Padrões de Projeto de Software Orientados a Objetos

Tecnologia em Análise e Desenvolvimento de Sistemas

Paulo Mauricio Gonçalves Júnior

Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco

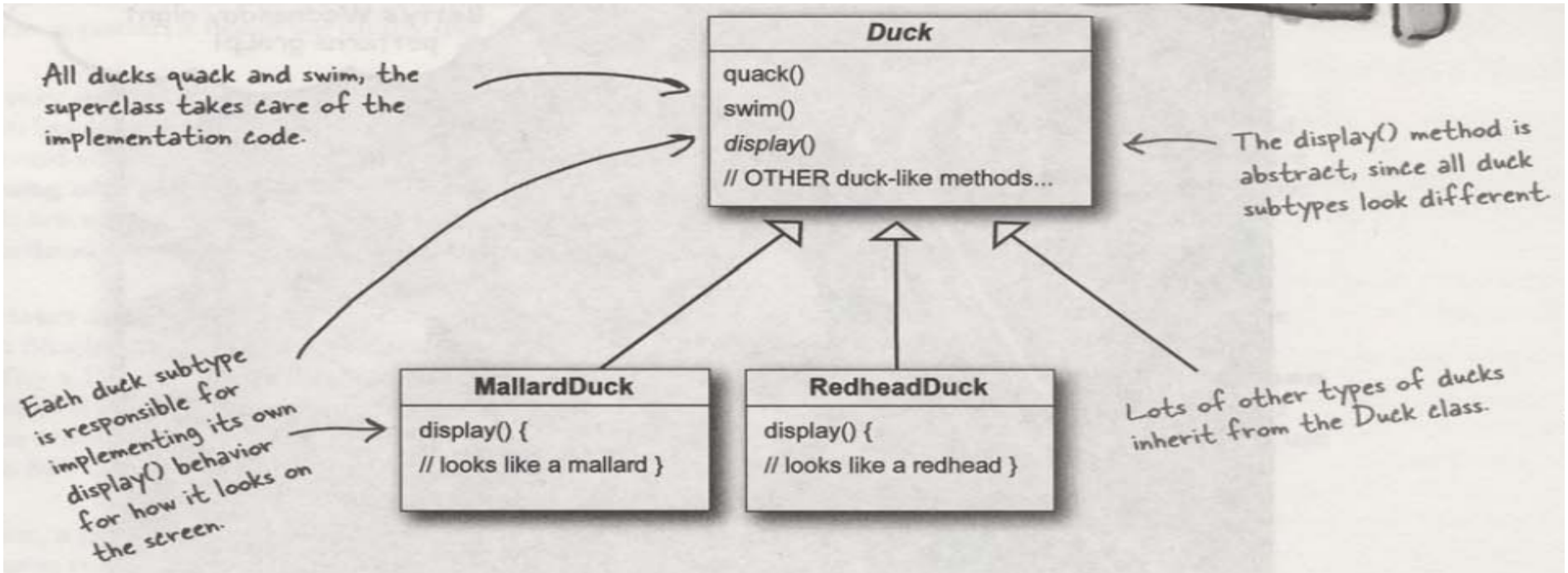
16 de março de 2018

Parte I

Introdução a Padrões de Projeto

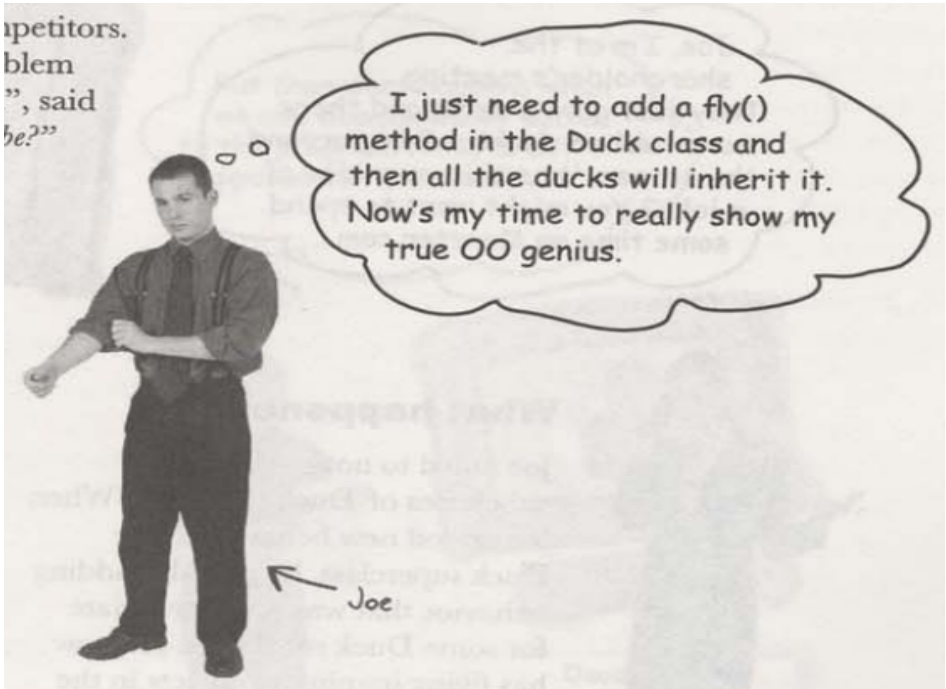
Introdução

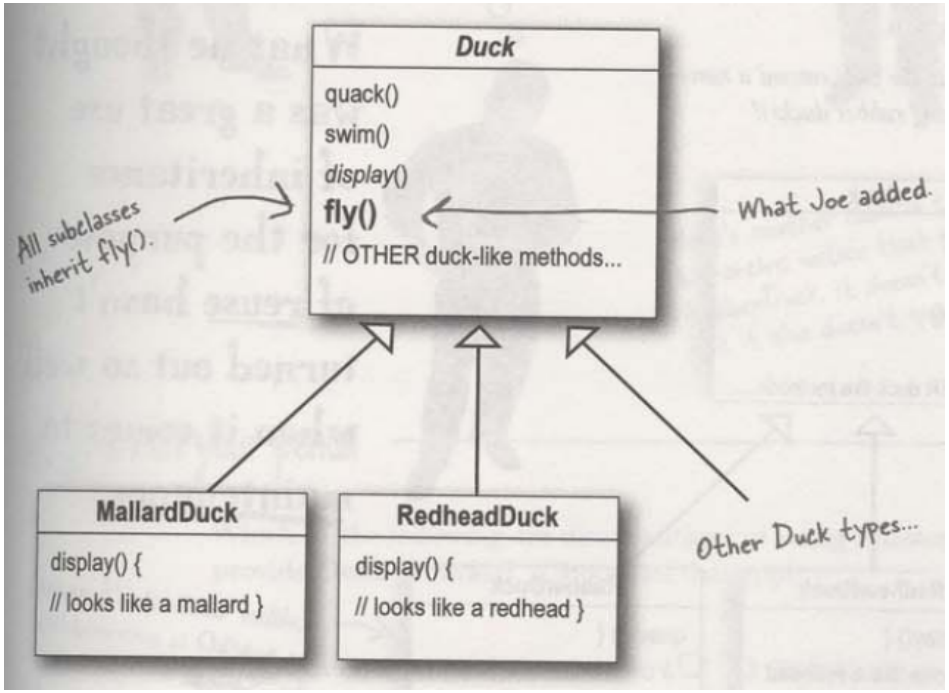
