5. homework assignment; JAVA, Academic year 2014/2015; FER

First: read last page. I mean it! You are back? OK. This homework consists of two problems.

Problem 1.

Write a simple database emulator. Put the implementation classes in package `hr.fer.zemris.java.tecaj.hw5.db`. In repository on Ferko you will find a file named `database.txt`. It is a simple textual form in which each row contains the data for single student. Atributes are: *jmbag*, *lastName*, *firstName*, *finalGrade*. Name your program `StudentDB`. When started, program reads the data from current directory from file `database.txt`. In order to achieve this, write a class `StudentRecord`; instances of this class will represent records for each student. Assume that there can not exist multiple records for the same student. Implement `equals` and `hashCode` methods so that the two students are treated as equal if jmbags are equal.

Write the class `StudentDatabase`: its constructor must get a list of `String` objects (the content of `database.txt`). It must create an internal list of student records. Additionally, it must create *an index* for fast retrieval of student records when jmbag is known (use map for this). Add the following two public methods to this class as well:

```
public StudentRecord forJMBAG(String jmbag);
public List<StudentRecord> filter(IFilter filter);
```

The first method uses index to obtain requested record in O(1); if record does not exists, the method returns `null`.

The second method accepts a reference to an object which is an instance of `IFilter` interface:

```
public interface IFilter {
   public boolean accepts(StudentRecord record);
}
```

The method `filter` in `StudentDatabase` loops through all student records in its internal list; it calls `accepts` method on given filter-object with current record; each record for which `accepts` returns **`true`** is added to temporary list and this list is then returned by the `filter` method.

The system reads user input from console. You must support a single command: **query**. Here are several legal examples of the this command.

```
query jmbag="0000000003"
query lastName="B*"
query firstName>"A" and lastName="B*ć"
query firstName>"A" and firstName<"C" and lastName="B*ć" and jmbag>"0000000002"
```

Only `jmbag`, `lastName` and `firstName` are supported for querying. If query contains expression of form `jmbag=SOMETHING`, you must resolve this query using index, so that is can be resolved in O(1). Multiple conditions must always be connected by and (don't write support for any other operators, negations or grouping using parentheses; this decision is made in order to simplify query parsing and execution). String literals must be written in quotes, and quote can not be written in string (so no escapeing is needed; another simplification). You must support following six comparison operators: >, <, >=, <=, =, `!=`. On the left side of a comparison operator a field name is required and on the left side string literal. This is OK:

`firstName="Ante"` but following examples are invald: `firstName=lastName`, `"Ante"=firstName`.

If comparison = is used, string literal can contain wildcard `*` (other comparisons don't support this and tread `*` as regular character). This character, if present, can occur at most once, but it can be at the beginning, at the end or somewhere in the middle). If user enters more wildcard characters, throw an exception (and catch it where appropriate and write error message to user; don't terminate the program).

Please observe that query is composed from one or more conditional expressions. Each conditional expression has field name, operator symbol, string literal. Since only `and` is allowed for expression combining, the whole query can be an array (or list) of conditional expressions.

Define a strategy[1] named `IComparisonOperator` with one method:

```
public boolean satisfied(String value1, String value2);
```

Implement concrete strategies for each comparison operator you are required to support. Arguments of previous method are two string literals (not field names).

Then define another strategy: `IFieldValueGetter` which is responsible for obtaining a requested field value from `StudentRecord`. This interface must define the following method:

```
public String get(StudentRecord record);
```

Write three concrete strategies: one for each `String` field of `StudentRecord` class.

Finally, model the complete conditional expression with the class `ConditionalExpression` which gets through constructor three arguments: a reference to `IFieldValueGetter` strategy, a reference to string literal and a reference to `IComparisonOperator` strategy. Add getters for these properties (and everything else you deem appropriate). If done correctly, you will be able to use code snippet such as the one given below:

```
ConditionalExpression expr = new ConditionalExpression(
  new LastNameFieldGetter(),
  "Bos*",
  new WildCardEqualsCondition()
);

StudentRecord record = getSomehowOneRecord();

boolean recordSatisfies = expr.getComparisonOperator().satisfied(
  expr.getFieldGetter().get(record),  // returns lastName from given record
  expr.getStringLiteral()             // returns "Bos*"
);
```

Note: class names in previous example are made up; you are free to use any (consistent and meaningful) naming.

Create a class `QueryFilter` which implements `IFilter`. It has a single public constructor which receives one argument: query string (everything user entered **after** `query` keyword). For query parsing you can write additional classes.
Add to `QueryFilter` class (not to the interface `IFilter`) following method:

```
public Optional<String> getJMBAG() { … }
```

---

1 See: http://en.wikipedia.org/wiki/Strategy_pattern as well as text in book (there is glossary; look up Strategy desing pattern)

If during parsing of the query an expression `jmbag=SOMETHING` (with no wildcards) is found (or more expressions like that but with same literal), constructor of `QueryFilter` must remember this and must offer this JMBAG in `getJMBAG()` method.

