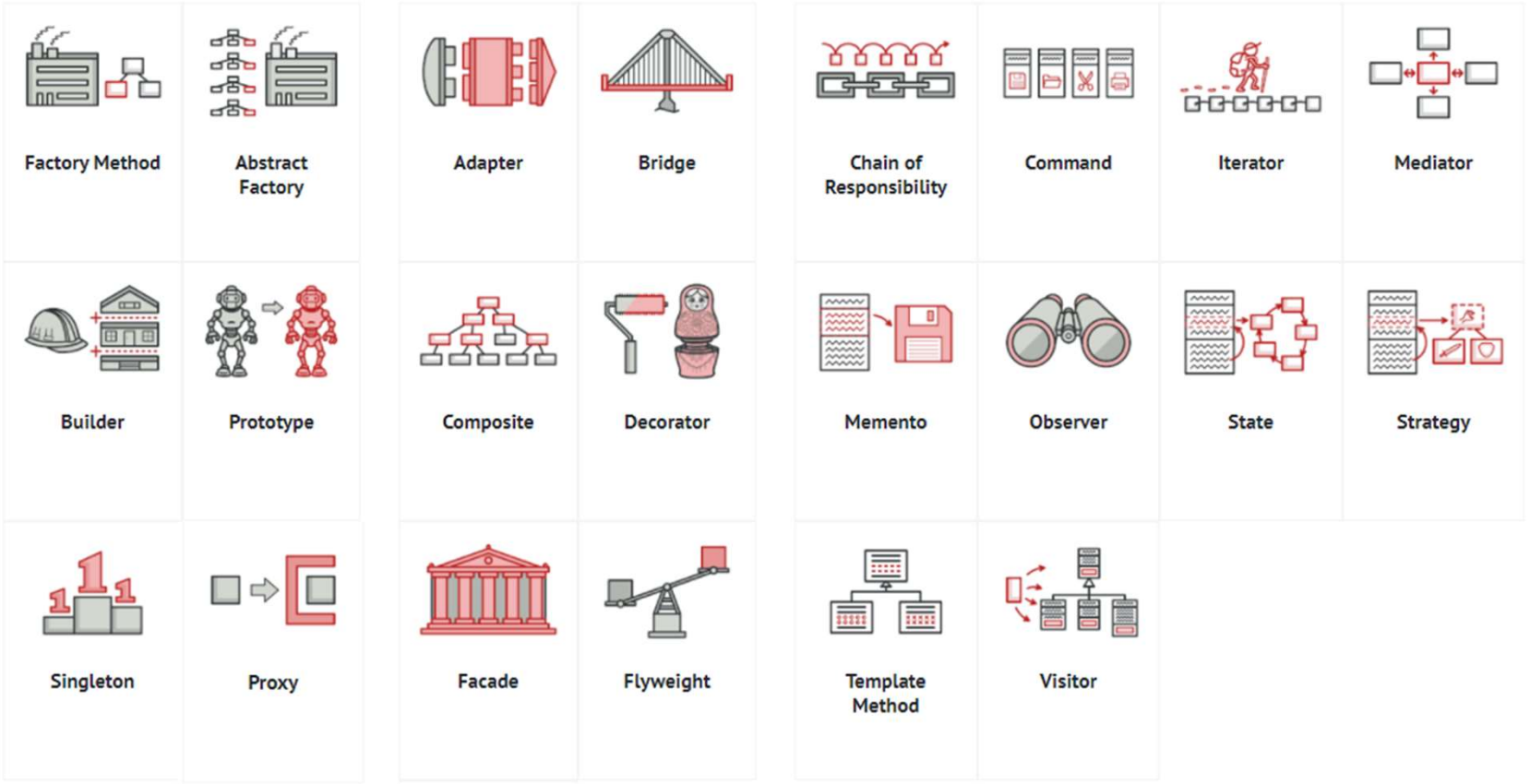
Design Pattern

are typical solutions to common problems in software design.

Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

https://refactoring.guru/design-patterns

Pattern Overview



Classification

- Design patterns
  - differ by their complexity
    level of detail and scale of applicability.
  - they can be categorized by their intent
    and divided into three groups:

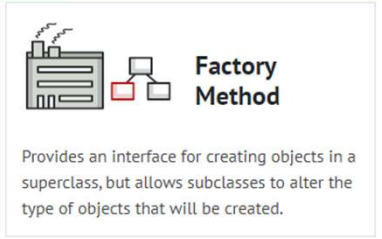
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns

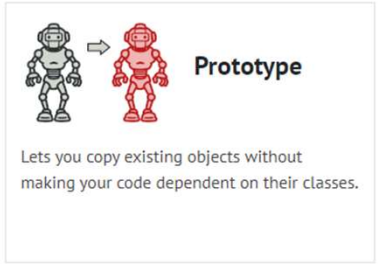
Benefits of patterns

- Patterns are
  - are a toolkit of **tried and tested solutions** to common problems in software design.
    - knowing patterns is useful because it teaches you how to solve all sorts of problems using principles of object-oriented design
  - Define a common language
    - Can be use to communicate more efficiently.
    - You can say, “Oh, just use a Singleton for that,” and everyone will understand the idea behind your suggestion.

What does the pattern consist of?

