Software Engineering Group Projects – Java Coding Standards

Author:	C. J. Price, A. McManus
Config Ref:	SE.QA.09
Date:	17th Oct 1998
Version:	1.0
Status:	Release

Department of Computer Science
University of Wales
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1. INTRODUCTION

1.1 Purpose of this Document

The purpose of this document is to help software engineering project groups produce high quality Java programs through the use of a set of rules and guidelines.

1.2 Scope

This document specifies the standards for writing Java software on departmental second year group projects. It is solely concerned with Java code which will be submitted to a Java compiler.

The document is presented as a set of rules and guidelines which should be followed when writing Java programs. Examples of the rules are provided, and in the case of restrictions to the use of the language, some rationale is given.

This document should be read by all project members. It is assumed that the reader is already familiar with the QA plan for group projects [1].

The document covers the following aspects of writing Java:

Code organisation: strategies for organising and naming packages.

Identifier Naming Conventions: rules for naming identifiers to make clear their type and purpose.

Class organisation: rules for how to arrange the parts of a class to make them easier to read.

Comments: rules for content of mandatory comments in programs,

Indentation: how to set out Java code for the group project

Language features: rules and guidelines for use or avoidance of some of the features of Java.

1.3 Objectives

The objective of this document is to provide guidance for the production of clear, readable, understandable, maintainable Java code. This will include formatting and presentation, code structure, and elements of the language to be used.

2. Code Organisation

Java bundles related classes into packages. Sun specify that a company's package names should start with the reversed domain-name of the company, in order to enforce uniqueness. In our case, this means all packages should start with cs221.<groupname>. The package identifiers should be entirely lower-case, without underscores. For general purpose classes (i.e. ones that would be useful outside the group project), we follow the naming convention used by Java (and by Netscape for the Internet Foundation Classes (IFC)). For example, a general utility class should be stored in cs221.<groupname>.util.

Packages are hierarchical, which makes it easier to organise their contents. If there is a subset of related classes within a package, it often makes sense to create a new sub-package for them. For example, the com.sun.java.swing package contains the Swing GUI components, and it has a tree sub-package that contains GUI classes that support the JTree component. Note that the JTree itself is in the higher-level package - this means that programmers wanting to use JTree can do so without importing the sub-package, unless they want access to its more complicated functionality.

Each new application should be given a new package. This package should contain the top-level application class, and any other classes that may be of use in other code. Classes that are specific to the application (dialogs, for example), should be placed in a sub-package called app.

Objects that are not specific to a particular application should be in a separate package. For example, a Diary class might be associated with the Heating Control application, but is also used in the Heating Booking Entry system - it is given its own package (cs221.<groupname>.diary).

Maintaining a clear package structure is important: it makes it easier to locate classes, aiding reuse and reducing the risk of duplicating classes. For this reason, a package registry should be created on-line, listing all packages and their purpose. Programmers should check this to find out which packages should be used for a new classes. If the registry does not contain an appropriate package, the programmer should put in a request for a new entry in the registry to the project librarian

3. Identifier Naming Conventions

All identifiers should use U.S. spelling. This is for consistency with external libraries, including the standard Java library. For example:

Color not Colour
MyClass.initialize( ) not MyClass.initialise( ).

3.1 General

When choosing names, try to apply the following guidelines:

Choose names that are as self-documenting as possible. *indexVariable* rather than *i*.

Use real world object names for objects, e.g. *diaryEntry*.

Use predicate clauses or adjectives for boolean objects or functions, e.g. *heatingShouldBeOn*.

Use action verbs for procedures and entries, e.g. *removeNode*.

Use constants rather than variables for constant values.

