EECS3311 Fall 2017
Lab Exercise 4
Extending the Peg Solitaire Game with Undo and Redo Features

Chen-Wei Wang

Due Date: 12:00 Noon, Friday, November 17

Check the Amendments section of this document regularly for changes, fixes, and clarifications.

1 Policies

- **Your (submitted or unsubmitted) solution to this lab exercise (which is not revealed to the public) remains the property of the EECS department. Do not distribute or share your code in any public media (e.g., github) in any way, shape, or form. The department reserves the right to take necessary actions upon found violations of this policy.**
- You are required to **work on your own** for this lab. **No** group partners are allowed.
- When you submit your lab, you claim that it is **solely** your work. Therefore, it is considered as **an violation of academic integrity** if you copy or share **any** parts of your Java code during **any** stages of your development.
- When assessing your submission, the instructor and TA will examine your code, and suspicious submissions will be reported to the department if necessary. **We do not tolerate academic dishonesty**, so please obey this policy strictly.
- You are entirely responsible for making your submission in time. Back up your work **periodically**, so as to minimize the damage should any sort of computer failures occur.
- The deadline is **strict** with no excuses: you receive **0** for not making your submission in time.
- You are free to work on this lab anywhere, but you are advised to test your code via your EECS account before the submission.

2 Learning Outcomes of this Lab Exercise

1. Implement the Undo/Redo Pattern.
2. Understand the separation between: 1) software features; and 2) user interface (abstract, not necessarily graphical).
3. Write unit tests to verify the correctness of your software.
4. Draw professional architectural diagram using the draw.io tool.

3 Required Readings

– Chapter 21 from OOSC2 on a design pattern supporting the undo/redo features. This pattern exploits the object-oriented notions of polymorphism and dynamic binding. You can access an electronic copy of OOSC2 from the course moodle page.

– You can also find an abundance of resources on DbC, TDD, ESpec tests, and Eiffel code examples from these two sites:

- `http://eiffel.eecs.yorku.ca`
- `https://wiki.eecs.yorku.ca/project/eiffel/`

Contents

1 Policies **1**

2 Learning Outcomes of this Lab Exercise **2**

3 Required Readings **2**

4 Problems **4**
4.1 Programming 4
4.1.1 Getting Started 4
4.1.2 Error Reporting in *SOLITAIRE_USER_INTERFACE* 5
4.2 BON Architectural Diagram 5
4.3 Report 6

5 Submission **6**

6 Amendments **8**

4 Problems

4.1 Programming

This lab exercise builds upon your work of a previous lab (i.e., the peg solitaire game). All requirements of the game remain the same as those stated in the assignment instructions. You will be assumed good knowledge about how the classes and features work, particularly those in the *BOARD* and *GAME* classes.

- You are given a version of the peg solitaire game project with working implementation.

  You must not modify any of the implementations/contracts given to you. If you find issues about the given implementations/contracts, report them to the instructor.

- Extend the peg solitaire game with the undo and redo features.

  To complete this task, **you are required to read Chapter 21 from OOSC2** on a design pattern supporting the undo/redo features. This pattern exploits the object-oriented notions of polymorphism and dynamic binding. You can access an electronic copy of OOSC2 from the course moodle page. Note that:

  - Syntax that is used in the chapter may not be up to date, in which case you are expected to look up the corresponding syntax from the resources (e.g., lecture notes, sites, *etc.*).
  - The PDF of OOSC2 is provided to you for your own study only, and you are **not** supposed to distribute this PDF to the public.

4.1.1 Getting Started

- Download the starter project archive `Lab_4.zip` from the course moodle, unzip it, and launch EStudio.
- Choose `Add Project`, browse to the project configuration file

  `Lab_4/peg_solitaire_undo_redo/peg_solitaire.ecf`, then select `Open`.

It is expected that the starter project does not compile, as you will need to properly set the relevant attributes. The starter code contains a sample solution for your assignment (i.e., for the game and board features). You are encouraged to study the contracts and implementations of this sample solution, which may help you on the upcoming lab test and exam.

In addition, there is now a new cluster named *abstract_ui* for you to implement the undo/redo features:

- The *COMMAND* class and all its descendant classes (e.g., *COMMAND_MOVE_LEFT*) are skeletons for you to implement the data abstraction of commands as discussed in the assigned chapter.
- Similarly, the *HISTORY* class is for you to implement the multi-level undo/redo mechanism as discussed in the assigned chapter.
- The *GAME_ACCESS* class provides a singleton access to a unique *GAME* object. This singleton access should ensure that the solitaire user interface, as well as the various commands initiated by the player, all share the *same* game. For example, a series of invocations on the undo/redo feature will operate on the *same* game.

