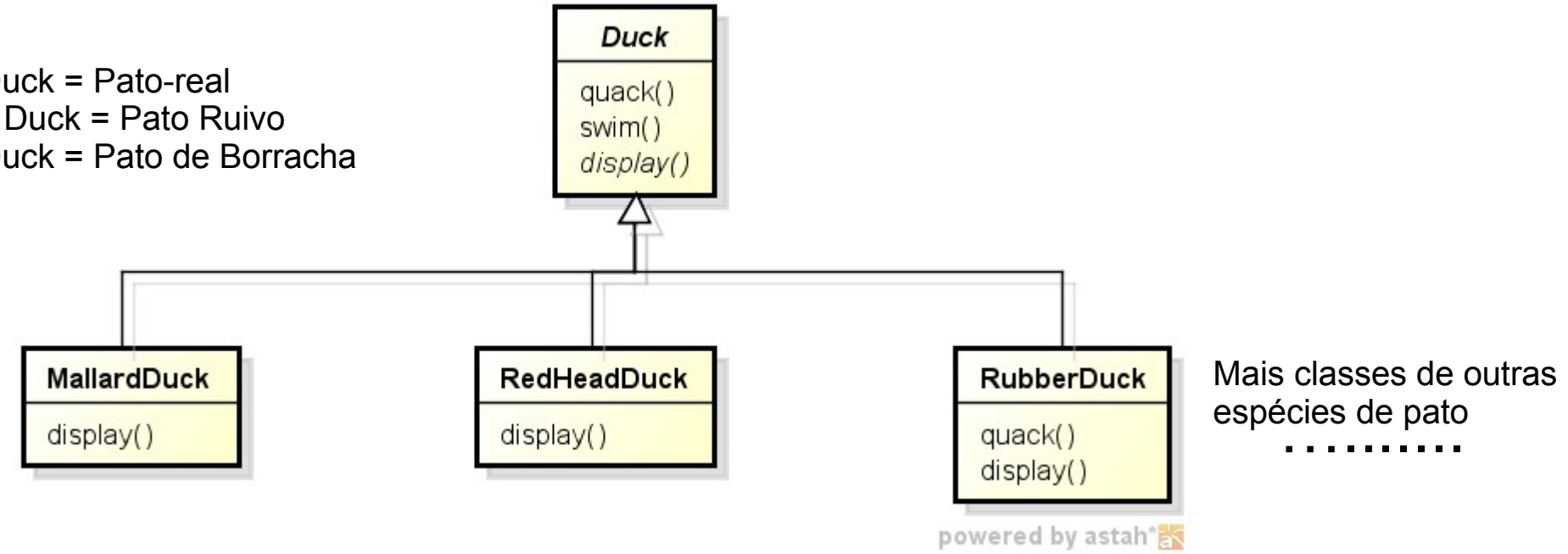
Padrões de Projeto
- Motivação -

Um Aplicativo

- Desenvolvemos um jogo que simula um lago com diferentes espécies de patos
- Os patos nadam e grasnam

Solução Inicial

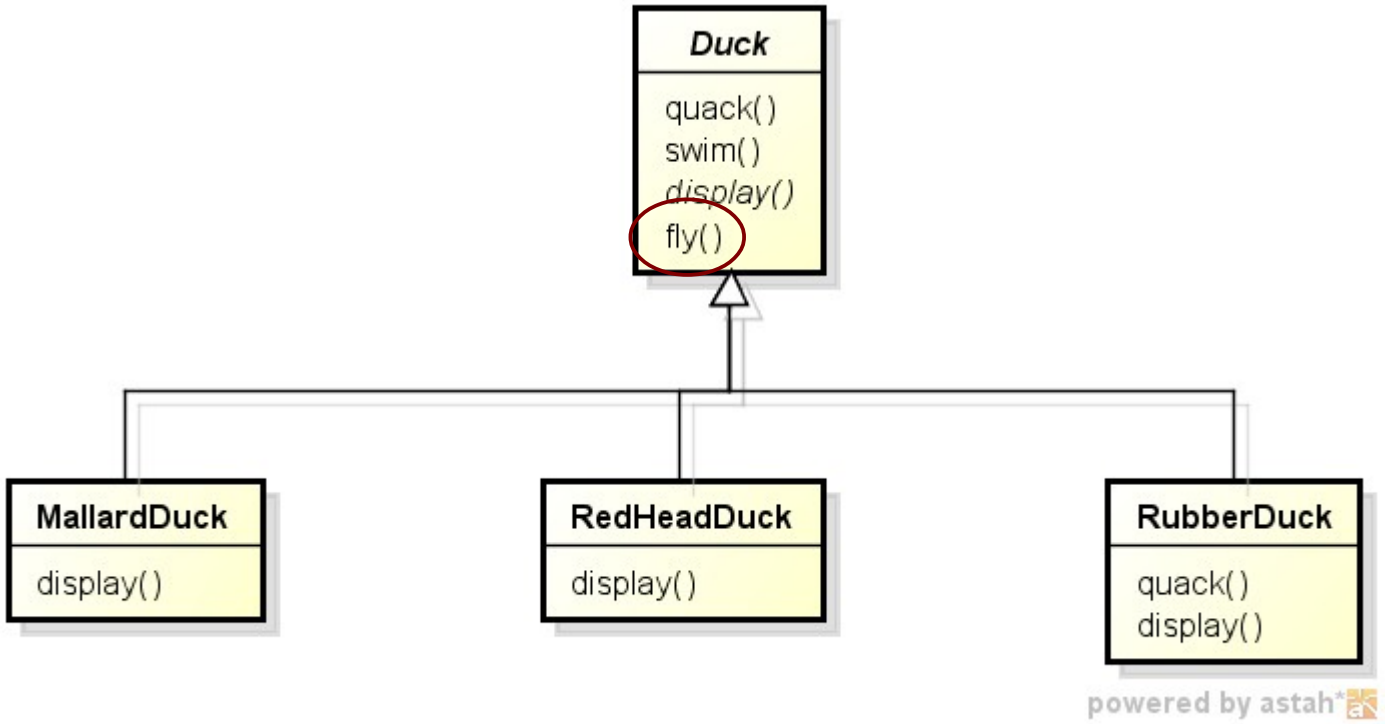
Mallard Duck = Pato-real
Redhead Duck = Pato Ruivo
Rubber Duck = Pato de Borracha