3.2 Classes and Interfaces

Class and interface names start with a capital letter, then use lower-case with capitals separating words (rather than underscores). For example:

```
public class StateMachine ...

public class DataManager ...
```

When the word in a class would be upper-case (such as an abbreviation like FMEA), only the first letter should be a capital when used in an identifier. For example:

```
public class FmeaEditor ...

public class GuiResourceBundle ...
```

Exceptions in Java always end with the word Exception. In addition, abstract classes should be prefixed with Abstract, and standard (baseline) implementations of an abstract class or interface should be prefixed with Std. Implementations of an interface that only provide skeleton implementations of the methods should end in Adapter. For example:

```
// Abstract and Std.
public abstract class AbstractWibble {
}

public class StdWibble extends AbstractWibble {
}

// Interface and Adapter.
public interface MouseListener {
}

public class MouseAdapter
implements MouseListener {
}
```

3.3 Methods

Method names start with a lower-case letter, and use capitals to separate words (rather than underscores). For example:

```
public void buildTree( Node root );
```

The naming of methods should follow the Java Beans convention. This means that properties should have a get<PropertyName>( ) method (or is<PropertyName>( ) for booleans), and read-write properties should also have a set<PropertyName>( ) method. For example:

```
// Read-only size property.
public int getSize( );

// Read-write name property.
public String getName( );

public void setName( String name );

// Boolean readOnly property.
```

```
public boolean isReadOnly( );

public void setReadOnly( boolean b );
```

Indexed properties should normally have get and set methods that allow you to access individual values, or an entire array:

```
// Correct indexed property.

public String getUser( int index );

public String[] getUser( );

public void setUser( int index, String user );

public void setUser( String users[] );
```

This convention is awkward when the number of elements in a property can vary. In this case it is better to provide a get<PropertyName>( ) and a get<PropertyName>Count( ) method. The elements can be modified through add<PropertyName>( ) and remove<PropertyName>( ) methods, or with a set<PropertyName>( ) method. Clearly, this is not compatible with the Java Beans standard, but a BeanInfo class may be provided if this compatibility is necessary.

```
// More flexible indexed property.

public String getUser( int index );

public int getUserCount( );

public void addUser( String user );

public void removeUser( String user );
```

Event listeners should provide add<EventListener>( ) and remove<EventListener>( ) methods:

```
public void addMouseListener( MouseListener l );

public void removeMouseListener( MouseListener l );
```

3.4 Variables

Variable names start with a lower-case letter, and use capitals to separate words (rather than underscores). This applies to class, instance and local variables. There are no standard prefixes or suffixes for different data types - Java has much stronger type-checking than C++, which makes this unnecessary. Container variable names should be in the plural. For example:

```
public class MyClass {
   protected static int instanceCount;
   protected String names[10];

   public void doSomething( ) {
      String currentName = names[0];
      ...
   }
   ...
}
```

Instance variables that represent properties should be named after the property:

```
public class MyClass {

   protected String name;

   public String getName( ) {
      return name;
   }


   public void setName( String name ) {
      this.name = name;
   }
}
```

The setName( ) method in the last example shows variable shadowing: the name parameter shadows the name instance variable. Trying to avoid shadowing often leads to obscure parameter names which can affect readability. However, shadowing can make code obscure because it may not be clear which value is being referred to. Shadowing is allowed (and encouraged) subject to the following conditions:

The value of the parameter is going to be assigned to the corresponding instance variable.

The assignment is the first statement in the method. After this, both variables refer to the same value, and any confusion is avoided.

3.5 Constants

Constants (static final variables) follow the same conventions are normal variables. For example:

```
public class File {
   public static final String pathSeparator = "\";
   ...
}
```

The exception to this is when the constants form part of an enumeration (in fact, Java does not have enumerations, but uses int data types and constants). In this case, the variable name is entirely upper-case, using underscores to separate words. If a set of constants are associated directly with the class or interface, they do not need a common suffix. If this is not the case, a suffix based on the enumeration should be used. For example:

```
// Enumeration constants associated with an interface.
public interface StatusConstants {
   public static final int READY = 0;
   public static final int STANDBY = 1;
}

// Enumeration constants not directly associated with a class.
public class WibbleManager {
   public static final int NORMAL_MODE = 0;
   public static final int SAFE_MODE = 1;
   public static final int SPEED_MODE = 2;
   public void setMode( int mode ) {
...
   }
...
}
```

4. Class Organisation

4.1 File Structure

Java requires every public class and interface to be defined in a file with the same name. In order to keep the size of files small, and to make it easy to locate classes, we require that every top-level class should be defined in its own file, regardless of its access modifier (private, protected, etc). The only exception is for test classes which are not used outside the file.

