Strategy

Princípios e Padrões de Projeto

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Campus Rio Grande

Agenda

Introdução

O que é um padrão de projeto?

Cada padrão descreve um problema no nosso ambiente e o cerne da sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma forma (Christopher Alexander)

Características

- Os padrões de projeto tornam mais fácil reutilizar projetos e arquiteturas bem-sucedidas
- Os padrões de projeto ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e a evitar alternativas que comprometam a utilização
- Os padrões de projeto podem melhorar a documentação e a manutenção de sistemas ao fornecer uma especificação explícita de interações de classes e objetos e o seu objetivo subjacente

Elementos de um Padrão de Projeto

1. **O nome do padrão** é uma referência que podemos usar para descrever uma problema de projeto, suas soluções e consequências em uma ou duas palavras.
2. **problema** descreve em que situação aplicar o padrão. Ele explica o problema e seu contexto. Algumas vezes, o problema incluirá uma lista de condições que devem ser satisfeitas para que faça sentido aplicar o padrão.

Elementos de um Padrão de Projeto

3. **solução** descreve elementos que compõem o padrão de projeto, seus relacionamentos, suas responsabilidades e colaborações.
    - A solução não descreve um projeto concreto ou uma implementação em particular porque um padrão é como um gabarito que pode ser aplicado em muitas situações diferentes.

Elementos de um Padrão de Projeto

4. As **consequências** são os resultados e análises das vantagens e desvantagens da aplicação do padrão.
   - Embora as consequências sejam raramente mencionadas quando descrevemos decisões de projeto, elas são críticas para a avaliação de alternativas de projetos e para a compreensão dos custos e benefícios da aplicação do padrão

Classificação

- Classificamos os padrões de projeto por 2 critérios:
  - **Finalidade**: reflete o que um padrão faz.
    - Os padrões podem ter finalidade de **criação**, **estrutural** ou **comportamental**.
  - **Escopo**: especifica se o padrão se aplica primariamente a classes ou objetos.
    - **Os padrões de classes** lidam com os relacionamentos entre classes e subclasses (estático)
    - **Os padrões de objetos** lidam com relacionamentos entre objetos que podem ser mudados em tempo de execução (dinâmico)

A classificação ajuda a aprender os padrões mais rapidamente, bem como direcionar esforços na descoberta de novos.

Classificação

1. Os padrões de criação se preocupam com o processo de criação dos objetos
2. Os padrões estruturais lidam com a composição de classes ou de objetos
3. Os padrões comportamentais caracterizam as maneiras pela quais classes ou objetos interagem e distribuem responsabilidades

Nosso Primeiro Padrão de Projeto

Joe trabalha para uma empresa que cria um jogo de simulação de lago com patos de grande sucesso, o SimDuck.

O jogo pode mostrar uma grande variedade de espécies de patos nadando e produzindo sons.

Os designers iniciais do sistema usaram técnicas OO padrão e criaram uma superclasse Duck (Pato) herdada por todos os tipos de pato.

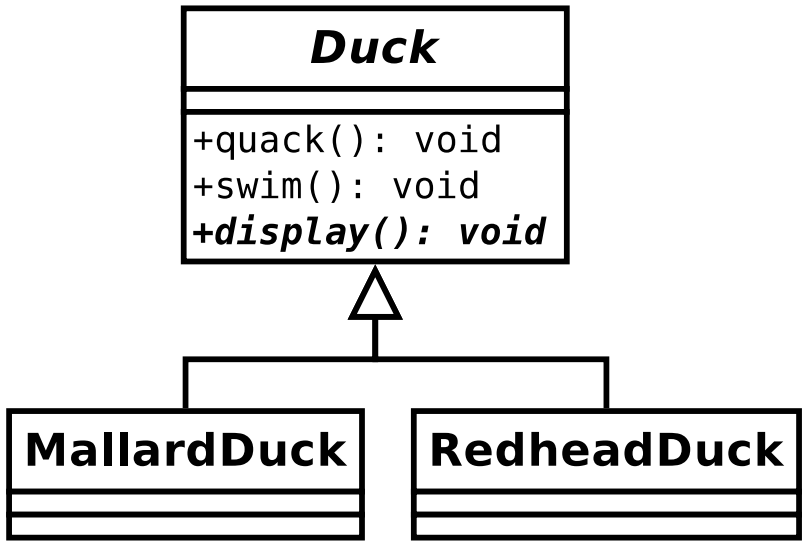
Nosso Primeiro Padrão de Projeto

Todos os patos grasnam e nadam, a superclasse cuida do código de implementação.

