Software Engineering I (02161)

Additional Informaton for Programming Assignment 5

Assoc. Prof. Hubert Baumeister

Spring 2015

Contents

1 Adding a Command Line Interface to the Library Application

Library Application

- The goal of programming assignment 5 is to add a user interface to the library application. The user interface will be a command line interface consisting of several screens/menus, each presenting the user with a choice of actions to choose from. For example the first screen will be

```
0) Exit
1) Login as administrator
```

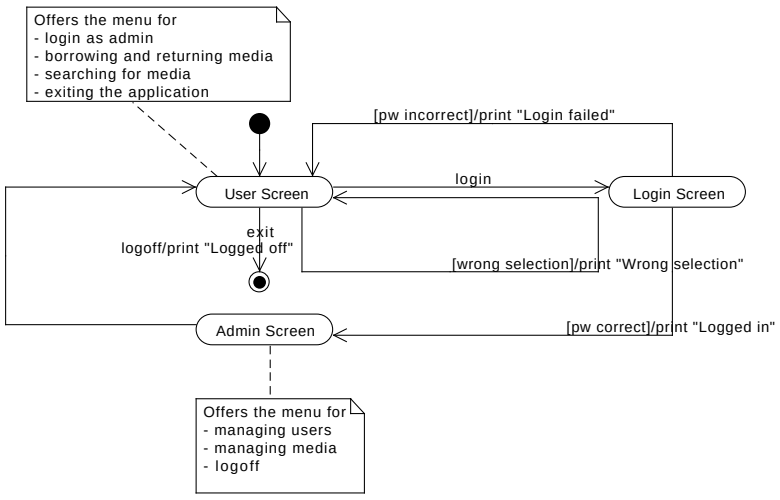
- The user can choose between 0 or 1. Depending on his choice, a different menu will be shown. For example, when the user presses 1, he will see

```
enter password
```

- where the user answers with the password, i.e., `adminadmin`, which will lead to the following menu

```
0) Logoff
1) Add new media
2) Register new user
...
```

- State machines are very good to describe screen based (i.e. modal) user interfaces. Depending on its input, the user is guided from one screen to another. Each screen representing a state in the state machine, and each user interaction a trigger for a transition.



- This state machine corresponds to the the following two screens (user inputs are writtin in *italics*)

User Screen

0) Exit

1) Login as administrator

1

Login Screen

enter password

adminadmin

Logged in.