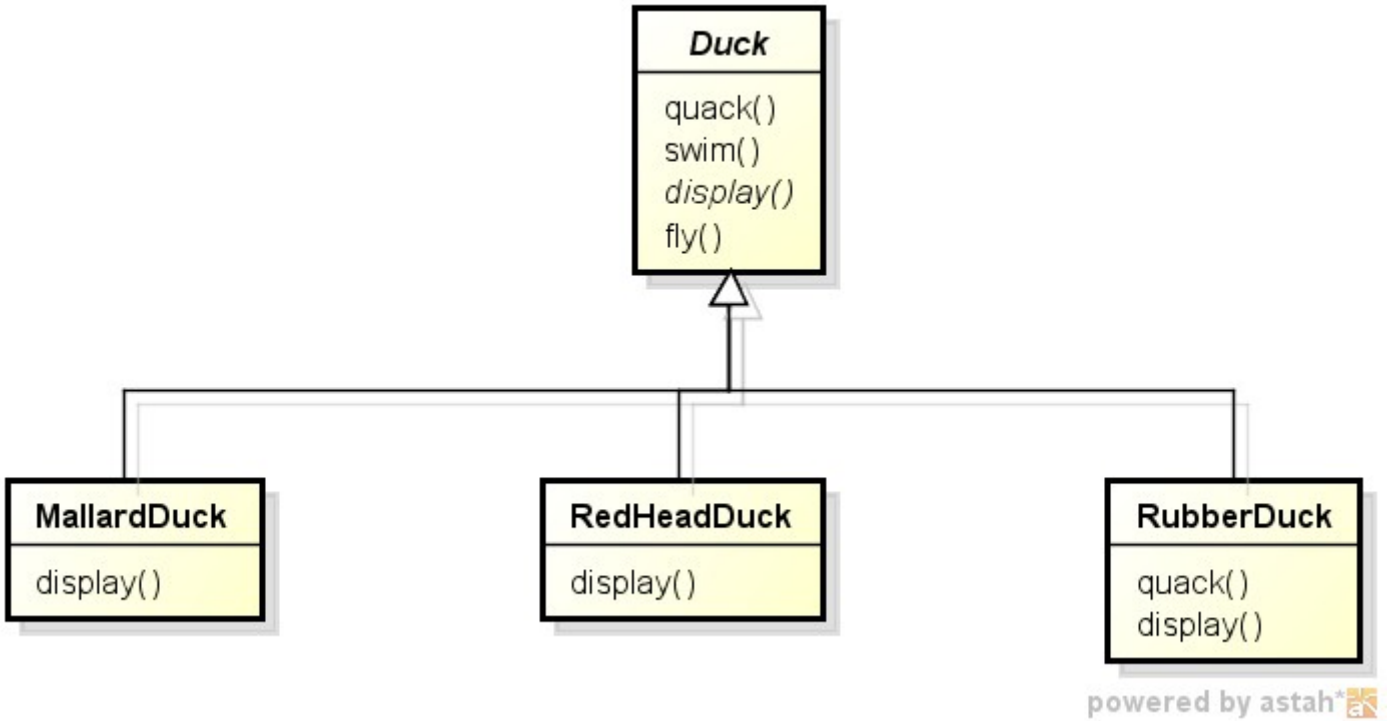
- Todos os patos grasnam e nadam. A superclasse Duck implementa os métodos quack() e swim().
- Cada subtipo de Duck implementa seu próprio comportamento display().
- O método display() de Duck é abstrato.
- O método quack() de RubberDuck substitui o som de grasnar pelo som de pato de borracha.

Novos requisitos

- Agora os patos tem que voar.
- Como usamos herança, a solução é simples, incluímos o método fly() na classe Duck, e todos os patos irão herdá-lo.