O método display():void é abstrato já que todos os subtipos de pato são diferentes

Muitos outros tipos de pato herdam da classe Duck

Cada subtipo de pato é responsável por implementar seu próprio comportamento display() para o modo como aparece na tela

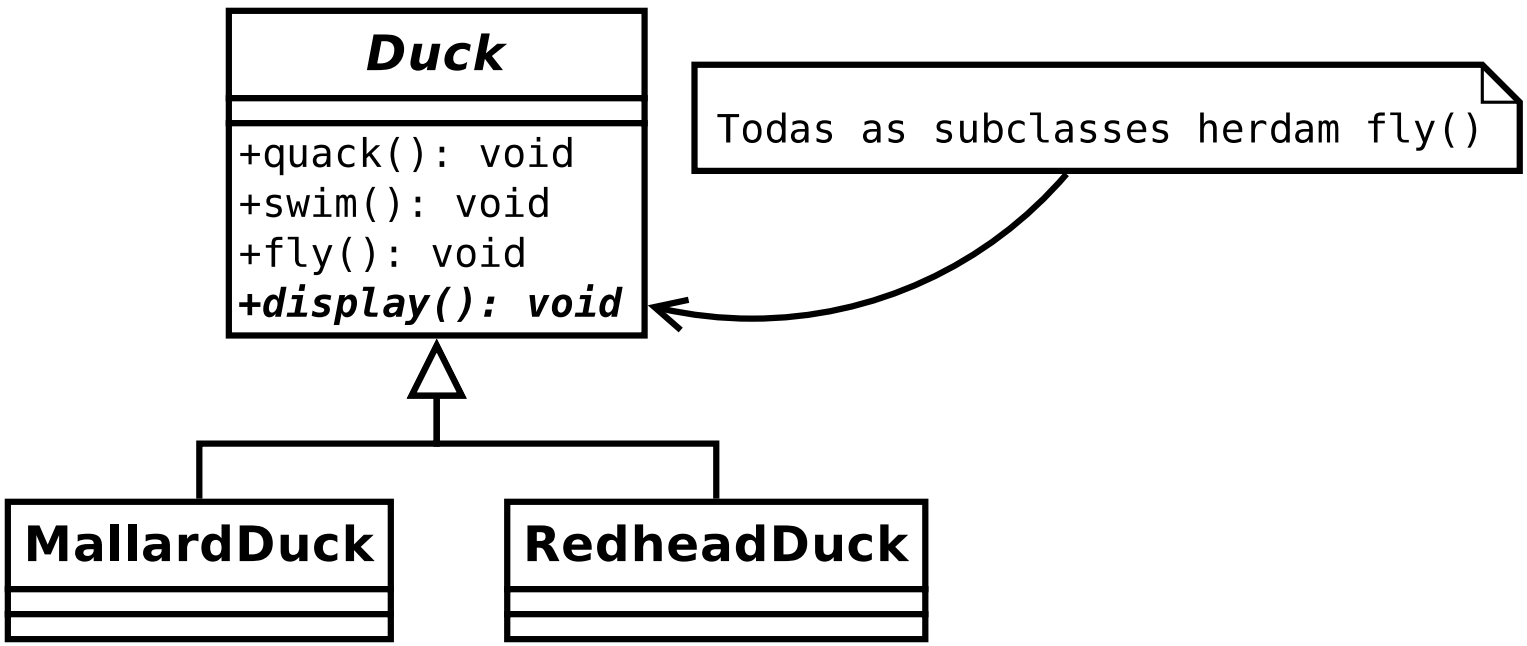


Nosso Primeiro Padrão de Projeto

Necessidade de Alteração....

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

Sugestão de Joe - Utilizando Herança



Nosso Primeiro Padrão de Projeto

Mas alguma coisa deu muito errado...

