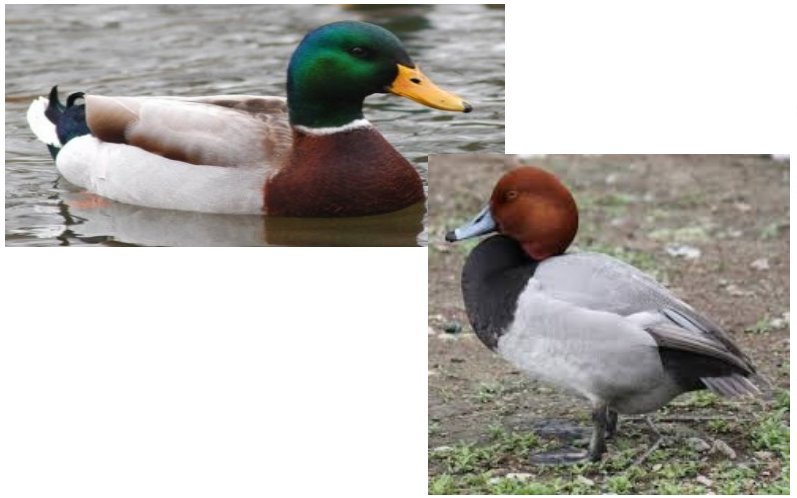
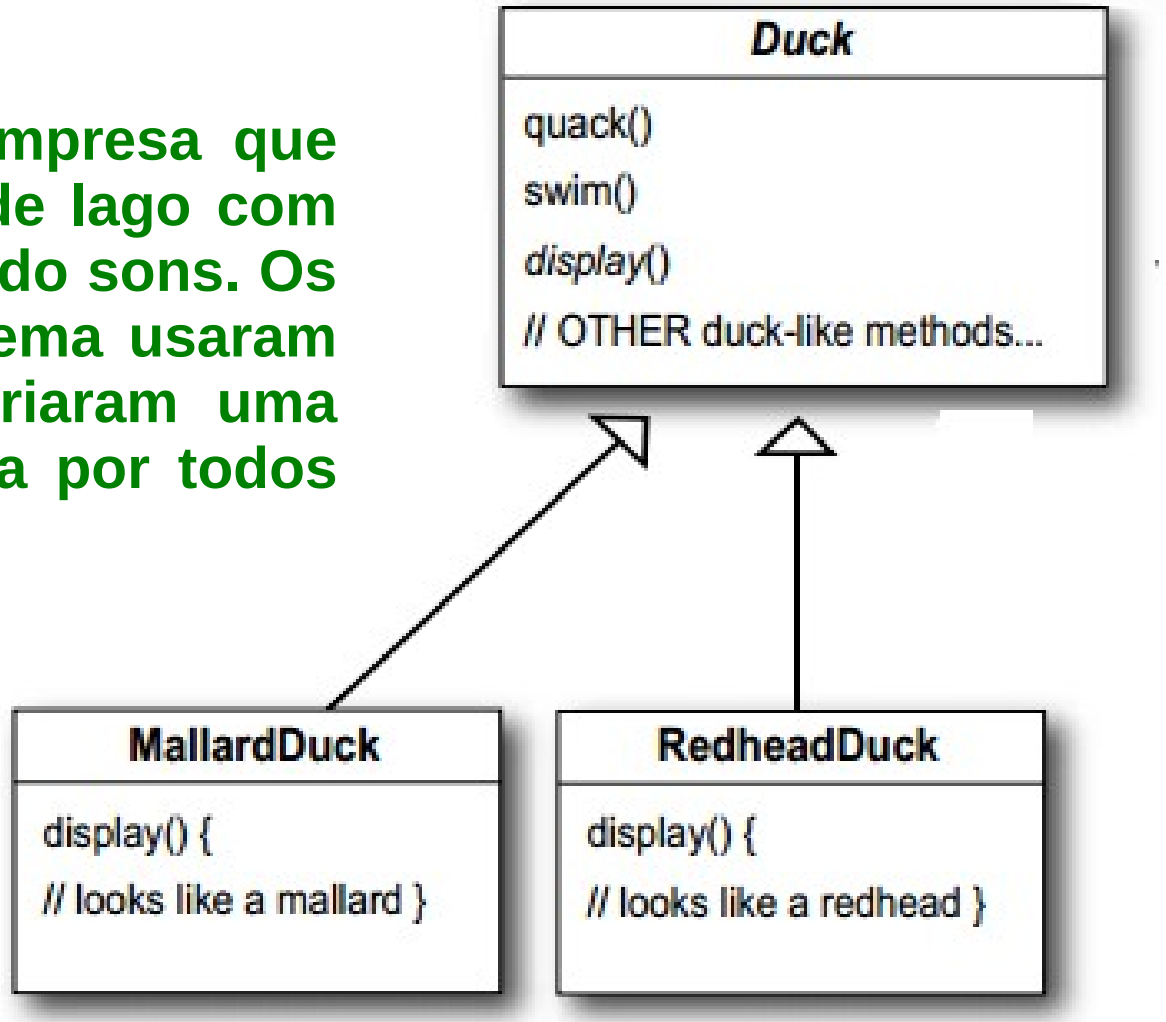
Padrões de Projeto (*Design Patterns*)

Alguém resolveu seus problemas!!!

Começou com um simples aplicativo SimUDuck

Joe trabalha para uma empresa que cria jogos de simulação de lago com patos nadando e produzindo sons. Os designers iniciais do sistema usaram técnicas OO padrão e criaram uma superclasse *Duck* herdada por todos os outros tipos de patos.





