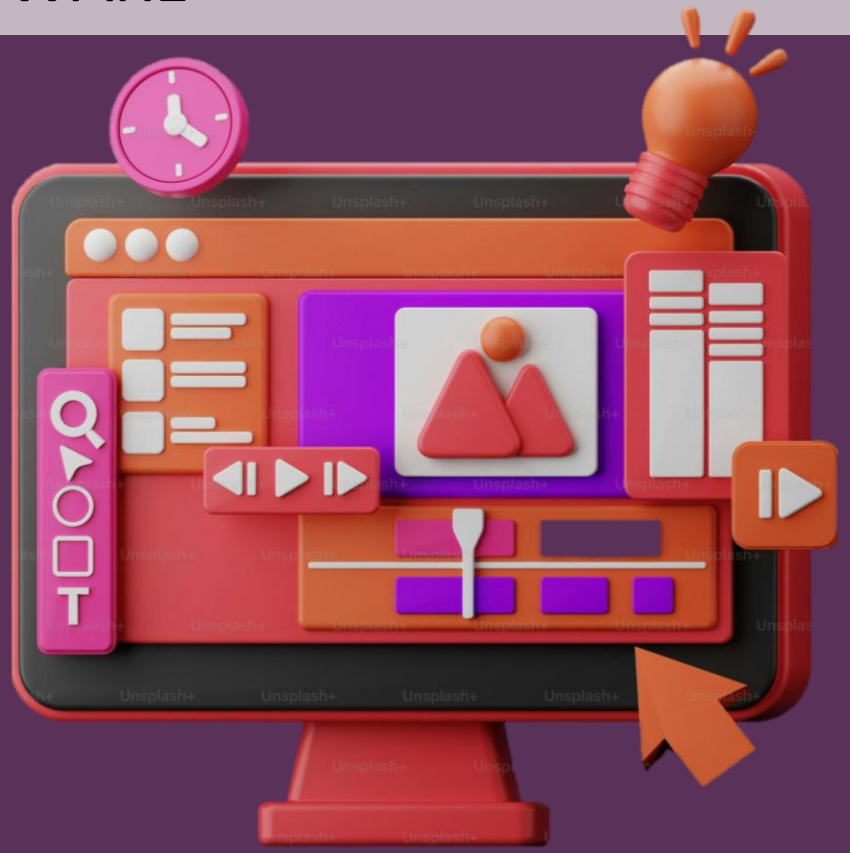
ENGENHARIA DE SOFTWARE

AULA 12 – PADRÕES DE PROJETOS DE SOFTWARE



WALTER TRAVASSOS SARINHO

@PROF.WALTERTRAVASSOS

WALTER.TRAVASSOS@SECTRAS.EDU.BR

Aula de Hoje

- Compreender as motivações para o uso de padrões de projetos.
- Princípios SOLID
- Apresentar alguns exemplos dos princípios SOLID em Python e Java.
- Exemplificar a aplicabilidade de um padrão de projeto.
- Apresentar o padrão de projeto Strategy.
- Implementar em Java.

[S] Princípio da Responsabilidade Única
[O] Princípio do Aberto/Fechado
[L] Princípio da Substituição de Liskov
[I] Princípio da Segregação de Interfaces
[D] Princípio da Inversão de Dependências

Antes de continuar...

Vamos pedir ao ChatGPT para explicar de forma resumida os princípios SOLID. Em seguida solicita pra explicação ser no contexto de algum personagem, universo, filme, série, anime, novela... De sua preferência.

vc consegue explicar utilizando o contexto do anime dragonball?

Claro! Aqui vão os **5 princípios SOLID explicados usando situações do universo de Dragon Ball**, de um jeito simples e divertido:

1. Single Responsibility Principle (SRP)

Princípio da Responsabilidade Única

Cada classe deve ter apenas uma responsabilidade ou motivo para ser modificada.

1. SRP - Exemplo

- Imagine uma classe que lida com **relatórios**. De acordo com o SRP, separamos as **responsabilidades de geração e impressão do relatório.**
- O SRP garante que cada classe tenha uma única função, facilitando a **manutenção**.
- Neste exemplo, **ReportGenerator** apenas gera relatórios, enquanto **ReportPrinter** se encarrega de imprimi-los.

```python
class ReportGenerator:
    def generate_report(self):
        return "Relatório Gerado"

class ReportPrinter:
    def print_report(self, report):
        print(report)
```

2. Open/Closed Principle (OCP)

Princípio Aberto/Fechado

Classes devem estar abertas para extensão, mas fechadas para modificação.

2. OCP - Exemplo

- Suponha uma **classe de cálculo de desconto**. O OCP permite adicionar **novos tipos de desconto** sem modificar a classe original.
- O OCP incentiva a criação de novas funcionalidades por meio de **herança ou composição**, sem alterar a classe original. Aqui, **PercentageDiscount** é uma extensão de **Discount** sem modificar a classe base.

```python
class Discount:
    def apply_discount(self, price):
        return price  # Nenhum desconto por padrão


class PercentageDiscount(Discount):
    def apply_discount(self, price):
        return price * 0.9  # 10% de desconto
```

3. Liskov Substitution Principle (LSP)

Princípio da Substituição de Liskov

Objetos de uma classe derivada devem poder substituir objetos da classe base sem alterar o comportamento esperado.

Barbara Liskov

- Esse princípio foi descrito pela pesquisadora Barbara Liskov, em seu artigo de 1988, em que ela explica que: **antes de optar pela herança**, precisamos pensar nas **pré-condições** e **pós-condições** da sua classe.
- Mesmo a herança sendo um mecanismo poderoso, ela **deve ser utilizada de forma contextualizada** e **moderada**, **evitando** os casos de classes serem estendidas **apenas** por possuírem algo em comum.

Princípio da Substituição de Liskov (LSP)

Diz que “Os subtipos devem ser substituíveis pelos seus tipos base”, e que as classes/tipos base podem ser substituídas por qualquer uma das suas subclasses, ponderando sobre os cuidados para usar a herança no seu projeto de software.

Ou seja...

1. Ao invés de usar herança, **ponderar se podemos utilizar a composição**.
2. Ao invés de estender classes, **colocar essas possíveis subclasses como uma variável de instância da classe que seria candidata a ser classe pai**.

Para ficar mais claro...

- Vamos analisar o exemplo das classes **ContaCorrenteComum** e **ContaSalario**.
- A **ContaCorrenteComum** representa, dentro do nosso contexto simplificado, uma conta de banco qualquer.
- Tem os métodos deposita(), getSaldo(), saca() e rende().

```java
public class ContaCorrenteComum {

    protected double saldo;

    public ContaCorrenteComum() {
        this.saldo = 0;
    }

    public void deposita(double valor) {
        this.saldo += valor;
    }

    public void saca(double valor) {
        if(valor <= this.saldo) {
            this.saldo -= valor;
        }else{
            throw new IllegalArgumentException("Saldo insuficiente.");
        }
    }

    public double getSaldo() {
        return saldo;
    }

    public void rende()     {
        this.saldo*= 0.02;
    }
}
```

```java
public class ContaSalario extends ContaCorrenteComum {

        public void rende() {
        throw new Exception("Essa conta não possui rendimento");
   }

}
```

ContaSalario

- Uma **ContaSalario** é idêntica a classe **ContaCorrenteComum**, exceto pelo método **rende()**.
- Afinal, uma conta salário não tem rendimento, é só para recebimento.

Dessa forma, a solução apresentada para o problema consiste em estender a classe ContaCorrenteComum e fazer com que o método rende() da classe filha lance uma exceção.

Mas... isso dá certo?

Essa não é uma boa ideia!

- Se fossemos tentar acessar o método **rende()** de todas as contas do Banco em um **loop**, por exemplo, e uma delas é uma **ContaSalario**, ai pronto.
- Coisou!
- A aplicação não funciona, porque para qualquer conta salário, uma exceção é lançada nesse método.

```java
public class ContaSalario extends ContaCorrenteComum {

        public void rende() {
        throw new Exception("Essa conta não possui rendimento");
    }

}
```

```java
import java.util.ArrayList;
import java.util.List;

public class Banco {

    public static void main(String[] args) {

        List<ContaCorrenteComum> listaDeContas = new ArrayList<>();
        listaDeContas.add(new ContaCorrenteComum());
        listaDeContas.add(new ContaSalario());

        for (ContaCorrenteComum conta : listaDeContas) {
            conta.rende();

            System.out.println("Novo Saldo:");
            System.out.println(conta.getSaldo());
        }
    }

}
```

A solução é aplicar o princípio LSP!

Aplicando a composição

- Devemos **refatorar** e optar pelo uso da **composição**.
- Vamos criar um **GerenciadorDeConta** e essa classe é que fará o controle das movimentações das contas.
- As classes **ContaCorrenteComum** e **ContaSalario** passam a ter um **GerenciadorDeContas**, retirando a relação de pai-filho desnecessária entre elas.

```java
package io.github.mariazevedo88.solid.lsp;

public class GerenciadorDeContas {

    private double saldo;

    public void deposita(double valor) {
        this.saldo += valor;
    }

    public void saca(double valor) {
        if(valor <= this.saldo) {
                this.saldo -= valor;
        }else{
                throw new IllegalArgumentException("Saldo insuficiente.");
        }
    }

    public double getSaldo() {
        return saldo;
    }

    public void rende(double taxa){
        this.saldo = this.saldo + (this.saldo*taxa);
    }
}
```

```java
public class ContaCorrenteComum {

    private GerenciadorDeContas gerenciador;

    public ContaCorrenteComum() {
        this.gerenciador = new GerenciadorDeContas();
    }

    public void deposita(double valor) {
        this.gerenciador.deposita(valor);
    }

    public void saca(double valor) {
        this.gerenciador.saca(valor);
    }

    public double getSaldo() {
        return this.gerenciador.getSaldo();
    }

    public void rende() {
        this.gerenciador.rende(0.02);
    }

    @Override
    public String toString() {
        return "Saldo conta corrente-> " + this.getSaldo();
    }
}
```

ContaSalario

Repare que na versão refatorada da classe **ContaSalario**, nem precisamos implementar o método **rende()**, só utilizamos o gerenciador em comportamentos que a classe possui.

```java
public class ContaSalario {

    private GerenciadorDeContas gerenciador;

    public ContaSalario() {
        this.gerenciador = new GerenciadorDeContas();
    }

    public void deposita(double valor) {
        this.gerenciador.deposita(valor);
    }

    public void saca(double valor) {
        this.gerenciador.saca(valor);
    }

    public double getSaldo() {
        return this.gerenciador.getSaldo();
    }

    @Override
    public String toString() {
        return "Saldo conta salario-> " + this.getSaldo();
    }
}
```

4. Interface Segregation Principle (ISP)

Princípio da Segregação de Interface

Classes não devem ser forçadas a implementar interfaces que não utilizam.

