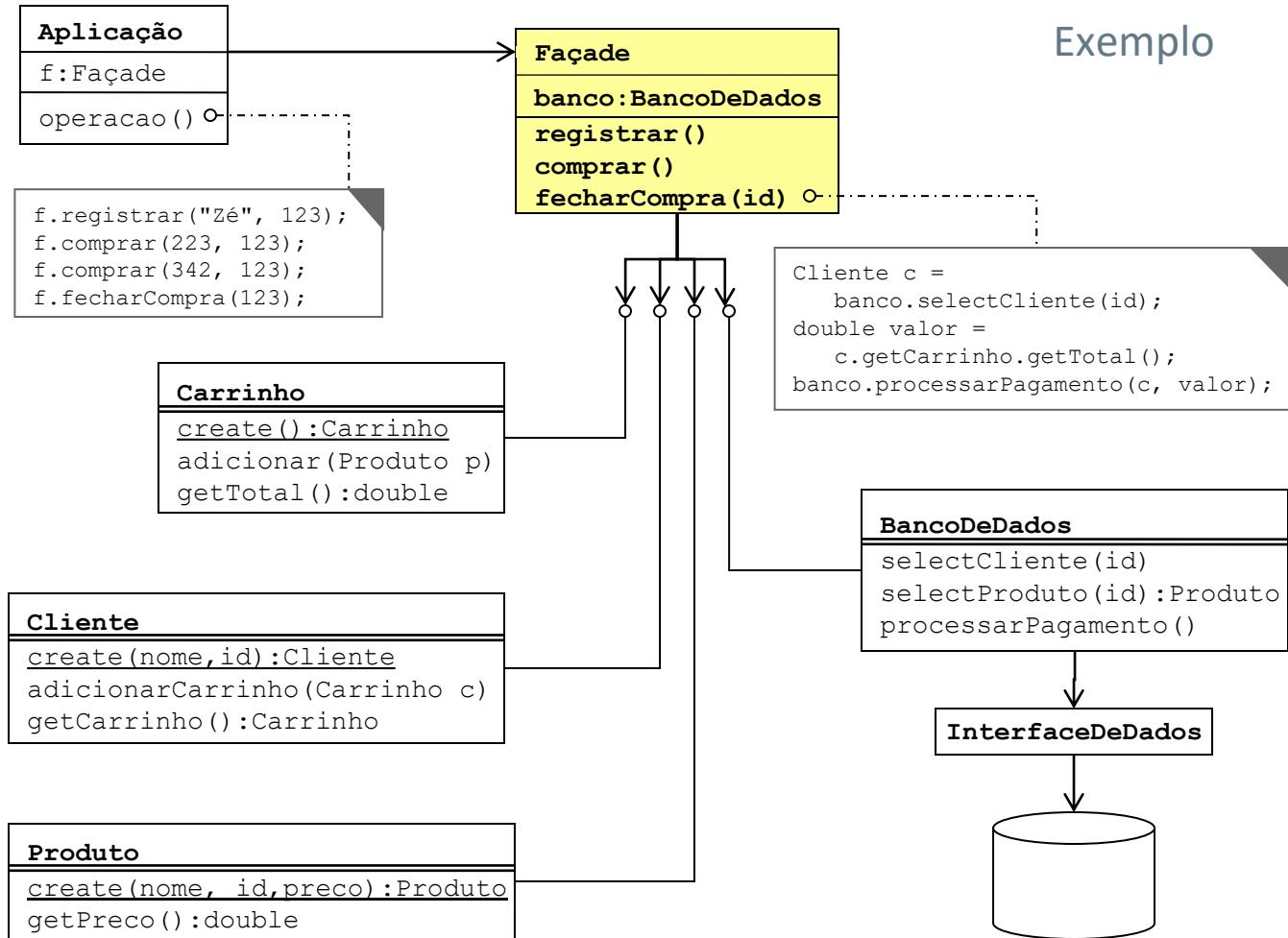
*"Oferecer uma interface única para um conjunto de interfaces de um subsistema. Façade define uma interface de nível mais elevado que torna o subsistema mais fácil de usar." [GoF]*

