EECS3311 Software Design
Assignment: Peg Solitaire Game

Due: 12 p.m. (noon), Thursday, July 9

Abstract

This documents contains the requirements for you to complete the assignment. There are three expected learning outcomes from this assignment: **1)** document implementations with *complete* contracts; **2)** write unit tests to verify the correctness of your software; and **3)** draw professional architectural diagram using the Visio tool.

Contents

1 Problem

Peg solitaire is a single-player board game where the player attempts to continuously move pegs around, and remove pegs from, the board. The game can be played either on a physical board with holes (LHS of Figure 1) or in a computer app (RHS of Figure 1).

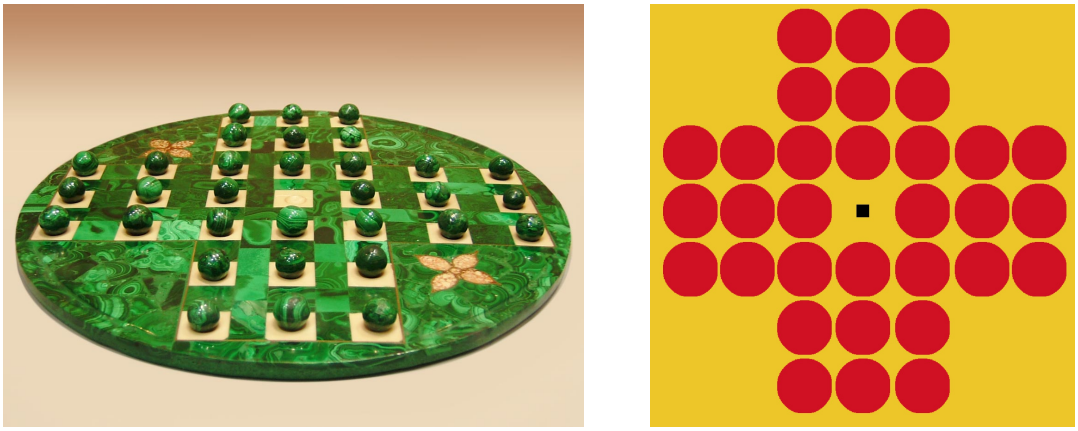


Figure 1: Peg Solitaire: Physical Board vs. Computer App

For the purpose of this assignment, we will use ASCII characters to describe each board that is resulted from a valid move in the game. Figure 2 shows the ASCII representation of the starting board in Figure 1.

row \ column	1	2	3	4	5	6	7
1	*	*	O	O	O	*	*
2	*	*	O	O	O	*	*
3	O	O	O	O	O	O	O
4	O	O	O	.	O	O	O
5	O	O	O	O	O	O	O
6	*	*	O	O	O	*	*
7	*	*	O	O	O	*	*

Figure 2: Peg Solitaire: Represented in ASCII Characters

In Figure 2, we partition each board into seven *rows* and seven *columns*. Each slot on a board is located using its vertical coordinate (i.e., row) and horizontal coordinate (i.e., column). We write *(r, c)* to abbreviate the slot location at row r and column c. To represent slots:

- We use the star character `*` to denote slots (e.g., one at (2, 1)) that are unavailable for placing pegs (i.e., slots with "no holes" on the LHS of Figure 1 and "invisible" slots on the RHS of Figure 1).
- We use the upper case of alphabet `O` to denote slots (e.g., one at (3, 5)) that are available for placing pegs but currently occupied by some pegs.
- We use the dot character `.` to denote slots (e.g., one at (4, 4)) that are available for placing pegs and currently not occupied by pegs.

A peg on the board is considered as *movable to left* if: 1) the slot to its immediate left (on the same row) is occupied by another peg; and 2) the slot that is two columns away from it to the left (on the same row) is available and not occupied by a peg. Symmetric definitions apply to the other three directions:

movable to right, movable to up, and movable to down. A peg on the board is considered as *movable* if it is movable to at least one of the four directions. As examples, in Figure 2, pegs at (4, 6), (4, 2), (2, 4), and (6, 4) are movable to, respectively, left, right, down, and up.