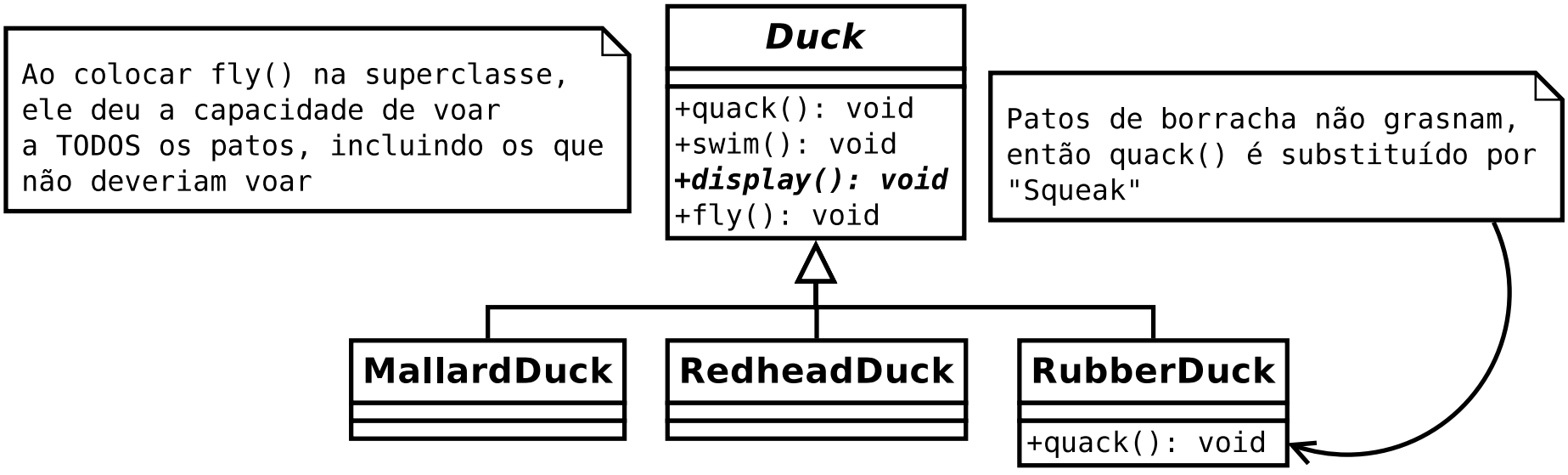
Joe, estou na reunião com os acionistas. Eles fizeram uma demonstração e havia **patos de borracha** voando pela tela!!!

Joe não percebeu que nem todas subclasses de Duck deveriam voar.

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

Quando Joe adicionou um comportamento a superclasse Duck, também está adicionando um comportamento que não era apropriado a algumas subclasses de Duck.

Nosso Primeiro Padrão de Projeto



Ao colocar fly() na superclasse ele deu a capacidade de voar TODOS os patos, incluindo aqueles que não deveriam voar.

Nosso Primeiro Padrão de Projeto

Joe percebeu que a herança não era a resposta.

Ele recebeu o aviso de que os executivos agora querem atualizar o produto a cada 6 meses (eles ainda não decidiram como).

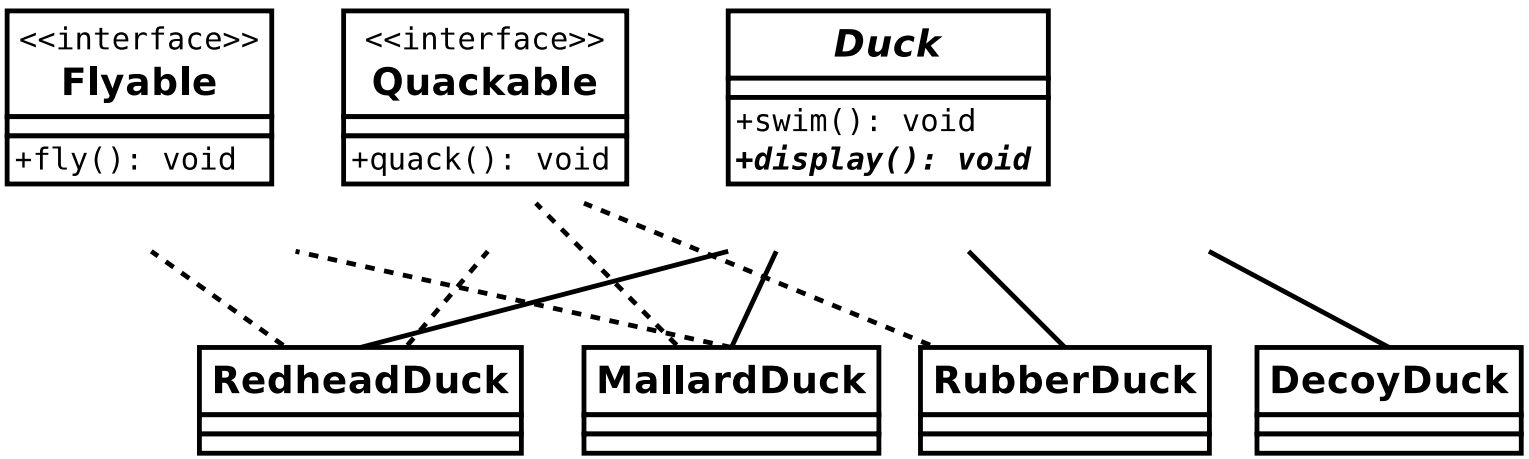
Joe sabe que as especificações vão continuar mudando e que ele será forçado a analisar e possivelmente substituir fly() e quack() para subclasse Duck adicionada ao programa .... sempre

Que tal interface?

