**UWROV Coding Style Guidelines**

**Table of Contents**

* Preface | 1
* Commenting | 1
* Documentation | 2
* File Header | 2
* Code Format | 3
* Class Design | 4
* Naming Conventions | 5
* Efficiency | 6
* Logic and Redundancy |
* Miscellaneous |

**Preface**

Similar to the major companies in the software industry, many classes at UW has developed a style guideline specific for their classroom setting. In effort of organizing the club structure, the officers of UWROV has decided to also adapt to a general style guideline that is easily followable by everyone to develop, manageable, understandable, and dynamic code for those in the future to work with.

The goal of this style guideline is the same as the goal of any other style guideline: making it easier for those in the future to be able to read and understand the code. Following these guidelines will reduce the probability of having unmanageable messes that is hard to follow. This in turn makes debugging much easier and hopefully speed up the process of development. We will also focus on providing a strong documentation that does not require hours of scrolling through code that has been written by others. This in turn will allow others to use code snippets as modules without necessarily needing to understand the inner workings of the module unless out of curiosity.

In general summary, our major goals for this first year will be:

* Developing readable and manageable code for the future.
* Provide documentation that is precise, has the necessary information to understand the state of the current code, and lets new users able to use the code with only the documentation.
* Set a standard for the future teams so everything is organized for them.

Although at first glance these guidelines may seem like an annoyance and extra work for you all, I must stress the importance of organization. With proper organization from the start of development, it prevents more work in the future as everything will be documented and much more manageable to those that do start with no understanding. I hope that the guidelines are intuitive and understandable to everyone and cheers to another great year!

**1 Commenting**

Although there are many other topics to start on, I start on the commenting and documentation of programs to emphasis the importance we will start putting on comments and documentation of each file that will be in the final production. To prevent the exhausting search of looking through programs made by previous members to understand how everything works, each file will be commented relatively thoroughly and be followed by a documentation file (Section 2) that has a thorough documentation on the functionality, and the general workings of the program so that people in the future will be able to manage the program without needing to understand all of the code that has been written by those in the past.

* 1. **The Header**

The header should describe a simple general overview of the program and a summary of what sort of functionality it has. This is also the place to put your name and date if you wanted to.

* 1. **Class Comments**

If this is java, then your class comment is equivalent to the header so you may skip this. In other languages where classes are defined, it will be useful to have a general description of what the class contains and what it represents.

* 1. **Class Fields / Global Variables**

Although global variables are highly discouraged to the point where we would consider it “bad style”, this commenting applies to just variables and fields that have the scope of the whole file. It is generally helpful to label what each variable represents if the variable name does not provide the most obvious definition.

* 1. **Method Headers**

Method commenting is usually the most difficult things especially if you have taken one of the computer science courses at UW. Hopefully this section will clearly define what is considered “good” style when it comes to commenting.

We will use a basic standard of “Javadoc” commenting because it is organized and easy to read and understand. This structure also easily applies to multiple different languages.

You can look up a tutorial on how to write Javadoc comments but what matters for us is just the general structure of the commenting style.

General Structure:

// (Description: a general description of the “**behavior**” of the program)

//

// @param (object type) (variable name) (general description of what the parameter represents)

// …

// @return (object type) (general description of what the function returns)

//

// @exception (exception type) (A description of why the exception is thrown)

The param, return, and exception, is only required when the method has parameters, returns objects, and throws exceptions. If they do not apply, they are unnecessary.

* + 1. **Difference between behavior vs implementation**

There is a difference between commenting the behavior of a method and the implementation of a program. What should be commented is the behavior. Simply describe what the method does; if it has an input, what is done with the input? if there is an output, what is the output? The user of the method does not necessarily need to know what is happening “under the hood”. Just simply comment what is done with the method. Not how it is done. The how is what implementation is.

**2 Documentation**

Documentation is also an important part of organizing code. The purpose of writing out more documentation is to just reinforce the foundations of organization. By putting all of the comments into one readable file, everything can be looked at there with less confusion while trying to look through the code and the commenting at the same time. In an ideal situation, the documentation file will not be any more than the collection of the comments in the program into a new document that can be read like a book. This is where the programmer could also put extra details and notes for those in the future.

