**Template of the Coding Convention**

1. File names **<Assigned to Travis>**

<*Define commonly used file suffixes and names. For example, if the programming language is Java, the common suffixes are .java and .class. Frequently used file names are README*, *GNUmakefile*, *etc.*>

1. File organization **<Assigned to Travis>**

<*A file consists of sections that should be separated by blank lines and an optional comment identifying each section. Files longer than 2000 lines are cumbersome and should be avoided. For example, Java source files have the following ordering:*

1. *Beginning comments*
2. *Package and Import statements*
3. *Class and interface declarations*>
4. Indentation **<Assigned to Travis>**

<*Use either spaces or tabs as indentation unit to format the source code. The coding convention should clearly define the makeup of indentation in terms of number of spaces or tabs.*>

1. Comments **<Assigned to Aaron>**

<*Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. If Java is the programming language, Doc comments should be defined. Doc comments are meant to describe the specification of the code, from an implementation-free perspective, to be read by developers who might not necessarily have the source code at hand.* >

Comments should contain information or notes about the code that will follow the comment(s). The comments themselves should be separated using the /\* (to begin) \*/ (to end) convention so that the comments form a block prior to the code the comment relates to. The comments should be a short description of what the code is supposed to achieve and how/why that method is used. In Java, the /\*\* \*/ syntax can be used to automatically write the comments to documentation if we choose to utilize the JDK javadoc tool. If the comment is simply describing a single declaration, then the comment could be a few words after the declaration using the // syntax.

1. Declarations **<Assigned to Aaron>**

<*Define how many declarations can be put in one line (e.g., one declaration per line to make comments easier), how to initialize the variables (e.g., better at the time of declaration), the best place to put the declarations (e.g., at the beginning of the blocks), and class and/or interface declarations.*>

Declaration will follow these guidelines:

* One variable declaration per line
* Initilization of variables will occur at the time of declaration
* Declarations/initilizations will be at the beginning of the area where they are used

1. Statements **<Assigned to Aaron>**

<*Define the formats of simple statements, compound statements, if-then-else statements, loop statements, try-catch statements, etc.*>

The format of statements will be as follows:

* Simple statements will only have one statement per line
* Compound statements will be formatted using a curly brace on its own line with the statements following as one statement per line within the curly braces, and a curly brace on its own line ending the statement.
* The rest of the statements (if-then-else, try-catch, loop) will follow the same guidelines (i.e. if(condition) followed by the same format of the compound statements regarding curly braces beginning and ending the statements.

1. White space **<Assigned to Owen>**

<Define how to use blank lines and blank spaces to improve readability.>

A single space should appear after a method declaration and before the opening brace of the method to help provide visual separation of the syntatic units of method definitions. The closing brace of method definitions should appear on a separate line and not immediately following the last statement of the method definition. All code files will use the Kernighan Ritchie brace style. Some would argue that the Allmann brace style is easier to read. The Allmann brace style lacks compactness as the opening brace appears on its own line. The Kernighan Ritchie brace style produces compact code. Readers can view more statements in one screen. Using this style it is just as easy to match the closing brace to the loop statement or method declaration it belongs to by looking for the first character of a word at the same indentation level as the closing brace. If one suspects they might be missing an opening brace they can simply look at each method declaration line and loop declaration line to see if they have opening braces.

A blank line should be used to separate the major sections of a code file. In Java class files a blank line should separate the beginning comments from the package statements, and the package statements from the import statements.

A blank line should be used to separate the closing brace of one method definition from the comments of the next method definition. A blank line should be used to separate the last method definition from any other members within the enclosing scope.

No tab character should be used to indent statements, directives, or declarations at the global file level. All methods and fields of a class should be indented one level more than the enclosing class declaration. All methods of interface declarations should be indented one level more than the interface declaration.

All simple statements, compound statements, and nested methods of a method should be indented one level further than the enclosing method declaration.

In multiple selection statements (switch statements in Java) each case statement should be indented with a tab character to make the structure of the multiple selection statements more easier to read. It will be easier to identify at a glance which case option values belong to which multiple selection statements.

For multiple line comments which use the “/\* comment \*/” syntax, the text of the comment shall start on the same line as the opening “/\*” token. The “/\*” token should be followed by a single space. Next the text of the comment should appear. The end of the text of the comment should immediately be followed by the closing “\*/” token with no blank line following the end of the text. Formatting comments in this way will conserve space helping to prevent readers of the code from having to continually scroll though the file to learn how the different methods work together. This format places the method declaration on the line following the end of the text of the comment. This allows the reader’s eyes to move directly from the comment to the method declaration. In this way it is very easy to identify which method belongs to which comment at a glance.

The end of each statement in a code file should be followed by a new line character. Some statements, such as for statements should be split into several lines. In a for statement the for statement header with the opening brace fill appear on one line, each statement in the loop body should appear on its own line, and the closing brace should appear on its own line. Other types of loops, including while loops, should be formatted in the same way.

Nested classes and methods should be preceded by one blank line and followed by one blank line to make them easier to identify.