If a peg is currently movable to left, then *moving it left* will result in: **1)** its currently occupied slot becoming unoccupied; **2)** its immediate left slot (which used to be occupied) becoming unoccupied; **3)** the slot that is two columns away from it to the left (which used to be unoccupied) becoming occupied; and **4)** the rest of slots on the board remaining unchanged. Symmetric definitions apply to the other three directions: move right, move up, and move down. Visually speaking, each valid move of a peg (towards a particular direction) "jumps over" its adjacent peg and removes that peg from the board, hence a reduction of the total number of pegs on the board by one.

Each movement of the game involves only one move of a movable peg. When there is not a single peg on the board that is movable, then the game is considered as *over*. When a game is over, if there is only one peg left, then the game is considered as being *won*; otherwise, when a game is over but there are more than one pegs left on the board, it is considered as being *lost*.

As an example, Figure 3 illustrates three *possible* consecutive moves, starting from the board in Figure 2.

board_1		*board_2*		*board_3*		*board_4*
000		**000**		**000**		**000**
000		**000**		**00.**		**..0**
0000000		0000000		0000.00		0000.00
000.000	move_left(4, 6)	0000..0	move_down(2, 5)	00000.0	move_right(2, 3)	00000.0
0000000		0000000		0000000		0000000
000		**000**		**000**		**000**
000		**000**		**000**		**000**

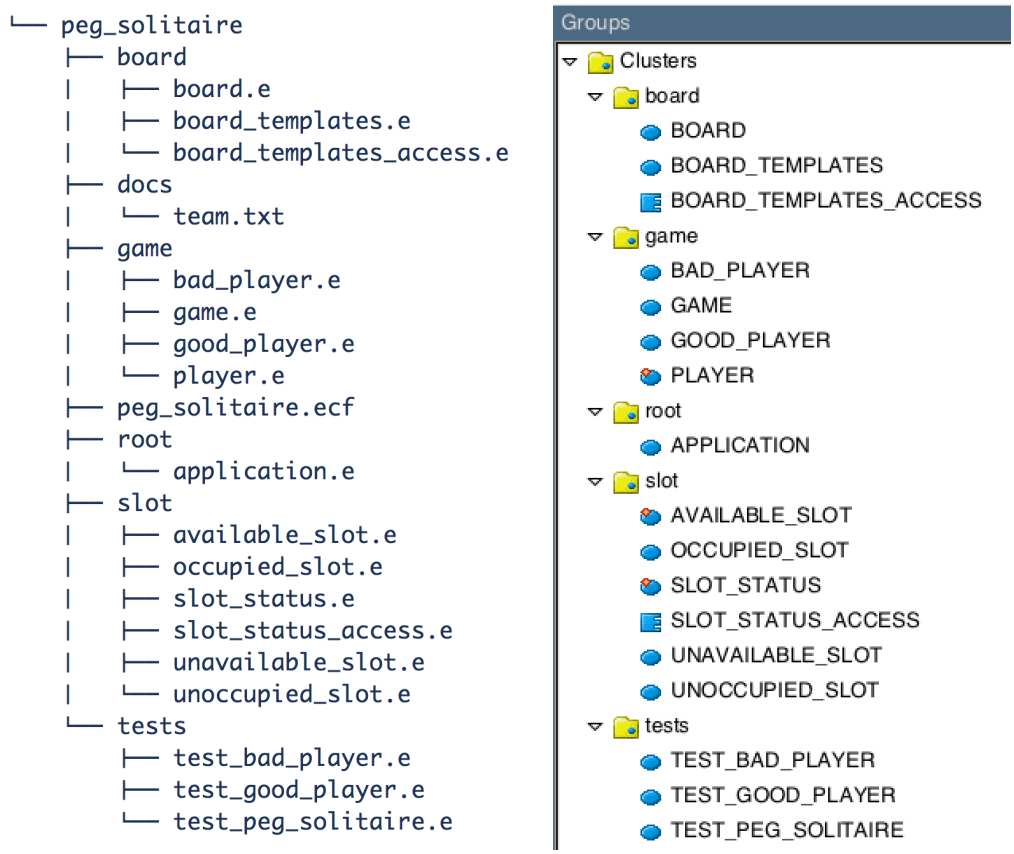
Figure 3: Example Movements of Pegs: Three Consecutive Movements