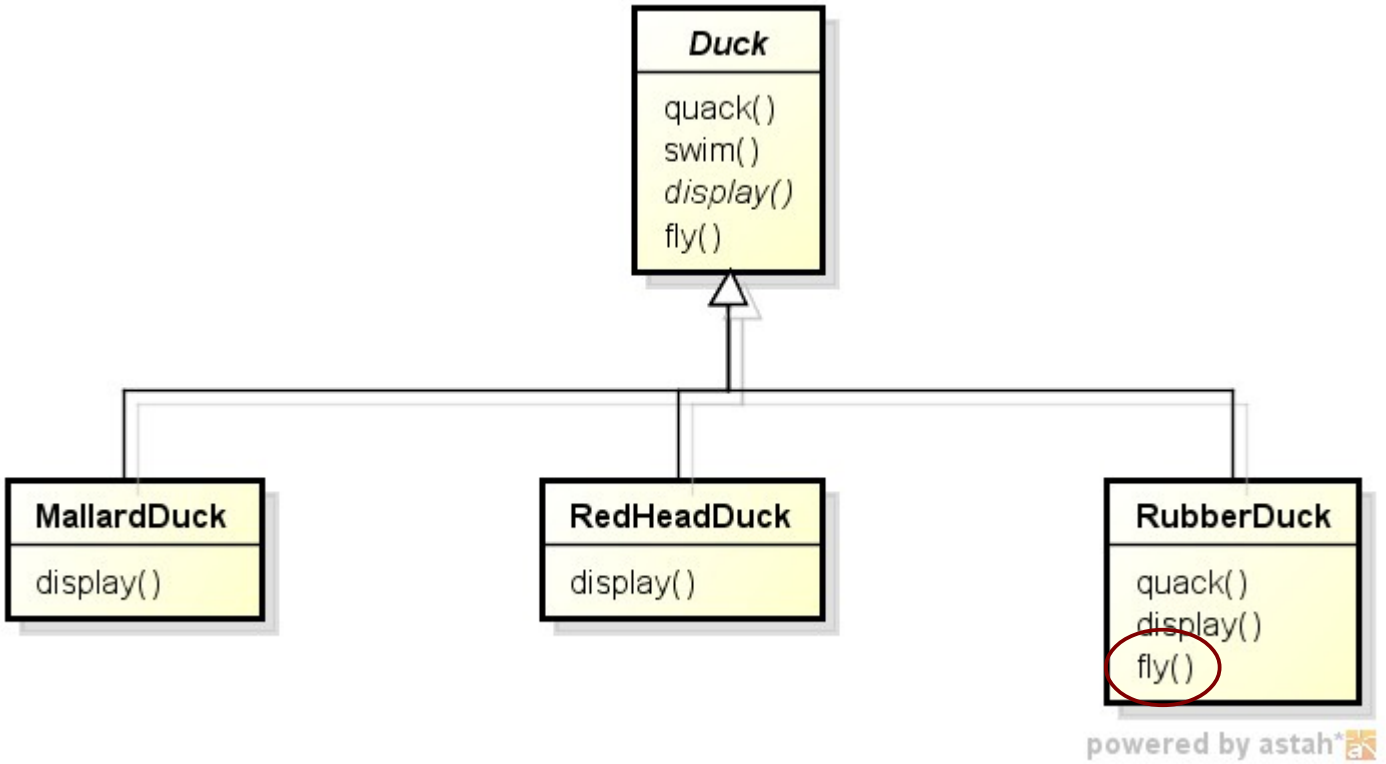


O que aconteceu?

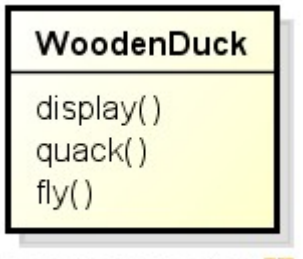


- Apareceram patos de borracha voando no jogo.
- Não percebemos que nem todos os patos deveriam voar.
- Uma atualização localizada no código (na classe Duck) causou um efeito colateral não local (patos de borracha voadores).

Uma possível solução



- Colocar um método fly() **vazio** na classe RubberDuck.
- E se precisarmos incluir um pato de madeira, que não voa e nem grasna?
  - teremos que incluir o método fly() **vazio**, na nova classe.
  - teremos que incluir o método quack() **vazio**, na nova classe.



