Hochschule für Technik Stuttgart Leistungsnachweis im **Sommersemester 2016**		
Masterstudiengang **S**oftware **T**echnology		Name:
Fach:	Software Engineering 2 (Part 2)	Vorname:
Datum:	8. Juli 2013, 11.00 – 13.00 (Both Parts)	Semester:
Zeit:	120 min (Both Parts)	Matrikel-Nr.:
Prüfer:	Prof. Dr. Deininger / Prof. Dr. Wanner	
Hilfsmittel:	Non-programmable Pckedt-Calculator, One double sided sheet A4 for Part 2	
Anlagen:	-	

General Note: You may give all answers directly on these pages. If you remove the staples please add your name and matriculation number on **each** page. Overall you can reach 50 points in part 1 and 50 points in part 2.

Examination Question 1 "Advanced Testing" (10 Points)

Given is the following class under test. The class creates and opens a Swing-window with a button on it. The supplied ActionListener contains a (possible) complex interaction logic. For solving the Questions it is not necessary to understand this code fully.

```
import java.awt.Dimension;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class GUI extends JFrame {

     public GUI(ActionListener al){
          super("Sample");

          JButton b = new JButton("Press");
          b.addActionListener(al);

          JPanel p = new JPanel();
          p.add(b);

          this.setContentPane(p);
          this.setPreferredSize(new Dimension(200, 300));
          this.setDefaultCloseOperation(EXIT_ON_CLOSE);
          this.setLocationByPlatform(true);
          this.pack();
          this.setVisible(true);
     }

}
```

For testing the above class a test double is crreated which implements the interface `ActionListener`. An ActionEvent contains several details about an action triggered (e.g. the originater of the action, which would be the pressed button in this example). Again, For solving the questions it is not necessary to understand ActionEvents. The test double is implemented the following way:

```
import java.awt.event.*;
import java.util.*;

public class TestListener implements ActionListener {

	List<ActionEvent> events = new ArrayList<>();

	public List<ActionEvent> getEvents() {
		return events;
	}

	@Override
	public void actionPerformed(ActionEvent event) {
		events.add(event);
	}

}
```

The above test double is used in the context below.

```
public class TestGUI {

	private static GUI gui;
	private static TestListener listener;

	@BeforeClass
	public static void setUp(){
		listener = new TestListener();
		gui = new GUI(listener);
	}
	@Test
	public void test(){
		// run some tests on gui here
		List<ActionEvent> events = listener.getEvents();
		// assert that some events happened here
	}
}
```

What kind of test double is the class `TestListener`?

What are the characteristics of this kind of test double?

What would you have to add (if at all) to make it a “Mock” (just explain, no code needed)?

What would be advantages of using mockito for supplying the test double, what would be disadvantages?

Advantages	Disadvantages

Java-Reflection, mockito, and AspectJ all provide means to access or change otherwise inaccessible or intangible parts of a java program. For each of these technologies give one sample use case, for what it could be used.

Java-Reflection	
mockito	
AspectJ	

Examination Question 2 "Design Patterns" (10 Points)

With the help of the composite pattern the following task should be solved:

> For a drawing program shapes should be organized. Shapes should be either primitive shapes like Circle or Rectangle or a group of shapes. Groups may contain (an unlimited number of) either primitive shapes or again groups. A method `count():int` should return the number of primitive shapes contained within one shape. This method should be available for all shapes.

Draw an UML-Diagram with all the classes / interfaces and their relationships, variables and methods (no constructors needed).

Sketch the Java Code for your data structure (no constructors needed, it is sufficient to code only one primitive shape).

```
// Class/Interface


// Variables – if needed




// Methods

_____________________________ int count()
```

```
// Class/Interface


// Variables – if needed




// Methods

_____________________________ int count()
```

```
// Class/Interface

// Variables – if needed



// Methods

_____________________________ int count()
```

```
// Class/Interface

// Variables – if needed



// Methods

_____________________________ int count()
```

What is the role of design patterns for software maintenance?

What would be the risks of maintaining code not containing patterns?

What would be the risks of overdoing patterns?

Examination Question 3 "Metrics" (10 Points)

Given are the definitions of the following two Martin-Metrics (as discussed in the lecture)

Afferent Couplings (CA_p) of a package p: the number of classes **outside** of this package that depend upon classes within this package.

