Project

EECS-3311 – Project – Battleship

2/28/19 6:43:02 PM

This Project is done on your own or with at most one partner (from either section of EECS3311). You may re-use the code of either partner from Lab4, but you may not cooperate with any other team or outsider. The Project must be solely you own work. Academic integrity rules apply.

The team only submits once under either team member's Prism account (details of submission will be supplied). 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.

Please ensure that there are no duplicate submissions which may result in no grade at all.

1 Design Goals

```
require
     across 0 |..| 4 as i all lab_completed(i) end
     read accompanying document: Eiffel101[1]
     attendance at lectures
ensure
     submitted on time
     no submission errors
     lab submission is reliable (correct and robust) for any input
     correct means it matches the output of the oracle character for character
rescue
     ask for help during scheduled labs
     attend office hours with TA William
```

- Lab1: Sorted Trees (use of `SEQ[G]` from Mathmodels)
- Lab2: Sorted Map (use of `FUN[K, V]` from Mathmodels)
- Lab3: Sorted Variants, design of efficient sorted maps e.g. with red-black trees
- Lab4/**ETF**: ETF Battleship Game **Design** and implementation
- Project/**ETF**: project reusing Battleship from Lab4

The goal of this project is to design and implement an undo/redo mechanism using *polymorphism*, *static typing* and *dynamic binding* for the battleship game (already treated in Lab4).

- A general undo/redo design pattern is described in OOSC2, chapter 21, and students must use this design pattern.
- We also provide you with sample code for the undo/redo design pattern in ETF. See the SVN (with anonymous login): http://svn.eecs.yorku.ca/repos/sel-open/misc/tutorial/undoRedo-DesignPattern/index.html. You can explore the documented code, checkout the project and see a BON class diagram.

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:

1. The mechanism should be applicable to a wide class of interactive applications, regardless of the application domain.
2. The mechanism should not require redesign for each new input command.
3. It should make reasonable use of storage.
4. It should be applicable to arbitrary-levels of undo/redo.

[1] See: https://www.eecs.yorku.ca/~eiffel/eiffel101/Eiffel101.pdf

2 Specification

You obtain much of the detailed specification of the Project via a retrieve (as in the Labs). You are provided with the following:

```
└── battleship
    ├── Battleship-Project-Spec.pdf
    ├── battleship.errors.txt        -- status/error messages
    ├── battleship.ui.txt            -- abstract grammar specification
    ├── initial-battleship/          -- generation of initial ship position
    ├── oracle.exe
    ├── regression-testing/
    └── tests/                       -- 4 acceptance tests are provided
                                     -- you must write many tests of your own
                                     -- we will be grading you with 129+ of our tests
```

Requirements elicitation has resulted in an abstract grammar for the user interface. Eventually, there will be a desktop application for users (the operators of the nuclear plant). However, the concrete GUIs will be developed at a later stage and are beyond the scope of this project. The concrete GUIs will support all the operations in the abstract grammar. The grammar specfies additional operations not present in Lab4 including:

- undo
- redo
- custom_setup
- custom_setup_test
- give_up

Also, you are provided with 4 acceptance tests, two from Lab4 and two new ones involving the undo/redo operations.

Instead of a complete specification, you are also provided with an Oracle. You will thus need to develop your own additional acceptance tests (in the student folder). It is interesting to note that these acceptance tests are enoded in an ASCII format (understandable to our customers) but independent of the programming language used to develop the system. These acceptance tests can thus be written by users who are not familiar with programming. Such acceptance tests can thus be developed before the system is ever implemented. A good set of acceptance tests is thus part of the Battleship specification.

You must develop a program that can be tested from the console. For example, executing

```
battleship -b at001.txt > at001.expected.txt
```

shall produce an output text file such as *at001.expected.txt*, with the proper agreed upon error messages. The "-b" switch stands for batch mode (there is also a "-i" switch for interactive mode). Both relative and absolute paths using Linux conventions shall be supported.

Your program must match the Oracle output character-for-character.

In all cases the behavior of the Oracle is definitive.

3 Report documenting your design

In addition to submitting your code, you will also be required to document your design.

Your report must follow the 3311 SDD structure template documented at https://wiki.eecs.yorku.ca/project/sel-students/p:tutorials:sdd:start

Please read all the details at the above URL carefully.

If necessary, more detail will be supplied in the course wiki. Ensure that you reda the forum and check with the wiki regularly.

