

## **Java Development Guidelines**

James Yao and Andrew Okimi  
May 30, 2006

---

## Table of Contents

---

|                           |    |
|---------------------------|----|
| Documentation .....       | 3  |
| Class Implementation..... | 5  |
| General Coding .....      | 6  |
| Implementation Model..... | 8  |
| Appendix A .....          | 27 |
| Naming operations .....   | 27 |
| Appendix B .....          | 29 |
| Glossary .....            | 29 |
| Appendix C .....          | 36 |
| Java Reference list ..... | 36 |

## Documentation

### Standard

**[Package]** Have a "Pack" Class or HTML for each Package. Put documentation about the Package and any functionality that applies to the package "as a whole" into the Pack.

**[Document on interface]** Use Interfaces for Public Documentation. A good example is that the component specifications standard can simply walkthrough all interfaces contributing to this component specification. Later the user of this component can choose any interesting interface documentation to read for more detailed information.

**[Comment metric]** We expect 50 per cent or 70 per cent of code to be comments. Use the free comment counter named "commentcounter" to enhance this standard. (\*Unless you write it on paper and do it on review, few developers will follow it strictly.)

"Another programmer should be able to look at your member function and be able to fully understand **what** it does, **why** it does it, and **how** it does it in less than 30 seconds. If he or she can't then your code is too difficult to maintain and should be improved. Thirty seconds, that's it." [Amby]

---

### Guidelines

**[Self-document]** Try to make code as self-documented as possible before resorting to commenting it. This can both improve the design and better describe an existing design. A good practice is to write the comment and process first, then write the code for it.

**[Comment on method and argument, not the code]** Provide comments that augment, not repeat, program code. [ChiMu] So, the comment will explain "why". For those who complain about not enough materials to put on the comment, here are some hints:

- What and why the member function does what it does?
- What a member function must be passed as parameters?
- What a member function returns?
- Known bugs;
- Any exceptions that a member function throws out;

---

*Continued on next page*

---

## Documentation, Continued

---

### **Guidelines** (continued)

- How a member function changes the object?
- Examples of how to invoke the member function if appropriate.
- Applicable preconditions and post conditions. [Amby]

**[Good document styles]** We encourage that good coding styles be followed.  
E.g. Specify a standard keyword order; augment javadoc keywords; etc.

---

---

## Class Implementation

---

**Standard** [No public variable] Never declare instance variables as public.

---

**Guidelines** [Readability] Focus on readability over special language skills. Considering bad style for those styles, e.g., take advantage of “clean” language features. E.g., Prefer **i=i+1** than **i++** ; forbidden **while (i++>10)** or **while ((i=a+b)>0)** ... [ChiMu]

---

---

## General Coding

---

### Guidelines

**[Naming]** Standardize naming patterns and choose intention revealing operation names. When naming operations, always create a name that suggests what the operation "provides for the caller", not how a method could accomplish this service. See Appendix A on naming operations. There are plenty of guidelines for a good name. But one point to highlight here is that a long name is always preferred to a short one as long as this long name provides accurate clear information for the reader.

**[Testing class]** We recommend using **ZTestCLASSNAME** for unit test class. This class is put into the same package. It provides a good tradeoff for documentation, testability, sample of usage, and force-to-test principles over the delivery concern. Document the test harness member functions. **Note:** The documentation should include a description of the test as well as the expected results of the test.

**[Test first]** Write the test code first (before you write the class) in order to verify that your class design is complete. If you can't write test code, you don't know what your class looks like. In addition, the act of writing the test code will often flush out additional features or constraints that you need in the class—these features or constraints don't always appear during analysis and design. Tests also provide example code showing how your class can be used.

---

### Some basic guidelines

The following is a list of guidelines for coding in Java:

- Minimize statics
- Minimize reliance on implicit initializers
- Prefer abstract methods to those with default no-op implementations
- Avoid giving a variable the same name as one in a superclass
- Use final and/or comment conventions for instance variables
- Avoid unnecessary instance variable access and update methods
- Minimize direct internal access to instance variables inside methods
- Ensure that non-private statics have sensible values
- Consider whether any class should implement Cloneable and/or Serializable.

---

*Continued on next page*

---

## General Coding, Continued

---

**Some basic guidelines**  
(continued)

- Whenever reasonable, define a default (no-argument) constructor
  - Generally prefer “long” to “int”, and “double” to “float”
  - Use method equals instead of operator “=” when comparing objects
  - Prefer declaring arrays as **Type[] arrayName** rather than **Type arrayName[]**
  - Minimize \* forms of import When sensible, consider writing a main for the principal class
  - The class with main should be separate from those containing normal classes
  - Always Initialize Static Fields
  - Always use accessor
  - First make it work, then make it fast
  - Make execution structure obvious
  - Avoid overloading methods on argument type
-

## Implementation Model

### Naming conventions

The following table lists naming conventions.

**Note:** Any violation to the specification is allowed if it enhances readability.

| Recommendation  | Rationale  | Examples                                 |
|---|--|--|
| <b>General Naming Conventions</b>   |  |  |
| Names representing packages should be in all lower case   | This is the package naming convention used by Sun for the Java core packages. The initial package name representing the domain name must be in lower case.   | mypackage,<br>com.company.application.ui |
| Names representing types must be nouns and written in mixed case starting with upper case.              | This is common practice in the Java development community, and also the type naming convention used by Sun for the Java core packages.   | Line, AudioSystem                        |
| Variable names must be in mixed case starting with lower case.  | This is common practice in the Java development community, and also the naming convention for variables used by Sun for the Java core packages. Makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration <code>Line line</code>   | line, audioSystem                        |
| Names representing constants (final variables) must be all uppercase using underscore to separate words | Common practice in the Java development community and also the naming convention used by Sun for the Java core packages. In general, the use of such constants should be minimized. In many cases implementing the value as a method is a better choice:<br><code>int getMaxIterations()</code><br><br><code>// NOT: MAX_ITERATIONS = 25</code><br><code>{</code><br><code>  return 25;</code><br><code>}</code><br>This form is both easier to read, and it ensures a uniform interface towards class values. | MAX_ITERATIONS,<br>COLOR_RED             |
| Names representing methods must be verbs and written in mixed case starting with lower case.            | Common practice in the Java development community and also the naming convention used by Sun for the Java core packages. This is identical to variable names, but methods in Java are already distinguishable from variables by their specific form.   | getName(),<br>computeTotalWidth()        |

