

**My Notes for Design Pattern**

Contents

[1. Strategy Pattern 2](#_Toc155737570)

[What it is? 2](#_Toc155737571)

# Strategy Pattern

## What it is?

**Strategy** is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

## How to Detect?

Basically see if a specific task is being solved by different methods.

## Problem

I made a middleware which let me authenticate user using “email and password” and then I added a feature to authenticate user using “Oauth” too and then another feature of inserted to authenticate user using “Pass code”… This causes [Authenticate class] to have double code each time another method of authentication is added.

## Solution

Make a class Authentication Context and create a member named authStrategy which is instance of an interface

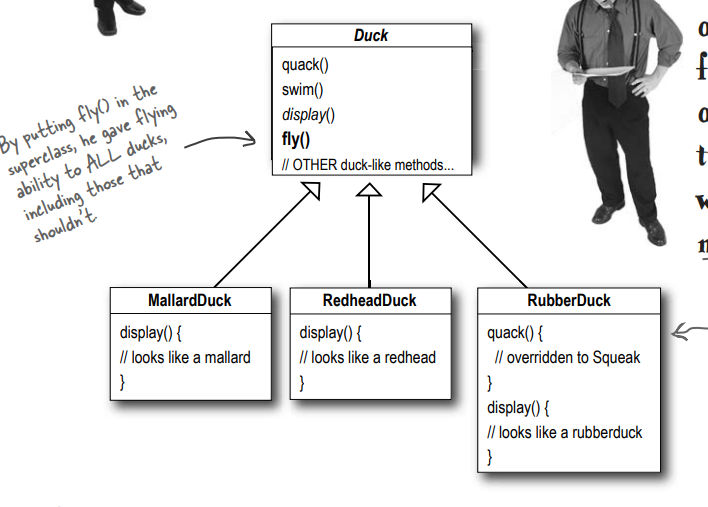
## Other Point of Views

Using of Composition and Interfaces for inheritance instead of Inheritance from Class.

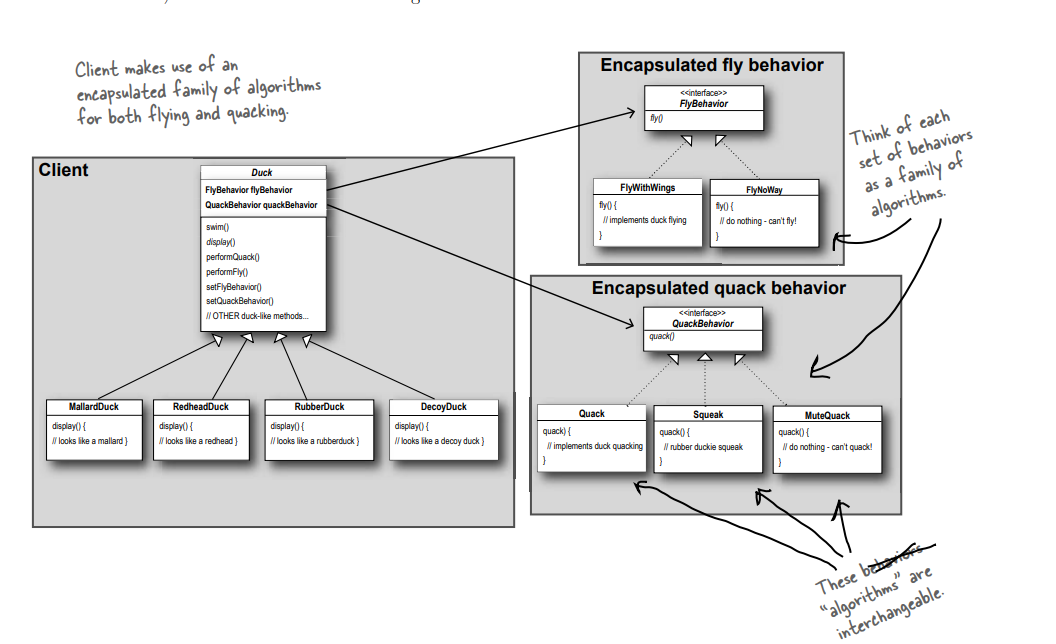
“The Solution to Problems with Inheritance, is certainly not more inheritance”

## Duck Problem

Problem



Solution:



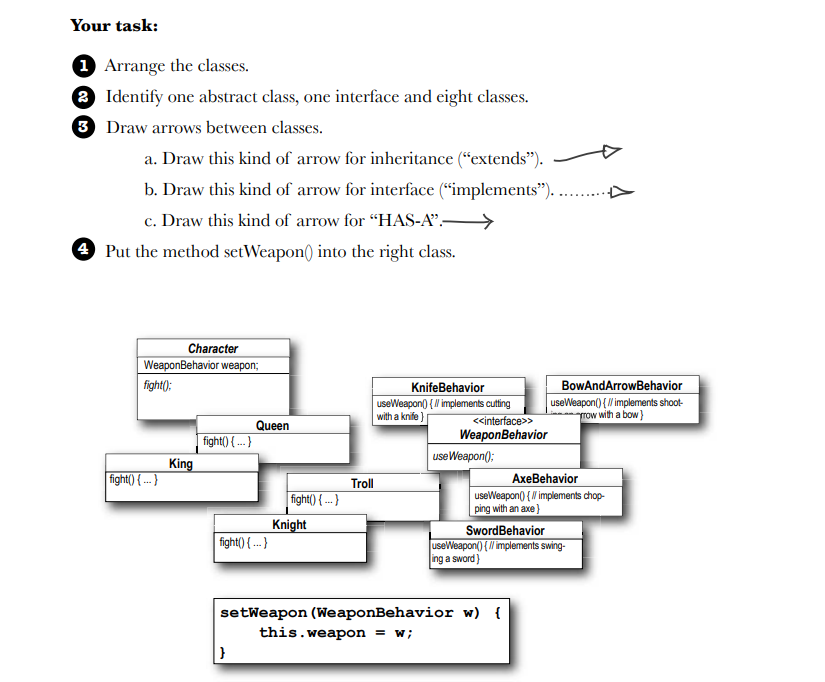
## Design Principles That were Used

* Identify the behavior of code that can change time to time and separate it from the code that stays the same.
* Program to Interface/supertype, not to an implementation (i.e. using Animal type variable to store Dogs and Cats Objects)
* Favor Composition over Inheritance (As it gives more flexibility, like storing set of behaviors as a family of algorithms and changing behavior at run time).

## Real Life Problems for Strategy Pattern

* Sorting: We want to sort these numbers, but there can be different sorting techniques suggested at Runtime (Like if the small data then use Selection Sort, Large Data but almost sorted use Bubble Sort, Memory Constraint use Insertion Sort, Random Access is not provided and it is linked list use Merge Sort, Have Random Access Memory and Memory is not an issue use QuickSort)
* Validation: We need to check items according to some rules, but it’s not yet clear what that rule will be and we may think of new ones.
* Games: We want player to either walk or run when he moves but may be in the future he should also be able to swim, fly, teleport, burrow underground etc.
* Storing Information: WE want the application to store information to the Database, but later it may need to be able to save a file. Or make a webcall.
* Outputting: We need to output X as plain string, but later may be CSV, XML, JSON, etc.
* Many More….

## Exercises from Head First Design Pattern



A diagram of a computer

Description automatically generated with medium confidence

# Observer Pattern

## What it is?

**Observer** is a behavioral design pattern that let list of objects know that there has been a change made in an instance (Basically it’s telling other objects that instance have changed instead of asking the instance again and again that if it has changed or not), Whenever change is made notify the objects.

## How to Detect?

If Object needs to check state of another object and if a change is made in targeted object than the object needs to be changed.

## Problem

Let’s suppose there is an object of class “Observable”, and there is another object of class “Observer”. In order to check Observable state change Observer object needs to have a loop on Observable and check it’s state again and again. The more the objects of Observer the more slower the code gets.

## Solution

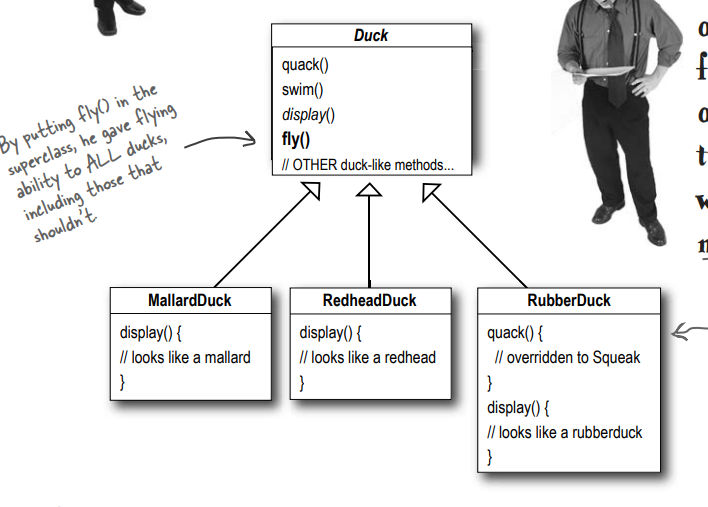
Instead of checking the Observable again and again, why not just Notify Observer that there has been a change in Observable. Who does that, right? Observable itself. Design will be given in Problem Section, But Main idea is to notify (Pushing instead of Polling).