In Figure 3, when the game starts with *board_1*, there are only four possible moves: 1) move the peg at (2, 4) down; 2) move the peg at (4, 2) right; 3) move the peg at (6, 4) up; or 4) move the peg at (4, 6) left. All other pegs in *board_1* are *unmovable*. For example, the peg at (1, 4) cannot be moved down because the peg at (3, 4) is currently occupied; it cannot be moved right or left because pegs at, respectively, (1, 6) and (1, 2) are unavailable; and it cannot be moved up because it is already on the board boundary. From *board_1*, moving the peg at (4, 6) left results in *board_2*, where slots at (4, 5) and (4, 6) become unoccupied, the slot at (4, 4) becomes occupied, and the rest of the board remains unchanged. Now the peg at (4, 4) on *board_2* cannot be moved to right because its right neighbour at (4, 5) is unoccupied. Similar explanations apply for the other two moves.

It is important to observe that in each state of the game, there might be more than one moves (for more than one pegs) possible, and it is completely up to the player's own strategy of choosing among those possible moves. Therefore, the series of moves illustrated in Figure 3 is not the unique play of the game! See this online animations for one way to solving the game in Figure 2.

2 Getting Started

1. Download the starter code here.
2. Unzip and open `peg_solitaire.ecf` in Eiffel Studio.
3. You will build on this project by adding your own contracts and implementations (Section 3) and submit it for assessment.
4. Figure 4a shows the directory structure (on your file system) and the cluster structure (on Eiffel studio) of the project.
   - Notice that the `docs` directory, where you will place the source file of your Visio drawing and its exported PDF file, is a directory but not added as a cluster. Also, you will need to modify the existing file `team.txt` to include the CSE login ID's of yourself, and of your team partner (if you choose to work as a group of two).
   - Throughout your development for the assignment, you should **not** modify the directory and cluster structures.
5. Figure 4b shows the initial run of the workbench system.
6. Have a look at the *BOARD_TEMPLATES* class (in the *board* cluster), which enumerates the kinds of boards that we consider for the purpose of this assignment.
7. Before proceeding to the next section, you should spend time understanding how classes from different clusters interact with each other.
   - For example, you should study how the given 3 tests manipulate objects and features of the game.



(a) Project of Peg Solitaire: Directories in File System vs. Clusters in Eiffel Studio

APPLICATION

Note: * indicates a violation test case

FAILED (3 failed & 0 passed out of 3)

Case Type	Passed	Total
Violation	0	0
Boolean	0	3
All Cases	0	3

State	Contract Violation	Test Name
Test1	TEST_PEG_SOLITAIRE	
FAILED	Postcondition violated.	test: game creation with easy board
Test2	TEST_GOOD_PLAYER	
FAILED	Postcondition violated.	test: good player wins an easy board
Test3	TEST_BAD_PLAYER	
FAILED	Postcondition violated.	test: bad player loses an easy board

(b) Initial Run of the Workbench System

Figure 4: Starter Code: Structure and Initial Run

3 Tasks to Complete

This section serves to guide your development. In the starter code, where you need to complete the contract or implementation, there is a line indicated as "`-- Your task`". You must complete **all** of them. You must **not** change names of or remove contract tags. You must also **not** modify names or types of features. Moreover, as you complete each contract, remember to delete the default value *True* that is originally placed there.

