6. homework assignment; JAVA, Academic year 2013/2014; FER

A usual, please see the last page. I mean it! You are back? OK. This homework consists of four+ problems.

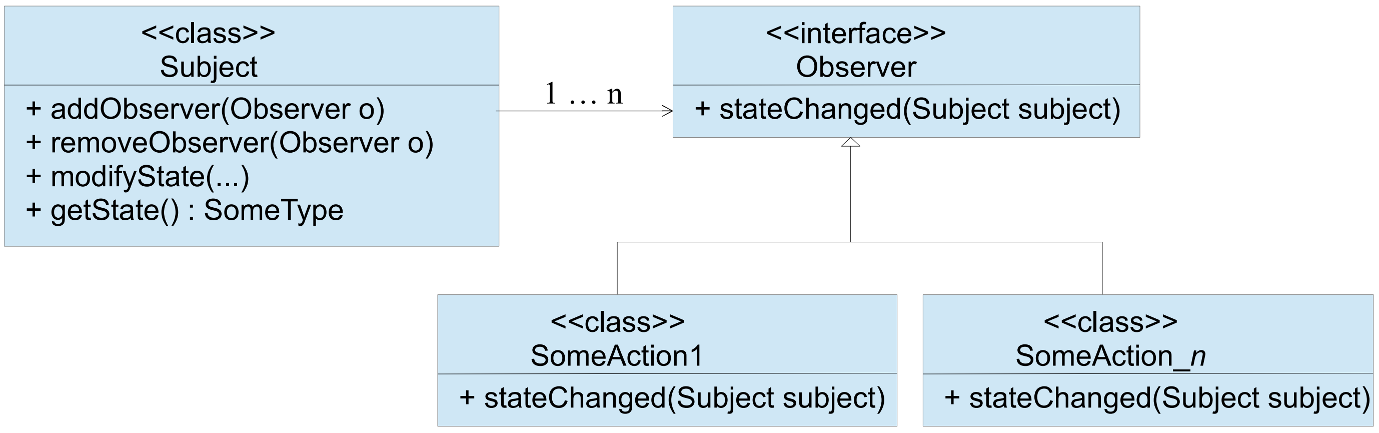
Problem 1.

When writing non-trivial programs, you are often confronted with the following situation: there is an object that holds some data (let's call it the object state) and each time that data changes, you would like to execute a certain action or even more than one action (to process the data, to inform user about the change, to update the GUI, etc). If you have full control of the object and if there is always the same (single) action, this can be easily coded into the object itself. However, often you will develop the object separately (or will be provided with one that is not under your control), and you will want to be able to change the actions when necessary, and to develop new actions without ever changing or recompiling the object itself.

The *Observer pattern* is an appropriate solution that can be utilized in previously described situation. The basic idea is this: your object (denoted as the *Subject* in this design pattern) does not need to know anything about the concrete actions (the *Concrete Observers* in this design pattern) you will develop; however, it will mandate that in order to be able to invoke your action/actions, you must satisfy following conditions:

1. the object (the *Subject*) will have to be able to "talk" with your actions (*Concrete Observers*) in a way that it expects – this means that the object will prescribe a certain interface your actions will have to implement; this interface is usually called the *Observer* interface (or *Abstract Observer*);
2. the object (the *Subject*) will provide you with a method that will allow you to register actions (*Concrete Observers*) you developed, and which, of course, implements the *Observer* interface;
3. the object (the *Subject*) can provide you a method for action removal so that you can at any time de-register previously registered actions;
4. every time the objects' state changes, the object will invoke all of the registered actions by using methods of prescribed interface.

At its simplest case, the observer pattern can be depicted as following picture shows.



<<class>>
Subject
+ addObserver(Observer o)
+ removeObserver(Observer o)
+ modifyState(...)
+ getState() : SomeType
1 … n
<<interface>>
Observer
+ stateChanged(Subject subject)
<<class>>
SomeAction1
+ stateChanged(Subject subject)
<<class>>
SomeAction_*n*
+ stateChanged(Subject subject)

In this picture, the instance of the `Subject` class represents the *object* from previous example. It holds private list of registered observers. The interface `Observer` is the interface that our object expects all actions to implement, and it has a single method:

```
stateChanged(Subject subject);
```

