EECS3311 Fall 2019
Lab Exercise 3
Building a Chess-Solitaire Game App using ETF (the Eiffel Testing Framework)

Chen-Wei Wang

Common Due Date for Sections A & E: 2pm, Friday, November 1

- **Check the Amendments section of this document regularly for changes, fixes, and clarifications.**
- **Ask questions on the course forum on the moodle site for Sections A and E.**

1 Policies

- **Your (submitted or un-submitted) 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., a non-private Github repository) 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 Eiffel code during **any** stages of your development.
  - When assessing your submission, the instructor and TA may examine your code, and suspicious submissions will be reported to the department/faculty if necessary. **We do not tolerate academic dishonesty**, so please obey this policy strictly.
- You are entirely responsible for making your submission in time.
  - You may submit **multiple times** prior to the deadline: only the last submission before the deadline will be graded.
  - Practice submitting your project early **even before it is in its final form**.
  - No excuses will be accepted for failing to submit shortly before the deadline.
  - Back up your work **periodically**, so as to minimize the damage should any sort of computer failures occur. Follow this tutorial series on setting up a **private** Github repository for your Eiffel projects.
  - The deadline is **strict** with no excuses: you receive **0** for not making your electronic submission in time.
  - Emailing your solutions to the instruction or TAs will not be acceptable.
- You are free to work on this lab on your own machine(s), but you are responsible for testing your code at a Prims lab machine before the submission.

Contents

2 Learning Outcomes of this Lab Exercise

1. Work with a separation of abstract user interface and business model logic.
2. Design your own classes.
3. Write unit tests to verify the correctness of your software.
4. Draw professional architectural diagram using the draw.io tool.

3 Required Tutorials

1. **Tutorial Videos on ETF**
   - Before starting the above tutorial videos, set up a starter project as follows.
   - Once you login into your Prism account, follow the steps below to get settled for the tutorial videos:

   1.1 Type the following commands to create a new subdirectory `ETF/bank` in your workspace (assuming that a directory `eecs3311-workspace` is on your Desktop):

   ```
   cd ~/Desktop/eecs3311-workspace
   mkdir ETF
   mkdir ETF/bank
   ```

   1.2 Inside the subdirectory `bank`, create a plain text file `bank-events.txt`:

   ```
   -- declaration of system name
   system bank

   -- declaration of event signatures

   new(id: STRING)
           -- create a new bank account for "id"
   deposit(id: STRING; amount: INTEGER)
           -- deposit "amount" into the account of "id"
   withdraw(id: STRING; amount: INTEGER)
           -- withdraw "amount" from the account of "id"
   transfer(id1: STRING; id2: STRING; amount: INTEGER)
           -- transfer "amount" from the account of "id1" to that of "id2"
   ```

   1.3 Run the following command from your Prism account:

   Noice that we do not distribute executables of the `etf` tool (for generating a starter project). You can get access to the ETF generator via your Prism account.

   ```
   cd ~/Desktop/eecs3311-workspace/ETF/bank
   etf -new bank-events.txt .
   ```

   1.4 A list of files should be automatically generated.

   1.5 If you decide to work through the tutorial videos on your Prism account, you may now get started.

   1.6 if you decide to work on you own computer:
   - Compress the generated starter project and transfer the zip file to your own computer:

```
cd ~/Desktop/eecs3311-workspace/ETF
zip -r bank.zip bank
```

- Proceed to Section 4 to set up the mathmodels library, which is required for compiling the generated ETF starter project.

2. Link to a Written Tutorial: **Tutorial on ETF: a Bank Application**

4 Working from Home

– To generate the ETF project, you must use the `etf` command that is available on your Prism account.

– For a generated ETF project to compile on your machine, you need to first download a library called `MATHMODELS`, and then set a environment variable `MATHMODELS` which points to the location of its download. See your Lab0 instructions for details.

5 Grading Criterion of this Lab

- When grading your submission (separate from the BON diagrams), your ETF project will be compiled and built from scratch, and then executed on a number of acceptance tests (similar to `at01.txt`, `at02.txt`, ..., `at07.txt` given to you).
- Each acceptance test is considered as **passing only if** the output generated by your program is character-by-character identical to that generated by the *oracle*. No partial marks will be given to a test case even if the output difference is as small as a single character.
- It is therefore critical for you to always switch to the command-line mode of your ETF project and use either `diff` or `meld` to compare its output and that of the *oracle*.