4. ISP - Exemplo

- Imagine que temos uma interface **AnimalActions** com muitos métodos, quebrá-la em interfaces menores torna o código mais flexível.
- O ISP divide interfaces grandes em menores e mais específicas.
- Aqui, **Bird** implementa apenas métodos de **BirdActions**, enquanto **Fish** implementa **FishActions**.

```python
class BirdActions:
    def fly(self):
        pass

class FishActions:
    def swim(self):
        pass

class Bird(BirdActions):
    def fly(self):
        print("O pássaro está voando")

class Fish(FishActions):
    def swim(self):
        print("O peixe está nadando")
```

5. Dependency Inversion Principle (DIP)

Princípio da Inversão de Dependência

Módulos de alto nível não devem depender de módulos de baixo nível, ambos devem depender de abstrações.

Princípio da Inversão de Dependências

Diz que devemos “depender de abstrações e não de classes concretas”. Uncle Bob quebra a definição desse princípio em dois subitens:

1. “Módulos de alto nível não devem depender de módulos de baixo nível.”
2. “As abstrações não devem depender de detalhes. Os detalhes devem depender das abstrações.”

E isso se dá porque **abstrações mudam menos** e facilitam a mudança de comportamento e as futuras evoluções do código.

5. DIP - Exemplo

- Em vez de **Pedido** depender diretamente de **EmailService**, cria-se uma interface **MensagemService** para abstrair a dependência.
- O DIP promove o uso de abstrações, permitindo que **Pedido** dependa de **MensagemService**, e não de uma implementação específica como **EmailService**.

```python
class MensagemService:
    def enviar(self, mensagem):
        pass

class EmailService(MensagemService):
    def enviar(self, mensagem):
        print(f"Enviando email: {mensagem}")

class Pedido:
    def __init__(self, mensagem_service):
        self.mensagem_service = mensagem_service

    def confirmar_pedido(self):
        self.mensagem_service.enviar("Pedido confirmado!")
```

Exemplo com Java

Spotify & DIP

✕ Sem DIP (acoplamento forte)

```
class SpotifyService {
    public void playMusic(String music) {
        System.out.println("Tocando " + music + " no Spotify...");
    }
}

class MusicPlayer {
    private SpotifyService service = new SpotifyService(); // DEPENDÊNCIA DIRETA

    public void play(String music) {
        service.playMusic(music);
    }
}
```

Problemas

- MusicPlayer depende diretamente de SpotifyService.
- Se quiser mudar para Apple Music, tem que alterar o código do player.

Spotify & DIP

✔Com DIP — Agora seguindo o 5º princípio SOLID

1. Criamos uma abstração (interface):

```
interface MusicService {
    void playMusic(String music);
}
```

2. Implementação concreta do Spotify:

```
class SpotifyService implements MusicService {
    @Override
    public void playMusic(String music) {
        System.out.println("Tocando " + music + " no Spotify...");
    }
}
```

3. O player depende da abstração, não da implementação:

```
class MusicPlayer {
    private MusicService service;  // DEPENDÊNCIA DA ABSTRAÇÃO

    public MusicPlayer(MusicService service) {
        this.service = service; // Injeção de dependência
    }

    public void play(String music) {
        service.playMusic(music);
    }
}
```

Spotify & DIP

✔Com DIP — Agora seguindo o 5º princípio SOLID

- 4. Uso:

```
public class Main {
    public static void main(String[] args) {
        MusicService spotify = new SpotifyService();
        MusicPlayer player = new MusicPlayer(spotify);

        player.play("Blinding Lights");
    }
}
```

Trocar o serviço sem tocar no MusicPlayer

```
class AppleMusicService implements MusicService {
    @Override
    public void playMusic(String music) {
        System.out.println("Tocando " + music + " no Apple Music...");
    }
}

// Uso:
MusicPlayer player = new MusicPlayer(new AppleMusicService());
player.play("Halo");
```

O MusicPlayer não muda uma única linha, isso é DIP.

Code-smells <💩>

- Se conseguirmos seguir passo a passo todas esses princípios, teremos um código fácil de se manter, testar, reaproveitar, estender e evoluir.
- Comece fazendo mudanças em **pontos específicos**, evitando a propagação de problemas e trechos “mal-cheirosos” (*code-smells*).
- Assim, você começa a treinar seu cérebro a pensar de forma mais madura quando estiver diante de situações de desenvolvimento mais complexas do seu dia-a-dia.

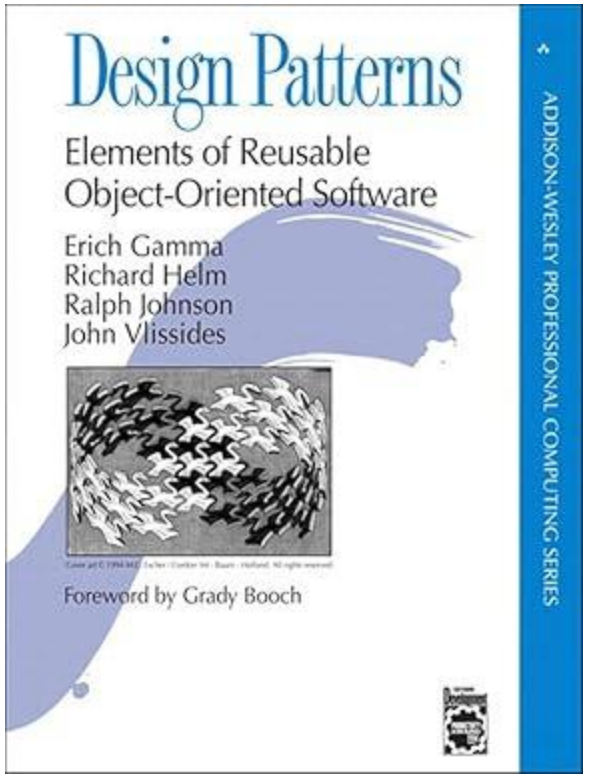
Padrões de Projeto

O que são Padrões de Projeto? Design Patterns

- São **soluções reutilizáveis** para **problemas recorrentes** no desenvolvimento de software.
- Eles representam práticas testadas e aprovadas para resolver problemas comuns de design, ajudando a criar sistemas mais **robustos**, **flexíveis** e **manuteníveis**.
- Cada **padrão descreve um problema específico** no design de software, o contexto em que esse problema ocorre e a solução proposta, permitindo aos desenvolvedores **aplicar essas soluções em diferentes situações**.

Gang of Four

- GoF (Gang of Four) refere-se aos quatro autores do livro "*Design Patterns: Elements of Reusable Object-Oriented Software*", que se tornou uma obra fundamental para desenvolvedores.
- Os autores Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, conhecidos coletivamente como o "Gangue dos Quatro", foram pioneiros ao documentar **23 padrões de design de software**, consolidando boas práticas e servindo como referência para arquitetos e desenvolvedores de software ao redor do mundo.



Os padrões de projetos são classificados geralmente em 3 categorias:

- **Criacionais** – Focados em como objetos são criados, eles buscam uma maior flexibilidade e reutilização do código na **criação de instâncias**.
- **Estruturais** – Ajudam a compor classes e objetos para formar **estruturas maiores e mais complexas**, ou seja, definem a **composição** de classes e objetos.
- **Comportamentais** – Focados na **comunicação entre objetos**, facilitando a flexibilidade e a extensão do código, ou seja, definem a **interação** entre classes e objetos.

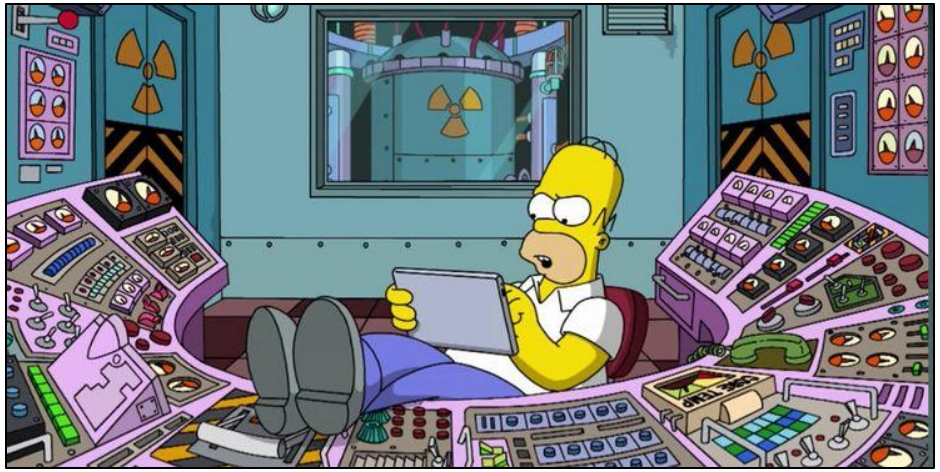
23 Padrões de Projeto segundo GoF

- **5 padrões criacionais**: Factory Method, Abstract Factory, Builder, Prototype e Singleton.
- **7 padrões estruturais são**: Decorator, Proxy, FlyWeigth, Facade, Composite, Bridge e Adapter.
- **11 padrões comportamentais**: Template Method, Visitor, Command, Strategy, Chair of Responsability, Iterator, Mediator, Memento, Interpreter, State e Observer.

Como vou saber quando devo aplicar um Padrão de Projeto?

