EECS3311 — SDD (software design document)

This document provides a template for writing an SDD. Carefully read the documentation wiki before proceeding.[[1]](#footnote-1)

A formal template is provided on the wiki for future reference. However, the template in this document should be used for the 3311 project.

Software designers are experts at developing software products that are correct, robust, efficient and maintainable. Correctness is the ability of software products to perform according to specification. Robustness is the ability of a software system to react appropriately to abnormal conditions. Software is maintainable if it is well-designed according to the principles of abstraction, modularity, and information hiding.

|  |
| --- |
| A software *design* is the combination of a suitable *architecture* for the system modules and *specifications* of each of the modules.[[2]](#footnote-2)  The architecture is the decomposition of the system into modules (with clean interfaces) and a description of relationship between the modules (e.g. inheritance or client-supplier). The module behaviours – and their interconnection with other modules – are specified via contracts (pre-conditions, post-conditions and class invariants). A design that is not correct cannot be a good design. A design that does not have a suitable modular architecture is not a good design. |

A Software Design Document (SDD) thus describes the architecture, the design decisions (based on abstraction and information hiding) and the module specifications. Some principles about software documentation[[3]](#footnote-3):

|  |
| --- |
| * Every piece of paper should be forced to earn its keep as a useful document. * An SDD is useful only if it is of significant help during the design process itself (or in the maintenance phase to fix bugs and extend the functionality of the system). * If paper seems useless, ask how it can be made useful before deciding to abandon it. * Throw out any paper that isn’t kept updated. * If code isn’t important enough to document then throw it away right now.[[4]](#footnote-4) * Any project that is not documented should be abandoned when the original developer switches to working on a new project. * An SDD must have (a) all the information needed by the programmers to implement the design and (b) *nothing more*, i.e. it must be free of all implementation detail. |

For the EECS331 Project SDD, the sections of the SDD are as follows:

|  |
| --- |
| 1. Software Product Requirements  2. BON class diagram overview (*architecture* of the design)  3. Table of modules — responsibilities and secrets  4. Expanded description of design decisions  5. Significant Contracts (Correctness)  6. Summary of Testing Procedures  7. Appendix (Contract view of all classes, i.e. their *specification*) |

**1. Requirements.** This should be less than a page. It gives a brief outline and then refers elsewhere for the formal software requirements document (SRD).

**2. Architecture Overview.** This part consists of two pages.

(a) On the first page, provide a BON class diagram – at the right level of abstraction – that is an overview of the software that you edited or wrote.[[5]](#footnote-5) The diagram fits on a single page, is clean and relevant (no small fonts). Use the BON Visio template to draw this diagram.[[6]](#footnote-6) Note that a Visio BON diagram can be inserted into a Word document so that the display of the diagram is crisp. In an Appendix, you may add additional information such as UML statecharts and sequence diagrams, if needed.[[7]](#footnote-7)

(b) On the second page, describe the overall design and the main design decisions. This may include criteria such as simplicity, choosing the right abstractions, information hiding, reliability (decreasing the likelihood of bugs), re-usability (minimizing work needed to use components elsewhere) and extendibility (minimizing adaptation effort when the problem varies).[[8]](#footnote-8)

**3. Table of modules** (information hiding).

This is where you provide an overview of each module in your system, its “secret” (based on *information hiding*) and the design decisions used in the production of the module.

In OO design, a module might be a single class or cluster of classes. Each module has a clean public API but also an *information hiding* secret that other modules need not know to use the services of the module. Below we provide an example of how you can document the modules and their secret.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | LIST[G] | **Responsibility**: a sequence of items of type G | **Alternative**: see ARRAY[G] |
| Abstract | **Secret**: none |

|  |  |  |  |
| --- | --- | --- | --- |
| 1.1 | LINKED\_LIST[G] | **Responsibility**: see LIST[G] | **Alternative**: see ARRAYED\_LIST[G] |
| Concrete | **Secret**: implemented via cells each with a reference to the next cell without circularity or duplication. See 1.1.1. |