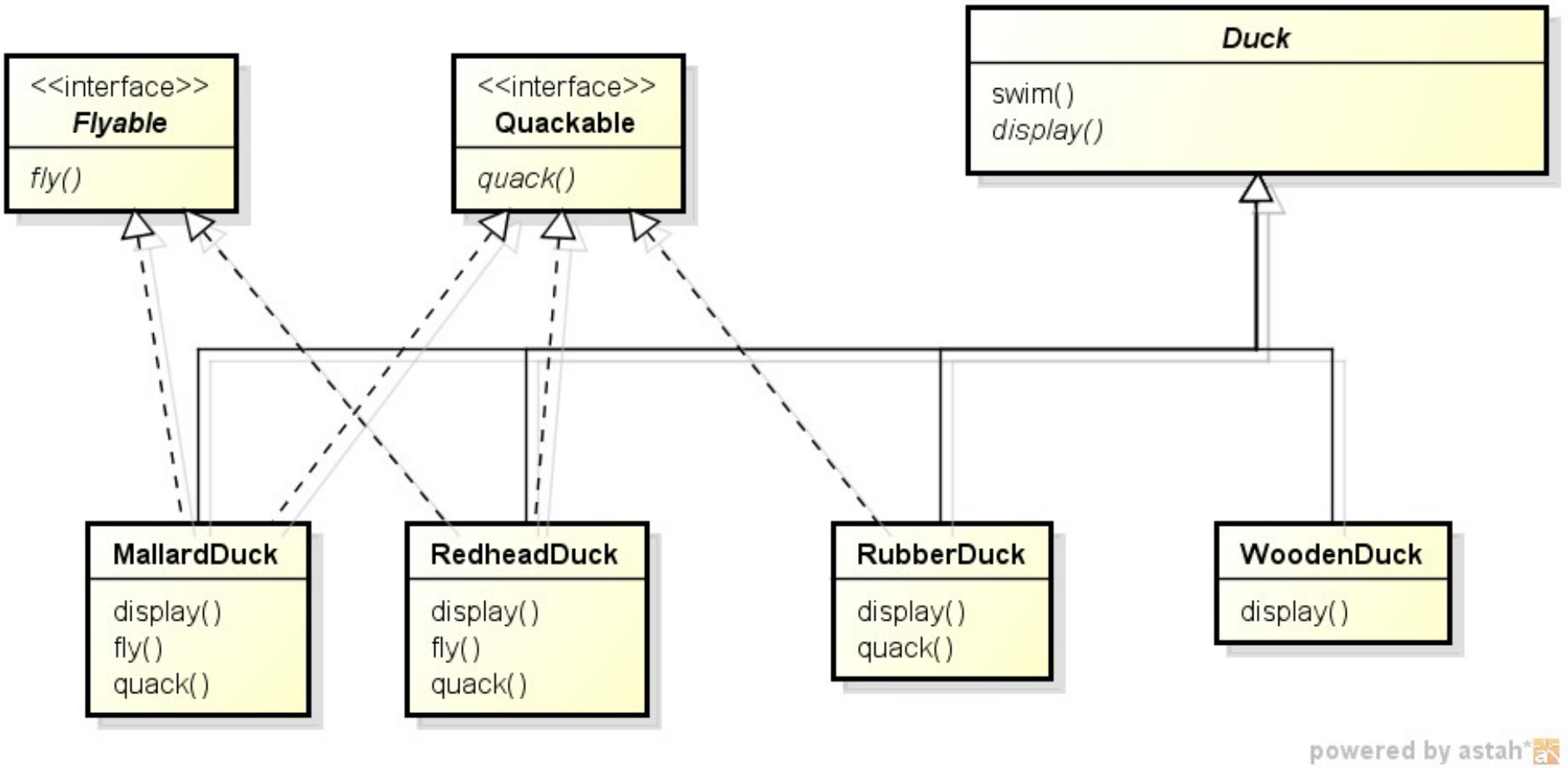
Algumas Desvantagens do Uso de Herança

- Quais problemas o uso de herança gera nesse caso?
  - Código duplicado entre as subclasses
    - métodos vazios
    - se tivéssemos outros tipos de patos de borracha com mesmo som, a implementação de método quack() seria duplicado em suas classes
  - As alterações podem afetar sem querer outros patos
  - É difícil conhecer o comportamento de todos os patos

Que tal usarmos interface?

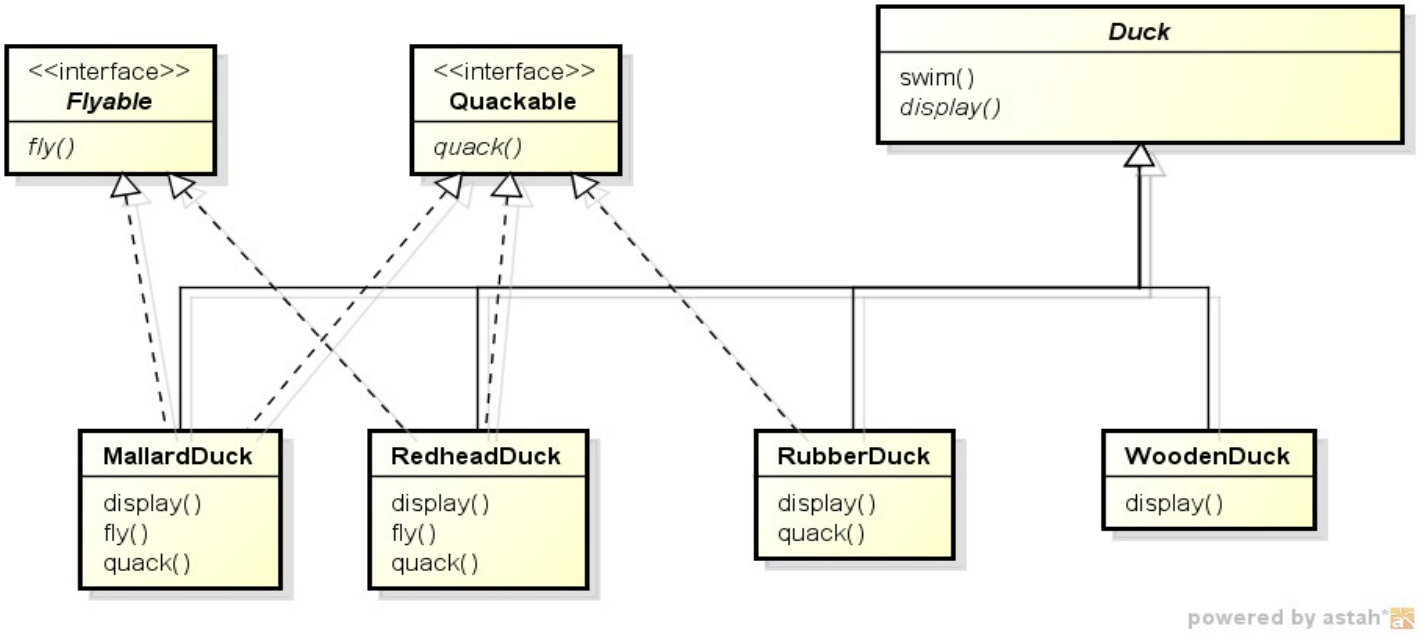
- Poderíamos tirar fly() da superclasse Duck e criar uma interface **Flyable** com o método fly(). Assim somente as classes dos patos que voam implementarão essa interface e terão o método fly().
- Da mesma forma, podemos criar uma interface **Quackable**, já que nem todos patos grasnam.

Poderia ser assim?



- A classes RubberDuck e WoodenDuck não precisam implementar fly()

Quais os problemas dessa solução?



- Perde-se a reutilização de código para os comportamentos de voar e grasnar
  - Muito código duplicado
- Se tivermos que alterar, por exemplo, o comportamento de vôo, teremos que alterar os métodos fly de várias classes (nesse caso são duas, mas podem ser dezenas).

A Certeza do Desenvolvimento de Software

- MUDANÇA!!!
  - Independente do software e de como ele seja desenvolvido, com o tempo ele precisará ser alterado para não "morrer".

Acomodando Mudanças

- No jogo de patos, o uso de herança não funcionou bem para acomodar as mudanças.
  - Nem todas as subclasses tem novos comportamentos.
- O uso das interfaces Flyable e Quackable prejudicou a reutilização de código.
  - Sempre que for modificar um comportamento terá que alterar todas as classes que tem esse comportamento.