This is done so you can decide if index can be used for record retrieval or filtering is required. For each query where index is used, write this information to console.

Here is an example of interaction with program, as well as expected output and formatting. Symbol for prompt which program writes out is "`> `".

```
> query jmbag="0000000003"
Using index for record retrieval.
+============+========+========+===+
| 0000000003 | Bosnić | Andrea | 4 |
+============+========+========+===+
Records selected: 1

> query lastName="B*"
+============+===========+===========+===+
| 0000000002 | Bakamović | Petra     | 3 |
| 0000000003 | Bosnić    | Andrea    | 4 |
| 0000000004 | Božić     | Marin     | 5 |
| 0000000005 | Brezović  | Jusufadis | 2 |
+============+===========+===========+===+
Records selected: 4

> query lastName="Be*"
Records selected: 0
```

Please observe that the table is automatically resized and that the columns must be aligned using spaces. The order in which the records are written is the same as the order in which they are given in database file.

If users input is invalid, write an appropriate message. Allow user to write spaces: following is also OK:

```
query     lastName="Be*"
query lastName    ="Be*"
query lastName=   "Be*"
query lastName  =    "Be*"
```

Note: for reading from file please use the code snippet. In the example, we are reading the content of `prva.txt` located in current directory:

```
List<String> lines = Files.readAllLines(
    Paths.get("./prva.txt"),
    StandardCharsets.UTF_8
);
```

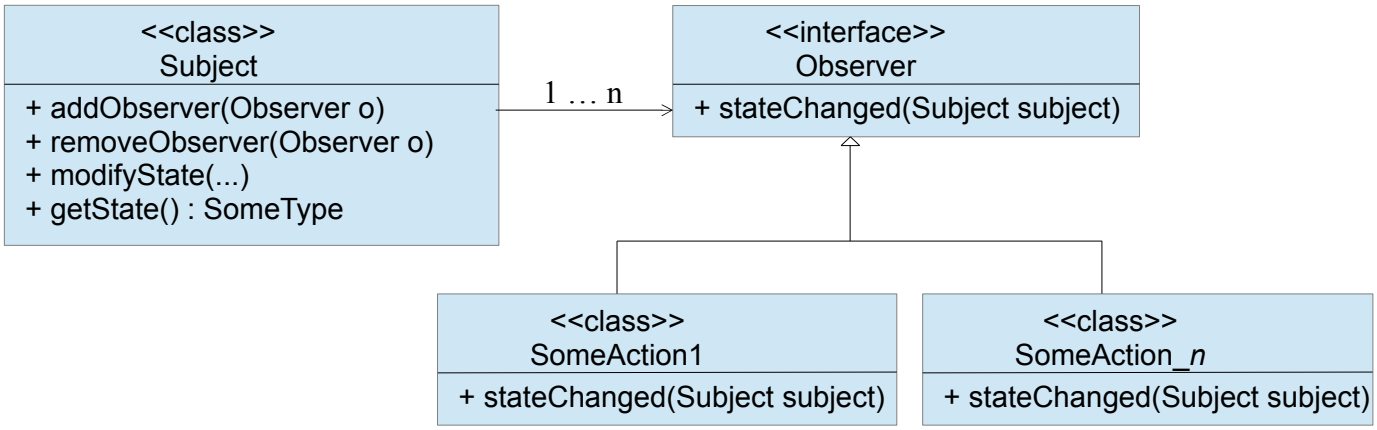
Problem 2.

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.



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.hw5.observer1;

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.hw5.observer1;

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

The main program is `ObserverExample`:

```
package hr.fer.zemris.java.tecaj.hw5.observer1;

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.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. 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. 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
```

After you finish this task, copy the content of subpackage `observer1` into `observer2`. You will continue your work here while package `problem1a` will preserve your previous solution.

Lets recapitulate what we have done so far. We have developed our `Subject` to allow a registration of multiple observers. We have defined the `Observer` interface and have developed more than one actual observer (classes that implemented the `Observer` interface). Now you will modify your code from package `observer2` 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(...)`.

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 all of the classes developed in problem 1.

You must equip your project with `build.xml` script; see previous homework for additional information on this script.

All of the classes in all of the problems should have appropriate javadoc. Failure to do so will be rewarded with mark 1.

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) `HW05-`*yourJMBAG*; for example, if your JMBAG is 0012345678, the project name and the directory name must be `HW05-0012345678`. 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 16th 2015. at 11:59 PM.