6 Problem

We consider the *chess-solitaire* game, a one-player game on a 4×4 board (see Figure 1a). Row indices start with 1 and increase as we move downwards vertically. Similarly, column indices start with 1 and increase as we move rightwards horizontally. A *slot* (r, c) on the board refers to the intersection of some valid row r and column c (see Figure 1b). An *occupied* slot is placed with a chess piece; otherwise, it is *unoccupied*.

```
....
....
....
....
```

(a) A Board with No Occupied Slots

```
(1,1)(1,2)(1,3)(1,4)
(2,1)(2,2)(2,3)(2,4)
(3,1)(3,2)(3,3)(3,4)
(4,1)(4,2)(4,3)(4,4)
```

(b) Referencing Slots on the Board

Figure 1: Representing the 4×4 Board for a Chess-Solitaire Game

6.1 Rules of the Game

1. Before a game starts, the board is set up with chess pieces of various kinds: kings, queens, knights, bishops, rooks, and pawns. Multiple chess pieces of the same kind may be placed on distinct slots.
2. Each kind of chess pieces has a set of *possible* moves (see below).
3. A *possible* move is *valid* if it can *capture* another chess without being blocked on the way of that move.
   - A *possible* move is not necessarily a *valid* move (if it cannot capture a chess piece or it is blocked by some other chess pieces on the way of the move).
   - A *valid* move is always a *possible* move.
4. The player wins if they can keep making *valid* moves such that that one chess piece is left on the board.
5. The player loses if there are no *valid* moves but more than one chess pieces are left on the board.

6.2 Possible Chess Movements

Table 1 summarizes the rules of possible movements for each kind of chess, and gives examples where the chess is placed at slot `(2, 2)`. We use `K` to denote king, `Q` for queen, `N` for knight, `B` for bishop, `R` for rook, and `P` for pawn. We also use `+` to denote the slot that can be reached by a *possible* (but not necessarily *valid*) move. **Examples given in Table 1 are incomplete: you should consider the chess placed at all other slots on the board.**

Rules on Possible Moves	Examples

King may move exactly one slot horizontally, vertically, or diagonally.	`+++.`
	`+K+.`
	`+++.`
	`....`

Queen may move across any number of unoccupied slots horizontally, vertically, or diagonally.	`+++.`
	`+Q++`
	`+++.`
	`.+.+`

Knight may move in an "L" shape or some rotated "L" shape. More precisely, it may move to a slot that is two slots away vertically and one slot horizontally, or to a slot that is one slot away vertically and two slots horizontally.
For simplicity, we restrict that knights can only first move vertically, then move horizontally: either **1)** vertically (northwards or southwards) for one slot, then horizontally (eastwards or westwards) for two slots; or **2)** vertically for two slots, then horizontally for one slot. For example: e.g., consider the knight at (2, 2), it may move to (1, 4) only by first moving vertically to (1, 2), then moving horizontally to (1, 4). This restriction will make it easier for you to determine if a possible move is invalid due to some block.
	`...+`
	`.N..`
	`...+`
	`+.+.`

Bishop may move across any number of unoccupied slots diagonally.	`+.+.`
	`.B..`
	`+.+.`
	`...+`

Rook may move across any number of unoccupied slots horizontally or vertically.	`.+..`
	`+R++`
	`.+..`
	`.+..`

Pawn may move diagonally up one slot.	`+.+.`
	`.P..`
	`....`
	`....`

Table 1: Rules and Example Chess Moves

6.3 Assumptions

For simplicity, we assume that:

- Before a game starts, the board is set up such that there is at least one solution to it. That is, there is no need for your app to check if after invoking `start_game`, there is at least one series of valid movements that will lead to a single chess piece left.
- Knights can only first move vertically, then move horizontally. See Table 1 for details.This restriction will make it easier for you to determine if a possible move is invalid due to some block.

6.4 An Online Version of the Game

An online version of this game can be found here:

`https://www.thinkfun.com/play-online/solitaire-chess/?src=YouTube`

However, note rules of the game have been simplified for this lab (e.g., movement of *knight*). Always consult with the *oracle* program given to you, or report possible errors about the *oracle* to `jackie@eecs.yorku.ca`.