- Imagine que você trabalha em uma empresa que possui um jogo de simulação onde patos estão em uma lagoa, nadando e grasnando.
- Os projetistas inicialmente usaram técnicas padrão de orientação a objetos, criando uma superclasse a partir da qual todos os tipos de patos herdam.



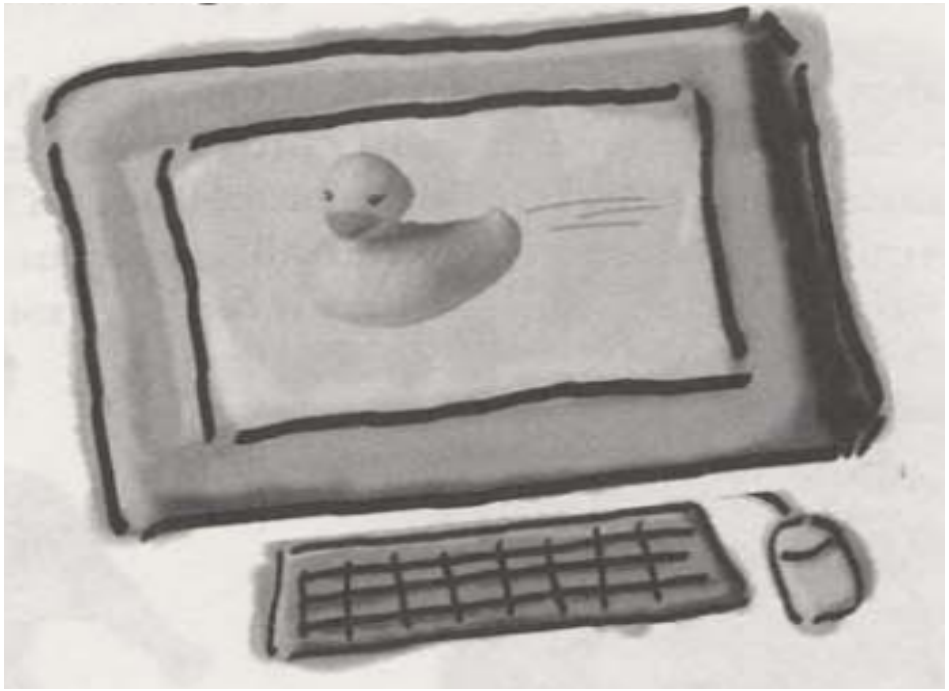
Introdução

- Os executivos decidem que os patos devem poder voar também. Como fazer isso?