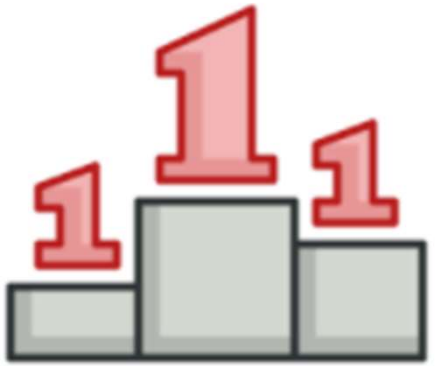
- Sections that are usually present in a pattern description:
  - **Intent** of the pattern briefly describes both the problem and the solution.
  - **Motivation** further explains the problem and the solution the pattern makes possible.
  - **Structure** of classes shows each part of the pattern and how they are related.
  - **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern.





Creational Design Patterns

provide various object creation mechanisms, which increase flexibility and reuse of existing code.

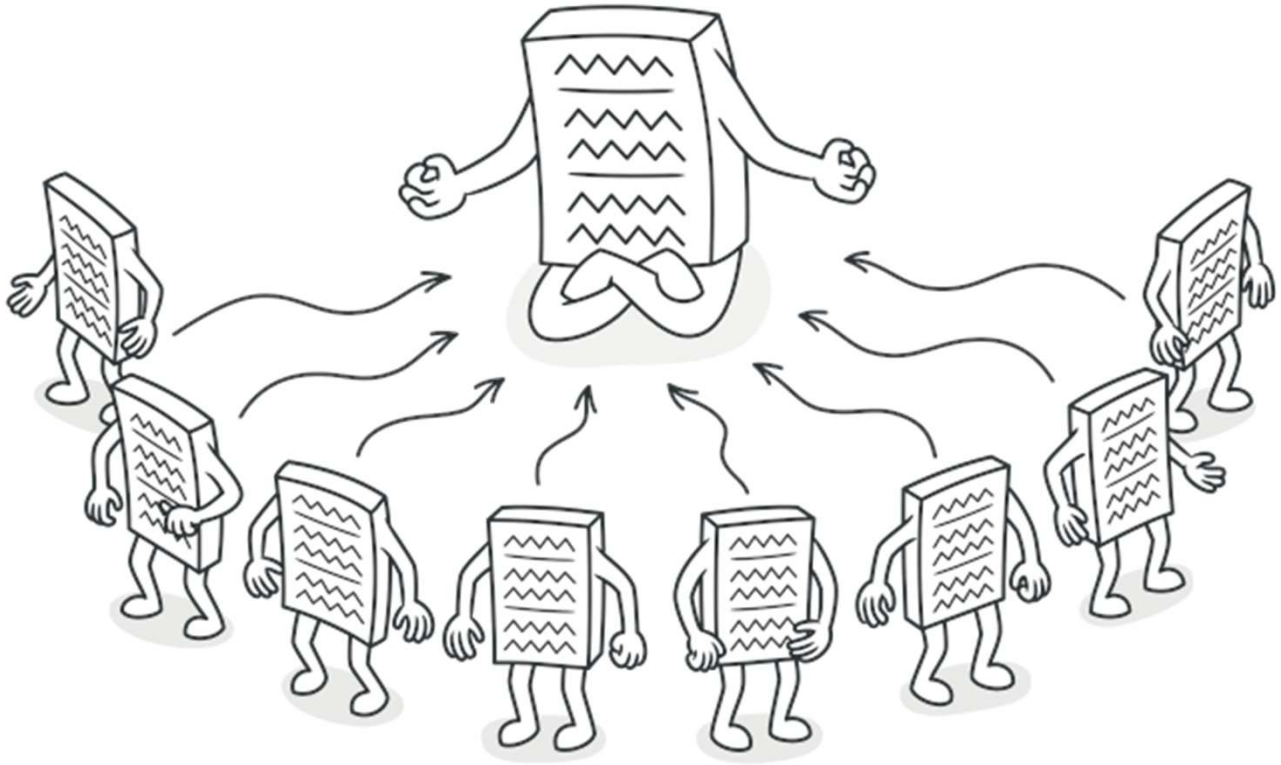


Singleton Pattern

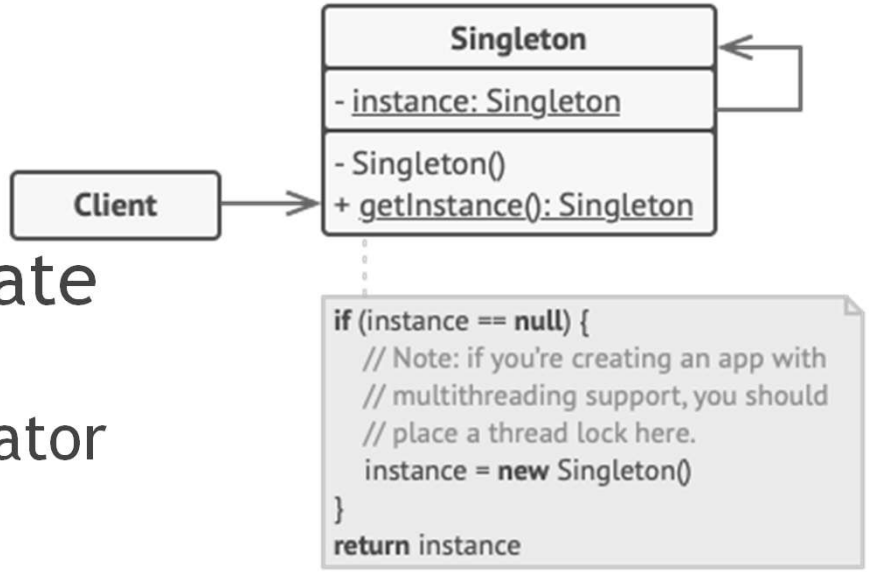
ensures that a class has only one instance