4.2 Class Structure

Every class and interface of a reasonable size should have its variables and methods arranged into groups, preceded by a comment. The idea is to group related methods together, which should assist someone maintaining the code to navigate through the class. Note that the access modifier is not one of the criteria for grouping the methods. This is because the javadoc web pages provide the API reference (which can separate out the different modifiers) rather than the source code.

The order of groups should be:

- Constants (static final)
- Class variables (static)
- Class methods (static)
- Instance variables
- Constructors (includes finalize( ))
- Properties
- Other methods
- Test methods

An example class:

```
public class MyClass
implements FocusListener {

   // ////////// //
  // Constants. //
  // ////////// //
   public static int READY_MODE = 0;
   public static int STANDBY_MODE = 1;

  // //////////////// //
  // Class variables. //
  // //////////////// //
  protected static int instanceCount = 0;

   // ////////////// //
   // Class methods. //
   // ////////////// //
  public static int getInstanceCount( ) {
      return instanceCount;
   }

   // /////////////////// //
   // Instance variables. //
   // /////////////////// //
   protected String name;
   // ///////////// //
   // Constructors. //
   // ///////////// //
   public MyClass( String name ) {
      this.name = name;
      instanceCount++;
   }

   // ///////////////////// //
   // Read-only properties. //
   // ///////////////////// //
   public String getName( ) {
      return name;
   }

   // ////////////////////// //
   // FocusListener methods. //
   // ////////////////////// //
   public void focusGained( FocusEvent e ) {
      ...
   }

   public void focusLost( FocusEvent e ) {
      ...
   }

   // ///////////// //
   // Test methods. //
   // ///////////// //
   public static void main( String arvg[] ) {
      ...
   }
}
```

Templates are listed in appendices for classes and interfaces, including some standard groupings. The programmer should choose the most appropriate groups for each class - often a method could fit into a few different groups.

4.3 Inner Classes

Inner classes may be used to break up the complexity of a large class. They are also useful when creating GUIs with nested panels. Inner classes should not be used by code outside the parent class, unless the inner class could be considered an attribute of the parent class.

4.4 Anonymous Classes

Anonymous classes should only be used to pass simple implementations of an interface as parameters to a method. For example:

```
Runnable runnable = new Runnable( ) {
public void run( ) {
      System.err.println( "I'm running!" );
   }
};

Thread thread = new Thread( runnable );

thread.start( );
```

5. Comments

The commenting style is generally driven by the requirements of javadoc. All javadoc comments start with "/**" and end with "*/". For non-javadoc comments, the single line style should be used to help distinguish the two (starting with "//"). The multi-line comment can be useful for commenting out blocks of code, but clearly any finished code should not include this!

Note that the templates available for classes and interfaces include file and class/interface headers.

5.1 Files

Each file should have a simple header giving the filename, the authors, a copyright message, and the version and date. The SCCS substitution codes should be used for most of these:

```
// %M%
// By Alex McManus.
// Copyright 1998 Centre For Intelligent Systems,
// University of Wales, Aberystwyth.
// All rights reserved.
// %I% %D%
```

5.2 Classes and Interfaces

Each class or interface should have a standard javadoc class header. Note that:

- The description should provide an overview of the class, but need not go into great detail. The description should be separated from the tags by an empty line.
- An @author tag must be included for each author (except for inner-classes).
- A @version tag must be included for each version of the file (except for inner-classes).
- @see tags should be used to cross-reference related classes.
- Anonymous classes do not need headers.