3.1 The *slot* Cluster

3.1.1 The *SLOT_STATUS* Class

This deferred class represents the common parent class of the three possible statuses of a slot. Your tasks are to complete the implementation and contract of the *is_equal* feature. Two slot statuses are considered as equal if their ASCII representations are identical.

3.1.2 The *AVAILABLE_SLOT* Class

This deferred class represents the common parent class of occupied and unoccupied slots. For this class, there are no tasks for you to complete.

3.1.3 The *UNAVAILABLE_SLOT*, *OCCUPIED_SLOT*, and *UNOCCUPIED_SLOT* Classes

These three effective classes are descendant classes of the *SLOT_STATUS* class. Your tasks are to complete the implementations and contracts of the *out* feature in these classes. The string representation of a slot status is considered as correct if it is identical to the ASCII representation as discussed in Section 1.

3.1.4 The *SLOT_STATUS_ACCESS* Class

This expanded class implements a singleton pattern, making sure that there is only one instance ever created for class *SLOT_STATUS*. Your tasks are to complete (or to fix if appropriate) implementations of the three features *unavailable_slot*, *occupied_slot*, and *unoccupied_slot* and the class invariant. Review relevant lecture notes on the singleton patten if needed.

3.2 The *board* Cluster

3.2.1 The *BOARD_TEMPLATES* Class

- Under the *Templates* feature section:
  - Here declares the 8 kinds of boards that we consider for this assignment.

    You must not modify this section.
- Under the *Constant String Representations of Boards* feature section:
  - Here declares the string representations of the 8 kinds of boards.

    You must not modify this section.
- Under the *Constructor* feature section:
  - The implementation of feature *make* is already completed, and you must not modify it.

    You should observe that the string constants defined here will be compared against values returned by the *out* query in class *BOARD* (which you will implement).

Important Note: Each line of the string constants corresponds to a row on the board. Each line contains characters that correspond to the seven slots. Each of the first six lines is also followed by a new-line character. The last line does **not** contain a new-line character.

- Under the *invariant* section:
  - Your tasks are to complete each tagged invariant:
    * The first category (e.g., *correct_easy_board_output*) makes the string value of a board (*easy_board_out*) visible in the contract view of class *BOARD_TEMPLATES*. That is, when this class is examined in the contract view, its clients must be able to see, in the invariant section, the exact string values of these constants (currently, this information is hidden, as the assignments of these string constants occur in the implementation of the *make* feature).
    * The second category (e.g., *consistent_easy_board_outputs*) asserts that the string representation of a board template (*easy_board*) matches its constant definition (*easy_board_out*) in the *BOARD_TEMPLATES* class.

3.2.2 The *BOARD_TEMPLATES_ACCESS* Class

This class implements a singleton pattern, making sure that there is only one instance ever created for class *BOARD_TEMPLATES*. Your tasks are to complete (or to fix if appropriate) the implementation of the *templates* feature and the class invariant. Review relevant lecture notes on the singleton patten if needed.

3.2.3 The *BOARD* Class

- Under the *Implementation* feature section:
  - You must not modify this section.
  - For this assignment, you are required to use a two-dimensional array (an Eiffel library class) *imp: ARRAY2[SLOT_STATUS]* to implement the board.
  - The two attributes *ssa* and *bta* are for accessing singleton objects of type *SLOT_STATUS* and *BOARD_TEMPLATES*. If you are unsure about how to use them, refer to your lecture notes on the singleton pattern.
- Under the *Constructor* feature section:
  - The implementation of *make_default* is completed, but its postcondition is your task.
  - The *make_easy* feature is already completed for you.
  - The implementations and contracts of all other *make_* features are your tasks.
- Under the *Auxiliary Queries* feature section:
  - The implementations and contracts for features *unavailable_slot*, *occupied_slot*, and *unoccupied_slot* are already completed. You may use them, which return singleton slot status objects, in either implementations or contracts of other features.
  - The implementation and contract of the feature *matches_slots_except* are your tasks.