We can implement several different actions (classes `SomeAction1`, `SomeAction2`, …, `SomeAction_n`) that all implement the `Observer` interface. Our object provides usually three methods. Method `addObserver` is used to register a concrete action with our object. Method `getState()` is used to retrieve current state and method `modifyState` allows us to modify the objects state. Each time when a state is modified, the subject automatically notifies all registered observers of this change by calling `stateChanged` method on each registered observer. This organization of code will allow us to write the following example:

```
Subject s = new Subject();  // create object with interesting state

Observer observer1 = new SomeAction1();  // create first action
s.addObserver(observer1);                // and register it

Observer observer2 = new SomeAction2();  // create second action
s.addObserver(observer2);                // and register it

s.modifyState(...);  // modify objects state; observer1.stateChanged(s) and
                     // observer2.stateChanged(s) is called as a consequence
s.modifyState(...);  // modify objects state; observer1.stateChanged(s) and
                     // observer2.stateChanged(s) is called as a consequence

s.removeObserver(observer1);

Observer observer3 = new SomeAction3();  // create another action
s.addObserver(observer3);                // and replace the old one with this now

s.modifyState(...);  // modify objects state; now observer2.stateChanged(s)
                     // and observer3.stateChanged(s) is called as a consequence
```

Since this is widely utilized design pattern (for example, it is used throughout graphical user interface libraries in Java), you will practice it on a following example.

The `Subject` class here will be `IntegerStorage`.

```
package hr.fer.zemris.java.tecaj.hw6.problem1a;

public class IntegerStorage {

	private int value;
	private List<IntegerStorageObserver> observers;

	public IntegerStorage(int initialValue) {
		this.value = initialValue;
	}

	public void addObserver(IntegerStorageObserver observer) {
		// add the observer in observers if not already there ...
		// … your code …
	}

	public void removeObserver(IntegerStorageObserver observer) {
		// remove the observer from observers if present ...
		// … your code …
	}

	public void clearObservers() {
		// remove all observers from observers list ...
		// … your code …
	}

	public int getValue() {
		return value;
	}
	public void setValue(int value) {
		// Only if new value is different than the current value:
		if(this.value!=value) {
			// Update current value
			this.value = value;
			// Notify all registered observers
			if(observers!=null) {
				for(Observer observer : observers) {
					observer.valueChanged(this);
				}
			}
		}
	}
}
```

The `Observer` interface will be `IntegerStorageObserver`.

```
package hr.fer.zemris.java.tecaj.hw6.problem1a;

public interface IntegerStorageObserver {
	public void valueChanged(IntegerStorage istorage);
}
```

The main program is `ObserverExample`:

```
package hr.fer.zemris.java.tecaj.hw6.problem1a;

public class ObserverExample {

	public static void main(String[] args) {

		IntegerStorage istorage = new IntegerStorage(20);

		IntegerStorageObserver observer = new SquareValue();

		istorage.addObserver(observer);
		istorage.setValue(5);
		istorage.setValue(2);
		istorage.setValue(25);

		istorage.removeObserver(observer);

		istorage.addObserver(new ChangeCounter());
		istorage.addObserver(new DoubleValue());
		istorage.addObserver(new LogValue(Paths.get("./log.txt")));
		istorage.setValue(13);
		istorage.setValue(22);
		istorage.setValue(15);

	}

}
```

Copy these three sources into your Eclipse project. Your task is to implement four concrete observers: `SquareValue` class, `ChangeCounter` class, `DoubleValue` class, and `LogValue` class. Instances of `SquareValue` class write a square of the integer stored in the `IntegerStorage` to the standard output, instances of `ChangeCounter` counts (and writes to the standard output) the number of times value stored integer has been changed since the registration. Instances of `DoubleValue` class write to the standard output double value of the current value which is stored in subject, but only first two times since its registation with subject; after writing the double value for the second time, the observer automatically de-registers itself from the subject. Instances of `LogValue` class for each notification open a log file whose path is given in its constructor, appends a new row with current value obtained from subject and closes the file. The output of the previous code should be as follows:

```
Provided new value: 5, square is 25
Provided new value: 2, square is 4
Provided new value: 25, square is 625
Number of value changes since tracking: 1
Double value: 26
Number of value changes since tracking: 2
Double value: 44
Number of value changes since tracking: 3
```

and the content of file "log.txt" which will be created in current directory must be:

```
13
22
15
```

After you finish this task, copy the content of subpackage `problem1a` into `problem1b`. You will continue your work here while package `problem1a` will preserve your previous solution.