Only the deferred class *COMMAND* is completed for you. For all classes in the *abstract_ui* cluster, you are expected to fill in all missing implementations and contracts. However, you must not modify the signature of any feature.

There is a simple test completed for you to get started. You are expected to test and debug your implementation thoroughly before the labtest.

4.1.2 Error Reporting in *SOLITAIRE_USER_INTERFACE*

There is a clear separation between: 1) software features; and 2) user interface (abstract, not necessarily graphical).

Software features (e.g., the *move_left* command in *GAME*) may make certain assumptions about their inputs, and state these constraints as preconditions (e.g., see the list of tagged preconditions for *move_left*). When a feature is called without its precondition being satisfied, a contract violation occurs, which prevents the body implementation of that feature from being executed.

User interface, on the hand, deals with users who can make mistakes on entering incorrect input values, in which case they would expect the interface to display a proper error message as to how they might re-enter, rather than seeing the system crashing due to exceptions. For example, when a user attempts to withdraw money from their bank account, given that they have already reached their daily maximum, they certainly would not expect the ATM machine to crash and restart, but to display an error message detailing the reason of failure.

The purpose of the *SOLITAIRE_USER_INTERFACE* class is to simulate the interface between players and the peg solitaire game. That is, features such as *new_cross_game*, *move_left*, *undo*, *etc.*, represent events such as a "button-click" or a "drop-down-menu-selection" initiated by a player, and these features should in turn invoke the corresponding features of *GAME* to change its state. Consequently, we call *SOLITAIRE_USER_INTERFACE* an **abstract interface**.

All features declared under the *Events* section in *SOLITAIRE_USER_INTERFACE* need to be implemented with error reporting (i.e., by setting the value of attribute *message* properly). For each event feature, the list of possible errors corresponds to the list of preconditions of the software feature that it may invoke. When multiple errors occur simultaneously, only the one that is first discovered in the precondition list gets reported. When there are no errors on the input values and the corresponding game feature gets invoked, the *message* attribute still needs to be properly set to signal the success. For your convenience, you do not have to define the string values of error message: features defined under the *Messages* section are meant for you to invoke in order to set the value of attribute *message* properly.

For example, for the event feature *move_left* in *SOLITAIRE_USER_INTERFACE*, before it can invoke the *move_left* feature in *GAME*, it has to check that each of the preconditions is satisfied, in the order of their definitions. Say a player enters 8 for the row and 8 for the column, then the message should be set as "Row 8 is not valid" (that is, only the first error gets reported on the *message* attribute).

4.2 BON Architectural Diagram

Using draw.io, draw a BON diagram illustrating the design of the peg solitaire game, in particular the extended undo/redo features. You are restricted to **1 page only** for the diagram. Therefore, you do not have to fit all classes into your diagram. Instead, only include critical classes that illustrate the game and the undo/redo features. Classes that should be at least included in your diagram are:

- *GAME_ACCESS*
- *SOLITAIRE_USER_INTERFACE*
- *COMMAND* and all its descendant classes
- *HISTORY*
- *BOARD*
- *GAME*

Also, for the *COMMAND*, *HISTORY*, and *GAME_ACCESS* classes, show them in the expanded (detailed) view.

- When completing your drawing, also export the diagram into a PDF.
- Move the source of your draw.io drawing (name it `solitaire.xml`) and its exported PDF (name it `solitaire.pdf`) to the `docs` directory of your project.

4.3 Report

– Compile (into a single PDF file named `Report.pdf`) including:

- A cover page that clearly indicates: 1) course (`EECS3311`); 2) semester (`Fall 2017`); 3) name; and 4) CSE login;
- **Section "Architectural Diagram"**:
  - ◇ Insert the BON diagram `solitaire.pdf` (exported from `draw.io`) for your design here.
  - ◇ Explain (in no more than 1 page) how the Undo-Redo Pattern is implemented in your submission.

5 Submission

To get ready to submit:

– Close EStudio

– Type the following command (only available via your lab account) to clean up the `EIFGENs` directory:

```
cd ~/Desktop/EECS3311_Labs/Lab_4
eclean peg_solitaire_undo_redo
```

– Make sure the directory structure of your project is identical to Fig. 1 (with **no** `EIFGENs`).

