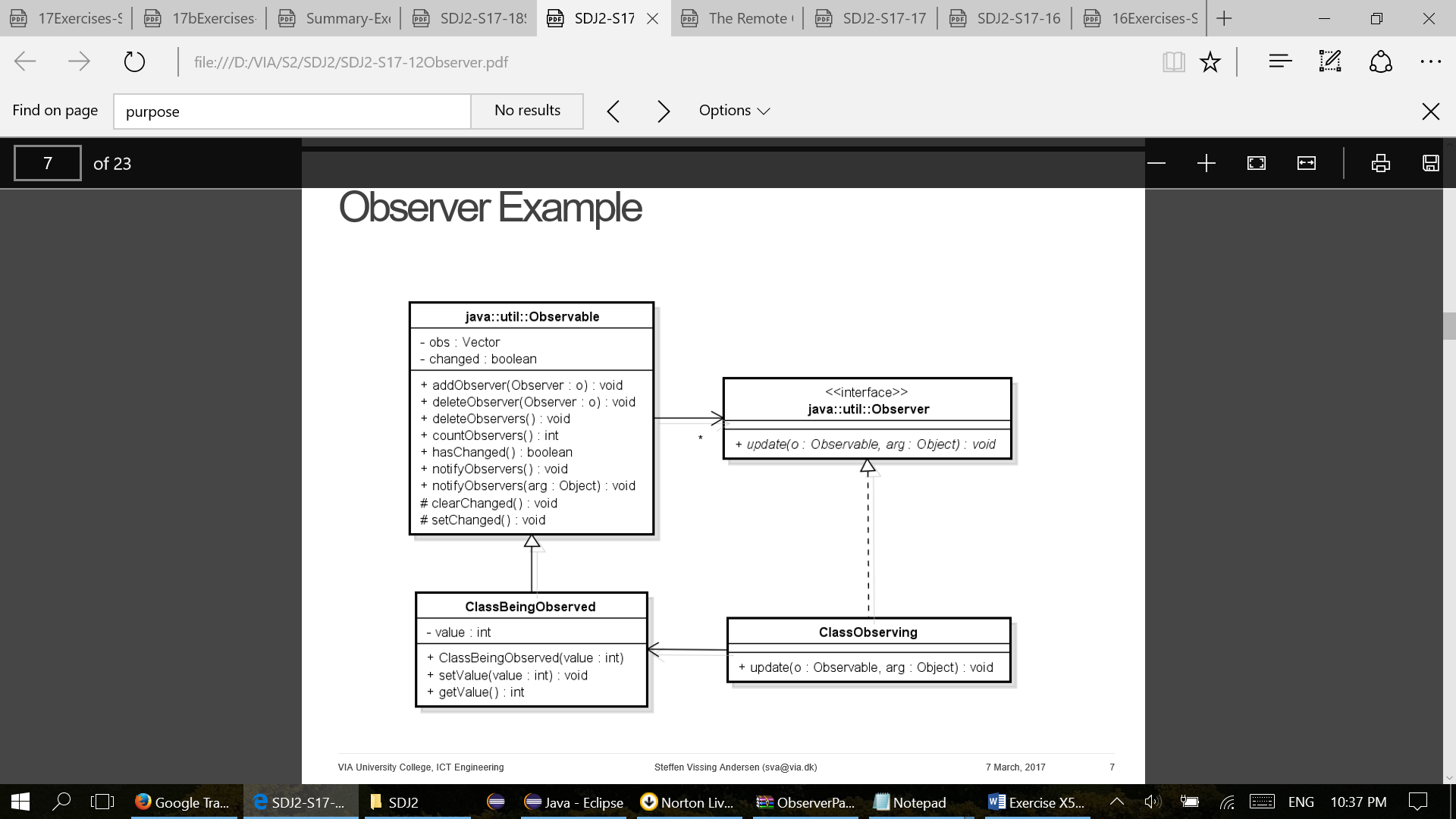
**Exercise X5 – Observer**

1. **Find (and insert into a document) a general UML class diagram of an Observer design pattern.**



1. **Describe the overall purpose for the Observer design pattern.**

Purpose - to automatically update observing objects when one observable 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

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(…)

**c) Describe in text each part of the general UML class diagram (each interface and class). Purpose and relation.**

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.