Exemplo do Software Patinho na Lagoa

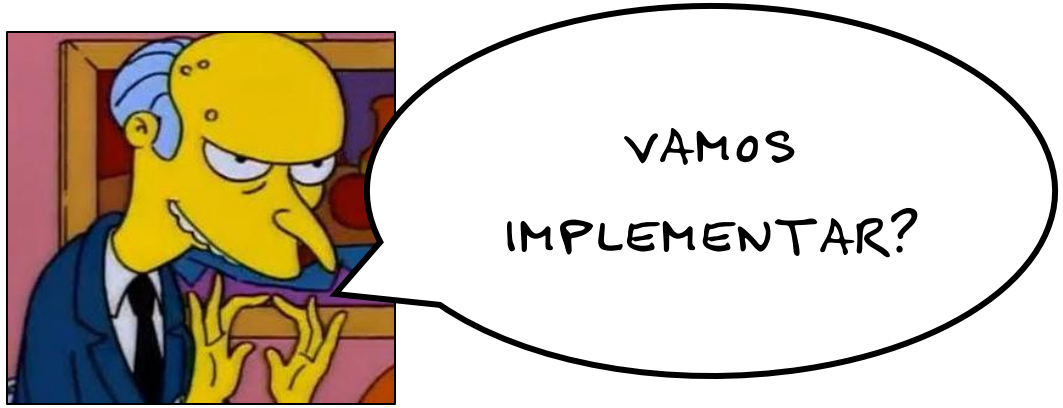
Introdução

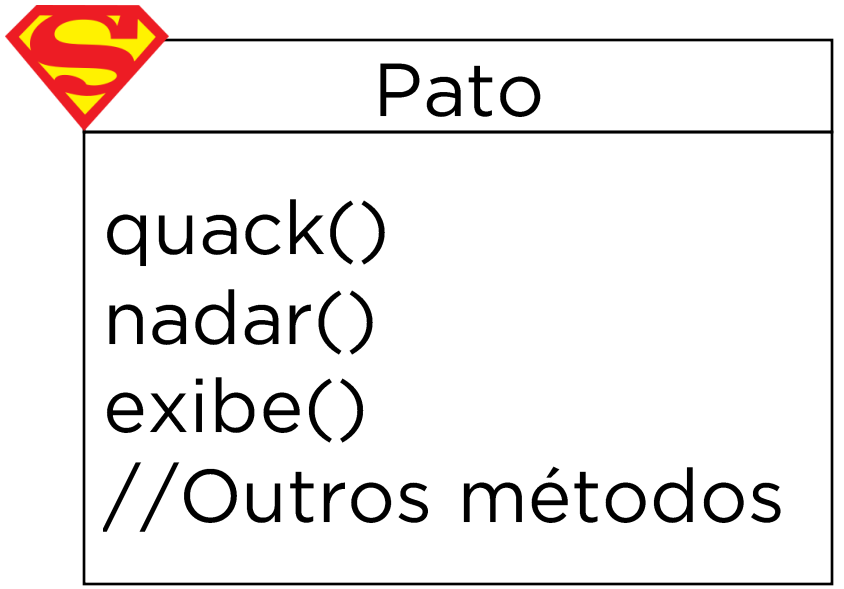


- Homer trabalha numa startup proprietária de um software chamado **Patinho na Lagoa**.
- O software mostra uma grande variedade de espécies de patos nadando (método **nadar**) e produzindo sons (grasnar – sendo representado pelo método **quack**).
- Cada pato também possui um método chamado **exibe** que mostra o pato de cada classe de uma forma diferente na tela do usuário.

Diagrama de Classe do Patinho na Lagoa

Os designers iniciais do sistema usaram técnicas OO padrão e criaram uma superclasse **Pato** que é herdada por todos os outros tipos de pato.





Patinho na Lagoa

Pato

quack()
nadar()
exibe()
//Outros métodos

PatoReal

exibe()

PatoDeBicoVermelho

exibe()

Tudo estava indo (relativamente) bem, até chegarem algumas solicitações de mudanças

- Em 2019 a startup esteve sob uma pressão crescente dos concorrentes (pandemia se alastrando) e, após uma sessão de brainstorming, os donos acharam que estava na hora de **implementar uma grande inovação** na forma que o software funciona.
- Essa mudança é algo realmente importante para ser mostrado na próxima reunião com os **investidores**.
- Assim, foi decidido que...

HOMER... VOCÊ É O CARA DA ORIENTAÇÃO A OBJETOS. IMPLEMENTE ESSA **FEATURE**. TENHO CERTEZA QUE VAI SER SUPER TRANQUILO!

OS PATOS VÃO VOAR!

VAI SER MOLEZA!

Então Homer teve a ideia de criar um método voar() na superclasse Pato para resolver o problema.

Ao fazer isso, todas as classes filhas de Pato herdam o método voar.

Resultado

Expectativa

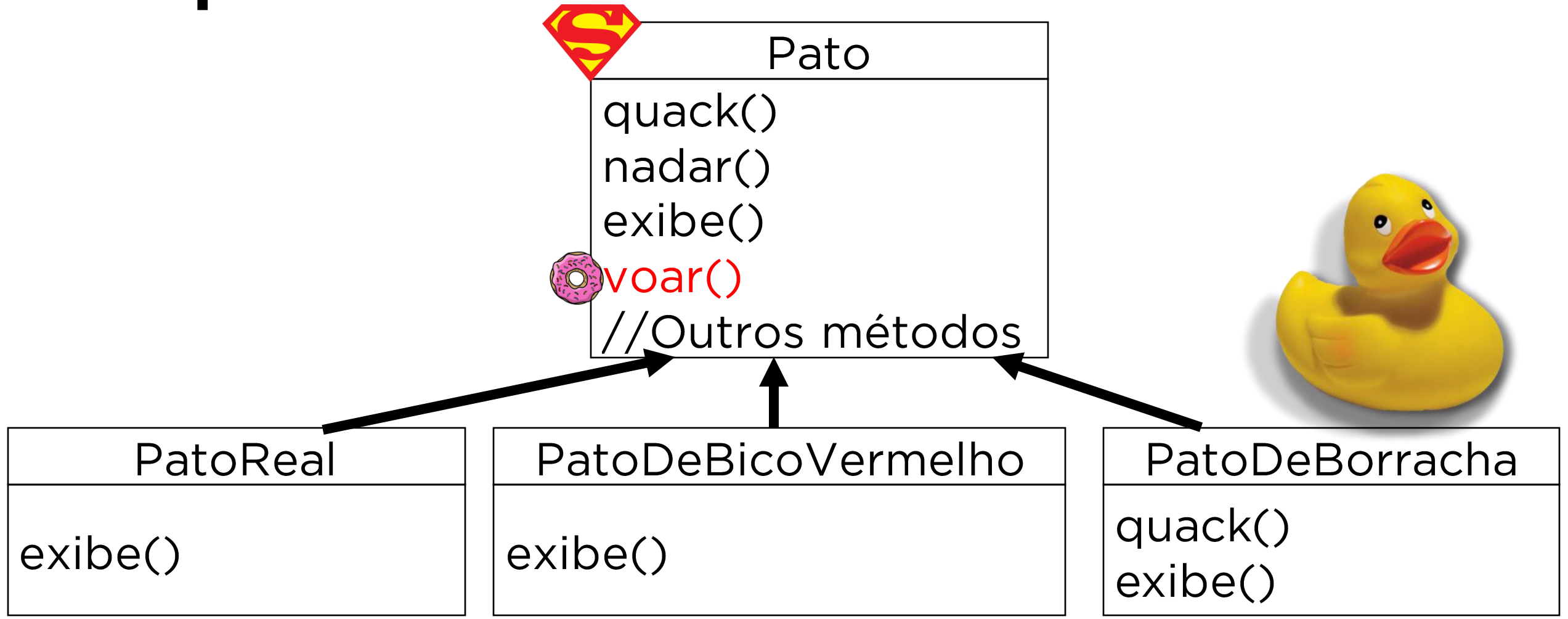
O que os donos realmente queriam.

Realidade

O que aconteceu!?

Tem pato de borracha voando!

O que Homer fez?



Patos de borracha não grasnam, mas podem cuspir água.
Assim, iremos sobrescrever o comportamento de **quack** na classe Pato de Borracha.

No fim,
não foi um uso tão bom da herança!

- Homer não percebeu que nem todas as subclasses de um Pato deveriam voar.
- Ao adicionar um novo comportamento na superclasse Pato, também estava adicionando em classes onde esse comportamento não era apropriado.
- Agora... Os **objetos inanimados** estavam voando no software Patinho na Lagoa.

- Parem a implementação. Vamos discutir algumas ideias.

Homer começa a pensar em como resolver esse problema...

Possibilidade 1

Mas o que acontece
quando adicionarmos
Patos de Madeira ao software?
Eles não devem voar nem grasnar.

Possibilidade 2


UH-HULL!
VAI FICAR LINDO!

Patinho na Lagoa

Pato

nadar()
exibe()
//Outros métodos

Flyable

voar()

Quackable

quack()

PatoDeMadeira

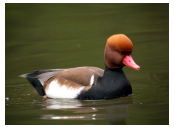
exibe()

PatoReal

exibe()

PatoDeBicoVermelho

exibe()



PatoDeBorracha

exibe()


O QUE VOCÊ
ACHOU DO MEU
PROJETO?

Tem muito código duplicado!

- Se a gente estava achando ruim ter que substituir alguns métodos, como será que vai ser quando precisar fazer uma pequena alteração no comportamento de voo?
- E se tivéssemos 48 subclasses de Pato?
- Iriamos ter que atualizar todas, afinal, (originalmente) interfaces não carregam código.

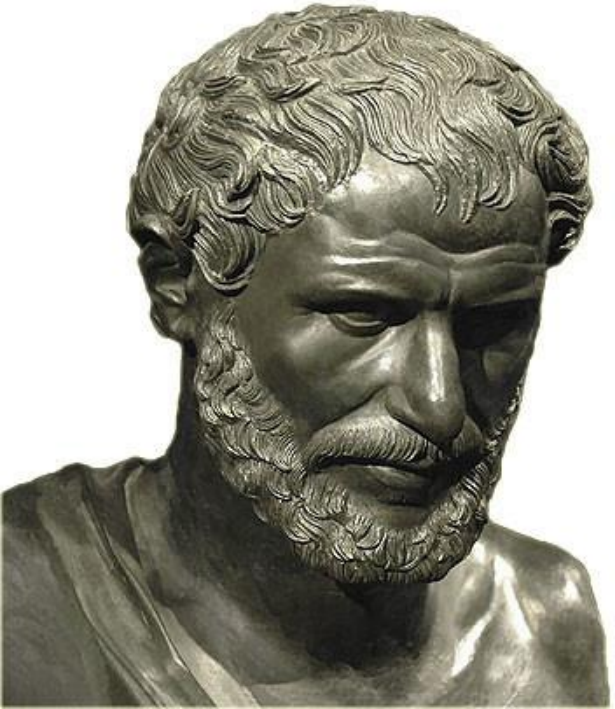
Vamos aplicar alguns princípios de Design OO

Não seria maravilhoso se...

- Existisse uma maneira de criar um software de modo que **quando precisássemos alterá-lo, pudéssemos fazer isso com o menor impacto possível no código existente?**
- Poderíamos **perder menos tempo retrabalhando o código** e ter **mais tempo para permitir que o software faça coisas mais legais...**

A única constante é a mudança.