Nosso Primeiro Padrão de Projeto

Pensamento do Joe

Eu poderia tirar fly() da superclasse Duck e criar uma interface Flyable() com o método fly(). Assim, somente os patos que devem voar irão implementar essa interface e ter o método fly()...



Nosso Primeiro Padrão de Projeto

Mensagem vinda do gerente de Joe...

Joe 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 vôo...de todas as 48 subclasses de vôo de Duck?!

Nosso Primeiro Padrão de Projeto

O que você faria se fosse o Joe?

Sabemos que nem todas as subclasses devem ter o comportamento de voar ou grasnar, então a herança não é a resposta certa.

Mas, embora fazer subclasses implementar Flyable e/ou Quackable resolva parte do problema (sem fazer patos de borracha voar inadequadamente), destrói completamente a reutilização de código para esses comportamentos, criando apenas um pesadelo de manutenção diferente. Isso significa que sempre que você precisar modificar um comportamento será forçado a monitorar e alterá-lo em todas subclasses diferentes onde esse comportamento é definido.

Nesse ponto você pode estar esperando que um Padrão de Projetos apareça em um cavalo branco e salve o dia...kkkk

Nosso Primeiro Padrão de Projeto

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

Princípio de Projeto

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

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

Todos os padrões fornecem uma maneira de deixar que alguma parte de um sistema varie independentemente de todas as outras partes.

Nosso Primeiro Padrão de Projeto

Por onde começamos?

Pelo que sabemos, além dos problemas com fly() e quack(), a classe Duck está funcionando bem e não há outras partes que variam ou são alteradas com frequência.

Assim, afora algumas pequenas mudanças, vamos deixar a classe Duck em paz.

Agora, para separar "partes que são alteradas das que ficam iguais", vamos criar 2 conjuntos de classes (totalmente separadas de Duck) , um para voar (fly) e uma para grasnar (quack). Cada conjunto de classes irá conter todas as implementações do seu respectivo comportamento.

