CBOE Java Programming Guidelines

Version 1.2

3/18/03

Prepared by:

Mark A. Woyna
CBOE Systems Division

Steven Jones
ISA Services, Inc.

Introduction

The following criteria are for use on all Java programming projects being done at The Chicago Board Options Exchange (CBOE). The goal of these rules is to provide a uniform coding style that, if followed, will lead to highly readable, maintainable code.

1.1 The Need for Standards

Coding standards promote consistency. The greater the consistency, the easier it is to understand the code, which in turn results in code that is easier to develop, maintain, and reuse. This ultimately reduces the cost of application development.

Having software developed in multiple "dialects" of a language also impacts software development teams. The ability to use and integrate software components (here we specifically mean Java classes or collections of these classes, i.e. packages), can be greatly impacted by language usage.

Increasingly, the single highest cost of many organizations is the cost of custom software development. A common development practice, first starting with standards for language usage, style, organization, and naming coupled with a set of design guidelines, can move toward the promotion of re-use of software across disparate, but coordinated, development efforts leading to lower costs. This purpose of this document is to provide the programming standards only.

1.2 Code for Readability

The major cost of any software project is no longer the computer time or storage space; it is the development and maintenance labor. Also, problems will occur in production in all applications. Most likely the original developer will not be the initial individual supporting the software in a production environment. The readability of the program source code is paramount and management techniques like code reviews will focus attention on this area. Ways to improve readability include using meaningful member, method, and variable names. Look for opportunities to break up compound statements. Every developer should feel compelled to enforce the readability standard because there is a high likelihood he/she may be called upon to support or maintain someone else's software in production.

1.3 Code Consistently

Perhaps the only thing worse than failure to adhere to established project or enterprise coding conventions is the failure of a developer to follow his/her own conventions. At a minimum, each developer should adopt a consistent style in the absence of all other guidelines. If a developer lacks any particular style, then he/she should obtain source code from respected, mature developers and emulate conventions found there. The consistency of the coding style is crucial to the maintainability of the code. Specifically, the developer should pay attention to the organization of attributes and methods within a class, naming conventions, and indentation style. Where possible, the developer should use tools such as code formatters, or Integrated Development Environments (IDE) to maintain code consistency.

Also, when modifying and/or maintaining code previously developed by others, take special care to continue developing in the existing style rather than introduce multiple, and conflicting styles within the same file, module, package, or library. A developer should not intentionally introduce a new order where one already exists, except in the case where the previous effort is being completely rewritten.

1.4 Type Convention

There is a range of applicability and importance to any standardization effort. These standards have been divided into two major types, based upon importance, *Standards* and *Guidelines*. The subsequent sections are assumed to be standards unless explicitly noted or named as guidelines. The differences between these categories are explained below

1.4.1 Standards

A standard is subject to the highest level of enforcement. A standard can under no circumstances be violated. Standards will be stringently reviewed by not only the standards group, but by the development community as a whole. The failure to follow a standard is assumed to greatly increase software risk. Since standards are not subject to modification, they are worded as imperatives: Do this, Don't do that. This harshness is intended to remove all vagueness and subjectivity from the review process.

A programmer should be able to take this standards document and walk through his/her code, line by line, and anticipate the outcome of an upcoming code review.

1.4.2 Guidelines

A guideline is the next lower level of enforcement. The guideline is a convention that was accepted by the standards group, but one in which the group was aware of situations in which the guideline might need to be violated. All known, justifiable guideline violations should be noted using comments in the program. The guideline violation is to be noted during the software inspection process, to determine if indeed the violation is warranted.

It is also possible that the feature in question is complex enough that not all facets are fully understood; as such, the obvious shortcoming would be documented as a guideline, but until further evidence can be presented, all usage cannot be prohibited, or endorsed out of hand.

2. Standard Guidelines

2.1 General conventions

A consistent naming scheme for class members should be enforced throughout the entire system. The scheme should be based upon the member's scope, which has special meaning in object oriented languages.