*Continued on next page*

## Implementation Model, Continued

### Naming conventions (continued)

| Recommendation  | Rationale  | Examples   |
|---|--|--|
| Abbreviations and acronyms should not be uppercase when used as name. | Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type would have to be named dVD, hTML etc. which obviously is not very readable. Another problem is illustrated in the examples above; When the name is connected to another, the readability is seriously reduced; The word following the acronym does not stand out as it should.  | <pre>exportHtmlSource(); // NOT: exportHTMLSource(); openDvdPlayer(); // NOT: openDVDPlayer();</pre>   |
| Private class variables should have underscore (_) suffix.            | <p>Apart from its name and its type, the scope of a variable is its most important feature. Indicating class scope by using underscore makes it easy to distinguish class variables from local scratch variables. This is important because class variables are considered to have higher significance than method variables, and should be treated with special care by the programmer.</p> <p>A side effect of the underscore naming convention is that it nicely resolves the problem of finding reasonable variable names for setter methods:</p> <pre>void setName (String name) {   name_ = name; }</pre> <p>An issue is whether the underscore should be added as a prefix or as a suffix. Both practices are commonly used, but the latter is recommended because it seem to best preserve the readability of the name.</p> <p>It should be noted that scope identification in variables have been a controversial issue for quite some time. It seems, though, that this practice now is gaining acceptance and that it is becoming more and more common as a convention in the professional development community.</p> | <pre>class Person {   private String name_;   ... }</pre>  |
| Generic variables should have the same name as their type.            | <p>Reduce complexity by reducing the number of terms and names used. Also makes it easy to deduce the type given a variable name only. If for some reason this convention doesn't seem to fit it is a strong indication that the type name is badly chosen.</p> <p>Non-generic variables have a role. These variables can often be named by combining role and type:</p> <pre>Point startingPoint, centerPoint; Name loginName;</pre>  | <pre>void setTopic (Topic topic) // // NOT: void setTopic (Topic // NOT: void setTopic (Topic // NOT: void setTopic (Topic x) value) aTopic) (Topic x)  void connect (Database database) // NOT: void connect (Database db) // NOT: void connect (Database oracleDB)</pre> |

*Continued on next page*

## Implementation Model, Continued

### Naming conventions (continued)

| Recommendation  | Rationale   | Examples   |
|---|---|--|
| All names should be written in English.   | English is the preferred language for international development.  | fileName; // NOT: filNavn  |
| Variables with a large scope should have long names; variables with a small scope can have short names. | Scratch variables used for temporary storage or indices are best kept short. A programmer reading such variables should be able to assume that its value is not used outside a few lines of code. Common scratch variables for integers are i, j, k, m, n and for characters c and d.   |  |
| The name of the object is implicit, and should be avoided in a method name.                             | The latter might seem natural in the class declaration, but proves superfluous in use, as shown in the example.   | line.getLength();<br><br>// NOT: line.getLength();   |
| <b>Specific Naming Conventions</b>  |   |  |
| The terms get/set must be used where an attribute is accessed directly                                  | This is the naming convention for accessor methods used by Sun for the Java core packages. When writing Java beans this convention is actually enforced.  | employee.getName();<br>matrix.getElement (2, 4);<br>employee.setName (name);<br>matrix.setElement (2, 4, value); |
| <i>is</i> prefix should be used for boolean variables and methods.                                      | This is the naming convention for Boolean methods and variables used by Sun for the Java core packages. When writing Java beans this convention is actually enforced for methods. Using the <i>is</i> prefix solves a common problem of choosing bad Boolean names like status or flag. <i>isStatus</i> or <i>isFlag</i> simply doesn't fit, and the programmer is forced to chose more meaningful names. There are a few alternatives to the <i>is</i> prefix that fits better in some situations. These are <i>has</i> , <i>can</i> and <i>should</i> prefixes:<br>boolean hasLicense();<br>boolean canEvaluate();<br>boolean shouldAbort = false | isSet, isVisible, isFinished,<br>isFound, isOpen   |
| The term compute can be used in methods where something is computed.                                    | Give the reader the immediate clue that this is a potential time consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability.  | valueSet.computeAverage();<br>matrix.computeInverse();   |
| The term find can be used in methods where something is looked up.                                      | Give the reader the immediate clue that this is a simple look up method with a minimum of computations involved. Consistent use of the term enhances readability.   | vertex.findNearestVertex();<br>matrix.findMinElement();  |

*Continued on next page*

## Implementation Model, Continued

### Naming conventions (continued)

| Recommendation   | Rationale  | Examples   |
|--|--|--|
| The term initialize can be used where an object or a concept is established. | The American initialize should be preferred over the English initialise. Abbreviation init must be avoided.  | <code>printer.initializeFontSet();</code>  |
| JFC (Java Swing) variables should be suffixed by the element type.           | Enhances readability since the name gives the user an immediate clue of the type of the variable and thereby the available resources of the object   | <code>. widthScale, nameTextField, leftScrollbar, mainPanel, fileToggle, minLabel, printerDialog</code>            |
| Plural form should be used on names representing a collection of objects.    | Enhances readability since the name gives the user an immediate clue of the type of the variable and the operations that can be performed on the object.   | <code>Collection points; // of Point<br/>int[] values;</code>  |
| n prefix should be used for variables representing a number of objects.      | The notation is taken from mathematics where it is an established convention for indicating a number of objects.<br>Note that Sun use num prefix in the core Java packages for such variables. This is probably meant as an abbreviation of number of, but as it looks more like number it makes the variable name strange and misleading. If "number of" is the preferred statement, number Of prefix can be used instead of just n. num prefix must not be used. | <code>nPoints, nLines</code>   |
| No suffix should be used for variables representing an entity number.        | The notation is taken from mathematics where it is an established convention for indicating an entity number.<br>An elegant alternative is to prefix such variables with an i: iTable, iEmployee. This effectively makes them named iterators.   | <code>tableNo, employeeNo</code>   |
| Iterator variables should be called i, j, k etc.                             | The notation is taken from mathematics where it is an established convention for indicating iterators. Variables named j, k etc. should be used for nested loops only.   | <pre>while (Iterator i = points.iterator(); i.hasNext(); ) { : }  for (int i = 0; i &lt; nTables; i++) { : }</pre> |