If commenting is already properly done, documentation can just be copied and pasted into a documentation file. It should include:

1. A description of the file and its contents, its functionality.
2. A list of the fields/important variables in the file.
3. The methods and the method comments.
4. Maybe some examples and usage of the methods.

Again, with good commenting, good documentation would require less effort.

**3 File Header**

Now that commenting and documentation is out of the way, everything else will be relatively straightforward.

**3.1 Package/Import statements**

Any import and package statements should always be at the top of the file.

**3.2 Order of constants, fields, and methods.**

We will want to keep constants and fields at the beginning of the file preferably in that order. Methods don’t specifically have a general order other than hopefully they are in a logical order. Believe in your intuition and your ability to organize.

**4 Code Formatting**

We want our code format to look uniform throughout. Especially details such as indentation space and spacing. If everyone adapts to a singular format, all code will be readable to anyone that knows the formatting that we use. This also sets the precedent for indentation spacing and line characters for those working in different coding environments.

* 1. **Indentation**

Indentation should be formatted based on blocks of code. Python is simple because the whole structure is based on proper indentation. However, for other languages such as java, JavaScript, CSS, we will want indentation to be consistent with the curly braces and new blocks of code. Every time curly braces starts; the next line should be indented once and then return back to the initial indentation once the curly brace has returned.

Example:

functionHeader() {

//

// code

//   
}

**IMPORTANT:** We will have default indentation set to 3 spaces in order for all developing environments to be displayed and indented correctly. Without a uniform indentation spacing, it becomes hard to manage code from multiple sources with different indentations. **Make sure to** **set** the default indentation in your developers’ environment to be set to 3 spaces for tabs.

* 1. **Spacing**

Spacing between code should make everything readable. With some exceptions in html, there should always be spacing between individual parts in a line of code. Make sure there are no random double-spaced areas or no spaced areas.

Examples:

* Bad:

for(x=0; x<3;x++)

* Good:

for ( x = 0; x < 3; x++) {

* Bad:

int x=4 \*(3+ 2)- 1;

* Good:

int x = 4 \* (3 + 2) – 1;

You get the point

* 1. **One statement per line**

Don’t have multiple statements in one line just to make things shorter. It doesn’t matter if a method is all in one line if it is unreadable.

Don’t do!!!!!

Return method1(var1).method2(var2, var3).method4(method5.());

There are times where it seems reasonable to have multiple methods inside another method, but they are very rare cases. Relatively split these methods into multiple lines and make them readable!

* 1. **Column limit and line wrapping**

No line should exceed 100 characters in length.

If method or functions are in need to be wrapped to the next line, they should be logically indented to indicate that the previous line has been wrapped to the next line.

Good example:

java example but applies to any other language

public static void method(Object variable1, Object variable 2, Object variable 3

Object variable4, Object variable 5) throws ExampleException {

}

* 1. **Exceptions**

Exception code should be isolated and have priority at the top of the code. This means that if an exception can be checked at the beginning of a method, it should be checked there. Exception check statements should also not have any of the main code attached to the block. In other words, don’t have the code start with an If statement checking for an exception that is connected to an Else statement that has the method behavior in it.

Non-isolated exception check (bad style):

Method() {

If(Exception happens) {

Throw exception;

} else {

// method behavior

// ie. Print(“hello world!”);

}

}

* 1. **Curly Braces**

There isn’t a specific way curly braces should be formatted but the general rules are:

* No line break before the opening brace.
* Line break after the opening brace.
* Line break before the closing brace.
* Line break after the closing brace, *only if* that brace terminates a statement or terminates the body of a method, constructor, or class. For example, there is *no* line break after the brace if it is followed by else.

**5 Class Design**

Like many skillsets, a program that is really good at one single thing is generally better than a program that is okay at many things. Classes are just a smaller part of a program; they should be very good at a single thing. This doesn’t mean that each class should have one function. They should have a single general purpose that it excels at and minimal ways for it to “break down”. In that sense, the ideas that are in this section are to minimize ways for classes to “break down”. It is your job to make the program to excel at its job. Also, many of the access modifiers specifics are specifically for java.