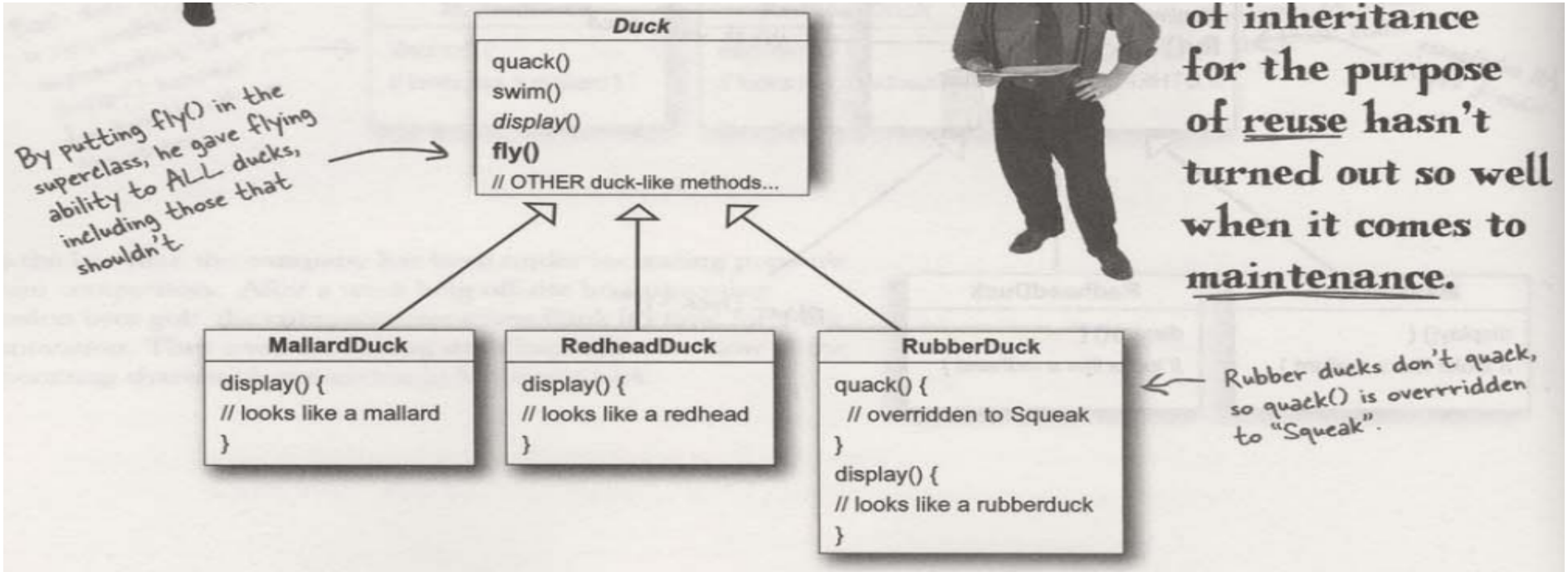


Mas algo deu errado... I

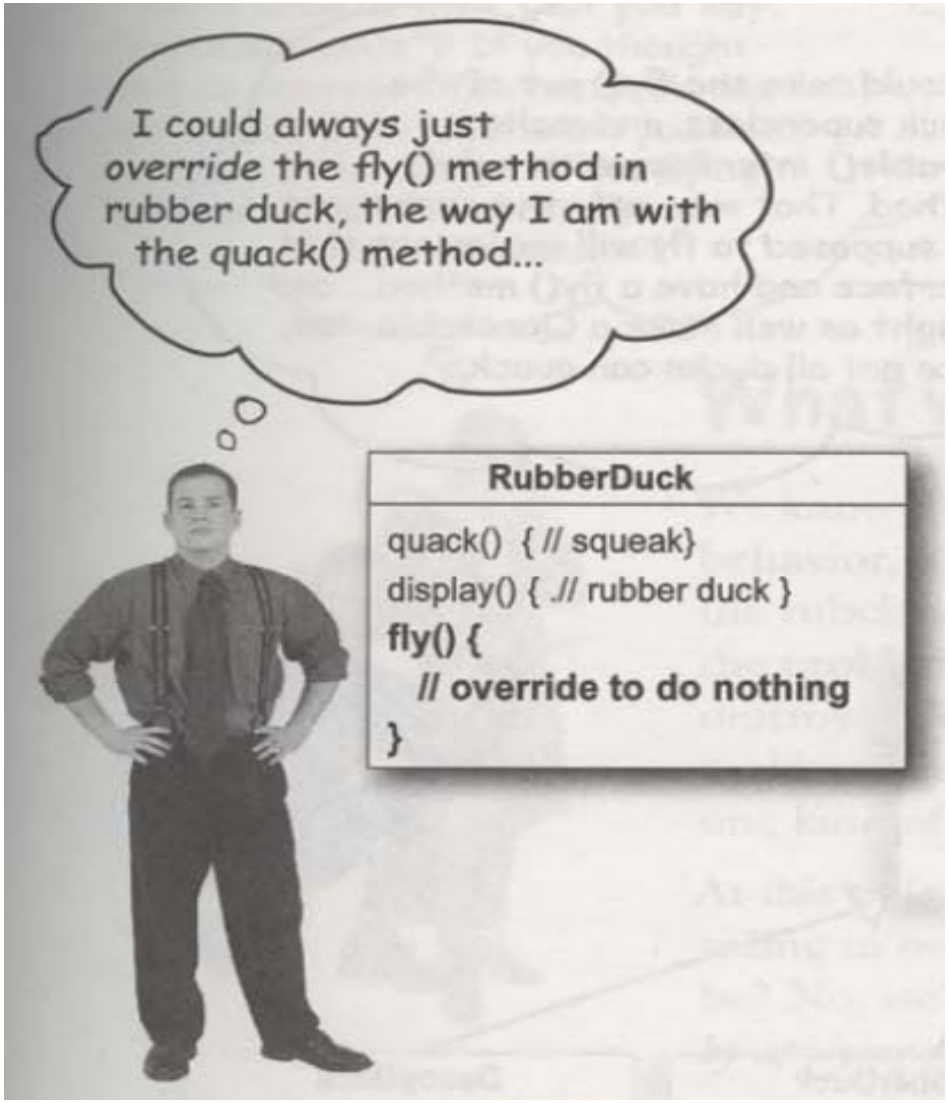


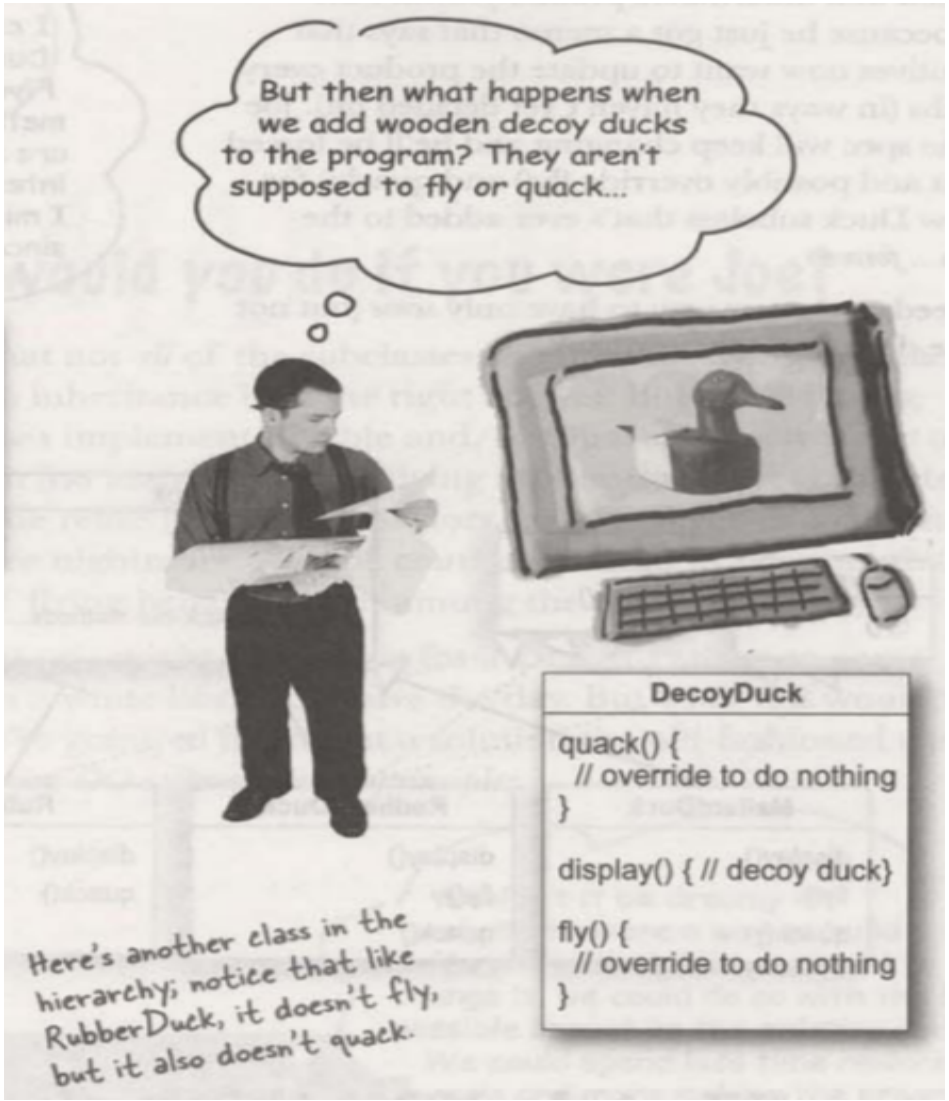
- Não percebemos que nem todos os patos deveriam voar. Ao acrescentar o método na superclasse, estamos adicionando comportamento a subclasses não apropriadas.
- O que parecia um bom uso de herança com o objetivo de reuso não é tão bom com relação à manutenção.