For example:

```
/**
 * A class that generates new wibbles.
 * This class generates new instances that implement the Wibble interface.
 * The exact class that is returned depends on the current WibbleSystem
 * that is active.
 * <p>
 * Static getFactory( ) method should be used to create new instances of
 * WibbleFactory rather than the constructor, and new wibbles may be
 * obtained through the createNewWibble( ) method.
 *
 * @author Alex McManus
 * @author Richard Joseph
 * @version 1.1  Initial development.
 * @version 1.2  BN998: Now works with modified database structure.
 * @see Wibble
 * @see WibbleSystem
 * @see #getFactory( )
 * @see #createNewWibble( )
 */

public class WibbleFactory ...
```

5.3 Methods

Each method should a standard javadoc header. Note that:

- The description should cover the purpose of the method, and any side-effects.
- All parameters and return values should have @param or @return tags, even if they seem obvious. This helps give the resulting documentation a more complete feel.
- Tags of the same type should be lined up (e.g. all @param tags).
- Every type of exception thrown by the method should have an @exception tag (even if there is already a tag for one of the exception's superclasses).
- @see tags should be used to cross-reference related methods or classes.
- Methods in anonymous classes do not always need headers.
- Methods in skeletal test classes do not always need headers.

For example (most method headers won't be as long as this!):

```
/**
 * Create a new instance of the Wibble interface.
 * The actual class that is created depends on the current WibbleSystem
 * that is active.
 *
 * @param newName  the name to give the new Wibble.
 * @param mode     the mode to give the new Wibble, one of READY_MODE or
 *                 STANDBY_MODE.
 * @return a new Wibble.
 * @exception LockException  if a lock cannot be obtained for the Wibble.
 * @exception SQLException   if an SQL error occurs.
 * @see Wibble
 * @see WibbleSystem
 * @see #READY_MODE
 * @see #STANDBY_MODE
 */

public Wibble createNewWibble( String newName, int mode )

throws LockException, SQLException ...
```

5.4 Variables

Instance and class variables should each have a javadoc comment. When possible this should be on one line, to avoid the code being lost in comments. For example:

```
/** The mode that indicates the Wibble is ready for input.
 *  @see #STANDBY_MODE */
public static int READY_MODE = 0;

/** The name of the Wibble. */
protected String name;
```

Local variables in Java are generally declared at the point they are first used, and so their purpose is usually obvious from their context. If this is not the case, a brief comment should be appended to the line in which they are declared.

5.5 Blocks

Block comments are used to describe a group of related code. Most block comments should be one line, if possible, and reside immediately above the block being commented. If more than a one line comment is needed, the extra lines should each begin with the double slash.

Block comments should be indented to match the indentation of the line of code following it. A single blank line should precede the comment and the block of code should follow immediately after. Small blocks of code that do a specific job should be commented but not individual lines, unless the line is complex or not intuitive.

It is often useful to put comments before control structures (for-loops, ifs, whiles, etc.) to explain the purpose of the code in the blocks that follow. Some example block comments are shown below.

```
// For every node in the list...
for( int i = 0; i < nodes.size( ); i++ ) {

  Node node = nodes.elementAt( i );

   // If the node is a leaf, remove it from the tree and print...
   if( node instanceof LeafNode ) {
      tree.removeNode( node );
      System.out.println( node );

   // ...or if the node is binary, recursively print its children...
   } else if( node instanceof BinaryNode ) {
      printNode( ((BinaryNode)node).getChild( 0 ) );
      printNode( ((BinaryNode)node).getChild( 1 ) );

   // ...otherwise, simply print the node.
   } else {
      System.out.println( node );
   }
}
```

6. Indentation

The standard unit of indentation is three spaces. Note that tab characters should not be used, as tabs can be mapped to different numbers of spaces on different systems.