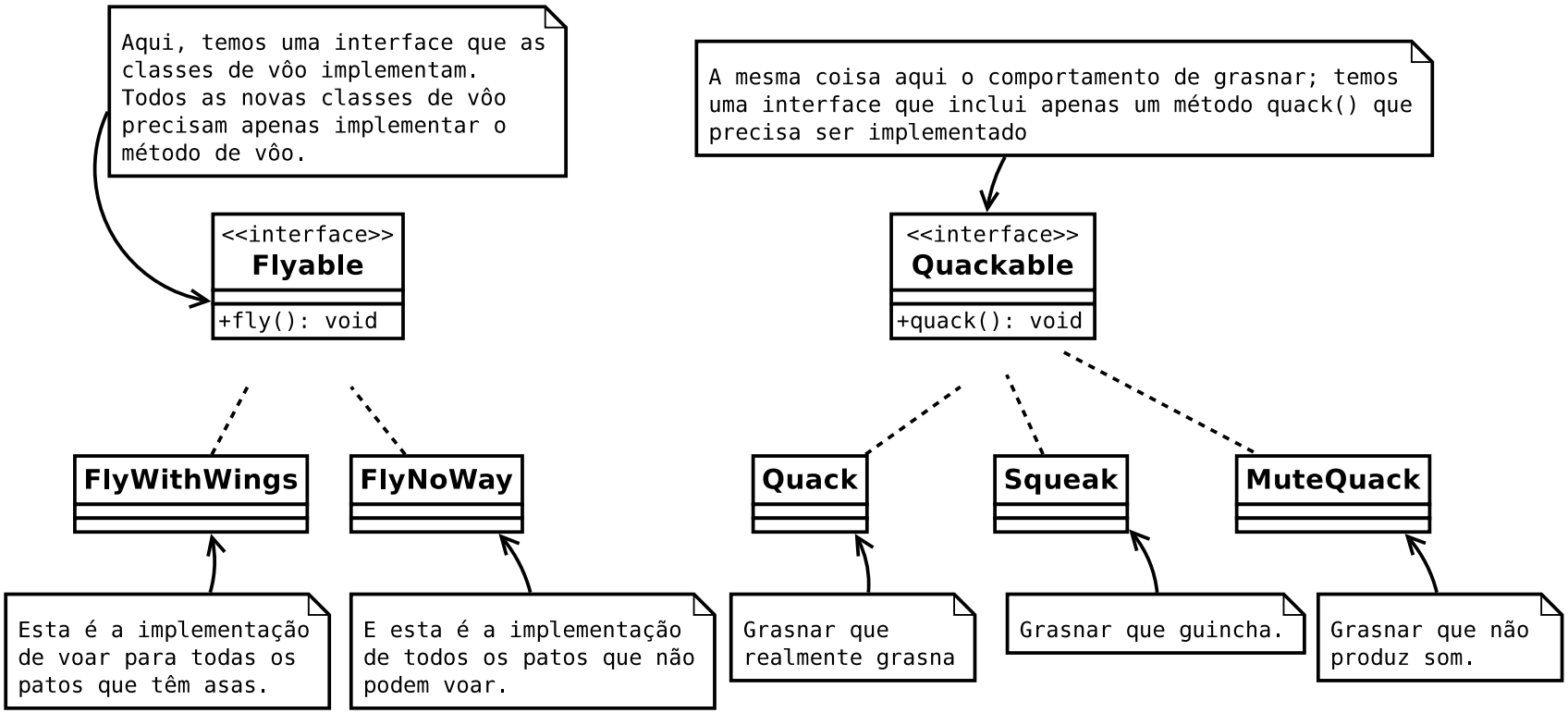
Nosso Primeiro Padrão de Projeto

Além disso, gostaríamos manter as coisas flexíveis, isto é, poderíamos querer criar uma nova instância MallardDuck e iniciá-la com um *tipo* específico de comportamento de vôo.

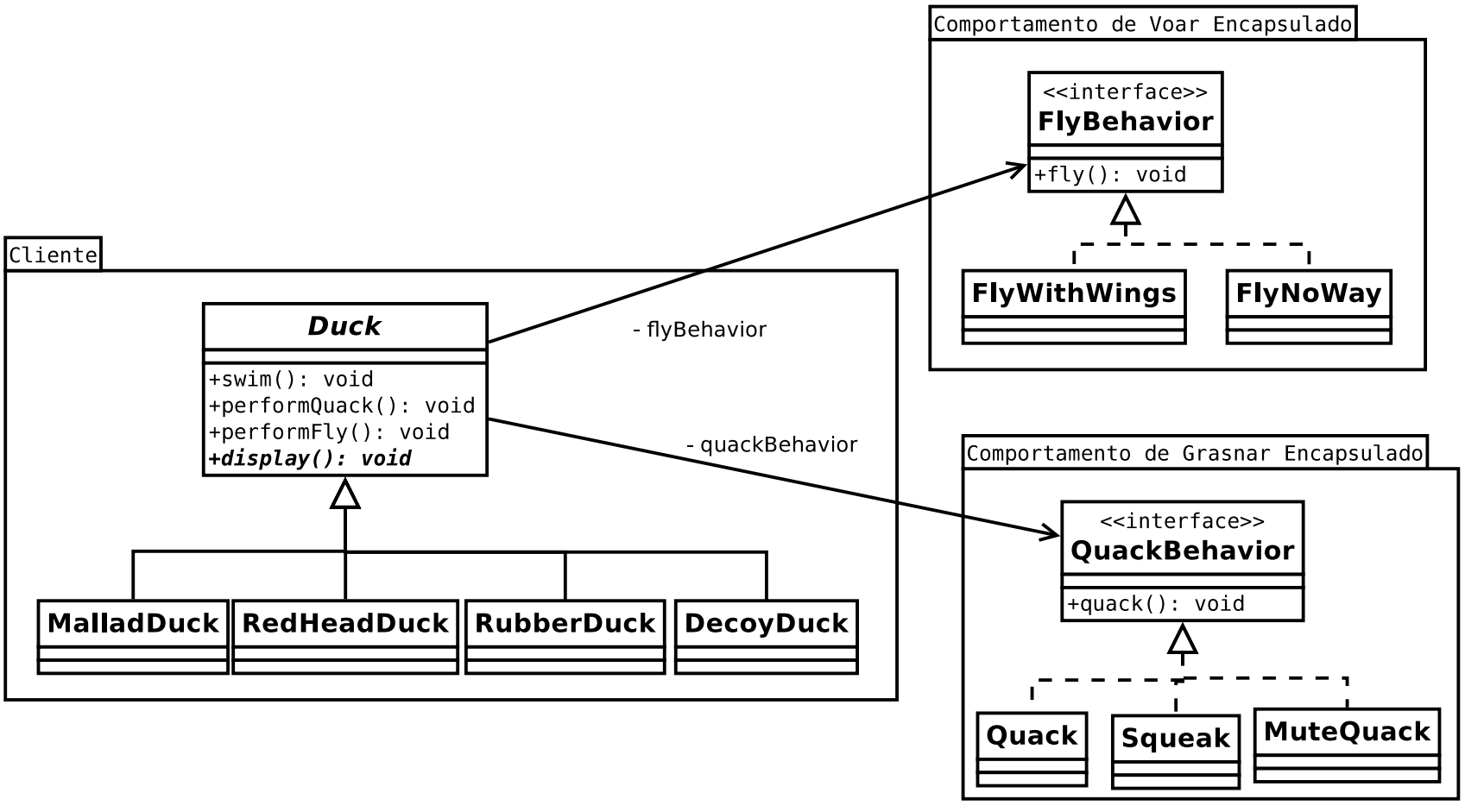
E, enquanto estivermos ali, por que não garantir que possamos alterar o comportamento de um pato de maneira dinâmica (em tempo de execução)?