The design document will count for a significant part of your Project grade. Note that when we test your design for correctness that also signals to us that if your design is *feasible*. A penalty will be applied to your design document if your design fails our acceptance testing. We thus require that your design be (a) feasible and correct (which we check via acceptance testing) and (b) that it follows good design principles such as meaningful contracts, information hiding, modularity etc.

4 Design

A major goal of a clean design is to facilitate subsequent bug fixes and changes to the requirements. This is where information hiding, modularity, abstraction and a separation of concerns become helpful. The design of the main business logic for the game must demonstrate good modularity, clean specified APIs, suitable information hiding, separation of concerns and helpful abstractions.

In our solution, we have many more than 10 classes in the business logic. If you have not designed modular systems before (with modules being classes or clusters of classes), this is your opportunity to achieve the skills of a designer. You may have more or fewer classes, but they must demonstrate elements of good design. Classes must be **designed** so that it pertains to a single well-defined abstraction (see OOSC2 chapters 22 and 23):

- A class should be known by its interface, which specifies the services offered independently of their implementation.
- Class designers should strive for simple, coherent interfaces.
- One of the key issues in designing modules is which features should be exported, and which should remain secret.
- When considering the addition of a new exported feature to a class, the feature must be relevant to the data abstraction represented by the class; it must be compatible with the other features of the class; it must not address the same goal as another feature of the class; and must maintain the invariant of the class.
- The design of reusable modules is not necessarily right the first time, but the interface should stabilize after some use. If not, there is a flaw in the way the interface was designed.
- Proper use of assertions (preconditions, postconditions, invariants) is essential for documenting interface **specifications**.
- **Documentation principle**: Try to write the software so that it includes all the elements needed for its documentation, recognizable by the tools that are available to extract documentation elements automatically at various levels of abstraction (e.g. contract view vs. text view vs. BON/UML class diagram).
- Follow good style (see OOSC chapter 26).
- Do regular rigorous regression testing as you develop the design and implementation using unit tests and acceptance tests.

So you may not get the design just right in Lab4, but in the Project you will be able to revisit and improve your design.

4.1 Eiffel Testing Framework for Design

As in Lab4, you use the ETF framework to separate out the user interface (view/controller) from the business logic (the model). To read more about the Eiffel Testing Framework (ETF), see:

- http://seldoc.eecs.yorku.ca/doku.php/eiffel/etf/start
- Videos: https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#eiffel_testing_framework
- Conference paper: https://www.eecs.yorku.ca/~jonathan/publications/2018/MoDRE18.pdf

The course wiki will have precise details of information we provide you with as well as how to submit your Project.

5 Appendix (from Lab4)

5.1 Battleship as specified in Lab4

The normal game of Battleship is a 2-player game where each player is given an 8×8 board. Each player is given a set of ships to place on their own board, which is hidden from the other player. Then each player takes turns guessing coordinates to fire upon in an attempt to sink the other players ships. After firing upon a coordinate, the other player announces whether the shot was a hit or miss and whether a specific ship has been sunk. The game ends when one player sinks all of the other players ships.

In this Lab version, a Player plays against the computer. The computer randomly places the ships and it is the goal of the Player to sink the ships. The Player is also given a set of shots and bombs. The game ends when either all the ships are sunk (Player wins) or all the shots and bombs have run out (Player loses). This version also allows for board sizes anywhere from 4×4 to 12×12. A 4 x 4 board has the following coordinates:

[A,1]	[A,2]	[A,3]	[A,4]
[B,1]	[B,2]	[B,3]	[B,4]
[C,1]	[C,2]	[C,3]	[C,4]
[D,1]	[D,2]	[D,3]	[D,4]

Rows are A..L and columns are 1..12.