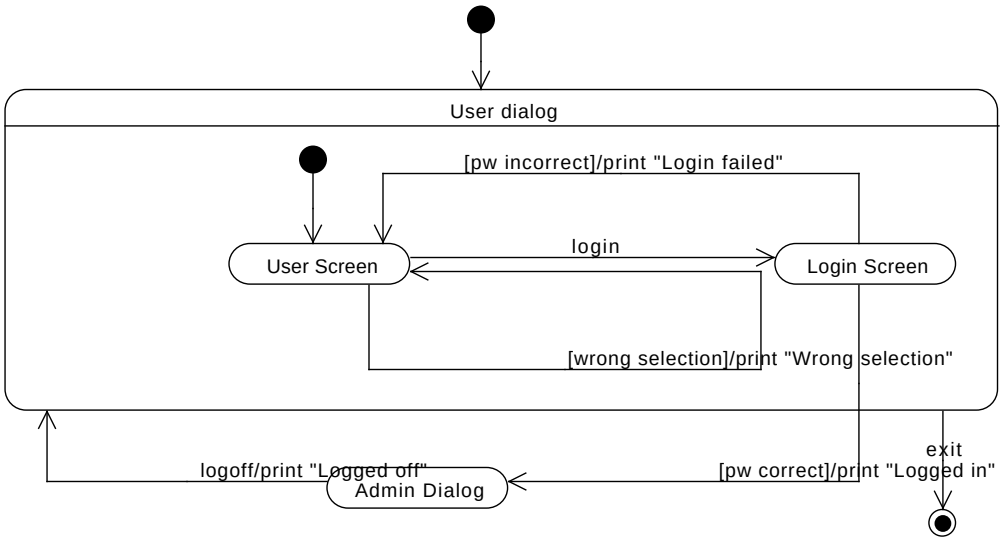
Admin Screen

0) Logoff

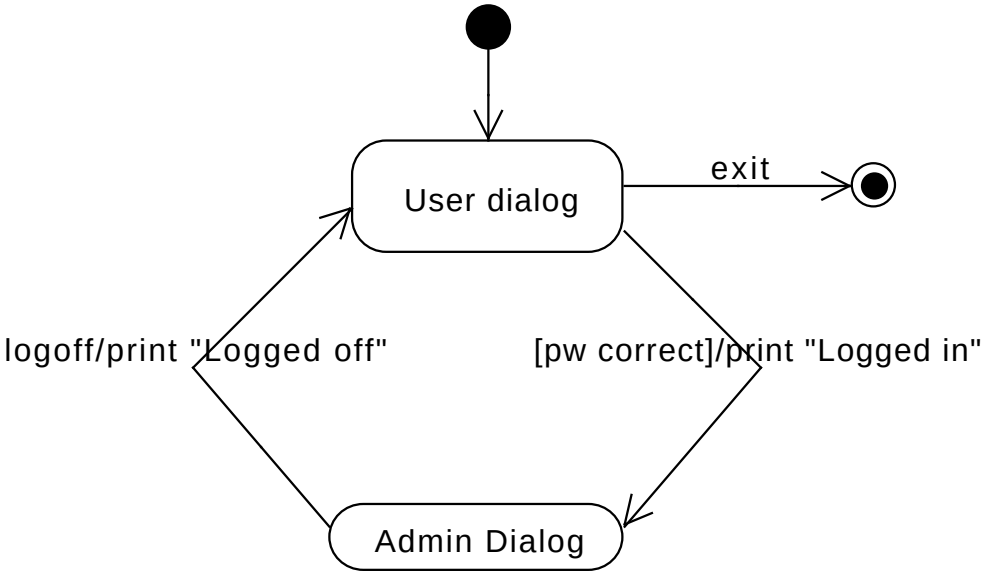
0
Logged off.

Using substates to build more complex user interfaces

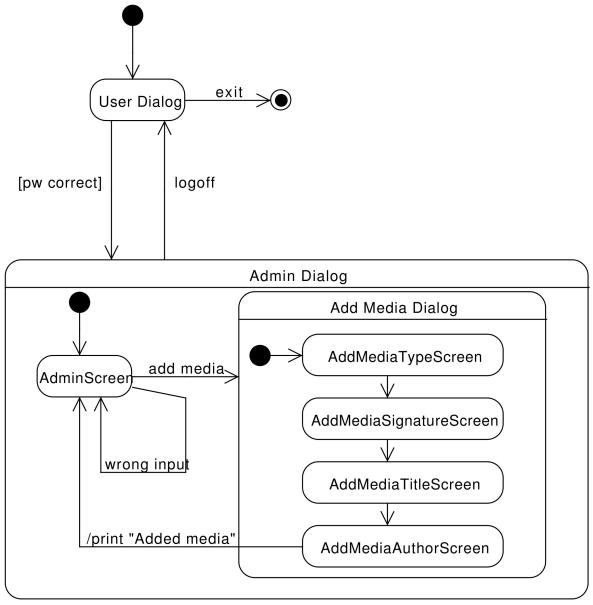
- In state machine, states can contain again state machines. States which can include other states are called substates. Substates help structure complex state diagrams (similar to subroutines)
- In the following diagram, the substate ”User dialog” is used to represent the interaction of the user with the user screen.



- We can make the state machine of the substate visible, or we can hide it.



- We can also look into more detail of the Admin Dialog substate. As one can see, this can contain a substate ”Add Media Dialog”, describing the user interface for adding a media to the library.



