# EECS331 Lab4

Design of Tic-Tac-Toe game with undo/redo design pattern

L4

## Table of Contents

# 1. Teams

In this Lab you may work on your own or in teams of at most two members.

If you work in a team, you are still individually responsible for submitting the Lab by the due date. Your team should thus setup a private *github* repository where both members of the team have access to all the design and coding material. If the team dissolves for any reason whatsoever, each individual member of the team is responsible for completing and submitting the project on their own.

The team submits electronically only under the Prism ID of one of the team members. Please ensure that there are no duplicate submissions which may result in no grade at all. Likewise, only one of the team members submits their Design document on *moodle*.

# 2. Problem statement

The main purpose of this Lab is to design and implement an undo/redo mechanism using *polymorphism*, *static typing* and *dynamic binding*. A general undo/redo design pattern is described in OOSC2, chapter 21, and students must use this pattern.

Students must design and construct a game, supporting undo/redo, that the user can play from the console. To keep it simple, the game will nominally be Tic Tac Toe.

In constructing the undo/redo mechanism we seek a design that is re-usable, i.e. can be used in other games and other applications regardless of the application domain. Your design of the undo/redo mechanism should be constructed to satisfy the following design goals:

- G1: The mechanism should be applicable to a wide class of interactive applications, regardless of the application domain.
- G2: The mechanism should not require redesign for each new input command.
- G3 (efficiency): It should make reasonable use of storage.
- G3: It should be applicable to both one-level and arbitrary-level Undos.

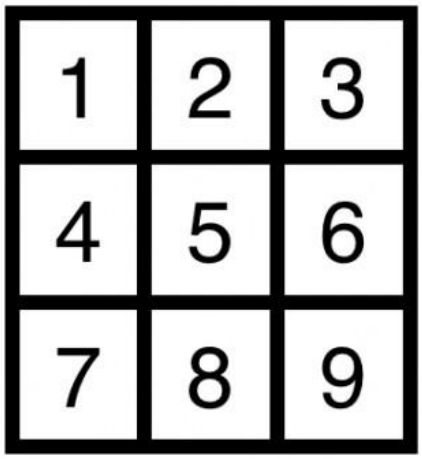
# 3. Tic Tac Toe

From *Wikipedia*: Tic-tac-toe (or noughts and crosses, X's and O's) is a paper-and-pencil game for two players, X and O, who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three respective marks in a horizontal, vertical, or diagonal row wins the game.

Our customer will support the game as a distributed webapp, a mobile app, and a desktop app. The precise user interfaces, however, have yet to be defined. We would like a design that provides the correct business logic (model) irrespective of the eventual user interface. By distributed, we mean that two players may be at two different geographic locations linked via the internet. In the first instance, you are required to design the game to work from the console.

## 4. User Interface

The user interfaces will be constructed from objects such as buttons, windows and form objects. At the user interface level, the buttons are numbered 1 ... 9:




The abstract input commands from the user interface can be summarized by an ETF input grammar as follows:

```
system tictac

type NAME = STRING
type BUTTON = 1..9

new_game (player1: NAME; player2: NAME)
-- add players `player1' and `player2'
-- `player1' starts X

play (player: NAME; press: BUTTON)
```

```
play_again

undo
-- last action while in play
--otherwise no effect

redo
-- last action while in play
-- otherwise no effect
```

# 5. Analysis and Design (Undo/Redo design pattern)

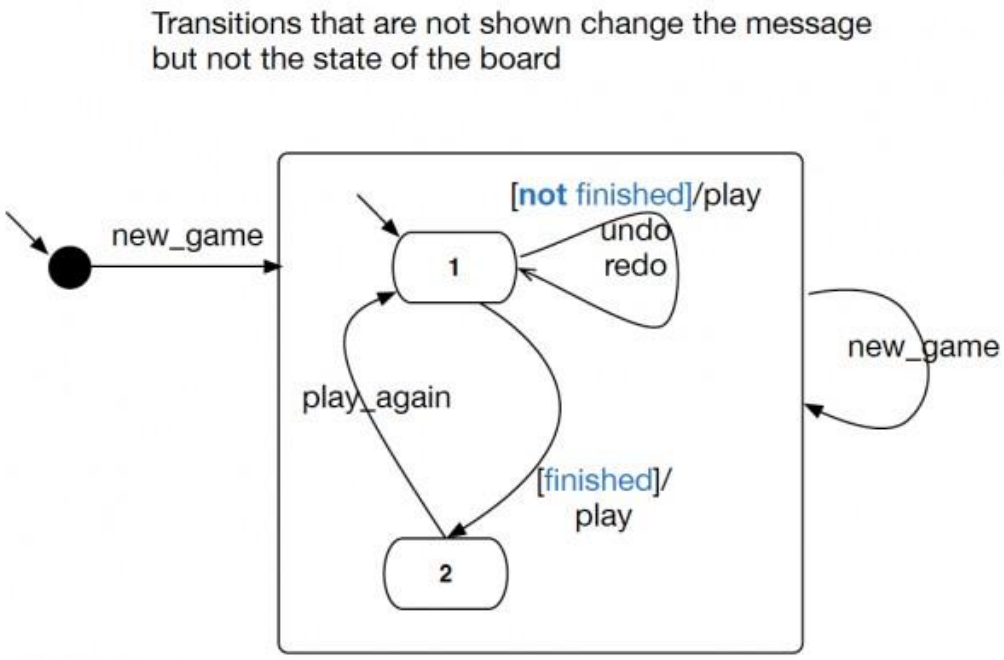
You are required to read OOSC2, chapter 21, and to implement the undo/redo pattern described in that chapter.