|  |  |  |  |
| --- | --- | --- | --- |
| 1.1.1 | CELL[G] | **Responsibility**: record of data and a reference to another cell. | **Alternative**: none |
| Concrete | **Secret**: none |

|  |  |  |  |
| --- | --- | --- | --- |
| 1.2 | ARRAYED\_LIST[G] | **Responsibility**: see LIST[G] | **Alternative**: see LINKED\_LIST[G] |
| Concrete | **Secret**: implemented in contiguous memory amortized over constant time re-allocation |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | ADT\_BAG[G 🡪 {COMPARABLE,  HASHABLE}] | **Responsibility**: unordered collection of hashable items with possible multiplicity and a sorted domain. | **Alternative**: a more generic bag without the constraint of a sorted domain, and/or without the constraint of hashable items. |
| Abstract | **Secret**: none |

|  |  |  |  |
| --- | --- | --- | --- |
| 2.1 | BAG[G 🡪 {COMPARABLE,  HASHABLE}] | **Responsibility**: see ADT\_BAG | **Alternative**: implement with two arrays, the first for the data item and the second to store the multiplicity. This would not take advantage of the look-up efficiency of hashable items. |
| Concrete | **Secret**: implemented with hashing and counting to take multiplicity into account. See HASH\_TABLE |

Note the numbering system to denote the abstraction hierarchy. The LINKED\_LIST module includes the CELL class.

“Responsibility” means the primary responsibility. If there too many secondary responsibilities, then the module might be a “Superman” module attempting to do too much. The *Responsibility* and *Secret* should be documented in the header of the class text. Responsibilities and secrets must be described briefly in less than two or three crisp sentences. The next section provides opportunities for expanded description.

For an ETF project, do not include generated classes in the module table. Include only those classes that you edited or created in the *model* cluster.

**4. Expanded description of modules.** This is where you provide detailed descriptions of all the sub-systems and modules in your design and their relationship to the rest of the design. Describe the module design decisions in terms of data structures and algorithms used, trade-offs etc.

For the 3311 project, choose only *one* module to document — the most important module in your design. Specify the module and the design decisions associated with it (i.e. expand the information provided in the table). You may use additional BON and UML diagrams in this section. Limit this section to 2 pages or less.

**5. Contracts.** This should be less than two pages. Choose a module that has the most significant contracts and describe the contracts and their significance.

**6. Testing.** (a)Provide a table of all the acceptance tests that you ran and whether they were successful. The table might be constructed as follows:

|  |  |  |
| --- | --- | --- |
| **Test file** | **Description** | **Passed** |
| *at1.txt* | Normal scenario where product types are created, orders are placed and invoiced. | ✓ |
| *at2.txt* |  |  |

(b) Provide a screen shot of the *ESpec* unit tests that you ran. Ensure that the test comments are descriptive.

**7. Appendix (Contract view of all classes).** Use the EStudio/IDE documentation tool to generate a formatted (RTF) *contract view* for each class mentioned in the module table (except for the input command classes). Edit the text so that the formatting is readable and professional, and there are no line wraps.

If your contract view is empty (just feature signatures) then that might mean that your specification of behaviour (via meaningful comments, preconditions, postconditions and class invariants) is incomplete.

The actual template for your submission starts on the next page. *Delete pages 1-4.*

EECS3311-W?? — Project Report

Submitted electronically by:

|  |  |  |  |
| --- | --- | --- | --- |
| **Team members** | **Name** | **Prism Login** | **Signature** |
| Member 1: |  |  |  |
| Member 2: |  |  |  |
| \*Submitted under Prism account: | |  |  |

\* Submit under **one** Prism account only   
Also submit a printed version with signatures in the course Drop Box

**Contents**

[1. Requirements for Invoicing System 6](#_Toc415149333)

[2. BON class diagram overview (architecture of the design) 7](#_Toc415149334)