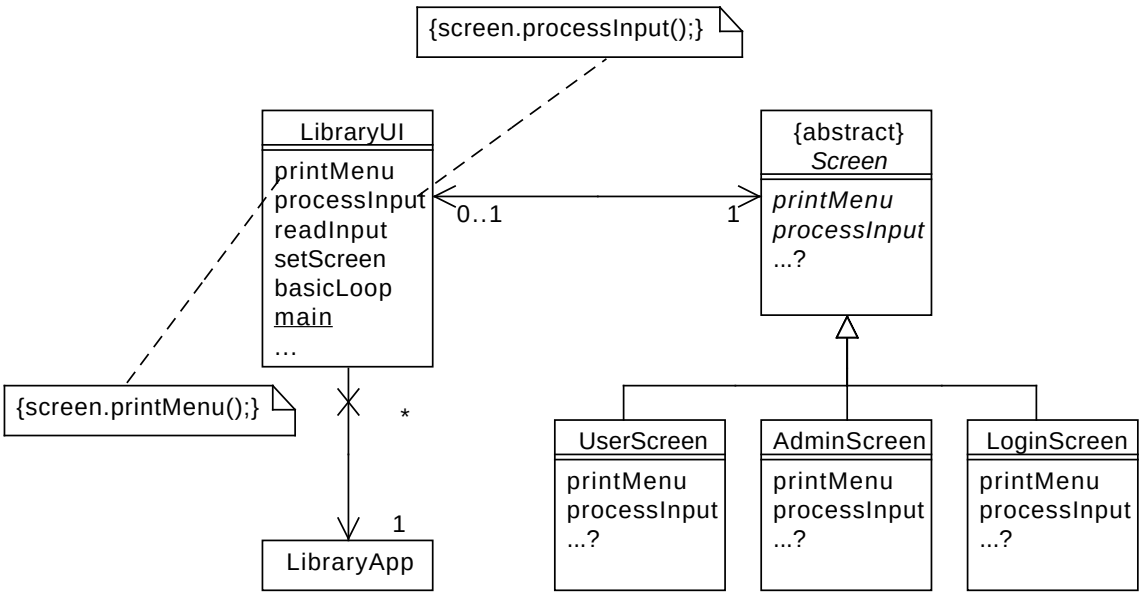
Library App user interface exercise

Program assignment 5 has two tasks:

1) Given tests for the functionality login, implement the tests using the state pattern
2) Design, test, and implement the remaining functionality of the library application
   - Design should happen by drawing state machines.
     * You should be drawing a high level state machine, showing the different types of dialogs the application has. In this diagram, you probabably don’t want to show the state machines of most substates.
     * For each dialog provide a substate with its state machine containing the design of that dialog

Library App UI: State Pattern

I have already provided you with a skeleton for the implementation of the command line interface based on the state pattern from the lecture.



Library App: main application

- The main application creates a LibraryUI object and calls the basicLoop method of that object with a BufferedReader as stream for all inputs and a PrintWriter for all outputs. The main method connects the BufferedReader to System.in (the user's input) and the PrintWriter to System.out (the console).
- The reason, why we not directly use System.in and System.out in method basicLoop, is so that we can create automatic tests. Now we can test the methods in the basicLoop with stream contents, the test method provide, and we don't require a human to input data and interpret the data written to the console.

```
public static void main(String[] args) throws IOException {
   BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
   PrintWriter out = new PrintWriter(System.out, true);
   LibraryUI ui = new LibraryUI();
   ui.basicLoop(in, out);
}
```

- The basicLoop method encodes the basic interaction of a command line UI: the system prints some information on the screen, like a menu, or a specific question, like the request to enter a password, it then reads the information from the user, and finally prints an answer on the console.
- The basicLoop proceeds until processInput decides that one of the inputs the user did, e.g. `0)` on a menu item that reads `0) Exit Application`, leads to the termination (i.e., processInput returns true).

```
public void basicLoop(BufferedReader in, PrintWriter out)
      throws IOException {
  String selection;
  do {
    printMenu(out);
    selection = readInput(in);
  } while (!processInput(selection, out));
}

public void printMenu(PrintWriter out) throws IOException {
   screen.printMenu(out);
}

public boolean processInput(String input, PrintWriter out) throws IOException {
   return screen.processInput(input,out);
}
```