6.1 Blocks

The '{' character that starts a block should be at the end of the preceding line. The lines in the block should be indented, and the block closed with '}' at the same level as the line that started the block. For example:

```
for( int i = 0; i < nodes.size( ); i++ ) {
   Node node = (Node)nodes.elementAt( i );

   if( node instanceof LeafNode ) {
      System.out.println( node );
   }
}
```

Blocks do not need to be used after control statements, but if used for one part of a compound statement (such as if-then-else), it should be used for all parts. For example:

```
// Potentially unclear.
if( x > y )
   System.err.println( "Less than" );
else if( x < y ) {
   System.err.println( "Greater than" );
} else {
   System.err.println( "Equal" );
}

// Better.
if( x > y ) {
   System.err.println( "Less than" );
} else if( x < y ) {
   System.err.println( "Greater than" );
} else {
   System.err.println( "Equal" );
}


// Alternative.
if( x > y )
   System.err.println( "Less than" );
else if( x < y )
   System.err.println( "Greater than" );
else
   System.err.println( "Equal" );
```

6.2 Classes

The first line of a class or interface should declare the name of the class and (optionally) its parent. If a class implements an interface, this should be declared on the following line. For example:

```
public interface Wibble {
   ...
}

public class StdWibble extends Object
implements Wibble {
   ...
}


public class MyWibble extends Wibble
implements ExceptionListener, FocusListener {
   ...
}
```

6.3 Methods

The first line of a method should declare the return type, name and parameters of the method. If the method throws any checked exceptions, these should be declared on the following line. For example:

```
public int getSize( ) {
   ...
}

public String getName( DataConnection connection )
throws SQLException {
   ...
}
```

6.4 Case Statements

The first line of a switch statement contains the switch itself, and the start of the block. Each case statement (and the default) is level with the switch. The code under each case is indented (including the break). For example:

```
switch( mode ) {

case READY_MODE:
   ...
   break;

case STANDBY_MODE:
   ...
   break;

default:
   ...
   break;
}
```

6.5 Over-long Lines

For code that cannot fit on one line (80 characters), the basic formatting rule is that it should be formatted to make the code as clear and readable as possible. Guidelines for achieving this follow:

- Split the line at a comma or logical symbol if possible.
- Indent the following line. Say, for example, the line is split in the middle of a set of parameters to a method call:
- The following line should be indented to match the first parameter:

```
canvas.getGraphics( ).drawRect( x, y,
                                w, h);
```

- However, if this approach would cause the statement to spill over many lines, the second line should be indented by three spaces from the start of the method call:

```
// A bit silly.
canvas.getGraphics( ).drawRoundRect( x, y,
                                     w, h,
                                     arcW,
                                     arcH )
```

```
// Better.
canvas.getGraphics( ).drawRoundRect(
                      x, y, w, h, arcW, arcH );
```

- If this still causes the statement to spill over many lines, the second line should be indented by three spaces from the start of the line:

```
// Silly.
getCanvas( ).getGraphics( ).drawRoundRect(
                              x, y, w, h,arcW,
                              arcH );

// Better.
getCanvas( ).getGraphics( ).drawRoundRect(
   x, y, w, h, arcW, arcH );
```

- If the line wrapping causes the second line of a control statement to be at (or near) the same level of indentation as the body of the control statement, indent the second line a further three spaces. For example:

```
// Not very clear if the second line is inside the block.
if( dialog.calcCost( avgCost, interest,
    ratio, code ) == -1 )
    {System.err.println( "Error" );
}

// Better.
if( dialog.calcCost( avgCost, interest,
       ratio, code ) == -1 )
    {System.err.println( "Error" );
}
```

- If you have really deep indentation causing lots of line wrapping, you may have a case for splitting the method into further sub-methods. Alternatively, you can split the line into separate statements:

```
Graphics g = getCanvas( ).getGraphics( );
g.drawRoundRect( x, y, w, h, arcW, arcH );
```

- Braces {} and blank lines can be used to make code clearer when there is a lot of line wrapping in control statements.