**5.1 Fields**

1. Fields should be instantiated inside of the constructor of a class; not when it is declared at the beginning of the class.
2. Fields should generally be declared as private.

* Having private fields allows the programmer to manage the class strictly and prevent unintended usage.
* However, there is “generally” in this rule because there are moments where public class fields are allowed if the class is meant to have its fields used publicly.

1. There should only be a necessary number of fields. Try to avoid unnecessary fields that could be calculated without much processing power.

* An example of this would be a length field for an array. The length of an array can be calculated with array.length; having another field in the class for the length of the array would be redundant.

**5.2 Class Constants**

The use of constants is mostly to help organize “magic numbers” into meaningful variables. Similar to how we use π(pi) to represent the number 3.141592…., it helps organize our constant values to be isolated as class constants.

Class constants should be declared with public, static, final, modifiers to ensure that the constant is a constant.

Note: Class constants are instantiated when t. hey are declared unlike class fields.

**5.3 Methods**

1. Methods don’t all need to be public. Only have methods that are used by the instances of the class be declared public. Other methods should be private.
2. Only have parameters that are required. If there are other means of getting the variables outside of having a parameter, try to do that instead. Managing many parameters become very difficult. This also means that get rid of all of the parameters that are not used in the method.
3. Only certain methods require data to be returned. Specific to java, if a method changes an object passed as a parameter, it is unnecessary to return the object again because the object has been modified already.

**5.4 Unnecessary behavior**

This is generally just for random “fluff” inside your classes. If there are methods, fields, or details that are unnecessary, get rid of them! This doesn’t mean get rid of helper methods or reduce them; those are necessary. Unnecessary things are things that do not serve any purpose to the class.

* 1. **Constructors**

1. When applicable, reduce redundancy by using the this() keyword to call another constructor in the same class. Generally, the constructors that assume more defaults (fewer parameters) call the more general constructors (more parameters).
2. Clients of a class should never have to set fields of an object immediately after construction - there should be a constructor included for this situation. The general behavior of a constructor is to set up the state of an object correctly, and this should encompass setting fields.
3. Clients of a class should never have to pass in dummy values to a constructor to set up the state they want - instead, there should be a constructor included so that only the relevant parameters need to be passed.
4. **Naming Conventions**

Depending on which language is used, there are many different naming conventions. The standard casing for languages such as Java and JavaScript is camel casing while python uses underscore casing. Depending on the language there are standards for each language that are followed.

Generally, the conventions for casing is:

* Lower case (lowerCase) for start of variables, methods, pointers,
* Upper case (UpperCase) for classes
* Screaming case (THIS\_IS\_A\_CONSTANT) for constants.

**6.1 Description**

Every variable name, method, class should have intuitive naming. Although there should be general description for the purpose of each variable ideally, it will be sufficient enough to have a descriptive name that allows anyone reading your code to know what the component represents. Good naming conventions will also help organize your code so debugging will be easier in the future.

Examples:

number - this name is not very descriptive because it does not say what the number represents. The only time this sort of variable would be acceptable is if it was a temporary variable and even then the variable would be more descriptive if it was tempNumber.

numberOfCars – this is more helpful because the variable describes that the variable is a number and what the number represents.

1. **Efficiency**

It takes a lot of experience and effort to write efficient programs and there is no one method to make the most efficient program. Some questions you should ask to check if your program can improve on efficiency are:

1. Are you using more data or resources than is necessary?
2. Are you recomputing values that take more resources than just saving it?
3. How much time is your program taking to run?

Some common mistakes that can be prevented include:

* Creating unnecessary objects:
  + In the case of Java, this is very easy to overlook. Whenever the “new” keyword is used, a new object is created which takes up space in the memory storage. This means that creating objects that are discarded without being used is a waste of resources.

Example:

List<String> words = new ArrayList<String>();

Words = generateWords();

This may at first look mundane, but the program creates a new ArrayList object and instantly dereferences it. This takes up unnecessary resources and can be prevented by changing this to.

List<String> words = generateWords();

* Method calls and computing math:
  + Making computations and calling methods takes up resources so it is important to think about if the method is worth calling and if it is more efficient to save values.

