Coding Standards

For

Java Projects



HTC Global Services

3270 W. Big Beaver Road,

Troy, MI 48084

Phone: (248) 786 2500

Unit 31, SDF II, Phase II, MEPZ

Chennai - 600 045. India.

Phone: (44) 2262 3522/4783/5044

|  |  |  |  |
| --- | --- | --- | --- |
| Document Number: | DEPT\_ JAVA\PD02\JAVA\_CODE\_STD | Version #: | 1.4 |
| Author | Balaji Desikan | Signature: | Balaji Desikan |
| Approved by | Chandrasekhar M | Signature: | Chandrasekhar M |
| Location of soft copy: | \\SVN | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Template Number: | QLTY\QMS\TPL\PRG\_SPC | Tpl Ver: | 1.4 | Rel.Date | Feb\_13\_2008 |

Amendment Record

|  |  |  |  |
| --- | --- | --- | --- |
| Version # | Description | Page # | Date |
| 1.0 | Base lined as per DCF No. JAVA\_CODE\_STD/0.0a/1/PD03 | All | May\_02\_2006 |
| 1.1 | Updating as per latest technologies |  | Mar\_04\_2009 |
| 1.2 | Added Coding standards for Database |  | Mar\_05\_2009 |
| 1.3 | Cross-site Scripting (XSS) |  | Mar 10 2010 |
| 1.4 | Baselined as per DCF No. Java Standards/1.3a/1/PD03 | All | Sep\_15\_2014 |
| 1.5 | Added OWASP ref documents | 16.7 | Aug-18-2015 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

[Java Projects 1](#_Toc382840209)

[Amendment Record 2](#_Toc382840210)

[1. Introduction 3](#_Toc382840211)

[1.1. Why Have Code Conventions 3](#_Toc382840212)

[2. Java Coding Standards 3](#_Toc382840213)

[2.1. File Names 3](#_Toc382840214)

[2.1.1. File Suffixes 3](#_Toc382840215)

[2.1.2. File Type Suffix 3](#_Toc382840216)

[3. File Organization 3](#_Toc382840217)

[3.1. Java Source Files 3](#_Toc382840218)

[3.1.1. Beginning Comments 3](#_Toc382840219)

[3.1.2. Package and Import Statements 3](#_Toc382840220)

[3.1.3. Class and Interface Declarations 3](#_Toc382840221)

[3.1.4. Part of Class/Interface Declaration Notes 3](#_Toc382840222)

[3.1.5. Class / package / directory naming 3](#_Toc382840223)

[4. Indentation 3](#_Toc382840224)

[4.1. Line Length 3](#_Toc382840225)

[4.2. Wrapping Lines 3](#_Toc382840226)

[5. Comments 3](#_Toc382840227)

[5.1. Implementation Comment Formats 3](#_Toc382840228)

[5.1.1. Block Comments 3](#_Toc382840229)

[5.1.2. Single-Line Comments 3](#_Toc382840230)

[5.1.3. Trailing Comments 3](#_Toc382840231)

[5.1.4. End-Of-Line Comments 3](#_Toc382840232)

[5.2. Documentation Comments 3](#_Toc382840233)

[5.2.1. Javadoc and Comments 3](#_Toc382840234)

[6. Declarations 3](#_Toc382840235)

[6.1. Number Per Line 3](#_Toc382840236)

[6.2. Initialization 3](#_Toc382840237)

[6.3. Placement 3](#_Toc382840238)

[6.4. Class and Interface Declarations 3](#_Toc382840239)

[7. Statements 3](#_Toc382840240)

[7.1. Simple Statements 3](#_Toc382840241)

[7.2. Compound Statements 3](#_Toc382840242)

[7.3. return Statements 3](#_Toc382840243)

[7.4. if, if-else, if else-if else Statements 3](#_Toc382840244)

[7.5. for Statements 3](#_Toc382840245)

[7.6. while Statements 3](#_Toc382840246)

[7.7. do-while Statements 3](#_Toc382840247)

[7.8. switch Statements 3](#_Toc382840248)

[7.9. try-catch Statements 3](#_Toc382840249)

[8. White Space 3](#_Toc382840250)

[8.1. Blank Lines 3](#_Toc382840251)

[8.2. Blank Spaces 3](#_Toc382840252)

[9. Naming Conventions 3](#_Toc382840253)

[10. Programming Practices 3](#_Toc382840254)

[10.1. Providing Access to Instance and Class Variables 3](#_Toc382840255)

[10.2. Referring to Class Variables and Methods 3](#_Toc382840256)

[10.3. Constants 3](#_Toc382840257)

[10.4. Variable Assignments 3](#_Toc382840258)

[10.5. Miscellaneous Practices 3](#_Toc382840259)

[10.5.1. Parentheses 3](#_Toc382840260)

[10.5.2. Returning Values 3](#_Toc382840261)

[10.5.3. Expressions before ‘?’ in the Conditional Operator 3](#_Toc382840262)

[10.5.4. Special Comments 3](#_Toc382840263)

[11. Code Examples 3](#_Toc382840264)

[11.1. Java Source File Example 3](#_Toc382840265)

[12. Coding Checklist: 3](#_Toc382840266)

[12.1. Variable Naming Convention: 3](#_Toc382840267)

[12.1.1. Naming conventions for Java Collections: 3](#_Toc382840268)

[12.2. Method Naming Convention: 3](#_Toc382840269)

[12.3. Check for spelling mistakes: 3](#_Toc382840270)

[12.4. Avoid duplication of Codes 3](#_Toc382840271)

[12.5. Provide appropriate comments for each method & class 3](#_Toc382840272)

[12.6. Separate Different functionality’s into different methods 3](#_Toc382840273)

[12.7. No method should exceed beyond 100 lines 3](#_Toc382840274)

[12.8. Identify the common codes and move them to common files for usage in other functions 3](#_Toc382840275)

[12.9. Always try to make the code generic for multiple usage 3](#_Toc382840276)

[12.10. Tune your queries 3](#_Toc382840277)

[12.11. Avoid unnecessary global variables 3](#_Toc382840278)

[12.12. Initialize objects while declaration to prevent Null Pointer Exception. 3](#_Toc382840279)

[12.13. Use For, While Loops effectively 3](#_Toc382840280)

[12.14. Improvise the habit of taking the latest code frequently from the common repository 3](#_Toc382840281)

[12.15. Use Proper exception handling mechanism 3](#_Toc382840282)

[12.16. Align the codes as you write them 3](#_Toc382840283)

[12.17. Use Pleasing words to report any error messages 3](#_Toc382840284)

[12.18. Make sure the XML Data is handled appropriately 3](#_Toc382840285)

[12.19. Log in-sensitive and required Data 3](#_Toc382840286)

[12.20. Try to make use of the common codes available in the project without over ridding 3](#_Toc382840287)

[12.21. Make sure the file names all are in lower case except that of Java Files. 3](#_Toc382840288)

[12.22. Try to use project level methods for displaying messages / confirmation 3](#_Toc382840289)

[12.23. Avoid retrieving data from Session / Request many times . instead get them once and use them where ever required 3](#_Toc382840290)

[12.24. Try to keep the logic as simple as it can be 3](#_Toc382840291)

[12.25. Use a separate Java script files for writing scripts 3](#_Toc382840292)

[12.26. Use String Buffer for concatenating Strings 3](#_Toc382840293)

[12.27. Avoid hardCoding 3](#_Toc382840294)

[12.28. Implement toString method 3](#_Toc382840295)

[12.29. Use Object Oriented Concepts effectively 3](#_Toc382840296)

[12.29.1. Logging 3](#_Toc382840297)

[12.29.2. Jars 3](#_Toc382840298)

[12.29.3. Java Code 3](#_Toc382840299)

[12.29.4. JUnit Tests 3](#_Toc382840300)

[12.29.5. Appendix A: Java and Javadoc Coding Conventions 3](#_Toc382840301)

[12.29.6. Appendix B: Examples for Development Standards Items 3](#_Toc382840302)

[13. Hibernate Standards for Use with Java Projects 3](#_Toc382840303)

[13.1. Standards - Currently Effective 3](#_Toc382840304)

[13.1.1. Configuration 3](#_Toc382840305)

[13.1.2. Session Management 3](#_Toc382840306)

[13.1.3. Class/Entity Mapping 3](#_Toc382840307)

[13.1.4. Implementation 3](#_Toc382840308)

[13.2. Best Practices 3](#_Toc382840309)

[13.2.1. Configuration 3](#_Toc382840310)

[13.2.2. Session Management 3](#_Toc382840311)

[13.2.3. Class/Entity Mapping 3](#_Toc382840312)

[13.2.4. Implementation 3](#_Toc382840313)

[14. EJB 3.0 3](#_Toc382840314)

[15. Database Independency Rules 3](#_Toc382840315)

[15.1.1. JDBC\_001: Do Not Hard Code A JDBC Driver Class Name 3](#_Toc382840316)

[15.1.2. JDBC\_002: Do Not Hard Code A JDCB Connection URL 3](#_Toc382840317)

[15.1.3. JDBC\_003: Do Not Use A Driver Managment Method Of java.sql.DriverManager 3](#_Toc382840318)

[15.1.4. JDBC\_004: Do Not Import A JDBC Vendor Specific Class 3](#_Toc382840319)

[15.1.5. JDBC\_005: Close Connection, Statement And ResultSet 3](#_Toc382840320)

[15.1.6. JDBC\_006: Close JDBC Resources In The Correct Order 3](#_Toc382840321)

[15.1.7. JDBC\_007: Use PreparedStatement Instead Of Statement When Possible 3](#_Toc382840322)

[15.1.8. JDBC\_008: Check The Return value Of A Navigation Method Of ResultSet 3](#_Toc382840323)

[15.1.9. JDBC\_009: Use Standard SQL Only 3](#_Toc382840324)

[15.1.10. JDBC\_010: Check For A Nested SQLException 3](#_Toc382840325)

[15.1.11. JDBC\_011: Check For A SQLWarning 3](#_Toc382840326)

[16. Best Practices to Prevent Vulnerability Attacks 3](#_Toc382840327)

[16.1. SQL Injection 3](#_Toc382840328)

[ Principle of Least Privilege 3](#_Toc382840329)

[ Validate All Input 3](#_Toc382840330)

[ Avoid Dynamic SQL 3](#_Toc382840331)

[ Use Double Quotes 3](#_Toc382840332)

[16.2. Buffer overflow 3](#_Toc382840333)

[16.3. Parameter Tampering 3](#_Toc382840334)

[16.4. Cookie Poisoning 3](#_Toc382840335)

[16.5. Cross Site Scripting or XSS 3](#_Toc382840336)

[16.6. Session hijacking 3](#_Toc382840337)

# Introduction

## Why Have Code Conventions

Code conventions are important to programmers for a number of reasons:

• 80% of the lifetime cost of a piece of software goes to maintenance.

• Hardly any software is maintained for its whole life by the original author.

• Code conventions improve the readability of the software, allowing engineers to

Understand new code more quickly and thoroughly.

• If you ship your source code as a product, you need to make sure it is as well packaged and clean as

any other product you create.

For the conventions to work, every person writing software must conform to the code conventions.

# Java Coding Standards

## File Names

This section lists commonly used file suffixes and names.

### File Suffixes

Java Software uses the following file suffixes:

### File Type Suffix

* Java source .java
* Java byte code .class

# File Organization

* A file consists of sections that should be separated by blank lines and an optional comment
* Identifying each section.
* Files longer than 2000 lines are cumbersome and should be avoided.
* For an example of a Java program properly formatted, see “Java Source File Example”

## Java Source Files

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file.

Java source files have the following ordering:

* Beginning comments (see “Beginning Comments” on page 2)
* Package and Import statements
* Class and interface declarations (see “Class and Interface Declarations” on page

### Beginning Comments

All source files should begin with a c-style comment that lists the class name, version

Information, dates, and copyright notice:

/\*

\* Class name

\*

\* Version information

\*

\* Date

\*

\* Copyright notice

\*/

### Package and Import Statements

The first non-comment line of most Java source files is a package statement. After that,

import statements can follow. For example:

package java.awt;

import java.awt.peer.CanvasPeer;

### Class and Interface Declarations

The following table describes the parts of a class or interface declaration, in the order that they should appear. See “Java Source File Example” for an example that includes comments.

### Part of Class/Interface Declaration Notes

* Class/interface documentation comment (/\*\*...\*/) See “Documentation Comments” for information on what should be in this comment.
* Class or interface statement
* Class/interface implementation comment (/\*...\*/), if necessary. This comment should contain any class-wide or interface-wide information that wasn’t appropriate for the class/interface documentation comment.
* Class (static) variables First the public class variables, then the protected, then package level (no access modifier), and then the private.
* Instance variables First public, then protected, then package level (no access modifier), and then private.
* Methods These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal isto make reading and understanding the code easier.

|  |  |
| --- | --- |
| Class / package / directory naming | |
| 3.1.5.1 | For simple interface/implementation pairs, interfaces are named [descriptive name], implementations are named [interface name]Impl. This is as opposed to the style of I[descriptive name] for interfaces and [descriptive name] for implementations. |
| 3.1.5.2 | JUnit test classes should be named XxxTest. If devoted mainly to testing a particular class, name it [class being tested]Test. If testing a class which has an interface and a single implementation, name it [interface being tested]Test. Only if testing multiple implementations of a single interface should it be named [implementation being tested]Test. For example, a class to test the PaymentDAOImpl implementation of the PaymentDAO interface should be called PaymentDAOTest, not PaymentDAOImplTest (unless there is more than one implementation of the PaymentDAO interface). |
| 3.1.5.3 | All JUnit test classes which should be run automatically as part of a continuous build process (e.g. Cruise Control) should be included in a test suite. Suites should be named XxxTestSuite. Cruise Control will be configured to automatically run all XxxTestSuite classes with JUnit. [We will start doing this now even though we don’t have a Cruise Control build yet; developers can manually run the Test Suites before merging. Search for All\*TestSuite.] |
| 3.1.5.4 | Classes which contain only constants should be named XxxConstants, for example “FileProcessingConstants”, not just “Constants”. |
| 3.1.5.5 | The source directories within any Java project are named /src and /test. For an ejb project, it should also contain directory /ejbModule which contains the META-INF directory only, and is where RAD should be set to generate the ejb stub and implementation code to. For a web project, it should also contain directory /web which contains the META-INF and WEB-INF directories, jsp’s and so on. |
| 3.1.5.6 | Projects (except web projects) should be set to compile to /bin (not /build/classes). Web projects can be left to compile to the default /web/WEB-INF/classes.  Note that when we get an Ant/Maven build in place, it may use /build, so we want that separate from what RAD uses. Also, the Ant/Maven build will NOT include the /test directory tree in JAR files used in other projects, although of course /test will be used for JUnit tests within that particular project. |
| 3.1.5.7 | See the separate document PackageOrganizationAndNaming.doc for package naming standards. |

# Indentation

Four spaces should be used as the unit of indentation. The exact construction of the indentation

(spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4).