Começou com um simples aplicativo SimUDuck

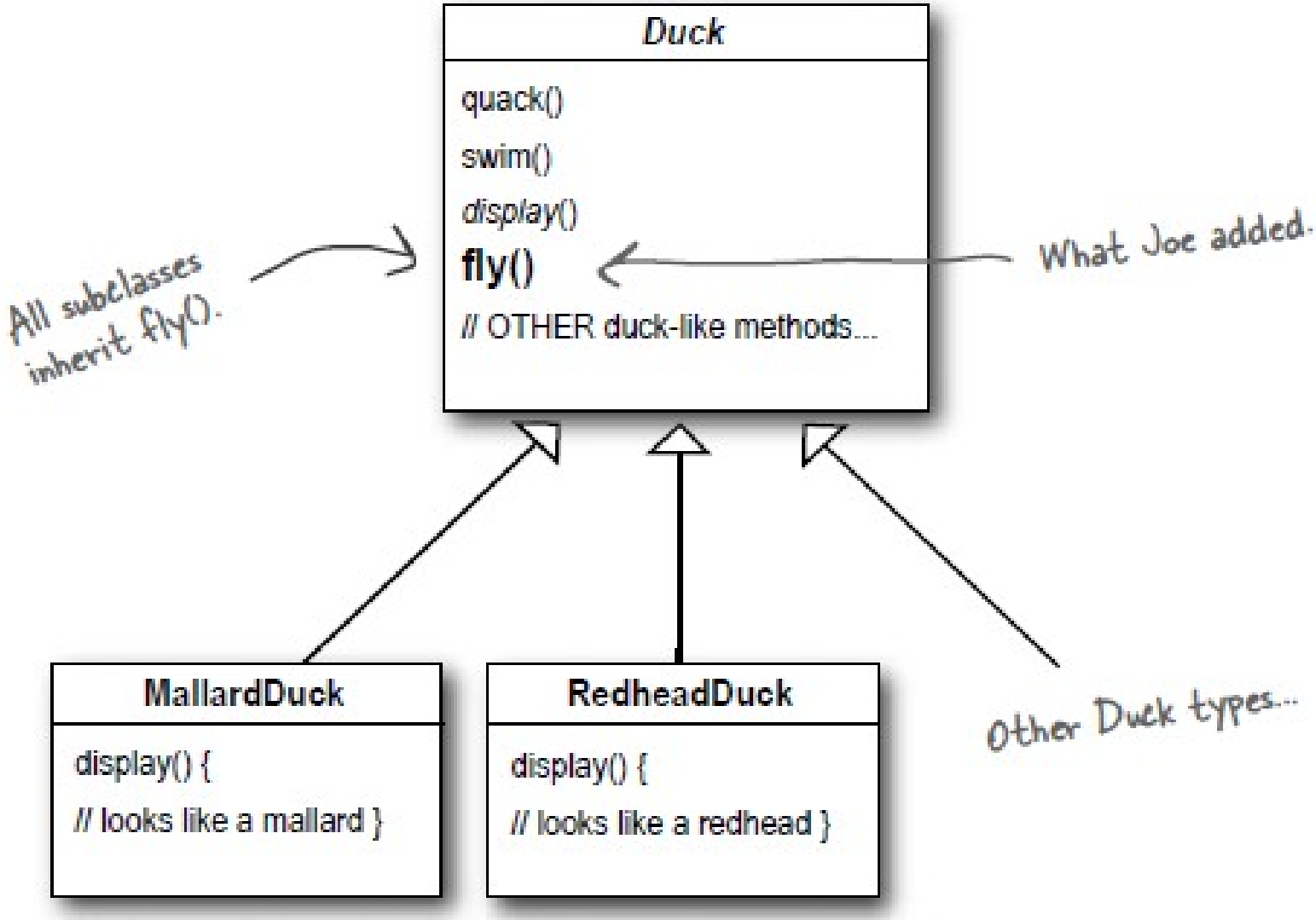
Ano passado, a empresa esteve sob pressão crescente dos concorrentes. Após uma sessão de *brainstorming* de uma semana jogando golfe, os executivos da empresa acham que está na hora de fazer uma grande inovação. Eles precisam de algo realmente impressionante para mostrar na reunião de acionistas em Maui na *semana seguinte!!*

Agora precisamos que os patos voem!!

Os executivos decidiram que fazer os patos voarem é o que o simulador deve fazer para acabar com a concorrência!!
O gerente comunicou que o Joe poderia criar algo em uma semana, afinal ele é programador OO e não pode ser difícil...

Agora precisamos que os patos voem!!

Agora precisamos que os patos voem!!



Mas alguma coisa deu muito errado!!!!

Joe, estou na reunião com os Acionistas. Eles acessaram uma demonstração e havia **patos de borracha** voando pela tela. Isso foi alguma piada?

Ok, então existe uma pequena falha em meu Design. Não sei porque eles não podem chamar Isso de um “recurso”. É até bonitinho...

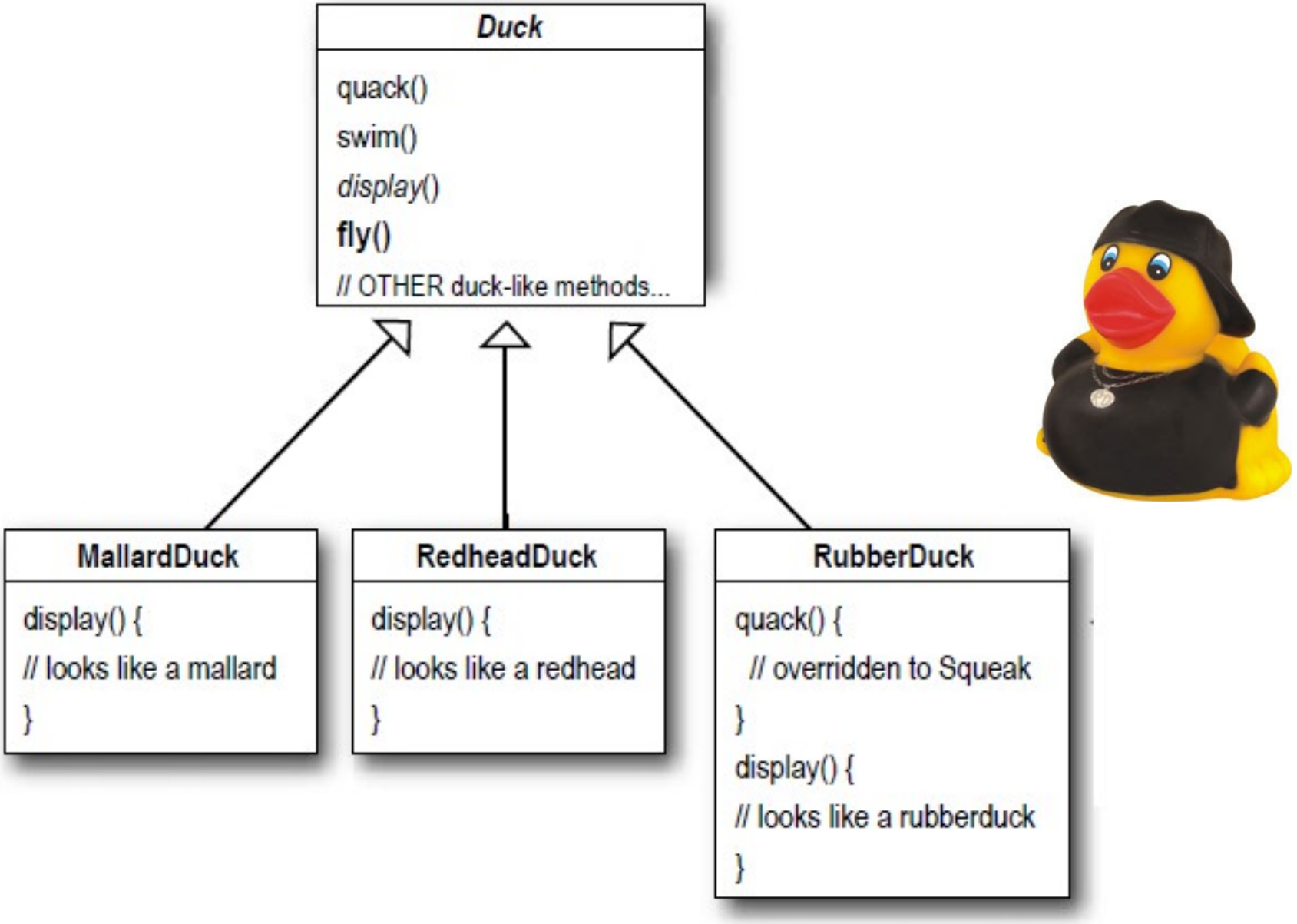
O que aconteceu?