Assim, desta vez não serão as classes Duck que irão implementar a interface de voar e grasnar. Em vez disso, vamos criar um conjunto de classes cuja razão de existir é representar um comportamento, e é a classe *comportamento* e não a classe Duck, que irá implementar a interface de comportamento.

Nosso Primeiro Padrão de Projeto



Nosso Primeiro Padrão de Projeto



Nosso Primeiro Padrão de Projeto

Com nosso novo design, as subclasses Duck irão usar um comportamento representado por uma interface FlyBehavior e QuackBehavior) de modo que a *implementação* real do comportamento não fique presa na subclasse Duck.

Com esse projeto, outros tipos de objetos podem reutilizar nossos comportamentos de voar e grasnar porque esses comportamentos não são escondidos em nossas classes Duck!

E podemos adicionar novos comportamentos sem modificar nenhuma de nossas classes de comportamento existentes ou tocar nenhuma classe Duck que utiliza comportamento de vôo

Nosso Primeiro Padrão de Projetos

Você acaba de aplicar o padrão STRATEGY.

Graças a esse padrão, o simulador está pronto para qualquer alteração que esses executivos possam aprontar.

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

Pontos Positivos e Negativos

Pontos Positivos

- o algoritmo poder ser alterado sem a modificação da classe
- novas implementações dele podem ser criadas e introduzidas posteriormente.
- Outra consequência positiva está no fato de a implementação poder ser trocada em tempo de execução, fazendo que o comportamento da classe possa ser trocado dinamicamente.

Pontos Positivos e Negativos

Pontos Negativos

- aumento da complexidade na criação do objeto, pois a instância da dependência precisa ser criada e configurada.
  - Caso o atributo seja nulo, a classe pode apresentar um comportamento inesperado.
- dessa solução está no aumento do número de classes: há uma para cada algoritmo, criando uma maior dificuldade em seu gerenciamento.

Nosso Primeiro Padrão de Projeto

Vamos implementar nosso padrão???

Após implmentação: Questionamentos

1. Usando nosso projeto, o que você faria se precisasse adicionar um vôo de um foguete ao aplicativo?
2. Você consegue pensar numa classe que poderia querer usar o comportamento Quack que não seja um pato?

Trabalho

Em um jogo de ação e aventura há os seguintes Personagens:

- Rainha
- Rei
- Cavaleiro

Cada Personagem pode *lutar* com as seguintes Armas:

- Espada
- Faca
- Arco e Flecha
- Machado

IMPORTANTE

Cada Personagem pode lutar utilizando uma Arma de cada vez, mas pode alterar as Armas a qualquer momento durante o jogo.

Trabalho

Tarefa

Implemente corretamente o padrão STRATEGY para o jogo em questão.

Dicas:

- O que são **comportamentos**?
- O que são **classes concretas**?
- Quais são **classes abstratas**?
- O que são **interfaces**?

Strategy

Princípios e Padrões de Projeto

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Campus Rio Grande