Heráclito de Éfeso



Recapitulando o problema

- **Herança não funcionou muito bem** já que o comportamento do pato fica mudando entre as subclasses e não é apropriado que todas as subclasses tenham o mesmo comportamento.
- As **interfaces** Flyable e Quackable **pareciam promissoras**, no entanto as interfaces não possuem código de implementação e assim **não dá pra reutilizar código**. Isso significa que se houver alteração, precisará ser realizado em todas as classes que implementaram a interface.

Por sorte, existe um **princípio de projeto** para esta situação...

1º 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 código.

As alterações são mais fáceis de serem realizadas nesses métodos encapsulados.

Além disso, podemos estender as partes que variam sem afetar as que não variam.

Essa é a Base dos Padrões de Projetos

- Esse conceito simples **forma a base de quase todos os padrões de projetos.**
- Todos os padrões fornecem uma maneira de deixar que alguma parte do sistema varie independentemente de todas as outras partes.

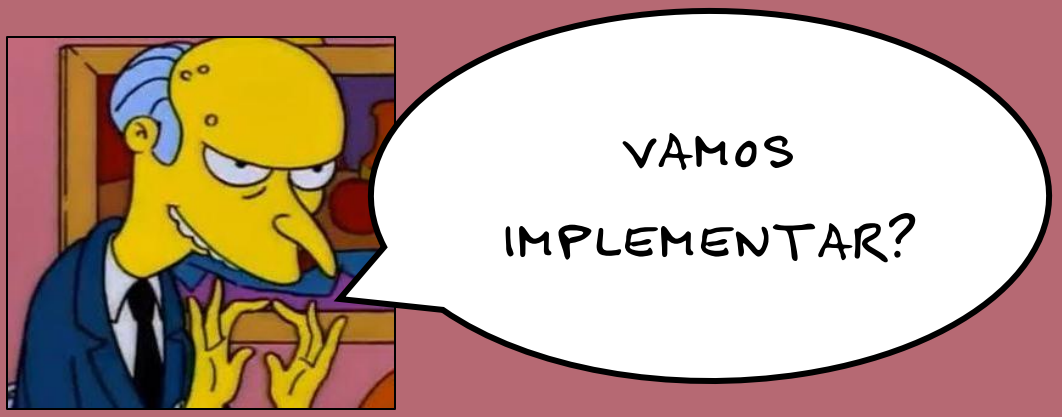

Hora de separar
o que muda do que
fica igual no
Patinho na Lagoa

Para iniciar essa separação...

Vamos criar 2 **conjuntos** de classes separadas da superclasse Pato:

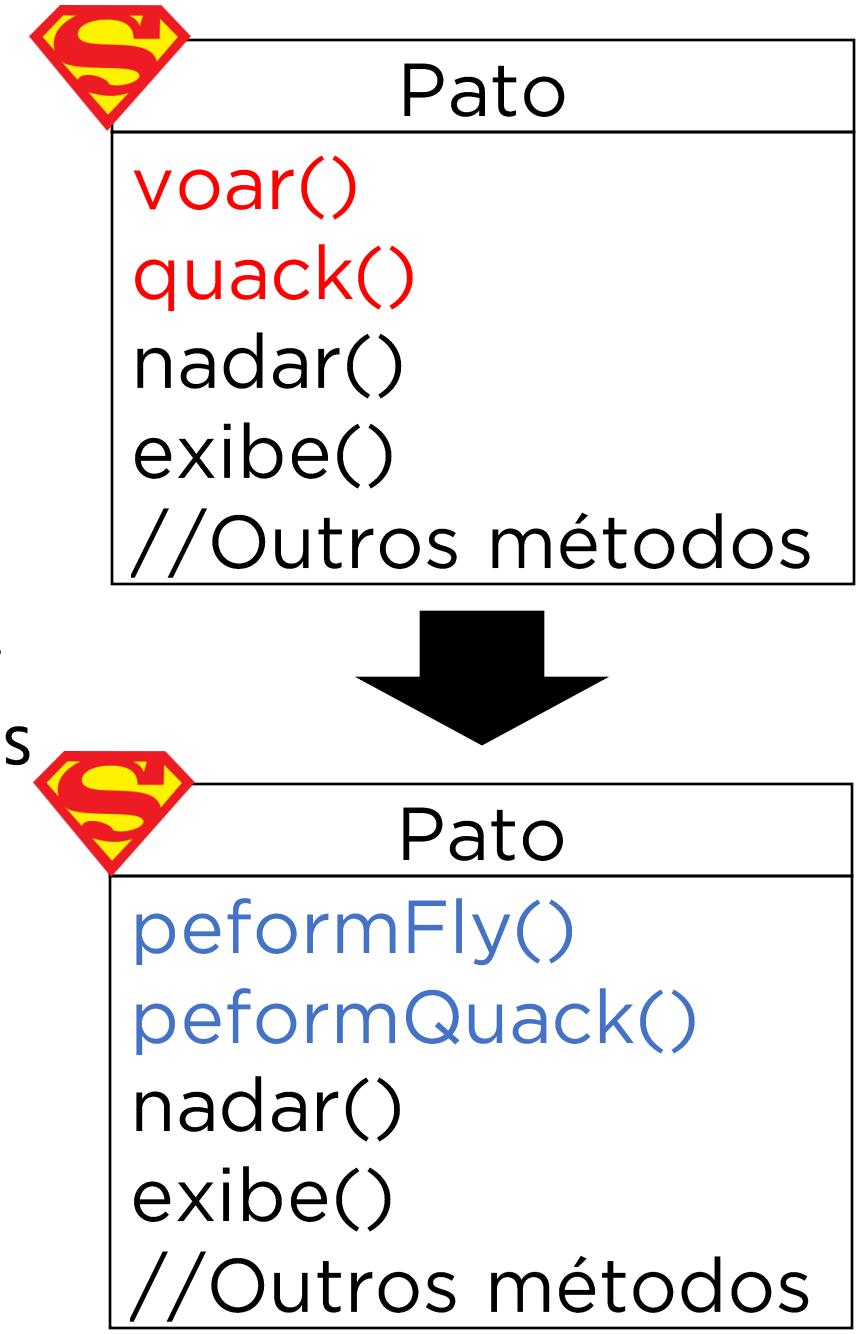
1. uma para voar
2. outra para grasnar (quack).

Cada conjunto de classe vai conter todas as **implementações possíveis de seus comportamentos** (métodos) diferentes.