- O que ele pensou que fosse um excelente uso de herança para fins de reutilização não dá tão certo quando se trata de manutenção.

- Nem todas as subclasses de Duck deveriam voar

- Uma atualização localizada no código causou um efeito colateral não-local (patos de borracha voadores!!)

O que aconteceu? Como resolver?



Pensando sobre herança...

Eu poderia substituir sempre o método fly() em pato de borracha como faço com o método quack()...

RubberDuck

```
quack()  { // squeak}
display() { // rubber duck }
fly() {
  // override to do nothing
}
```

Mas então o que acontece quando adicionamos patos de madeira ao programa? Eles não devem voar nem grasnar...

DecoyDuck

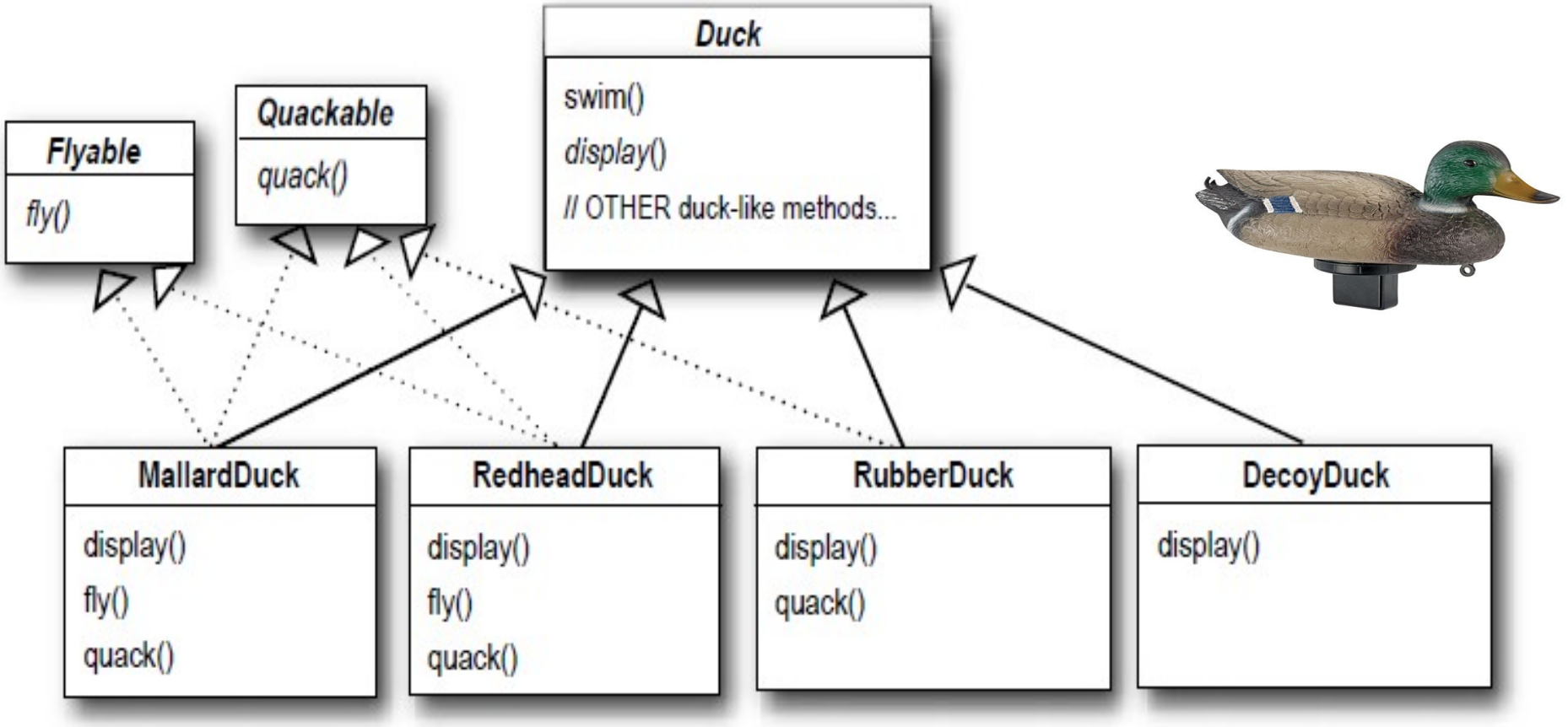
```
quack() {
  // override to do nothing
}

display() { // decoy duck}

fly() {
  // override to do nothing
}
```

Qual a sua ideia?

Que tal uma interface?



O que você acha desse projeto?

Que tal uma interface?

Esta é a ideia mais idiota que você já teve. Você consegue dizer "código duplicado”? Se você achava que ter que substituir alguns métodos era ruim, como irá se sentir quando precisar fazer uma pequena alteração no comportamento de voo... de todas as 48 subclasses de voo de Duck?