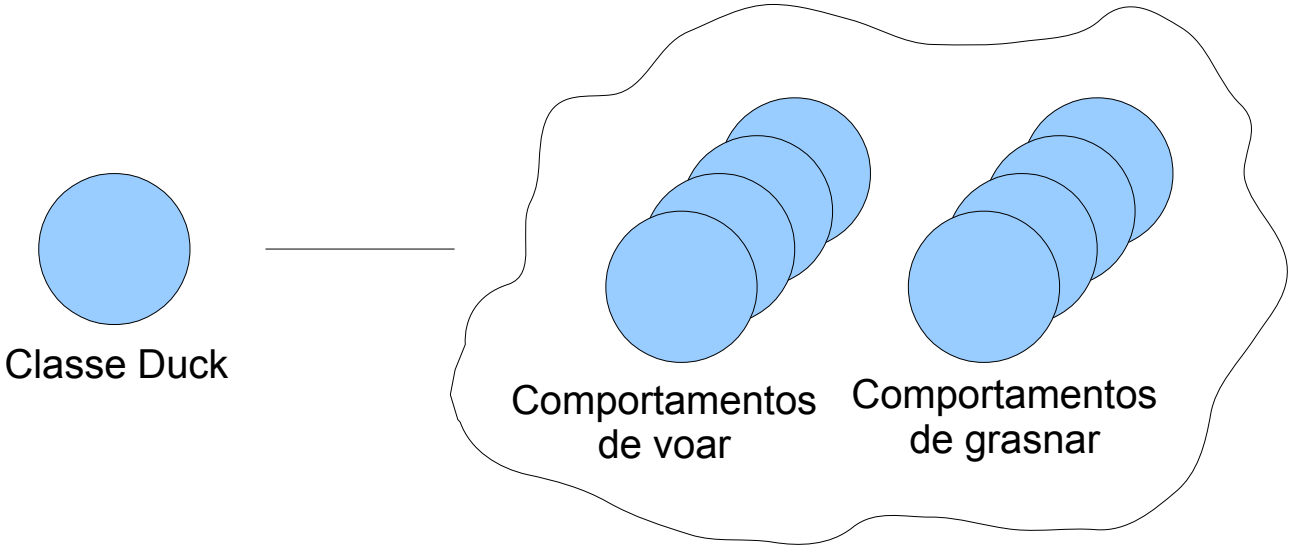
Um Princípio de Projeto

- Identifique os aspectos do sistema que variam e separe-os dos que não variam.
- Em outras palavras: pegue as partes que variam e encapsule-as para depois poder alterá-las ou estendê-las sem afetar as partes que não variam.
- Esse princípio forma a base da maioria dos padrões de projeto

O que varia?

- Nem sempre é fácil antecipar o que varia em um sistema.
  - algumas vezes só percebemos o que varia depois de algumas solicitações de mudanças
- Podemos perceber que no jogo de patos fly() e quack() são as partes da classe Duck que variam entre os patos

Separando fly() e quack()



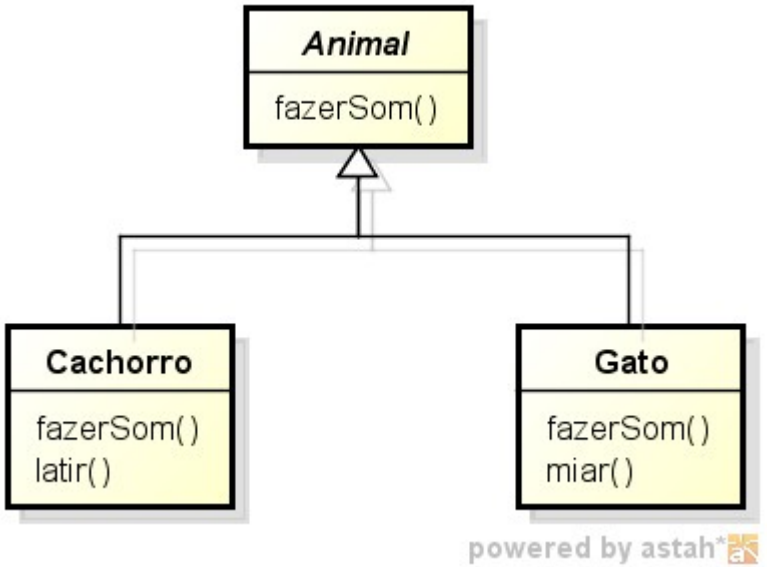
- A classe Duck ainda será a superclasse de todos os patos.
- Mas vamos tirar os comportamentos de voar e grasnar, colocando-os em outra estrutura de classes.
- Essa nova estrutura de classes terá diferentes implementações de voar e grasnar.

Outro Princípio de Projeto

- Programe para uma interface, não uma implementação.
  - Interface aqui não quer dizer interface de Java
    - Você pode programar para uma interface usando uma interface Java ou uma classe abstrata.
  - Poderia ser: programe para um **supertipo**.
  - O tipo declarado das variáveis deve ser um supertipo (interface ou classe abstrata).
  - Os objetos atribuídos às variáveis podem ter qualquer implementação concreta do supertipo
  - A classe que declara a variável não deve conhecer o tipo real do objeto.

Programar para Interface

- Considere o projeto abaixo:



A implementação do método fazerSom() da classe cachorro chama o método latir().

A implementação do método fazerSom() da classe gato chama o método miar().

Programar para Interface

- Nesse caso, **programar para implementação** seria:

```
Cachorro c = new Cachorro();
c.latir();
```