## Other Point of Views

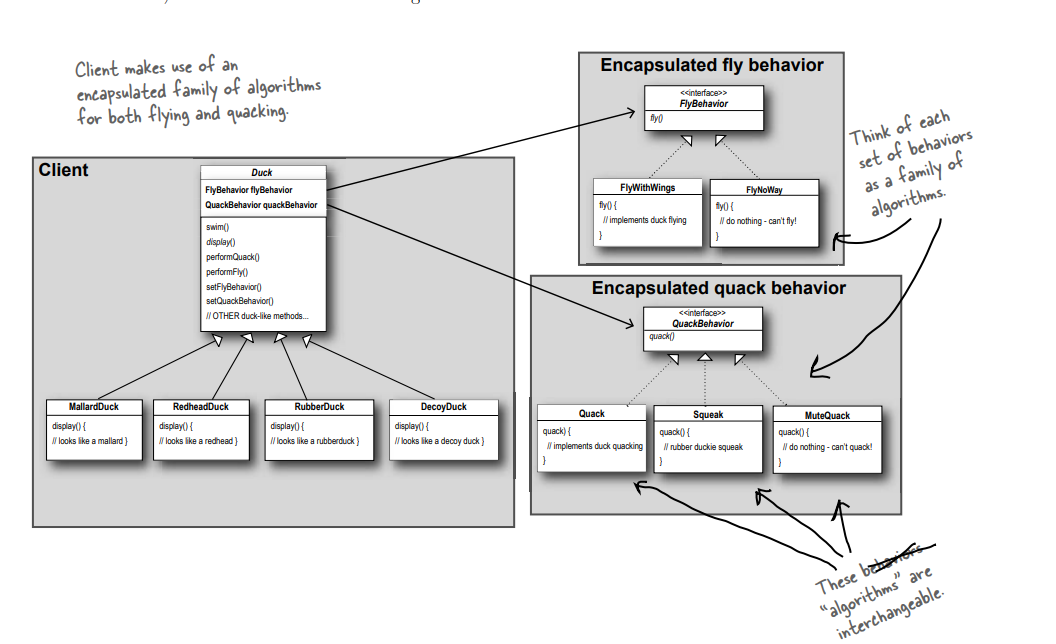
**Observer** is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.

## Duck Problem

Problem



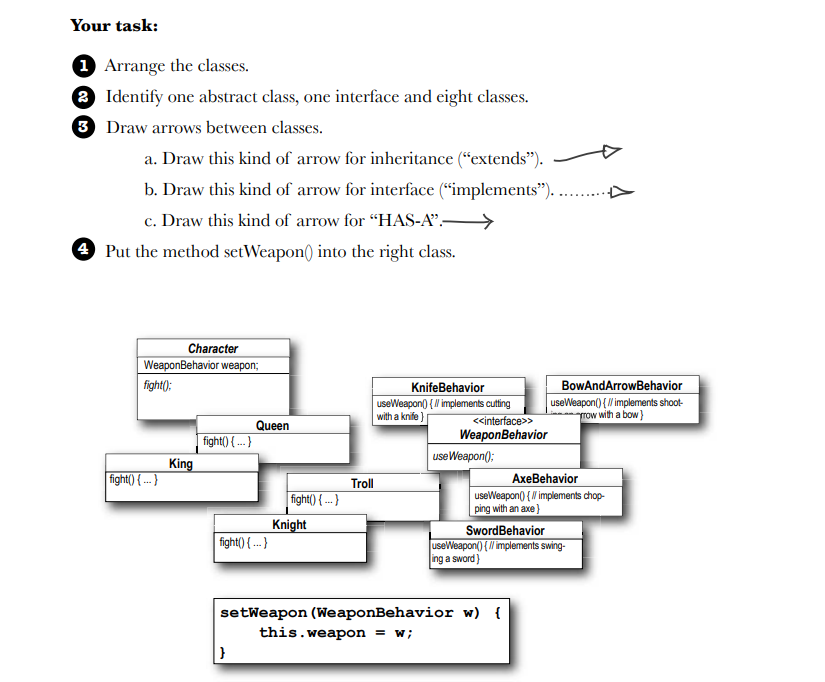
Solution:



## Design Principles That were Used

* Strive for loosely coupled designs between objects that interact.
* Program to Interface/supertype, not to an implementation (i.e. using Animal type variable to store Dogs and Cats Objects, it gives run time flexibility)
* Favor Composition over Inheritance (As it gives more flexibility, like storing set of behaviors as a family of algorithms and changing behavior at run time) .

## Exercises from Head First Design Pattern



A diagram of a computer

Description automatically generated with medium confidence