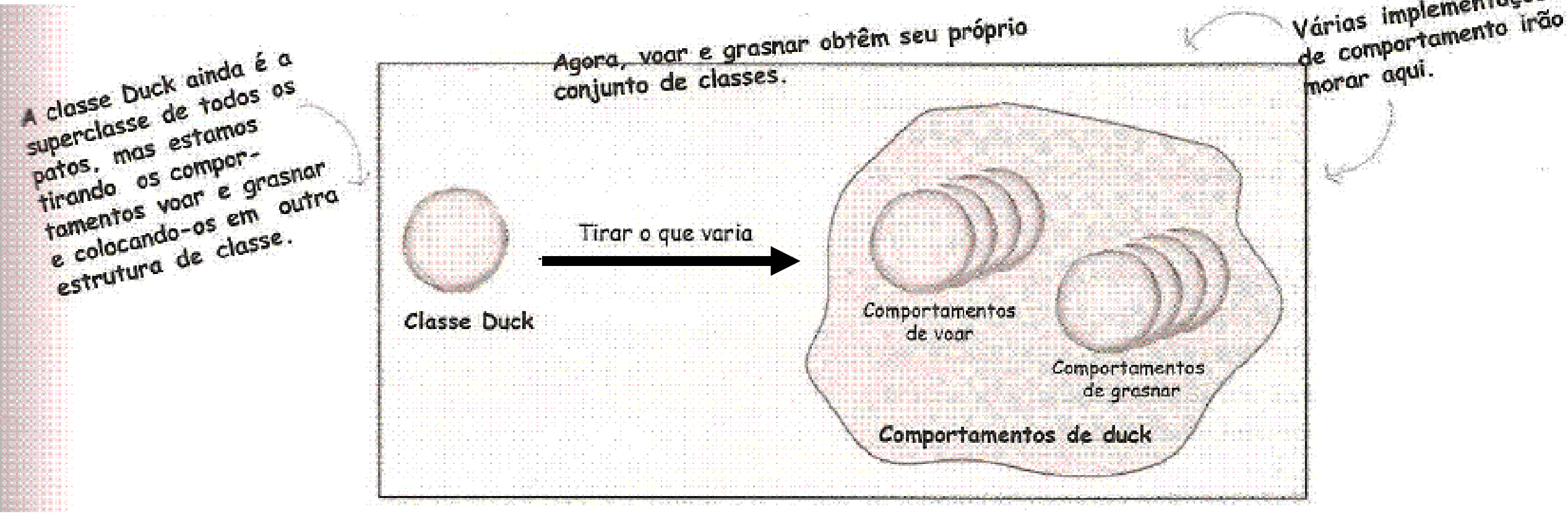


Alterando a classe Pato

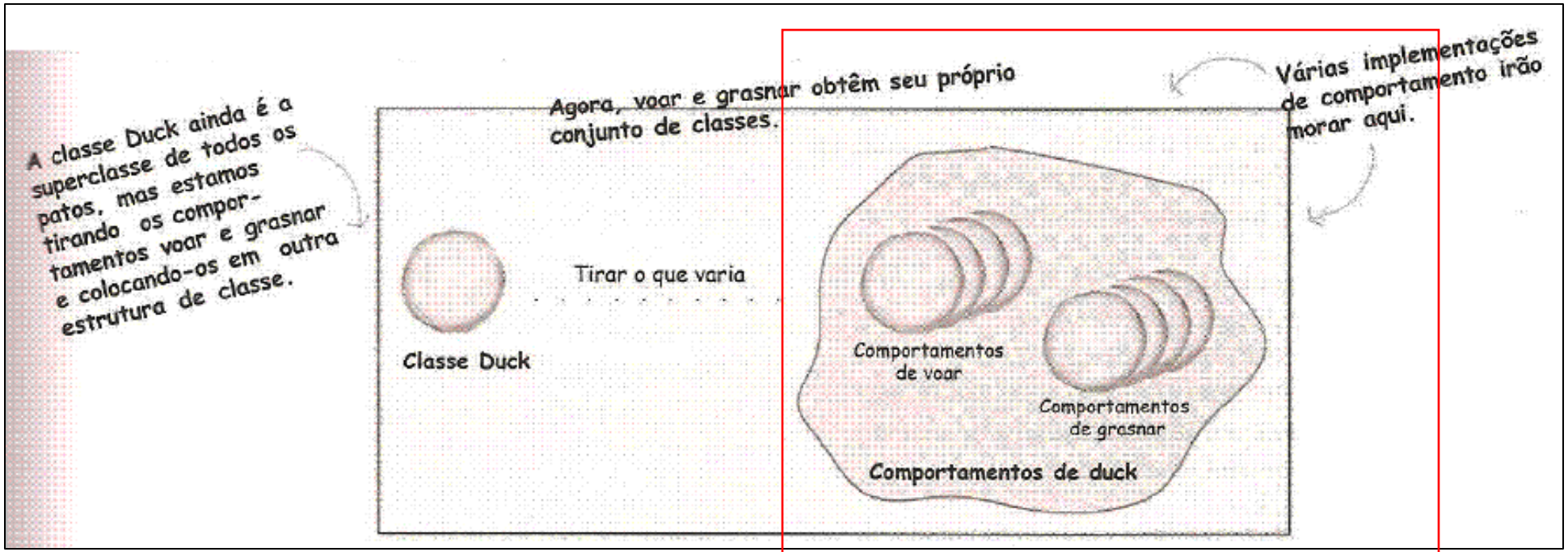
- Os métodos **voar()** e **quack()** estavam pertencendo a super classe Pato. Assim, vamos iniciar as mudanças tirando esses métodos da classe Pato (e de todas as classes filhas de Pato).
- Em seguida criamos um novo conjunto de classes que irão representar esses **comportamentos de voar e grasnar**.
- Daqui a pouco explicaremos os novos métodos: **peformFly()** e **peformQuack()**.



Vamos criar uma nova "família de classes"



Vocês imaginam como iremos desenvolver o conjunto de classes que vão implementam os comportamentos de voar e grasnar?



Temos que manter as coisas flexíveis

- Foi a inflexibilidade nos comportamentos da classe Pato que nos criou esse problema.
- Já sabemos que queremos atribuir comportamentos as instâncias de Pato.
- Por exemplo, poderíamos querer criar uma nova instância de Pato-Real e querer que ele tenha um comportamento específico de voo.

Além dessa possibilidade...

- Seria possível garantir que possamos alterar um comportamento de um pato de **forma dinâmica**?
- Ou seja, alterar o comportamento de voo de Pato-Real em tempo de execução?

2º Princípio de Projeto

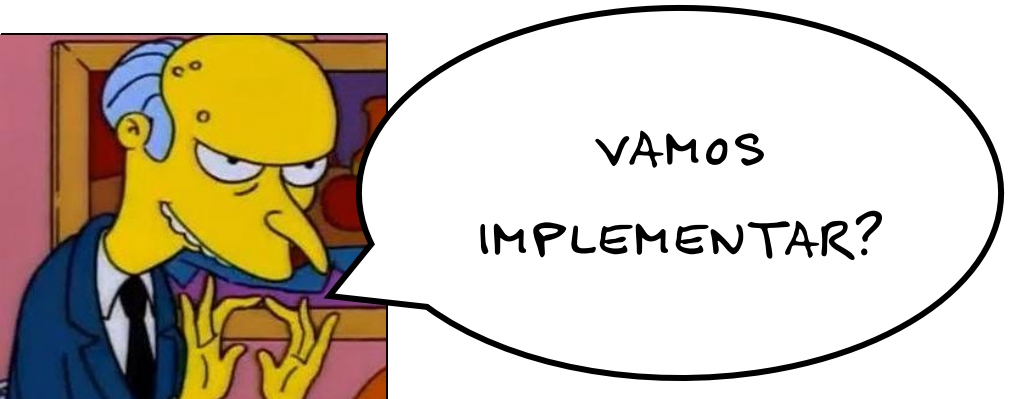
Programa para uma interface,
não para uma implementação!

Lembre-se do 5º princípio SOLID:

Programe para uma **abstração**, não para uma **implementação**.

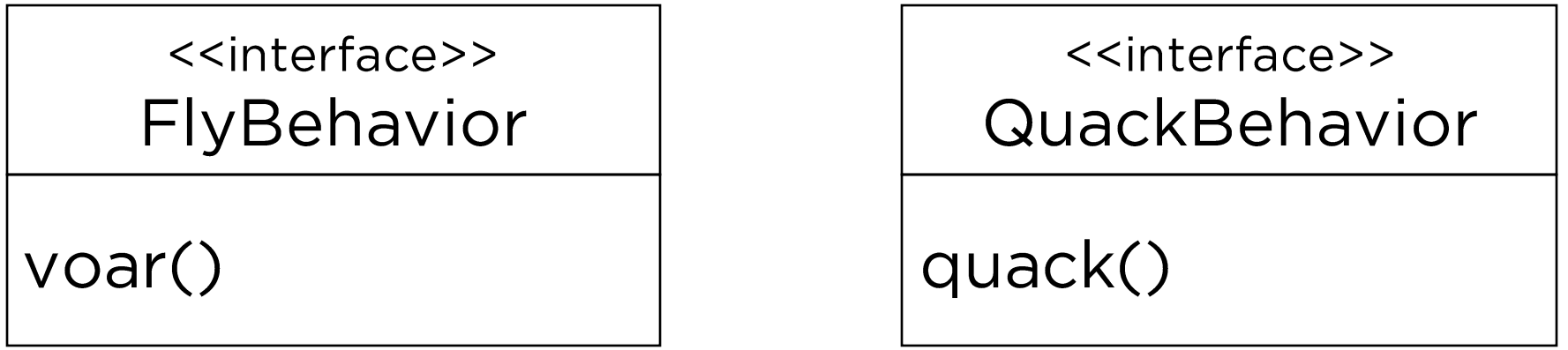
Recapitulando os 2 Princípios OO

1. Identifique os aspectos de seu aplicativo que variam e separe-os do que permanece igual.
2. Programa para uma interface, não para uma implementação!



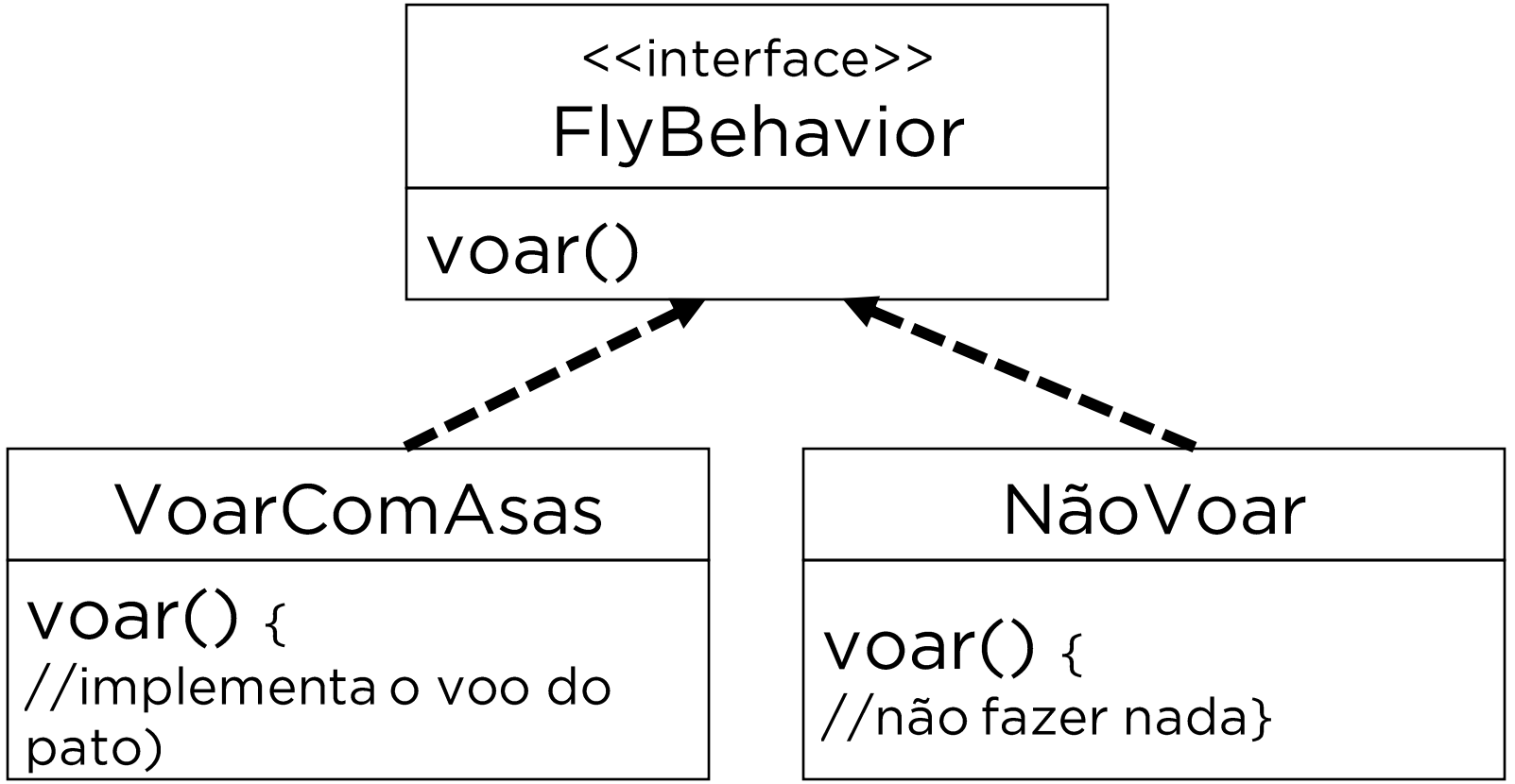
Implementando os conjuntos de comportamentos com Interfaces

- Vamos utilizar uma interface para representar cada comportamento específico: **FlyBehavior** e **QuackBehavior**.
- Cada implementação de um comportamento deverá implementar uma dessas interfaces.