Package naming conventions:

- Package naming should follow as closely as possible the recommendations set forth in sections 6.8.1 and 7.7 of the Java Language Specification (Gosling et. al, *Addison Wesley*). As such, all software created within CBOE will be organized within packages with the prefix com.cboe.
- All package names should be in mixed case with the first letter lowercase and the first letter of any subsequent words capitalized (e.g. com.cboe.businessServices.quoteService)

Class member naming conventions:

- All members (**except constructors)** should be in mixed case with a lowercase first letter and the first letters of any subsequent words capitalized. This includes class fields, local fields, formal parameter names, and method names.
- All constant (final) fields should be in uppercase with underscores separating the words. Do not use leading underscores for constants. (e.g. MAX_VALUES)
- Constructor names do not follow these conventions since they must be named exactly as the class name

Class and interface naming conventions:

- All class and interface names should be in mixed case with the first letter capitalized and the first letter of any subsequent words capitalized (e.g. QuoteService)

2.2 Use meaningful names.

The general guideline for variable, data member, and methods (functions) is to use as descriptive names as possible. Single letter names, such as a, c or p should be avoided because they provide little semantic

content and are very difficult to search for in a program file. The goal is readability. The only exceptions to this rule are: loop counters (i, j, k, etc.), streams (in, out), and exceptions (e).

2.3 Do not use abbreviations.

All compilers support variable names that are longer than eight characters. To repeat, the goal is readability. If proper object design is used, the actual name should not have to be typed that often, so a shorter name is not going to save much time during the programming phase. The longer more descriptive name may save considerable time during the maintenance life of a program.

The goal of no abbreviations must be balanced with that fact that names that are too long can obscure readability due to having language statements span multiple lines.

- a class named CommitteeRoster is preferred over ComRost
- committeRosterID is preferred over committeeRosterIdentification

If you must abbreviate, use the following Abbreviation Standards.

Full Name	Abbreviation
Description	Desc (except in DB, this is a reserved word)
Identifier	ID
Number	No
Information	Info
Calculation	Calc

2.4 The use of Hungarian notation shall be prohibited.

CBOE prohibits the use of Hungarian notation, whereby every name begins with the type. The proper method of naming a member variable is provided within this section. The reason why the Hungarian notation is prohibited is that the object-oriented approach provides a method for representing object types. The object's properties, such as its type can change without changing the identity or name of the object. Projects that do use Hungarian notation have often found that once a name is used widely, it is difficult to change the name when a type change is required. When this happens, the names that previously described an object's type accurately are not up to date and serve only to mislead the developer.

2.5 Importing classes from other packages

The programmer should only import classes that are used within the compilation unit (e.g. single source file). The programmer should avoid using import-on-demand declarations (e.g. import java.awt.*), as these tend to make it more difficult to identify the source package for an imported class.

If a name conflict occurs due to two or more packages declaring classes with identical names, the programmer will be forced to use fully qualified class names (e.g. com.cboe.collections.Vector), in place of the abbreviated class name.

Do not import a class more than once.

Programmers should not explicitly import the `java.lang` package as it is done automatically for each compilation unit.

The following table shows the recommended approach:

Recommended use of import statement	Not recommended
// java packages import java.util.Vector; import java.util.Enumeration;	import myPackage.BaseClass; import java.lang.*; // unnecessary

```
                                        import java.util.*;              // demand
// local packages
import mypackage.BaseClass;             import myPackage.BaseClass; // duplicate
```

2.6 Class Names

Classes should be named to accurately represent the entity that is being modeled. Class name lengths should be kept to a minimum without resorting to abbreviations or other such obscuring of meaning. It is assumed that names longer than eight characters for the filename and the class name are acceptable and are preferred to abbreviations.

- Class names should be singular rather than plural.
- Avoid using empty words in class naming such as 'Thing', 'Object', 'Class', etc.
- Words such as "Info" and "Data" when included in a class name should be avoided unless specifically included to denote abstract or vagueness.

Person is preferred to People as a class name. Object classes imply both data and function. A class created to model employees should be named Employee rather than EmployeeInfo or EmployeeData.