In **Error! Reference source not found.** (in the Appendix) we are provided with an example of an **acceptance test** (a text file with a sequence of Player actions) as follows:

```
-- Battleship acceptance test: uc1.txt
new_game(easy)
fire([A,4])
fire([B,4])
bomb([C,4],[D,4])
bomb([B,3],[B,2])
fire([C,3])
fire([A,2])
fire([A,3])
new_game(medium)
…
```

The above acceptance test is in a text file `uc1.txt` (use case 1). **Error! Reference source not found.** also provides the expected output (`uc1.expected.txt`). Here is part of the expected output as seen at the command line (EECS Linux Centos7):

```
red% ./oracle.exe -b uc1.txt
  state 0 OK -> Start a new game
->new_game(easy)
  state 1 OK -> Fire Away!
     1  2  3  4
  A  _  _  _  _
  B  _  _  _  _
  C  _  _  _  _
  D  _  _  _  _
  Current Game: 1
  Shots: 0/8
  Bombs: 0/2
  Score: 0/3 (Total: 0/3)
  Ships: 0/2
    2x1: Not Sunk
    1x1: Not Sunk
-> ...
...
->bomb([C, 4],[D, 4])
  state 4 OK -> 1x1 ship sunk! Keep Firing!
     1  2  3  4
  A  _  _  _  O
  B  _  _  _  O
  C  _  _  _  X
  D  _  _  _  O
  Current Game: 1
  Shots: 2/8
  Bombs: 1/2
  Score: 1/3 (Total: 1/3)
  Ships: 1/2
    2x1: Not Sunk
    1x1: Sunk
->...
  ...
  ...

->fire([A, 3])
  state 8 OK -> 2x1 ship sunk! You Win!
     1  2  3  4
  A  _  X  X  O
  B  _  O  O  O
  C  _  _  O  X
  D  _  _  _  O
  Current Game: 1
  Shots: 5/8
  Bombs: 2/2
  Score: 3/3 (Total: 3/3)
  Ships: 2/2
    2x1: Sunk
    1x1: Sunk
```

```
->new_game(medium)
  state 9 OK -> Fire Away!
     1  2  3  4  5  6
  A  _  _  _  _  _  _
  B  _  _  _  _  _  _
  C  _  _  _  _  _  _
  D  _  _  _  _  _  _
  E  _  _  _  _  _  _
  F  _  _  _  _  _  _
  Current Game: 2
  Shots: 0/16
  Bombs: 0/3
  Score: 0/6 (Total: 3/9)
  Ships: 0/3
    3x1: Not Sunk
    2x1: Not Sunk
    1x1: Not Sunk
   >  ...
   >
```

Every time a new game is requested in a session, the board is populated randomly with a variety of ships.

In a requirements document, in addition to acceptance tests, we should describe rules of the game, how ships are randomly generated and placed (row location, column location, vertically or horizontally), how scores are calculated incrementally as new games are invoked by the Player.

Instead we provide you with an executable **Oracle** (*oracle.exe),* which acts in place of a complete specification of the game. Thus, you use the Oracle to answer all questions of required behavior that is not described in this document:

Major Requirement: Your design/submission must match the Oracle output character for character for any input specified by the user interface grammar. The output of your submission must also match -- character for character -- any status/error messages generated by the Oracle. In addition to being **correct**, the output of your submission must be **robust**, i.e. your submission must not ever crash with an exception, and it should never enter a non-terminating loop. Failure in any respect cannot be guaranteed a passing grade. A reliable software product must be correct and robust.

With the Oracle you can construct your own acceptance tests. When we test the correctness of your design we have over 40 acceptance tests (some long some short) to check whether you have developed a reliable product. See Section 5.8 for a description of the role of acceptance tests in developing reliable software.

Requirement
Develop your own suite of acceptance tests using the Oracle, and do rigorous **regression testing** against the Oracle to check the correctness of your design and development.

In the normal game, the Player does not see the coordinates of the ships. We also require a **debug mode** to help the game designers to construct a reliable tested software product.

```
red% ./oracle.exe -i
  state 0 OK -> Start a new game
->debug_test(easy)
  state 1 OK -> Fire Away!
     1  2  3  4
  A  _  _  _  _
  B  _  _  _  v
  C  _  _  h  v
  D  _  _  _  _
  Current Game (debug): 1
  Shots: 0/8
  Bombs: 0/2
  Score: 0/3 (Total: 0/3)
  Ships: 0/2
    2x1: [B, 4]->v;[C, 4]->v
    1x1: [C, 3]->h
->fire([A,4])
  state 2 OK -> Miss! Keep Firing!
     1  2  3  4
  A  _  _  _  O
  B  _  _  _  v
  C  _  _  h  v
  D  _  _  _  _
  Current Game (debug): 1
  Shots: 1/8
  Bombs: 0/2
  Score: 0/3 (Total: 0/3)
  Ships: 0/2
    2x1: [B, 4]->v;[C, 4]->v
    1x1: [C, 3]->h
->bomb([C,4],[D,4])
  state 3 OK -> Hit! Keep Firing!
     1  2  3  4
  A  _  _  _  O
  B  _  _  _  v
  C  _  _  h  X
  D  _  _  _  O
  Current Game (debug): 1
  Shots: 1/8
  Bombs: 1/2
  Score: 0/3 (Total: 0/3)
  Ships: 0/2
    2x1: [B, 4]->v;[C, 4]->X
    1x1: [C, 3]->h
->
```

```
-- user acceptance test 1
debug_test(easy)
fire([A,4])
fire([B,4])
bomb([C,4],[D,4])
...
```