**d) Describe how to use an Observer design pattern.**

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

**e) Give overall a few remarks to implementation, what to remember.**

**f) Insert code examples (your own code) for each part of the Observer pattern. Not full code only code fractions directly related to the pattern.**

Observer class 1

import java.util.Observable; import java.util.Observer;

public class Class1Observing implements Observer { private String name; private ClassBeingObserved observable;

public Class1Observing(String name, ClassBeingObserved obs)

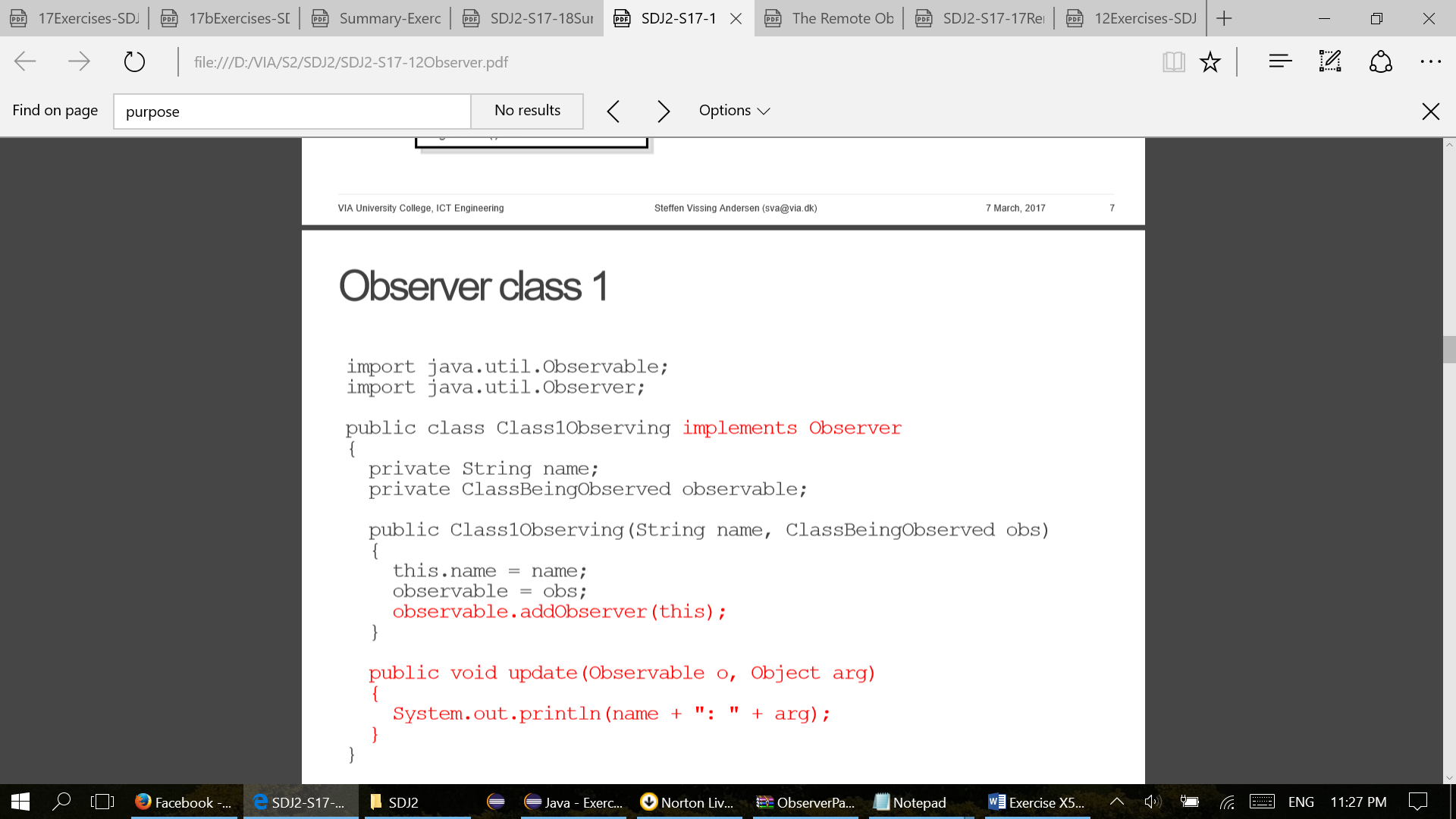
{ this.name = name;

observable = obs;

observable.addObserver(this); }

public void update(Observable o, Object arg)

