**coding standardS AND GUIDELINES**

**Introduction**

This document contains coding guidelines for developing consistent and uniform code among software engineers on a given project. The purpose of this document is to provide software engineers with a standard to facilitate code reviews and analysis. These standards and guidelines are intended to be language independent and as an initial starting point for a project-defined coding standard. This document may be modified to suit the needs of a particular project, but any revisions must be signed off on by the course instructor and the updated copy of this document must be available on the team’s Trac site.

**Coding Standards**

The following sections are coding standards. These sections express rules that must be followed.

**Headers**

All source files should contain a standard header, defined by the project’s software lead. Source headers must contain the following minimum information:

1) The name of the project.

2) The company for which the program is being developed.

3) The name of the original author of the source code.

4) The creation date for the original file.

5) A description of the code.

6) The status of code reviews.

A sample header is shown below:

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\*

\* ROCKET IMPROVEMENT PROGRAM (RIP)

\*

\* Yoyodyne Inc.

\*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\*

\* Name: RIP\_TYPES\_PS.PKG

\* Creation Date: 2/23/2008

\* Author: M. Bopp

\*

\* Description: This package contains type definitions for

\* use in the RIP software. These types are to be

\* used instead of the predefined INTEGER and

\* FLOAT types to preserve precision.

\*

\*

\* Code Review: Code reviewed 3/25/2008 M. Bopp, J. Able, M. Smith

\*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

**Indentation, Spacing, and Alignment**

Standard horizontal indentation interval is three spaces. Tabs are not used for portions of the indentation interval. Spaces, blank lines, and indentation can be used effectively along with comments to facilitate readability:

1) Precede and follow single and compound delimiters by a space   
Example: if ( a < -b )

2) Precede and follow binary operators by a space.  
Example: a + b

3) Precede unary operators by a space.   
Example: if ( !isValid )

4) In general, begin each statement on a separate line, not just separated by spaces.   
Example:

Bad Example: Good Example:

int AddCounter() int AddCounter()

{ {

numberOfItems++; location -= 1; numberOfItems++;

location -= 1;

return location;

} return location;

}

5) Use extra spaces, tabs, and blank lines to align closely related statements and declarations in adjacent lines.   
Example:

int numberOfItems = 10; // variable names are all aligned

int numberOfErrors = 5; // variable values are also aligned

double errorPrecision = 0.005; // comments are aligned as well

6) Indent blocks of one level from the delimiters (e.g. {}, begin-end, if-endif) with each sub block indented another level.   
Example:

procedure Build\_Queue is void build\_queue()

begin {

LoadRecords; LoadRecords();

InitializePtrs; InitializePointers();

exception if ( errors )

when QUEUE\_ERROR => {

InitializeQueue(); InitializeQueue;

. . . . . .

end Build\_Queue; }

}

7) When a statement or declaration requires multiple program lines, attempt to break the statement or declaration into smaller pieces and spread it across multiple lines. A line is considered to be 100 characters for the purposes of this document.  
Example:

Bad Example:

finaleResult = ( variableA + variableB ) / (variableC – variableE )\* variableF;

Good Example:

temporaryA = variableA + variableB;

temporaryB = variableC – variableE;

finalResult = temporaryA / temporaryB \* variableF;

Good Example:

finaleResult = ( variableA + variableB ) /

( variableC – variableE ) \*

variableF;

**Naming Conventions and Capitalization**

Selection of names and consistency of naming conventions for data structures, packages, classes, and functions can either make programs easy to interpret or very difficult to follow. Use of precise names helps the reader see the intent of the code and minimizes the need for extensive comments. Other guidelines, such as capitalization and scope, make identification and determining use of data/constants/functions easier.

1) Enumeration literals, types, constants, macro names and named numbers are entirely in upper–case letters. Underscores may be used between words to improve readability. (e.g. VALUE\_OF\_PI, VALUE\_OF\_NUMBER\_E, MAXIMUMPROCESSES)

2) Class, Function, Procedure, or Method names are written with the initial letter of each word capitalized and the rest of the word lower case. (e.g. GetNumberOfErrors, SetCursorPosition)

3) Data elements/variable names are written with the initial letter of each word capitalized and the rest of the word lower case except for the first word which is all lower case. (e.g. numberOfErrors, cursorPosition)

4) Common acronyms may be used in identifiers. If they are used, they may be written in all capitals, regardless of the guidelines presented above. (e.g. numberOfDSP, valueOfHNF)

5) Choose identifiers that are meaningful and “read well” when used in context. Use verb-object for procedures (e.g., RunBit, HaltProcessor). Use object names for data. (e.g., numberOfErrors, failedProcessor)

6) Consider scope when declaring data. Don’t allow visibility to code not requiring access. Provide access routines (i.e., set/get…) for data that needs to be visible outside of the package/class in which it is defined. Example:

public void setNumberOfItems(int number)  
 {  
 numberOfItems = number;  
 }  
  
 public int getNumberOfItems()  
 {  
 return numberOfItems;  
 }

7) Do not use the same name for more than one data element, especially if scopes overlap. Data local to a small block of code (e.g., simple loop counts like i, j, cnt) or identical data element names within different structures or classes are fine.