Lets recapitulate what we have done so far. We developed our `Subject` to allow a registration of multiple observers. We defined the `Observer` interface and developed more that one actual observers (classes that implemented the `Observer` interface). Now you will modify your code from package `problem1b` to support following.

- Change the `Observer` interface (i.e. `IntegerStorageObserver`) so that instead of a reference to `IntegerStorage` object, the method `valueChanged` gets a reference to an instance of `IntegerStorageChange` class (and create this class). Instances of `IntegerStorageChange` class should encapsulate (as read-only properties) following information: (a) a reference to `IntegerStorage`, (b) the value of stored integer before the change has occurred, and (c) the new value of currently stored integer.
- During the dispatching of notifications, for a single change only a single instance of `IntegerStorageChange` class should be created and a reference to that instance should be passed to all registered observers (the order is not important). Since this instance provides only a read-only properties, we do not expect any problems.
- Modify all other classes to support this change.
- Modify the main program so that it registers all developed observers at the beginning of the program and then performs calls to `istorage.setValue(...)`.

Problem 2.

Create an implementation of `ObjectMultistack`. You can think of it as a `Map`, but a special kind of `Map`. While `Map` allows you only to store for each key a single value, `ObjectMultistack` must allow the user to store multiple values for same key and it must provide a stack-like abstraction. Keys for your `ObjectMultistack` will be instances of the class `String`. Values that will be associated with those keys will be instances of class `ValueWrapper` (you will also create this class). Let me first give you an example.

```java
package hr.fer.zemris.java.custom.scripting.demo;

import hr.fer.zemris.java.custom.scripting.exec.ObjectMultistack;
import hr.fer.zemris.java.custom.scripting.exec.ValueWrapper;

public class ObjectMultistackDemo {

	public static void main(String[] args) {
		ObjectMultistack multistack = new ObjectMultistack();

		ValueWrapper year = new ValueWrapper(Integer.valueOf(2000));
		multistack.push("year", year);

		ValueWrapper price = new ValueWrapper(200.51);
		multistack.push("price", price);

		System.out.println("Current value for year: "
							+ multistack.peek("year").getValue());
		System.out.println("Current value for price: "
							+ multistack.peek("price").getValue());

		multistack.push("year", new ValueWrapper(Integer.valueOf(1900)));
		System.out.println("Current value for year: "
							+ multistack.peek("year").getValue());

		multistack.peek("year").setValue(
				((Integer)multistack.peek("year").getValue()).intValue() + 50
		);
		System.out.println("Current value for year: "
							+ multistack.peek("year").getValue());

		multistack.pop("year");
		System.out.println("Current value for year: "
							+ multistack.peek("year").getValue());

		multistack.peek("year").increment("5");
		System.out.println("Current value for year: "
							+ multistack.peek("year").getValue());
		multistack.peek("year").increment(5);
		System.out.println("Current value for year: "
							+ multistack.peek("year").getValue());
		multistack.peek("year").increment(5.0);
		System.out.println("Current value for year: "
							+ multistack.peek("year").getValue());

	}

}
```

This short program should produce the following output:

```
Current value for year: 2000
Current value for price: 200.51
Current value for year: 1900
Current value for year: 1950
Current value for year: 2000
Current value for year: 2005
Current value for year: 2010
Current value for year: 2015.0
```

Your `ObjectMultistack` class must provide following methods:

```java
package hr.fer.zemris.java.custom.scripting.exec;

public class ObjectMultistack {

	public void push(String name, ValueWrapper valueWrapper) {...}
	public ValueWrapper pop(String name) {...}
	public ValueWrapper peek(String name) {...}
	public boolean isEmpty(String name) {...}

}
```

Of course, you are free to add any private method you need. The semantic of methods `push/pop/peek` is as usual, except they are bounded to “virtual” stack defined by given name. In a way, you can think of this collection as a map that associates strings with stacks. And this virtual stacks for two different string are completely isolated from each other.

Your job is to implement this collection. However, **<u>you are not allowed</u>** to use instances of existing class `Stack` from Java Collection Framework. Instead, **you should define your inner static class** `MultistackEntry` that acts as a node of a single-linked list. Then use some implementation of interface `Map` to map names to instances of `MultistackEntry` class. Using `MultistackEntry` class you can efficiently implement simple stack-like behaviour that is needed for this homework.

Methods `pop` and `peek` should throw an appropriate exception if called upon empty stack.