{ System.out.println(name + ": " + arg); } }



Observable class

import java.util.Observable; import java.util.Observer;

public class ClassBeingObserved extends Observable {

private int value;

public ClassBeingObserved() {

this.value = 0; }

public int getValue() {

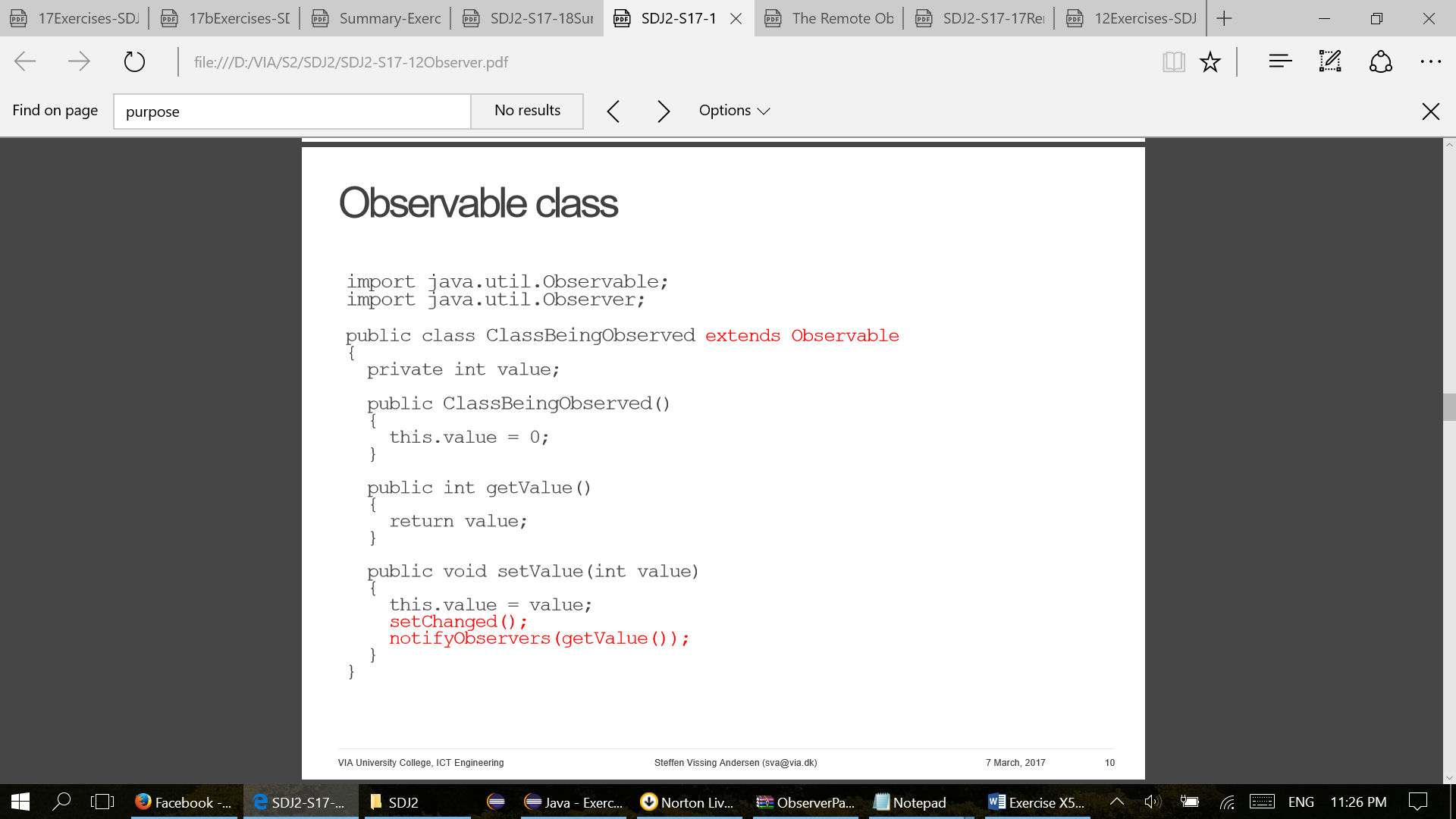
return value; }

public void setValue(int value) {

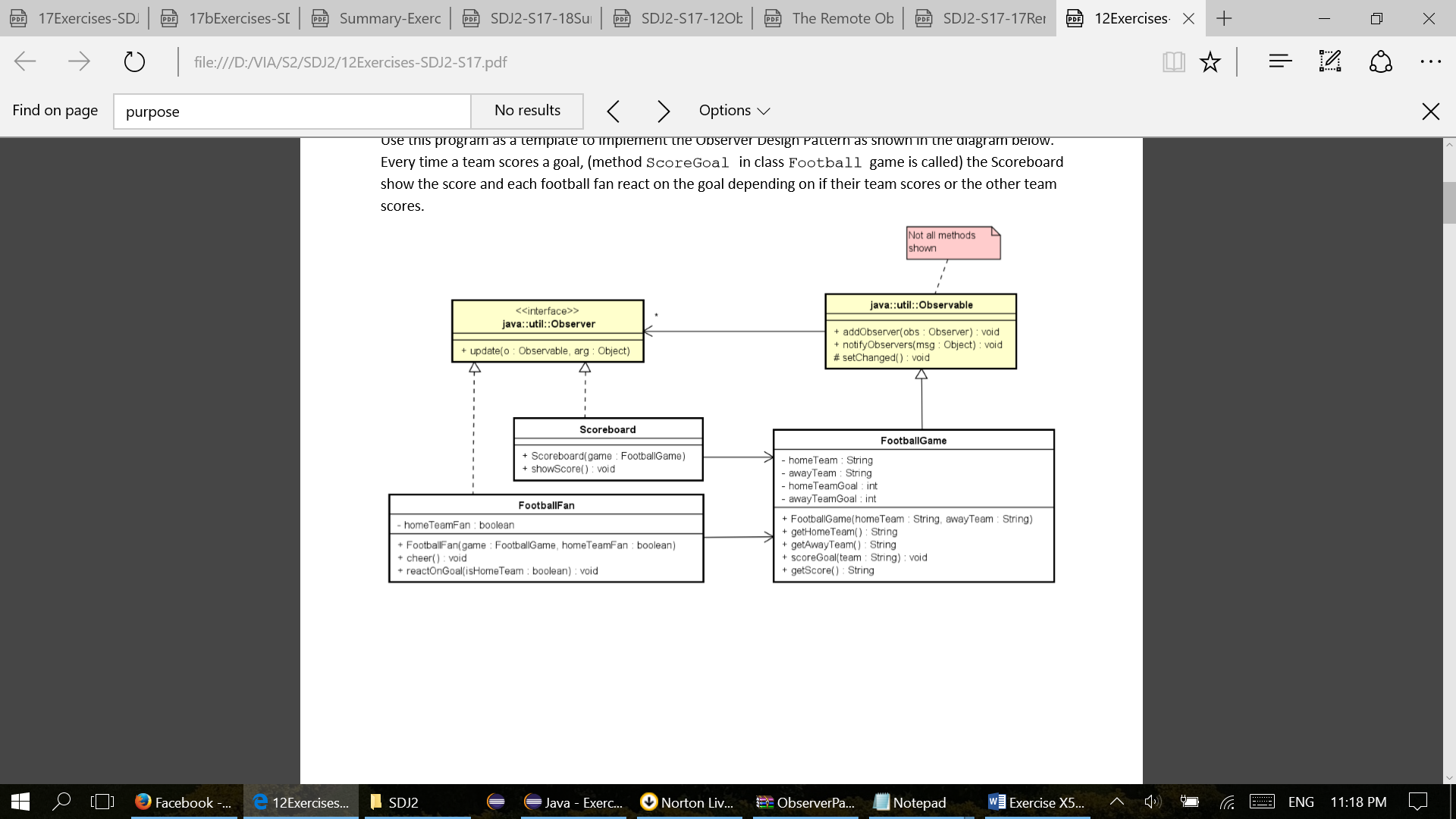
this.value = value;

setChanged();

notifyObservers(getValue()); } }



**g) Create and insert into the document a UML class diagram of the Observer design pattern you implemented.**



**h) Describe 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

**i) Give a short description of the Remote Observer and Remote Observer Delegate pattern.**

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

