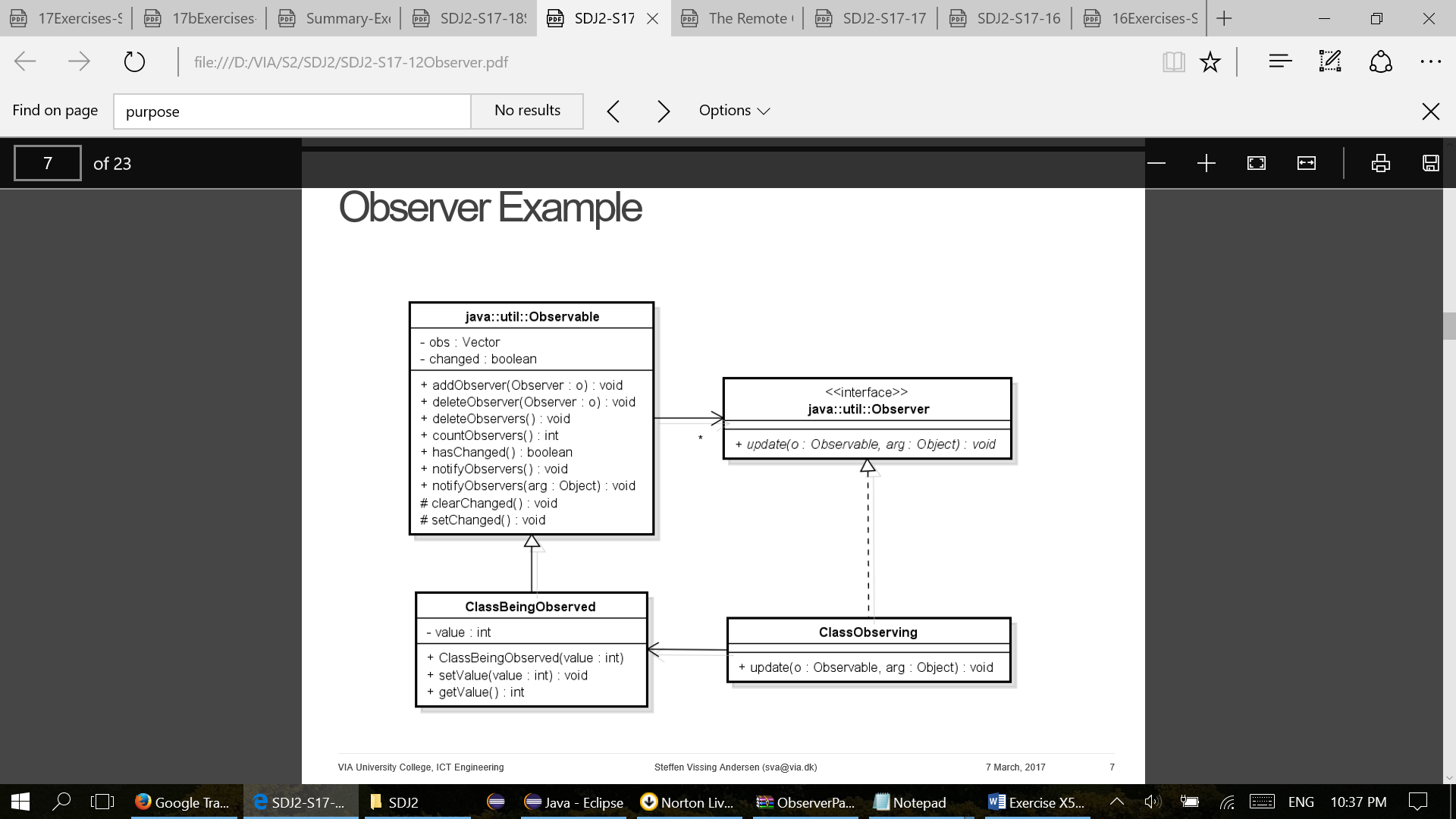
**Observer**

**General UML class diagram of an Observer design pattern.**



**The overall purpose for the Observer design pattern.**

Purpose - to automatically update objects when one object changes state.

Intent - Define a one-to-many dependency between objects so that when one object change state, all its dependents are notified automatically

2) Getting a message every time there is an update

**Example:**

1) Subscribe to a service

2) Getting a message every time there is an update

3) Act upon the update

In Java:

1) addObserver(observer)

2) setChanged() and notifyObservers(message)

3) observers implement method update(…)