O que você faria se fosse Joe??

Que tal uma interface?

- Nem todas as subclasses devem ter o comportamento de voar ou grasnar

- Fazer subclasses implementar Flyable e/ou quackable resolve parte do problema mas....
  - Destrói completamente a reutilização de código para esses comportamentos...
  - Pode haver mais um tipo de comportamento de voo até entre os patos que voam.

Então, como resolver o problema?

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

ALTERAÇÃO!!!!

- Herança não funcionou muito bem
- Nem todas as subclasses possuem o mesmo comportamento
- A ideia da Interface pareceu promissora, mas classes de interface não possuem implementação, acabando com a reutilização
- Se o comportamento mudar, todas as subclasses onde esse comportamento é definido devem ser alteradas

Então, como resolver o problema?

Princípio de projeto

Identifique os aspectos de seu aplicativo que variam e separe-os do que permanece igual

- Pegue o que variar e “encapsule” para que isso não afete o resto do seu código.
  - Assim é possível alterar ou estender as partes que variam sem afetar as que não variam

Esse conceito simples forma a base de quase todos os padrões de projeto.

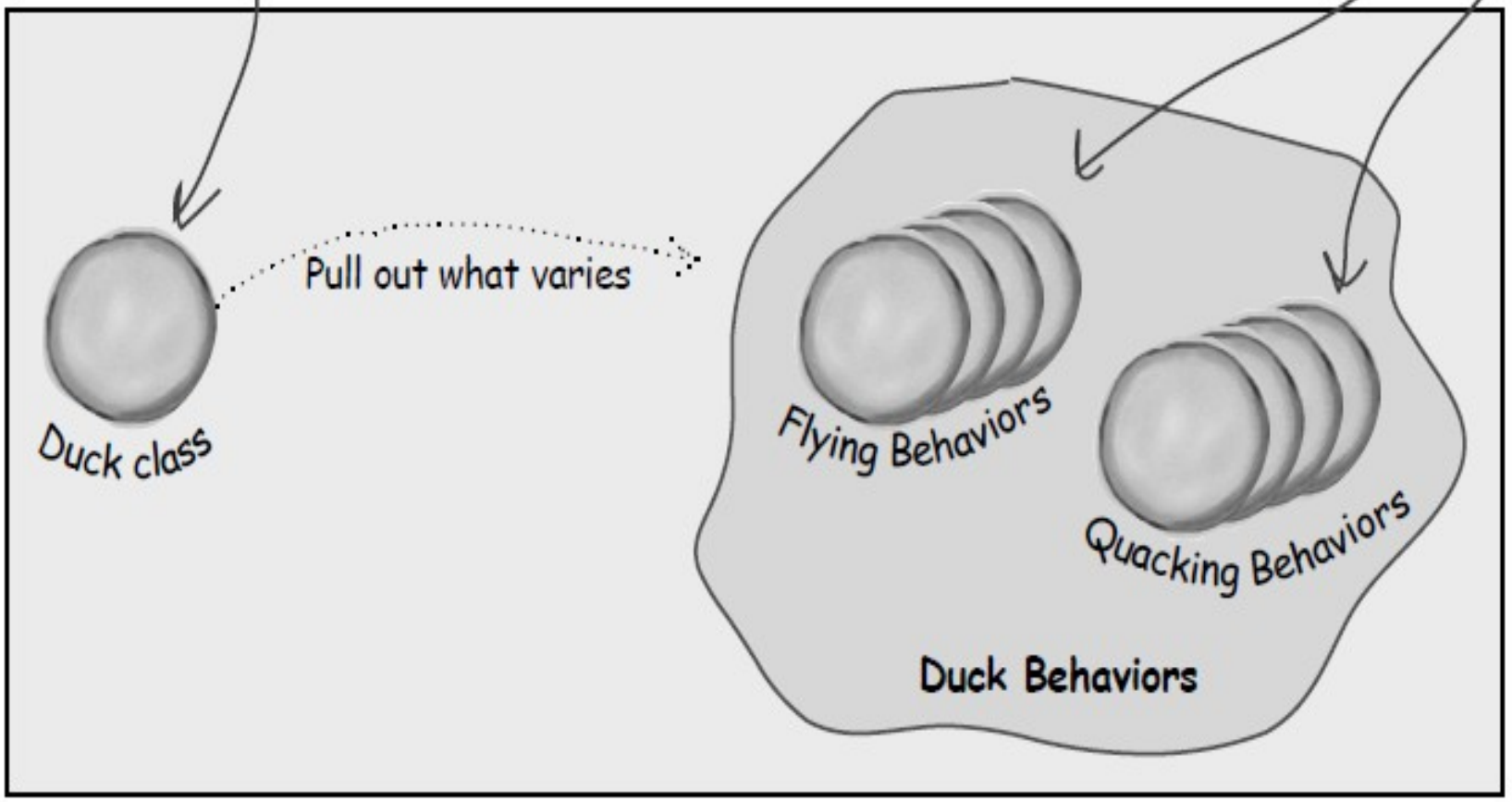
Então, como resolver o problema?

- Ok, hora de tirar o comportamento do pato das classes Duck!!
  - Sabemos que fly() e quack() são as partes da classe Duck que variam entre os patos
  - Para separar esses comportamentos da classe Duck, iremos tirar os dois métodos da classe Duck e criar um novo conjunto de classes para representar cada comportamento.

Então, como resolver o problema?

A classe Duck ainda é a superclasse de todos os patos, mas estamos tirando os comportamentos voar e grasnar e colocando-os em outra estrutura de classe

Várias implementações de comportamento irão viver aqui!!