Mas algo deu errado... II



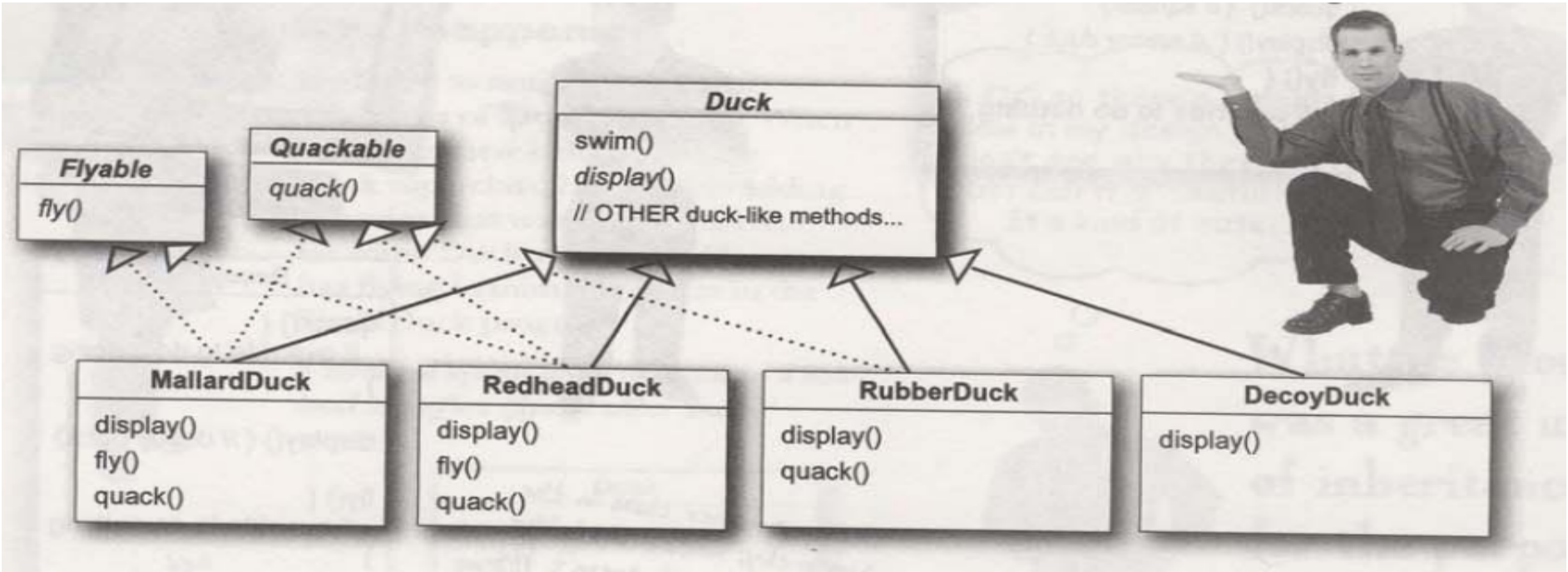
Pensando sobre herança...





E se usarmos interfaces? I

- Os executivos agora querem atualizações no jogo a cada seis meses.
- Podemos criar duas interfaces (Flyable e Quackable) cada uma com um método, e fazer as classes específicas as implementarem caso necessário.