Singleton

- ensures that a class has only one instance
- providing a global access point to this instance



Singleton

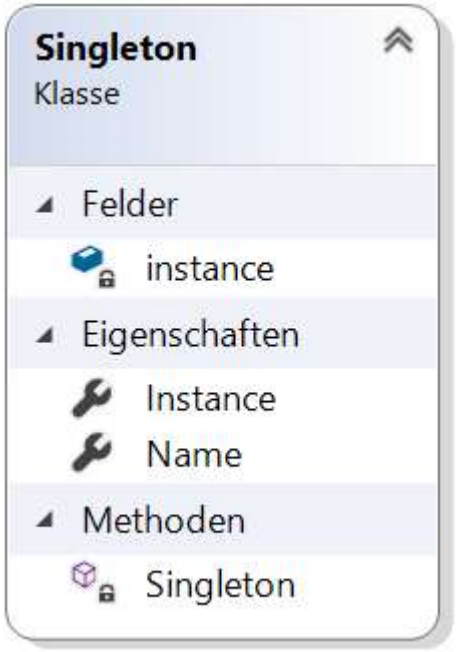


- default constructor private
  - to prevent other objects from using the new operator
- create a static creation method
  - that acts as a constructor
  - this method/property calls the private constructor to create an object and saves it in a static field
  - all following calls to this method return the cached object

Implementation Singleton

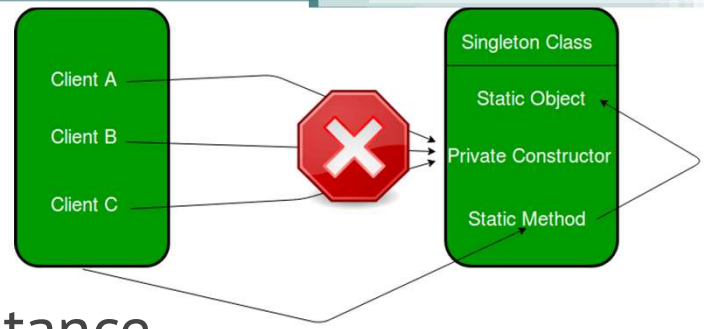
```
class Singleton
{
    2 Verweise
    public string Name { get; set; }
    1-Verweis
    private Singleton() { }
    private static Singleton instance = null;

    1-Verweis
    public static Singleton Instance {
        get {
            if (instance == null)
            {
                instance = new Singleton();
            }
            return instance;
        }
    }
}
```

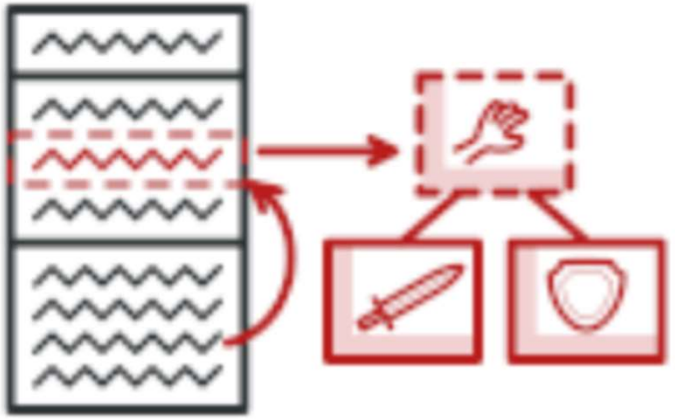



```
Singleton myInstance = Singleton.Instance;
myInstance.Name = "Kurt";
Console.WriteLine(myInstance.Name);
```

How to Implement



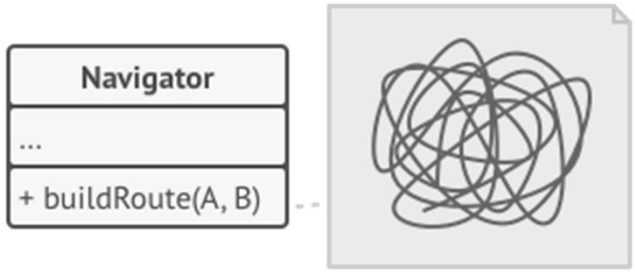
1. Add a private static field to the class for storing the singleton instance.
2. Declare a public static creation method for getting the singleton instance.
3. Implement “lazy initialization” inside the static method. It should create a new object on its first call and put it into the static field. The method should always return that instance on all subsequent calls.
4. Make the constructor of the class private. The static method of the class will still be able to call the constructor, but not the other objects.
5. Go over the client code and replace all direct calls to the singleton’s constructor with calls to its static creation method.