**General Practices**

There are several general conventions that the programmer should follow when coding program units. These guidelines are as follows:

1) Do not use variables of anonymous or implicit types.   
Eample:

Bad Example:   
richard = new {ID = 1, firstName = "Richard"};   
  
Good Example:  
Richard = new PersonnelMember(1, "Richard");

2) Use named numbers instead of numeric literals. (e.g., VALUEOFPI for 3.14159)

3) Conceal any information irrelevant to the user of a package through the use of private and limited types. (i.e. class variables should be declared as private)

4) Handle exceptions at the lowest possible level at which the program can properly respond to the error. (i.e. exceptions should be caught and handled as soon as possible)

5) Subprograms (functions and procedures) should perform a single logical function. (i.e. use two different functions to calculate the mean and standard deviation instead of having one function calculate both)

6) Avoid overloading operators or functions if possible. Use of the overloading feature is permissible when the only difference is the type of parameters a subprogram takes; for example, when doing the same mathematical operation on two different numeric types.

Example:

public int calculateValueOfHNF(int x, int y)

{

...

}

public int calculateValueOfHNF(double x, double y)

{

...

}

7) Place all include, import, or similar statements together immediately after the header.  
Example:

/\* ...   
\* end of header

\*/

import java.util.\*;

import java.swing.\*;

...

8) Declare all variables at the beginning of blocks and one per line.  
Example:

if(valueOfNHF > MAXIMUM\_ERROR\_TOLERENCE)

{

int oldValueOfNHF = valueOfNHF;

int valueOfErrorCorrection = valueOFNHF – MAXIMUM\_ERROR\_TOLERENCE;

...

this.adjustValueOfNHF();

...

}

9) Initialize variables where they are declared unless they are otherwise dependent on some computation.

**Comments**

Comments are used to enhance the readability and understandability of a program by describing the program in a higher level or theoretical manner. It is neither necessary, nor desirable to duplicate information in the code unless the code construct is potentially confusing or uses a seldom used feature of the language. In general, comments should make code easily understandable for those who are not familiar with it.

1) Each type and object name declaration should be accompanied by a brief comment if the identifier is not self-explanatory.   
Example:

double valueOfHNF; //This is the value of the Hiland-Nomsky function.

2) Comment subroutines and functions where they are specified and where they are defined. Be sure to explain any exception raised and not handled by the subprogram. Include sufficient information so that a user of the package need not refer to the implementation.  
Example:

// Function to determine altitude.

// Function throws exceptions generated from non-responsive hardware.

// Any module implementing this function must handle those exceptions.

3) When using debug statements to test code, comment out the debug statements instead of deleting them to simplify future software maintenance. Ensure that additional comments are added to specify that the code is debug code.   
Example:

. . .  
 // System.out.println(errorVal); // Debug code to test while loop  
 . . .

4) Interject comments into the code that briefly describe the intent of the few lines of code that follow. Assembly and other low level code will require more comments than high-level code, possibly a comment on every instruction line.   
Example:

// This section of code calculates the speed of the ball object.  
 // This value is later used when determining impact force.

5) Indent comments such that they are on the same level as the code they reference.  
Example:

// The comment should be indented to match the code  
 int totalErrors;  
 . . .  
 // This section of code calculates the total errors.  
 totalErrors = spellingErrors + grammarErrors;

6) Consider the use of doc code if it is available.

**Coding Guidelines**

The following sections include guidelines that should be followed, but contain good advice rather than hard rules.

**Security**

Security is an important issue to keep in mind while writing code.

1) Filter user input to prevent against code injection attacks. If possible, use parameterized statements or other methods of filtering to prevent SQL and other code injection attacks.

2) Check for and prevent buffer overruns if the programming language does not offer built in memory protection.

3) Include user authentication when dealing with sensitive or company data. Take appropriate measures to ensure that only users who should have access to a system are able to access it.

**UI Guidelines**

The user interface design is just as important as the underlying code that drives the application. In general, interface design should attempt to conform to the eight golden rules of interface design.

1) Use a consistent design and commands when developing the user interface.

2) Include shortcuts and options for frequent or power users to increase the pace of user interaction.

3) The interface should provide informative feedback for users in response to their actions.

4) Sequences of actions performed by users should be grouped and designed to provide closure to a user upon completing those actions.

5) Design the system to protect users from committing critical errors and include mechanisms for handling any errors that may occur.

6) Allow users to easily reverse their actions in the event that they make a mistake.

7) Design the user interface so that the user feels in control of the system.

8) Reduce the short-term memory load on users by keeping the design simple and using common menu options and shortcuts.

**Complexity**

Modularize programs as much as possible. The McCabe complexity metric (National Institute of Standards and Technology special publication 500–99) can be used as a guideline for measuring the effectiveness of modularization (with case statements allocated a complexity of 1 plus the complexity of the most complex branch, regardless of the number of cases, as suggested by NBS 500–99). A rule of thumb is to keep the complexity of any given unit under ten.

Also, perform refactoring any function containing over 200 lines of code into multiple sub-functions. This improves the readability and testability of the code as long functions tend to be complex.

**Coupling AND Cohesion**

Code developed should be highly cohesive and loosely coupled. These properties make code more readable and can reduce the amount of time spent locating and fixing defects in the code.

Cohesion is the measure of how well parts in a module fit together. For example, the methods in the java Math class are all used to calculate mathematical functions. If a module contains many parts that are not related to each other, it may worthwhile to use code refactoring techniques to create multiple new modules.

Coupling is the degree of interaction between different modules in a system. Generally, encapsulation should be used to prevent modules from reaching into each other. Limiting the use of global data and keeping modules from being dependent on another’s implementation details will help achieve loose coupling.