## Line Length

Avoid lines longer than 80 characters, since they’re not handled well by many terminals and tools.

Note: Examples for use in documentation should have a shorter line length—generally not more than 70 characters.

## Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

* Break after a comma.
* Break before an operator.
* Prefer higher-level breaks to lower-level breaks.
* Align the new line with the beginning of the expression at the same level on the previous line.
* If the above rules lead to confusing code or to code that’s squished up against the right margin, just indent 8 spaces instead.

Here are some examples of breaking method calls:

someMethod(long Expression1, long Expression2, long Expression3,

long Expression4, long Expression5);

var = someMethod1(long Expression1, someMethod2(long Expression2,

long Expression3));

Following are two examples of breaking an arithmetic expression. The first is preferred, since

the break occurs outside the parenthesized expression, which is at a higher level.

longName1 = longName2 \* (longName3 + longName4 - longName5)+ 4 \* longname6;

//CONVENTIONAL INDENTATION

someMethod(int anArg, Object anotherArg, String yetAnotherArg,

Object andStillAnother) {

...

}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS

private static synchronized horkingLongMethodName(int anArg,

Object anotherArg, String yetAnotherArg,

Object andStillAnother) {

...

}

Line wrapping for if statements should generally use the 8-space rule, since conventional (4

space) indentation makes seeing the body difficult. For example:

//DON’T USE THIS INDENTATION

if ((condition1 && condition2)

|| (condition3 && condition4)

||!(condition5 && condition6)) { //BAD WRAPS

doSomethingAboutIt(); //MAKE THIS LINE EASY TO MISS

}

//USE THIS INDENTATION INSTEAD

if ((condition1 && condition2)

|| (condition3 && condition4)

||!(condition5 && condition6)) {

doSomethingAboutIt();

}

//OR USE THIS

if ((condition1 && condition2) || (condition3 && condition4)

||!(condition5 && condition6)) {

doSomethingAboutIt();

}

Here are three acceptable ways to format ternary expressions:

alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta

: gamma;

alpha = (aLongBooleanExpression)

? beta

: gamma;

# Comments

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which are delimited by /\*...\*/, and //. Documentation comments (known as “doc comments”) are Java-only, and are delimited by /\*\*...\*/. Doc comments can be extracted to HTML files

using the javadoc tool.

Implementation comments are means for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective to be read by developers who might not necessarily have the source code at hand.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Discussion of nontrivial or non obvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

Note: The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Comments should not be enclosed in large boxes drawn with asterisks or other characters.

Comments should never include special characters such as form-feed and backspace.

## Implementation Comment Formats

Programs can have four styles of implementation comments: block, single-line, trailing and end-of-line.

### Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A blank line to set it apart from the rest of the code should precede a block comment.

/\*

\* Here is a block comment.

\*/

Block comments can start with /\*-, which is recognized by indent(1) as the beginning of a block comment that should not be reformatted. Example:

/\*

\* Here is a block comment with some very special

\* formatting that I want indent(1) to ignore.

\*

\* one

\* two

\* three

\*/

Note: If you don’t use indent(1), you don’t have to use /\*- in your code or make any other

concessions to the possibility that someone else might run indent(1) on your code.

See also “Documentation Comments” .

### Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can’t be written in a single line, it should follow the block comment format (see section 5.1.1). A blank line should precede a single-line comment. Here’s an example of a single-line comment in Java code:

if (condition) {

/\* Handle the condition. \*/

...

}

### Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

Here’s an example of a trailing comment in Java code:

if (a == 2) {

return TRUE; /\* special case \*/

} else {

return isPrime(a); /\* works only for odd a \*/

}

### End-Of-Line Comments

The // comment delimiter can comment out a complete line or only a partial line. It shouldn’t be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code. Examples of all three styles follow:

if (foo > 1) {

// Do a double-flip.

...

}

else{

return false; // Explain why here.

}

//if (bar > 1) {

//

// // Do a triple-flip.

// ...

//}

//else{

// return false;

//}

## Documentation Comments

Note: See “Java Source File Example” on page 18 for examples of the comment formats described here.

For further details, see “How to Write Doc Comments for Javadoc” which includes

information on the doc comment tags (@return, @param, @see):

<http://java.sun.com/products/jdk/javadoc/writingdoccomments.html>

For further details about doc comments and javadoc, see the javadoc home page at:

<http://java.sun.com/products/jdk/javadoc/>

Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters /\*\*...\*/, with one comment per class, interface, or member. This comment should appear just before the declaration:

/\*\*

\* The Example class provides ...

\*/

public class Example { ...

Notice that top-level classes and interfaces are not indented, while their members are. The first line of doc comment (/\*\*) for classes and interfaces is not indented; subsequent doc comment lines each have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have 4 spaces for the first doc comment line and 5 spaces thereafter.

If you need to give information about a class, interface, variable, or method that isn’t appropriate for documentation, use an implementation block comment (see section 5.1.1) or single-line (see section 5.1.2) comment immediately *after* the declaration. For example, details about the implementation of a class should go in in such an implementation block comment *following* the class statement, not in the class doc comment. Doc comments should not be positioned inside a method or constructor definition block, because Java associates documentation comments with the first declaration *after* the comment.

|  |  |
| --- | --- |
| Javadoc and Comments | |
| 5.2.1.1 | Write class-level javadoc which states the purpose of the class. |
| 5.2.1.2 | Write method-level javadoc which states the purpose of the method and identifies parameters and output. You can use any combination of initial comment, @param(s) and @return which accomplishes this goal. For example, very simple methods can be described in a comment alone, whereas more complicated ones may require a comment plus @param(s) and @return. |
| 5.2.1.3 | For List, Set or any other collection type parameters and/or return values for a method, specify in the javadoc what class the elements of the collection are (as well as how they are used, per 1.020). |
| 5.2.1.4 | Substantive properties exposed by getters and setters, for domain objects and others, should be javadoc’ed at the instance variable underlying the property, and then this javadoc should be propagated automatically to the getter and setter methods using the Beaniac plug-in.  Setters whose only function is to serve as an entry point for Spring injection, however, should be javadoc’ed as:  /\*\* IOC setter \*/  and those instance variables simply grouped together and preceded with a single //Following are injected by Spring comment. |
| 5.2.1.5 | Interfaces should be javadoc’ed the same as an implementing class, not left undocumented. In fact if you wish the interface can contain the primary documentation and the implementing class refer to the interface methods being implemented (setting eclipse preferences does this automatically). |
| 5.2.1.6 | Javadoc should not simply restate the name being explained; that is useless. |
| 5.2.1.7 | The following copyright notice should appear in the javadoc at the top of the source file:  \* Copyright (C) 1994-2007 HTC. All rights reserved. (setting eclipse preferences does this automatically for new files.) |
| 5.2.1.8 | Common utility methods should describe the results for boundary input conditions rather than leaving these to chance (and unit tests should verify them). For example if an encode() method takes a byte array, in addition to of course describing the normal behavior, it should describe the results when the input is null, and when it is zero length. The purpose of this is so that common methods’ behavior is fully defined and hence actually usable without regard to the specific implementation code. |
| 5.2.1.9 | Use in-line comments and white space to clarify and explain the purpose of code chunks within methods and elsewhere. Comments should convey the intent of the code, not reiterate the syntax. |
| 5.2.1.10 | Javadoc should not contain any invalid tags, misspelled or non-existent parameters, broken links, or other syntax problems. Note that RAD automatically highlights these problems with a yellow Warning. @see is for a separate line; for inline references use {@link }. Use HTML tags as appropriate, especially <code> for programmatic names. |
| 5.2.1.11 | See Appendix A of this document, Java and Javadoc Coding Conventions, for additional notes on standard Javadoc mechanics. |
| 5.2.1.12 | When implementing an interface or an abstract method, or when overriding a method, eclipse preferences is set to insert a comment like the following:  /\*  \* (non-Javadoc so parent javadoc displays, assuming no changes needed)  \* @see com.htc.eis.epg.domain.junit.MyInterface#myMethod()  \*/  As it says, this is not javadoc as it has just a “/\*” rather than “/\*\*” to start. The reason is also stated: Per the javadoc standard, an implemented/overridden method will automatically display the javadoc of the defining class if there is none in the overriding class. Therefore this java comment may be left as is assuming you are simply implementing the interface/abstract method and the parent javadoc is accurate (if there is no parent javadoc, or it is incomplete or wrong, then write or update the parent javadoc also). However, if you are overriding a non-abstract method and/or if the parent javadoc is vague by necessity, then you must change this to a real javadoc comment (add an “\*” to “/\*”), delete the “non-Javadoc” line, leave the @see line, and document whatever about your implementation is different than the parent javadoc. |

# Declarations

## Number Per Line

One declaration per line is recommended since it encourages commenting. In other words,

int level; // indentation level

int size; // size of table

is preferred over

int level, size;

Do not put different types on the same line. Example:

int foo, fooarray[]; //WRONG!

Note: The examples above use one space between the type and the identifier. Another acceptable alternative is to use tabs, e.g.:

int level; // indentation level

int size; // size of table

Object currentEntry; // currently selected table entry

## Initialization

Try to initialize local variables where they’re declared. The only reason not to initialize a variable where it’s declared is if the initial value depends on some computation occurring first.

## Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces “{” and “}”.) Don’t wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

void myMethod() {

int int1 = 0; // beginning of method block

if (condition) {

int int2 = 0; // beginning of "if" block

...

}

}

The one exception to the rule is indexes of for loops, which in Java can be declared in the for statement:

for (int i = 0; i < maxLoops; i++) { ... }

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

int count;

...

myMethod() {

if (condition) {

int count; // AVOID!

...

}

...

}

## Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

• No space between a method name and the parenthesis “(“ starting its parameter list

• Open brace “{” appears at the end of the same line as the declaration statement

• Closing brace “}” starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the “}” should appear immediately after the“{“

class Sample extends Object {

int ivar1;

int ivar2;

Sample(int i, int j) {

ivar1 = i;

ivar2 = j;

}

int emptyMethod() {}

...

}

• Methods are separated by a blank line

# Statements

## Simple Statements

Each line should contain at most one statement. Example:

argv++; // Correct

argc++; // Correct

argv++; argc--; // AVOID!

## Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces “{statements}”. See the following sections for examples.

• The enclosed statements should be indented one more level than the compound statement.

• The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.

• Braces are used around all statements, even single statements, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

## return Statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

return;

return myDisk.size();

return (size ? size : defaultSize);

## if, if-else, if else-if else Statements

The if-else class of statements should have the following form:

if ( condition) {

statements;

}

if ( condition) {

statements;

} else {

statements;

}

if ( condition) {

statements;

} else if ( condition) {

statements;

} else {

statements;

}

Note: if statements always use braces {}. Avoid the following error-prone form:

if ( condition) //AVOID! THIS OMITS THE BRACES {}!

statement;

## for Statements

A for statement should have the following form:

for ( initialization; condition; update) {

statements;

}

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

for ( initialization; condition; update);

When using the comma operator in the initialization or update clause of a for statement, avoid the complexity of using more than three variables. If needed, use separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause).

## while Statements

A while statement should have the following form:

while ( condition) {

statements;

}

An empty while statement should have the following form:

while ( condition);

## do-while Statements

A do-while statement should have the following form:

do {

statements;

} while ( condition);

## switch Statements

A switch statement should have the following form:

switch ( condition) {

case ABC:

statements;

/\* falls through \*/

case DEF:

statements;

break;

case XYZ:

statements;

break;

default:

statements;

break;

}

Every time a case falls through (doesn’t include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the /\* falls through \*/ comment.

Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

## try-catch Statements

A try-catch statement should have the following format:

try {

statements;

} catch (ExceptionClass e) {

statements;

}

A try-catch statement may also be followed by finally, which executes regardless of whether or not the try block has completed successfully.

try {

statements;

} catch (ExceptionClass e) {

statements;

} finally {

statements;

}

# White Space

## Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:

• Between sections of a source file

• Between class and interface definitions One blank line should always be used in the following circumstances:

• Between methods

• Between the local variables in a method and its first statement

• Before a block (see section 5.1.1) or single-line (see section 5.1.2) comment

• Between logical sections inside a method to improve readability

## Blank Spaces

Blank spaces should be used in the following circumstances:

• A keyword followed by a parenthesis should be separated by a space. Example:

while (true) {

...

}

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

• A blank space should appear after commas in argument lists.

• All binary operators except . should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment (“++”), and decrement (“--”) from their operands. Example:

a += c + d;

a = (a + b) / (c \* d);

while (d++ = s++) {

n++;

}

prints("size is " + foo + "\n");

• The expressions in a for statement should be separated by blank spaces. Example: for (expr1; expr2; expr3)

• Casts should be followed by a blank space. Examples:

myMethod((byte) aNum, (Object) x);

myMethod((int) (cp + 5), ((int) (i + 3))+ 1);

# Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier—for example, whether it’s a constant, package, or class—which can be helpful in understanding the code.

Identifier Type Rules for Naming Examples

Packages The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying

countries as specified in ISO Standard 3166, 1981. Subsequent components of the package name vary according to an organization’s own internal

naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names.

com.sun.eng

com.apple.quicktime.v2

edu.cmu.cs.bovik.cheese

Classes Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple

and descriptive. Use whole words—avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the

long form, such as URL or HTML).

class Raster;

class ImageSprite;

Interfaces Interface names should be capitalized like class names.

interface RasterDelegate;

interface Storing;

Methods Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.

run();

runFast();

getBackground();

# Programming Practices

## Providing Access to Instance and Class Variables

Don’t make any instance or class variable public without good reason. Often, instance variables don’t need to be explicitly set or gotten—often that happens as a side effect of method calls.

One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, if you would have used a struct instead of a class (if Java supported struct), then it’s appropriate to make the class’s instance variables public.