Iremos utilizar classes que representam comportamentos

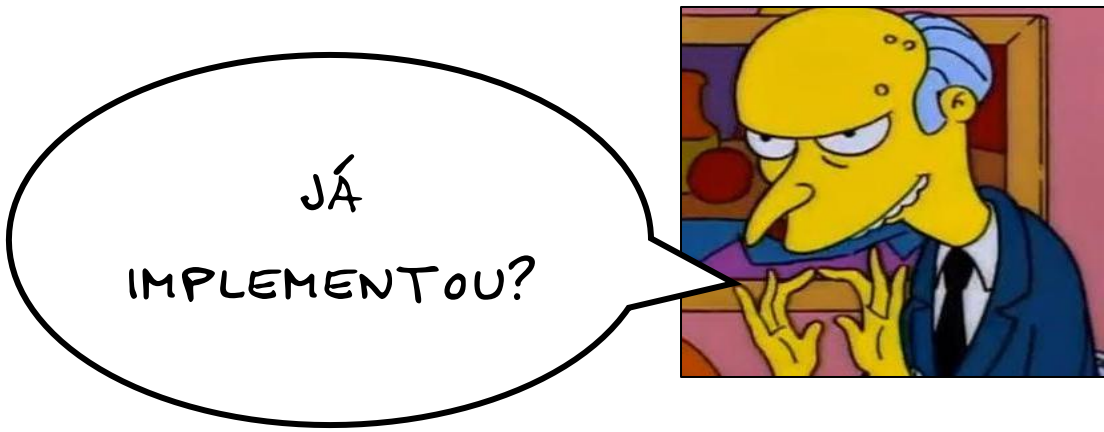
- Desta vez não serão as subclasses de Pato que irão implementar as interfaces de voar e grasnar.
- Iremos criar um conjunto de classes (VoarComAsas, NãoVoar) cuja razão de existir é representar um comportamento.

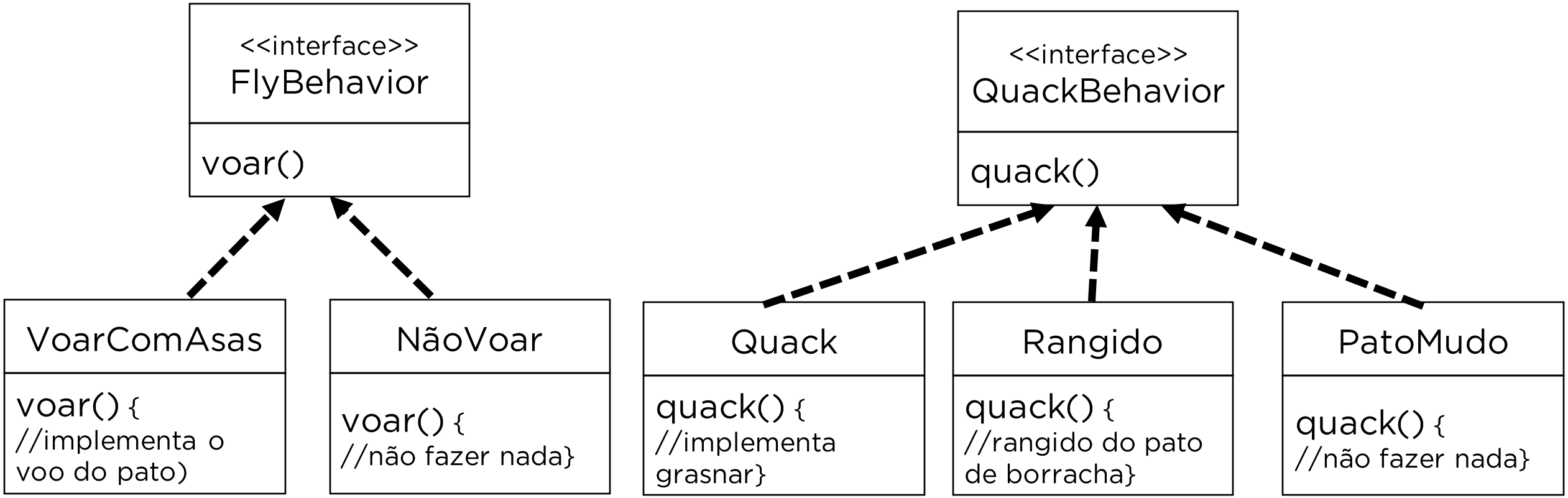


Encapsulamento

Como os **comportamentos** irão ficar **separados da superclasse Pato**, nenhuma subclasse de Pato precisa **conhecer os detalhes de implementação** dos comportamentos.

Como serão as Famílias de Classes?





Integrando as classes que definem comportamentos com a superclasse Pato.

Integrando o comportamento

- **Iremos adicionar duas variáveis de instância a super classe Pato,** chamadas de flyBehavior e quackBehavior que são variáveis das interfaces FlyBehavior e QuackBehavior.
- Cada objeto de Pato irá definir essas variáveis de instância de acordo com o comportamento específico que gostaria de ter em tempo de execução.
- Iremos também substituir os métodos voar() e quack() da classe Pato por dois métodos parecidos: **performFly()** e **performQuack()**.

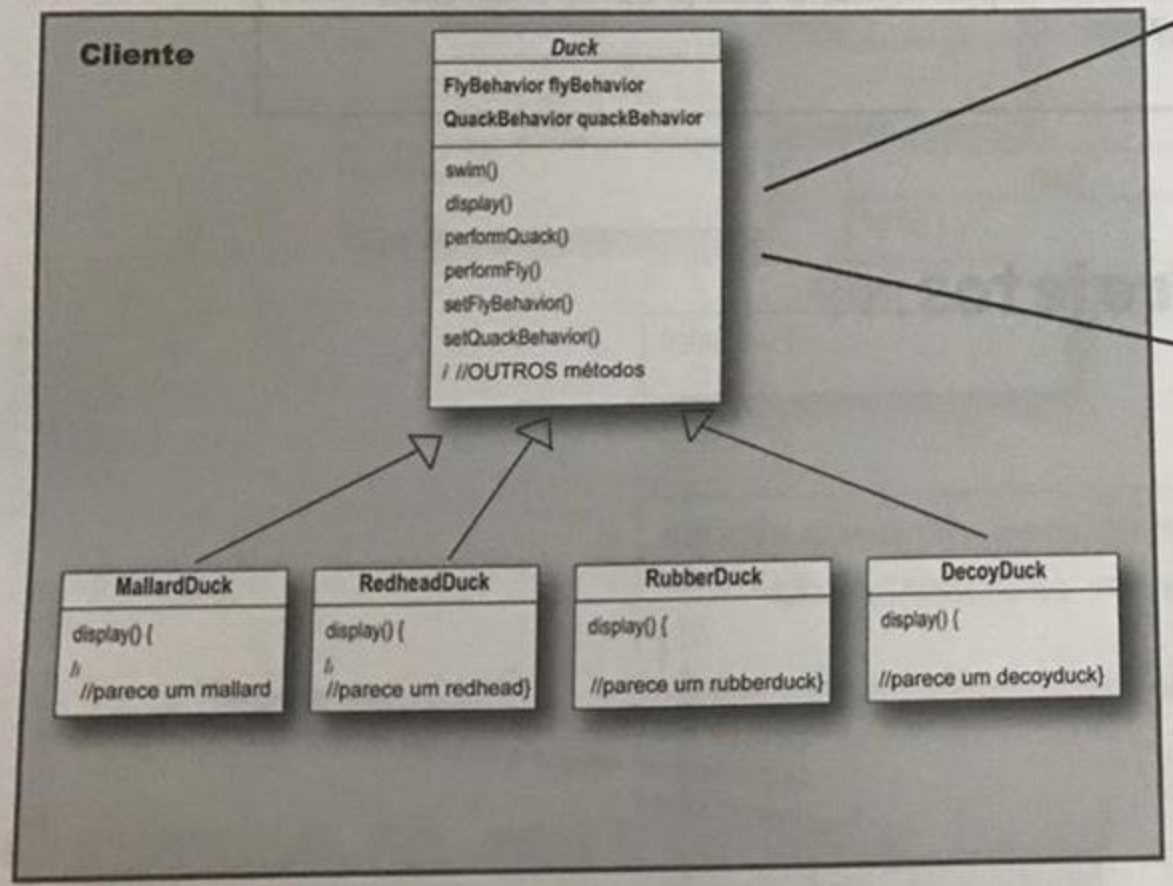
Classe Pato ajustada

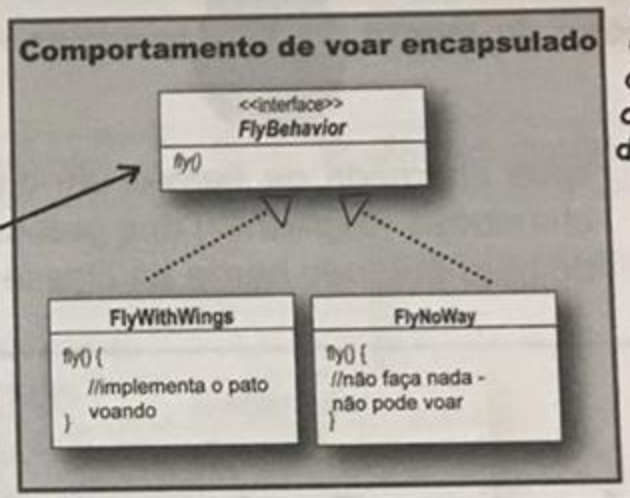
Pato
FlyBehavior **flyBehavior** QuackBehavior **quackBehavior**
peformQuack() peformFly() nadar() exibir() //outros métodos...

- As variáveis de comportamento flyBehavior e quackBehavior são declaradas com o tipo da interface.
- Essas variáveis de instância vão possuir uma referência para um comportamento específico em tempo de execução.

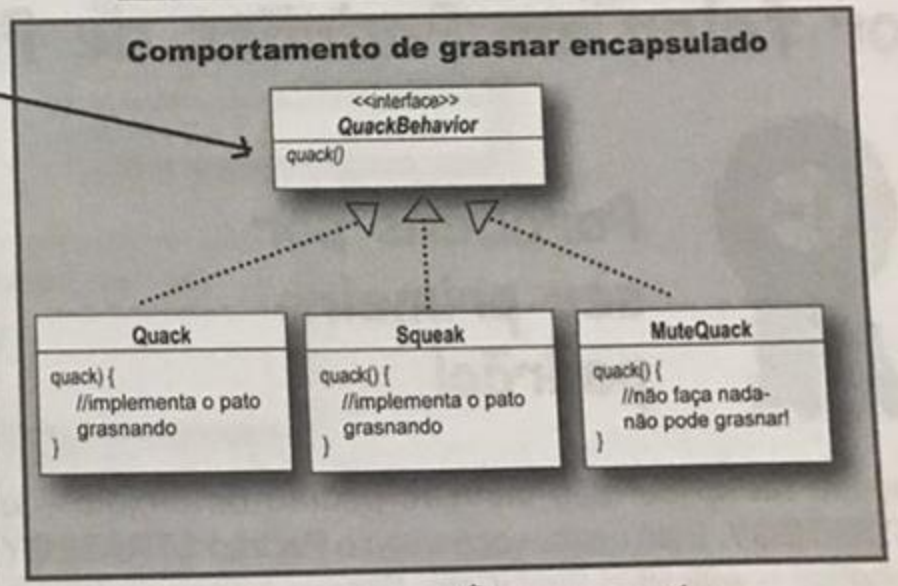
- **O que podemos colocar dentro dessas variáveis?**
- Objetos de classes que implementam suas interfaces.