[3. Table of modules — responsibilities and information hiding 8](#_Toc415149335)

[4. Expanded description of design decisions 9](#_Toc415149336)

[5. Significant Contracts (Correctness) 10](#_Toc415149337)

[6. Summary of Testing Procedures 11](#_Toc415149338)

[7. Appendix (Contract view of all classes) 12](#_Toc415149339)

|  |
| --- |
| **Documentation must be done to professional standards**. See OOSC2 Chapter 26: *A sense of style*. Code and contracts must be documented using the Eiffel and BON style guidelines and conventions. *CamelCase* is used in Java. In Eiffel the convention is *under\_score*. Attention must be paid to using appropriate names for classes and features. Class names must be upper case, while features are lower case. Comments and header clauses are important. For class diagrams, use the BON conventions, and use clusters as appropriate. Use the EiffelStudio document generation facility (e.g. text, short, flat etc. RTF views), suitably edited and indented to prevent wrapping, to help you obtain appropriately documentation (e.g. contract views). Each diagram must be at the appropriate level of abstraction. Use Visio for the BON class diagrams.  Your signature attests that this is your own work and that you have obeyed university academic honesty policies. Academic honesty is essentially giving credit where credit is due, and not misrepresenting what you have done and what work you have produced. When a piece of work is submitted by a student it is expected that all unquoted and uncited ideas and text are original to the student. Uncited and unquoted text, diagrams, etc., which are not original to the student, and which the student presents as their own work is considered academically dishonest. |

# Requirements for Project SimOdyssey

Our customer approached us in need of a galaxy exploration simulator to help train their clients for space exploration. As stated by our customer, because of “the combined effects of the hole in the ozone layer, global warming, nuclear catastrophes and the fear that ABBA might still reform”, there is a need to explore the galaxy and find another planet capable of supporting life. The simulator will help train future space explorers by having them navigate through a virtual representation of our galaxy where they will face real life scenarios. The simulation will end when an explorer finds a planet that can support life or the explorer dies. Only planets that orbit a “Yellow Dwarf” star will have a chance of containing life. If an explorer is in the same sector as a planet that is orbiting a “Yellow Dwarf” star, they can land on that planet to check if life is supportable. There are multiple scenarios in which an explorer can die: running out of fuel, being devoured by a black hole, getting destroyed by an asteroid and being killed by extra-terrestrial beings.

The galaxy will be represented by a two-dimensional 5x5 grid. Each sector of the grid will be identified by its coordinates in terms of row number and column number. Sectors contain the entities of our galaxy (i.e. planets and stars). A sector contains 4 quadrants that hold one entity each meaning a sector can contain at most 4 entities. An explorer can move to any of its 8 adjacent sectors given that they are not full. One of the specifications as stated by our customer is that “the grid wraps along its boundaries meaning if we go north from a sector in the first row, we will move into the fifth row at the bottom of the grid”.

There are 2 types of entities in our galaxy, movable and stationary. Movable entities, as the name suggests, can move to different sectors whereas stationary entities never move and remain in the same location it was created in. These entities each have different behaviours that will help simulate what it is like to explore the depths of space. More information regarding entities and the actual simulator can be found in the *Software Requirements Document.*

# BON class diagram overview (architecture of the design)

The game Simodyssey is divided into several clusters which communicate with each other. The cluster ‘user\_commands’ is the cluster that deals with the user interface. This cluster contains nothing besides the user input commands and is structured to be ported correctly to a GUI or any other user interface. The cluster ‘user\_commands’ depends on the model cluster, but not vice versa. This allows the business logic design of the game to be ported to any user interface with only the user interface needing changes. The cluster ‘user\_commands’ accesses the model class ‘GAME’ through a singleton design pattern, as there is only one instance of the game Simodyssey. This instance is accessed through the class ‘GAME\_ACCESS’ which returns the single instance of the game. This is also how the cluster ‘user\_commands’ accesses the cluster ‘model’ as can be seen in the BON diagram. Within the cluster ‘model’, is the class GAME which is the main model class of our design. This class correctly handles all user input commands and executes them accordingly. This is a singleton class, and thus only one instance of the GAME class, and thus the Simodyssey game exists at once.