## Referring to Class Variables and Methods

Avoid using an object to access a class (static) variable or method. Use a class name instead. For example:

classMethod(); //OK

AClass.classMethod(); //OK

Variables Except for variables, all instance, class, andclass constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore \_ or dollar sign $ characters, even

though both are allowed.

Variable names should be short yet meaningful. The choice of a variable name should be mnemonic— that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except

for temporary “throwaway” variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

int i;

char c;

float myWidth;

Constants The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores (“\_”). (ANSI constants should be avoided, for ease of debugging.)

static final int MIN\_WIDTH = 4;

static final int MAX\_WIDTH = 999;

static final int GET\_THE\_CPU = 1;

Identifier Type Rules for Naming Examples

anObject.classMethod(); //AVOID!

## Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.

## Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

fooBar.fChar = barFoo.lchar = 'c'; // AVOID!

Do not use the assignment operator in a place where it can be easily confused with the equality operator. Example:

if (c++ = d++) { // AVOID! (Java disallows)

...

}

should be written as

if ((c++ = d++) != 0) {

...

}

Do not use embedded assignments in an attempt to improve run-time performance. This is the

job of the compiler. Example:

d = (a = b + c) + r; // AVOID!

should be written as

a = b + c;

d = a + r;

## Miscellaneous Practices

### Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others—you shouldn’t assume that other programmers know precedence as well as you do.

if (a == b && c == d) // AVOID!

if ((a == b) && (c == d)) // USE

### Returning Values

Try to make the structure of your program match the intent. Example:

if ( booleanExpression) {

return true;

} else {

return false;

}

should instead be written as

return booleanExpression;

Similarly,

if (condition) {

return x;

}

return y;

should be written as

return (condition ? x : y);

### Expressions before ‘?’ in the Conditional Operator

If an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesized. Example:

(x >= 0) ? x : -x;

### Special Comments

Use XXX in a comment to flag something that is bogus but works. Use FIXME to flag something that is bogus and broken.

# Code Examples

## Java Source File Example

The following example shows how to format a Java source file containing a single public class.Interfaces are formatted similarly. For more information, see “Class and Interface Declarations” on page 3 and “Documentation Comments”

/\*

\* @(#)Blah.java 1.82 99/03/18

\*

\* Copyright (c) 1994-1999 Sun Microsystems, Inc.

\* 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.

\* All Rights Reserved.

\*

\* This software is the confidential and proprietary information of Sun

\* Microsystems, Inc. ("Confidential Information"). You shall not

\* disclose such Confidential Information and shall use it only in

\* accordance with the terms of the license agreement you entered into

\* with Sun.

\*/

package java.blah;

import java.blah.blahdy.BlahBlah;

/\*\*

\* Class description goes here.

\*

\* @version 1.82 18 Mar 1999

\* @author Firstname Lastname

\*/

public class Blah extends SomeClass {

/\* A class implementation comment can go here. \*/

/\*\* classVar1 documentation comment \*/

public static int classVar1;

/\*\*

\* classVar2 documentation comment that happens to be

\* more than one line long

\*/

private static Object classVar2;

/\*\* instanceVar1 documentation comment \*/

public Object instanceVar1;

/\*\* instanceVar2 documentation comment \*/

protected int instanceVar2;

/\*\* instanceVar3 documentation comment \*/

private Object[] instanceVar3;

/\*\*

\* ... constructor Blah documentation comment...

\*/

public Blah() {

// ...implementation goes here...

}

/\*\*

\* ... method doSomething documentation comment...

\*/

public void doSomething() {

// ...implementation goes here...

}

11 - Code Examples

20

/\*\*

\* ...method doSomethingElse documentation comment...

\* @param someParam description

\*/

public void doSomethingElse(Object someParam) {

// ...implementation goes here...

}

}

# Coding Checklist:

## Variable Naming Convention:

The variable names should be a noun and should exactly say what it means:

For e.g.:

a) public static final String FORM\_PARAMNAME\_STATUS = "status";

Which is a constant to take the status as a request Form parameter.

* Note: Always use FORM\_PARAM to denote any parameters from request.

1. public static final String FORM\_PARAMVALUE\_STATUS\_EFFECTIVE = "effective";

Which is a constant to denote the Expected value for the form parameter Status as ‘Effective”

Note: Always use FORM\_PARAM\_VALUE to denote a Value expected for a form parameter.

1. public static final String TAG\_PART\_LENGTH = "PartLength";

Constant to say that this value is used to create a TAG:

Note: Always use TAG to denote a tag, ATTRIBUTE to denote any attribute for xml and make sure the tag always starts with upper case & Attribute is always in lower case.

### Naming conventions for Java Collections:

Always the collection name should say what kind of collection does it belongs:

#### E.g. : arrListErrors ,hMapMailParameters

Note: Collection should always be represented in plural.

Some Samples for Bad Naming Conventions:

a) private String commandTypeParam = null;

This is a variable kept for taking the command values like “Search”, ”Add”, ”Update” . So by conventions it should have been named as “Command”.

b) String CHANGE\_STATUS\_URL="/statuschange";

As you see the variable is a verb. This should be STATUS\_CHANGE\_URL instead.

c) Element StausElement

In this case, the variable starts with Upper case! Should be statusElement.

## Method Naming Convention:

Method names should be verb and should exactly say what the method does.

For Example:

private void setPagingManager(Long userID) throws Exception {

HttpSession session = request.getSession();

if (session != null) {

pagingManager = (ResultSetPagingManager) session

.getAttribute(BroadcastMessageConstants.RESULT\_SET\_PAGING\_MANAGER);

}

broadcastMessages = BroadcastMessagesFactory.getInstance()

.fetchBroadcastMessagesByBroadcastUserID(connection, userID);

if (pagingManager == null) {

pagingManager = new ResultSetPagingManager(

BroadcastMessageConstants.RESULT\_SET\_PAGING\_MANAGER);

}

if (broadcastMessages.length >= 0) {

pagingManager.reset(broadcastMessages.length);

}

pagingManager.handleRequest(request, getValidationErrors());

session.setAttribute(SearchConstants.RESULT\_SET\_PAGING\_MANAGER,

pagingManager);

}

The above method says the method is setPagingManager, but it is also doing a job of fetching the broadcastMessages.

Better way of writing code:

Public Broadcasts[] fetchBroadcasts(Connection conn, Long userID)

{

return BroadcastMessagesFactory.getInstance()

.fetchBroadcastMessagesByBroadcastUserID(connection, userID)

}

public void setPagingManager(BroacCasts[] broadcasts)

{…..}

## Check for spelling mistakes:

Please avoid spelling mistakes. Some useful ways to overcome them

1. Be patient while writing the code.
2. Incase if you feel like you are not confident with the spelling, use Microsoft Word to check the spelling. (Copy the word which needs to be checked , paste it in Microsoft word , Check for spelling )
3. Perform more self review more frequently. This practice can remove 80% of spelling errors.
4. Give the codes to others for review, frequently to find out spelling errors.

## Avoid duplication of Codes

* Every time we write functionality, we need to check thoroughly for code duplication. Incase if you find similar codes, try to separate them into a different method and use them wherever required, by passing appropriate arguments.

E.g. :

private void addSupplier(Connection conn, HttpServletRequest request)

throws Exception {

Supplier supplier = new Supplier();

DTOBuilder.constructDTO(SupplierConstants.getSupplierColumnMappings(), request, supplier);

SupplierFactory.getInstance().insert(conn, supplier);

displayConfirmation(SupplierConstants.MSG\_SUPPLIER\_ADDED);

}

private void updateSupplier(Connection conn, HttpServletRequest request)

throws Exception {

Long supplierID = utilities.getLong(request .getParameter(SupplierConstants.FORM\_PARAM\_NAME\_SUPPLIER\_ID));

Supplier supplier[] = SupplierFactory.getInstance()

.fetchSupplierBySupplierIDForUpdate(conn, supplierID);

DTOBuilder.constructDTO(SupplierConstants.getSupplierColumnMappings(),

request, supplier[0]);

SupplierFactory.getInstance().update(conn, supplier[0]);

displayConfirmation(SupplierConstants.MSG\_SUPPLIER\_UPDATED);

}

Instead we can write a method buildSupplierDTO to which we can pass the request & supplier object for building the DTO from request.

## Provide appropriate comments for each method & class

Please have the practice of giving Java Doc type comments to all the methods & classes. It will help us to create a Java Doc API spec by using the Java tools. The comments should say exactly what the method does, what are the arguments, what is the result for the given arguments.

E.g.:

/\*\*

\* This method deletes a supplier.

\*

\* @param request

\*/

private void deleteSupplier(Connection conn, HttpServletRequest request)

throws Exception {

The method takes connection & request as arguments, but in the comment, only request is mentioned. Many a time we forget to modify the comments as we modify the methods.

E.g. :

/\*\*

\* Compares two Dates for ordering.

\*

\* @param anotherDate the <code>Date</code> to be compared.

\* @return the value <code>0</code> if the argument Date is equal to

\* this Date; a value less than <code>0</code> if this Date

\* is before the Date argument; and a value greater than

\* <code>0</code> if this Date is after the Date argument.

\* @since 1.2

\*/

public int compareTo(Date anotherDate) {

## Separate Different functionality’s into different methods

Lets consider we need to strip the time from date and build a date object from String representation of Date. We may write a method as:

Public Date stripTimeFromDate(String date) throws Exception {

Date dt = null;

if ((date != null) && (date.length() > 0)) {

SimpleDateFormat dFormat = new SimpleDateFormat("MM/dd/yyyy");

dt = dFormat.parse(date);

}

Calendar calFromDate = new GregorianCalendar();

calFromDate.setTime(dt);

Calendar calendar = Calendar.getInstance();

calendar.set(calFromDate.get(Calendar.YEAR), calFromDate

. get(Calendar.MONTH), calFromDate.get(Calendar.DATE), 0, 0, 0);

return calendar.getTime();

}

But as you see there are actually three different functionality . As we combine them together as a single function we may end up writing up duplicate codes incase if we have a requirement like stripping the time from a date object.

To make it reusable & Better we should be writing it as

public Date getDate(String date) throws Exception {

Date dt = null;

if ((date != null) && (date.length() > 0)) {

SimpleDateFormat dFormat = new SimpleDateFormat("MM/dd/yyyy");

dt = dFormat.parse(date);

}

return dt;

}

public Date stripTimeFromDate(Date date) throws Exception {

Calendar calFromDate = new GregorianCalendar();

calFromDate.setTime(date);

Calendar calendar = Calendar.getInstance();

calendar.set(calFromDate.get(Calendar.YEAR), calFromDate

.get(Calendar.MONTH), calFromDate.get(Calendar.DATE), 0, 0, 0);

return calendar.getTime();

}

public Date stripTimeFromDate(String date) throws Exception {

Date dateObj = getDate(date);

if (dateObj == null) {

return null;

}

return stripTimeFromDate(dateObj);

}

This makes the code Not only better, but also futuristic & reusable.

## No method should exceed beyond 100 lines

Please make sure we split the functionality into separate methods in order to avoid a single method running 100’s of lines. By doing this we are making sure the code is in a good shape and also it helps us to identify errors easily.

## Identify the common codes and move them to common files for usage in other functions

Many a times we write a code, as per the requirement, but it may be very specific to the requirement. If we put some thoughts, we can make these specific codes to reusable codes. Inorder to make these codes reusable, we must follow the logic of splitting different functionality’s to different methods. Make sure that we move the codes to Common Files for further usage.

Refer stripTimeFormDate example.

All projects should contain some common files like commonfunctions for JavaScript validations and validator, utility classes for performing validations on Java side.

Attached some of the common files that can be used in Java Projects.

## Always try to make the code generic for multiple usage

Refer stripTimeFromDate example.

## Tune your queries

Tuning queries is something that we forget to do many a times. The reasons for not doing this is because when we develop any code, we normally test them with few 10’s of records. So the query will execute faster. So we don’t mind tuning them. But when the application is tested with live data we will find that the screens take so much of time to come. Almost 80% of the reason is that the queries used in the screen are not tuned.

E.g:

Select SPI\_FORM\_DATA.SPI\_ID from SPI\_FORM\_DATA where STATUS = ‘Complete’ and STATUS! = ‘PreDraft’.

In the above-mentioned query we are already checking status =’Complete ‘, then what is the need to check Status! = ‘PreDraft’?

Remember not to use ‘\*’, NOT IN in any queries as they reduce the performance drastically.

## Avoid unnecessary global variables

Maintaining global variables in Java is not a good practice of coding. But some times it becomes necessary to use class level variables to access across the methods.

We shall eliminate the global variables if the method calls are well within the control of the class. But if we have a framework, which sequentially calls the methods, and we need to share the values across these methods, we need to maintain global variables. Even though maintaining global variable becomes unavoidable, we shall organize the class in such a way that it doesn’t contain too much of global variables.

Try to eliminate all non-static global variables. Only Static variables that can be shared across instances should be existing as Class level Variables.

## Initialize objects while declaration to prevent Null Pointer Exception.

Around 80% of the exceptions that we get in our projects are due to Null Pointer Exception .We need to

Handle this carefully. Most of the times the Null pointer Exception are caused from the Data that we retrieve from the Request.

String command = request.getParameter (“COMMAND”);

If (command.equalsIgnoreCase (“SAVE”) {…}

We may be confident that we always get the command from request. Because we are not checking null for command, we may end up in Null Pointer Exception. We shouldn’t be blindly checking null for these kind of variables. We must check for blank values also.

We shall come up with a method to process the request parameter as

Public String processRequestParam (String parmName)

{

String param = request.getParameter (paramName);

If(param != null && param.trim().length() > 0)

{

return param;

}

return “”;

}

Now the above method solves the NullPointer Exception, but when we try to get a numeric value from the request, it throws NumberFormatException. To avoid this we shall modify the method as

Public String processRequestParam (String parmName, String defaultValue)

{

String param = request.getParameter (paramName);

If(param != null && param.trim().length() > 0)

{

return param;

}

return defaultValue;

}

So that we shall call new Long (processRequestParam (“SPIID”,”0”));

## Use For, While Loops effectively

Many a time we might be confused to use whether “While” loop or “For” loop.

We should use while loop’s when we don’t know the size of the data. For Example when we read data from a file, when we try to iterate the HashMap, or when we are trying to remove all the data from a collection.