Finally, you must implement `ValueWrapper` class whose structure is as follows.

- It must have a read-write property `value` of type `Object`.
- It must have a single public constructor that accepts initial value.
- It must have four arithmetic methods:
  - `public void increment(Object incValue);`
  - `public void decrement(Object decValue);`
  - `public void multiply(Object mulValue);`
  - `public void divide(Object divValue);`
- It must have additional numerical comparison method:
  - `public int numCompare(Object withValue);`

All four arithmetic operation modify current value. However, there is a catch. Although instances of `ValueWrapper` do allow us to work with objects of any types, if we call arithmetic operations, it should be obvious that some restrictions will apply at the moment of method call (e.g. we can not multiply a network socket with a GUI window). Here are the rules. Please observe that we have to consider three elements:

1. what is the type of currently stored value in `ValueWrapper` object,

2. what is the type of argument provided, and
3. what will be the type of new value that will be stored as a result of invoked operation.

We define that allowed values for current content of `ValueWrapper` object and for argument are `null` and instances of `Integer`, `Double` and `String` classes. If this is not the case, throw a RuntimeException with explanation.

Further, if any of current value or argument is `null`, you should treat that value as being equal to `Integer` with value 0.

If current value and argument are not `null`, they can be instances of `Integer`, `Double` or `String`. For each value that is `String`, you should check if `String` literal is decimal value (i.e. does it have somewhere a symbol '.' or 'E'). If it is a decimal value, treat it as such; otherwise, treat it as an `Integer` (if conversion fails, you are free to throw `RuntimeException` since the result of operation is undefined anyway).

Now, if either current value or argument is `Double`, operation should be performed on `Double`s, and result should be stored as an instance of `Double`. If not, both arguments must be `Integer`s so operation should be performed on `Integer`s and result stored as an `Integer`.

If you carefully examine the output of program `ObjectMultistackDemo`, you will see this happening!

Please note, you have four methods that must somehow determine on which kind of arguments it will perform the selected operation and what will be the result – please do not copy&paste appropriate code four times; instead, isolate it in one (or more) private methods that will prepare what is necessary for these four methods to do its job.

Rules for `numCompare` method are similar. This method does not perform any change. It perform numerical comparison between currently stored value in `ValueWrapper` and given argument. The method returns an integer less than zero if currently stored value is smaller than argument, an integer greater than zero if currently stored value is larger than argument or an integer 0 if they are equal.

- If both values are `null`, treat them as equal.
- If one is `null` and the other is not, treat the `null`-value being equal to an integer with value 0.
- Otherwise, promote both values to same type as described for arithmetic methods and then perform the comparison.

Problem 3.

You are to write a program that will allow user to encrypt/decrypt given file using AES crypto-algorithm and 128-bit encryption key or calculate and check SHA-1 file digest. Since this kind of cryptography works with binary data, use octet-stream Java based API for reading and writing of files. What needs to be programmed is illustrated by following use cases:

```
java hr.fer.zemris.java.tecaj.hw6.crypto.Crypto checksha file1.pdf
Please provide expected sha signature for file1.pdf:
> fa50c8f37ee0f77ef20149061b7414768291cb13
Digesting completed. Digest of file1.pdf matches expected digest.

java hr.fer.zemris.java.tecaj.hw6.crypto.Crypto checksha file1.pdf
Please provide expected sha signature for file1.pdf:
> da50c8f37ee0f77ef20149061b7414768291cb13
Digesting completed. Digest of file1.pdf does not match the expected digest. Digest
was: fa50c8f37ee0f77ef20149061b7414768291cb13

java hr.fer.zemris.java.tecaj.hw6.crypto.Crypto encrypt file1.pdf file1.crypted.pdf
Please provide password as hex-encoded text (16 bytes, i.e. 32 hex-digits):
> a52217e3ee213ef1ffdee3a192e2ac7e
Please provide initialization vector as hex-encoded text (32 hex-digits):
> 000102030405060708090a0b0c0d0e0f
Encryption completed. Generated file file1.crypted.pdf based on file file1.pdf.

java hr.fer.zemris.java.tecaj.hw6.crypto.Crypto decrypt file1.crypted.pdf file1.pdf
Please provide password as hex-encoded text (16 bytes, i.e. 32 hex-digits):
> a52217e3ee213ef1ffdee3a192e2ac7e
Please provide initialization vector as hex-encoded text (32 hex-digits):
> 000102030405060708090a0b0c0d0e0f
Decryption completed. Generated file file1.pdf based on file file.crypted.pdf.
```

Please consult the following references:

http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html#MessageDigest
http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html#Cipher
http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html#MDEx
http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html#SimpleEncrEx

Encryption keys and initialization vectors are byte-arrays each having 16 bytes. In the above examples it is expected from the user to provide these as hex-encoded texts.