*Cliente precisa saber  
muitos detalhes do  
subsistema para  
utilizá-lo!*



## Estrutura de Façade





## Façade em Java

```
class Aplicação {
    ...
    Facade f;
    // Obtem instancia f
    f.registrar("Zé",
123);

    f.comprar(223, 123);
    f.comprar(342, 123);

    f.fecharCompra(123);
    ...
}
```

```
public class Facade {
    BancoDeDados banco = Sistema.obterBanco();
    public void registrar(String nome, int id) {
        Cliente c = Cliente.create(nome, id);
        Carrinho c = Carrinho.create();
        c.adicionarCarrinho();
    }
    public void comprar(int prodID, int clienteID) {
        Cliente c = banco.selectCliente(cliente ID);
        Produto p = banco.selectProduto(prodID) {
            c.getCarinho().adicionar(p);
        }
    public void fecharCompra(int clienteID) {
        Cliente c = banco.selectCliente(clienteID);
        double valor = c.getCarinho.getTotal();
        banco.processarPagamento(c, valor);
    }
}
```

```
public class Carrinho {
    static Carrinho create() {...}
    void adicionar(Produto p) {...}
    double getTotal() {...}
}
```

```
public class Produto {
    static Produto create(String nome,
                           int id, double preco) {...}
    double getPreco() {...}
}
```

```
public class Cliente {
    static Cliente create(String nome,
                          int id) {...}
    void adicionarCarrinho(Carrinho c)
    ...
    Carrinho getCarinho() {...}
}
```

```
public class BancoDeDados {
    Cliente selectCliente(int id) {...}
    Produto selectProduto(int id) {...}
    void processarPagamento() {...}
}
```

- ◎ Sempre que for desejável criar uma interface para um conjunto de objetos com o objetivo de **facilitar o uso da aplicação**
  - Quando usar?
- Permite que objetos individuais cuidem de uma única tarefa, deixando que a fachada se encarregue de divulgar as suas operações
- Fachadas viabilizam a separação em camadas com alto grau de desacoplamento
- Existem em várias partes da aplicação
  - Fachada da aplicação para interface do usuário
  - Fachada para sistema de persistência: Data Access Object

◎ Fachadas podem oferecer maior ou menor isolamento entre aplicação cliente e objetos  
Nível de acoplamento

◎ Nível ideal deve ser determinado pelo nível de acoplamento desejado entre os sistemas

◎ A fachada mostrada como exemplo isola totalmente o cliente dos objetos

```
Facade f; // Obtem instancia f
f.registrar("Zé", 123);
```

◎ Outra versão com menor isolamento (requer que aplicação-cliente conheça objeto Cliente)

```
Cliente joao = Cliente.create("João", 15);
f.registrar(joao); // método registrar(Cliente c)
```

- ◎ Você conhece algum exemplo de na API do J2SDK
- ◎ Onde você criaria fachadas em uma aplicação Web J2EE?
- ◎ Você saberia dizer qual a diferença entre Façade e Adapter

# Padrões de Projeto

Você está desenvolvendo um sistema para a empresa Starbuzz e deve guardar informações sobre a preparação de bebidas:

## Receita de Café Starbuzz

- (1) Ferver um pouco de água
- (2) Mistrurar o café na água quente
- (3) Servir o café na xícara
- (4) Acrescentar açúcar e leite

## Receita de Chá Starbuzz

- (1) Ferver um pouco de água
- (2) Misturar o chá em infusão de água fervente
- (3) Despejar o chá na xícara
- (4) Acrescentar limão