We should use For Loops only if we are aware of the size and we can iterate the collection through its size.

We must not a point that modification of loop variables inside the loop is not advisable.

For example

For(int I=0;I<arrListData.size() ; I++)

{

if(arrListData.get(I).equalsIgnoreCase(“RemoveMe”))

{

arrListData.remove(I);

}

}

The above code wont work because lets say we have the following data in the arrayList [“RemovMe”,”RemoveMe”,”DontRemoveMe” ,”RemoveMe”];

When the Loop iterates, after removing the first record, the second record becomes the first record and the loop variable increments by one to skip the second Data. So we are forced to decrement the Loop variable by one after removing the data from arrayList.

I.e.)

For (int I=0;I<arrListData.size () ; I++)

{

if(arrListData.get(I).equalsIgnoreCase(“RemoveMe”))

{

arrListData.remove(I);

I--;

}

}

Which is a very bad practice. Instead use the RemoveAll (collection) method to remove all the occurrences of this particular string.

Another point to be noted is for (I=0;I< arrListData.size (); I++ ) , here arrListData.size() is called every time the loop iterates . Instead we should be getting the size and store it in a variable before the loop and use them .

## Improvise the habit of taking the latest code frequently from the common repository

Please make sure that we have the latest codes, by synchronizing the codes with the project repository. By doing this we shall make sure that we don’t lag far behind the project implementation.

## Use Proper exception handling mechanism

Proper exception handling is critically important; but, unfortunately, the ways in which an application or class library generates and handles exceptions is one of the most ignored aspects of program design. Good exception handling practices don't just make your programs more robust and easier to maintain; when something bad happens, the thrown exception might provide the only clue as to what went wrong. If your programs use exceptions sensibly and generate meaningful error messages, the user is less likely to grow more confused and annoyed (at a time when your program has failed, and the user is probably already annoyed). Good exception handling will also make your support staff better able to understand a problem and fix it. Properly using exceptions ensures that all parties -- code and humans alike -- have the error-recovery information they need.

Exceptions should be propagated to the caller. We shouldn’t be suppressing the exceptions. Incase if we need to do so, we must document the reason for suppressing the exception. Usage of PrintStackTrace should be avoided, instead use logging mechanism.

Lets consider an example of ResourceLoader .

public class ResourceLoader {  
 public getResource(String name) throws ResourceLoadException {  
 ...  
}

When the getResource () method throws a ResourceNotFoundException, it communicates three important pieces of information:

1. The kind of exception that occurred (in this case, a ResourceNotFoundException)
2. The location where the exception occurred, in the form of a stack trace contained within the exception
3. Additional information about the exception, in the form of the message string

## Align the codes as you write them

Now a days all editors come with code alignment features. The code we write can be read by others only if they are in a readable format. To make the codes readable we must align the codes. Make sure to align the codes to a standard format of 80 columns per lines.

There are lots of tools available to align the codes. Jalopy comes as a plugin to almost all the Java Editors, if not we can configure Jalopy with ANT to make sure we deliver only the aligned codes to customer.

## Use Pleasing words to report any error messages

The Error Messages that we show to the user should never be commanding. Always use “Please” to report any errors to the user. By using pleasing words we actually make the users feel comfortable to use the application.

Some examples:

a) public static final String ERROR\_COMMENTS\_LENGTH = "Comments should not exceed more than 4000 characters"

Should be “Please enter a valid comment. Comments should not exceed 4000 characters”. And look at the Variable name. Should be something like INVALID\_COMMENTS\_ERR\_MESSAGE.

## Make sure the XML Data is handled appropriately

One of the most important things we need to concentrate is handling the XML data appropriately. Most of the times this is not taken care. First Of all we need to decide the mode of parsing the XML.

Decide DOM/SAX parsers based on requirement to modify / Read the XML Data.

## Log in-sensitive and required Data

Logging information is one of the most important things in a project. Unless we log the information we will not be able to debug the application. At the same time usage of Excessive logging information is not advisable as the data is going to be stored in the log file.

Kindly make sure that we are not logging any sensitive data like Password etc. If we log the password, then there is no need to maintain any password. Any one can open the log file and take the password of the logged in user.

When we log any information, please use appropriate Log methods & DEBUG LEVEL. Please use log.error for logging errors, log.info for logging information’s & messages etc.

## Try to make use of the common codes available in the project without over ridding

Always look for common codes available in project. Please try to reuse existing codes instead of rewriting them in each place wherever it is required.

Consider the stripTimeFromDate example.

## Make sure the file names all are in lower case except that of Java Files.

Whenever we code, we always make an assumption that the operating system in which we are developing will be the same as that of production. But the truth is not that. We are developing in “Windows” Operating System, but most of the times the application is deployed only in linux. Linux OS is very sensitive with respect to files.

So in order to make sure that our application works in both environment, please follow few instructions

1. Always have the File names in lower case ( Except Java files )
2. Make sure we use the File.SeparatorChar to specify the file path inorder to refer any Files from the Server or use ResourceLoaders to load any files from Server. Do not specify the file path as XXX//XXXX//XX.XX or XXX\XXXX\XX.XX
3. Try to avoid storing any files physically in the server through Coding. Because the user may not have privileges to store any files.

## Try to use project level methods for displaying messages / confirmation

Always have a project level methods for displaying any messages / Confirmations to the user. Always use these wrapper methods in the code instead of calling alert & confirm straight away. So that incase if necessary to show the custom message dialogs we shall change it in a fraction of second.

## Avoid retrieving data from Session / Request many times . instead get them once and use them where ever required

Try to avoid fetching of same data from Request / Session

if (request  
.getParameter(SupplierConstants.FORM\_PARAM\_NAME\_ROWS\_PER\_PAGE) != null) {  
selectedRowsPerPage = request  
.getParameter(SupplierConstants.FORM\_PARAM\_NAME\_ROWS\_PER\_PAGE);  
}

Instead we shall retrieve the data from request once and then use them later.

## Try to keep the logic as simple as it can be

Lets consider a sample that logged in user has some roles , we need to create an XML data with all roles , by having an attribute for specifying whether the logged in used has this role assigned to him or not .

Normally we write a step like this

private boolean isLoginUserRole(Connection conn, String roleDesc)

throws Exception {

String[] roles = profile.getRoles();

boolean isLoginUserRoleflag = false;

for (int i = 0; i < roles.length; i++) {

if (roleDesc.equalsIgnoreCase(roles[i])) {

isLoginUserRoleflag = true;

return isLoginUserRoleflag;

}

}

return isLoginUserRoleflag;

}

Then call this method for each role .Now lets consider the complexity and poor usage to Java .

Lets consider there are 15 roles in the Database, and the logged in user has 7 roles.

This method is called 15 times with the loop running 15X7 times. Each time the loop gets through the value , we are comparing the Role by using “EqualsIgnoreCase” which again converts the role all to upper case ,all kinds of string modifications are going to happen .

Simplest way!!

1. Add all the Roles assigned to the user in a HashMap, pass the Role Name as key and find out whether the Role is assigned to the user or not based on the value returned by HashMap.
2. Add the Roles assigned to the used into an ArrayList, use the contains method to find them .

Why are these Collections effective than looping?

There is something called “HashCode” for all the objects that we create in Java and they are unique. When we search using a HashMap or ArrayList, they are going to fetch the data using this HashCode, which is the most efficient way.

When we write a code simpler, we can read it and understand it without having much trouble. Code maintenance will be simple and one can make the code to bend in all possible ways without modifying much.

Attached is some sample for how a complex & simple code look.



## Use a separate Java script files for writing scripts

If we have lot of JavaScript’s to be written, try to put them in a separate Js file and include them .By doing this we can also hide the logic of script implementation.

Avoid writing lots of JavaScript’s, as some of the browsers cannot execute JavaScript’s.

## Use String Buffer for concatenating Strings

As we all Know “String” is a immutable class, every time we try to modify the Data, it keeps creating a new Object in the memory, it is not advisable to use “String” objects for data manipulation. Always append method of StringBuffer to manipulate any String. StringBuffer class has got so many methods using, which we shall change, any part of a string, including String reversal.

## Avoid hardCoding

* Please avoid hard coding in any form. Always use Constants to overcome any hard coding. Maintenance of constants can be in any form like an XML File or the simplest is have a Java Constants file for declaring all constants.

## Implement toString method

The toString method is widely implemented. It provides a simple, convenient mechanism for debugging classes during development, by translating object state into text. It can also be used at runtime for passing informative error messages to Exception constructors and assertions. When used only in this manner, the exact format of toString does not form part of the normal operation of the program.

The toString method may be used more formally, however. A common example is a simple persistence mechanism, in which formatting (toString) and parsing (valueOf) services are provided for converting an object to and from a specific textual form. Here, it is particularly important that the exact format of the String used to represent the object be specified in javadoc. (Be aware that this use of toString ties an object to a particular representation of its state, however, and should be used with care. Some argue strongly that such details regarding persistence should be removed from a Model Object and placed in the persistence layer.)

When implementing toString, StringBuffer can be used instead of the + concatenation operator, since the StringBuffer.append operation is slightly faster.

## Use Object Oriented Concepts effectively

Lets travel through some examples where we fail to use the OOPS concepts.

a) Consider the method

public ArrayList validateFormData(CommonFormData commonFormData,

ArrayList arrListErrorMessages) throws Exception {

………………..

return arrListErrorMessages;

}

In the above method we are passing an ArrayList, which contains some error messages. The method is still going to validate the data and based on the Data, it will add the error Messages to the passed in ArrayList. If we closely look into the validateFormData method we see that this method is returning the ArrayList back which is a major violation of OOPS concept.

As we know the ArrayList’s are pass by reference and not pass by value, we must understand the concept that all the modified information will automatically reflect in the passed object.

Instead of validate method returning the ArrayList, it should return a boolean value stating whether the validate is success or Failed.

We should be very careful while copying any information from one object to another. We should always use Deep Copying mechanism while copying one object to other.

E.g.:

ArrayList arrListErrorMessages = new ArrayList ();

If (….)

{

arrListErrorMessages.add(“ Invalid …….”);

}

arrListErrorMessages = validateFormData(commonFormData);

Let’s consider the above example. Initially we are adding some messages into ArrayList .Now we are calling validateFormData, which is again going to return an ArrayList of ErrorMessages. Since we are assigning the return value directly the arrListErrorMessages we are actually loosing the already added messages.

That’s where we need to be very careful while assigning to Objects. Incase if we need to retain existing values, use Deep Copy instead of shallow copy.

|  |  |
| --- | --- |
| Logging | |
| 12.29.1.1 | System.out.println() or .print() statements should not be used in code checked into CVS. If you want to use these during development, fine, but they must be either stripped out by the time the code gets merged to trunk, or else converted to log.debug() statements or other log statements, if you want to retain them. Or, you can just use log.debug() statements from the start; these can be turned on by adding an entry to the appropriate log4j.xml or log4j.properties file temporarily. In test cases, you can use the display() method implemented in the base test classes. |
| 12.29.1.2 | The logging level for logged messages should be chosen carefully per the following:   * Error: A real error that someone should be made aware of, follows up on, and fix. These are always displayed in the log. * Warn: Something that is out of the ordinary and worth being aware of, but is not necessarily an error that needs fixing. Warnings are left on in our log4j configuration files, so you should expect them to be displayed in the log. * Info: Ordinary occurrence at a fairly high level within the class which is useful to get an idea of the high-level flow within an application. There shouldn’t be more than a few of these messages per business event. These are not ordinarily turned on in the log4j configuration files. However, you can set a specific logger to the INFO level in the log4j configuration if it is useful to display the message in production (for example, to display at startup certain critical settings that are in effect).   + For debugging or code exploration purposes, setting com.htc.eis=INFO in the log4j configuration should yield a “reasonable” number of messages per business transaction. That might be dozens, since a single event such as receipt of a TIF file can involve many processing activities, but it should not be hundreds and certainly not thousands.   + Therefore low-level classes which are called numerous times in the course of a single processing flow should probably not use the Info level. * Debug: Low-level messages which may occur frequently in the course of a transaction. As the name implies, these are turned on only for debugging problems as otherwise they might be so numerous as to make the rest of the log hard to read. * Trace: See commons logging documentation for the intent of these, but be aware that log4j, which we are using, does not distinguish between debug and trace levels. In other words, log4j.logger.xxx=DEBUG will result in both log.debug() and log.trace() statements appearing in the log, although you might expect only log.debug(). Simple “entering/exiting method” messages should be coded as log.trace() versus log.debug() if you feel the need to use them (but consider use of Spring-based Before/AfterMethodAdvice as an alternative). |
| 12.29.1.3 | To facilitate Tivoli monitoring of messages, any business events which the business analysts define as worthy of such monitoring should have a message logged twice, once to the normal logger at an appropriate level, and again to a second logger named “monitor.” + CLASS\_NAME. The log4j configuration file will then specify that “monitor.\*” will be logged to a separate file that Tivoli will monitor. [Not yet implemented as of 2/2/07; if you are the first, define “monitor.” in some constants class, and give an example here of how to code.] |
| 12.29.1.4 | As far as the content of logged messages, generally do not prefix the message with asterisks, exclamation points, or any other emphasis. This just clutters the log, and if everyone were to do it, it wouldn’t help your message stand out anyway. |

