# 2720 Project Coding Conventions

## File Prerequisites

In general,

-Having a .CC or .cc file in our repository should imply that the given Header/Specification file is present, complete or (preferably) both. An incorrect header should be listed in the .cc file and notated as such. This is to prevent a cascading incompletion – to stop us from prematurely assuming any necessary functionality is finished when, in fact, it is not.

-Test files and Test Data should be segregated (directory wise) from any actual data or save files, such as our XML savefiles and similar. This is to prevent any loss of data should a test case ‘go awry’.

## “For Later Compilation” Code

As some code will need to be overly rewritten when most of the headers are complete – that is, some code is overly dependent on the correctness of other header files that may yet be incomplete – we need to notate some functions and function areas as having to be rewritten later on.

As such, we will use markers such as ‘##’ and ‘!!’ followed by an explanation keyword (i.e. CHANGE\_BEFORE\_RELEASE or THIS\_IS\_BAD\_CODE (rewrite/optimize this code before compilation)

This will allow us to quickly locate (what we currently believe will be) the major problem points of the program.

## File Types and Names

All specification/header files will have ‘.h’ affixed to their filename.

All implementation/class/driver files shall have a ‘.cc’ affixed to the end of their filename.

This should serve to create distinction between the two types and allow for greater organization of them when they are completed.

It should also allow one to search/find a given header file simply by affixing the relative implementation name with ‘.h’.

## Method Names

All method names will adhere to certain rules as defined by the following. Methods will try to be comprised of few descriptive verbs that will explain what the method will accomplish in the shortest simplest way possible. The convention will try to be descriptive enough for self-documenting, however a balance will have to be found to ensure on going readability but not tiring to type while coding. The first letter of the verb used to comprise the method names will be capitalized this will also apply for multiple word method names. Here is an example:

SetCharacterJob()

## Variable Names

Variable names will follow the same convention that the method names follow, with one modification. The first letter of the variable name will not be capitalized to contrast the use of Method names. Variable names will be fully descriptive nouns rather then verbs as described under Method names. Here is an example:

Int characterExperience;

## Enumerated Types

Enums/Enumerated types will always be completely capitalized, and properly declared near the top of an implementation or specification file. They should be prefixed with what general data they apply to. This is to ensure that our enumerated types are self-documenting and thus, easy to alter or remove if necessary. As a side-effect, they are easier to search for if we’re dealing with the related type.

(*i.e. for the below STATS, we’d be in a level routine when we search for it*)

STATS\_MAXLEVEL = 0;

## General Coding Conventions

All curly braces used throughout the project will adhere to the following standards. The first brace right after an if-statement will follow on the next line right below the first letter. In the case of an if-statement the brace will be located right under the ‘I’. The code following the curly brace will then be indented four spaces in to indicate that the code is within the if-statement and to make the code readable for all those who are on the project team and third parties. If-else statements that contain only one line of code following will not be required to contain curly brackets. All logical statements located inside if-statements will not be split on multiple lines in order to reduce confusion.

int Foo(bool isBar, bool isNotBar)

{

if (isBar || isNotBar)

{

bar();

return 1;

}

else

return 0;

}

## The use of Comments

Standard documentation will be used throughout the project including standard block comments and single line comments to help reduce confusion across the team. Then to address Doxygen documentation requirement, standard doxygen comments will be used only in the header file.

To document further the use of Doxygen commenting, the team will also be required to specify the following flags on each class method.

* Parameters
* Preconditions
* Postconditions
* Return values

Each class will be heavily documented to show all the requirements of the class clearly in order to provide test cases for later use. This will also be useful for understanding the flow of data and how the classes fit together.

## Bug Tracking

The team has decided to use bug-tracking software; the software to be used will be bugzilla. Bugzilla will be installed on an external server, bought and paid for courtesy of Chad Klassen. The data will then be dumped once the project becomes due to show that bugs were tracked and to reduce bloat on the SVN server.

## General Document Tracking

General document tracking is already taking place, the team has established a common place to post information, talk and share files across multiple platforms and can be accessed via anywhere. Chad has also set up a blog on his paid hosting to support the efforts of this project. Team members have been posting document revisions, charts, ideas, code and comments already on the blog. The blog will remain in use, and just like the bug tracking all the information will be downloaded from the server and will be submitted along with the rest of the required documents. The blog is located at <http://2720.doodlespark.ca> if there are any questions or concerns about the project, steady progress can also be noted by documents that are posted every few days.

## Test Cases

Each class that we provide testing cases for will be designed to test all branches of the class and its methods. Each test case will have at least one case, then one will be added for every if, while, do while, for and each logical operator. On top of this, two test cases will be provided to test both sides of every requirement specified for each class and it’s methods. The requirements are required to be very specific in order to come up with optimal test cases. If a requirement has sub routines or sub requirements, each one will also have two more cases to test all branches.

Case documentation will be comprised of the following data fields:

* Date: The date the test case was made
* Author: Who made the test case. Typically this will be whoever wants to test that section of code.
* Class: Define the class that the test case is for. To refer back to it.
* Method/Subroutine: What method is this test for? If any. If its not a method test this will be left blank
* Type of Test
  + Branch: Test for the incremental testing for all ‘if’, ‘while’, ‘do-while’, ‘for’, conditional statements.
  + Requirement: Tests to make sure requirements are met
  + General: Other misc tests, this will be comprised of regression tests, unit tests,
  + Boundary Test: To test the boundaries of any given, arrays, vectors or lists.