Hochschule für Technik Stuttgart Leistungsnachweis im **Wintersemester 2017/18**		Page 1 of 12
Masterstudiengang **S**oftware **T**echnology		Name:
Fach:	Software Engineering 2 (Part Deininger)	Vorname:
Datum:	6. Februar 2018, 11.00 – 13.00 (Both Parts)	Semester:
Zeit:	120 min (Both Parts)	Matrikel-Nr.:
Prüfer:	Prof. Dr. Deininger / Prof. Dr. Wanner	
Hilfsmittel:	Non-programmable Pocket-Calculator, One double sided sheet A4 for this part	
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 "Design Patterns" (15 Points)

The contract of a prepaid cell phone should be modelled and implemented. A basic contract has a contract number (of type `int`) and a balance (of type double), but no monthly charges. The contract number is not automatically generated, but is to be set as a parameter by the constructor as well as the initial balance. The balance has a getter and a setter.

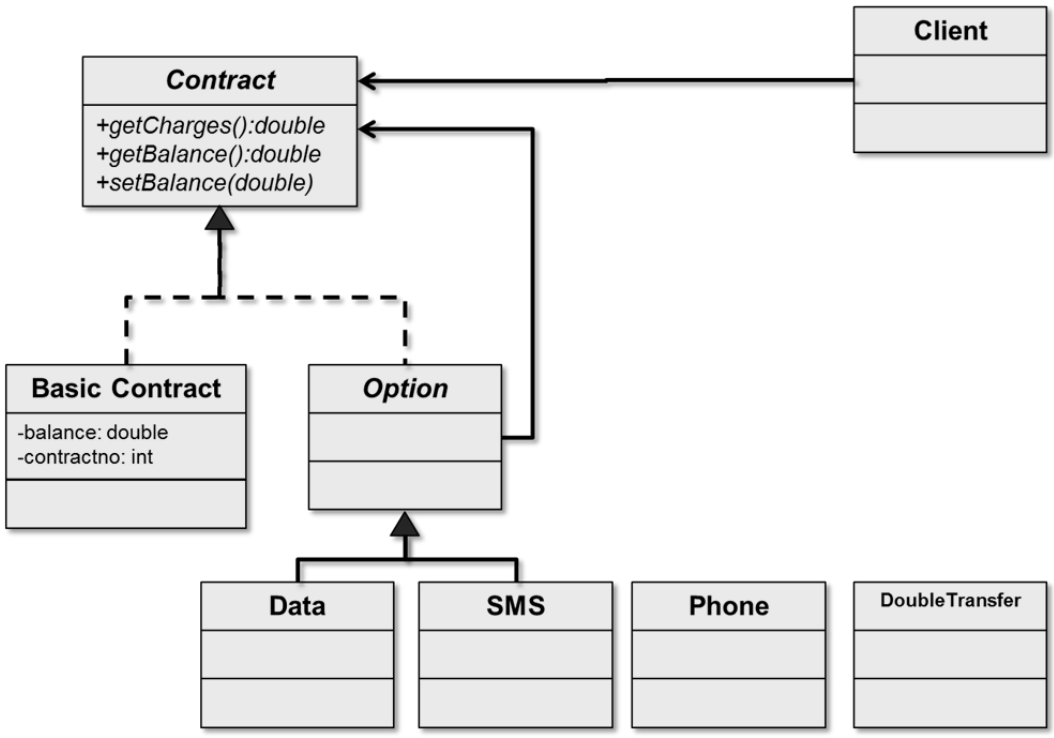
The following options can be added to a contract (if needed also several times):

- 100 MB of data (monthly charge 1.00 €)
- 50 SMS (monthly charge 0.50 €)
- 50 minutes (monthly charge 1.50 €)
- Double Transfer Rate (monthly charge 2.00 €)

Implement this this requirement with the help of the decorator pattern. All contract elements should be able to understand the methods **`getCharges():double`**, **`getBalance():double`** and **`setBalance (double)`**.

The method **`getCharges()`** should provide the monthly charge of a contract with all its options selected. The methods **`getBalance()`** and **`setBalance(…)`** should be passed through and access the basic contract.

- Draw an UML-Diagram with all the classes / interfaces and their relationships, fields and methods (no constructors needed).
- Also provide a client class which holds a contract.
- Mark in your diagram, which elements are open and which are closed.



Now sketch the Java-code for your classes or interfaces (one each page) – for concrete decorators you should select only one representative, the above client has not to be implemented.