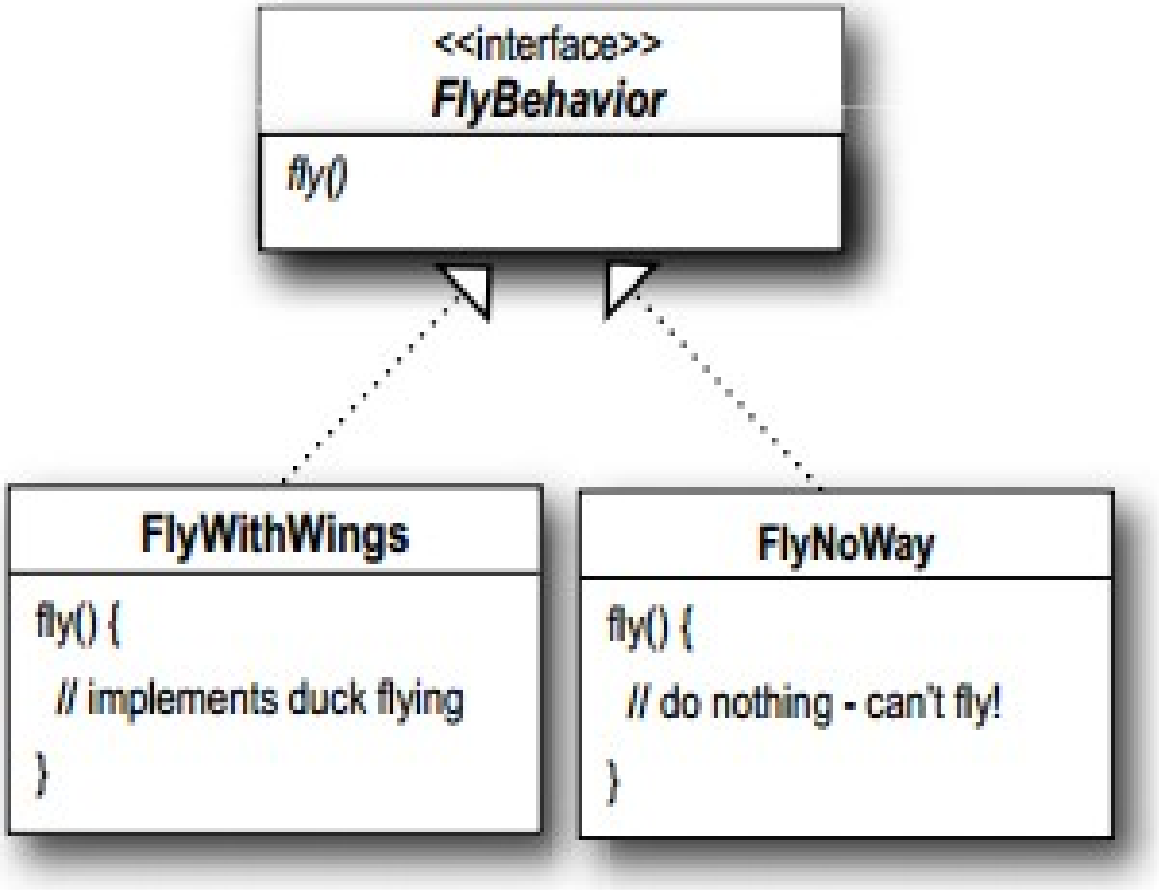


- A partir de agora, os comportamentos de Duck ficarão em uma classe separada – uma classe que implementa uma determina interface de comportamento

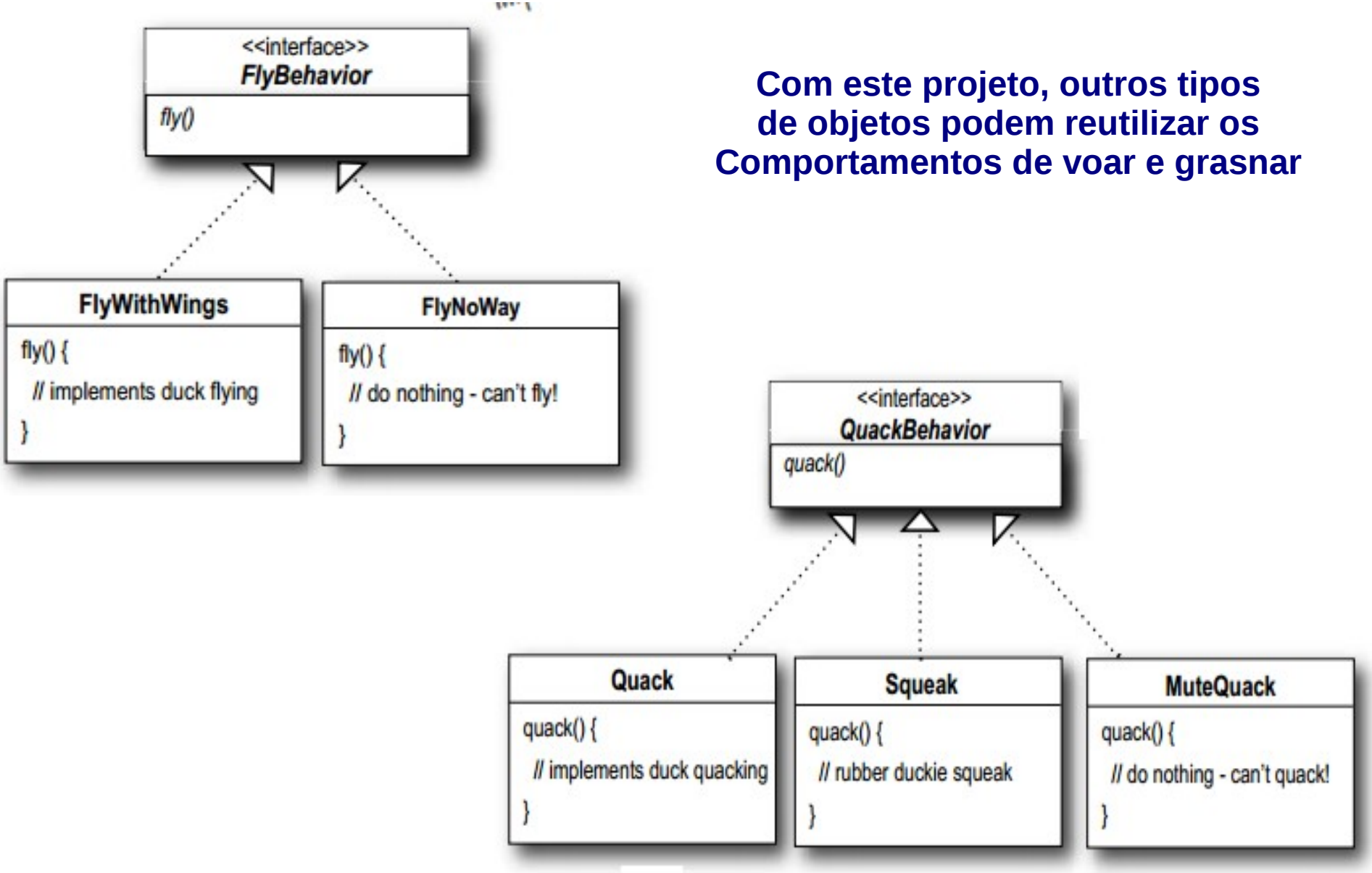
- Assim, as classes de Duck não vão precisar conhecer nenhum detalhe de implementação para seus comportamentos