    This feature identifies a rectangle of slots, outside of which the slots between the *current* and *other* boards match on their statues. Using Figure 3 (page3) as an example, we expect the following to return *True*:

    board_1.matches_slots_except (*board_2*, 4, 4, 4, 6)

That is, we require that slots other than those at (4, 4), (4, 5), and (4, 6) match on their statuses in *board_1* and *board_2*.

This is a very critical feature to understand and get right in order to write complete contracts for other features.

- Under the *Auxiliary Commands* feature section:
  - The implementations and contracts of both features *set_status* and *set_statues* are your tasks.

    The postconditions of these two features contain two parts: the first part ensures that the slot(s) identified by the argument row(s) and column(s) are properly set by the argument status; and the second part ensures that slots other than the one(s) identified remain unchanged.

    For the second part of both postconditions, you will use the Boolean query *matches_slots_except* in the same *BOARD* class.

    For the *set_statues* feature, it is required that $[r1,\ r2]$ and $[c1,\ c2]$ form two valid (closed) integer intervals for, respectively, rows and columns. These constraints should be specified as the preconditions with tags *valid_row_range* and *valid_column_range*, respectively.
- Under the *Queries* feature section:
  - The implementations and contracts of all features in this section are your tasks.

    The implementations and contracts of features *number_of_rows* and *number_of_columns* should be written in terms of the (hidden) attribute *imp*, declared as a two-dimensional array.

    All other queries should **not** refer to the attribute *imp*. Instead, use features *number_of_rows* and *number_of_columns* whenever possible.

    For the feature *number_of_occupied_slots*, you only need to complete its implementation.
- Under the *Equality* feature section:
  - The implementation and contract of feature *is_equal* are your tasks.

    Two boards should be considered as equal if their string representations (as defined by feature *out*) are identical.
- Under the *Output* feature section:
  - The implementation of feature *out* is your task.

    This is a critical feature to get right. Make sure that the return values from this feature are identical to the board string constants defined in class *BOARD_TEMPLATES*.

3.3 The *game* Cluster

3.3.1 The *GAME* Class

- Under the *Board* feature section:
  - You must not modify this section.
  - The *board* attribute stores the board of current game.
  - The *bta* attribute is for accessing the singleton object of type *BOARD_TEMPLATES*. If you are unsure about how to use them, refer to your lecture notes on the singleton pattern.
- Under the *Constructors* feature section:
  - The implementation of *make_from_board* is completed, but its postcondition is your task.
  - The *make_easy* feature is already completed for you.

  - The implementations and contracts of all other *make_* features are your tasks.

- Under the *Commands* feature section:
  - The implementations and contracts of the four movement commands are your tasks.
  - In the preconditions of all these commands, we use *from_slot* to denote the slot whose occupying peg is being moved, *middle_slot* the slot whose occupying peg is being "jumped over", and *to_slot* the slot to hold the "jumping" peg.

    The postconditions of these commands contain two parts: the first part ensures that **all** three slots involved in the movements are properly updated on their statuses; and the second part ensures that slots other than these three remain unchanged.

    For the second part of these postconditions, you will use the Boolean query *matches_slots_except* in the *BOARD* class.

- Under the *Status Queries* feature section:
  - The implementations of both queries *is_over* and *is_won* are your tasks.
  - You may find useful some auxiliary queries in the *BOARD* class.

- Under the *Output* and *Auxiliary Routines* feature sections:
  - Do not modify any of these sections. They are already implemented for you. However, you should understand how the *out* feature of *BOARD* is used in the *out* feature of *GAME*.

3.3.2 The *PLAYER* Class

- Under the *Attributes* feature section:
  - You must not modify this section.

- Under the *Constructor for descendant classes* feature section:
  - The postcondition of the *make* feature is your task.