|  |  |  |
| --- | --- | --- |
| Jars | | |
| 12.29.2.1 | All third-party jars (from Jakarta Commons, Struts, Hibernate, Spring, and so on) should be named with the version in their name, for example “commons-lang-2.1.jar”, not “commons-lang.jar”. In some cases the jar will come from the provider with a version as part of the standard name, but if not, you should add it. | |
| 12.29.2.2 | The master version of all third-party jars is lib, so that is where the master copy should go. Within Eclipse, the .class path for utility projects (for example framework, security) should reference the jars in repository. | |
| 12.29.2.3 | For enterprise the jars must be copied from lib to the app project, in order to run within RAD/Eclipse and in order to build the EAR via Eclipse. Also given the way RAD works, the EJB projects’ and Web/WAR projects’ .classpath file should reference the jars in the app project rather than in lib. (The reason is that RAD will automatically do this anyway, so you would wind up with duplicate entries in .classpath if you reference the lib ones.) | |
| 12.29.2.4 | When obtaining open source jars (including Jakarta Commons, Struts, Hibernate, Spring, and so on), commit the ZIP file from which the jar was obtained lib. That ZIP file usually includes the javadoc also, but it if doesn’t obtain that file also and put it in lib. If the ZIP file the jar came in doesn’t include the source, then obtain the source ZIP file and put that in lib too. | |
| 12.29.2.5 | As the standard third-party jar source location, the ZIP files in lib can be specified as the source file location within RAD, and that committed (in the .classpath file), so that the linkage to source is available to everyone for debugging purposes. | |
| Java Code | | |
| 12.29.3.1 | | All literals or numeric values which are codes, keys, Spring bean names, element/attribute names, etc. should reference constants (Java “final”) or enums rather than having values hard-coded everywhere they appear in the code. Statement such as “if (type == 12)” or “if (code.equals(“D”))” should never be used.   * Note several constants and enum classes are already defined. Search for \*Constants.java and \*Enum.java among the projects. * It is generally a good practice to use constants even for numeric values used locally within a specific class, which then makes the code self-documenting if a good constant name is chosen. (This excludes numbers like 1 when used to adjust an index pointer or whatever.) * Constants classes should not be defined as interfaces, and then have a class wanting to use the constants “implement that interface”, in order to avoid having to put the Constants class name in the code. This is a misuse of the interface concept, which is that a class implementing an interface behaves in certain ways for the benefit of clients of that class, or acts in a certain role within a framework. |
| 12.29.3.2 | | Constants are named using all upper case with words separated by underscores, for example “SOME\_CONSTANT” (not “someConstant”). Enum instances should be named in the same manner. |
| 12.29.3.3 | | Java has an Enum implementation based upon modified versions of Jakarta Commons Enum classes. Enums should be used in preference to constants where the concept of an enum applies, that are a property or method parameter which can assume only a limited number of pre-determined coded values which have specific meanings. Constants should continue to be used for things like XML file names, specific numbers, delimiters, and so on. Our implementation of enums will eventually be replaced with the Java 5 native implementation, once the project has switched to Java 5. Following are more specifics:   * Never use the word “enum” as the variable name to hold an enum, and never use “Enum” as a class name. “enum” is a Java 5 keyword. * Many enums are naturally thought of as “types”, for example address type, business owner type and so on. The name of the enum class in these cases should be AddressTypeEnum and BusinessOwnerTypeEnum. But the variable, method parameter and/or property names should be addressType and businessOwnerType, not addressTypeEnum or businessOwnerTypeEnum (in other words, should not end in “Enum”). Nor should they be something even more heinous such as typeEnum. * Not every enum is naturally named a “type” though. It is perfectly fine to have something like EventCategoryEnum or SpendingLevelEnum. Enum class names should always end in “Enum”. (Therefore you can find all the ones by doing a file search for \*Enum.java) * Use AddressTypeEnum.java and AddressTypeEnumType.java (EnumTypes are explained in a moment) as models for coding and javadoc’ing enums. Please keep the exact language used there, of course modifying where it refers specifically to address type. Also keep the commented out piece at the bottom regarding fromCode(). * The naming of the enum instances, and the javadoc explaining them, is critical. The name should be as self-explanatory as possible, and the javadoc should as usual not simply restate the name, but provide further explanation. * An enum instance consists internally of two properties, the “code” and the “description”. The code is guaranteed to be unique, the description is not. However, the current implementation will force you to specify a unique description, which makes sense because there must be some difference in meaning between each enum instance. * If the enum is only for in-memory use, the code values of the instances are arbitrary, although they should have some mnemonic value since they may appear in dumps or logs. However, if the enum is for a domain object property which corresponds to a database column which is the foreign key to a code table, then the code must exactly match the code table primary key value. (The same consideration applies to a database column which is subject to an Oracle constraint.) Furthermore the description should match the description in the code table, except that if the descriptions in the code table are unclear or misleading, those for the enum can be clearer and more meaningful. * The same enum class can be used as the type of as many properties (and method parameters, etc.) as it applies to. * To compare two enums, do not compare their code values. Compare the enums directly using equals(). Correct:  if (AddressTypeEnum.REMITTANCE.equals(addressType)) … Incorrect  if (AddressTypeEnum.REMITTANCE.getCode().equals(addressType.getCode())) … * Since Hibernate automatically handles conversion between the code value and the enum instance, the only time you should ever need to access getCode() is if you are persisting the value to an XML file or using in a Web page, or writing a log message. Implement a fromCode() method if needed for the opposite side of the first two scenarios (return enum instance based on code). * For domain object properties which are enums, you must also implement a Hibernate UserType subclassed from BaseEnumUserType, and specify this in the Hibernate mapping file. See the Hibernate Standards document, section “Use Type-safe Enumerations” for details. The naming standard for this class is the enum class name plus “Type”. So the one for AddressTypeEnum is named AddressTypeEnumType, or for EventCategoryEnum is EventCategoryEnumType. * You can obtain a list of all the enum instances for an enum class via the following:  BaseEnumUtils.getEnumList(MyEnum.class) See BaseEnumTest for an example. * Use enums wherever they apply. This will enable the compiler to catch common coding bugs, as well as reduce the javadoc you would otherwise have to write (no need to write “use one of the values in AddressTypeConstants” as was needed for parameter String adddressType in the past). * As of 8/24/07, we have concentrated on enums where the code value is a String. This is because all the code tables use strings (varchars) as their primary keys. There is also a Jakarta base class called BaseValuedEnum which is for an int rather than String type. However this has not yet been fully adapted . It would only take an hour or two to do so, basically changing “name” to “code”, “value” to “intValue”, the method “getValue” to “toInt”, and so on, as well as creating a BaseValuedEnumUserType for Hibernate use. Contact Mark Schechter if this becomes needed. |
| 12.29.3.4 | | Visibility of methods should be the minimum necessary to function. That is, methods should be declared private unless they are used outside the class, /\* package \*/ unless they need to be protected (used outside package by subclasses), and protected unless they need to be public (used by unrelated classes outside the package). The same principle, “keep it shy”, applies to visibility of variables and classes. |
| 12.29.3.5 | | Use braces around “if” and “else”(and other control flow: for, do, etc.) executed statements, even if there is only one. This reduces errors, especially during maintenance. |
| 12.29.3.6 | | Using deprecated methods is discouraged. There may be some cases where there is no alternative, but if a method you are using is showing as deprecated, check whether there is a current (non-deprecated) alternative that can be used instead. |
| 12.29.3.7 | | Do not use deprecated methods from class Date; use the Calendar class and subclasses instead. Not only have they been deprecated since Java 1.2, they return useless values. For example (new Date()).getYear() returns 106 for year 2006. |
| 12.29.3.8 | | [Reword: Be conscious of multi-threading issues, and practice good thread safety] |
| 12.29.3.9 | | Use try/catch/finally blocks on all JDBC code, and within the finally block close any open prepared statements and results sets, as well as any connections that you created (but not connections supplied by the context). Use try/catch blocks within the finally block to deal with errors if they occur. |
| 12.29.3.10 | | For JDBC code, use prepared statements including bound parameters as needed, versus constructing a SQL string on the fly, for improved performance and to avoid SQL injection attacks where user-entered parameters are involved. |
| 12.29.3.11 | | All user input, for example form fields on an HTML screen, should be validated on the server, even if it has already been validated on the client (browser). It is easy for attackers to avoid client-side validation. |
| 12.29.3.12 | | Keep code formatting readable by people. For example, removing spaces just because the compiler doesn’t care is discouraged where the result is difficult for a human to read. |
| 12.29.3.13 | | Tab should equate to 4 spaces (RAD/Eclipse default). Do not set it differently.  Also, tabs should be expanded to spaces (Eclipse preferences does this automatically). |
| 12.29.3.14 | | Reformatting an entire source file makes comparison between the original and modified versions difficult if not impossible. Therefore, do not reformat an entire source file:   * If there is a possibility that someone else is working on it locally or in another branch (makes merging difficult); or * Where you are making any other changes to it, whether changes to code or to documentation (makes it difficult for the code reviewer to see what was changed); or * At any time unless you wrote it or are the primary maintainer of it at the time (destroys the writer’s original format which may have been intentional).   Therefore, the best time to reformat a source file is when no one else is working on that file and when you are not changing it either, except to reformat it. |
| 12.29.3.15 | | When catching exceptions which are true unexpected errors, in order to ensure the stack trace is logged, the exception should be passed as a second parameter to a Logger instance error() method, which will then print the stack trace in the log. For example:  try {  ....  } catch (SomeSpecificException ex) {  log.error( "Error: SomeSpecificException happened while [explanation]... ", ex );  throw new FileParsingException( ex.getMessage() );  }  Typically this situation arises when the caught exception is being transformed into a different exception which is rethrown.  In any case, never catch any exception that is a true unexpected error without logging it with loggerInstance.error(message, theException) as per the example. |
| 12.29.3.16 | | It is not necessary or beneficial to enclose code in try/catch blocks unless there is a specific reason to do so. In most cases runtime exceptions should simply be allowed to flow up the stack and abort whatever transaction is in progress. However, if there is a specific reason, such as in an incoming file validator, or MDB in the event framework, which should not ever fail silently, it is best to catch specific runtime exceptions if these can be reasonably anticipated. In a few situations, it is valid to catch "RuntimeException" rather than specific runtime exceptions, but these situations should be rare.  Never catch just "Exception", except in a finally block (for example releasing database connections). The reason for this is that the same response to any kind of error whatsoever, including unpredictable runtime errors, is generally not appropriate. Instead catch checked exceptions individually or at some level more specific than "Exception", and if there is common processing for them all, put the common processing in a small method or set a gotError flag and handle it after the catch blocks. By the same token, never catch "Throwable", which has the same problems as catching "Exception" but worse, because it also catches Errors which may have rendered the state of the JVM unpredictable. (Note there may be very rare exceptions to these guidelines; if so, discuss with a tech lead.) |
| 12.29.3.17 | | Methods should never throw “Exception” and/or be declared as throwing “Exception”. If this were done, then 2.160 which says to never catch “Exception” could not be followed. If you are unclear on the purposes of checked exceptions versus RuntimeExceptions, discuss with a tech lead. |
| 12.29.3.18 | | See Appendix A of this document, Java and Javadoc Coding Conventions, for additional notes on standard Java coding mechanics. |
| 12.29.3.19 | | Collections and arrays should generally contain one type of element, not a miscellaneous group of objects of varying types. Disparate objects should be grouped by using a class with properties for each of the disparate objects. The temptation to use a collection to hold disparate objects may arise when coding a method that needs to return several disparate values. A class must be used as the return value, but to ease the coding burden and increase readability, a private inner class containing only public instance variables (fields) can be used if you wish, if the method in question is private (see Appendix B for example). If the method is not private, a standard class with getters and either setters or a non-default constructor must be used. |
| 12.29.3.20 | | Eliminate BO usage: For operations involving domain entities and database access, the code base as of 3Q 2007 currently has DAO classes (interfaces and implementations), and then in many cases a duplicate set of "BO" classes (interfaces and implementations). The BO classes in most cases do not add anything to the DAOs. Apparently when these were developed in 2Q/3Q 2006 the thought was that the BO layer would perform various encryption-related activities, but in fact all those are handled in the domain objects themselves. Since February 2007 we have been planning to remove the BO layer entirely, as soon as time allows.  The following policies are currently effective, and have been since June 2007:  No new methods are to be added to BOs, nor any new BOs created. New methods should be added only to DAOs, and fully javadoc'ed in the interface class as usual. Whatever class is calling the new method should be converted to use the DAO instead of the equivalent BO, for all calls, not just this new one. For example, if you are adding a method to FileDAO, and the class is currently using FileBO, change that class to use FileDAO instead and eliminate FileBO entirely. This is not difficult and should only require 5-10 minutes for a small class or 30-60 minutes for a large class.  No new or modified code should call any methods on BOs. The methods should be called on the DAOs instead. The paragraph above applies.  If you are working on the Admin app in places where currently a BO is necessary to implement the transaction, create or use an appropriately named Service class to implement the transaction, and get rid of all the existing BO usage. Configure the Service class to transact its methods in Spring. This is a far more significant refactoring but still not that big (probably around 8 hours). However it probably needs to be accounted for when you create your estimate for the task.  There are a few methods in existing BOs that actually do substantive work. These should be refactored into either DAO methods (if the substantive calls are to a single DAO class) or Service methods (if the substantive calls are to multiple DAO classes, or involve significant business logic).  This information has been communicated to all developers a number of times over the course of 2007, with additional detail. If you are just joining, or need a refresher, ask Mark Schechter to forward a copy of the cumulative email of 8/1/07, subject “Do not add any new BO usage in any manner (follow up)”. |
| JUnit Tests | | |
|  | | [These items are preliminary, pending development of the testing framework which is in process] |
| 12.29.4.1 | | Implementation of new or changed functionality, including bug fixes, should be accompanied by JUnit tests which verify that functionality. Verifications should include both success paths and, where validation or significant error handling is involved, failure paths.  Hint: If you write the tests at the same time you are writing the functionality, you will find that it helps your coding efforts and does not significantly increase the time required to develop (non-buggy) code. |
| 12.29.4.2 | | A test must include the following steps:  Set up initial conditions (do not rely on test records already being present in the database).  Execute the function under test.  Verify the results of the function. For example, if the result should have been that various records were written to the database, read them back from the database to verify that they are there with contents as expected.  For functions that can be tested independently, set up may be as easy as passing a particular function parameter, and checking the results as easy as checking the return value. |
| 12.29.4.3 | | Base test classes that, already exist and can be used to provide a test environment. SpringBaseTestCase for example will load the Spring configuration, and automatically roll back any database transactions at the end of each test.  If a test does not require Spring and/or Hibernate and/or other project specific objects that access the database, do not use SpringBaseTestCase, use BaseTestCase instead.  There are subclasses of SpringBaseTestCase which should be used within specific projects to load the correct Spring configuration files for that project. Search for \*SpringBaseTestCase to see which ones are available; you can also create one if necessary in a project which doesn’t have one yet. |
| 12.29.4.4 | | JUnit tests should not require external classpath settings or other special settings to be run within RAD. The purpose is to make it easy for developers to run JUnit test suites frequently.  Typically an external classpath setting would be used to pick up a configuration file from the file system. The need for this in the test environment can be avoided by having a test version of the configuration file stored in the /test source tree of the project, and having a method (typically static; package-level if possible) to internally override the name of the configuration file. For example if the normal configuration file is called someConfig.properties, then create /test/test-someConfig.properties, and method useTestConfiguration (no parameters) in the class which reads the configuration file. |
| 12.29.4.5 | | Independent utility functions such as Date manipulation, String manipulation, encode/decode, and so on should have JUnit tests written effective immediately, since the set up of initial conditions for these is easy. |
| 12.29.4.6 | | Any other functions which do not require elaborate set up of initial conditions, for example transformers or validators, should similarly have JUnit tests written effective immediately. |
| 12.29.4.7 | | Functions which require that a TIF has been processed to some extent, or that even further downstream events have occurred (to take an extreme example, an ODFI/Processor response to a PIF), may need to await the testing framework before it is reasonable to be able to write a JUnit test for them. |
| 12.29.4.8 | | SpringBaseTestCase, which is an ancestor class for all test classes which use Spring and/or Hibernate, provides a protected method isRollback() which may be overridden temporarily. The default behavior is to roll back all database changes after each test completes. In some cases you may wish to override this temporarily so you can see the results of the test in the database. If you choose to override isRollback() temporarily, then 1) you are responsible for cleaning up the database afterwards, for example removing records that your test inserted, and 2) you must delete the override entirely before committing the code back to SVN, whether in a branch or in trunk. |
| 12.29.4.9 | | Use Ejb3Unit for testing the EJB3.0 beans and in order to validate the complete functionality of the projects using EJB3.0 |

### Appendix A: Java and Javadoc Coding Conventions

The Development Standards are purposely kept short and devoted more to substantive matters. It has become apparent however that developers coming from different backgrounds may be unfamiliar with conventional Java coding style, and use of Javadoc. As stated in Sun’s *Code Conventions for the Java Programming Language* (<http://java.sun.com/docs/codeconv/>), the reasons for coding conventions include:

* 80% of the lifetime cost of a piece of software goes to maintenance.
* Hardly any software is maintained for its whole life by the original author.
* Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

In other words, code conventions are part of achieving the goal of making our software readable and understandable by someone other than the author.

