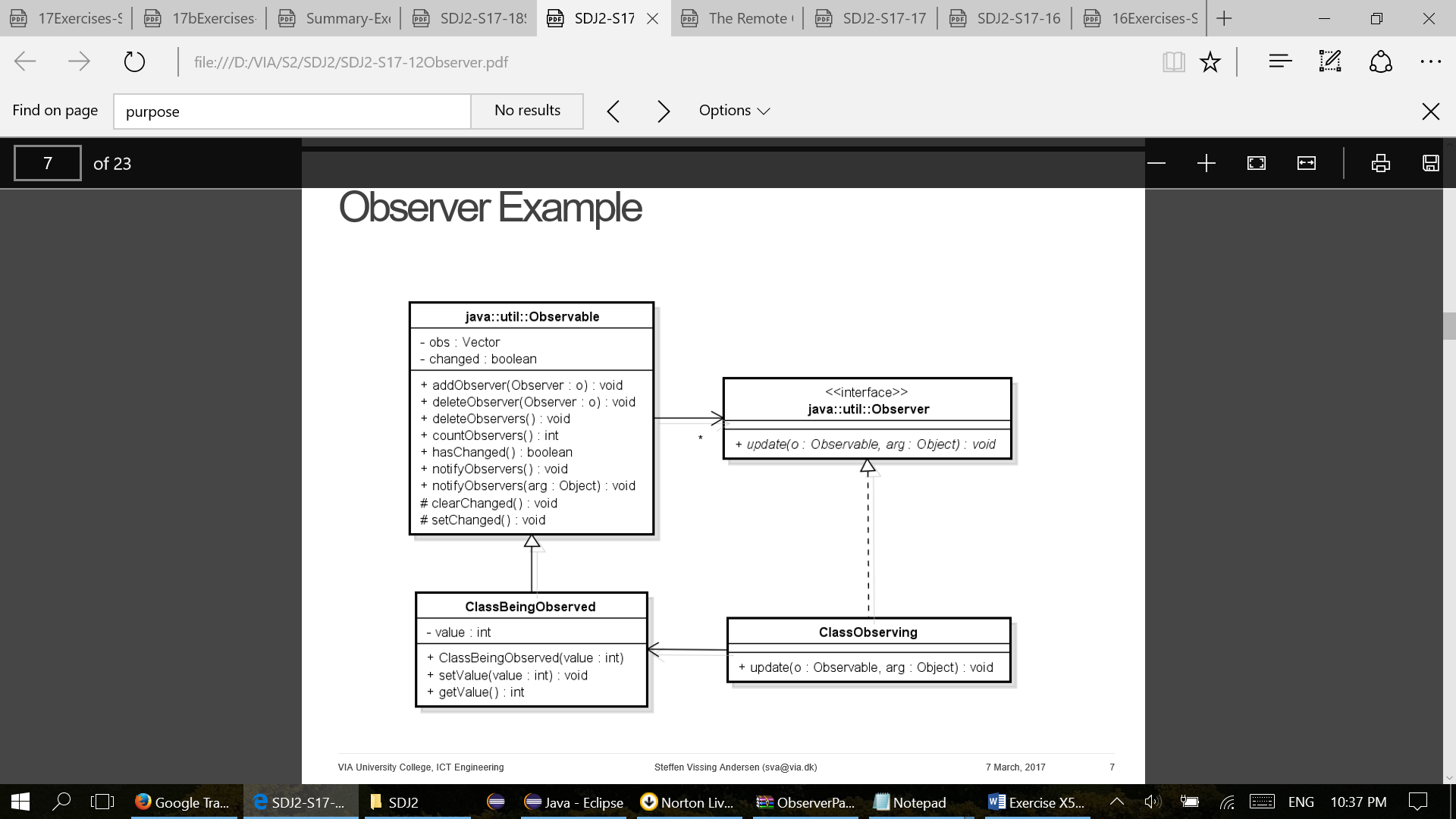
**Observer design pattern**

**General UML diagram for observer design pattern:**



**General purpose of observer design pattern:**

* Observer automatically updates observing objects when one observable object changes state.
* One-to-many dependency is defined. All observing objects are notified when observed object changes its state.
* General algorithm of observer:
* 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(…)