Figure 1 Start of an Interactive Session with the Oracle – Expected Behaviour

Figure 1 provides a view of an interactive session with the Oracle in debug mode. In normal mode, ships are placed in random order on the board that depend on a seed involving the current time. In the debug mode, ships are placed “randomly” on the board but the order is always the same using the same starting seed. This allows us to test the behavior of the system deterministically.

```
system battleship

type COLUMN = 1..12
     -- x-coordinate of the board
type ROW = {A, B, C, D, E, F, G, H, I, J, K, L}
     -- y-coordinate of the board
type COORDINATE = TUPLE[row: ROW; column: COLUMN]

type LEVEL = {easy, medium, hard, advanced}
     -- easy is 4x4 and gets 8 shots and 2 bombs for 2 ships
     -- medium is 6x6 and gets 16 shots and 3 bombs for 3 ships
     -- hard is 8x8 and gets 24 shots and 5 bombs for 5 ships
     -- advanced is 12x12 gets 40 shots and 7 bombs for 7 ships

-- Player Actions
new_game (level: LEVEL)
     -- game based on random placement of ships
     -- winner sinks all ships
     -- loser when we run out of shots and bombs

debug_test (level: LEVEL)
     -- for deterministic testing, see documentation

fire (coordinate: COORDINATE)

bomb (coordinate1: COORDINATE; coordinate2: COORDINATE)
     -- coordinates must be adjacent
     -- bombs do not count as shots
```

Figure 2 ETF User Interface Grammar: Specification of Legal Player Input

5.2 Syntax errors in acceptance tests

We will test your submission only with acceptance tests that are syntactically correct, i.e. satisfy the user interface grammar of Figure 2. For example, consider the following debug interactive session in which we invoke a syntactically flawed (**`fire([[B,4])`**) acceptance test:

```
red% ./oracle.exe -i
  state 0 OK -> Start a new game
->debug_test(easy)
  state 1 OK -> Fire Away!
     1  2  3  4
  A  _  _  _  _
  B  _  _  _  v
  C  _  _  h  v
  D  _  _  _  _
  Current Game (debug): 1
  Shots: 0/8
  Bombs: 0/2
  Score: 0/3 (Total: 0/3)
  Ships: 0/2
    2x1: [B, 4]->v;[C, 4]->v
    1x1: [C, 3]->h
->fire([C,3])
  state 2 OK -> 1x1 ship sunk! Keep Firing!
     1  2  3  4
  A  _  _  _  _
  B  _  _  _  v
  C  _  _  X  v
  D  _  _  _  _
  Current Game (debug): 1
  Shots: 1/8
  Bombs: 0/2
  Score: 1/3 (Total: 1/3)
  Ships: 1/2
    2x1: [B, 4]->v;[C, 4]->v
    1x1: [C, 3]->X
->fire([[B,4])
Syntax Error: specification of command executions cannot be parsed
parse error (Line 1, Column 7)
->
```

The ETF framework will generate a Syntax error. Thus this is not an acceptable acceptance test.

5.3 Semantic errors in acceptance tests

Your submission must be robust, i.e. it must be able to operate in the presence of erroneous input from players. We cannot control the user (the Player) and thus we must signal when the Player provides a problematic move. For example, the software must report an invalid coordinate:

```
->debug_test(easy)
  state 1 OK -> Fire Away!
     1  2  3  4
  A  _  _  _  _
  B  _  _  _  v
  C  _  _  h  v
  D  _  _  _  _
  Current Game (debug): 1
  Shots: 0/8
  Bombs: 0/2
  Score: 0/3 (Total: 0/3)
  Ships: 0/2
    2x1: [B, 4]->v;[C, 4]->v
    1x1: [C, 3]->h
->fire([A,5])
  state 2 Invalid coordinate -> Fire Away!
```

5.4 Modes of play

As mentioned earlier, in a game, there are two modes of play, normal and debug. In normal mode, the user sees a blank board. The ships are placed on the board randomly. After firing upon a coordinate, the coordinate is marked 'X' if it is a hit and 'O' if it is a miss. There is no other information inside the board of where the ships are. Below the board is a list of ships and whether they have been sunk or not.