If you haven’t read Sun’s *Code Conventions* document recently, please do so now. In general the project adheres to these conventions, although there are some places we don’t (for example, we permit more flexibility with braces style, as long as the developer is consistent).

If in doubt, and if the convention stated in the Sun document is not overridden or relaxed by our Development Standards, adhere to the Sun conventions.

One specific point additional point, in addition to braces style, where you don’t need to follow the Sun conventions is the statement “Put declarations only at the beginning of blocks” in Section 6.3, Placement. Variables can be declared at the point they are first used, if you wish. In fact, variables defined within methods, and which are used in only a limited section of code, should be defined where they are first used as this makes their purpose clearer and also adheres to the spirit of Development Standards 2.040 regarding keeping visibility to a minimum.

Following are specific points which some developers have not followed, and which result in significantly less readable or understandable code. Therefore developers must adhere to the following sections of the Sun document.

8.2 Blank Spaces -- Please read the section in the Sun document.

Here are some examples from actual code that violate these conventions:

for(int i=0;i<retrievedPaymentGroups.size();i++)

Should be:

for (int i = 0; i < retrievedPaymentGroups.size(); i++)

assertEquals(1,retrievedPayments.size());

Should be:

assertEquals(1, retrievedPayments.size());

final int PAYGROUP\_0\_GROUP\_CONTROL\_NUM=5;

Should be:

final int PAYGROUP\_0\_GROUP\_CONTROL\_NUM = 5;

If you don’t like typing spaces as you type, use the Source > Format command in RAD to format the code you wrote after you type it. (Important: only reformat code you just wrote. Only under certain conditions should you reformat existing code, even if you wrote it. The issue is around the ability to compare old and new versions of modules during code reviews and merges. See Development Standards 2.140 for more details.)

Not explicitly in Sun’s doc, but implied and generally accepted, and will follow

Do not use type-based Hungarian notation except sparingly.

Hungarian notation is the prefixing of variable names with an abbreviation of the variable type, for example strCustomerName (str means string). This practice achieved widespread recognition at Microsoft (Charles Simonyi) and was used in C and to some extent in Visual Basic. It was useful in past situations were variables were not strongly typed, or where C pointers could not be fully typed. In Java, and given current IDEs where the developer can hover over a variable and have the type displayed, it generally does not add value and makes the code more difficult to read. For example,

objPayment vs. payment (most things in Java are objects)

lstPayments vs. payments (collection already suggested by final “s”)

intPaymentIdx vs. paymentIdx (most indexes are ints anyway)

strBuyerName vs. buyerName (most names are Strings anyway)

For more discussion of this, see: <http://ootips.org/hungarian-notation.html>

However, sparing usage is OK, for example in a small section of code where two variables refer to different types representing the same entity, because there are several methods to be called and they take different types. Or there may be other specific situations.

Note: In case the online version of Sun’s Coding Conventions is not available, you can reference the downloaded version, found in the same directory as this document as SunJavaCodeConventions1999.zip.

Javadoc Usage

Use javadoc comments, that is /\*\* comment \*/, where javadoc is recognized by the javadoc generator, such as immediately before the class declaration (class-level javadoc), immediately before the method declaration (method-level javadoc), or immediately before class or instance level constants or variables (field-level javadoc).

Use regular java comments, that is /\* comment \*/, where you want to make a comment but it has no relevance to the javadoc generator, for example in the middle of a method.

Correct:

/\*\*

\* Reverse characters in the supplied String and return the result.

\* Null input returns null.

\*/

public String reverse(String toBeReversed) {

...

/\* Next we will do such and such \*/

...

}

Incorrect:

/\*

\* This is not javadoc and won’t appear in the generated output.

\*/

public String reverse(String toBeReversed) {

...

/\*\* Javadoc marker used unnecessarily since this comment

will never appear in generated javadoc. \*/

...

}

Do not use the @author tag in the javadoc of a method. (Usage in class-level javadoc is fine, per the standard Eclipse Preferences import.)

### Appendix B: Examples for Development Standards Items

This section of the document contains examples which expand on the points made above. The purpose of this separate section is to keep the main statement of the standards to a reasonable length.

2.180

Example of using an inner class with public instance variables (fields) only, as the return value from a private method. The following code snippets are all within the same source file.

/\*\* This does something. Internally it needs to access a private method that returns disparate objects. \*/

public void someCallingMethod(...) {

...

FindPopularPartnersResult result = findPopularPartners(buyerId);

if (result.supplier == null) { //No payments made in last 30 days

...

}

/\*\*

\* Find the Supplier and Processor which were involved in the largest number of

\* payments made by the specified Buyer for the past 30 days.

\*

\* @param buyerId unique identifier for the Buyer to find popular partners for

\* as described above.

\* @return see javadoc of FindPopularPartnersResult.

private FindPopularPartnersResult findPopularPartners(long buyerId) {

...

Supplier supplier = ...

Processor processor = ...

...

FindPopularPartnersResult result = new FindPopularPartnersResult();

result.supplier = supplier;

result.processor = processor;

return result;

}

/\*\* Return value from <code>findPopularPartners()</code> \*/

private class FindPopularPartnersResult {

/\*\* Supplier to which buyer made the most payments in past 30 days.

\* Will be null if no payments made. If two suppliers are tied, one

\* will be selected arbitrarily. \*/

Supplier supplier;

/\*\* Processor used for greater number of payments made by buyer in past 30

\* days. Will be null if no payments made or only EFT payments made. If two

\* Processors are tied, one will be selected arbitrarily. \*/

Processor processor;

}

# Hibernate Standards for Use with Java Projects

## Standards - Currently Effective

### Configuration

#### Externalize As XML

The Hibernate configuration is externalized using the XML format provided by Hibernate and Spring.

The top-level files are the Spring configuration files with names of the form \*spring-\*session.xml.

The second-level files are configuration files which list the Hibernate mapping files to be used. These have names of the form hibernate\*.cfg.xml. As of 6/20/07 these include hibernate.cfg.xml and hibernate-admin.cfg.xml.

Finally there are the Hibernate mapping files themselves. These have names of the form \*.hbm.xml. There are dozens of these, typically one for each table in the database schema.

#### Avoid the Second Level Cache

Due to the distributed nature of EIS applications, and the fact that each application does not have exclusive access to the database, the second-level cache can yield unexpected results (even when used with transactional/distributed cache providers). The second level cache should be disabled for the project.

The second-level cache is disabled by the following statement in spring-session.xml (and other \*spring-\*session.xml files):

<prop key="hibernate.cache.provider\_class">org.hibernate.cache.NoCacheProvider</prop>

If caching is desired for domain objects, the application should make use of a caching mechanism apart from Hibernate at a higher level in the application.

### Session Management

#### Avoid Direct Session Management

Spring is the primary transaction manager used in the project. Never create or close sessions directly, for example with statements like the following:

Session session = sessionFactory.createSession();

or

session.close();

This is handled instead by Spring, although no doubt it is calling Hibernate session management behind the scenes.

### Class/Entity Mapping

#### Mapping File Should Contain Only One Entity

Mapping files should only contain a single entity definition. This file should also include associations and named queries tied to the entity.

#### Use Named Parameters

Query parameters should always make use of the named parameter mechanism. Position dependant parameters are to be avoided.

Correct usage:

<query name="Employee.findByName">

<query-param name="name" type="string"/>

<![CDATA[

select emp from Employee emp where emp.name = :name

]]>

</query>

Incorrect usage:

<query name="Employee.findByName">

<![CDATA[

select emp from Employee emp where emp.name = ?

]]>

</query>

#### Use Comments to Document All Queries

<query> and <sql-query> HQL and SQL should be documented with a preceding comment which explains the business purpose of the query. For example:

<!-- Returns a list of TIF file IDs on which SLA action for

Acknowledgement response files has not been taken -->

<sql-query name="getAckSlaTIFIds">

If the query takes parameters whose meaning is not totally obvious, those should be explained in the comment also. The more complex the query, in most cases the more subtle it is and hence the more explanation required to truly state its purpose.

#### Associations with Code Value Tables

#### Tables that provide a static list of values should not have bi-directional mappings with the objects that are using those values.

<class name="com.htc.eis.epg.common.model.type.BusinessType" table="BUSN\_TYPE">

<id name="businessTypeCode" type="java.lang.String">

<column name="BUSN\_TYPE\_CD" length="1" />

<generator class="assigned" />

</id>

<property name="businessTypeDesc" type="java.lang.String">

<column name="BUSN\_TYPE\_DESC" length="50" not-null="true" />

</property>

<!—- No need for mapping these values to the objects using them (from this side of the mapping) -->

</class>

#### Bi-Directional Associations

When mapping bi-directional associations it is important to specify the “inverse=true” attribute on one side of the mapping. This tells Hibernate that those two mappings really equate to a single relationship, not two separate ones. This will prevent Hibernate from running the same SQL statement twice when saving those objects.

The java code for bi-directional associations should include helper methods so that you don’t manually have to set both sides of the association.

For example –

public class Company {

...

private Set addresses = new HashSet();

private void setAddresses(Set addresses) {

this.addresses = addresses;

}

private Set getAddresses() {

return addresses;

}

public void addAddress(Address address) {

address.setCompany(this); // Set the parent into the child

addresses.add(address); // Add the child to the parent collection

}

...

}

The concept of having helper functions for dealing with the collections of associations applies to other mappings as well, not strictly the bi-directional ones. It is actually preferable to make the getter/setter for the whole collection private and to have helper functions for the add/update/deletes to that collection. See the next point for more details.

#### Do Not Directly Expose Child Collections in Domain Objects

Although Hibernate itself deals with multiple children of an entity via a getter and setter for the internal collection that contains the multiple children (for example, a Supplier can have multiple SupplierLocation children), that does not mean the internal collection needs to be exposed in the domain object for direct use by application client code. Instead, certainly the setter used by Hibernate, and probably the getter, should be declared private. Then methods should be provided to enable application clients to do whatever operations are necessary, for example add(ChildEntity) and possibly remove(ChildEntity), removeAll() and size().

Exposing the Hibernate setter makes it very easy for the application client code to contain an error whereby it sets a new collection containing the child it wants to add, rather than first getting the existing collection and adding to that. Setting a new collection has the effect of deleting all the existing children from the database!

The purpose of declaring the Hibernate getter private is to prevent the application client from omitting the additional processing that may be going on in the methods specifically exposed to clients, for example the maintenance of bi-directional linkages in an add(ChildEntity) method as per the Bi-Directional Association example above.

Obviously, however, the child entities within the collection must be made available to the application client. The solution is to expose a different method which creates a new collection each time it is called, populated with the objects in the Hibernate-linked collection. For example, perhaps the Hibernate getter is private List getSupplierLocationList() and the application client getter is public List getSupplierLocations(). If a developer of a client class tried to circumvent the add(SupplierLocation) method by doing getSupplierLocations().add(SupplierLocation), he would quickly learn the correct way when he discovered that SupplierLocations added via the latter technique were not persisted.

Unfortunately, the techniques described here were not followed in the existing project code base and the internal collections are directly exposed. But we want to start using the proper techniques for new domain objects.

### Implementation

#### Always Provide a DAO Interface

This should require no additional explanation.

#### Write Stateless Implementations

Concrete DAO implementations should be written to be completely stateless. Thus they can be configured as Spring singletons (the Spring default).

#### Retrieval Methods Returning a Collection Should Never Return Null

Retrieval methods which query the database and can reasonably expect to return zero, one or more results, with the method return being defined as a collection of some sort (Collection, List, Set, etc.) should not return null if no matching entities are found. Instead, return an empty collection of the defined type. This will enable the calling code to just loop over the collection iterator, without having to test that the collection is not null.

For example, a method which returns a List can return Collections.EMPTY\_LIST, or a method which returns a Set can return Collection.EMPTY\_SET, if no matching entities are found.

#### Retrieval Methods Which Expect At Most One Entity

Hibernate HQL / SQL queries typically return a List. If the nature of the query is such that logically either zero or one entities can be found, but never more than one, the return value of the method should be the entity type, not a collection.

Within the DAO implementation that executes the Hibernate query, it should check that not more than one entity has been returned, and if so, throw a RuntimeException or subclass. Do not just take the first entity found and return that. If it should be logically impossible, then having more than one returned indicates a system bug or misconfiguration. This is valuable information, which throwing an exception makes obvious. If the problem is simply swallowed silently by returning the first of multiple entities, it is likely that the system behavior will be incorrect, and it will be harder to debug because the problem will likely become manifest at some unrelated point later in processing.