Strategy Pattern

lets you define a family of algorithms,
put each of them into a separate class,
and make their objects interchangeable

Problem

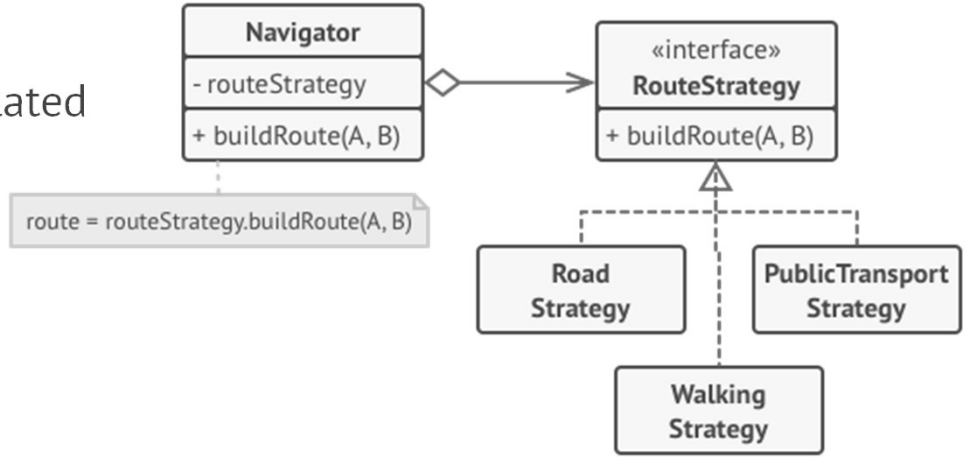


A story which describes the problem:

- One day you decided to create a navigation app for casual travelers. The app was centered around a beautiful map which helped users quickly orient themselves in any city.
- One of the most requested features for the app was automatic route planning. A user should be able to enter an address and see the fastest route to that destination displayed on the map.
- The first version of the app could only build the routes over roads. People who traveled by car were bursting with joy. But apparently, not everybody likes to drive on their vacation. So with the next update, you added an option to build walking routes. Right after that, you added another option to let people use public transport in their routes.
- However, that was only the beginning. Later you planned to add route building for cyclists. And even later, another option for building routes through all of a city's tourist attractions.
- While from a business perspective the app was a success, the technical part caused you many headaches. Each time you added a new routing algorithm, the main class of the navigator doubled in size. At some point, the beast became too hard to maintain.
- Any change to one of the algorithms, whether it was a simple bug fix or a slight adjustment of the street score, affected the whole class, increasing the chance of creating an error in already-working code.
- In addition, teamwork became inefficient. Your teammates, who had been hired right after the successful release, complain that they spend too much time resolving merge conflicts. Implementing a new feature requires you to change the same huge class, conflicting with the code produced by other people.

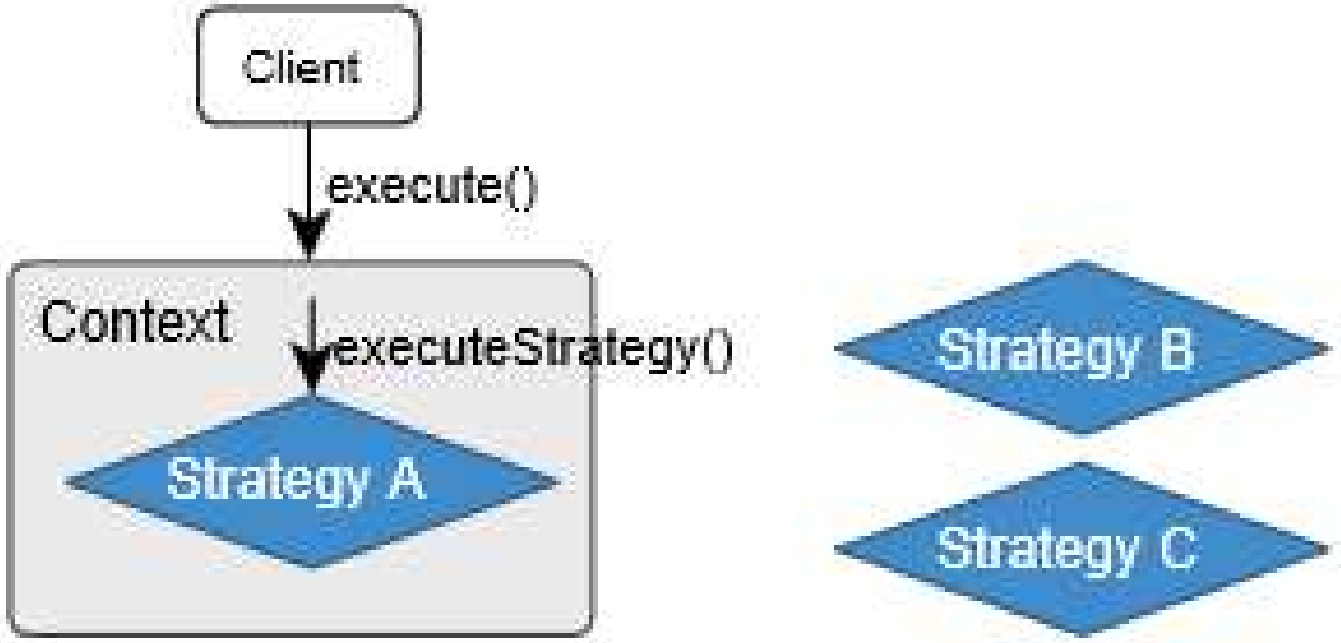
Solution

- Strategies
  - take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes
- *Context*
  - must have a field for storing a reference to one of the strategies
  - delegates the work to a linked strategy object instead of executing it on its own
  - context isn't responsible for selecting an appropriate algorithm for the job, it passes the desired strategy to the context
  - context doesn't know much about strategies
  - context works with all strategies through the same generic interface, which only exposes a single method for triggering the algorithm encapsulated within the selected strategy
  - context becomes independent of concrete strategies



