# Strategy Pattern

Preslav Gerchev 2487136  
Dimitar Vikentiev 2464020

# Introduction

In week 1 we had to develop a project regarding disc scheduling applying the Strategy Pattern.  
The following document will reflect on the project and on the pattern itself- positive and negative consequences, reusability, maintainability, extensibility.

# The Pattern

The formal definition for the pattern is as following**:**

**“Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it”**

Using that pattern with the disc scheduling gives us the possibility of changing the strategy at runtime and also improves reusability, extensibility, maintainability (described below).

# Reusability

In terms of reusability the design patterns does pretty well. The algorithms can be reused throughout the whole project because they are clearly separated from the context and are not hidden in it. Another benefit is that they can be used by classes that are not necessarily logically linked (the whole purpose of using interfaces is exactly that). An example for that benefit would be the QuackBehaviour interface being used by both the Duck and the DuckSimulator class (logically both classes have nothing in common).

# Extensibility

In terms of extensibility the strategy pattern is very good.

It allows for adding new classes to the family of algorithms and extending the client’s (context) behavior without changing the already existing algorithms and their implementation or changing the client’s implementation. I will give an example with the Ducks – we could add a thousand different FlyBehaviour implementations and that wouldn’t affect the existing code in any way (besides the downside of having 1000 different classes).

# Maintainability

The strategy patterns helps for improving the code’s maintainability. Since the algorithms are interchangeable and the pattern relies on aggregation rather than inheritance there is no chance that a change in the code of one algorithm leads to unexpected changes (something that might happen if inheritance were used). Using strategy pattern removes the usage of *if-else* statements, leads to less code in the client which by itself increases the maintainability.

# Others

One of the negative consequences of using the Strategy Pattern is the huge amount of classes needed. If we were to have 100 different implementations of an algorithm that would mean 100 new classes. And the client would need to be aware of all the different strategies.

A big plus in implementing the Strategy Pattern is the possibility of the changing the client’s behavior at *runtime.* Something that isn’t possible using inheritance (you can’t inherit from another class at runtime, but you can change the context’s interface reference to another interface’s implementation).

# Implementation

We decided to use Microsoft’s LINQ (Language Integrated Query) since it provides tons of possibilities when working with a list. Therefore we used LINQ to implement all 3 disc scheduling strategies – FCFS (First Come First Serve), SCAN (Elevator-like) and SSTF (Shortest Seek Time First).

FCFS was very easy – all we had to do is simply return the first number from the list.

SCAN was a bit harder – first we had to keep track of whether the elevator is going up or down (with a bool) and then before getting a number we had to re-evaluate- if the elevator is going down we had to check if there were bigger numbers left – if not the elevator would go up. If the elevator was going up we had to evaluate if there were smaller numbers left – if not the elevator would go down.

Based on the result we would either return the previous number (if going up) or the next one (if going down).

SSTF was a bit easier than SCAN – we used LINQ to compare each number in the list with the current one using absolute difference. The one that returned the smallest absolute difference was selected as the next one.

# Implementation Problems

The only problem we encountered with the implementation was using a Random to generate random numbers for the list. After running the project several times we came to the conclusion that those numbers aren’t really random generated and therefore the SSTF strategy ***sometimes*** wouldn’t lead to starvation.

We tried having a pre-defined list of numbers and just keep using that list (add the removed number at the end) but that leads to an infinite loop if using the SSTF strategy.

# Unit Testing

There is a separate project in the solution that is used for unit testing. There is 1 test method per disc scheduling strategy (3 in total).

# Notes

A small note regarding the UML diagram’s picture – since the diagram itself was pretty big we had to take the photo from afar. Please zoom in to see all the classes’ details (they should be crystal clear).