The model cluster communicates with another cluster, namely ‘universe’. It does this by storing an instance of the class ‘GALAXY’ in ‘GAME’. This is the only point of communication between these two clusters. The ‘model’ cluster depends on the cluster ‘universe’ but not vice versa. This allows the user commands to be completely disassociated from the logic of our Simodyssey universe. The cluster ‘universe’ is the representation of the universe that our explorer travels through in the game. This cluster handles all of the logic that is necessary to have a functional universe and allow the user to guide the explorer in the game. The class “GALAXY” in this cluster is the access point for the model into the ‘universe’ cluster. This class represents the galaxy in the universe in which our explorer travels. In this galaxy there are several entities, which are each represented by their own classes. The cluster ‘universe’ contains a sub-cluster, namely ‘entities’. This cluster represents all the entities in our universe, some of which can move around and other which are stationary. The stationary entities in this cluster are only stars and are represented by the class ‘STATIONARY\_ENTITY”. All other entities are moveable entities and their inheritance is correctly designed as can be seen in the BON diagram. The design of this inheritance relationship allows our universe to add more entities to our universe without changing existing code in the inheritance design. These new entities can be either movable or stationary entities, they will inherit from the respective class they fall under.

The design of our game as discussed and shown in the BON diagram thus is very reliable, reusable, and extendable. Breaking the game up into several clusters/modules as shown allows for simplicity, and the inheritance hierarchy of components allows for reusability, reliability, and extendibility.  The abstraction of the Simodyssey game into its several components allows for a decreased likely hood of bugs as well as a simple design to the game. The contracts in the class ‘GAME’ also allow for a decreased likely hood of bugs, as the post- conditions ensure that each feature is executing as expected. Pre-conditions are handled through error handling, with correct message outputs by the system. With this design, more entities as well as user commands can be added to the software with no required changes to already existing features.

# Table of modules — responsibilities and information hiding

# Expanded description of design decisions

*Only for the most important module in your design.*

*What alternative designs were considered and rejected based on the criteria of reliability, simplicity, and maintainability?* The design is maintainable if it exhibits conceptual integrity that defines the key abstractions so that designers and programmers can reason about the system you describe and predict its behaviour. Software developers reading your SDD should be able to grasp your design without having to read thousands of lines of code. This will make you system extendible and re-usable.

# Significant Contracts (Correctness)

Module: **GAME**

**move (dir: INTEGER\_32)**

***moved\_to\_diff\_sector:***This post condition ensures that the old explorer location is different from the current explorer location. This condition is only checked when the *move\_err* string is empty otherwise there might be some underlying problem that caused the explorer to not move (ex. sector is full) which would invalidate this contract. This condition is significant in the sense that it entails exactly what should happen after the move command is called, the explorer should move to another sector.

***move\_msg\_not\_empty:*** This condition ensures that regardless of if the move command was successful or not, there is still some feedback and information to output for the user. It returns true if either the *move\_err* or *move\_msg* string is not empty. If the *move\_msg* string is not empty it means the explorer moved successfully to another sector. Alternatively, if the *move\_err* string is not empty then there was an error that occurred when trying to move. This post condition is significant because it makes sure the move command returns an output even if it fails. This is important because without a failure message the user will have no way of knowing why a move was not successful.

**land**

***landed\_on\_a\_planet:*** This condition ensures that whenever a successful land command is called, the explorer is indeed landed on a planet. This contract implies that there were no errors when calling *land.* This is verified by checking if the *land\_err* string is empty or not. We know if the explorer is landed by checking its *is\_landed* field. This is a Boolean that is *true* when the explorer is grounded on a planet and *false* otherwise. This post condition is relevant because the premise behind the *land* command is to have the explorer landed on a planet. If this were not the case it would make it impossible for the explorer to find a planet that supports life since this can only be done by landing on the planet.