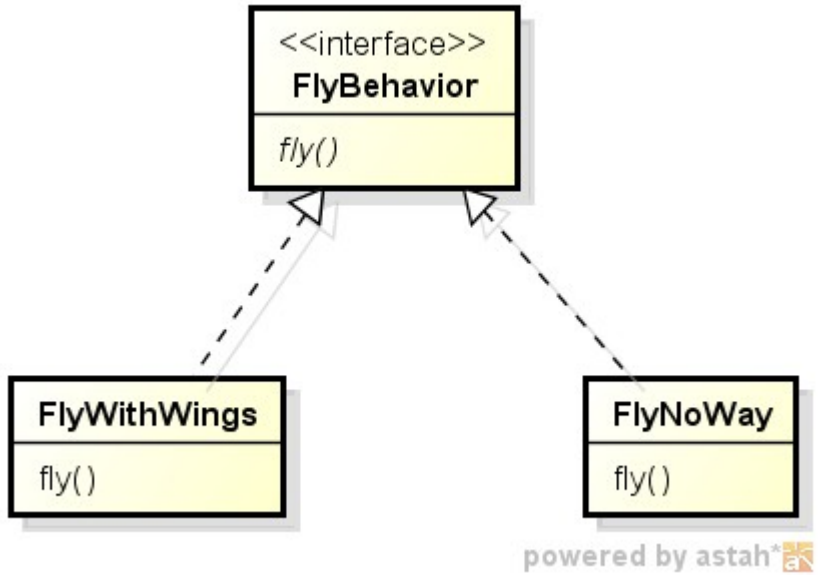
- **Programar para interface** seria:

```
Animal a = new Cachorro();
a.fazerSom();
```

- Melhor ainda seria:

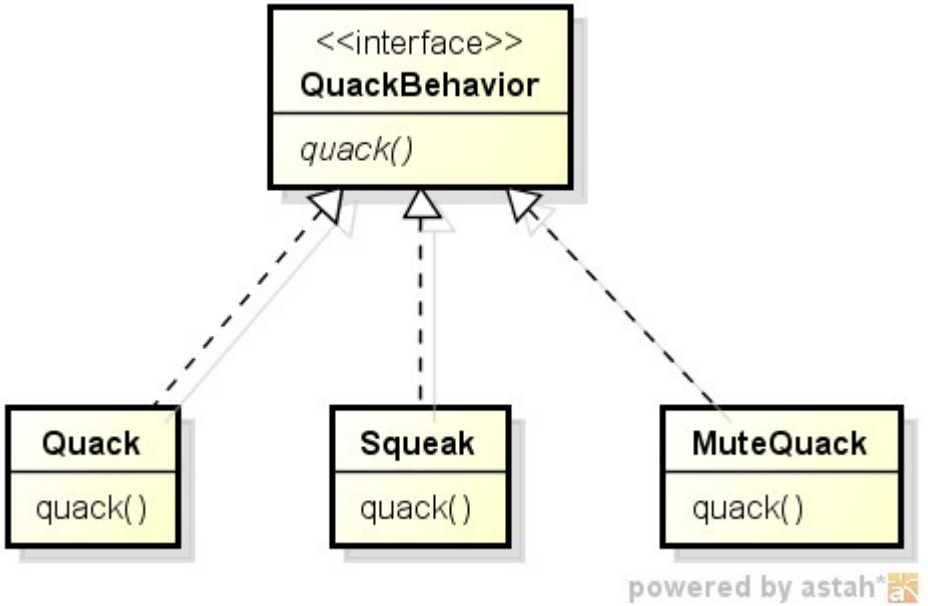
```
Animal a = fabrica.getAnimal();
a.fazerSom();
```

Comportamentos do Jogo de Patos



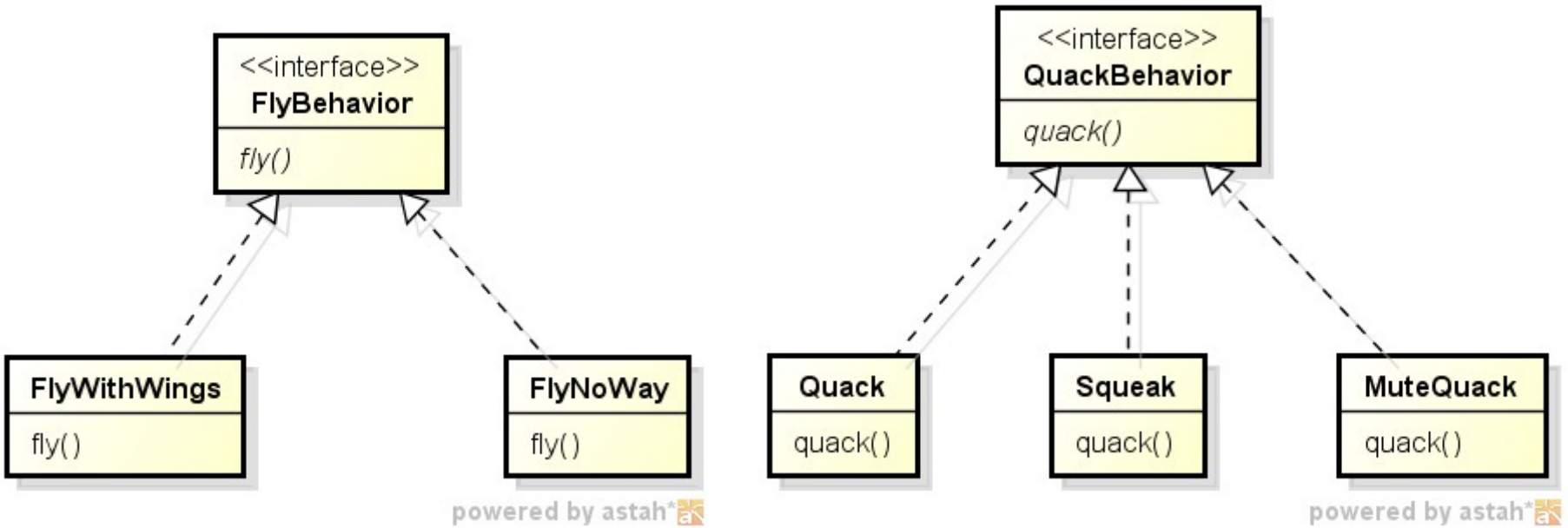
- FlyBehavior é a interface que as classes de vôos devem implementar.
- Todas as classes de vôo (inclusive novas) devem implementar essa interface e o método fly().
- O método fly() da classe FlyWithWings tem a implementação de voar para todos os patos que tem asas.
- O método fly() da classe FlyNoWay tem a implementação dos patos que não voam.

