AP-24 – Programming Assignment #1- v1

November 25, 2024

This programming assignment consists of two exercises, on Java Beans and Java Reflection and Annotations, respectively.

Exercise 1 (Java Beans) – The Matching Pairs game

The **Matching Pair** game (aka **Concentration**) is a game where some cards are laid face down on a surface and two cards are flipped face up over each turn. The goal of the game is to turn over pairs of matching cards. When a matching pair is found, it is eliminated from the game, and when all cards are eliminated, the game terminates. See the following Wikipedia page for more information: https://en.wikipedia.org/wiki/Concentration_(card_game).

The goal of the assignment is to implement a board which allows one player to play the game. Optionally, support for more players can be provided. The board and the other components of the game must be realized as Java beans, and they have to interact among themselves according to the Publish-Subscribe (or Observer) design pattern, using events and event listeners as much as possible.

Functional requirements

The game starts showing the **`Board`** (a bean which extends **`JFrame`**), on which eight **`Card`**s are displayed face down, together with a **`Shuffle`** and an **`Exit`** button, a **`Controller`** label, and a **`Counter`** label.

`Card` is a bean extending **`JButton`**, it has a property **`value`** and a property **`state`** that must be ***bound*** and ***constrained***. The **`state`** property can hold one out of three values: **`excluded`** (in this case the card is **red**), **`face_down`** (the card is **green**) and **`face_up`** (the card **shows its value)**.
The **`value`** of a **`Card`** is an integer. At the beginning of the game, and each time the **`Shuffle`** button is clicked, each card is initialized with a value. The **`Board`** is responsible of providing such values, by first generating a random sequence of length 8, made of 4 integers each repeated twice. Next the **`Board`** must fire an event **`shuffle`** wrapping the sequence of values. Each **`Card`** must be registered as listeners to **`shuffle`**, so that it can take its new value from the sequence.
Clicking on a card has the effect of changing its state. ***Note***: the state of the card can only be changed by invoking the **`setState`** method, which must take into account that **`State`** is both ***bound*** and ***constrained***. If the card is **`face_down`**, clicking on it changes to **`face_up`** (thus displaying the value). Instead, if the state is **`excluded`** or **`face_up`**, it changes to **`face_down`** (but as said later, this change will be vetoed).

The **`Controller`** is a label showing the number of already matched pairs (initially 0) and implementing the logic of the game. When a first card is turned **`face_up`**, it records its value **`v1`**. When a second card is turned **`face_up`** with **`v2`**, the **`Controller`** fires a **`matched`** event with value **`true`** (and increments the displayed value) if **`v1==v2`**, otherwise with value **`false`**. The **`matched`** event has to be delayed half a second to allow the player to see **`v2.`**

Each card ignores the **matched** event unless it is **face_up**: in that case, if the value is **true** it changes state to **excluded**, otherwise to **face_down**.
The **Controller** is registered as **VetoableChangeListener** to each **Card**. It must forbid the change of **state** if it is triggered by clicking on the card when it is **excluded** or **face_up**.
The value displayed by the **Controller** must be reset to 0 every time the **shuffle** event is fired.

The **Counter** label displays the total number of times some card is turned **face_up**. It is reset to 0 every time the **shuffle** event is fired.
The **Exit** button asks a confirmation to the player, and if positive it terminates the game, otherwise it has no effect.

Other requirements

At startup the **Board** creates all the other beans, and it registers them as listeners to events as needed.
All beans should interact using the Observer design pattern (using events and event listeners). If this looks not reasonable in certain situations, other forms of interactions (e.g. invoking public methods) are allowed, but they must be clearly justified in comments along the code, where they are used.
Note that from the specification it follows that when all cards become **excluded**, they cannot change state: At that point only the **Exit** and **Shuffle** buttons are reactive.