7 Abstract User Interface

Customers need not know details of your design of classes and features. Instead, there is an agreed interface for customers to specify the chess pieces they wish to set up and how they should move. This is why we are using ETF: customers only need to be familiar with the list of *events*, defined in the plain text file below that is used to generate the customized ETF for your project. We assume the following abstract user interface:

```
system chess_solitaire

type CHESS = {
  K, -- King
  Q, -- Queen
  N, -- Knight
  B, -- Bishop
  R, -- Rook
  P  -- Pawn
}

setup_chess(c: CHESS; row: INTEGER; col: INTEGER)
        -- Set up by adding a chess at (‘row‘, ‘col‘).
start_game
        -- Start the game after setting up chess pieces.
reset_game
        -- Start to setup a new game.

move_and_capture(r1: INTEGER; c1: INTEGER; r2: INTEGER; c2: INTEGER)
        -- Move the chess at (‘r1‘, ‘c1‘), in a way that is valid,
        -- and capture the chess at (‘r2‘, ‘c2‘).

moves(row: INTEGER; col: INTEGER)
        -- Show all possible moves of the chess located at (‘row‘, ‘col‘),
        -- including those that are not valid.
```

8 Outputting the Abstract State

For the purpose of using your implemented Chess-Solitaire game, users need to be informed of: **1)** whether a new game is being set up; **2)** whether a game, already started, is over (when it is certain that the player wins or loses); and **3)** state of the 4×4 board. These three pieces of information constitute the *abstract*[1] state of the Chess-Solitaire game.

As an example, consider the following acceptance test (which is given to you as the file `at04.txt`):

[1]The term “abstract” here suggests that we show only the relevant information to users, by filtering out all other (implementation-related) details of your software.

```
-- This use case file shows a simple success.

-- Set up some chess pieces for the game
setup_chess (N, 2, 1)
setup_chess (R, 1, 4)
setup_chess (P, 2, 4)
setup_chess (B, 4, 3)

start_game

-- show all possible (but not necessarily valid) moves of B@(4,3)
moves(4, 3)
move_and_capture(4, 3, 2, 1)
-- show all possible (but not necessarily valid) moves of R@(1,4)
moves(1, 4)
move_and_capture(1, 4, 2, 4)
-- show all possible (but not necessarily valid) moves of R@(2,4)
moves(2, 4)
move_and_capture(2, 4, 2, 1)

-- Error: game already over
moves(2, 1)
-- Error: game already over
move_and_capture(2, 1, 3, 3)
```

The above acceptance test, when being executed by a correctly-implemented Chess-Solitaire game app, has the expected output as specified in the file `at04.expected.txt`:

```
  # of chess pieces on board: 0
  Game being Setup...
  ....
  ....
  ....
  ....
->setup_chess(N,2,1)
  # of chess pieces on board: 1
  Game being Setup...
  ....
  N...
  ....
  ....
->setup_chess(R,1,4)
  # of chess pieces on board: 2
  Game being Setup...
  ...R
  N...
  ....
  ....
->setup_chess(P,2,4)
  # of chess pieces on board: 3
  Game being Setup...
  ...R
```

```
  N..P
  ....
  ....
->setup_chess(B,4,3)
  # of chess pieces on board: 4
  Game being Setup...
  ...R
  N..P
  ....
  ..B.
->start_game
  # of chess pieces on board: 4
  Game In Progress...
  ...R
  N..P
  ....
  ..B.
->moves(4,3)
  # of chess pieces on board: 4
  Game In Progress...
  ....
  +...
  .+.+
  ..B.
->move_and_capture(4,3,2,1)
  # of chess pieces on board: 3
  Game In Progress...
  ...R
  B..P
  ....
  ....
->moves(1,4)
  # of chess pieces on board: 3
  Game In Progress...
  +++R
  ...+
  ...+
  ...+
->move_and_capture(1,4,2,4)
  # of chess pieces on board: 2
  Game In Progress...
  ....
  B..R
  ....
  ....
->moves(2,4)
  # of chess pieces on board: 2
  Game In Progress...
  ...+
  +++R
  ...+
  ...+
->move_and_capture(2,4,2,1)
  # of chess pieces on board: 1
```

```
  Game Over: You Win!
  ....
  R...
  ....
  ....
->moves(2,1)
  # of chess pieces on board: 1
  Error: Game already over
  ....
  R...
  ....
  ....
->move_and_capture(2,1,3,3)
  # of chess pieces on board: 1
  Error: Game already over
  ....
  R...
  ....
  ....
```

In the above expected output file, the occurrence of each event is preceded by "dash-greater-than" (->). The initial state of the Chess-Solitaire game (before the first event occurs) is one where the board contains all `.` (i.e., unoccupied slots). After the occurrence of each event, your software is also expected to display its resulting post-state. Observe that **for any two consecutive event occurrences, the post-state of the earlier event occurrence is simultaneously the pre-state of the later event occurrence.**

9 Error Reporting

All possible errors should be reflected as feature preconditions in the `model` cluster. However, reporting the violations of these preconditions (as errors) must be done on the side of the abstract user interface (i.e., the corresponding descendant class of `ETF_COMMAND`). Table 2 lists each possible error message and its applicable events, where `r`, `c`, `r1`, `c1`, `r2`, and `c2` all should be replaced by the corresponding input integer values.