*Continued on next page*

## Implementation Model, Continued

### Naming conventions (continued)

| Recommendation   | Rationale   | Examples   |
|--|---|--|
| Complement names must be used for complement entities [1]. | Reduce complexity by symmetry.  | get/set, add/remove, create/destroy, start/stop, insert/delete, increment/decrement, old/new, begin/end, first/last, up/down, min/max, next/previous, old/new, open/close, show/hide |
| Abbreviations in names should be avoided.                  | <p>There are two types of words to consider. First are the common words listed in a language dictionary. These must never be abbreviated. Never write:</p> <p>cmd instead of command<br/>         comp instead of compute<br/>         cp instead of copy<br/>         e instead of exception<br/>         init instead of initialize<br/>         pt instead of point<br/>         etc.</p> <p>Then there are domain specific phrases that are more naturally known through their acronym or abbreviations. These phrases should be kept abbreviated. Never write:</p> <p>HypertextMarkupLanguage instead of html<br/>         CentralProcessingUnit instead of cpu<br/>         PriceEarningRatio instead of pe<br/>         etc.</p> | <pre>computeAverage(); // NOT: compAvg(); ActionEvent event; // NOT: ActionEvent e;</pre>  |
| Negated boolean variable names must be avoided.            | The problem arise when the logical not operator is used and double negative arises. It is not immediately apparent what !isNotError means.  | <pre>boolean isError; // NOT: isNotError boolean isFound; // NOT: isNotFound</pre>   |

*Continued on next page*

## Implementation Model, Continued

### Naming conventions (continued)

| Recommendation   | Rationale   | Examples   |
|--|---|--|
| Associated constants (final variables) should be prefixed by a common type name.   | This indicates that the constants belong together, and what concept the constants represents. An alternative to this approach is to put the constants inside an interface effectively prefixing their names with the name of the interface:<br><pre>interface Color {     final int RED = 1;     final int GREEN = 2;     final int BLUE = 3; }</pre> | <pre>final int COLOR_RED = 1; final int COLOR_GREEN = 2; final int COLOR_BLUE = 3;</pre> |
| Exception classes should be suffixed with Exception.   | Exception classes are really not part of the main design of the program, and naming them like this makes them stand out relative to the other classes. This standard is followed by Sun in the basic Java library.  | <pre>class AccessException {     : }</pre>   |
| Default interface implementations can be prefixed by Default.  | It is not uncommon to create a simplistic class implementation of an interface providing default behaviour to the interface methods. The convention of prefixing these classes by Default has been adopted by Sun for the Java library.   | <pre>class DefaultTableCellRenderer implements TableCellRenderer {     : }</pre>         |
| Functions (methods returning an object) should be named after what they return and procedures (void methods) after what they do. | Increases readability. Makes it clear what the unit should do and especially all the things it is not supposed to do. This again makes it easier to keep the code clean of side effects.  | N/A  |

**Files** The following table provides conventions to be used when working with Java files.

| Recommendation  | Rationale                   | Example    |
|---|-----------------------------|------------|
| Java source files should have the extension <code>.java</code> .  | Enforced by the Java tools. | Point.java |
| Classes should be declared in individual files with the file name matching the class name. Secondary private classes can be declared as inner classes and reside in the file of the class they belong to. | Enforced by the Java tools. |            |

*Continued on next page*

## Implementation Model, Continued

### Files (continued)

| Recommendation  | Rationale  | Example  |
|---|--|--|
| File content must be kept within 80 columns.                | 80 columns is the common dimension for editors, terminal emulators, printers and debuggers, and files that are shared between several developers should keep within these constraints. It improves readability when unintentional line breaks are avoided when passing a file between programmers.   |  |
| Special characters like TAB and page break must be avoided. | These characters are bound to cause problems for editors, printers, terminal emulators, or debuggers when used in a multi-programmer, multi-platform environment.  |  |
| The incompleteness of split lines must be made obvious.     | <p>Split lines occur when a statement exceeds the 80-column limit given above. It is difficult to give rigid rules for how lines should be split, but the examples should give a general hint. In general:</p> <ul style="list-style-type: none"> <li>• Break after a comma.</li> <li>• Break after an operator.</li> </ul> <p>Align the new line with the beginning of the expression on the previous line.</p> | <pre>totalSum = a + b + c +           d + e;  function (param1, param2,           param3);  setText ("Long line split" +          "into two parts.");  for (tableNo = 0; tableNo &lt; maxTable;      tableNo += tableStep)</pre> |