Comportamentos do Jogo de Patos



- Da mesma forma, QuackBehavior é a interface que as classes do comportamento de grasnar devem implementar.
- A classe Quack tem a implementação do grasnar normal.
- A classe Squeak tem a implementação do som de patos de borracha.
- A classe MuteQuack tem a implementação do grasnar para patos que não emitem som.

Comportamentos do Jogo de Patos

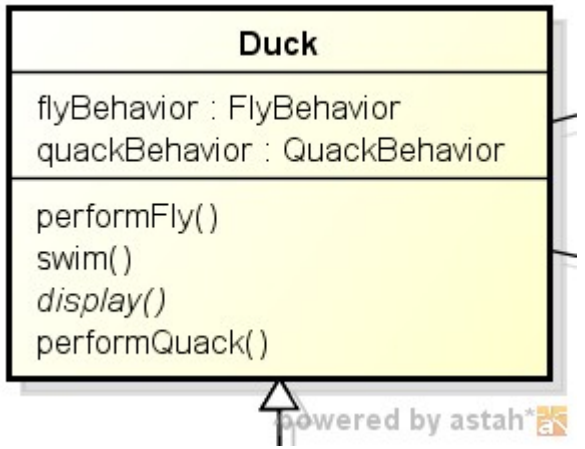


- A partir de agora, os comportamentos de voar e grasnar ficarão em classes separadas.
- Cada classe de comportamento implementa uma interface do comportamento.
- Assim, as classes de pato (Duck, MallarDuck etc) não precisarão conhecer os detalhes de implementação desses comportamentos.

Integrando os comportamentos com Duck

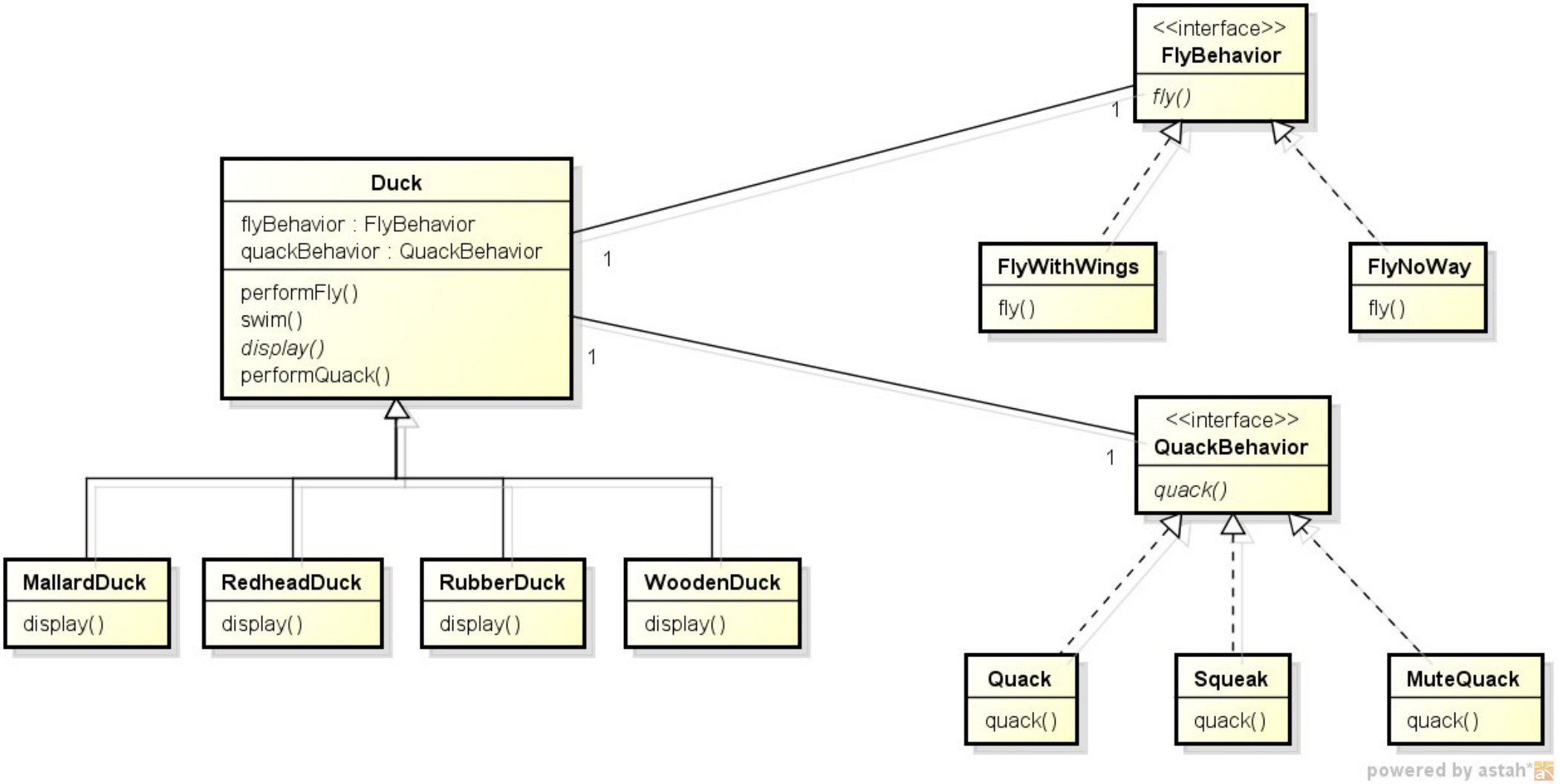
- A classe Duck irá agora **delegar** seu comportamento de voar e grasnar para as classes que definem os comportamentos.