Similarly, if the nature of the query is such that logically there must be an entity found, if none is found throw a RuntimeException to highlight that fact.

#### Return Specific Java Types, Not Generic “Object”s

Hibernate SQL queries can be defined which return a disparate object array for each element of the List that Hibernate returns. This is the case when <return-scalar> is used in the <sql-query>, for example. This must be hidden within the DAO implementation. The return from the DAO method should be either a specific Java type, or a collection of specific Java types. The method caller should never have to pick through an Object array at any level and know that element 0 is one thing and element 1 another thing. This will typically require the developer to create a simple bean to contain the returned values.

This is basically the same idea as stated more generally in Development Standards 2.180 (“Collections and arrays should generally contain one type of element, not a miscellaneous group of objects of varying types”). However it is worth noting here because the Hibernate query itself returns just such a miscellaneous group of objects. The point is that the miscellaneous group of objects should be dealt with in one place only, namely inside the DAO method implementation, not by the clients of that DAO method.

## Best Practices

### Configuration

#### Use Managed Resources

Note: This area is being handled via the Spring / Hibernate / WebSphere transactional setup. If interested, you can trace through this starting in the \*spring-\*session.xml configuration files.

Applications should never configure a SessionFactory instance to use a non-container managed resource.

DataSource (or XADataSource if JTA is used) resources should be used. These resources should be referenced via JNDI. Additionally, access to the resource should be made by a naming reference, never directly (this reference is normally managed in a deployment descriptor).

For example:

<property name="Hibernate.connection.datasource">java:comp/env/jdbc/MyDataSourceRef</property>

An incorrect direct reference could look like:

<property name="Hibernate.connection.datasource">jdbc/MyDataSource</property>

### Session Management

#### Avoid the 'Open Session In View' Pattern

Note: The architecture is in complete agreement with the sentiment below, namely that OSIV as encouraged in the Hibernate literature is not a good idea. Architecture calls for transactions to be begun and ended within the service interface method. A transaction is not extended across multiple service calls and hence not extended across multiple HTTP request/response cycles. However, not quite sure how “managed contextual sessions” relate to the architecture with its heavy use of Spring as the main transaction coordinator.

The 'Open Session In View' (OSIV) pattern provides a faux contextual session typically managed by a Servlet Filter running within a webapp. This pattern poses a number of problems with nested transactions and for applications which expose interfaces not covered by the Servlet specification (ie: EJB, JMS). This pattern does not allow for distributed transactions. Implementors should avoid the use of this pattern and opt for a managed contextual session implementation (see below).

#### Use Managed Contextual Sessions

*From the Hibernate* [*project manual*](http://www.hibernate.org/hib_docs/v3/reference/en/html/architecture.html#architecture-current-session):

Starting with version 3.0.1, Hibernate added the SessionFactory.getCurrentSession() method. Initially, this assumed usage of JTA transactions, where the JTA transaction defined both the scope and context of a current session. The Hibernate team maintains that, given the maturity of the numerous stand-alone JTA TransactionManager implementations out there, most (if not all) applications should be using JTA transaction management whether or not they are deployed into a J2EE container. Based on that, the JTA-based contextual sessions is all you should ever need to use.

An example of a contextual session in action:

public void create(Employee employee) {

sessionFactory.getCurrentSession().create(employee);

}

Whereas a non-contextual session would be used this way:

public void create(Employee employee) {

Session session = sessionFactory.createSession();

try {

session.create(employee);

session.flush();

} finally {

if (session != null) {

session.close();

}

}

}

Use of a contextual session adds several major benefits:

* The application does not need to manage session state; that is, it does not need to manually flush and close each session obtained from the SessionFactory.getCurrentSession() method. This reduces the risks usually associated with managing resources programmatically.
* Compatible with nested and distributed transactions.
* Objects stay in an attached state after an operation.
* Improves performance by allowing lazy loading behavior to extend throughout the entire application.

If contextual sessions are used by an EJB application, it is important to note that the CMTTransactionFactory should be used over the JTATransactionFactory implementation.

As of Hibernate 3.1, contextual sessions are pluggable.

In the event a container does not support a JTA provider out of the box (ie: Tomcat), applications should be written using the traditional method of managing session state (described above).

If the application makes use of associations, lazy loading should be disabled in the class mapping file.

Applications should never make use of non-container managed means of transaction demarcation or management (ie: using Spring Framework's transaction mechanism or the Open Session In View pattern).

### Class/Entity Mapping

#### Mapping File Names

Mapping files should always be named after the class name of the mapped entity.

For example:

Employee.hbm.xml -> com.htc.example.employee.Employee

Client.hbm.xml -> com.htc.example.client.Client

Entity names should never be abstracted by package naming as most relational models do not support namespacing within the same schema. In the event of a naming conflict, it should be resolved by refactoring one or both entities to more correctly identify their purpose.

#### Centralize Mapping Resources

*there must not be two Hibernate mapping files with the same name, regardless of whether they are in different directories or not. If we were to follow the idea of a single directory (per project), I would want that directory to be different than WEB-INF or META-INF. In other words, disagree with using WEB-INF or META-INF specifically.*

Hibernate mapping files (.hbm.xml) should be placed in a centralized location in the application classpath. This aids in keeping the entity namespace flat as no two entities can be named alike due to filesystem constraints.

Standard mapping file locations:

Web Applications -> WEB-INF/Hibernate

EJB Applications -> META-INF/Hibernate

#### Externalize Queries

All non-CRUD queries should be externalized into the Mapping File using the named query mechanism. The query name should follow the following syntax:

<Entity Name>.<Method Name>

For example, if an operation was needed to find a number of employees by name, the named query should be named:

Employee.findByName

#### Avoid HQL Shorthand Form

HQL supports a short-hand form of creating queries by leaving out the first part of the select clause. To avoid confusion, do not create queries with any part of the statement omitted.

Correct usage:

select emp from Employee emp where emp.name = :name

Incorrect usage:

from Employee emp where emp.name = :name

#### DDL Markup

The DDL attributes in the Hibernate mapping files are used largely by the Auto-export tool. Their use can provide documentation on the database schema without needing to refer to the db schema.

Certain attributes however, are directly used by Hibernate and are beneficial to provide. For example, the “not-null” attribute should be set to true on columns that are specified that way in the database. Hibernate will then verify a value is present in the object prior to trying to persist it to the database. This will save a trip to the db when a failure is imminent anyway.

Usage:

<property name="name">

<column name="EMP\_NAME" not-null="true" length="32" />

</property>

#### Use Type-safe Enumerations

Type-safe enumerations are the preferred approach for maintaining a list of static values which may be applied to a specific domain within an application. The explanation of this concept is beyond the scope of this article.

The ebusiness-common project provides a J2SE 1.4 compatible type-safe enumeration implementation. The ebusiness-Hibernate project provides a UserType implementation to support this construct.

To use a type-safe enumeration implementation from Hibernate using the ebusiness-common implementation, the following typedef should be defined in the mapping file:

<typedef class="com.htc.eis.ebiz.Hibernate.type.EnumType" name="status">

<param name="category">com.htc.example.Status</param>

<param name="componentType">string</param>

</typedef>

Then apply the type to a property:

<property name="status" column="EMP\_STATUS" type="status"/>

The category param is the fully qualified classname of the Enum implementation.

The componentType is a Hibernate type which corresponds to the type of data held in the enumeration. While the ebusiness-common implementation does allow for disparate component types within the same enumeration, the EnumType does not.

It should be noted that it is generally not good practice to mix component types within the same enumeration.

### Implementation

#### Update Operations Should Return void

Update operations (Create, Update, Delete) should be defined as returning void, and accepting the entity as the only argument.

#### Use Consistent Naming

CRUD operations should follow the method naming guidelines:

create() -> Create

findByPrimaryKey() -> Read

update() -> Update

delete() -> Delete

Additionally the following considerations should be made when naming query methods:

* If the query uses a single criterion, use it in the name (ie: findByName)
* Use the entity name if multiple criterion are used (ie: findEmployee)
* Overload the method name when disparate sets of criterion are used (ie: multiple findEmployee methods)
* Use plural form when returning a collection (ie: findEmployees)

For example:

public interface EmployeeDao {

void create(Employee employee);

void update(Employee employee);

void delete(Employee employee);

Employee findByPrimaryKey(long pk);

Employee findByName(String name);

List findEmployees(int age);

List findEmployees(int age, int yearsOfService);

}

#### Use Constructor Injection *.*

Concrete Hibernate DAO implementations should be written to accept a SessionFactory in the constructor. This should be the sole dependency of the DAO impl. Please note that it is not an acceptable practice to use setter-injection.

Correct usage:

public class HibernateEmployeeDao implements EmployeeDao {

private SessionFactory sessionFactory;

public HibernateEmployeeDao(SessionFactory sessionFactory) {

this.sessionFactory = sessionFactory;

}

...

}

Incorrect usage:

public class HibernateEmployeeDao implements EmployeeDao {

private SessionFactory sessionFactory;

public void setSessionFactory(SessionFactory sessionFactory) {

this.sessionFactory = sessionFactory;

}

...

}

#### Never return null

DAO operations should never return a null reference. By default, the Query.uniqueResult() method can return null if no results are returned by the executing query.

The ebusiness-Hibernate project provides a helper class to ensure that an exception is thrown if no results are returned by a query: Queries.requireResult().

Correct usage:

Query query =

session.getNamedQuery("Employee.findByName");

query.setString("name", "Joe Brown");

return (Employee) Queries.requireResult(query);

Incorrect usage:

Query query =

session.getNamedQuery("Employee.findByName");

query.setString("name", "Joe Brown");

return (Employee) query.uniqueResult(); //possible null

# EJB 3.0

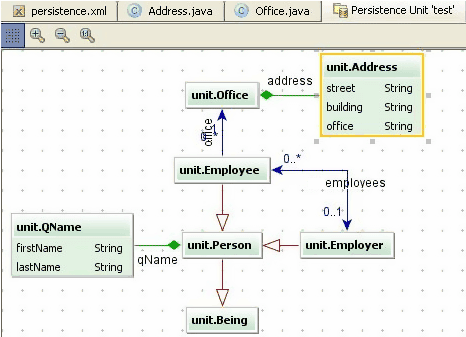
IntelliJ IDEA features rich support for EJB development. Supporting EJB specifications from 1.x to 3.0 and leveraging it through all its features, from coding assistance to refactoring, IntelliJ IDEA stands for the weapon of choice for developing EJB applications.

IntelliJ IDEA leverages all EJB in the dedicated module type — the EJB Module, which lets you handle EJB and relevant objects (ejb-jar.xml and other descriptors, EJB security and EJB deployment settings, etc.) under one functional entity. This simplifies EJB applications development, debugging and deployment.

#### EJB 3.0 specific features

IntelliJ IDEA fully supports annotation mechanism for creating EJB, and Interceptors, with automatic code generation, completion and dedicated binding editor.

EJB persistence support is powered with generating of persistence mapping from entity beans, Hibernate or JDBC source and the visual Persistence diagram builder that lets you get the complete picture of how your persistence entities relate one to another.



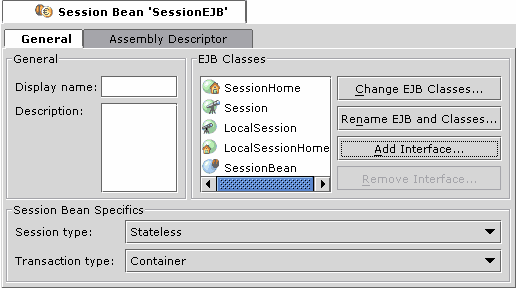
Additionally, IntelliJ IDEA supports entity listeners and Embeddable Enterprise Beans.

For your existing EJB projects (versions 1.x and 2.x), IntelliJ IDEA provides the migration to EJB 3.0 that includes:

* Converting EJB environment access
* Rebuilding EJB deployment descriptors
* Transforming EJB interfaces
* Turning Entity Beans into Container Managed Persistence

#### General EJB features

IntelliJ IDEA eliminates routine work through generating stub code for entity EJB, session EJB, message EJB and other EJB types, CMP EJB fields and EJB relationships. Dedicated context editors also help modify EJB and their properties.

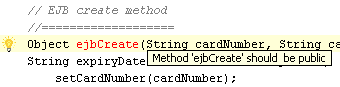


IntelliJ IDEA automatically builds standard EJB deployment packages either in JAR or exploded directory format and automatically generates appropriate EJB XML descriptors for all EJB included in the module (ejb-jar.xml).

#### EJB-aware coding assistance

IntelliJ IDEA leverages code completion for both EJB code and descriptor files and also supports

EJB error highlighting: IntelliJ IDEA continuously checks your EJB source code for EJB specification compliance, so that all possible inconsistencies are immediately detected and highlighted in the editor.



EJB intention actions and quick fixes: For most EJB errors detected in your code, IntelliJ IDEA will provide you with the ability to automatically correct the erroneous code by showing an intention action light bulb in the editor that expands into a list of possible corrections.

#### Powerful EJB-aware refactoring

All global refactorings, like Rename, Change Method Signature, etc., are aware of the EJB specification and will therefore correct all necessary references in your code so that your EJB structure is not broken. Refactorings also apply to all XML descriptors to ensure the total integrity.