// Class/Interface

```
public interface Contract {

      public double getCharges();

      public double getbalance();

      public void setBalance(double balance);

}
```

// Fields – if needed

// Constructor – if needed

// Methods – if needed, if not cross out

_____ `double getBalance()`

_____ `void setBalance(double balance)`

_____ `double getCharges()`

// Class/Interface

// Fields – if needed

// Constructor – if needed

```java
public class BasicContract implements Contract {

	@SuppressWarnings("unused")
	private int contractno;
	private double balance = 0;

	public BasicContract(double bakance, int
contractno) {
		this.contractno = contractno;
		this.balance = balance;
	}

	@Override
	public double getCharges() {
		return 0;
	}

	@Override
	public double getbalance() {
		return balance;
	}

	@Override
	public void setBalance(double balance) {
		this.balance = balance;
	}
}
```

// Methods – if needed, if not cross out

_____ `double getBalance()`

_____ `void setBalance(double balance)`

_____ `double getCharges()`

// Class/Interface

// Fields – if needed

// Constructor – if needed

```
public abstract class Option implements Contract
{

	private Contract contract;

	public Option(Contract contract) {
		this.contract = contract;
	}

	@Override
	public double getCharges() {
		return contract.getCharges();
	}

	@Override
	public double getbalance() {
		return contract.getbalance();
	}

	@Override
	public void setBalance(double balance) {
		contract.setBalance(balance);
	}
}
```

// Methods – if needed, if not cross out

_____ **double** getBalance()

_____ **void** setBalance(double balance)

_____ **double** getCharges()

// Class/Interface

```
public class Data extends Option {

	public Data(Contract contract) {
		super(contract);
	}

	@Override
	public double getCharges() {
		return super.getCharges() + 1;
	}
}
```

// Fields – if needed

// Constructor – if needed

// Methods – if needed, if not cross out

_____ `double getBalance()`

_____ `void setBalance(double balance)`

_____ `double getCharges()`

Now create a contract with number 1234567, 10€ initial balance and with 200 MB data, Double Transfer Rate and 50 min. Then print its charges.

```
public static void main(String[] args){

            Contract c = new DoubleTransfer(
                  new Data (
                        new Data(
                              new Phone(
                              new BasicContract(10, 1234567)))));
            System.out.println(c.getCharges());




}
```

What is the advantage of using the decorator pattern in this scenario?

Simple extension

Which principles (as stated in the lecture) have been fulfilled through this pattern?

High coupling / low cohesion

Open / closed principle

What is the overall role of design patterns for software maintenance?

Predefined maintenance

- At patterns changes may / should happen
- At other places no changes should happen.

Examination Question 2 "Advanced Testing" (15 Points)

Given is the following class under test called Locator that internally uses the interface `LocatorService`.

```
package de.hft.stuttgart.application;
import java.awt.Point;

import org.external.api.LocatorService;

public class Locator {

    private final LocatorService locatorService;

    public Locator(LocatorService locatorService) {
        this.locatorService = locatorService;
    }

    public Point locate(int x, int y) {
        if (x < 0 || y < 0) {
            return new Point(Math.abs(x), Math.abs(y));
        } else {
            return locatorService.geoLocate(new Point(x, y));
        }
    }
}
```

The method `locate(…)` behaves as following: If a point with some negative coordinates is passed, the method returns a new point with positive coordinates. If coordinates are positive then a search through a `LocatorService` is done. This class represents some external API that the `Locator` is calling. Since there is no control over this API and the internal structure is not known, a test double should be used in the unit tests insted of the real implementation.

The (external) interface is defined as following:

```
package org.external.api;
import java.awt.Point;

public interface LocatorService {

     public Point geoLocate(Point point);

}
```

For testing the above class a test double is created which implements the interface `LocatorService`. The double simply returns always the same Point (12, 15). The test double is implemented the following way:

```
package de.hft.stuttgart.application.test.doubles;

import java.awt.Point;
import org.external.api.LocatorService;

public class LocatorServiceDouble implements LocatorService {

    public static final Point TEST_POINT = new Point(12, 15);

    @Override
     public Point geoLocate(Point point) {
          return TEST_POINT;
     }
}
```

The above test double is used in the context below.

```
package de.hft.stuttgart.application.test;
import org.external.api.LocatorService;
import org.junit.*;
import de.hft.stuttgart.application.Locator;
import de.hft.stuttgart.application.test.doubles.LocatorServiceDouble;
import static org.junit.Assert.assertEquals;
import java.awt.Point;

public class LocatorTest {
    private Locator locatorUnderTest;
    private LocatorService locatorServiceDouble;

    @Before
    public void setUp() {
        locatorServiceDouble = new LocatorServiceDouble();
        locatorUnderTest = new Locator(locatorServiceDouble);
    }

    @Test
    public void testLocateWithServiceResult() {
        assertEquals(LocatorServiceDouble.TEST_POINT,
                        locatorUnderTest.locate(1, 1));
    }

    @Test
    public void testLocateLocalResult() {
        Point expected = new Point(1, 1);
        assertEquals(expected, locatorUnderTest.locate(-1, -1));
    }
}
```

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

Stub

What are the characteristics of this kind of test double?

provides output on calls

sometimes irregular outputs

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

Spy

stubbing

records interactions

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

Mock

spying

configuration / verification of interaction

Java-Reflection, mockito, AspectJ and Selenium all provide means for testing. For each of these technologies give one sample use case, for what it could be used (no code needed – textual explanation is sufficient).

Java-Reflection	Accessing private variables, methods
mockito	Mocking interfaces, injection, define behavior, recording , verification
AspectJ	Changing existing behavior, reaching otherwise unreachable code parts

Selenium	Programmatically controlling a browser

Which of the above technologies or frameworks would be most suited for implementing/providing the above double of LocatorService?

Mockito

What would be advantages and disadvantages of using this seleceted technology/framework?

Advantages	Disadvantages
Simple usage	Having additional library, learning a framework