**Purpose and relation of the general UML class diagram.**

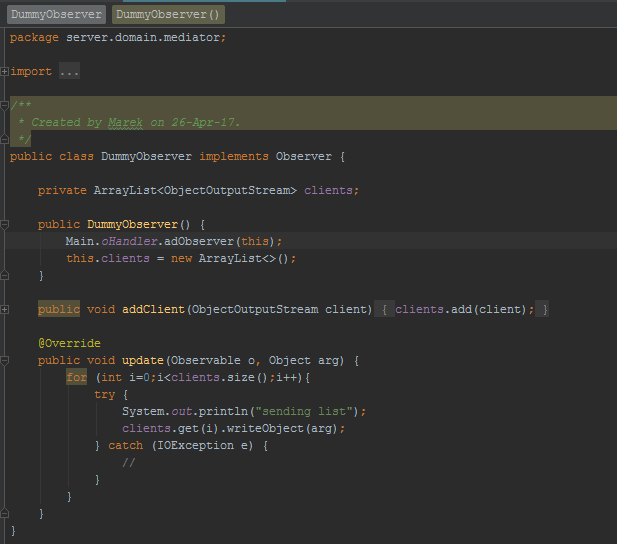
The pattern has two parts: a subject being observed and observers observing changes in the subject. The subject keeps a list of observers and when the state changes the observers are being notified making a loop and calling method Update for each observer.

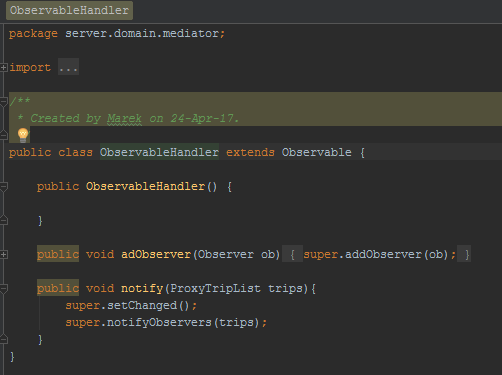
The Observer is an abstract interface with an abstract method Update which is implemented in the subclass ClassObserving. It is up to the ClassObserving what to do when a subject calls the update method but as indicated in the diagram this should get the updated state value.

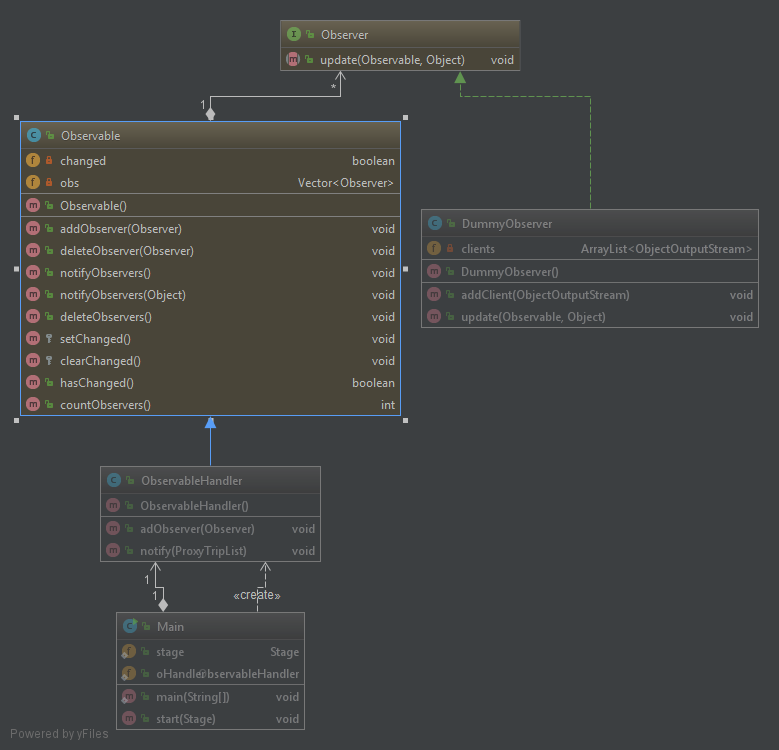
The subject/observable part is the abstract class Subject containing a list of observers and with methods to add an observer to the list, remove an observer from the list and notify all observers, respectively. The subclass ClassBeingObserved handles the logic of the subject without direct information about the observers simply calling method Notify in its superclass Subject. The observable side is thereby nicely divided into general parts which are handling observers with methods independent on the actual subject and are being observed on a general part with specific information about the subject state.

**Use of an Observer design pattern.**

The Observer defines a one-to-many relationship that changes state of one object, the others are notified and updated automatically.

 **Code fractions directly related to the pattern.**



**UML class diagram of the Observer design pattern we implemented.**

**The problems related to the Observer design pattern as implemented in the Java API.**

1. Observable is not abstract

2. The pattern in this form cannot be used in RMI – Observer is not Remote – Observable is not a Remote interface

3. Observer is not generic – Sending Object’s and may have to type cast in method update

4. Observable/Subject is not an interface – Calling addObserveryou may need a type cast

5. Method notifyObserversis not using threads – The application calling notifyObserversmay block until finished

6. The ObservableClassneeds to extend Observable – In Java multiple inheritance is not an option