# Database Independency Rules

| Rules |
| --- |
| [JDBC\_001: Do Not Hard Code A JDBC Driver Class Name](mhtml:file://C:\Documents%20and%20Settings\Chandra\Desktop\response\jjguidelines%20Database%20Independency%20Rules.mht!https://jjguidelines.dev.java.net/book/html/apbs03.html#JDBC_001) |
| [JDBC\_002: Do Not Hard Code A JDCB Connection URL](mhtml:file://C:\Documents%20and%20Settings\Chandra\Desktop\response\jjguidelines%20Database%20Independency%20Rules.mht!https://jjguidelines.dev.java.net/book/html/apbs03.html#JDBC_002) |
| [JDBC\_003: Do Not Use A Driver Managment Method Of java.sql.DriverManager](mhtml:file://C:\Documents%20and%20Settings\Chandra\Desktop\response\jjguidelines%20Database%20Independency%20Rules.mht!https://jjguidelines.dev.java.net/book/html/apbs03.html#JDBC_003) |
| [JDBC\_004: Do Not Import A JDBC Vendor Specific Class](mhtml:file://C:\Documents%20and%20Settings\Chandra\Desktop\response\jjguidelines%20Database%20Independency%20Rules.mht!https://jjguidelines.dev.java.net/book/html/apbs03.html#JDBC_004) |
| [JDBC\_005: Close Connection, Statement And ResultSet](mhtml:file://C:\Documents%20and%20Settings\Chandra\Desktop\response\jjguidelines%20Database%20Independency%20Rules.mht!https://jjguidelines.dev.java.net/book/html/apbs03.html#JDBC_005) |
| [JDBC\_006: Close JDBC Resources In The Correct Order](mhtml:file://C:\Documents%20and%20Settings\Chandra\Desktop\response\jjguidelines%20Database%20Independency%20Rules.mht!https://jjguidelines.dev.java.net/book/html/apbs03.html#JDBC_006) |
| [JDBC\_007: Use PreparedStatement Instead Of Statement When Possible](mhtml:file://C:\Documents%20and%20Settings\Chandra\Desktop\response\jjguidelines%20Database%20Independency%20Rules.mht!https://jjguidelines.dev.java.net/book/html/apbs03.html#JDBC_007) |
| [JDBC\_008: Check The Return value Of A Navigation Method Of ResultSet](mhtml:file://C:\Documents%20and%20Settings\Chandra\Desktop\response\jjguidelines%20Database%20Independency%20Rules.mht!https://jjguidelines.dev.java.net/book/html/apbs03.html#JDBC_008) |
| [JDBC\_009: Use Standard SQL Only](mhtml:file://C:\Documents%20and%20Settings\Chandra\Desktop\response\jjguidelines%20Database%20Independency%20Rules.mht!https://jjguidelines.dev.java.net/book/html/apbs03.html#JDBC_009) |
| [JDBC\_010: Check For A Nested SQLException](mhtml:file://C:\Documents%20and%20Settings\Chandra\Desktop\response\jjguidelines%20Database%20Independency%20Rules.mht!https://jjguidelines.dev.java.net/book/html/apbs03.html#JDBC_010) |
| [JDBC\_011: Check For A SQLWarning](mhtml:file://C:\Documents%20and%20Settings\Chandra\Desktop\response\jjguidelines%20Database%20Independency%20Rules.mht!https://jjguidelines.dev.java.net/book/html/apbs03.html#JDBC_011) |

### JDBC\_001: Do Not Hard Code A JDBC Driver Class Name

Don't hard code the database driver class name when loading it using the Class.forName method. It is better store the driver class name in an external configuration file that is read when the database connection must be created.

### JDBC\_002: Do Not Hard Code A JDCB Connection URL

Don't hard code the database connection URL. It is better to store this URL in an external configuration file that is read when the database connection must be created.

If possible try not include the username and password in the URL but store them separately. This allows you to encrypt your password and to reuse the same username password for different URLs (in the case where you have to support more than one database).

### JDBC\_003: Do Not Use A Driver Managment Method Of java.sql.DriverManager

The java.sql.DriverManager is responsible for managing the known JDBC driver classes. Although the DriverManager has some method related to the management of drivers (the register and deregister methods) they should never be called directly.

The JDBC API states that all driver implementation should have a static initializer that calls the DriverManager.register method when the driver class is loaded. This means that when a driver is loaded there is no need to register it with the DriverManager.

Calling DriverManager.deregister is even more dangerous since this can seriously disrupt the DriverManager's internal bookkeeping.

### JDBC\_004: Do Not Import A JDBC Vendor Specific Class

If your code needs to be database independent it is important that you do not import classes that are specific to one JDBC driver vendor. Using these imports means that you probably use database specific features that will prevent you from moving to another type of database without changing your code.

Stick to the classes in the java.sql and javax.sql packages.

### JDBC\_005: Close Connection, Statement And ResultSet

All JDBC resources (Connection, Statement, ResultSet) must be closed when they are no longer used. If you don't do this you can introduce memory leaks and possible disrupt the functionality of the database. For example you can have too many open connections on your database, to many open cursors in your database, etc.

Always close you resources in a finally clause of a try block so you know that they always get closed. See rule [JAC\_068: Close A Connection Inside A finally Block (Enforced)](mhtml:file://C:\Documents%20and%20Settings\Chandra\Desktop\response\jjguidelines%20Database%20Independency%20Rules.mht!https://jjguidelines.dev.java.net/book/html/apas04.html#JAC_068).

WRONG

public String[] getPersons(DataSource ds) throws SQLException {

Connection conn = datasource.getConnection();

Statement stat = conn.createStatement();

ResultSet rs = stat.executeQuery("Select \* from Person");

while (rs.hasNext()) {

// ...

}

rs.close();

stat.close();

conn.close();

// return something

}

RIGHT

public String[] getPersons(DataSource ds) throws SQLException {

Connection conn = null;

Statement stat = null;

ResultSet rs = null;

try {

conn = datasource.getConnection();

stat = conn.createStatement();

rs = stat.executeQuery("Select \* from Person");

while (rs.hasNext()) {

// ...

}

} finally {

if (rs != null) {

rs.close();

}

if (stat != null) {

stat.close();

}

if (conn != null) {

conn.close();

}

}

// return something

}

|  |  |
| --- | --- |
| [Tip] | Tip |
| The code in the finally clause can become messy if you also have to catch the SQLException which is thrown when you close a resource. One technique to get rid of the messy code in the finally block is to use provide methods that will take care of closing these resources for you.  For example:  public void close(Connection conn, Statement stat,  ResultSet rs) throws SQLException {  if (rs != null) {  rs.close();  }  if (stat != null) {  stat.close();  }  if (conn != null) {  conn.close();  }  }  This method can be made available in for example a JDBC util class. This method also guarantees that the resources are closed in the correct order (see [JDBC\_006: Close JDBC Resources In The Correct Order](mhtml:file://C:\Documents%20and%20Settings\Chandra\Desktop\response\jjguidelines%20Database%20Independency%20Rules.mht!https://jjguidelines.dev.java.net/book/html/apbs03.html#JDBC_006)). | |

### JDBC\_006: Close JDBC Resources In The Correct Order

The JDBC resources must be closed in the correct order:

1. ResultSet
2. Statement
3. Connection

### JDBC\_007: Use PreparedStatement Instead Of Statement When Possible

Always try to use java.sql.PreparedStement instead of the standard java.sql.Statement. Especially when executing SQL statements accepting arguments. Besides the fact that prepared statements get precompiled there is the fact the prepared statements handle the formatting of argument values for you. For example one database could format its string using single quotes where another one may use double quotes.

Another added value is the fact that you don't have to bother about writing long, error prone and complex string concatenation code.

WRONG

Date date = new Date();

// Format the date using the correct pattern (Note: the pattern

// used by SimpleDateFormat is not equal to that used by SQL-99)

SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");

String dateString = sdf.format(date);

// Build the SQL statement (using a StringBuffer is more

// efficient)

StringBuffer sql = new StringBuffer("update Person ");

sql.append("set dob=to\_date('");

sql.append(dateString);

sql.append("', 'dd-mm-yyyy') where name=");

sql.append("'");

sql.append(pName);

sql.append("'");

//execute the statement

connection.createStatement();

statement.executeUpdate(sql.toString());

Using a PreparedStatement, we don't need that much code and we delegate the formatting to the driver.

RIGHT

Date date = new Date();

//prepare the statement

statement = connection.prepareStatement(

"update person set dob=? where name=?");

//bind the parameters

statement.setDate(1, date);

statement.setString(2, pName);

//execute the statement

statement.executeUpdate();

### JDBC\_008: Check The Return value Of A Navigation Method Of ResultSet

Always look at the return value after calling one of the ResultSet navigation methods; never assume that results will be available.

ResultSet navigation methods include: next, first, last, previous

WRONG

Statement stat = conn.createStatement();

ResultSet rs = Stat.executeQuery("SELECT name FROM person");

rs.next();

String firstName = rs.getString(1);

RIGHT

Statement stat = conn.createStatement();

ResultSet rs = Stat.executeQuery("SELECT name FROM person");

if (rs.next()) {

String firstName = rs.getString(1);

} else {

// error handling

}

### JDBC\_009: Use Standard SQL Only

Stick to the most common SQL standard when writing SQL statements. Don't use vendor specific extensions. If you do so you will be stuck to the vendor's database.

The most common standard (at the time this document was written) is SQL-92.

### JDBC\_010: Check For A Nested SQLException

The java.sql.SQLException has always been chainable (even before the exception chaining introduced in JDK1.4). The exception chain can help you in understanding what went wrong when some JDBC API method threw a SQLException. Walking trough the exception chain is done using the SQLException.getNextException method.

WRONG

try {

Statement stat = connection.createStatement();

ResultSet rs = stat.executeUpdate("SELECT name FROM person");

// process rs

} catch (SQLException e) {

// printStackTace used as an example

e.printStackTrace();

}

RIGHT

try {

Statement stat = connection.createStatement();

ResultSet rs = stat.executeUpdate("SELECT name FROM person");

// process rs

} catch (SQLException e) {

// printStackTace used as an example

while (e != null) {

e.printStackTrace();

e = e.getNextException();

}

}

### JDBC\_011: Check For A SQLWarning

The main JDBC API interfaces provide access to a possible warning (SQLWarning) that was generated. Besides logging the warning there is not much you can do with it.

A suggestion is to check for warnings when you close the resources. See rule [JDBC\_006: Close JDBC Resources In The Correct Order](mhtml:file://C:\Documents%20and%20Settings\Chandra\Desktop\response\jjguidelines%20Database%20Independency%20Rules.mht!https://jjguidelines.dev.java.net/book/html/apbs03.html#JDBC_006). This way logging possible warnings could be added to the helper method that can be used to close the Connection, Statement and

# Best Practices to Prevent Vulnerability Attacks

Cross-Site Scripting attacks are a type of injection problem, in which malicious scripts are injected into the otherwise benign and trusted web sites. Cross-site scripting (XSS) attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user in the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user’s browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site. These scripts can even rewrite the content of the HTML page.

## SQL Injection

The opening for SQL-injection attacks comes primarily through Web applications that combine user input with dynamic SQL to form SQL commands that the application sends to the database. Here are four important steps you can take to protect your Web applications from SQL-injection attacks.

### Principle of Least Privilege

The account an application uses to connect to the database should have only the privileges that application requires. The security permissions that an intruder gains from a compromised application define the harm that the intruder can inflict. Applications shouldn't connect as sa or with the Administrator account. Instead, the account should have permissions to access only the database objects it needs.

### Validate All Input

If an input field should contain numeric data, then verify that users enter only numbers. If character data is acceptable, check for unexpected characters. Make sure your application looks for characters such as semicolons, equals signs, double dashes, brackets, and SQL keywords. Regular expressions can use used to handle complex pattern matching This should be used to check user input.

### Avoid Dynamic SQL

Combining dynamic SQL with user input creates exposure that makes SQL-injection attacks possible. Replacing dynamic SQL with prepared SQL or stored procedures is feasible in most applications. Prepared SQL and stored procedures accept user input as parameter data rather than as SQL commands, thus limiting what an intruder can do. Of course, replacing dynamic SQL with a stored procedure won't help you if you use the user input to build dynamic SQL statements in your stored procedures. In that case, the dynamic SQL that the user input creates will still be corrupted, and your database will still be in danger of SQL-injection attack.

### Use Double Quotes

Replace all the single quotes that your users' input contains with double quotes. This simple precaution will go a long way toward warding off SQL-injection attacks. Single quotes often terminate SQL expressions and give the input more power than is necessary. Replacing the single quotes with double quotes will cause many SQL-injection attacks to fail.

## Buffer overflow

It occurs usually with fixed length buffers when some data is going to be written beyond the boundaries of the current defined capacity. This could lead to mal functioning of the system since the new data can corrupt the data of other buffers or processes could be altered in order to execute the injected code and take control of the system. The best practices to be followed to prevent this problem are -

* Performing bounds checking
* Checking configuration files
* Checking command-line parameters
* Checking environment variables
* Setting initial values for data
* Monitoring logs
* Implementing file integrity solutions
* Using stack protection

## Parameter Tampering

The best way to prevent parameter tampering is to ensure that all parameters are validated before they are used. A centralized component or library is likely to be the most effective, as the code performing the checking should all be in one place. Each parameter should be checked against a strict format that specifies exactly what input will be allowed. "Negative" approaches that involve filtering out certain bad input or approaches that rely on signatures are not likely to be effective and may be difficult to maintain.

Parameters should be validated against a "positive" specification that defines:

* Data type (string, integer, real, etc...)
* Allowed character set
* Minimum and maximum length
* Whether null is allowed
* Whether the parameter is required or not
* Whether duplicates are allowed
* Numeric range
* Specific legal values (enumeration)
* Specific patterns (regular expressions)

## Cookie Poisoning

On the Web, cookie poisoning is the modification of a cookie (personal information in a Web user's computer) by an attacker to gain unauthorized information about the user for purposes such as identity theft. The attacker may use the information to open new accounts or to gain access to the user's existing accounts.

Cookies should be protected by encryption. Cookie encryption creates a digital signature that is used to validate the content in all future communications between the sender and the recipient. If the content is tampered with, the signature will no longer match the content and will be refused access by the server.

## Cross Site Scripting or XSS

The following list outlines the general approaches to prevent cross-site scripting attacks:

* Encode output based on input parameters.
* Filter input parameters for special characters.
* Filter output based on input parameters for special characters.

Any string that is inserted into a page must have the following characters replaced with the corresponding HTML/SGML entities:

Convert < into &lt;

Convert > into &gt;

Convert & into &amp;

Convert " into &quot;

Convert ' into &#39;

## Session hijacking

To offer protection against these threats, ASP.NET forms authentication provides the following countermeasures:

* **Hashed MACs (HMACs).** These use either SHA1 or MD5 to provide tamper-proofing. Any changes to the authentication ticket are detected at the server and an exception is thrown if it has been modified.
* **Encryption.** Encryption turns the clear text data contained in the forms authentication ticket into unintelligible cipher text. ASP.NET version 2.0 uses AES symmetric encryption to prevent anyone from viewing the contents of the forms authentication ticket.
* **Enforced transmission over HTTPS.** You can prevent authentication tickets being transmitted over HTTP connections. This prevents an attacker from being able to view or modify the authentication ticket while it crosses the network.

## OWASP References OWASP-Top 10 Vulnerability and best coding practices to prevent Vulnerability.

* 