In debug mode, the ships are displayed to the user with each coordinated either marked by a 'h' or by a 'v'. If it is 'h', the ship is horizontal, otherwise it is vertical. Like in random, an 'X' or 'O' marks a hit or miss respectively. Furthermore, instead of listing the sunk or not sunk status of the ship, all the coordinates of each ship are listed along with the what symbol appears on the board at that coordinate. The other important distinction of debug mode is the ships are placed deterministically, meaning that if given the same sequence of games, the board layouts should not change between those sequences.

When switching between modes, game counter and the scores shall be reset.

5.5 initial-battleship: Ship Placement Code

We provide you with an Eiffel project called `initial-battleship` which includes a class `RANDOM_GENERATOR`. It has two creation routines, `make_debug` and `make_random`. Command `make_random` should be used with debug mode off and `make_debug` with it on. This little project is to help you obtain the same ship placement in debug mode that the Oracle has. The code is not well-designed; it is merely there to help you obtain ship placement matching the Oracle. Your design should do better.

Class RANDOM_GENERATOR produces 3 random variables: direction, row and column. Row and column are the y and x variables respectively. If direction is odd, the ship is vertical, otherwise, it is horizontal. Ships are placed from largest to smallest: 1st is of size n×1, 2nd is (n – 1)×1, ... , nth is 1×1 where n is the number of ships being placed.

Vertical ships are placed from top to bottom, horizontal from left to right. Generated values are guaranteed to be within the board's boundaries, with enough space to place the ship, given that the correct board_size and ship_size are provided. A newly generated ship may, however, collide with an existing ship. If it does, that ship's coordinates should be discarded and a new set of coordinates should be generated for a ship of same size. The forth command can be used to generate a new set of coordinates.

5.6 Design documentation: BON/UML Diagrams

In this course, we use the following diagrams: BON and UML class diagrams, UML sequence diagrams, and UML state-charts. For the UML diagrams, study the following slides: **https://wiki.eecs.yorku.ca/project/eiffel/_media/bon:uml.pdf** (Prism login required).
In the project, you will have to provide a Design Document. For the Lab, we will require you to submit two BON class diagrams. Details will be provided on the course wiki.

Requirement: BON diagrams
Please see Lab3 for an example of a BON class diagram generated using the IDE and a BON diagram using the *draw.io* template.

5.7 Design Decisions

- Use ETF to construct an application that does precisely what the oracle does (character for character).
- Design a suitable architecture for the business logic. Specify the game logic using expressive contracts.
- Write your own acceptance test (in addition to the ones provided) to check all aspects of the game for conformance with the oracle.

You will be making several design decisions for which you must ultimately be prepared to justify your choice:

- The design of the board and player moves
- When a game has been won or lost (i.e. when to terminate)
- Reporting of scores
- Error/status reporting
- See OOSC2 chapter 23: *Principles of Class design*
- See OOSC2 chapter 26: *A sense of Style*

We grade your project by:

- Testing the submission of your design/code for correctness
- We also evaluate your design by examining your architecture (via BON class diagrams).

We test your project for correctness by running our own suite of acceptance tests. The behaviour of your system bust be **character-by-character** the same as the Oracle specification.

The following problems are serious and will impact on your ability to obtain a passing grade:

- Your project does not meet the required submission guidelines and directory structure
- Your project does not compile
- An exception is generated when running one of our acceptance tests
- Your system does not provide the correct status messages when running one of our acceptance tests
- The messages are correct but the business logic is incorrect (ie the output is wrong, either in a detail or in the macro sense)

5.8 Validation and Verification: acceptance tests

Figure 3 shows how acceptance tests fit into the Validation and Verification cycle. An acceptance test is a description of the behavior of a software product from the point of view of the user's requirements, generally expressed as an example or a usage scenario. In many cases the aim is to automate the execution of such tests. The Eiffel Testing Framework (ETF) provides a means of doing this. Regression testing at the acceptance testing level is an important method for developing reliable software.