2.7 Interface Names

Interfaces should be named to accurately model the behavior that is being modeled. Interface names should use descriptive adjectives, such as Runnable, or Cloneable. Descriptive nouns such as Singleton, or DataInput, are also common. Interface names should not include any identifiers that attempt to indicate that it is and interface (e.g. MyServiceInterface, IMyService, MyServiceIF).

Interface names may also resemble class names when they are auto-generated from Interface Definition Language (IDL) specifications. An IDL compiler will generate a Java interface for each IDL interface definition. In this case, a corresponding implementation class will typically implement the interface and have "Impl" appended to the name of the class.

```
// Order.idl
interface Order {
};

// Order.java
public interface Order extends org.omg.CORBA.Object {
}

// OrderImpl.java
public class OrderImpl implements Order {
}
```

2.8 Data Members and Member Functions.

Class member function names should accurately describe the action the object is performing. Member function names should contain a verb that specifies the action or behavior, such as "load", or "initialize". Methods that are tests or provide state information should use "is" or "has" as part of their name. Do not include the class name as part of the function name.

person.getFirstName() is preferred to person.GetFirstName() or person.get_first_name().

2.8.1 Accessor / Modifier Methods

Accessor or Modifier functions should use the standard prefixes “get/set” and “is/set” as set forth in the Java Beans Programming “design patterns”. There are two patterns in particular worth emulating. For example, the Person class has a pair of accessors and modifiers for its “age” and “married” properties.

```
class Person {
     private  int age;                // “Age” property
     private  boolean marriedFlag;    // “Married” property

     public int getAge() {
          return age;
     }

     public void setAge(int anAge) {
          age = anAge;
     }

     public boolean isMarried() {
          return marriedFlag;
     }

     public void setMarried(boolean aBoolean) {
          marriedFlag = aBoolean;
     }
}
```

2.8.2 Initializing Data Members

Data members should be initialized within the classes’ constructor(s), rather than in the declaration of the data member. Static data members can be initialized upon declaration.

```
class Person {
     private  int age;                // “Age” property - no initialization
     private  boolean marriedFlag;    // “Married” property - no initialization

     private static int MIN_AGE = 16; // minimum age allowed with the application

     public Person() {
          age = MIN_AGE;
          marriedFlag = false;
     }
}
```

2.9 Style

Developers should use a coding style intended to enhance the readability of source code. In general, the developer must continually keep in mind that someone will be required to maintain and enhance the application in the future. This section describes some style conventions that have been used successfully on many projects.

2.9.1 Indentation and Spacing

Use extra white space and indentation to provide a very uniform simple coding style.

- Use 4 spaces for each indentation level. Due to variations between IDEs, do not use tabs.
- Provide white space around parentheses on the if, while, for, and do-while statements.
- White space should be used liberally to improve readability.
- Whenever possible, line up rows of attribute declarations
- Vertical whitespace (i.e. blank lines) can help in setting off blocks of code

```
boolean someMethod(boolean existsFlag, String name) {
     int     value  = 1;              // we indented 4 spaces
     boolean retVal = false;          // line up variable declarations
```

```
        if ( existsFlag == true ) {       // spaces around the parenthesis
             value = 2;                   // we indented another 4 spaces here
        }
    }

    protected int     numValues;          // lining up instance variables
    protected long    currentCount;
                                          // separate class (static) variables
    private static final boolean DEBUGGING = true;
    private static final String APP_NAME  = "Hello World";
```

2.9.2 Braces

Always use braces in your source code when employing any block (if/else, while, for, do-while, switch, try/catch/finally, static). The opening braces should be located at the end of the line that creates the block. All code contained within that block should then be indented the standard indentation distance (4 spaces). The training brace should be at the same level as the beginning of the block. An acceptable alternative style is to place the opening brace one the line following the line that creates the block, aligned with the beginning of the block. Regardless of which style is chosen, consistency is key..

```
//Although this is still legal in Java, it is a very bad example
// that shows how a maintenance programmer might be led astray
if ( someTest() )
    count = count + 1;
```