In this course we use the following diagrams: BON and UML class diagrams, UML sequence diagrams, and UML statecharts. For the UML diagrams study the following slides: https://wiki.eecs.yorku.ca/project/eiffel/_media/bon:uml.pdf
(Prism logon required).[1]

The requirements for the undo/redo mechanism are that they are only in effect during a game. The UML statechart in the figure below describes the modes of the design, needed to effect this requirement.[2]

[1] For BON class diagrams see http://seldoc.eecs.yorku.ca/doku.php/eiffel/bon/start

[2] It is required that you familiarize yourself with the notion of a UML statechart. The original paper by David Harel is on the course SVN.




From the statechart, there are at least two types of operations that might be undone. (1) A play operation (2) Other operations that deliver a message without changing the state (e.g. if the user attempts play_again during a game, or invokes a play with arguments that cannot be satisfied).

From the point of view of extensibility we require that your undo design pattern also supports the possibility of adding additional operations for undo/redo. For example, new_game cannot currently be undone, but in future we may wish to do so.

In OOSC2 (ch. 21) the command pattern is used. Since ETF_COMMAND is already an abstract class in ETF, used for handling user commands from the console, you might consider using a new abstract class OPERATION for the undo/redo abstraction so as not to confuse it with the ETF class.

## 6. Requirements and acceptance tests

In the course directory on Prism, there is a user interface grammar, an acceptance test, and an oracle *tictac.exe* (constructed via ETF) as follows:

```
red% cd /cs/course/3311/labs/lab4/
red% ls
at1.expected.txt
at1.txt
```

```
oracle.exe
tictac.definitions.txt
tictac-status-messages.txt
```

The oracle is oracle.exe and the ETF grammar is provided in *tictac.definitions.txt*. Running the oracle on one of the acceptance tests (at1.txt) we obtain something like:

```
ok:  => start new game
  ___
  ___
  ___
  0: score for "" (as X)
  0: score for "" (as O)
->new_game("Xavier","Ora")
  ok: => Xavier plays next
  ___
  ___
  ___
  0: score for "Xavier" (as X)
  0: score for "Ora" (as O)
->play("Xavier",1)
  ok: => Ora plays next
  X__
  ___
  ___
  0: score for "Xavier" (as X)
  0: score for "Ora" (as O)
…
```

You can also run the application in interactive mode via: `./oracle.exe -i`.

Our customer has told us that the above is correct output for the given inputs. The oracle thus acts as your "always available customer", and you consult it for all the requirements.

# 7. Design and Constructions

- Use ETF to construct an application that does precisely what the oracle does (character –for-character).
- Design a suitable undo/redo mechanism that satisfies the design goals G1 - G4 (using OOSC2, chapter 21).
- Design a suitable architecture for the business logic. Specify the game logic using expressive contracts.

- Write your own acceptance tests (in addition to the ones provided) to check all aspects of the game for conformance with the oracle.

## 8. Design Decisions

You will be making a number of design decisions for which you must be prepared to justify the choice

- The design of the board and player moves
- When a game has been won (i.e. when to terminate)
- Reporting of scores
- The undo/redo design pattern
- Error/status reporting.
- Error status reporting is discussed further below.

In all cases, provide the rationale based on the principles of modularity, separation of concerns, abstraction and information hiding.

## 9. Design Document

You must submit a design document. For more on how to write design documentation, please study the material at
https://wiki.eecs.yorku.ca/project/sel-students/p:tutorials:sdd:start
(Prism logon required). This site is directed at the documentation you will produce for your project.

In this Lab, we require a professional report with the following components:

- Title page with your name, signature and Prism login, and table of contents (first page).