Integrando os comportamentos com Duck



- Adicionamos dois atributos (variáveis de instância) à classe Duck chamadas de flyBehavior e quackBehavior, declaradas como o tipo das interfaces FlyBehavior e QuackBehavior, respectivamente.
- Em cada objeto de Duck, essas variáveis terão referências para instâncias das classes que implementam seu comportamento específico (FlyWithWings, Quack, etc).
- Substituimos os métodos fly() e quack() pelos métodos performFly() e performQuack().

Visão Geral


<<interface>>
FlyBehavior
fly()
1
Duck
flyBehavior : FlyBehavior
quackBehavior : QuackBehavior
performFly()
swim()
display()
performQuack()
1
1
FlyWithWings
fly()
FlyNoWay
fly()
<<interface>>
QuackBehavior
quack()
1
MallardDuck
display()
RedheadDuck
display()
RubberDuck
display()
WoodenDuck
display()
Quack
quack()
Squeak
quack()
MuteQuack
quack()
powered by astah*

Detalhes da Implementação

```
public abstract Class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public Duck() {
    }

    public abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }

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

    public void swim() {
        System.out.println("All ducks float, even wooden ones!");
    }
}
```

Detalhes da Implementação

```
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“);
    }
}
```

Detalhes da Implementação

```
public Interface FlyBehavior {
    public void fly();
}

public class FlyWithWings implements FlyBehavior {
    public void fly() {
        System.out.println(“I'm flying“);
    }
}

public class FlyNoWay implements FlyBehavior {
    public void fly() {
        System.out.println(“I can't fly“);
    }
}
```

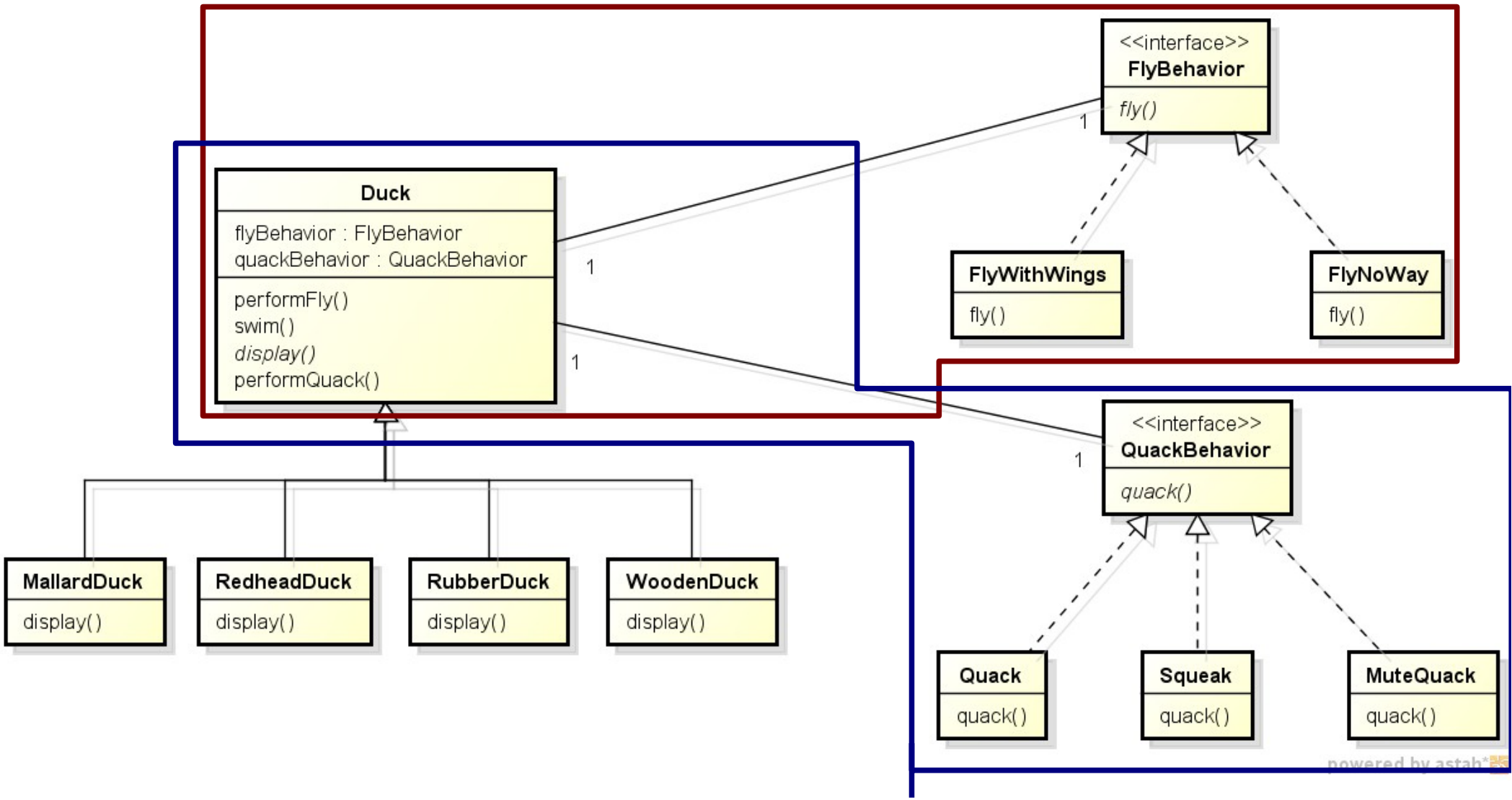
Definindo comportamentos em tempo de execução

```
public Class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public Duck {}

    public abstract void display();

    public void performFly {
        flyBehavior.fly();
    }

    public void performQuack {
        quackBehavior.quack();
    }

    public void swim() {
        System.out.println("All ducks float, even wooden ones!");
    }

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

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

Temos um Padrão de Projeto

- Padrão Strategy
  - Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. A estratégia deixa o algoritmo varias independentemente dos clientes que o utilizam.

Padrão de Projeto Strategy



Duas ocorrências do padrão Strategy