EECS3311 — SDD (software design document)

This document provides a template for writing an SDD. Carefully read the documentation wiki before proceeding.[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. 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[2]:

- 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.[3]
- 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) be free of all implementation detail.

[1] https://wiki.eecs.yorku.ca/project/sel-students/p:tutorials:sdd:start. (Login with your Prism account, at the bottom, for access)
[2] See Philip Koopman, *Better Embedded Software*, Drumnadrochit Press, 2010.
[3] Koopman, *op.cit*. p20.

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). This page is already filled in for you.

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.[4] The diagram fits on a single page, is clean and relevant (no small fonts). Use the BON Visio template to draw this diagram.[5] Note that a Visio BON diagram can be inserted into a Word document so that the display of the diagram is crisp.

(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).[6]

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

[4] 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.
[5] See https://wiki.eecs.yorku.ca/project/eiffel/bon:start.
[6] See B. Meyer, Touch of Class, Springer 2009, p684-685.

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

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

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-W15 — 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

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. See model solution for Assignment 1, posted outside LAS2056.

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.

1. Requirements for Invoicing System

Our customer provided us with the following statement of their needs: The subject is to invoice orders. To invoice is to change the state of an order (to change it from the state “pending” to “invoiced”). On an order, we have one and one only reference to an ordered product of a certain quantity. The quantity can be different to other orders. The same reference can be ordered on several different orders. The state of the order will be changed into “invoiced” if the ordered quantity is either less or equal to the quantity which is in stock according to the reference of the ordered product. You have to take into account new orders, cancellations of orders, and entries of quantities in the stock. A console based application for user input suffices.

Analysis of the requirements and further description may be found at the following URL: https://wiki.eecs.yorku.ca/course_archive/2014-15/W/3311/protected:assign:project:phase1

2. BON class diagram overview (architecture of the design)

3. Table of modules — responsibilities and information hiding

4. Expanded description of design decisions

(Only for the most important module in your design)

5. Significant Contracts (Correctness)

(only for the module with the most significant contracts)

.

6. Summary of Testing Procedures

7. Appendix (Contract view of all classes)

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