Implement these methods. To obtain properly initialized `Cipher` object, use following code snippet:

```java
SecretKeySpec keySpec = new SecretKeySpec(hextobyte(keyText), "AES");
AlgorithmParameterSpec paramSpec = new IvParameterSpec(hextobyte(ivText));
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
cipher.init(encrypt ? Cipher.ENCRYPT_MODE : Cipher.DECRYPT_MODE, keySpec, paramSpec);
```

Method `hextobyte(keyText)` should take hex-encoded String and return appropriate `byte[]`. You are, of course, expected to write this method as well.

Please note, **you are not allowed** to use `CipherInputStream` or `CipherOutputStream` (or any of its subclasses); you are required to implement encryption/decryption directly using `Cipher` object and a series of `update/update/update/...` completed by `doFinal()`. Also, you are not allowed to read a complete file into memory, then encrypt/decrypt it and then write the result back to disk since files can be huge. You are only allowed to read a reasonable amount of file into memory at each single time (for example, 4k). The same goes for constructing the resulting file. For reading file you must use an instance of `FileInputStream` and for writing an instance of `FileOutputStream`. Ensure that they are both buffered.

Problem 4.

Download file `hw06.bin` and save it in you current directory. Now run your program:

```
java hr.fer.zemris.java.tecaj.hw6.crypto.Crypto checksha hw06.bin
Please provide expected sha signature for hw06.bin:
> 91373c6a381f4c405bee37f21344fda2e1611267
Digesting completed. Digest of hw06.bin matches expected digest.
```

If you obtain different result, there is something wrong; either the file `hw06.bin` is corrupted (redownload it again) or you have bug in your program (fix it). When you do obtain result as expected, run following command:

```
java hr.fer.zemris.java.tecaj.hw6.crypto.Crypto decrypt hw06.bin hw06-2.pdf
Please provide password as hex-encoded text (16 bytes, i.e. 32 hex-digits):
> a52217e3ee213ef1ffdee3a192e2ac7e
Please provide initialization vector as hex-encoded text (32 hex-digits):
> 000102030405060708090a0b0c0d0e0f
Decryption completed. Generated file hw06-2.pdf based on file hw06.bin.
```

Open the file you just generated, read it and proceed as instructed by the text in that file.

Important notes

Solve all of the problems in a single Eclipse project. Configure Eclipse to use two source directories: `src/main/java` for your source files and `src/test/java` for sources files of unit tests.

You are required to write the adequate number of unit tests for developed instructions in problem 2.

You must equip your project with `build.xml` script so that the project can be build from the command line. In that script, you must integrate all of the quality-checking tools which I have described in the book (by integrate, I mean have a new target for each tool so that you can call each tool separately). It should be possible to run at least the following targets: `init`, `compile`, `compile-tests`, `run-tests`, `quality`, `reports`, `clean`.

Target `quality` must run unit tests and all of the quality checks (i.e. *checkstyle*, *pmd*, *findbugs*): make it just a target which depends on all tool-targets. Unit tests must be run with the code coverage analysis. Target `reports` is a wrapper that will run all of the unit tests, the quality checks and the javadoc generation.

All of the classes in all three problems should have appropriate javadoc.

Please note. You can consult with your peers and exchange ideas about this homework *before* you start actual coding. Once you open you IDE and start coding, consultations with others (except with me) will be regarded as cheating. You can not use any of preexisting code or libraries which is not part of Java standard edition (Java SE) unless explicitly provided by me. This means that from this point on, you can use Java Collection Framework classes or its derivatives (moreover, I recommend it). Document your code!

In order to solve this homework, create a blank Eclipse Java Project and write your code inside. You must name your project's main directory (which is usually also the project name) `HW06-`*yourJMBAG*; for example, if your JMBAG is 0012345678, the project name and the directory name must be `HW06-0012345678`. Once you are done, export the project as a ZIP archive and upload this archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted. Deadline is April 22th 2014. at 8 PM.