Strategy Pattern

- Behavior is independable of the context
- context is independable of the imlementation

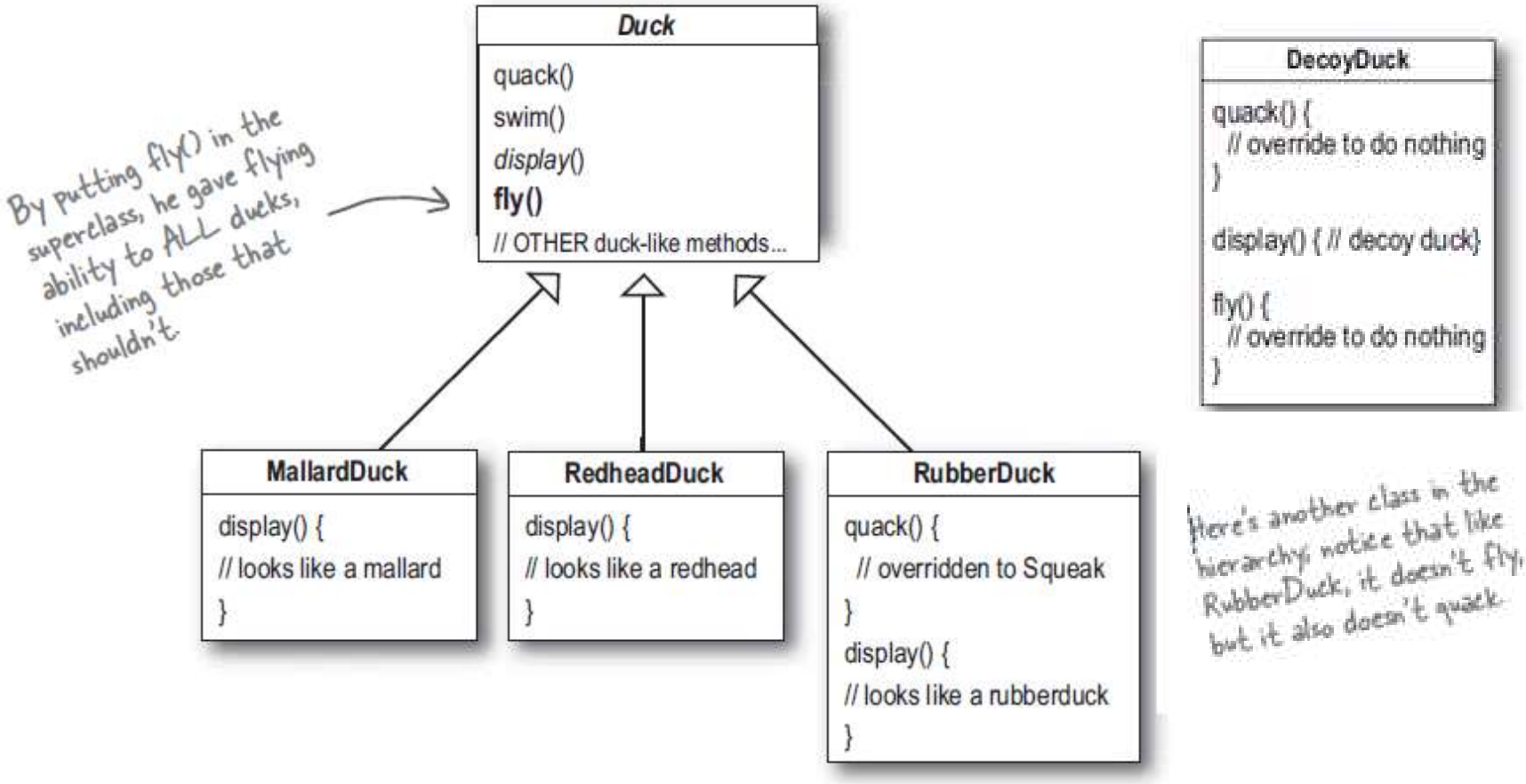


Class Diagramm

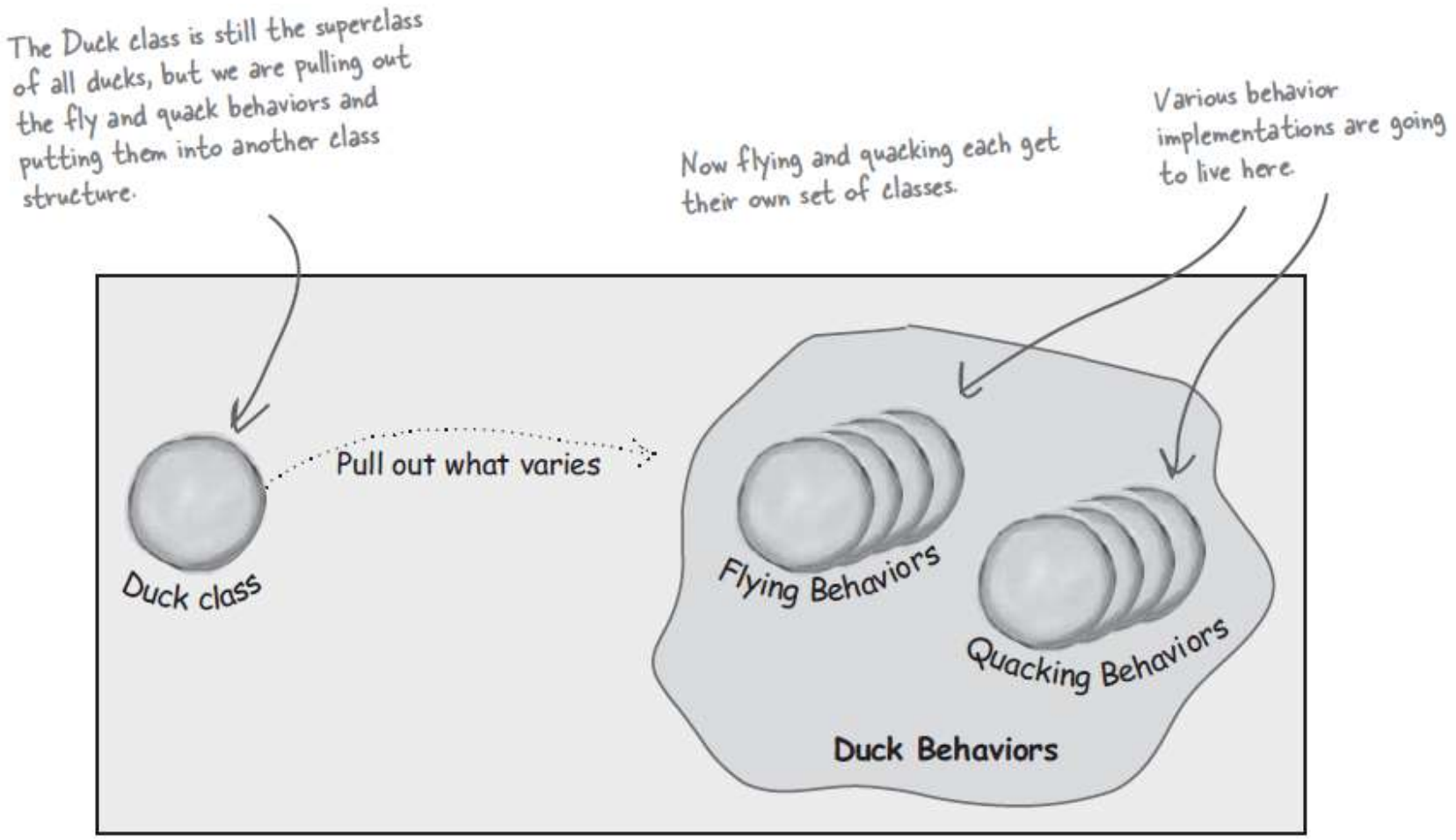
- IBehaviour (Strategy)
  - an interface that defines the behavior
- Conctete Strategies:
  - each of them defines a specific behavior.
- Context class
  - It keeps or gets context information and passes necessary information to the Strategy class

Strategy with Ducks

- Ducks can Swim, most of them can Quack, some of them can Fly.