*Continued on next page*

## Implementation Model, Continued

**Statements** The following table provides conventions to be used when working with Java statements.

| Recommendation   | Rationale   | Example  |
|--|---|--|
| <b>Package and Import Statements</b>   |   |  |
| The package statement must be the first statement of the file. All files should belong to a specific package.  | The package statement location is enforced by the Java language. Letting all files belong to an actual (rather than the Java default) package enforces Java language object oriented programming techniques.  |  |
| The import statements must follow the package statement. import statements should be sorted with the most fundamental packages first, and grouped with associated packages together and one blank line between groups.   | The import statement location is enforced by the Java language. The sorting makes it simple to browse the list when there are many imports, and it makes it easy to determine the dependencies of the present package. The grouping reduce complexity by collapsing related information into a common unit. | <pre>import java.io.*; import java.net.*;  import java.rmi.* import java.rmi.server.*;  import javax.swing.*; import javax.swing.event.*;  import org.apache.server.*;</pre> |
| <b>Classes and Interfaces</b>  |   |  |
| Class and Interface declarations should be organized in the following manner: <ol style="list-style-type: none"> <li>1. Class/Interface documentation.</li> <li>2. class or interface statement.</li> <li>3. Class (static) variables in the order public, protected, package (no access modifier), private.</li> <li>4. Instance variables in the order public, protected, package (no access modifier), private.</li> <li>5. Constructors.</li> </ol> Methods (no specific order). | Reduce complexity by making the location of each class element predictable.   |  |

*Continued on next page*

## Implementation Model, Continued

### Statements (continued)

| Recommendation  | Rationale  | Example  |
|---|--|--|
| <b>Methods</b>  |  |  |
| Method modifiers should be given in the following order:<br><access> static abstract synchronized<br><unusual> final native<br>The <access> modifier (if present) must be the first modifier. | <access> is one of public, protected or private while <unusual> includes volatile and transient. The most important lesson here is to keep the access modifier as the first modifier. Of the possible modifiers, this is by far the most important, and it must stand out in the method declaration. For the other modifiers, the order is less important, but it make sense to have a fixed convention. |  |
| <b>Types</b>  |  |  |
| Type conversions must always be done explicitly. Never rely on implicit type conversion.  | By this, the programmer indicates that he is aware of the different types involved and that the mix is intentional.  | <pre>floatValue = (float) intValue; // NOT: floatValue = intValue;</pre> |
| <b>Variables</b>  |  |  |
| Variables should be initialized where they are declared and they should be declared in the smallest scope possible.   | This ensures that variables are valid at any time. Sometimes it is impossible to initialize a variable to a valid value where it is declared. In these cases it should be left uninitialized rather than initialized to some phony value.  |  |
| Variables must never have dual meaning.   | Enhances readability by ensuring all concepts are represented uniquely.<br><b>Reduce chance of error by side effects.</b>  |  |
| Class variables should never be declared public.  | The concept of Java information hiding and encapsulation is violated by public variables. Use private variables and access functions instead. One exception to this rule is when the class is essentially a data structure, with no behavior (equivalent to a C++ struct). In this case it is appropriate to make the class' instance variables public   |  |

*Continued on next page*

## Implementation Model, Continued

### Statements (continued)

| Recommendation  | Rationale  | Example   |
|---|--|---|
| Related variables of the same type can be declared in a common statement. Unrelated variables should not be declared in the same statement. | The common requirement of having declarations on separate lines is not useful in the situations like the ones above. It enhances readability to group variables. Note however that whenever possible, values should be initialized where they are declared (see Rule #40), in case this situation doesn't arise. | <pre>float x, y, z; float revenueJanuary, revenueFebruary, revenueMarch;</pre>  |
| Arrays should be declared with their brackets next to the type.   | The reason for is twofold. First, the array-ness is a feature of the class, not the variable. Second, when returning an array from a method, it is not possible to have the brackets with other than the type (as shown in the last example).  | <pre>double[] vertex; // NOT: double vertex[]; int[] count; // NOT: int count[];  public static void main (String[] arguments)  public double[] computeVertex()</pre> |
| Variables should be kept alive for as short a time as possible.   | Keeping the operations on a variable within a small scope, it is easier to control the effects and side effects of the variable.   |   |
| <b>Loops</b>  |  |   |
| Only loop control statements must be included in the for() construction.  | Increase maintainability and readability. Make a clear distinction of what controls and what is contained in the loop.   | <pre>sum = 0; // NOT: for (i=0, sum = 0; i &lt; 100; i++) for (i = 0; i &lt; 100; i++) // sum += value[i]; sum += value[i];</pre>                                     |
| Loop variables should be initialized immediately before the loop.   |  | <pre>boolean isDone = false; // NOT: isDone = false; while (!isDone) { // : : // while (!isDone) { } // : // }</pre>  |

*Continued on next page*

## Implementation Model, Continued

### Statements (continued)

| Recommendation   | Rationale   | Example   |
|--|---|---|
| <p>The use of do .... while loops should be avoided.</p>   | <p>There are two reasons for this. First is that the construct is superfluous; Any statement that can be written as a do .... while loop can equally well be written as a while loop or a for loop. Complexity is reduced by minimizing the number of constructs being used.</p> <p>The other reason is of readability. A loop with the conditional part at the end is more difficult to read than one with the conditional at the top.</p> |   |
| <p>The use of break and continue in loops should be avoided.</p>   | <p>These statements should only be used if they prove to give higher readability than their structured counterparts.</p>  |   |
| <b>Conditionals</b>  |   |   |
| <p>Complex conditional expressions must be avoided. Introduce temporary boolean variables instead.</p>     | <p>By assigning boolean variables to expressions, the program gets automatic documentation. The construction will be easier to read, debug and maintain</p>   | <pre>if ((elementNo &lt; 0)    (elementNo &gt; maxElement)        elementNo == lastElement) {     : } should be replaced by: boolean isFinished = elementNo &lt; 0    elementNo &gt; maxElement; boolean isRepeatedEntry = elementNo == lastElement; if (isFinished    isRepeatedEntry) {     : }</pre> |
| <p>The nominal case should be put in the if-part and the exception in the else-part of an if statement</p> | <p>Makes sure that the exceptions does not obscure the normal path of execution. This is important for both the readability and performance.</p>  | <pre>boolean isError = readFile (fileName); if (!isError) {     : } else {     : }</pre>  |