When an error or multiple errors occur, exactly one error message is reported back to the user, whereas the state of your Chess-Solitaire game *must* remain unchanged. When multiple errors occur for an event, the highest error in Table 2 is reported. For example, invoking the event `move_and_check(1, 5, 2, 6)` (where source column number `5` and target column number `6` are both invalid) when the game has not yet started should result in: `Error: Game not yet started`. As another example, invoking the same event `move_and_check(1, 5, 2, 6)` when the game has already started should result in: `Error: (1, 5) not a valid slot`.

10 Getting Started

First of all, make sure you have already acquired the basic knowledge about the Eiffel Testing Framework (ETF) as detailed in Section 3.

Download the **`chess_solitaire.zip`** file from the course moodle page and unzip it. The text file `chess-solitaire-events.txt` is for you to generate the starter ETF project for your Chess-Solitaire game application. The seven input files (e.g., `at01.txt`, `at02.txt`, ..., `at07.txt`) are **example** use cases for you to test your software. The seven expected output files (e.g., `at01.expected.txt`, `at02.expected.txt`, ..., `at07.expected.txt`) contain outputs that your software must produce to match. You are advised to, before start coding, study the given expected output files carefully, in order to obtain certain reasonable grasp of how your Chess-Solitaire game app is supposed to behaviour.

Message	Triggering Events
`Error: Game already started`	`setup_chess(_, r, c)` `start_game`

`Error: Game not yet started`	`move_and_capture(r1, c1, r2, c2)`
	`moves(r, c)`
	`reset_game`

`Error: Game already over`	`move_and_capture(r1, c1, r2, c2)` `moves(r, c)`

`Error: (r, c) not a valid slot`
`Error: (r1, c1) not a valid slot`
`Error: (r2, c2) not a valid slot`
	`setup_chess(_, r, c)`
	`move_and_capture(r1, c1, r2, c2)`
	`moves(r, c)`

`Error: Slot @ (r, c) already occupied`	`setup_chess(_, r, c)`

`Error: Slot @ (r, c) not occupied`
`Error: Slot @ (r1, c1) not occupied`
`Error: Slot @ (r2, c2) not occupied`
	`move_and_capture(r1, c1, r2, c2)`
	`moves(r, c)`

`Error: Invalid move from (r1, c1) to (r2, c2)`	`move_and_capture(r1, c1, r2, c2)`
`Error: Block exists between (r1, c1) and (r2, c2)`	`move_and_capture(r1, c1, r2, c2)`

Table 2: Messages: String Values and Triggering Events

All your development will go into this downloaded `chess-solitiare` directory, and when you make the submission, you must submit this directory. To begin your development, follow these steps:

1. Open a new command-line terminal. Change the current directory into this downloaded `chess_solitaire` directory, type the following command to generate the ETF project:

   ```
   etf -new chess-solitaire-events.txt .
   ```

   Notice that there is a dot (`.`) at the end to denote the current directory.

2. There are two `chess\_solitaire` directories: one is the top-level, downloaded directory that contains all generated ETF code and your development; the other is the sub-directory that contains the `model` cluster. **When you submit, make sure that you submit the top-level `chess_solitaire` directory.**

3. Open the generated project in Eiffel Studio by typing:

   ```
   estudio19.05 chess_solitaire.ecf &
   ```

4. Once the generated project compiles successfully in Eiffel Studio, go to the `ROOT` class in the `root` cluster. Change the implementation of the `switch` feature as:

   ```
   switch: INTEGER
       -- Running mode of ETF application
     do
   -- Result := etf_gui_show_history  -- GUI mode
     Result := etf_cl_show_history
   -- Result := unit_test     -- Unit Testing mode
     end
   ```

   This overrides the default GUI mode of the generated ETF. To make it take effect, re-compile the project in Eiffel Studio.