The following occurs quite frequently in a maintenance or enhancement situation, especially under hurried conditions:

```
if ( isSomethingOrAnother() )
    count = count + 1; // this line is part of the default block
    total = total + 1; // this line will always gets executed!!!
```

The proper way to prevent this trivial, but potentially very expensive problem from occurring is to always use braces.

```
if ( isSomethingOrAnother() ) {       // even if you only have 1 statement!
     count = count + 1;
}
```

2.9.3 Member organization

Java provides four protection levels to facilitate data and behavior hiding: public, protected, package, and private. Public specifies the interface for the world, protected specifies an additional interface for derived classes, package indicates an interface shared only with other classes in the same package, and private specifies implementation members for the class only. There are two popular conventions for organizing or grouping a class' members based on access control modifiers. In both cases, short comment blocks can assist in readability. (Note: Developers utilizing an IDE that manages source code internally (e.g. IBM VisualAge for Java) may have little or no control over code organization when the code is exported to text files).

Access level first, constructor/method/field second	**Member first, then by access level**
//----------	//----------
// public interface	// attributes
//----------	//----------
public attributes	*public attributes*
public constructors	*protected attributes*
public methods	*package attributes*
	private attributes
//----------	
// protected interface	//----------

```
//---------------------------------------------------------
protected attributes
protected constructors
protected methods

//---------------------------------------------------------
// package interface
//---------------------------------------------------------
package attributes
package constructors
package methods

//---------------------------------------------------------
// private implementation
//---------------------------------------------------------
private attributes
private constructors
private methods
```

```
// constructors
//---------------------------------------------------------
public constructors
protected constructors
package constructors
private constructors

//---------------------------------------------------------
// methods
//---------------------------------------------------------
public methods
protected methods
package methods
private methods
```

2.9.4 Constructor and method modifier placement

Access control modifiers as well as the other field and attribute modifiers will always be specified on the line including the signature of the member. This simplifies the signature and creates more readable code. For example:

```
public MyClass() {
}

public static final synchronized Foo getFoo() {
    . . .
}

protected abstract boolean isRunning() {
    . . .
}
```

3. Standard Documentation Section

All classes created at CBOE must be commented using the Javadoc standards. This section contains excerpts from Sun's Javadoc webpage.

3.1 Standard File Header

Each source file should begin with the a standard header identifying the source file name, the package, and a copyright notice:

```
//
// ------------------------------------------------------------------------
// Source file: com/cboe/businessServices/brokerService/BrokerImpl.java
//
// PACKAGE: com.cboe.businessServices.brokerService;
//
// ------------------------------------------------------------------------
// Copyright (c) 2000 The Chicago Board Options Exchange.  All Rights Reserved.
//
// ------------------------------------------------------------------------
```

3.2 Description for javadoc

The **javadoc** program parses the declarations and doc comments in Java source files and formats the public API into a set of HTML pages. Within doc comments, **javadoc** supports the use of special *doctags* to augment the API documentation. **javadoc** also supports standard HTML within doc comments. This is useful for code samples and for formatting text. Note that **javadoc** uses *.java* files not *.class* files. **Javadoc** reformats and displays all public and protected declarations for:

- Classes and Interfaces
- Methods
- Variables
- Exceptions Thrown
- Doc Comments

Java source files should include doc comments. Doc comments begin with /** and indicate text to be included automatically in generated documentation.

3.3 Standard HTML

You can embed standard html tags within a doc comment. However, don't use tags heading tags like <h1> or <hr>. Because **javadoc** creates an entire structured document and these structural tags interfere with the formatting of the generated document.

3.4 Special javadoc Tags

The program **javadoc** parses special tags that are recognized when they are embedded within an Java doc comment. These doc tags enable you to auto-generate a complete, well-formatted API from your source code. The tags start with an @. Tags must start at the beginning of a line. Keep tags with the same name together within a doc comment. For example, put all your @author tags together so **javadoc** can tell where the list ends.