**ATENÇÃO PREPARADORES:** sigam criteriosamente estas receitas sempre que preparam bebidas Starbuzz.

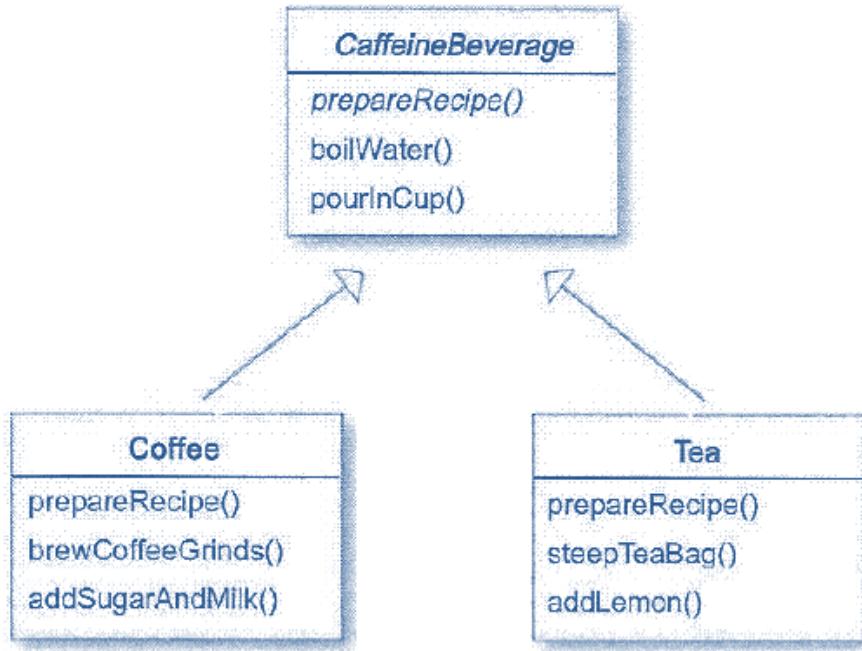
# Uma Solução....

```
public class Cafe {  
    public void preparaReceita() {  
        fervaAgua();  
        adicionaCafeNaAgua();  
        derrameNaXicara();  
        adicionaAcucaELeite();  
    }  
    public void fervaAgua() {  
        System.out.println("Fervendo a água.");  
    }  
    ...  
}
```

# Uma Solução....

```
public class Cha {  
    public void preparaReceita(){  
        fervaAgua();  
        adicionaSacoChaNaAgua();  
        derrameNaXicara();  
        adicionaLimao();  
    }  
    public void fervaAgua(){  
        System.out.println("Fervendo a água.");  
    }  
    ...  
}
```

# Padrões de Projeto



# Padrões de Projeto

## PADRÃO TEMPLATE METHOD

### Classificação

- Comportamental de classe.

### Propósito

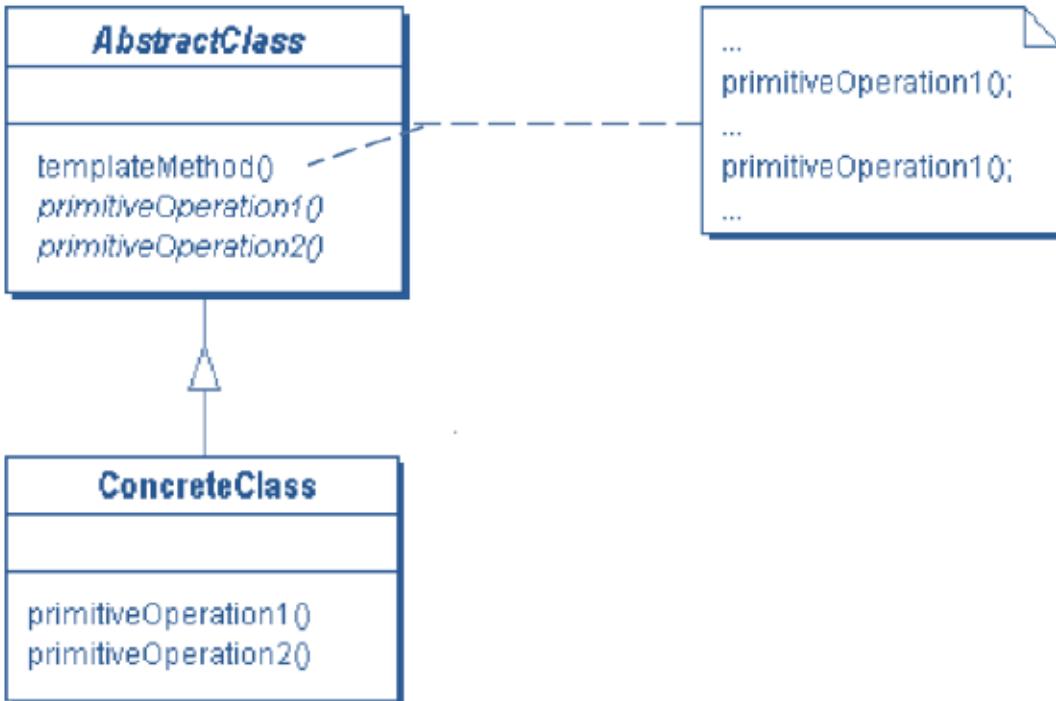
- Definir o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses.
- Template Method permite que suas subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura.

### Aplicação

- Quando se deseja definir um algoritmo geral, que estabelece uma série de passos para cumprir um requisito da aplicação. Porém, seus passos podem variar e é desejável que a estrutura da implementação forneça uma forma para que eles sejam facilmente substituídos.