Optional extensions

1. Make the number of cards of the game parametric. Define a constant **N** set to 4 in **Board** and use `2xN` as the number of cards everywhere. Then it must be possible to change **N** to a different value and to recompile the **Board** only, to play with a different even number of cards.
2. Add a **Challenge** label that displays the best score seen since the start of the game, that is the smallest number of moves (flipped cards) that was needed to complete one round of the game.

Solution format

Suitably commented source files for the beans and one **jar** archive for each bean.

Exercise 2 (Java Reflections/Annotations) – Unit testing with args

Write the Java program **`RunTests`** that takes as argument from the command line the name of a Java class, say **`MyClass`**. **`RunTest`** loads class **`MyClass`** using the reflection API, and it creates an instance **`t`** of **`MyClass`**. Next, it must invoke on **`t`** all the non-**`private`** instance methods of **`MyClass`** which are annotated by **`@Testable`**, checking if it satisfies the annotated **`@Specification`**. These two annotations are defined as follows (copy the text in suitable files):

- `Testable` annotation:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Testable {}
```

- `Specification` annotation:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Specification {
    String[] argTypes () default {}; //
    String[] argValues () default {};
    String resType () default "";
    String resVal () default "";
}
```

The **`@Specification`** annotation provides the list of actual arguments to be used to invoke the method, and the expected result. Array **`argTypes`** lists the argument's types (each of which can be **`"int"`**, **`"double"`**, **`"bool"`** or **`"string"`**), and array **`argValues`** lists the argument's values, using their string representation. For example, **`"42"`** represents the integer **`42`** if the corresponding type is **`"int"`**, the floating point number **`42.0`** if the type is **`"double"`**, or the string **`"42"`** if the type is **`"string"`**. The boolean values are represented by **`"true"`** and **`"false"`**. String **`"resType"`** is the expected type of the method's result (one of **`"int", "double", "bool"`** or **`"string"`**, or the empty string indicating **`void`**). String **`resVal`** is the expected result of the method. The default values of the **`@Specification`** attributes indicate that the method must be invoked with no arguments and that it does not return a value.

To report the outcome of the tests, your program **`RunTests`** must use the class **`Report.java`**. Such class cannot be modified, but a "**`package`**" statement can be added as a first line. For each **`@Testable`** method, **`RunTests`** must invoke method **`Report.report(…)`** exactly once**.** The second and the third arguments are the method name and the **`@Specification`** object annotating the method. The first argument is one of the four possible values of the enumeration **`Report.TEST_RESULT`**, describing the outcome of the test. More precisely, the first argument should be:

(1) **`Report.TEST_RESULT.WrongArgs`**, if the method to be tested cannot be invoked because the provided arguments are incorrect (e.g., the length of **`argTypes`** or of

argValues is not equal to the number of arguments of the method, or the types do not correspond, or some elements of **argValues** cannot be converted to the expected type);

(2) **Report.TEST_RESULT.WrongResultType,** if the method could be invoked using the provided arguments, but the expected result type is incorrect (e.g., the type is different from the return type of the method, or **resValue** cannot be converted to the expected type, for example. **resValue=="3a"** and **resType=="int"**);

(3) **Report.TEST_RESULT.TestFailed**, if the method can be tested (because neither (1) nor (2) hold), but the expected result is not equal to the value returned by invoking the method;

(4) **Report.TEST_RESULT.TestSucceeded**, if the method can be tested and its invocation returns the expected result.

The **RunTestsAssignment.zip** archive contains the following files to be used to debug your program:

- **Report.java**, to be used as described above;
- **MathOpsTests_simple.java**, containing some method definitions annotated for testing: the specifications are well-formed, so cases (1) and (2) above should not show up;
- **RunTests_MathOpsTests_simple.output**, a text file containing the expected output of running **RunTest** with **MathOpsTests_simple** as argument;
- **MathOpsTests.java**, containing more method definitions annotated for testing;
- **RunTests_MathOpsTests.output**, a text file containing the expected output of running **RunTest** with **MathOpsTests** as argument;

Solution format

A well-commented **RunTests.java** file. I will check that the program produces the expected output on the two test classes above.