3.4.1 Class Documentation Tags

`@see classname`
Adds a hyperlinked "See Also" entry to the class.

`@see fully-qualified-classname`
Adds a hyperlinked "See Also" entry to the class.

`@see fully-qualified-classname#method-name`
Adds a hyperlinked "See Also" entry to the method in the specified class.

`@version version-text`
Adds a "Version" entry. This is usually the date created, or some keys that are specific to your version control system.

`@author your-name`
Creates an "Author" entry. There can be multiple author tags. Each class must have at least one author tag. An example of a class comment:

```
/**
 * A class representing a window on the screen.
 * For example:
 * <pre>
 *    Window window = new Window(parent);
 *    window.show();
 * </pre>
 *
```

```
 * @see            awt.BaseWindow
 * @see            awt.Button
 * @version        1.2 12 Dec 1994
 * @author         Sami Shaio
 */

class Window extends BaseWindow {
        ...
}
```

3.4.2 Variable Documentation Tags

In addition to HTML text, variable comments can contain only the `@see` tag (see above).
An example of a variable comment:

```
/**
 * The X-coordinate of the window
 * @see window#1
 */
int x = 1263732;
```

3.4.3 Method Documentation Tags

Can contain `@see` tags, as well as:

`@author your-name`
Creates an "Author" entry. Author tags are generally not required for each method. However, if a method is created, or significantly modified by someone other than the principle author, a method should have am Author tag associated with it.

`@since version-text`
Adds a "Since" entry. This identifies the class version in which the method was introduced.

`@param parameter-name description...`
Adds a parameter to the "Parameters" section. The description may be continued on the next line.

`@return description...`
Adds a "Returns" section, which contains the description of the return value.

`@exception fully-qualified-class-name description...`
Adds a "Throws" entry, which contains the name of the exception that may be thrown by the method. The exception is linked to its class documentation. A @throws tag can also be used.
An example of a method comment:

```
/**
 * Return the character at the specified index. An index ranges
 * from <tt>0</tt> to <tt>length() - 1</tt>.
 * @param index      The index of the desired character
 * @return           The desired character
 * @exception        StringIndexOutOfRangeException When the index
 *                   is not in the range <tt>0</tt>> to <tt>length() - 1</tt>.
 * @since            version 2.3
 */

public char charAt(int index) throws StringIndexOutOfRangeException {
    ...
}
```

4. Protection Levels (Access Control)

Java provides four protection levels to facilitate data and behavior hiding: public, protected, package, and private. Public specifies the interface for the world, protected specifies an additional interface for derived classes and other classes in the package, package indicates an interface shared only with other classes in the same package, and private specifies implementation members for the class only. In general, it is good practice to:

- always declare attributes as private. Protected accessors should be used by subclasses.
- include only member functions in the public interface
- declare classes as protected to limit their visibility (coupling) outside the package
- always select and specify the protection level unless package access is desired

Typically, the public interface specifies one or more constructors and member functions necessary for fully describing the published behavior for the class. If you wish to share fields with derived classes, make all the fields private and provide protected accessors and modifiers.

The private region typically contains all data members and implementation level member functions. The latter are member functions that the class needs to modularize its implementation, but are irrelevant (and inaccessible) to the outside world.

4.1 Always Provide a Default Constructor

The compiler will generate a default public constructor if it is not specified in the class. It is bad programming practice to assume this and rely on this behavior even if it is well documented in the class. In most cases, this behavior is undesirable or it would have been explicitly provided by the author. To suppress the creation of an unwanted constructor, create at least one public, protected, package, or private constructor in the class. Note that an empty constructor (taking no arguments) is NOT required if the class provides at least one other constructor.

5. Standard Component Libraries

Wherever possible use standard components instead of custom made constructs. Classes are provided for Strings, Dates, Time, and miscellaneous collections and helper classes. Take some time to explore the existing class library APIs and refrain from building any components that you perceive to be missing without discussing the development with an experienced programmer. Chances are, most of the generic classes have been provided and are just named differently or located in different places than ones with which you are familiar.