5.9 Regression Testing

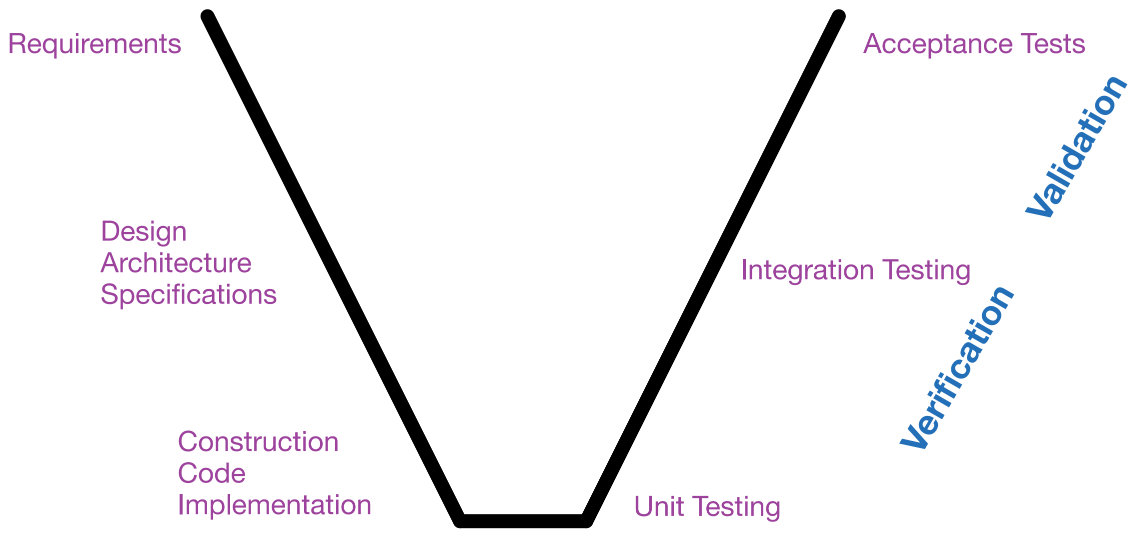
We provide you with Python3 scripts to do regression testing as you design/develop the software. Ensure that you use it and that you run the regression tests every time you make a change.

Regression testing is the process of testing changes to software to ensure that the product still works with the new changes. Regression testing is a normal part of the program development process. The old test cases are run against the changed product to ensure that the old features still work and that new bugs have not been introduced by the changes. Changing or adding new code to a program can easily introduce unintended behaviors and errors.

ETF adds the ability to introduce user requirements into a seamless process of software construction at a point before concrete user interfaces can be specified. ETF allows software developers to produce use cases that can be turned into acceptance tests, and that then free the developer to develop the business logic (the *model*}) while not losing sight of the user requirements. This allows requirements to become part of a seamless development from requirements to implemented code, and allowing change even at the level of requirements. Bertrand Meyer writes:

> The worldview underlying the Eiffel method ... [treats] the whole process of software development as a continuum; unifying the concepts behind activities such as requirements, specification, design, implementation, verification, maintenance and evolution; and working to resolve the remaining differences, rather than magnifying them.

Figure 3 Acceptance Tests in V&V



The terms **Verification** and **Validation** are commonly used in software engineering to mean two different types of analysis. The usual definitions are:

Validation: Are we building the right system?

Verification: Are we building the system right?

Validation is concerned with checking that the system will meet the customer's actual needs (the requirements), while **Verification** is concerned with whether the system is well-engineered, safe, and error-free.

Verification will help to determine whether the software is of high quality, but it will not ensure that the system is fit for use and useful to the customer.

Anyone who has worked in both specification and programming knows how similar the issues are. Formal specification languages look remarkably like programming languages; to be usable for significant applications they must meet the same challenges: defining a coherent type system, supporting abstraction, providing good syntax (clear to human readers and parsable by tools), specifying the semantics, offering modular structures, allowing evolution while ensuring compatibility.

The same kinds of ideas, such as an object-oriented structure, help on both sides. Eiffel as a language is the notation that attempts to support this seamless, continuous process, providing tools to express both abstract specifications and detailed implementations. One of the principal arguments for this approach is that it supports change and reuse. If everything could be fixed from the start, maybe it could be acceptable to switch notations between specification and implementation. But in practice specifications change and programs change, and a seamless process relying on a single notation makes it possible to go back and forth between levels of abstraction without having to perform repeated translations between levels.