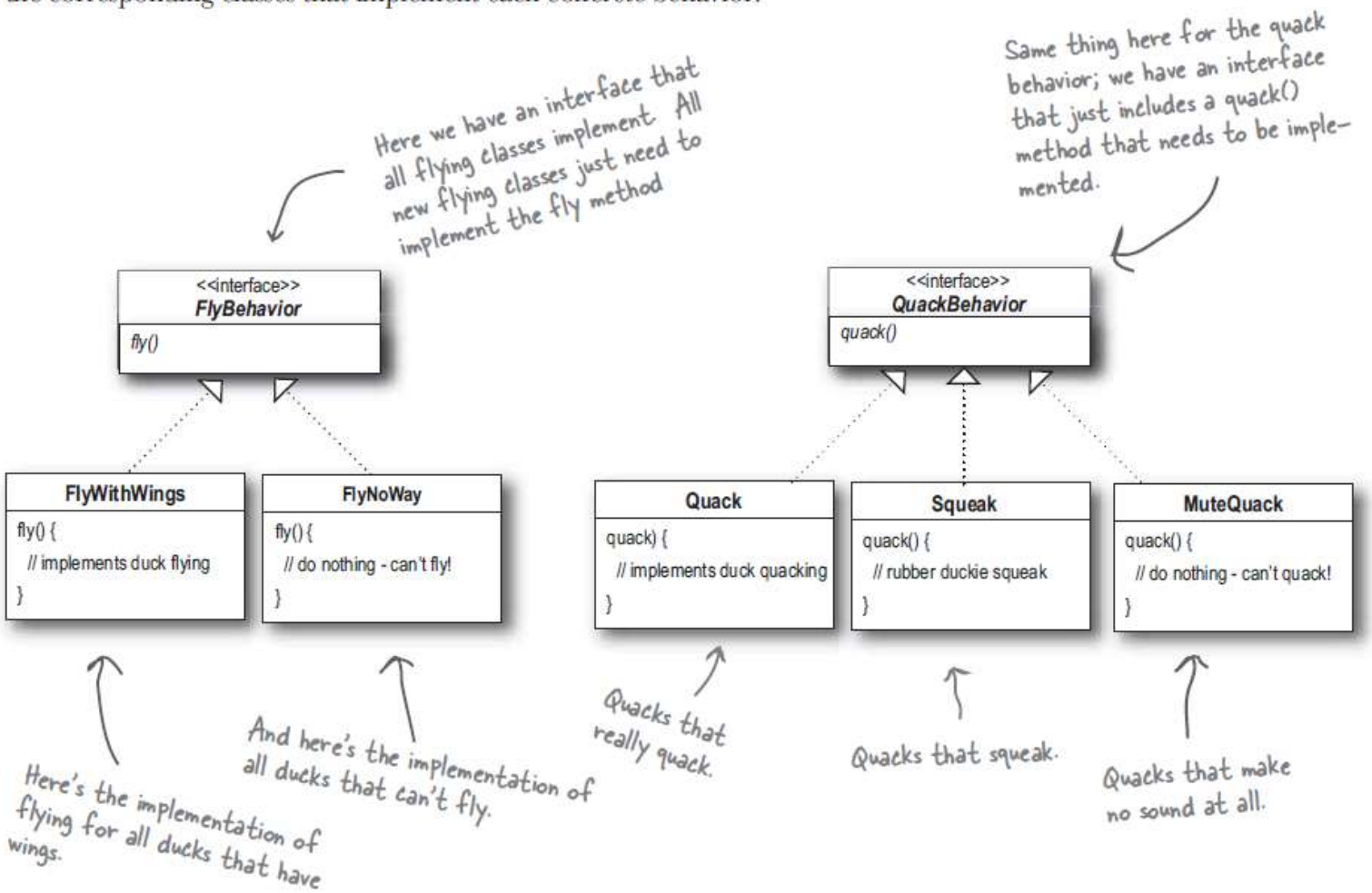


Strategy Solution

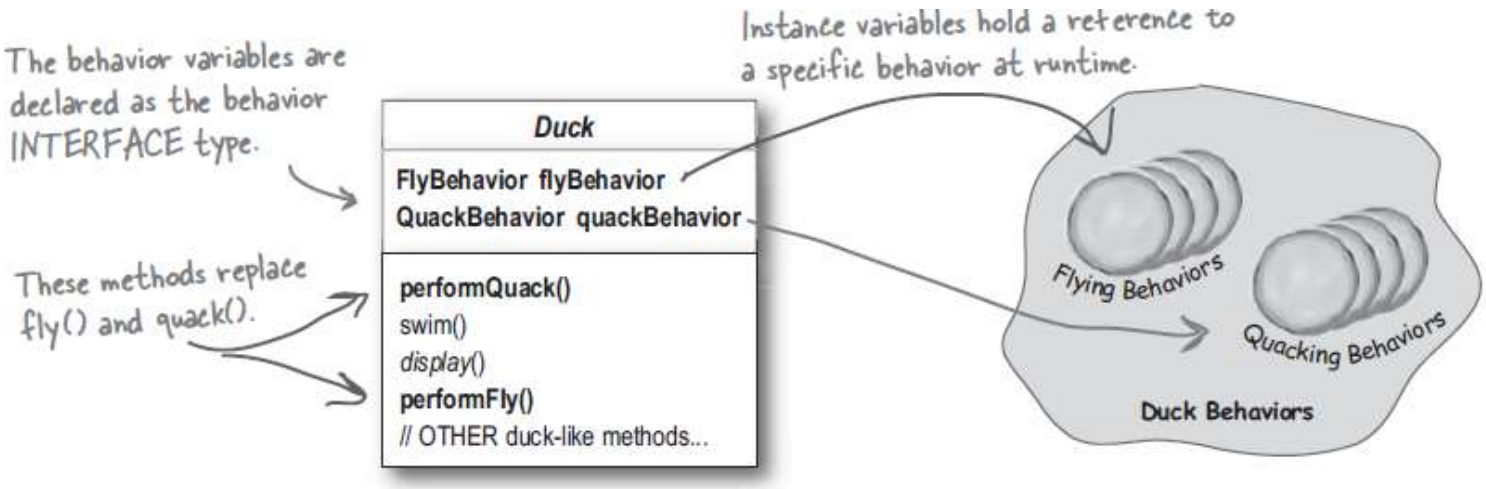


IFlyBehavior & IQuackBehavior

Here we have the two interfaces, FlyBehavior and QuackBehavior along with the corresponding classes that implement each concrete behavior:



Implementing Quack-Method



Now we implement performQuack():