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





Pense em cada conjunto de comportamentos como uma família de algoritmos.



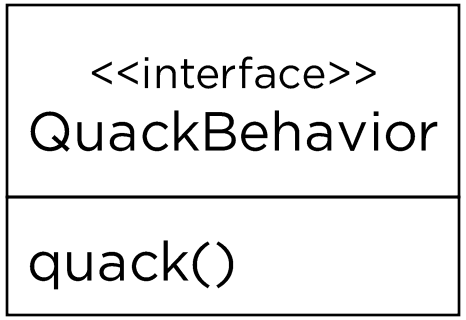
Esses algoritmos de comportamento são intercambiáveis.

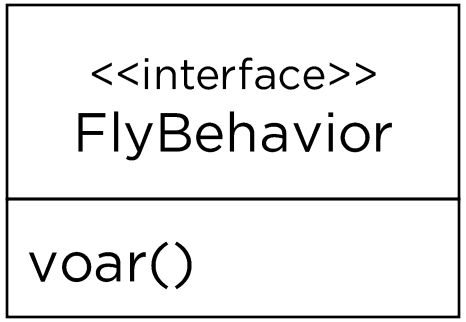
E como implementamos os métodos peformQuack() e peformFly()?

Implementação

```
public class Pato {
        QuackBehavior quackBehavior;
        FlyBehavior flyBehavior;

        public void peformQuack() {
                quackBehavior.quack();
        }
        public void peformFly() {
                flyBehavior.voar();
        }
}
```



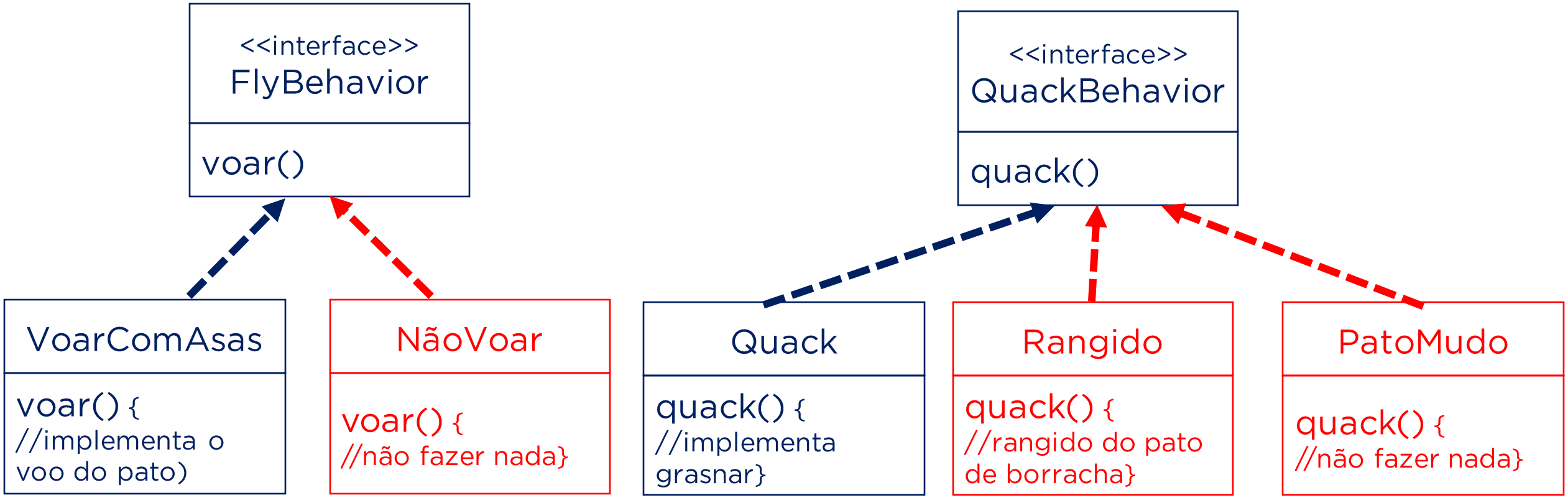


E as subclasses de Pato?

```
public class PatoReal extends Pato {
        public PatoReal() {
                quackBehavior = new Quack();
                flyBehavior = new VoarComAsas();
        }

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

Em azul, estão os comportamentos selecionados para a classe PatoReal



Parabéns! Você entendeu como funciona o primeiro padrão de projeto

- Acabamos de aplicar o padrão de projeto **Strategy**.
- Na aula de hoje, utilizamos o Strategy para refatorar o software Patinho na Lagoa.
- Graças a essa alteração, agora o software está pronto para qualquer alteração que os investidores possam exigir.

Conceito do 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.

Lembre-se:

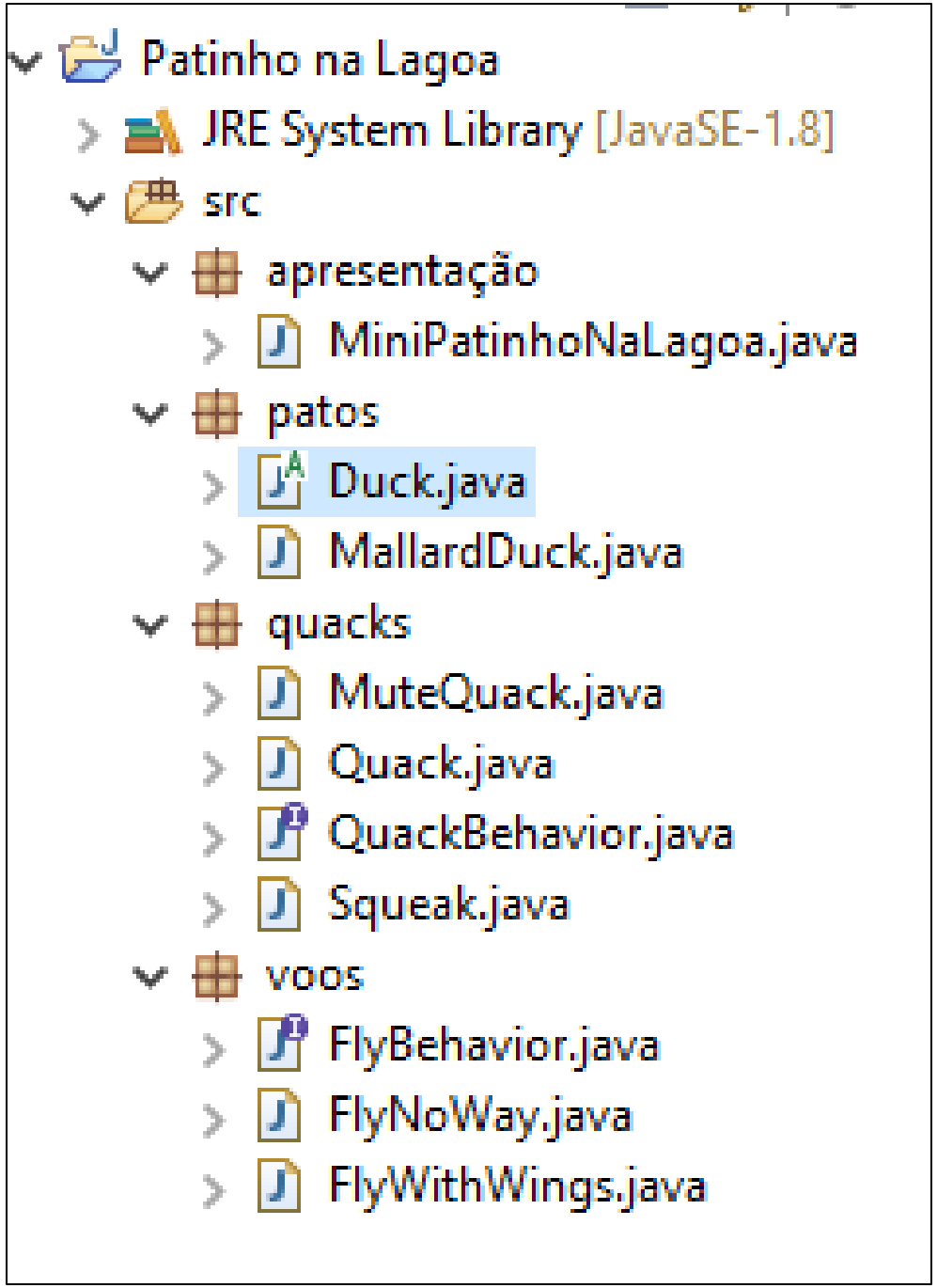
- A melhor maneira de utilizar os padrões é carregar seu cérebro com eles e depois reconhecer seus designs e aplicativos existentes onde eles possam ser aplicados.
- Em vez da reutilização do código, com os padrões você obtém **reutilização de experiência**.

Atividade

Implementar o código do livro
Use a Cabeça. Padrões de Projeto. p. 37-40

Já está na sala virtual!

Estrutura do Projeto



Interface FlyBehavior e classes FlyNoWay e FlyWithWings

*Duck.java | FlyBehavior.... | FlyW

```
1  package voos;
2
3  public interface FlyBehavior {
4      public void fly();
5  }
6
```

*Duck.java | FlyBehavior.... | FlyWithWing... | FlyNoWay.java

```
1  package voos;
2
3  public class FlyNoWay implements FlyBehavior{
4
5      @Override
6      public void fly() {
7          System.out.println("Eu não consigo voar :( ");
8
9      }
10
11 }
12
```

*Duck.java | FlyBehavior.... | FlyWithWing... | FlyN

```
1  package voos;
2
3  public class FlyWithWings implements FlyBehavior{
4
5      @Override
6      public void fly() {
7          System.out.println("To voando!!!");
8      }
9
10 }
11
```

Interface QuackBehavior e classes Quack, Squeak e MuteQuack

*Duck.java | QuackBehavi... | FlyB