Example:

Let’s say that an object contains data of hundreds of test scores. Rather than having a method that calculates the mean every time it is called, it will be much more efficient to save the mean value and update the value once when the data is updated and save the value to a variable.

1. **Logic and redundancy**

There are moments where code is seemingly copied and pasted in multiple different methods. This section describes some practices that helps prevent code from being redundant and messy. The less complicated loops, if/elses, computations, the easier the code is read.

**8.1 Boolean Zen**

Sometimes it’s easy to forget that Booleans can be passed and be simply used in if statements. Boolean Zen is all about using Boolean values efficiently and concisely.

Examples:

* Wrong:

If (test == true) {

// stuff

}

* Right:

If (test) {

// stuff

}

Note: test is a Boolean value which makes “test == true” equivalent to “test”.

* Wrong:

If (test) {

return true;

} else {

return false;

}

* Right:

return test;

Note: Since the if statement is returning the state of the variable “test”, it is more readable to just return the variable rather than encasing it in an if/else statement. This also applies to comparison statements such as “max == 3”. Rather than encasing the statement with an if/else statement that returns true or false, it is much clearer to state “return max == 3”.

**8.2 Inappropriate if/else**

If an if block has the same behavior across multiple if blocks, combine the if blocks into one if block with multiple arguments.

Example:

Wrong:

If (arg 1) {

print(“Hello World!”);

}

If (arg 2) {

print(“Hello World!”);

}

Right:

If (arg 1 && arg 2) {

print(“Hello World!”);

}

Wrong:

if (someTest) {

System.out.println("hello!");

}

if (!someTest) {

// Because this code always wants to do one or the other, (and

// doesn't involve a return or exception) we want to express this

// code more simply as an if/else

System.out.println("I'm redundant");

}

Wrong:

if (max < 0) {

// It doesn't matter if you think of conditions/cases or their negated versions, but after revising

// your code don't include empty condition blocks with no line of code inside. Instead, just flip

// the condition to have if (max >= 0), and no else.

} else {

...

}

**8.3 Factoring**

If there are code that looks very similar or is identical, there should be a way to factor it out and prevent multiple identical lines.

Some methods to remove redundant code:

* Private helper methods: Repeated or similar behavior code can be factored out into a private method that can be used in multiple places. By having methods that are well-named, the behavior of the code will be much more understandable.
* Factor similar behavior out of if/else statements: If there are lines of code that happen in both the if and the else statement, factor the code out of the if/else statement so there is one line of the code rather than two in both blocks.

**8.4 Loop Zen**

Clean loops will produce efficient loops. One unnecessary expression between a loop will cause the program to calculate that unnecessary expression multiple times. This means that with each calculation should be consider if completely necessary.

Example:

----------------------------------------------------------------

for (int i = 0; i < 4; i++) {

if (i != 3) {

System.out.println("working hard!");

} else {

System.out.println("hardly working!");

}

}

----------------------------------------------------------------

The if statement is unnecessary because “hardly working” is only printed at the end. This loop would cause the program to check the expression (I != 3) each loop making this if statement actually making the program last efficient. Although it isn’t a very big difference, visually, the code is also harder to interpret than if “hardly working” was printed outside of the loop:

----------------------------------------------------------------

for (int i = 0; i < 4; i++) {

System.out.println("working hard!");

}

System.out.println("hardly working!");

----------------------------------------------------------------

This is much clearer to see than the first example.

1. **Miscellaneous**

Other things to consider while writing code.

**9.1 Scope**

The scope of variables should always be considered when declaring them. If the variable is not used outside of a function/method, then it should be declared inside the method. This also applies to loops. A variable that is only used inside a loop should be declared inside the loop so that it is discarded when the loop is done.

Class fields or module variables should only be declared when the variable is used throughout the whole class or module.

**9.2 Convenience methods/functions**

Convenience methods have been created for a reason. No need to reinvent the wheel. In the case of java ArrayLists, rather than checking if arrayList.size() == 0, there is the arrayList.isEmpty() method that could be used instead. This applies to other languages too. Check if there is a method that covers the functionality already before implementing or using alternatives. This is something that cannot be wholly checked but is very helpful to keep in mind while coding.

**Section 2: Language specific style guides**

(to be written)