*Continued on next page*

## Implementation Model, Continued

### Statements (continued)

| Recommendation  | Rationale  | Example  |
|---|--|--|
| The conditional should be put on a separate line.   | This is for debugging purposes. When writing on a single line, it is not apparent whether the test is really true or not.  | <pre>if (isDone) // NOT: if (isDone) doCleanup(); doCleanup();</pre>   |
| Executable statements in conditionals must be avoided.  | Conditionals with executable statements are simply very difficult to read. This is especially true for programmers new to Java.  | <pre>file = openFile (fileName); // NOT: if ((file = openFile (fileName)) != null) { if (file != null) { // : : // } }</pre>                   |
| <b>Miscellaneous</b>  |  |  |
| The use of magic numbers in the code should be avoided. Numbers other than 0 and 1 can be considered declared as named constants instead. | If the number does not have an obvious meaning by itself, the readability is enhanced by introducing a named constant instead.   | <pre>private static final int TEAM_SIZE = 11; : Player[] players = new Player[TEAM_SIZE]; // NOT: Player[] players = new Player[11];</pre>     |
| Floating point constants should always be written with decimal point and at least one decimal.  | This emphasize the different nature of integer and floating point numbers. Mathematically the two model completely different and non-compatible concepts. Also, as in the last example above, it emphasize the type of the assigned variable (sum) at a point in the code where this might not be evident. | <pre>double total = 0.0; // NOT: double total = 0; double speed = 3.0e8; // NOT: double speed = 3e8; double sum; : sum = (a + b) * 10.0;</pre> |
| Static variables or methods must always be referred to through the class name and never through an instance variable.                     | This emphasize that the element references is static and independent of any particular instance. For the same reason the class name should also be included when a variable or method is accessed from within the same class.  | <pre>Thread.sleep (1000); // NOT: thread.sleep (1000);</pre>   |

*Continued on next page*

## Implementation Model, Continued

### Statements (continued)

| Recommendation   | Rationale  | Example   |
|--|--|---|
| <b>Layouts and Comments</b>  |  |   |
| Basic indentation should be 2.   | Indentation is used to emphasize the logical structure of the code. Indentation of 1 is too small to achieve this. Indentation larger than 4 makes deeply nested code difficult to read and increase the chance that the lines must be split. Choosing between indentation of 2, 3 and 4; 2 and 4 are the more common, and 2 chosen to reduce the chance of splitting code lines. Note that the Sun recommendation on this point is 4. | <pre>for (i = 0; i &lt; nElements; i++)   a[i] = 0;</pre>   |
| Block layout should be as illustrated in example 1 below (recommended) or example 2, and must not be as shown in example 3. Class, Interface and method blocks should use the block layout of example 2. | Example 3 introduces an extra indentation level which doesn't emphasize the logical structure of the code as clearly as example 1 and 2.   | <pre>Ex 1. while (!isDone) {   doSomething();   isDone = moreToDo(); } Ex. 2 while (!isDone) {   doSomething();   isDone = moreToDo(); } Ex. 3. while (!isDone) {   doSomething();   isDone = moreToDo(); }</pre> |
| The class or interface declarations should have the following form:  | This follows from the general block rule above. Note that it is common in the Java developer community to have the opening bracket at the end of the line of the class keyword. This is not recommended.   | <pre>class SomeClass extends AnotherClass   implements SomeInterface,   AnotherInterface {   ... }</pre>  |

*Continued on next page*

## Implementation Model, Continued

### Statements (continued)

| Recommendation   | Rationale   | Example   |
|--|---|---|
| <p>The method declarations should have the following form:</p> | <p>This follows partly from the general block rule above. However, it might be discussed if an else clause should be on the same line as the closing bracket of the previous if or else clause:</p> <pre>if (condition) {   statements; } else {   statements; }</pre> <p>This is equivalent to the Sun recommendation. The chosen approach is considered better in the way that each part of the if-else statement is written on separate lines of the file. This should make it easier to manipulate the statement, for instance when moving else clauses around.</p> | <pre>if (condition) {   statements; } if (condition) {   statements; } else {   statements; } if (condition) {   statements; } else if (condition) {   statements; } else {   statements; }</pre> |
| <p>The for statement should have the following form:</p>       | <p>This follows from the general block rule above.</p>  | <pre>for (initialization; condition; update) {   statements; }</pre>  |
| <p>An empty for statement should have the following form:</p>  | <p>This emphasize the fact that the for statement is empty and it makes it obvious for the reader that this is intentional.</p>   | <pre>for (initialization; condition; update) ;</pre>  |
| <p>The while statement should have the following form:</p>     | <p>This follows from the general block rule above.</p>  | <pre>while (condition) {   statements; }</pre>  |
| <p>The do-while statement should have the following form:</p>  | <p>This follows from the general block rule above.</p>  | <pre>do {   statements; } while (condition);</pre>  |

*Continued on next page*

## Implementation Model, Continued

### Statements (continued)

| Recommendation   | Rationale   | Example  |
|--|---|--|
| <p>The switch statement should have the following form:</p>  | <p>This differs from Sun both in indentation and spacing. In particular, each case keyword is indented relative to the switch statement as a whole. This makes the entire switch statement stand out. Note also the extra space before the : character. The explicit Fallthrough comment should be included whenever there is a case statement without a break statement. Leaving the break out is a common error, and it must be made clear that it is intentional when it is not there.</p> | <pre>switch (condition) {   case ABC :     statements;     // Fallthrough   case DEF :     statements;     break;   case XYZ :     statements;     break;   default :     statements;     break; }</pre> |
| <p>A try-catch statement should have the following form:</p> | <p>This follows partly from the general block rule above. This form differs from the Sun recommendation in the same way as the if-else statement described above.</p>   | <pre>try {   statements; } catch (Exception exception) {   statements; }  try {   statements; } catch (Exception exception) {   statements; } finally {   statements; }</pre>                            |