Princípio de projeto

Programe para uma interface,
não para uma implementação



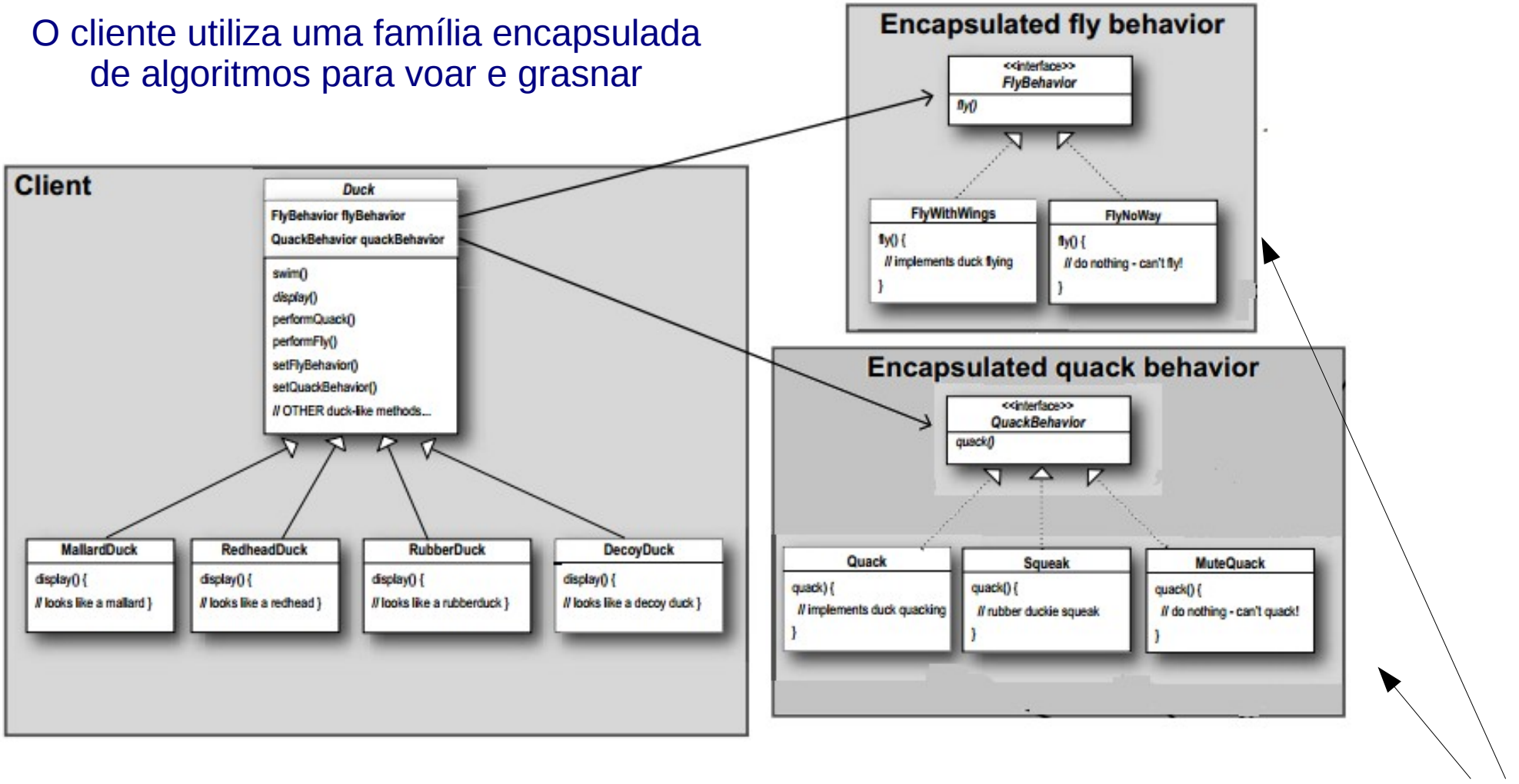
Implementando os comportamentos de Duck



Com este projeto, outros tipos de objetos podem reutilizar os Comportamentos de voar e grasnar

Uma visão geral

O cliente utiliza uma família encapsulada de algoritmos para voar e grasnar



Pense em cada conjunto de comportamentos como uma família de algoritmos

TEM-UM pode ser melhor do que É-UM

- Cada pato tem um FlyBehavior e um QuackBehavior aos quais delega a capacidade de grasnar e voar
- Quando duas classes são juntas dessa forma estamos usando **composição**
- Em vez de herdar seu comportamento, os patos obtêm seu comportamento ao serem compostos com o objeto de comportamento certo

Princípio de projeto

Dar prioridade a composição

- A composição dá mais flexibilidade ao projeto, permite encapsular uma família de algoritmos em seu próprio conjunto de classes e alterar o comportamento de um objeto em tempo de execução
- A composição é usada em muitos padrões de projetos!!

E por falar em padrões de projetos....

Parabéns por conhecer seu primeiro padrão!!!

O Padrão STRATEGY define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. A estratégia deixa o algoritmo variar independentemente dos clientes que o utilizam.

Um pouco de formalidade...