***land\_msg\_not\_empty:*** This post condition ensures that there is always an output being returned even if the *land* command fails. It is significant because it gives the user insight on what happened when trying to land on a planet. This is important since the objective of the simulation is to find a planet that supports life and the *land\_msg* is what indicates whether or not this is the case.

**liftoff**

***not\_landed:*** This contract verifies that the explorer is not landed on a planet whenever the *liftoff* command is successfully called. The nature of the *liftoff\_err* string will determine if there were any errors. Similar to the ***landed\_on\_a\_planet*** contract, we know the explorer is not landed on a planet when its *is\_landed* field is set to *false.* This contract is significant because it ensures that the explorer detaches from a planet whenever a successful *liftoff* command is called. If this were not the case, it would be impossible for an explorer to get back into space since this can only be done by calling *liftoff*. Many complications would arise from this since the explorer can only move to different sectors when not landed on a planet meaning it would be impossible to search for other potential habitable planets in the galaxy which is essentially the goal of this simulation.

***lift\_off\_msg\_not\_empty:*** This post condition ensures that an output is being returned when the *liftoff* command is called. This condition is significant since there is no way of visually seeing if the explorer is landed or not on a planet, the only way to determine this is by the messages that are outputted.

**movement (m\_ent: MOVABLE\_ENTITY)**

***movements\_increased:*** This condition ensures that whenever the *movement* feature is called, the move that was made is appended to the *movements* linked list. This is done by verifying that the size of *movements* is greater than its old version. The significance behind this contract comes from the fact that whenever a movable entity switches sectors, we need to keep track of this so we can output all the movements made on a given turn. Even if an entity failed to move, there is still a string appended with the current sector of that entity. This indicates that it tried to move but failed somewhere in the process.

**play**

***is\_in\_game:*** This contract ensures that when the user calls *play*, the *in\_game* Boolean is set to *true.* This Boolean is important because it is how the simulator deciphers if there is a current game being played. In the case that the *play* command was successful this Boolean should be set to true since the user is now in game. This is also the case when the *play* command fails because the only time this command fails is if the user is already in a game so in both these cases the Boolean would be set to *true.* This contract is significant because a considerate amount of the error handling that occurs comes from if we are currently in game or not so any invalidity of this command could seriously jeopardize the wellbeing of the entire simulation.

# Summary of Testing Procedures

# Appendix (Contract view of all classes)

(Only classes that you created; do not include user input command classes, only model classes)

1. <https://wiki.eecs.yorku.ca/project/sel-students/p:tutorials:sdd:start>. (Login with your Prism account, at the bottom, for access) [↑](#footnote-ref-1)
2. “*Architecture* is the structure needed to reason about the system. Each structure comprises elements, the relations among them, and the properties of the elements and relations.” (L. Bass et. al., *Software Architectures in Practice*, Addison-Wesley, 2013). Architectures must exhibit conceptual integrity that defines the key abstractions so that designers and programmers can reason about the system and predict its behaviour. [↑](#footnote-ref-2)
3. See Philip Koopman, *Better Embedded Software*, Drumnadrochit Press, 2010. [↑](#footnote-ref-3)
4. Koopman, *op.cit*. p20. [↑](#footnote-ref-4)
5. Every design document should have, as a minimum, a boxes and arrows diagram (with well-defined meanings) that describes the overall structure. In OO designs, we use BON and UML. [↑](#footnote-ref-5)
6. See <https://wiki.eecs.yorku.ca/project/eiffel/bon:start.> [↑](#footnote-ref-6)
7. For UML, see <https://wiki.eecs.yorku.ca/project/eiffel/bon:start#uml>. [↑](#footnote-ref-7)
8. See B. Meyer, Touch of Class, Springer 2009, p684-685. [↑](#footnote-ref-8)