*Continued on next page*

---

## Implementation Model, Continued

---

### Statements (continued)

| Recommendation   | Rationale  | Example  |
|--|--|--|
| Single statement if-else, for or while statements can be written without brackets. | It is a common recommendation (Sun Java recommendation included) that brackets always be used in all these cases. However, brackets are a language construct that groups several statements. Brackets are by definition superfluous on a single statement. A common argument against this syntax is that the code will break if an additional statement is added without also adding the brackets. In general, code should never be written to accommodate for changes that might arise. | if (condition)<br>statement;<br><br>while (condition)<br>statement;<br><br>for (initialization; condition; update)<br>statement; |

---

*Continued on next page*

## Implementation Model, Continued

### Statements (continued)

| Recommendation  | Rationale   | Example   |
|---|---|---|
| <b>White Space</b>  |   |   |
| <ul style="list-style-type: none"> <li>Operators should be surrounded by a space character.</li> <li>- Java reserved words should be followed by a white space.</li> <li>- Commas should be followed by a white space.</li> <li>- Colons should be surrounded by white space.</li> <li>- Semicolons in for statements should be followed by a space character.</li> </ul> | Makes the individual components of the statements stand out and enhances readability. It is difficult to give a complete list of the suggested use of whitespace in Java code. The examples above however should give a general idea of the intentions.   | <pre> a = (b + c) * d;           // NOT: a=(b+c)*d while (true) {           // NOT: while(true) ... doSomething (a, b, c, d); // NOT: doSomething (a,b,c,d); case 100 :                // NOT: case 100: for (i = 0; i &lt; 10; i++) { // NOT: for(i=0;i&lt;10;i++){                     </pre>                     |
| Function names should be followed by a white space when it is followed by another name.   | Makes the individual names stand out and enhances readability. When no name follows, the space can be omitted since there is no doubt about the name in this case. An alternative to this approach is to require a space after the opening parenthesis. Those that adhere to this standard usually also leave a space before the closing parentheses: doSomething( parameter );. This do make the individual names stand out as is the intention, but the space before the closing parenthesis is rather artificial, and without this space the statement looks rather asymmetrical (doSomething( parameter);). | <pre> doSomething (parameter); // NOT: doSomething(parameter); doSomething();           // OK                     </pre>  |
| Logical units within a block should be separated by one blank line.   | Enhances readability by introducing white space between logical units of a block.   | <pre> Matrix4x4 matrix = new Matrix4x4();  double cosAngle = Math.cos (angle); double sinAngle = Math.sin (angle);  matrix.setElement (1, 1, cosAngle); matrix.setElement (1, 2, sinAngle); matrix.setElement (2, 1, -sinAngle); matrix.setElement (2, 2, cosAngle);  multiply (matrix);                     </pre> |

*Continued on next page*

## Implementation Model, Continued

### Statements (continued)

| Recommendation   | Rationale  | Example  |
|--|--|--|
| Methods should be separated by 3-5 blank lines.                  | By making the space larger than space within a method, the methods will stand out within the class.  |  |
| Variables in declarations should be left aligned.                | Enhances readability. The variables are easier to spot from the types by alignment.  | <pre> TextFile file; int    nPoints; double x, y;           </pre>   |
| Statements should be aligned wherever this enhances readability. | <p>There are a number of places in the code where white space can be included to enhance readability even if this violates common guidelines. Many of these cases have to do with code alignment. General guidelines on code alignment are difficult to give, but the examples above should give some general hints. In short, any construction that enhances readability should be allowed.</p> | <pre> if (a == lowValue)   compueSomething(); else if (a == mediumValue)   computeSomethingElse(); else if (a == highValue)   computeSomethingElseYet();  value = (potential    * oilDensity) / constant1 +         (depth        * waterDensity) / constant2 +         (zCoordinateValue * gasDensity) / constant3;  minPosition  = computeDistance (min, x, y, z); averagePosition = computeDistance (average, x, y, z);  switch (value) {   case PHASE_OIL : phaseString = "Oil";   break;   case PHASE_WATER : phaseString = "Water"; break;   case PHASE_GAS ; : phaseString = "Gas";   break; }           </pre> |

*Continued on next page*

## Implementation Model, Continued

### Statements (continued)

| Recommendation   | Rationale   | Example   |
|--|---|---|
| <b>Comments</b>  |   |   |
| Tricky code should not be commented but rewritten  | In general, the use of comments should be minimized by making the code self-documenting by appropriate name choices and an explicit logical structure.  |   |
| All comments should be written in English.   | In an international environment English is the preferred language.  |   |
| Use // for all non-JavaDoc comments, including multi-line comments.  | Since multilevel Java commenting is not supported, using // comments ensure that it is always possible to comment out entire sections of a file using /* */ for debugging purposes etc.   | // Comment spanning<br>// more than one line  |
| Comments should be indented relative to their position in the code   | This is to avoid that the comments break the logical structure of the program.  | while (true) { // NOT: while (true) {<br>// Do something // // Do something<br>something(); // something();<br>} // } |
| The declaration of collection variables should be followed by a comment stating the common type of the elements of the collection.                   | Without the extra comment it can be hard to figure out what the collection consists of, and therefore, how to treat the elements of the collection. In methods taking collection variables as input, the common type of the elements should be given in the associated JavaDoc comment. | private Vector points_; // of Point<br>private Set shapes_; // of Shape   |
| All public classes and public and protected functions within public classes should be documented using the Java documentation (javadoc) conventions. | This makes it easy to keep up-to-date online code documentation.  |   |

## Appendix A

### Naming operations

#### Prefixes

The following table provides examples of the naming convention using prefixes. The table also indicated their category, and adds descriptions.

| Prefixes                      | Category     | Description  |
|-------------------------------|--------------|--|
| <b>General Prefixes</b>       |              |  |
| is, can, has, will            | Testing      | Return a Boolean and test the state of the object  |
| new                           | Creating     | Create and return a new object from a factory that creates only a single type of object  |
| init, setup                   | Initializing | These methods are called before you can use an object. Only a single init function should be called which can then be followed by whatever setup methods you need to change the default configuration of the object. |
| <b>Type Specific Prefixes</b> |              |  |
| Find                          | Searching    | Retrieve a single object or null if unsuccessful   |
| select                        | Searching    | Retrieve multiple objects or an empty collection   |
| add                           | N/A          | Add an object to a collection  |