E se usarmos interfaces? II

- Se antes tínhamos que sobrescrever alguns métodos, agora teremos de fazer mudanças em todas as subclasses de Duck.
- Sabemos que nem todas as subclasses possuem o comportamento de voar e grasnar, então herança não é a resposta correta.
- Fazer as subclasses implementarem as interfaces resolvem parte do problema (patos de borracha não voam), ele acaba com o reuso de código para esses comportamentos, criando um problema de manutenção diferente.
- Podem ainda existir mais de um comportamento de voo mesmo nos patos que voam.

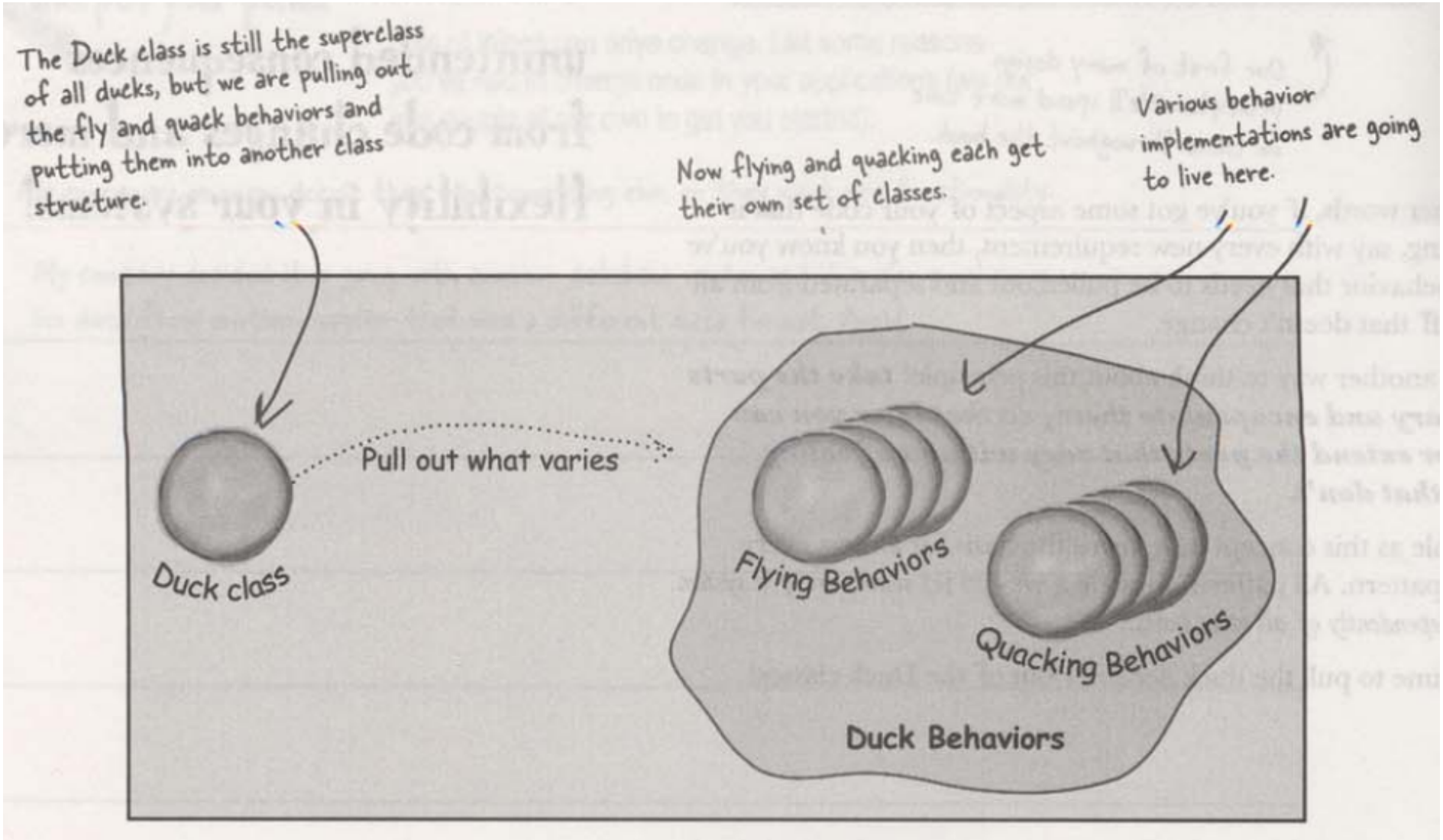
Solução I

Princípio de Projeto

Identifique os aspectos de sua aplicação que mudam e os separe dos que permanecem os mesmos.