- Standard utility classes are located in the com.cboe.util package.
- Common business domain (entity) classes are located in com.cboe.domain.
- Standard infrastructure frameworks and services are located in com.cboe.infrastructureServices.

6. Exception Handling

6.1 Background

Error handling is an extremely important part of all applications, yet may at times be overlooked or ignored until the last moment during development. There are many different ways in which to approach error handling, using error handling functions or exceptions are just a couple. CBOE will use the Java exception handling mechanism as the primary way to handle errors in all of its applications. This section provides some insight into the benefits of using exceptions and offers some guidelines for using and designing exception handling in CBOE applications.

6.2 Why use exceptions?

The use of exceptions was decided upon for the following reasons:

- Exceptions provide a very acceptable mechanism for reporting/recovering from errors which may easily be extended into an error processing framework.
- The use of exceptions provide an easy way to defer handling of the error to a "higher level" of processing, i.e. error propagation is made much simpler.
- Reduces the number of "if" tests in the code, making it easier to follow the normal path of execution
- The syntax for exceptions is part of the Java standard.

6.3 Underlying philosophy

Exceptions provide the programmer with a strongly-typed, independent system of error handling that is difficult to break by neglect. The typical error handling scheme requires that each and every programmer on the team observe and faithfully apply that scheme in every line of code produced. Failure to do so in even one line can result in a broken error processing chain (this is the neglect part), with catastrophic and usually quite colorful results. Exceptions allow programmers to bypass all but the most mundane errors, the ones they know and expect, and thus, can recover from. Also, the Java compiler directly enforces the use of exception handling by refusing to compile code that does not properly handle or promote exceptions.

6.4 Naming exception arguments

Because error handling is very common in Java coding, the use of the letter "e" for an exception is acceptable. For example:

```
try {
    Class c = Class.forName("java.util.Vector");
    Object o = c.newInstance();
}
catch ( ClassNotFoundException e ) {
}
catch ( IllegalAccessException e ) {
}
catch ( InstantiationException e ) {
}
```

7. Miscellaneous Java guidelines

The following is an unordered collection of unrelated guidelines concerning the development of Java code.

- Do not place more than one class definition in a single source file. This always leads to confusion since multiple class files will be generated from a single source file containing multiple classes. This is a guaranteed maintenance nightmare.
- In general, all methods (including constructors) should be kept to less than one full page if printed (i.e. < 70 lines). In fact, well-factored Java programs typically have methods that are less than 10 lines.
- Use a consistent wrapping technique for lines that exceed 72 characters long. This includes method signatures with one or more exceptions thrown, class declarations that implement one or more interfaces, etc.
- Always include the `default` case in `switch` statements to make it clear that that condition was considered. Avoid case statements altogether by exploiting polymorphism wherever possible.
- Optionally choose to end while, for, do/while, if/else, try/catch/finally blocks with comments indicating such (e.g. } // end while, } // end if, } // end try).
- End every source file with a single line comment indicating the end of the file (e.g. // EOF). This has proved to be useful in clearly defining the end of a source file when printing code as documentation. (Note: Not applicable with IBM VisualAge for Java)
- Avoid using the optional `this.` notation to refer to members (both fields and methods). If you follow the naming conventions presented in this document, there will be no confusion between formal parameters and class variables.
- Avoid using System.out for error or debug reporting. Use a Log utility class.
- Avoid hardcoded values that are likely to change. Use Properties.
- For each new model class (model-view-controller), consider implementing the `Cloneable`, `Copyable`, and/or `Serializable` interfaces.
- Do not put parentheses around return statements as in: `return (true);` The `return` keyword is not a method. Exceptions are allowed if calculations or complicated comparison are being performed in this statement.
- Avoid declaring variables already defined in an outer scope ("shadow" variables), which results in difficult to read code.

```
public MyClass() {
     private int value = 2;

     public Foo foo() {
          int value = 5; // already defined in class scope

          value = 6; // which value?
     }
}
```