*Continued on next page*

## Naming operations, Continued

**Non-prefix** The following table provides examples of non-prefix naming convention

| Operation Name | Category | Description   |
|----------------|----------|---|
| Any            | N/A      | Return any object that satisfies the request (findAny)  |
| All            | N/A      | Return all objects that satisfy the request (selectAll) |

**Categories for method**

The following list provides categories for method:

- **Constructing** a section and category. The constructors for the class.
- **Initializing** an additional method that should be applied directly after constructing the object.
- **Setup Methods** that can optionally be applied to an object but must be done immediately after construction and initialization and before using the object normally.
- **Validating Check** whether the current object is in an acceptable state (could also be under asking if this is possible after construction is finishing).
- **Asking.** Asking the state of the current object without causing any (visible) side effects. A pure function. ISE Eiffel 'Query'.
- **Testing.** An asking method that returns a Boolean value.

## Appendix B

### Glossary

**Terms** The following definitions are ChiMu’s distilling and reconciliation of the many concepts and work that have been contributed to Object Oriented Design. Many of the sources for these terms are mentioned in of this document. Some other notable sources are:

- The Dictionary of Object Technology [Firesmith+E 95].
- UML: The Unified Modeling Language [Rational 98]
- Design Patterns, especially [Gamma+HJV 96]

---

| Term                     | Description   |
|--------------------------|---|
| Adapter                  | An object that can convert an Interface of one Class to the interface that another Object expects   |
| Architecture             | A concepts, structures, and interactions of a system. Also, the description of a system’s desired architecture before construction.   |
| Association <sub>1</sub> | A relationship between two Types that allow or require Objects of those Types to be linked.   |
| Association <sub>2</sub> | An Association <sub>1</sub> , but must be between Types which have Objects with Identity. See Attribute and ValueObject.  |
| Attribute1               | A public property of an object that shows an aspect of the state of the object. Frequently there is a minimal collection of attributes that uniquely determine the state of the object. See also Property.          |
| Attribute2               | See BasicAttribute.   |
| Attribute3               | See Instance Variable.  |
| BasicAttribute           | An Attribute1 that takes its value from ValueObjects. This is as opposed to associations, which connect two or more objects with identity. A BasicAttribute is traversable only from the Object to the ValueObject. |
| Bean                     | An Object that knows about its own properties (can introspect) and has several other capabilities. Any Java Object can be a Bean but some Objects have more “Bean” functionality.                                   |

*Continued on next page*

## Glossary, Continued

### Terms (continued)

| Term           | Description   |
|----------------|---|
| Behavior       | The response of an object to a stimulus. An object's behavior is the answers it gives to messages both now and in the future. (See State).  |
| Binding        | Associating a client object to a database object, which turns the client object into a Proxy  |
| Class          | Provides a common implementation for a set of objects.  |
| Container      | A Registry but without the implication of a primary registration property (e.g. a "key").   |
| DomainModel    | All the static rules, constraints, and operations that apply to DomainObjects. The DomainModel can either be conceptual to help understand the behavior of DomainObjects or it can be implemented as DomainClasses  |
| DomainObject   | An object, which captures knowledge about a domain. DomainObjects allow a computer to inspect, imply, modify, and "reason" about that information in either very simple ways (the facts) or more complex ways (the rules and implications)  |
| Extend         | To define a new Type (called a Subtype) in terms of an existing type (called a Supertype). The new Subtype will have the same contract (operations) as the Supertype but can add new functionality: as either new operations or enhancement in capability to existing operations. |
| Extent         | The collection of all instances of a Type or Class.   |
| ExtentRegistry | A Registry that contains all the objects of a Type (the Extent of the Type). An ExtentRegistry is a close equivalent to a RelVar or Table in the context of Objects.  |
| Factory        | An object that can create other objects.  |
| Feature        | The Eiffel term for Operation where Operation includes both methods and attributes <sub>1</sub> .   |
| Forwarder      | A proxy, which immediately forwards messages, over process and machine boundaries, to the RealSubject.  |

*Continued on next page*

## Glossary, Continued

### Terms (continued)

| Term                  | Description   |
|-----------------------|---|
| Framework             | A strong partition of generalized functionality common to many parts of an application. Also, more formally, a collection of interacting classes that describes most of the behavior a client requires and can be subclassed and parameterized to customize and complete the functionality. |
| Function <sub>f</sub> | A functor that returns an Object  |
| Functor               | An object that models an operation.   |
| Functor               | “An object that models an operation” [Firesmith+E 95]. For a Java implementation, a basic functor is an Interface with a single, generic, operation.  |
| Getter <sub>f</sub>   | A functor that is designed to retrieve a value from an object (the first parameter)   |
| Identity              | The ability to tell one object from another object independently of whether their appearance (behavior) is identical  |
| IdentityKey           | A value that defines the RealIdentity of a Proxy  |
| Immutable             | Can not be changed after being created. Immutable objects can not be changed after they are created and fully initialized.  |
| Inherit               | To define a new Class in terms of an Existing Class (the Superclass) by starting with the Superclass’s implementation and i-directi or adding to it   |
| Instance              | An object is an Instance of a Type if an object supports all the exterior requirements of that type (see “Is-A”). An object is an instance of a Class if it is implemented by that class  |
| Instance Variable     | A way to store encapsulated state information for a particular object. Instance variables are completely hidden within the Object, but they enable two objects of the same Class to have different external Behavior.   |
| Interface             | A description of a Type focused on the Operations that the objects can respond to.  |