Each logically related group of statements in a method should be preceded by a blank line and followed by a blank line. This helps to clarify the purpose of each statement.

Most operators should be preceded by one blank space and followed by one blank space to make operations easier to identify visually. This rule should be applied for all occurences of the assignment and equality operators. There may be some situations where this should not be done so that operands are visually grouped with their operator.

A new line character should be used after the length of a line of code has reached 100 characters. Some readers of code files may have multiple windows open as they compare code files. It would be inconvenient for readers to have to scroll horizontally in a code file to read length lines of code. Placing such a limit also helps to logically separate the syntactic units of length statements. If the typing of a syntactic unit increases the length of the line in which it appears to over 100 characters, the entire syntactic unit should be moved to the next line provided that this does not decrease readability.

Each element of javadoc comments should begin on a new line. For example, each “@param” element and each @return element should begin on a new line.

After all closing braces a tab character should be entered followed by a comment indicating which method, block, or statement the closing brace applies to.

A blank space should follow each comma which follows a formal parameter or argument. Any other comma separated lists should follow this format to improve the visual separation of the elements of such lists.

No space should appear after a method name and before the opening parenthesis token in both method invocations and method declarations.

Array definitions which span multiple lines should be formatted so that the first character of the first array element of a line of code appears directly below the first character of the first array element appearing on the above line of code. This rule does not apply to the line containing the first element of an array.

1. Naming conventions **<Assigned to Owen>**

<*Naming conventions make programs more understandable by making them easier to read. Rules for naming packages, classes, interfaces, methods, variables, constants, etc., should be defined.*>

All variable, method, constant, interface, class, and package names should be unambiguous. All names should be long enough so that each line of code can be understood, such that minimal references are made to other sections of the code to understand the meaning.

The first letter of each word of a class or interface name should be capitalized. All other letters of class and interface names will be lowercase. Class and interface names should not contain spaces or underscore characters. All words of class and interface names should appear as one word.

Variable and method names in Java and PHP files should be in mixed case (or camel case) form. The first letter of the first word of a variable or method name should be lowercase. All subsequent words of variable and method names should begin with a capital letter. Variable and method names should not contain underscore characters.

PHP variables should begin with a $. For a variable which stores a volunteer name the variable should be declared as $volunteerName and should be assigned an initial value.

Package names are sequences of identifiers separated by dots. Package names should use only lowercase letters. Names given to packages should describe the kind of classes contained within. All classes of a package should be related. A reversed domain name, fictional for our system, should appear at the beginning of a package name. In our system all package names should begin with “net.softwareengineering.teamquartz.” without the quotation marks. There will be one package for each subsystem. The domain name prefix of the package name should be followed by the name of a subsystem. For example, the package for the event management subsystem can be referenced as “net.softwareengineering.teamquartz.eventmanagement”. A class of a subsystem can be referenced by adding a dot and the class name after the containing subsystem name in a package reference. For example, the “Message” class of the notification subsystem can be referenced as “net.softwareengineering.teamquartz.notfication.Message” in an import statement.

Named constants should appear in uppercase letters. The words of a named constant name should be separated by a single underscore character. Named constant names should be long enough to fully describe the value which it describes, disallowing ambiguity.

Subdirectories which contain class files must be named according to the package name corresponding to the classes of the class files. A subdirectory will exist for each package. Each subdirectory contains the class files for all the classes of a single package.

1. Programming practices **<Assigned to Owen>**

<*Define how to reasonably use “public” access, constants, variable assignments, etc.*)

The private visibility specifier should be used in the declaration of all fields other than public static final fields. When a subclass needs to access a private field of its superclass it should call a protected method to access it. A public method should be used to allow non-subclasses access to a private instance field of the class containing the public method.

All local variables shall be assigned values when they are declared to prevent garbage data from causing failures when the value of each variable is referenced.

Object oriented programs rarely contain static methods and fields. The use of static methods and fields should be minimized. Instance fields cannot be accessed by static methods.

In a class the public interface should appear above the instance fields.

Magic numbers or numeric literals should never be used outside of named constant declaration statements. For all other statements named constants should appear in place of integer literals. If a named constant is only needed in a single method the named constant should be declared in the method that uses it. If two or more methods use the named constant it should be placed in the class scope and should be specified as a static final field.

Array variable declarations should use the syntax array\_type[] array\_variable\_name as opposed to placing the square brackets after the array variable name. Someone quickly reading the type of each variable is more likely to notice that the variable is an array variable.

A variable should be declared as static if all instances of a class should use the same variable.

Methods which contain code that may generate a checked exception should use a throws clause to list the checked exception or a try block should be used to catch them. There should be a catch block corresponding to every try block. Each catch block should contain a variable of a checked exception class such as IOException or ClassNotFoundException.

Methods should be declared as public if any method in any class should be able to call them. A method should be declared as protected when a method of a subclass of the class containing the protected method needs to access the protected method. Methods should be declared private if only other methods of the class in which they are declared should be able to invoke them.

When a subclass method invokes the corresponding superclass method, the call to the superclass method should be the first statement of the subclass method.