- Criar dois conjuntos de classes, um para voar e outro para grasnar. Cada uma delas manterá todas as implementações de seus respectivos comportamentos.

Solução II


The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.
Now flying and quacking each get their own set of classes.
Various behavior implementations are going to live here.
Pull out what varies
Duck class
Flying Behaviors
Quacking Behaviors
Duck Behaviors

Solução III

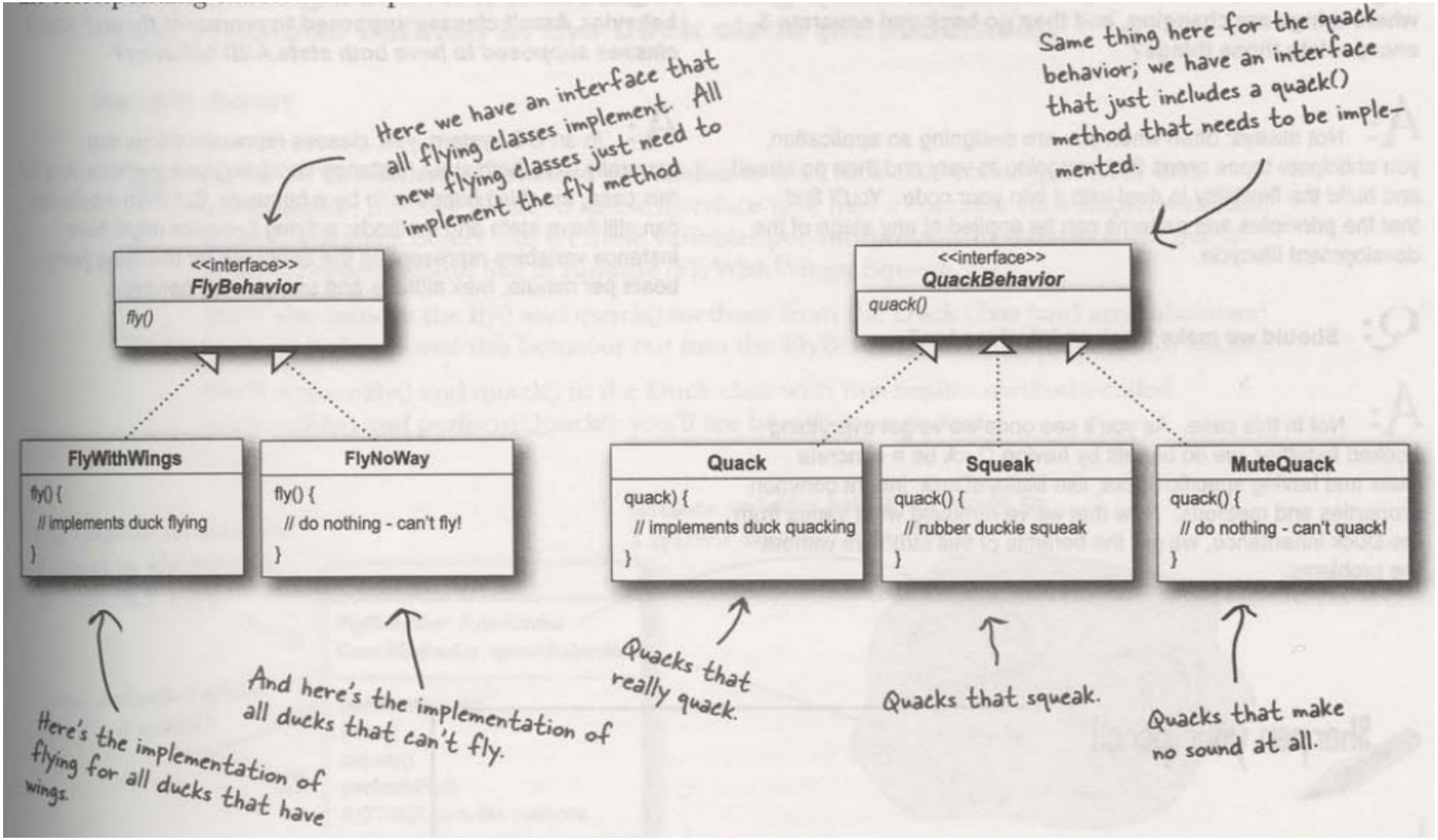
- Desejamos atribuir comportamentos a instâncias de Duck. Ou seja, devemos incluir métodos para setar os comportamentos das classes dinamicamente.

Princípio de Projeto

Programar para uma interface, não uma implementação.