# Padrões de Projeto

## PADRÃO TEMPLATE METHOD



# Padrões de Projeto

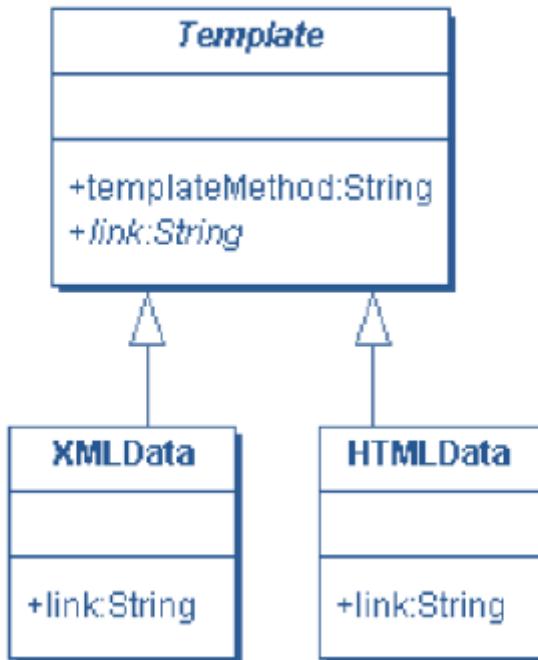
- **AbstractClass**

- Implementa um método template que define o esqueleto de um algoritmo. O método template chama as operações abstratas.
- Define operações primitivas abstratas que implementam passos de um algoritmo.

- **ConcreteClass**

- Implementa as operações primitivas para realizar passos específicos do algoritmo.

# Padrões de Projeto



# Padrões de Projeto

## PADRÃO TEMPLATE METHOD EM AÇÃO!



```
public abstract class Template{  
    public abstract String link(String texto, String url);  
    public String transform(String texto){  
        return texto;  
    }  
    public String templateMethod(){  
        String msg = "Endereço" +  
                    link("Empresa","http://www.empresacom");  
        return msg;  
    }  
}
```

# Padrões de Projeto

## PADRÃO TEMPLATE METHOD EM AÇÃO!

```
public class XMLData extends Template{  
    public String link(String texto, String url){  
        return "<endereco xlink:href ='" +url+ "'>" +  
            texto+"</endereco>";  
    }  
}  
  
public class HTMLData extends Template{  
    public String link(String texto, String url){  
        return "<a href='"+url+"'>"+texto+"";  
    }  
    public String transform(String texto) {  
        return texto.toLowerCase();  
    }  
}
```

# Padrões de Projeto

## Padrão Template Method

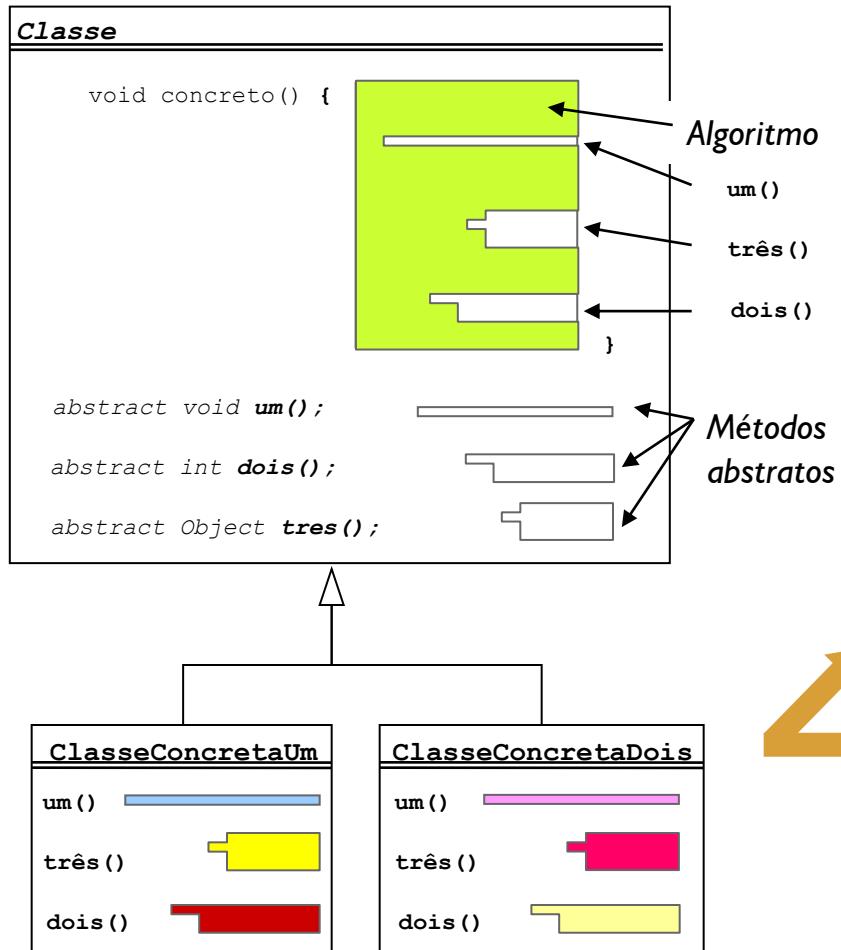
- Podemos reaproveitar o código relativo à parte comum de um algoritmo, permitindo que cada passo variável possa ser definido na subclasse.
- Isso também é uma forma de permitir que a funcionalidade da classe que define o algoritmo básico seja estendida. Assim é possível definir uma funcionalidade mais geral na qual pode ser facilmente incorporada a parte específica do domínio da aplicação.
- É importante lembrar que os modificadores adequados dos métodos devem ser utilizados para impedir que o contrato da superclasse com os seus clientes seja quebrado.