8. Full Java Class Example

The following is an example of a class that conforms to the standards presented in this document. Notice the style, layout, and ordering.

```
//-----------------------------------------------------------------------
// FILE:    Foo.java
//
// PACKAGE: com.cboe.orderService
//
//-----------------------------------------------------------------------
// Copyright (c) 2000 The Chicago Board Options Exchange. All Rights Reserved.
//
// Standard confidential and proprietary information may be inserted
// here.
//-----------------------------------------------------------------------
```

This must be first "real" line of code.

```
package com.cboe.orderService;

// java packages
import java.util.Vector;

// local packages
import com.thirdparty.widgets.Dialog;
```

Always use **javadoc** comments for classes.

```
/**
 * This class can be used to obtain the Foo behavior;
 * <B>Use basic HTML tags if necessary</B>
 * <UL>
 * <LI> Lists sometimes are useful for enumerating points
 * </UL>
 *
 *
 * @author Joe Schmo
 * @version Wed Feb 21 10:57:45 CST 1997
 */
public class Foo       extends        Dialog
                       implements     Runnable,
                                      Cloneable {
```

Organize the class into suitable sections.

```
    //-------------------------------------------------------------
    // public interface
    //-------------------------------------------------------------
```

Use **javadoc** comments for methods, too.

```
    /**
     * singleton access for Foo
     * @return   single instance of Foo
     */
    public static final Foo getFoo() {
        if ( instance == null ) {
             instance = new Foo();
        }
        return instance;
    }


    /**
     * Accessor for SomeValue
     * @return   current value of SomeValue
     */
    public String getSomeValue() {
        return someValue;
    }

    /**
     * Modifier for SomeValue
     * @param  newValue  Will replace current SomeValue
     */
    public void setSomeValue(String newValue) {
        someValue = newValue;
    }
```

```
    //------------------------------------------------------------
    // protected interface
    //------------------------------------------------------------

    /**
     * protected constructor
     */
    protected Foo() {
        someValue = new String();
        isAlive = false;
        listeners = new Vector();
    }

    //------------------------------------------------------------
    // package interface
    //------------------------------------------------------------

    //------------------------------------------------------------
    // private implementation
    //------------------------------------------------------------

    // Order of precedence should be instance, then class (i.e. static)

    // variables

    private String someValue; // description of someValue
    private boolean isAlive;  // description of isAlive
    private Vector listeners; // description of listeners

    private static final boolean DEBUGGING = true; // description of DEBUGGING
    private static Foo instance = new Foo(); // description of instance


    /**
     *This is an internal method used for calculating something
     */
    private void calcSomething() {
        . . .
    }
}

// EOF
```

This helps when printing out the source code.

Appendix A: SBT Programming Guidelines

A.1 Packaging

The Screen-Based Trading System is a component-based, multi-tier application. The following packages have been developed to increase the modularity of the system, thus increasing the potential for reuse, and minimizing the inter-package dependencies.

- **com.cboe.idl**

This package contains ALL generated IDL.

example: com.cboe.idl.businessServices.OrderHandlingService

- **com.cboe.interfaces.domain**

This package contains all the Java Service Interfaces and the interfaces for the Home Classes for all Entity objects.

example: com.cboe.interfaces.domain.Order
example: com.cboe.interfaces.domain.OrderHome

- **com.cboe.interfaces.businessServices**

This package contains all the Java Service Interfaces and the interfaces for the Home Classes for all Business Services objects. Any service that is defined as an IDL interface will have an interface defined in this package which extends the corresponding IDL interface in the com.cboe.idl package.

example: com.cboe.interfaces.businessServices.OrderHandlingServiceHome
example: com.cboe.interfaces.businessServices.OrderHandlingService

- **com.cboe.interfaces.events**

This package contains all the Java Consumer Interfaces for the Consumer (Callback) IDL interfaces. Each IDL interface will have an interface defined in this package which extends the corresponding IDL operations interface in the com.cboe.idl.consumers package.

