# Observer Pattern

Preslav Gerchev 2487136  
Dimitar Vikentiev 2464020

# Introduction

In week 2 we had to develop a project tracking match scores using the Observer 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 one-to-many dependency between objects where a state change in one object results with all its dependents being notified and updated automatically.”**

# Reusability

In terms of reusability the Observer Pattern does pretty well. The ISubject and IObserver interfaces can be used throughout the whole application and the Subject class that implements the ISubject can also be used anywhere- the only requirement is that the observers that are added implement the IObserver interface. Depending on what kind of update is used in the observer classes (push or pull both of who are described in details later on) the code can become even more reusable.

# Extensibility

In terms of extensibility the Observer pattern is very good.

It allows for adding new Observers as long as they implement the IObserver interface. We could have 100 different Observers with the only thing in common being the IObserver. Another plus of the Observer pattern is that the subject is easily modified during runtime – new observers can be added, old can be removed and those that are still in the list can be notified. It is worth to mention that those who unsubscribe can subscribe at any point later on during runtime.

# Maintainability

The Observer pattern helps for improving the code’s maintainability. The two main interfaces make sure that classes are independent of one another. If we change something in one Observer class there is no chance that the others will be affected. Same goes for the Subject classes that implement ISubject. The pattern is very easy to be maintained and as result of that both Java and the C# languages have native implementation of the pattern (Observable and Observer in Java, events and delegates in C#).

# Push vs. Pull

One of the most interesting things regarding the Observer pattern is the way we choose to pass data to the Observers which is either via a pull or via push.

Both have their advantages and disadvantages.

The main advantage of the push model is the lower coupling between the subject and the observers.

The disadvantage is that this model allows for less flexibility (something that we have demonstrated in our implementation of the pattern) – the Observers can’t choose what data to query specifically from the Subject (some Observers need one part of the data; others need other part of the data).

The main advantage of the pull model is the push model’s disadvantage. Using the pull model each Observer can choose what data to query and then react accordingly. The pull model allows for more flexibility.

The main disadvantage is that the pull model tightens the classes’ coupling –something that we always want to avoid.

The difference between the two models is the delegation of information handling – in push the Observer is not responsible for delegating information – it simply receives it-whether it needs the information or not is a whole another question.

In pull the Observer is responsible for “extracting” the information it needs. The subject is being passed as argument in the Update method and the Observers are responsible for querying it.

# Implementation

We decided to implement an app that will show exactly how the Observer pattern works – and therefore we have created a simple Match Tracking app. The goal is very easy – we’ve got a match tracker and match observers. Once the match’s result has been updated (a goal has been scored) the match tracker updates its list of match observers. We have implement two projects – one for showing the pull model and another one for showing the push model.