## 16

# Template Method

*"Definir o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses. Template Method permite que suas subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura." [GoF]*



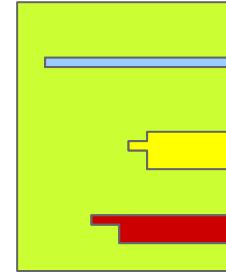
## Problema

### Algoritmos resultantes

```

Classe x =
new ClasseConcretaUm()
x.concreto()

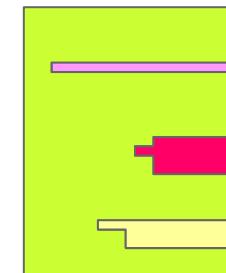
```



```

Classe x =
new ClasseConcretaDois()
x.concreto()

```



## Solução: Template Method

- ◎ O que é um Template Method
- Um Template Method define um algoritmo em termos de operações abstratas que subclasses sobrepõem para oferecer comportamento concreto
- ◎ Quando usar?
- Quando a estrutura fixa de um algoritmo puder ser definida pela superclasse deixando certas partes para serem preenchidos por implementações que podem variar

```
public abstract class Template {  
    protected abstract String link(String texto, String url);  
    protected String transform(String texto) { return texto; }  
    public final String templateMethod() {  
        String msg = "Endereço: " + link("Empresa", "http://www.empresacom");  
        return transform(msg);  
    }  
}
```

### Template Method em Java

```
public class XMLData extends Template {  
    protected String link(String texto, String url) {  
        return "<endereco xlink:href='"+url+"'>" + texto + "</endereco>";  
    }  
}
```

```
public class HTMLData extends Template {  
    protected String link(String texto, String url) {  
        return "<a href='"+url+"'>" + texto + "</a>";  
    }  
    protected String transform(String texto) {  
        return texto.toLowerCase();  
    }  
}
```

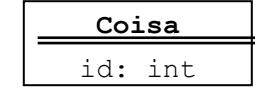
## Exemplo no J2SDK

◎ O método `Arrays.sort (java.util)` é um bom exemplo de Template Method. Ele recebe como parâmetro um objeto do tipo `Comparator` que implementa um método `compare(a, b)` e utiliza-o para definir as regras de ordenação

```
public class MedeCoisas implements Comparator<Coisa> {

    public int compare(Coisa c1, Coisa c2) {
        if (c1.getID() > c2.getID()) return 1;
        if (c1.getID() < c2.getID()) return -1;
        if (c1.getID() == c2.getID()) return 0;
    }
}

...
Coisa coisas[] = new Coisa[10];
coisas[0] = new Coisa("A");
coisas[1] = new Coisa("B");
...
Arrays.sort(coisas, new MedeCoisas());
...
```



Método retorna 1, 0 ou -1 para ordenar Coisas pelo ID

## Questões

- ◎ Cite outros exemplos de Template Method
- Em Java
- Em frameworks
- ◎ Que outras aplicações poderiam ser implementadas com Template Method?