- Um padrão de projeto descreve um problema comum que ocorre regularmente no desenvolvimento de software e descreve uma solução geral para esses problemas que pode ser utilizada em muitos contextos diferentes

- Em geral, a solução é a descrição de um pequeno conjunto de classe e suas interações

Padrão de Projetos

- Um padrão tem 4 elementos essenciais:
  - O nome
  - O problema
    - Descreve quando aplicar o padrão
    - Lista de condições que devem ser satisfeitas para que faça sentido aplicar o padrão
  - A solução
    - Descreve elementos que compõem o projeto, seus relacionamentos, suas responsabilidades e colaborações
  - As consequências
    - São os resultados e análises das vantagens e desvantagens da aplicação do padrão

Catálogo de Padrões de Projeto

- Padrões de Criação
  - Se preocupam com os processos de criação de classes e objetos
  - Classe
    - Factory Method
  - Objeto
    - Abstract Factory (Objeto)
    - Builder
    - Prototype
    - Singleton

Catálogo de Padrões de Projeto

- Padrões Estruturais
  - Lidam com a composição de classes ou objetos
  - Classe
    - Adapter (Class)
  - Objeto
    - Adapter (object)
    - Bridge
    - Composite
    - Decorator
    - Façade
    - Flyweigth
    - Proxy

Catálogo de Padrões de Projeto

- Padrões Comportamentais
  - Caracterizam as maneiras pelas quais classes ou objetos interagem e distribuem responsabilidades
  - Classe
    - Interpreter
    - Template Method
  - Objeto
    - Chain of Responsibility
    - Command
    - Iterator
    - Mediator
    - Memento
    - Observer
    - State
    - Strategy
    - Visitor

Padrões mais simples e mais comuns

- Abstract Factory
- Adapter
- Composite
- Decorator
- Factory Method
- Observer
- Strategy
- Template Method

Padrão STRATEGY

- *Classificação:*
  - Propósito: Comportamental
  - Escopo: Objetos

- Intenção:
  - define uma família de algoritmos, encapsula cada algoritmo e os torna intercambiáveis, permitindo que o algoritmo varie independente dos clientes que o utilizam

Padrão STRATEGY

- *Motivação:*

  - Clientes que necessitam de diferentes algoritmos se tornam mais complexos se os incluírem em seu código

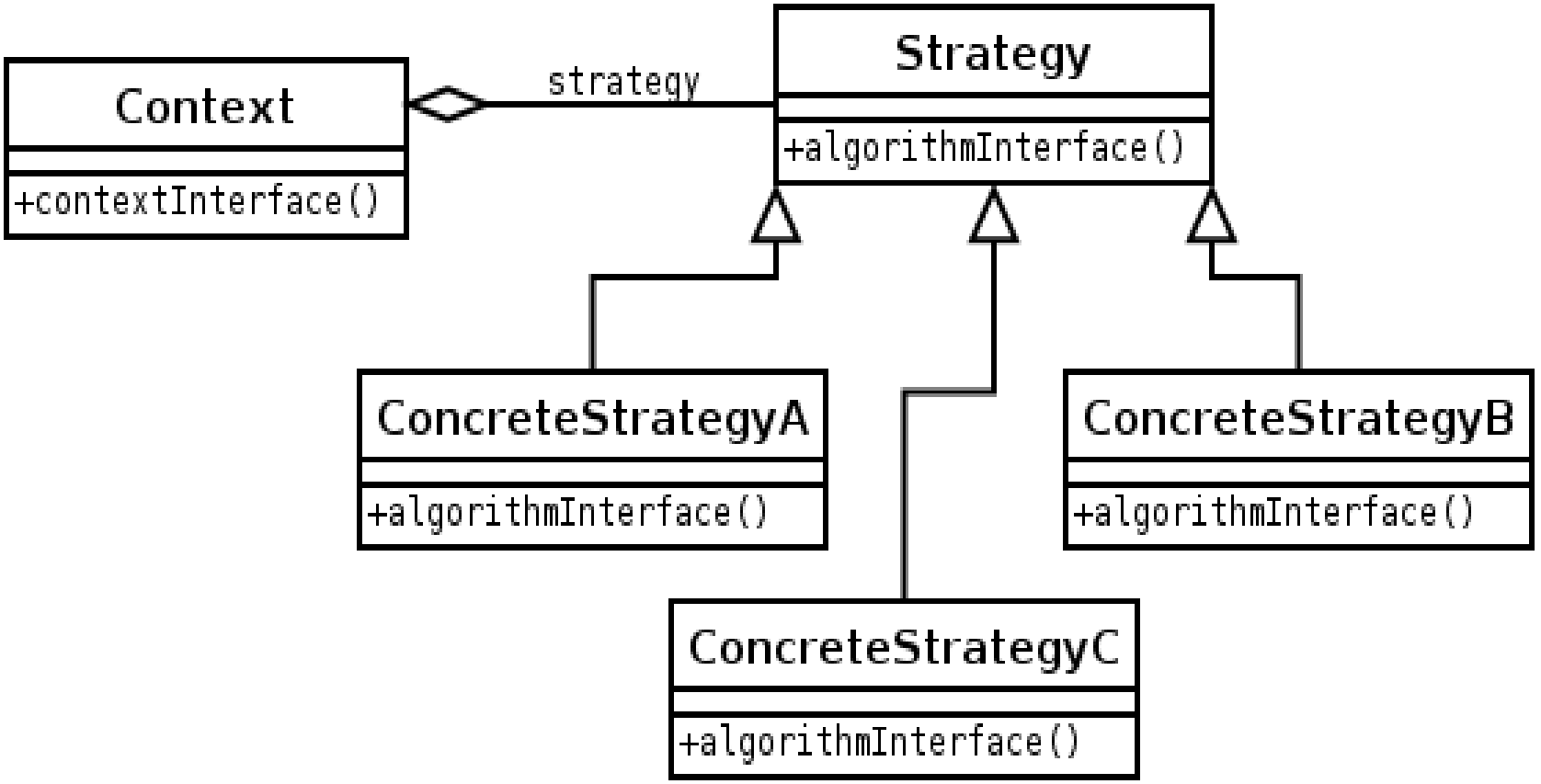
  - Diferentes algoritmos são adequados em diferentes situações na resolução de um mesmo problema

Padrão STRATEGY

- *Aplicabilidade (use strategy quando):*
  - Muitas classes relacionadas diferem somente no seu comportamento
  - Você necessita de variantes de um algoritmo
  - Um algoritmo usa dados sobre os quais o cliente não precisa ter conhecimento

Padrão STRATEGY

– *Estrutura:*



Padrão STRATEGY

- *Participantes*
  - Strategy: define uma interface comum para todos os algoritmos suportados
  - ConcreteStrategy: implementa o algoritmo usando a interface de **Strategy**
  - Context:
    - É configurado com um objeto **ConcreteStrategy**
    - Mantém uma referência para um objeto **Strategy**
    - Pode definir uma interface que permite a **Strategy** acessar seus dados