3.3.3 The *GOOD_PLAYER* and *BAD_PLAYER* Classes

- Under the *Commands* feature section:
  - Implementations of features *wins_easy_board* and *loses_easy_board* are completed for you as examples of, respectively, winning and losing a game.
  - Contracts of features *wins_easy_board* and *loses_easy_board* are your tasks.
  - Implementations and contracts of all other commands, to either win or lose the corresponding game, are your tasks.

3.4 The *tests* Cluster

This cluster contains three test classes, each of which already containing one test case for you.

- You are advised to add more tests to the *TEST_GOOD_PLAYER* and *TEST_BAD_PLAYER* classes to verify your software, although you will not be graded on these two test classes.

- You are **required** to add another **10** tests[1] to the *TEST_PEG_SOLITAIRE* class to verify your game software. Your 10 tests must **not** repeat and each one of them should test a different aspect of the game.

[1] Do not modify or delete the existing test case *test_easy_game_creation* that is given to you. So in total you should have at least 11 tests in this class.

3.5 The *root* Cluster

This cluster contains only the *APPLICATION* class which adds test cases. You should not need to modify this class.

3.6 BON Architecture Diagram

- Use Visio to draw a BON Architecture Diagram that details the relationships among classes in clusters *board*, *game*, *slot*, and *tests*.
- When completing your drawing, also export the diagram into a PDF (see the tutorial video on how to do this).
- Move the source of your Visio drawing (name it `solitaire.vsdm`) and its exported PDF (name it `solitaire.pdf`) to the `docs` directory of your project (see Figure 4a, page5).

4 Preparing Your Submission

4.1 Working as a Group

You can either work on your own, or work with another person to make it a group of two. A group with more than two members is **not** allowed.

4.2 Checklist before Submission

1. Complete all contracts and implementations indicated with "`-- Your task`" in the starter code.
   - Do **not** add or delete any classes or clusters. Do **not** modify signatures (names and types) of existing features. You **may** add auxiliary features to existing classes if necessary.
   - You should exercise the test-driven development method that was taught.
   - Write and pass as many tests as possible on your software, as only passing the 3 given tests will **not** ensure good quality of your development.
2. Add 10 additional tests to the *TEST_PEG_SOLITAIRE* class.
   - Do **not** modify the existing test case in that class.
3. Use Visio to draw a BON Architecture Diagram that details the relationships among classes in clusters *board*, *game*, *slot*, and *tests*.
   - When completing your drawing, also export the diagram into a PDF (see the tutorial video on how to do this).
   - Move the source of your Visio drawing (name it `solitaire.vsdm`) and its exported PDF (name it `solitaire.pdf`) to the `docs` directory of your project (see Figure 4a, page5).
4. Modify the `team.txt` file in the `docs` directory by including the CSE login names of yourself, and of your team parter (if you choose to work as a group of two). Here is an example of the contents of `team.txt` (with two members in the team):

   ```
   cse123456
   cse654321
   ```

5. Compile and print off a report including:
   - A cover page that clearly indicates: 1) course; 2) semester; 3) names; 4) CSE logins of the team member(s); and 5) CSE login of the submitting account;

- The **contract view** of the *GAME* and *BOARD* classes; and
- BON diagram for your design.

4.3 Submitting Your Work

Both hard-copy and electronic submissions are required.

- **Hard-Copy Submission**

  1. By the due date, drop the print-out of the report into the EECS3311 dropbox.

- **Electronic Submission**

  1. You are expected to submit from a Prism lab terminal.
  2. Each team must make their submission from only a single CSE account.
  3. Go to the directory containing the `peg_solitaire` project directory:

     3.1 Run the following command to remove the `EIFGENs` directory:

     ```
     eclean peg_solitaire
     ```

     3.2 Run the following command to make your submission:

     ```
     submit 3311 Assignment peg_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 Assignment
     ```

     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.