example: com.cboe.interfaces.events.OrderFillReportConsumer

- **com.cboe.interfaces.internalEvents**

This package contains all the Java Consumer Interfaces for the Consumer (Callback) IDL interfaces. Each IDL interface will have an interface defined in this package which extends the corresponding IDL operations interface in the com.cboe.idl.consumers package. These consumers are restricted for use between internal components.

example: com.cboe.interfacesinternalEvents.BestOfTheRestConsumer

- **com.cboe.proxy.businessServices**

This package contains the remote implementations for any business service that supports a remote impl.

example: com.cboe.proxy.businessServices.RemoteProductQueryServiceImpl

- **com.cboe.domain**

This package contains the implementations of all domain (entity) objects.

example: com.cboe.domain.product.ProductImpl
example: com.cboe.domain.bestQuote.BestQuoteImpl

- **com.cboe.businessServices**

This package contains the null implementations and the actual service implementations .

example: com.cboe.businessServices.brokerService.BrokerServiceHomeImpl
example: com.cboe.businessServices.brokerService.BrokerServiceImpl

- **com.cboe.internalBusinessServices**

This package contains the null implementations and the actual service implementations of internal business services.

example: com.cboe.internalBusinessServices.productService.ProductMaintenanceServiceImp.

- **com.cboe.externalIntegrationServices**

This package contains the null implementations and the actual service implementations of external integration services.

example: com.cboe.externalIntegrationServices.tpfAdapter.order.TPFOrderService

- **com.cboe.application**

This package contains the components constituting the Client Application Server (CAS).

example: com.cboe.application.order.OrderManager

- **com.cboe.internalApplication**

This package contains the null implementations and the actual service implementations of internal applications.

example: com.cboe.internalApplications.cas.SystemAdminUserAccessImpl
example: com.cboe.internalApplications.product.ProductMaintenanceServiceImpl

- **com.cboe.presentation**

This package contains the graphical user interface applications.

example: com.cboe.presentation.marketDisplay.MarketDisplayWindow

- **com.cboe.internalPresentation**

This package contains the graphical user interface applications for internal use.

example: com.cboe.internalPresentation.sysAdmin.TradingSessionWindow

- **com.cboe.infrastructureServices**

This package contains the null implementations and the facade implementations all infrastructure services and frameworks.

example: com.cboe.infrastructureServices.foundationFramework.FoundationFramework
example: com.cboe.infrastructureServices.traderService.TraderService

- **com.cboe**

These packages contain actual implementations of all infrastructure services and frameworks.

example: com.cboe.directoryService.LookupImpl
example: com.cboe.securityService.SecurityServiceImpl
example: com.cboe.Orb.ORB

- **com.cboe.util**

This package contains basic framework, utility, and helper classes

example: com.cboe.util.Price
example: com.cboe.util.event.EventChannelListener

A.2 Proxies

- All proxy implementations will be located in **com.cboe.proxy.businessServices, com.cboe.proxy.internalBusinessServices, and com.cboe.proxy.externalIntegrationServices.**.
- Proxy implementation classes will be named **<Service>Proxy** (e.g. ProductQueryServiceProxy). If there will be more than one type of proxy, insert a descriptor in front of the Proxy tag (e.g. ProductQueryServiceCachedProxy, OrderHandlingServiceDispatchingProxy)
- Home implementations for proxy classes will be named **<Service>HomeProxyImpl** (e.g.

ProductQueryServiceHomeProxyImpl).

- Null implementations will be named **<Service>NullImpl**. If there is more than one type of null implementation, insert a descriptor in from of the Null tag.
- If possible, use proxy home implementations for more than a single proxy implementation. For example, ProductQueryServiceHomeProxyImpl should be able to create ProductQueryServiceProxy and ProductQueryServiceCachedProxy instances. The Home implementation should be able to obtain the implementation class for the proxy from the Foundation Framework. If there are significant differences between the proxy classes, it may be necessary to have a custom Home implementation per proxy. Try to avoid this if possible.
- Since Remote classes are actually proxies, they should be renamed to reflect the Proxy naming pattern.