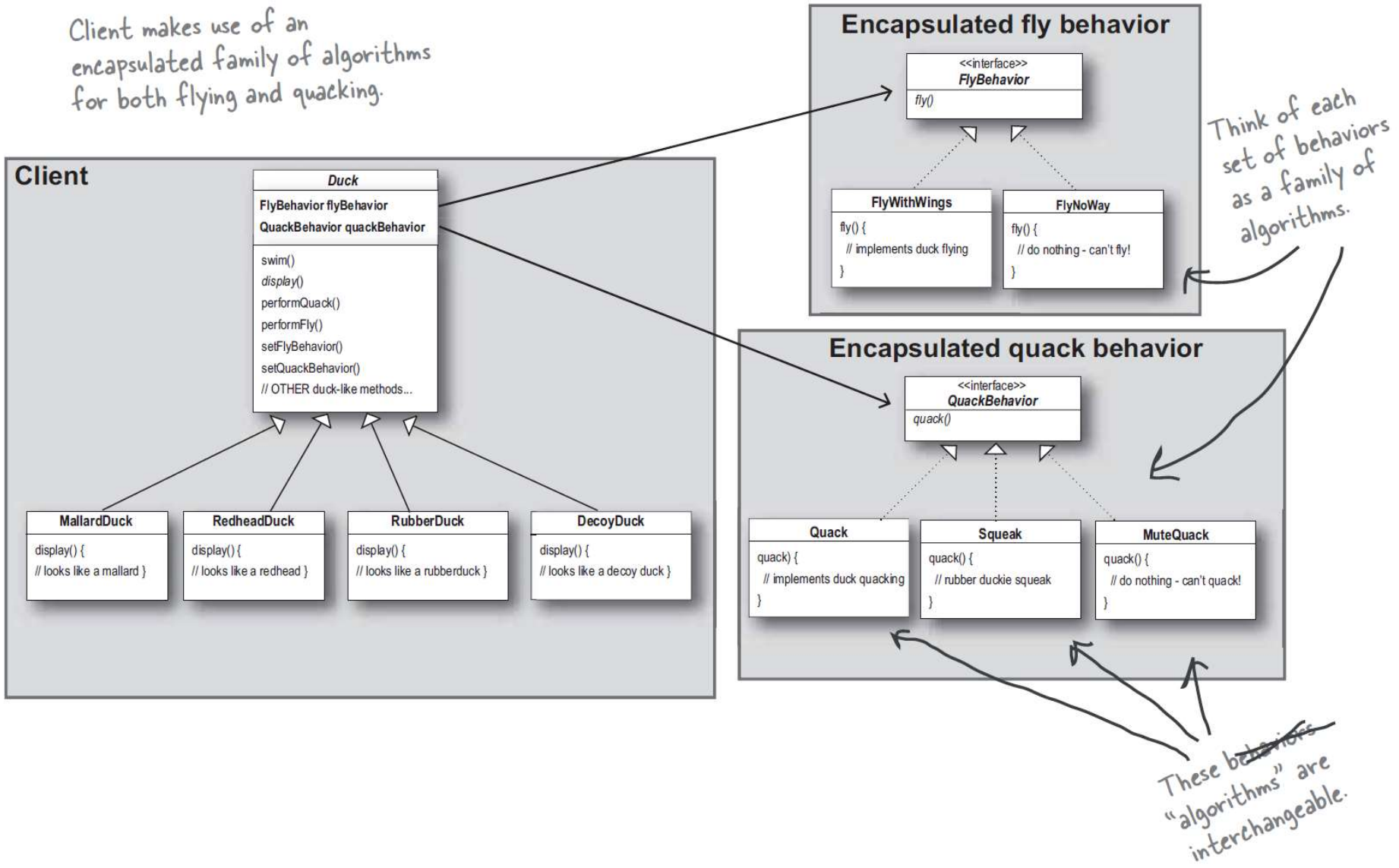
```
public class Duck {
   QuackBehavior quackBehavior;
   // more

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

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.

Strategy Pattern - Ducks



Abstract Duck

```
public abstract class ADuck {
    private readonly IFlyBehavior flyBehavior;
    private readonly IQuackBehavior quackBehavior;
    protected ADuck(IFlyBehavior flyBehavior, IQuackBehavior quackBehavior) {
        this.flyBehavior = flyBehavior;
        this.quackBehavior = quackBehavior;
    }

    public abstract void Display();

    public void PerformFly() {
        this.flyBehavior.Flying();
    }

    public void PerformQuack() {
        this.quackBehavior.Quacking();
    }

    public void Swim() {
        Console.WriteLine("All ducks float, even decoys!");
    }
}
```

Quack Behavior

```
public interface IQuackBehavior {
    void Quacking();
}

public class Quack : IQuackBehavior {
    public void Quacking() {
        Console.WriteLine("Quack");
    }
}

public class MuteQuack : IQuackBehavior {
    public void Quacking() {
        Console.WriteLine("<<  Silence  >>");
    }
}

public class Squeak : IQuackBehavior {
    public void Quacking() {
        Console.WriteLine("Squeak");
    }
}
```

Fly Behavior

```
public interface IFlyBehavior {
    void Flying();
}

public class FlyWithWings : IFlyBehavior {
    public void Flying() {
        Console.WriteLine("I'm flying!");
    }
}

public class FlyNoWay : IFlyBehavior {
    public void Flying() {
        Console.WriteLine("I can't fly!");
    }
}
```

Mallard Duck & Decoy Duck

```
public class MallardDuck : ADuck {
    public MallardDuck() : base(new FlyWithWings(), new Quack()) {
    }

    public override void Display() {
        Console.WriteLine("I'm a real Mallard duck");
    }
}
```

```
public class DecoyDuck : ADuck {
    public DecoyDuck() : base(new FlyNoWay(), new MuteQuack()) {
    }

    public override void Display() {
        Console.WriteLine("I'm a real decoy duck");
    }
}
```

Using Ducks

```
public class MiniDuckSimulator {
    public static void Main(string[] args) {
        ADuck mallard = new MallardDuck();
        mallard.PerformQuack();
        mallard.PerformFly();
        mallard.Swim();
        Console.ReadLine();

        ADuck decoy = new DecoyDuck();
        decoy.PerformQuack();
        decoy.PerformFly();
        decoy.Swim();
        Console.ReadLine();
    }
}
```

```
Quack
I'm flying!
All ducks float, even decoys!

<<  Silence  >>
I can't fly!
All ducks float, even decoys!
```