7. Language Features

7.1 Nested Assignments

No nested assignment. It is possible to write expressions like a = b + (c= d * e)) in Java, where both a and c are given a value. This saves very little, and makes the code less clear. We will avoid it.

7.2 Use of Interfaces

When an object forms part of a framework, it should be defined by an interface. An abstract or standard implementation of the interface should also be provided. Because the object is defined by an interface, a class from any hierarchy can implement it. However, the convenience of subclassing an existing class is also provided through the abstract or standard class.

7.3 Constant Definitions

When a large number of constants must be defined, it is usually a good idea to define them in an interface. That way a class can access the constants (without having to prefix an interface or class name) by implementing the interface.

7.4 Variable Initialisation

Class variables can be initialised where they are declared. If the initialisation is complex or if it could throw an exception, the initialisation should take place in a static constructor:

```
protected static int instanceCount = 0;

protected GuiResourceBundle resources;

// Static constructor.

static {

   try {

      resources = GuiResourceBundle.getResources( "resources.DataPackage"
);

   } catch( IOException e ) {

      e.printStackTrace( );
   }
}
```

Instance variables should always be initialised in a constructor. This is clearer than initialising some variables where they are declared and some in the constructor. If a class provides multiple constructors, they should be chained to each other or they should all call an initialisation method, so that the initialisation code is only written once (and nasty bugs don't arise when a new variable is added and some initialisation code is missed). For example:

```
public class MyClass {

   protected DataManager dataManager;
   protected int id;
   protected String name;
   protected int mode;

   // All initialisation takes place here.
   protected MyClass( DataManager dataManager ) {
      this.dataManager = dataManager;
      mode = UNSAVED_MODE;
      name = null;
      id = -1;
   }

   public MyClass( DataManager dataManager, int id ) {
      this( dataManager );
      this.id = id;
   }

   public MyClass( DataManager dataManager, String name ) {
      this( dataManager );
      this.name = name;
   }
   ...
}


public class MyClass {
  protected String name;
  protected int id;

   public MyClass( int id ) {
      this( id, null );
   }


   public MyClass( String name ) {
      this( -1, name );
   }

   // All initialisation takes place here.
   public MyClass( int id, String name ) {

      this.id = id;
      this.name = name;
   }
   ...
}
```

7.5 Finalizers

Finalize methods should be defined for all classes that must tidy up their resources before they are destroyed. However, it is important that the class does not rely on the finalize method being called. These methods are only called when an instance is garbage collected. If the instance is in memory when the virtual machine ends, the finalize method will not be called. The class documentation should warn the programmer that this is the case.

7.6 Exceptions

Methods should always throw exceptions of an appropriate class. If such a class does not exist, a new one should be defined.

Exceptions should only ever be used for exceptional circumstances - never as a means of communicating the result of a method. Exceptions used in this way can confuse the flow of control in code.

7.7 Access Modifiers and Methods

The use of access modifiers (public, protected, etc.) is subject to the following guidelines:

- Classes will nearly always be public, or default if there is a good reason that it should not be visible outside the package.
- Constants should be public if they should be visible outside the class, or protected if they are local.
- Class variables and instance variables should nearly always be protected.

- Accessor methods should be provided for variables that must be accessed from outside the class. Other classes should never access the variables directly (unless the class is effectively a structure). It is generally better practise to use accessor methods even within the class. This is not required because it can make the code much more long-winded, and more importantly, it has performance implications.

- The default modifier (i.e. none) should not be used for methods or variables - the classes in a package should only interact through their public interfaces.

- The private modifier should rarely be used for methods, as it severely restricts the flexibility of any subclasses. Performance considerations might justify its use (private methods can be inlined), or if there is a good reason that subclasses should not be allowed to call the method.