5. Switch back to the terminal and type the following command:

```
EIFGENs/chess_solitaire/W_code/chess_solitaire
```

Then you should see this output (rather than launching the default GUI of ETF):

```
Error: a mode is not specified
Run 'EIFGENs/chess_solitaire/W_code/chess_solitaire -help' to see more details
```

6. As you develop your ETF project for the Chess-Solitaire game, launch the batch mode of the executable. For example:

```
EIFGENs/chess_solitaire/W_code/chess_solitaire -b at01.txt
```

This prints the output to the terminal. To redirect the output to a file, type:

```
EIFGENs/chess_solitaire/W_code/chess_solitaire -b at01.txt at01.actual.txt
```

The `at01.actual.txt` file stores the *actual* output from your current software, and your goal is to make sure that `at01.actual.txt` is identical to `at01.expected.txt` by either typing:

```
diff at01.expected.txt at01.actual.txt
```

or typing:

```
meld at01.expected.txt at01.actual.txt
```

Of course, the actual output file produced by the default project is far from being identical to the expected output file.

7. You should first aim to have your software produce outputs that are identical to those of the twelve expected output files (i.e., `at01.expected.txt`, `at02.expected.txt`, ..., `at07.expected.txt`).

8. Then, as you develop further for your ETF project, create as many acceptance test files of your own as possible. Examine the outputs and make sure that they are consistent with the requirements as stated in this document.

9. You are also given an *oracle* program for you to test if your software and the oracle produce identical outputs on all of your acceptance test files. That is, given a test file `at100.txt` of yours, you need to make sure that your output matches that produced by the *oracle*:

```
EIFGENs/chess_solitaire/W_code/chess_solitaire -b at100.txt at100.actual.txt
./oracle -b -b at100.txt at100.expected.txt
meld at100.expected.txt at100.actual.txt
```

11 Reporting Errors of the Oracle

The *oracle* program given to you may not be perfect. If you believe that the *oracle* exhibits an incorrect behaviour on a test file of yours (e.g., `at100.txt`), send your test file and a short description to: `jackie@eecs.yorku.ca`.

12 Modification of the Cluster Structure

You must not change signatures of any of the classes or features that are generated by the ETF tool. You may only add your own clusters or classes to the `model` cluster as you consider necessary. However, when you add a new cluster, it is absolutely critical for you to make sure that a **relative path** (i.e., a path that is relative to the current project directory `.` and does not start with `/`) is specified to add that cluster in the project setting. **Specifying an absolute path in your project will make your submitted project fail to compile when being graded, and this will result in an immediate zero for your marks with no excuses.** So please, make sure you pay extra attention to all clusters that you add to the project.

13 Submission

13.1 Checklist before Submission

1. Make sure the `ROOT` class in the `root` cluster has its `switch` feature defined as:

```
switch: INTEGER
    -- Running mode of ETF application
  do
-- Result := etf_gui_show_history  -- GUI mode
     Result := etf_cl_show_history
-- Result := unit_test     -- Unit Testing mode
  end
```

2. A first BON diagram (`EECS3311_lab3_bon_1.xml`) that shows your design (classes inside the `model` cluster), and how it is related to the command classes (`ETF_START_GAME`, `ETF_MOVE_AND_CHECK`, *etc.*) in the `user_commands` cluster (including the singleton pattern).

   Here you do not need show details of your model classes (i.e., oval shapes with their names, indicating if they are deferred or effective, suffice). However, clearly show the important client-supplier and/or inheritance relations in your design.

3. A second BON diagram (`EECS3311_lab3_bon_2.xml`) that shows details of your model classes only (i.e., contracts, feature declarations, and any inheritance or client-supplier relation between them).

4. You must include the draw.io XML source files of your two BON diagrams and their two exported PDF files in the `docs` directory. **If the TA cannot find, in the `docs` directory, the two draw.io XML sources and PDF files of your BON diagrams, you will immediately lose 50% of your marks for that part of the lab.**

13.2 Submitting Your Work

- There are two `chess_solitaire` directories: one is the top-level directory that contains all generated ETF code and your development; the other is the sub-directory that contains the `model` cluster. **When you submit, make sure you submit the top-level `chess_solitaire` directory.**
- Go to the directory containing the top-level `chess_solitaire` project directory:
- Run the following command to remove the `EIFGENs` directory:

```
eclean chess_solitaire
```

– Run the following command to make your submission:

```
submit 3311 Lab3 chess_solitaire
```

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:

```
feedback 3311 Lab3
```

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.

14 Questions

There might be unclarity, typos, or even errors in this document. It is **your responsibility** to bring them up, early enough, for discussion. Questions related to the lab requirements are expected to be posted on the online course forum. It is also **your responsibility** to frequently check the forum for any clarifications/changes on the lab requirements.