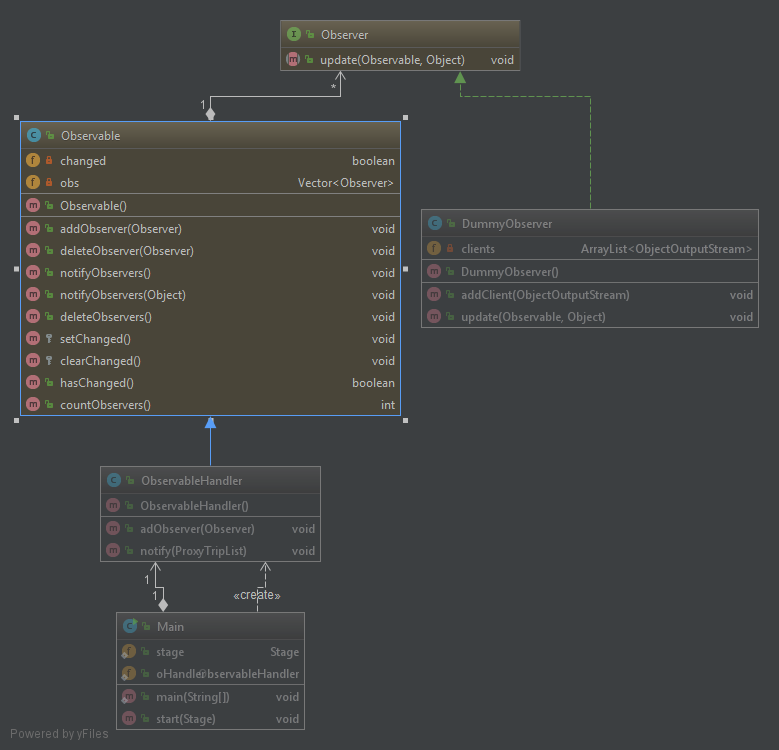
**Description of UML diagram:**

The pattern has two parts: a subject being observed and observers observing state change in the subject. The subject keeps a list of observers and when the subject's 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 to be 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 and with methods changing the state simply calling method Notify in its superclass Subject. The observable side is thereby nicely divided into a general part handling observers with methods independent on the actual subject being observed and a general part with specific information about the subject state.

**Our implementation:**

****

**Code examples:**

* ObservableHandler class

public class ObservableHandler extends Observable {  
   
 public void addObserver(Observer ob){  
 super.addObserver(ob);  
 }  
  
 public void notify(ProxyTripList trips){  
 super.setChanged();  
 super.notifyObservers(trips);  
 }  
}

* Update method from DummyObserver class

@Override  
public void update(Observable o, Object arg) {  
 for (int i=0;i<clients.size();i++){  
 try {  
 clients.get(i).writeObject(arg);  
 clients.get(i).reset();  
 } catch (IOException e) {  
 //  
 }  
 }  
}

* Method addTrip from DataHandler responsible for triggering state change in observable class

public void addTrip(Bus bus, Chauffeur chauffeur, Destination pickUp, Destination destination, int distance, LocalDate startDatePicker, String fieldStartTime, LocalDate endDatePicker, String fieldEndTime, int price, boolean food, boolean accommodation, boolean ticket) {  
 Trip trip = new Trip(bus, chauffeur, pickUp, destination, distance, startDatePicker, fieldStartTime, endDatePicker, fieldEndTime, price);  
  
 if (food) {  
 trip.setFood(true);  
 }  
 if (accommodation) {  
 trip.setAccommodation(true);  
 }  
 if (ticket) {  
 trip.setTickets(true);  
 }  
 trips.add(trip);  
  
 Main.*oHandler*.notify(trips);  
}

* DummyObserver is observing class which contains all clients connected to the server. Upon update it send updated trip list (arg) to all of them.