```
1  package quacks;
2
3  public interface QuackBehavior {
4      public void quack();
5  }
6
```

*Duck.java | QuackBehavi... | Quack.java

```
1  package quacks;
2
3  public class Quack implements QuackBehavior {
4
5      @Override
6      public void quack() {
7          System.out.println("Quack Quack!");
8      }
9
10 }
11
```

*Duck.java | QuackBehavi... | *Squeak.java

```
1  package quacks;
2
3  public class Squeak implements QuackBehavior {
4
5      @Override
6      public void quack() {
7          System.out.println("Squeeeakkkk");
8      }
9  }
10
```

*Duck.java | QuackBehavi... | *MuteQuack.java

```
1  package quacks;
2
3  public class MuteQuack implements QuackBehavior {
4
5      @Override
6      public void quack() {
7          System.out.println("<<silence>>");
8
9      }
10 }
11
```

Classe Duck

*Duck.java | FlyBehavior.... | FlyWithWing... | Qua

```
1  package patos;
2
3  import quacks.QuackBehavior;
4  import voos.FlyBehavior;
5
6  public abstract class Duck {
7      FlyBehavior flyBehavior;
8      QuackBehavior quackBehavior;
9
10     public Duck() {}
11
12     public void performFly() {
13         flyBehavior.fly();
14     }
15
16     public void performQuack() {
17         quackBehavior.quack();
18     }
19
20     public void swim() {
21         System.out.println("Todo pato flutua");
22     }
23 }
24
```

Classe MallardDuck

QuackBehavi... | *MuteQuack.java | *Squeak.java | MallardDuck....

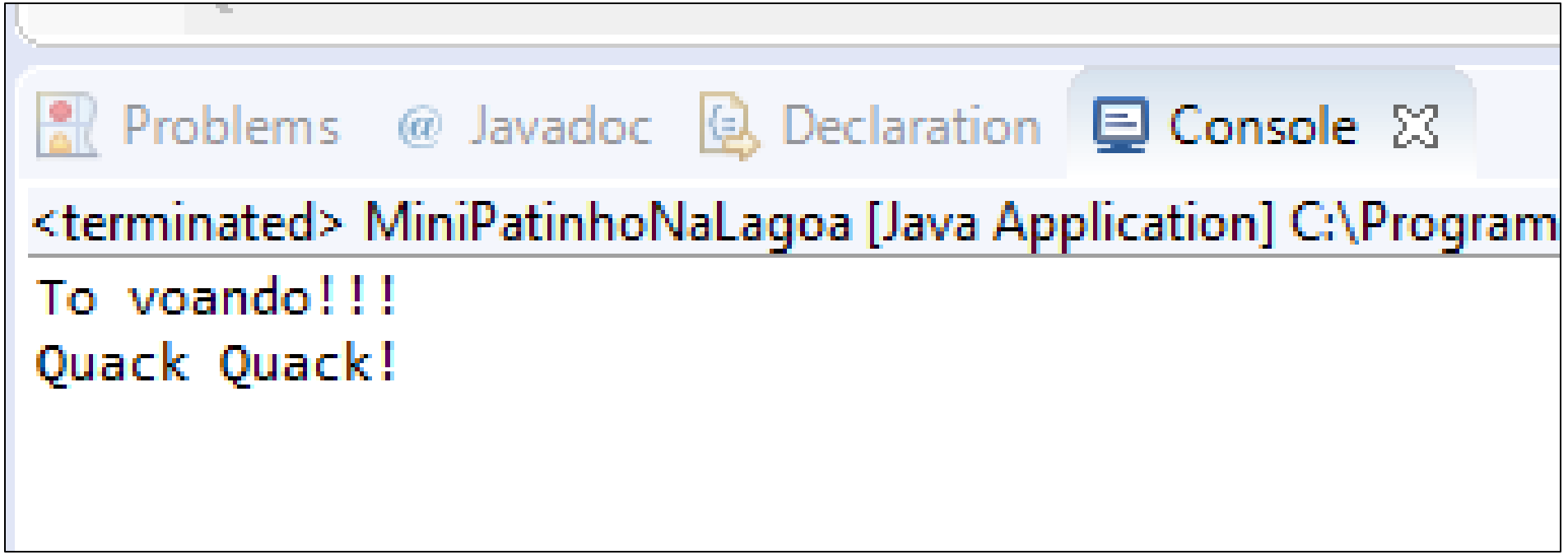
```
1  package patos;
2
3  import quacks.Quack;
4  import voos.FlyWithWings;
5
6  public class MallardDuck extends Duck {
7
8      public MallardDuck() {
9          quackBehavior = new Quack();
10         flyBehavior = new FlyWithWings();
11     }
12
13     public void display() {
14         System.out.println("Eu sou um verdadeiro Pato Real!");
15     }
16
17 }
18
```

Classe MiniPatinhoNaLagoa com método main

QuackBehavi... | MiniPatinhoN... | *MuteQuack.jav

```
1  package apresentação;
2
3  import patos.MallardDuck;
4  import patos.Duck;
5
6  public class MiniPatinhoNaLagoa {
7
8      public static void main(String[] args) {
9          Duck mallard = new MallardDuck();
10         mallard.performFly();
11         mallard.performQuack();
12     }
13
14 }
```

No console


Problems
Javadoc
Declaration
Console
<terminated> MiniPatinhoNaLagoa [Java Application] C:\Program
To voando!!!
Quack Quack!

Questão Enade

Outros padrões

Singleton

- Singleton é um (anti)padrão de projeto de software.
- Garante a existência de **apenas uma instância de uma classe,** mantendo um ponto global de acesso ao seu objeto.

```
1  package com.SingleTon;
2  /**
3   *
4   * @author mazzi
5   */
6      public class Janela2 {
7          private static Janela2 instancia;
8
9          private Janela2() {
10         }
11
12         public static synchronized Janela2 getInstance() {
13                 if (instancia == null) {
14                         instancia = new Janela2();
15                 }
16                 return instancia;
17         }
18
19 }
20
```

Template Method

- Permite definir o **esqueleto de um algoritmo** em uma classe base e permitir que as subclasses substituam as etapas sem alterar a estrutura geral do algoritmo.
- Ou seja, vai definir o esqueleto de um algoritmo em uma operação, **postergando** alguns passos para as subclasses.

Iterator

- Tem como objetivo encapsular uma iteração de uma lista.
- O padrão depende de uma interface chamada **Iterator** que tem 2 métodos: **next()** para iterar para o próximo elemento e **hasNext()** verifica se existem mais elementos para serem iterados.

- Este padrão permite percorrer elementos de uma coleção sem expor as suas representações.

QUESTÃO 25

Um Padrão de Projeto nomeia, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizáveis.

GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J. **Padrões de Projeto-Soluções Reutilizáveis de Software Orientado a Objetos**. Porto Alegre: Bookman, 2000.

Em relação a Padrões de Projeto, analise as afirmações a seguir.

É correto o que se afirma em?

I. *Prototype* é um tipo de padrão estrutural.

II. *Singleton* tem por objetivos garantir que uma classe tenha ao menos uma instância e fornecer um ponto global de acesso para ela.

III. *Template Method* tem por objetivo definir o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses.

IV. *Iterator* fornece uma maneira de acessar sequencialmente os elementos de um objeto agregado sem expor sua representação subjacente.

É correto apenas o que se afirma em

ADS 2011 – Questão 25

(A) I.

(B) II.

(C) I e IV.

(D) II e III.

(E) III e IV.

Quizlet
Padrões de Projetos

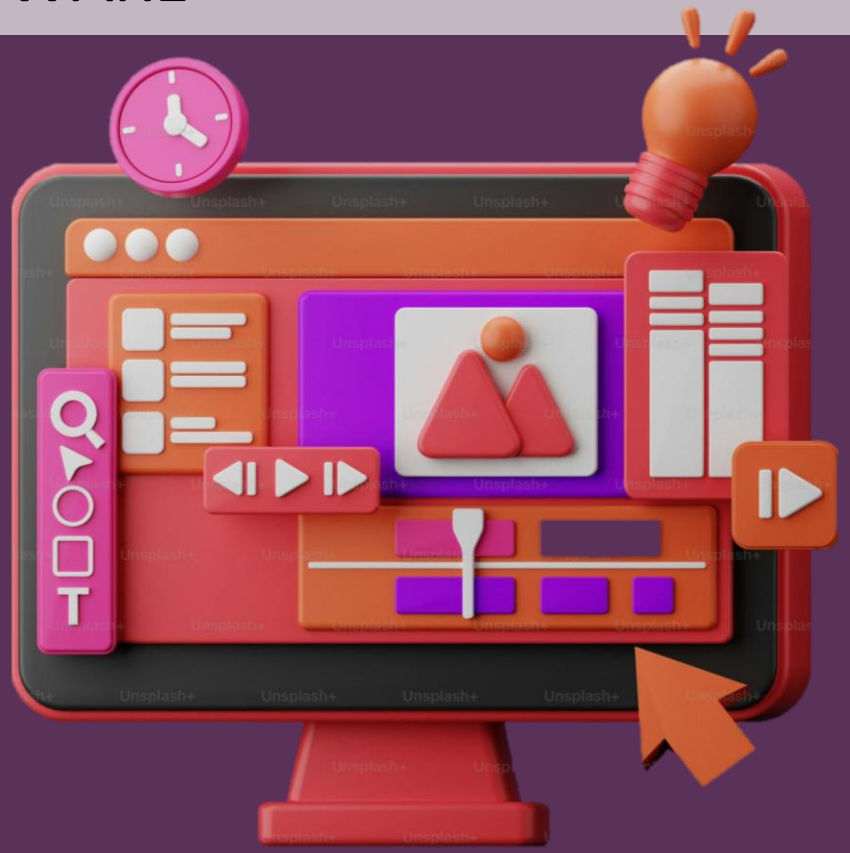
https://quizlet.com/_aj24ic?x=1qqt&i=14uvey

Referências

- FREEMAN, E.; FREEMAN, E. Padrões de Projeto. Use a cabeça!. Alta Books: Rio de Janeiro, 2009. p. 79-124, 235-288.
- https://www.devmedia.com.br/padrao-de-projeto-factory-method-em-java/26348
- https://www.devmedia.com.br/padrao-de-projeto-iterator-em-java/26733

ENGENHARIA DE SOFTWARE

AULA 12 – PADRÕES DE PROJETOS DE SOFTWARE



WALTER TRAVASSOS SARINHO

@PROF.WALTERTRAVASSOS

WALTER.TRAVASSOS@SECTRAS.EDU.BR