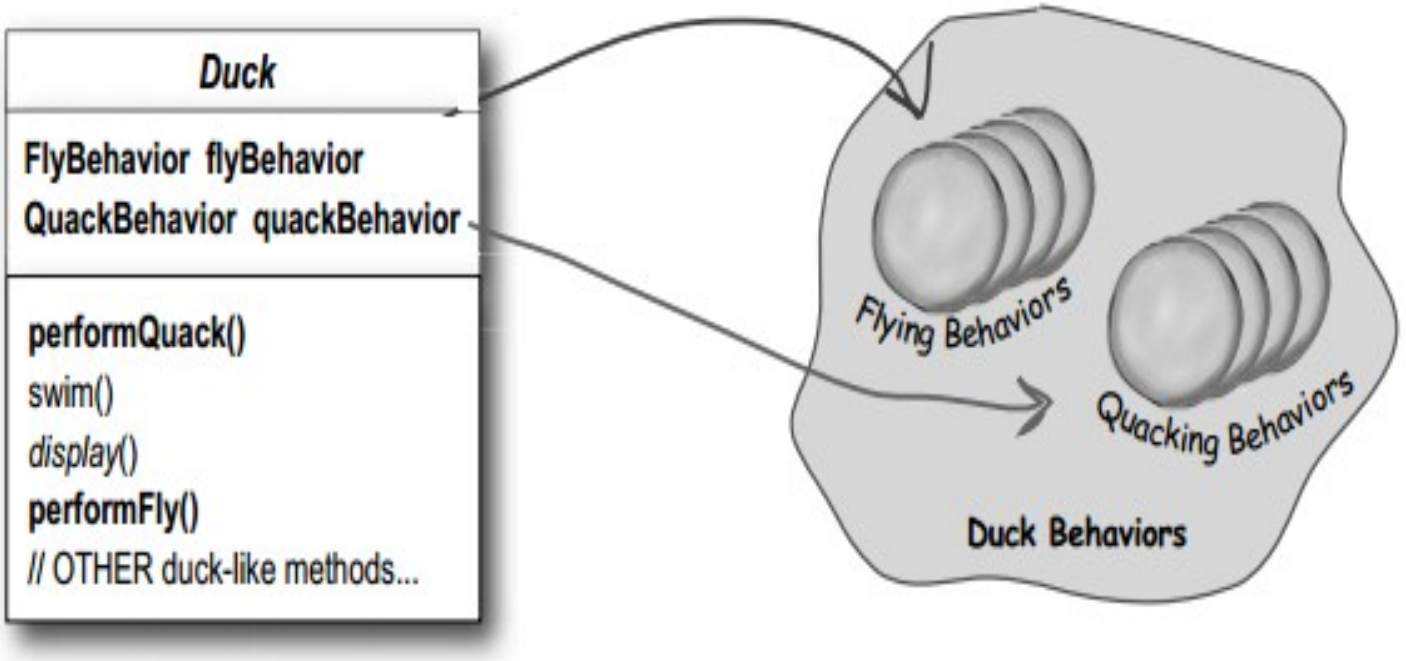
Padrão STRATEGY

– *Colaborações*

- **Strategy** e **Context** interagem para implementar o algoritmo escolhido

- Os clientes usualmente criam e passam um objeto **ConcreteStrategy** para o contexto e passam a interagir diretamente com o contexto

Integrando os comportamentos de Duck

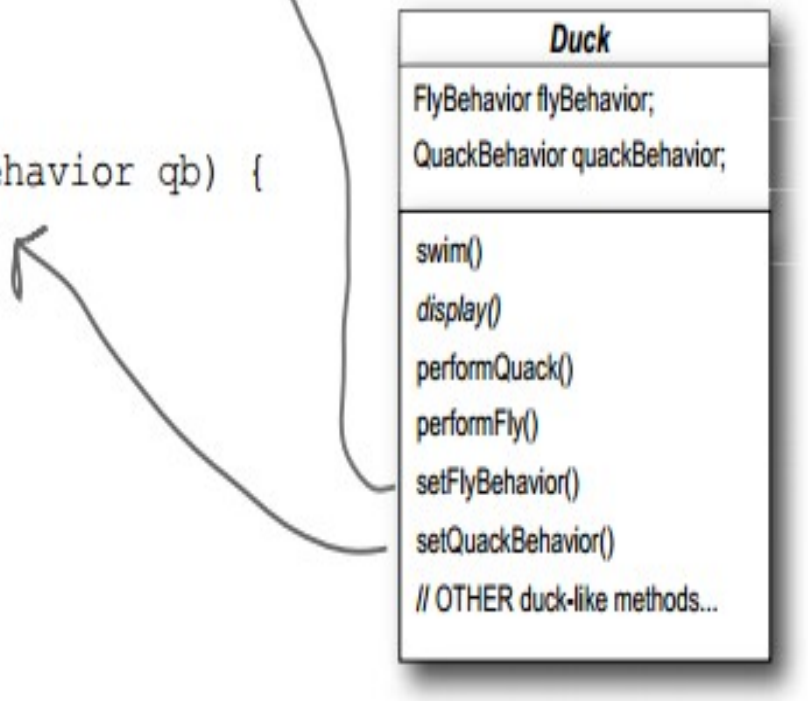


Vamos dar uma olhadinha no código...

Configurando comportamento de forma dinâmica

```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```



Criando um novo tipo Duck

```
public class ModelDuck extends Duck {
    public ModelDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new Quack();
    }

    public void display() {
        System.out.println("I'm a model duck");
    }
}
```

Criando um novo tipo FlyBehavior

```
public class FlyRocketPowered implements FlyBehavior {
    public void fly() {
        System.out.println("I'm flying with a rocket!");
    }
}
```

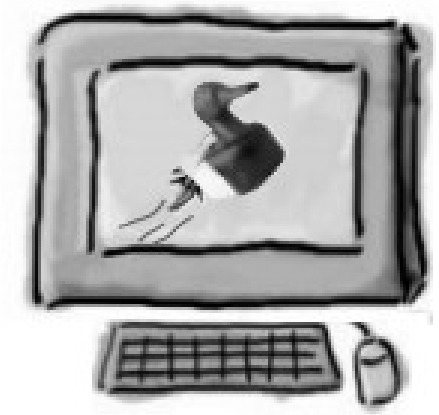
Executando

```
Duck model = new ModelDuck();

model.performFly();

model.setFlyBehavior(new FlyRocketPowered());

model.performFly();
```



Para alterar o comportamento de um pato em tempo de execução,basta chamar o método de configuração do pato para esse comportamento