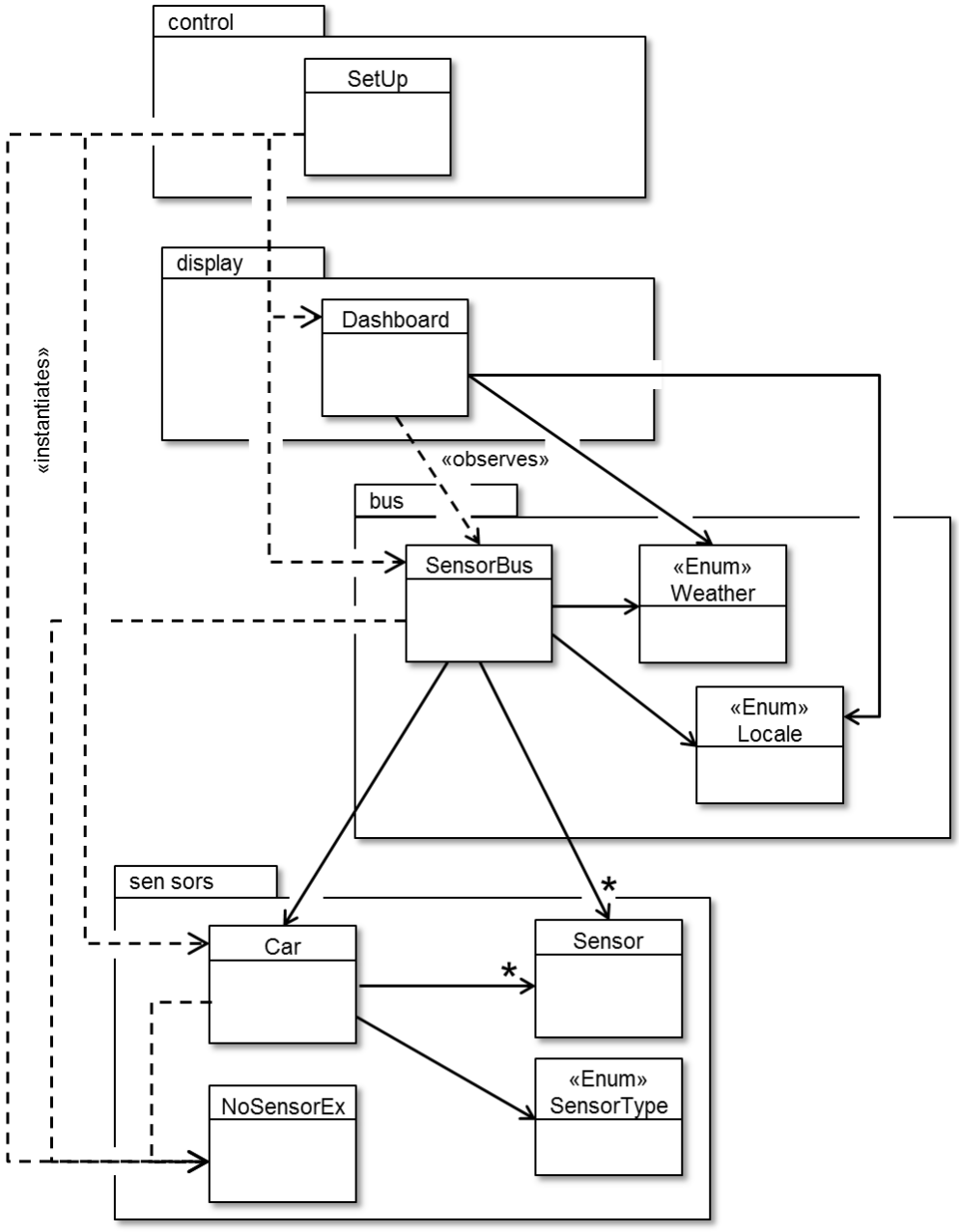
Efferent Couplings (CE_p) of a package p: the number of classes **inside** this package that depend upon classes outside of the package.

What does the afferent coupling of a package indicate – i.e. what does it want to measure?	

What does the efferent coupling of a package indicate – i.e. what does it want to measure?	

As discussed in the lecture – metrics are models too. According to the modeling-schema of the lecture, describe what is modeled here; i.e. what elements are actually mapped to the model?

Given is the following UML-Diagram of an application:



Calculate the afferent and efferent coupling for the above application.

	CA_p	CE_p
control	0	1
display	1	1
bus	2	1
sensors	2	0

The programmer of this application claims he did a layered implementation, that is, packages on a lower level are only used by packages of a higher level. Which numbers indicate / backup this statement?

Are these number(s) sufficient for claiming a layered architecture without inspecting the code or diagram? Give reasons for your decision.