*Continued on next page*

## Glossary, Continued

### Terms (continued)

| Term         | Description   |
|--------------|---|
| Is-A         | An object is a Type if an Object supports all the exterior requirements of that Type  |
| Layer        | A logical, horizontal division of a system that provides a particular system abstraction to the client above the layer.   |
| Link         | A connection between two objects which allows one or both to know about the other object. By default links are assumed to be i-directional in analysis, but they can be defined to be only traversable in one direction   |
| Message      | A stimulus sent to an object with a name and any parameters (as Objects) that the message requires. A message will cause the receiver Object to return an answer or nothing. In most Object Languages, the sender has to wait for the answer before continuing  |
| Method       | An implementation of an operation for a particular object/class.  |
| Module       | A base level subsystem: one which does not contain any other subsystems   |
| Object       | An identifiable, encapsulated entity that can only be interacted with by sending messages   |
| ObjectBase   | An ObjectBase captures the knowledge, operations, and rules required to usefully represent a particular part of the world in a computer. An ObjectBase contains all the objects that represent a particular state of your DomainModel and all the knowledge contained therein. Also called an ObjectSpace |
| ObjectShadow | The information needed to see that an object exists without any true representation of the real object. Relational databases could be considered to work with ObjectShadows: they record the information about an object but never have a real object to interact with.                                   |

*Continued on next page*

## Glossary, Continued

### Terms (continued)

| Term                   | Description  |
|------------------------|--|
| Observable             | An object that can be Observed. See Observer   |
| Observer               | An object that “looks at” another object (the Observable) and can respond to events in the Observable without the Observable being knowledgeable about the Observer  |
| Operation              | A description of the ability for an object to respond to a particular message and the contract/requirement for that message.   |
| Partition              | A vertical division of a system into areas of related functionality  |
| Predicate <sub>f</sub> | A functor that returns a Boolean.  |
| Procedure <sub>f</sub> | A functor that does not return a value.  |
| Property               | Synonym for Attribute <sub>1</sub> and sometimes for Attribute <sub>2</sub>  |
| Prototype              | An object that is used as a template for creating other Objects  |
| Proxy                  | An object that stands in for another object (the RealObject) and manages the client interaction with the RealObject  |
| RealIdentity           | The identity of the RealObject that a proxy represents instead of the proxy’s independent identity. For proxies we are rarely interested in their own identity, we just want to know the identity of the RealObject on the server. |
| Registry               | An object that remembers other objects and can search through and retrieve them through one or more properties. Usually the objects within a Registry are all of the same type   |
| Replicate              | A proxy which holds local state and performs local operations which are later propagated to the RealSubject  |
| Role                   | The name of the “position” within a relationship an Object or Type holds. For example, a binary association has two roles that distinguish the two participants in the relationship  |

*Continued on next page*

## Glossary, Continued

### Terms (continued)

| Term                | Description   |
|---------------------|---|
| Setter <sub>f</sub> | A functor that stores into an object (the first parameter) a value (the second parameter)   |
| Singleton           | An object that is the only instance of a Class  |
| State               | An abstraction to describe and simplify understanding an object's behavior. An object's behavior can be described as the answers it gives to current messages (which are determined by the current state) and the changes to its state caused by these messages |
| Strategy            | An object that encapsulates an algorithm to be used with an Object  |
| Stub                | A proxy which acts as a placeholder for the RealObject and must become another type of proxy (for example, forwarder or replicate) when interacted with by a client   |
| Subsystem           | A division of a system into a cohesive unit of functionality (tightly related classes and internal subsystems) with a public interface and a private implementation   |
| Subtype             | A Type that extends another Type (called the Supertype)   |
| Supertype           | A Type that has been extended by another Type   |
| Tier                | A level on a hierarchy of processes over which a system is divided  |
| Traverse            | To move from one object to another by a Link. If a Link is traversable from an Object than that Object can get to (knows about) the other Object  |
| Type                | Describes a common exterior (public behavior) of a set of Objects. Can also be used to conceptually group and understand objects by their similar behavior  |

*Continued on next page*

---

## Glossary, Continued

---

### Terms (continued)

| Term        | Description   |
|-------------|---|
| ValueObject | An object that does not have identity independent of its value. A ValueObject is immutable and should be considered identical to anything that it is equal to. Primitive data types in Smalltalk (most numbers, Symbols) are ValueObjects. Java Strings are very close to ValueObjects except they are not guaranteed to be identical for the same value (they would be if they did an automatic “intern()”). Java primitive types are not Objects. |
| Visitor     | An object representing an operation that can be performed on the elements of an Object structure (frequently a hierarchy or sequence).  |

---

## Appendix C

### Java Reference list

---

**List** The following is a list of manufacturers and their URLs where more information can be found on various Java coding standards and guidelines.

---

| Manufacturer      | URL  |
|-------------------|--|
| ChiMu             | <a href="http://www.chimu.com/publications/javaStandards/index.html">http://www.chimu.com/publications/javaStandards/index.html</a>  |
| The AmbySoft Inc. | <a href="http://www.ambysoft.com/downloads/javaCodingStandards.pdf">http://www.ambysoft.com/downloads/javaCodingStandards.pdf</a>  |
| MOST              | "The Most Important Design Guideline" by Scott Meyer   |
| IEEE              | <a href="http://www.aristeia.com/Papers/IEEE_Software_JulAug_2004.pdf">http://www.aristeia.com/Papers/IEEE_Software_JulAug_2004.pdf</a>  |
| SUN Microsystems  | <a href="http://java.sun.com/j2se/javadoc/writingdoccomments/index.html">http://java.sun.com/j2se/javadoc/writingdoccomments/index.html</a>  |
| SUN Microsystems  | <a href="http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html">http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html</a>  |
| Hammurapi         | Open source software developed by Pavel Vlasov, Hammurapi is a tool that can analyze a code base against a set of design guidelines. There is a plug in available for Eclipse (WSAD). See: <a href="http://www.hammurapi.org">http://www.hammurapi.org</a> |
| Bruce Eckel       | <i>Thinking in Java</i> is a book written by Bruce Eckel. For more information, go to: <a href="http://www.BruceEckel.com">http://www.BruceEckel.com</a>   |

---