Library App UI: tests

- In general, user interface code is difficult to test

→ use a layered architecture with a thin presentation layer

- However, sometimes the UI can be made testable

→ Here: the basic methods work on BufferedReader and PrintWriter instead of InputStream and PrintStream

→ Now the methods can be tested by using arbitrary streams instead of just System.in and System.out

Library App UI: tests

- The following tests tests, that already in the first menu after starting the application, there will be a menu point `0) Exit`. If the user then enters `0`, then the application will write `Exited` on the screen and terminate the basicLoop, i.e, `true`.

```
@Test
public void testTestExitApplication1() throws IOException {
  LibraryUI libraryUI = new LibraryUI();
  testScreenInteraction(libraryUI, "0) Exit", "0", "Exited.",true);
}
```

- testScreenInteraction is a helper function, which contains the actual assert statements. The first argument is the library UI to be tested. The second argument is a string that should be contained when the menu is printed. The third argument is the input of the user, and the fourth argument the response of the system to that input. The last argument is a boolean saying whether processInput returns true (then the application should be terminated) or false (i.e. the application should continue).

```
public void testScreenInteraction(LibraryUI libraryUI,
      String expectedMenu,
      String input, String expectedOutput, boolean expectedExitStatus)
      throws IOException {

  StringWriter out = new StringWriter();
  libraryUI.printMenu(new PrintWriter(out));
  assertTrue(out.toString().contains(expectedMenu));

  BufferedReader reader = new BufferedReader(new StringReader(input));
  String line = libraryUI.readInput(reader);
  assertEquals(input,line);

  out = new StringWriter();
  boolean exit = libraryUI.processInput(line, new PrintWriter(out));
  assertEquals(expectedOutput+"\n",out.toString());
  assertEquals(expectedExitStatus,exit);
}
```

Example LibraryUI implementation

- The following is an example to implement the exit functionality of the UserScreen using the state pattern and as described in the class diagram.
- The function printMenu delegates printing the screen to the printScreen method of the current screen.
- Similarily, the processInput method will delegate the processing of the input to the processInput method of the current screen
  - Note the line `out.println()`. The use of println ensures that the internal buffers are flushed, and that, if you are running your application on the console (i.e. outside of the test framework), that you see the output being printed to the console. Alternatively, you can use `out.flush()` to flush the internal buffers without creating a new line.
  - You could have the `out.prinln()` or `out.flush()` in each of the processInput methods of the screen itself, but this is a form of code duplication, you should try to prevent. For example, what happens if you forget to put the code in one of the processInput methods? Here it is in a central place and always executed after the screen's processInput method is called.

```
public class LibraryUI {
  private Screen screen;
  private LibraryApp libraryApp = new LibraryApp();

  public LibraryUI() {
    setScreen(new UserScreen());
  }

  public void printMenu(PrintWriter out) throws IOException {
    getScreen().printMenu(out);
  }

  public boolean processInput(String input, PrintWriter out) throws IOException {
    boolean exit = getScreen().processInput(input, out);
    out.println();
    return exit;
  }

  public static void main(String[] args) throws IOException {...}

  public void basicLoop(BufferedReader in, PrintWriter out) ...

  void setScreen(Screen screen) {
    this.screen = screen;
    this.screen.setLibraryUI(this);
  }
   ...
}
```

Example Screen implementation

- The following code implements not the complete UserScreen, but only the first test, the possibility to exit the application from the user screen.

```
public class UserScreen extends Screen {
  @Override
  public void printScreen(PrintWriter out) throws IOException {
    out.println("0) Exit");
  }

  @Override
  public boolean processInput(String selection, PrintWriter out) {
    if ("0".equals(selection)) {
      out.print("Exited.");
      return true;
    }
    return false;
  }
}
```

- Note that as part of the input processing, screen can change, e.g., there should be a transition from the UserScreen to the LoginScreen when the user logs in as the administrator. In this case you would use the code in processInput. This also explain, why each Screen object knows about its LibraryUI object.

```
libraryUI.setScreen(new LoginScreen());
```