- Starting on the next page, a top level view of your design for the Tic Tac Toe game. This includes a BON class diagram for the game including the command and model classes and descriptive text (2 pages).

- A Table of the significant modules (usually classes, but could also be a cluster) of your design. This is where you provide a brief description of each module in your system, its “secret” (based on information hiding) and the design decisions used in the production of the module (2 pages at most). For more detail see the footnote.

- Description of how you detect a winning game. This will include text, BON diagrams and contracts (as appropriated). (2 pages)

- A description of your undo/redo design for this game. (2 pages)

There are 9 pages in total at most (including the title page).

Please ensure that the document is prepared to professional standards using MS Word and the **draw.io** template for the BON diagrams, appropriate mathematical symbols and 12 point text. To use draw.io, you will need a template documented below:

| |
|---|
| http://seldoc.eecs.yorku.ca/doku.php/eiffel/faq/bon |

Import your diagrams into Word as PDFs so that the diagrams are clear (not fuzzy) and readable.

BON diagrams shall not contain all and every possible piece of information which will only confuse the reader. Only those parts needed at the appropriate level of abstraction for describing the design precisely and clearly, shall be in the class diagram. Thus, use *detailed* and *compressed* views appropriate to the design information being conveyed. The slides presented in class have many BON class diagrams and you should carefully use these as exemplars

# 10. Submission

```
tictac-lab/
├── docs
│   ├── team.txt
│   └── tictac.pdf
├── root
│   └── root.e
├── tests
│   └── acceptance
│       ├── instructor
│       │   └── at1.txt
│       └── student
│           ├── at1.txt
│           ├── at2.txt
│           ├── at3.txt
│           └── at4.txt
└── tictac.ecf
```

You must submit a **superset** of the above directory structure i.e. you may add additional folders (clusters) and files (classes) so that we can compile your project. Additional folders must use relative addressing. If you provide an absolute address to code in your home directory, we will

not be able to compile your project as we do not have access to your home directories. You will thus be submitting additional directories such as *tests, generated_code*, etc. You need to submit all the code needed so that we can compile and execute your project.

In the *docs* folder you must submit

- *tictac.pdf*: this is the report of your team, professionally prepared
- *team.txt*: is the Prism logins of the members of your team

The *team.txt* file looks like this:

```
cse99783
cse67999
```

The Prism login on the first line of the *team.txt* file shall also the login for the team submission. If *team.txt* is not precisely as specified, you will not receive a grade. You may work on your own (in which case there is only one login) or in a team of at most two members.

You shall write at least **four** acceptance tests of your own, which appear in the *student* directory. (The test in the *instructor* directory is the one that is given to you by the instructor). It is advisable that you also write unit tests to test your code, but no lower limit is supplied.

```
submit 3311 Lab3 tictac-lab
```

You shall ensure that there are no EIFGENs directories in your submission, i.e. you must eclean before you submit.

Moodle submission: You must also submit your report **tictac.pdf** to Moodle.

# 11. Appendix: Regression Testing (Haskell scripts)

You are required to ensure that the output of your program is character-for-character the same as the oracle, even with regards to white space. [Aside: e.g., if you write a script to email data to a set of contacts, and you put an incorrect space in the email address, your script will fail; thus, being off by a character is no excuse.]. You can do a "diff' on your actual output vs. the oracle output. But eventually, this gets tiresome.

Whenever developers change or modify their software, even a small change can introduce unintended errors elsewhere in the software. In Regression testing, we re-run all the older tests to ensure that they still pass and no new errors have been introduced. Such tests can be performed manually on small projects, but in most cases repeating a suite of tests each time an update is made is too time-consuming and complicated to consider, so an automated testing tool is typically required.

For Acceptance Testing we suggest that you use the Haskell scripts that we provided and documented in earlier Labs.

```
%./etf_test.sh
Building executable from ETF_Test.
...
Running ./ETF_Test
...
=========================
Test Results: 0/2 passed.
=========================
Error: ./log/at1.expected.txt and ./log/at1.actual.txt do not match.
Error: ./log/at2.expected.txt and ./log/at2.actual.txt do not match.
=========================
Test Results: 0/2 passed.
=========================
```

You can examine ETF_Test_Parameters.hs to see how to set up (1) the oracle (2) a W_Code executable and (3) the acceptance tests to be run. You will have to refactor this to your project. In the above, your W-code executable produces the actual output, which is compared to the expected output of the oracle.

As you can see, the acceptance tests at1.txt and at2.txt do not yet work. We now proceed step by step:

- Get at1.txt to pass
- Get at2.txt to pass, while still running at1.txt
- Add new acceptance tests and repeat, ..Etc. ..
- With each change to your code, re-run the script: ./etf_test.sh