- The private modifier can only be used on variables if accessor methods are provided.

- The final modifier should only be used if there is a good reason that a class should not be subclassed, or that a method should not be overridden. One situation in which it is useful, is to prevent a user overriding a method that is simply a wrapper for a method that does the work.

For example:

```
// This method does the work. If the behaviour of setting a name
// has to change, this method should be overridden so that the
// changes are propagated to both methods.
public void setName( String name ) {
     this.name = name;
}

// This method is a wrapper. If we allow this to be overridden,
// the two methods will now have different behaviour (perhaps
// unknown to the programmer).
public final void setName( StringBuffer name ) {
     setName( name.toString( ) );
}
```

7.8 Method Overloading

A class will often provide a number of overloaded methods (i.e. methods with the same name, but different parameters). The only restriction on this, is that the overloaded methods must all perform the same task.

7.9 Flow of Control

Java provides break and continue command to disrupt the flow of control within loops. Breaks are allowed within a loop, as they generally provide more readable code (compared to putting more complex conditions in the loop statements). Continues can be used where they enhance the readability of the code, but generally if-then-else statements are preferred. For example:

```
// Hmmm.
for( int i = 0; i < nodes.size( ); i++ ) {

   Node node = (Node)nodes.elementAt( i );

   if( node instanceof LeafNode ) {
      System.err.println( "Found leaf" );
      continue;
   }
   ...
}

// Better.
for( int i = 0; i < nodes.size( ); i++ ) {
   Node node = (Node)nodes.elementAt( i );
   if( node instanceof LeafNode ) {
      System.err.println( "Found leaf" );
   } else {
      ...
   }
}
```

8. REFERENCES

[1] H. R. Nicholls. Software engineering group projects – quality assurance plan. Technical Report SE.QA.01, University of Wales, Aberystwyth.

Appendix A: Listing of ClassTemplate.java

This template is also available on-line with the group project documents.

```
// %M%
// By Alex McManus.
// Copyright 1998 Centre For Intelligent Systems,
// University of Wales, Aberystwyth.
// All rights reserved.
// %I% %D%

package cs221.groupname?.?;


/**
 *
 * @author
 * @version 1.1  Initial development.
 */

public class  extends

implements  {


   // ////////// //
   // Constants. //
   // ////////// //


   // //////////////// //
   // Class variables. //
   // //////////////// //


   // ////////////// //
   // Class methods. //
   // ////////////// //


   // /////////////////// //
   // Instance variables. //
   // /////////////////// //


   // ///////////// //
   // Constructors. //
   // ///////////// //


   // ////////////////////// //
   // Read/Write properties. //
   // ////////////////////// //


   // ///////////////////// //
   // Read-only properties. //
   // ///////////////////// //



   // //////// //
   // Methods. //
   // //////// //


   // ///////////// //
   // Test methods. //
   // ///////////// //


   public static void main( String argv[] ) {


      if( argv.length < 1 ) {
         System.err.println( "Usage: " );
         System.exit( -1 );
      }


      try {

      } catch( Exception e ) {

         e.printStackTrace( );

      }

   }

}
```

Appendix B: Listing of InterfaceTemplate.java

This template is also available on-line with the group project documents.

```
// %M%
// By Alex McManus.
// Copyright 1998 Centre For Intelligent Systems,
// University of Wales, Aberystwyth.
// All rights reserved.
// %I% %D%


package cs221.groupname?.?;

/**
 *
 * @author
 * @version 1.1  Initial development.
 */

public interface  extends  {

   // ////////// //
   // Constants. //
   // ////////// //


   // //////// //
   // Methods. //
   // //////// //

}
```

DOCUMENT HISTORY

Version	*CCF No.*	*Date*	*Changes made to document*	*Changed by*
1.0	N/A	17/10/98	N/A - original version	CJP