By the due date, submit via the following command:

```
cd ~/Desktop/EECS3311_Labs/Lab_4
submit 3311 lab4 peg_solitaire_undo_redo
```

A check program will be run on your submission to make sure that you pass the basic checks (e.g., the code compiles, passes the given tests, *etc*). After the check is completed, feedback will be printed on the terminal, or you can type the following command to see your feedback (and later on your marks):

```
feedback 3311 lab4
```

In case the check feedback tells you that your submitted project has errors, you must fix them and re-submit. Therefore, you may submit for as many times as you want before the submission deadline, to at least make sure that you pass all basic checks.

Note. You will receive zero for submitting a project that cannot be compiled.

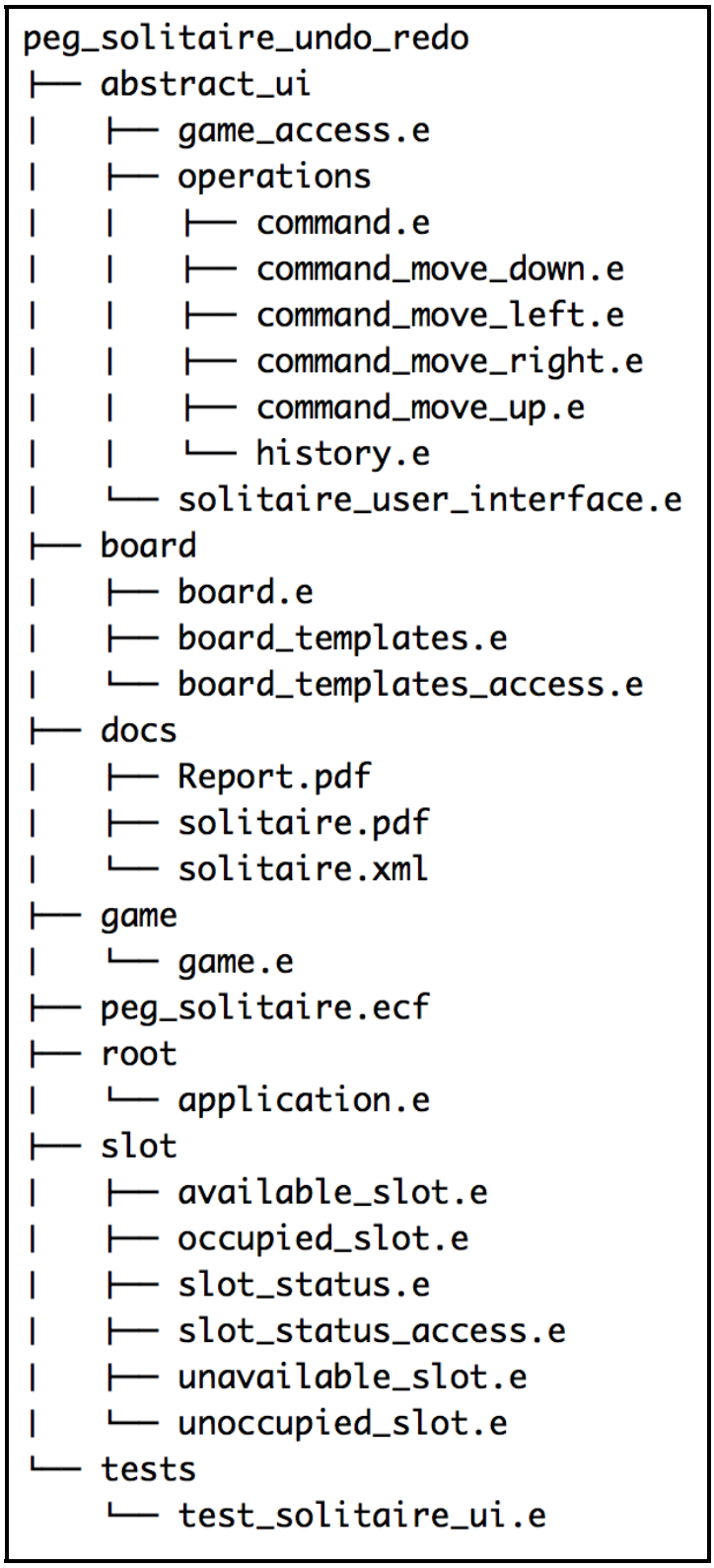

Figure 1: Project Structure of Lab 4

6 Amendments