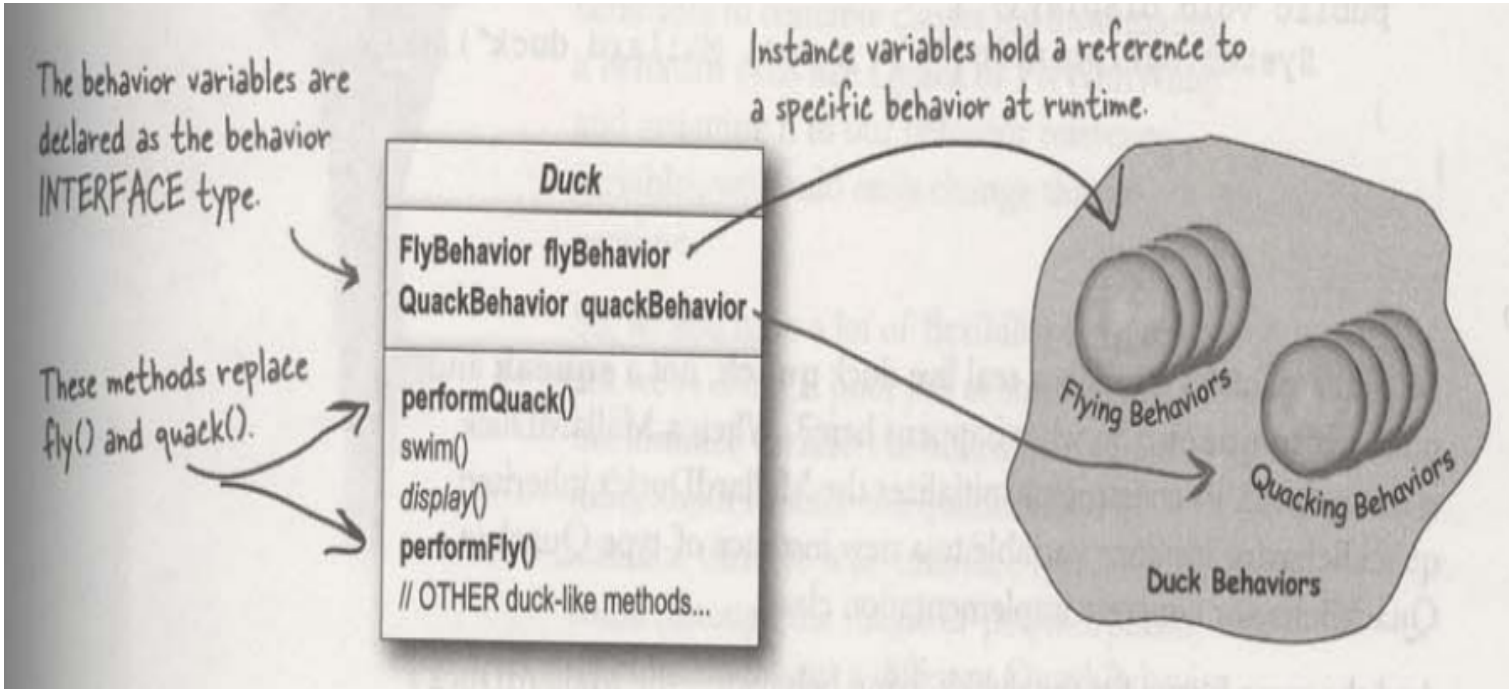
- Usaremos uma interface para representar cada comportamento e cada implementação destes comportamento irá implementar uma das interfaces.

Solução IV


Here we have an interface that all flying classes implement. All new flying classes just need to implement the fly method
Same thing here for the quack behavior; we have an interface that just includes a quack() method that needs to be implemented.
<<interface>>
FlyBehavior
fly()
<<interface>>
QuackBehavior
quack()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Quack
quack) {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
Here's the implementation of flying for all ducks that have wings.
And here's the implementation of all ducks that can't fly.
Quacks that really quack.
Quacks that squeak.
Quacks that make no sound at all.

Solução V

- Primeiramente acrescentaremos dois atributos na classe Duck.



- Depois, implementaremos os métodos para executar os comportamentos.

Solução VI

```
public class Duck {
  QuackBehavior quackBehavior;
  // mais

  public void performQuack() {
    quackBehavior.quack();
  }
}
```

- Criar as instâncias dos atributos.

```
public class MallardDuck extends Duck {
  public MallardDuck() {
    quackBehavior = new Quack();
    flyBehavior = new FlyWithWings();
  }

  public void display() {
    System.out.println("Eu sou um Pato Real!");
  }
}
```

Solução VII

- Podemos modificar os comportamentos dinamicamente

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

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

Princípio de Projeto

Priorize composição sobre herança.

Definição

O padrão Strategy define uma família de algoritmos, encapsula cada um e os torna intercambiáveis. Strategy permite que o algoritmo varie independentemente do cliente que o usa.