Source Code Commenting Instruction Manual

Version : Draft

Date : *12-08-2010*
Doc. Type : *Procedure*
Current Status : *Draft*

Revision history

Doc No:	Version no.	Change reference no.	Author	Published date	Sections changed	Description of changes
	Draft			12-08-2010		Draft

Topics Covered

Appendix:

Purpose of the document

This document describes a collection of standards, conventions, and guidelines that forms the working instructions for writing comments. They main motive is to instruct how to write proper comments that lead to code that is easy to understand, to maintain, and to enhance.

This document aims to serve as a single source containing instructions for proper code commenting. Following a common set of commenting standards will lead to a greater consistency, making teams of developers significantly more productive.

All the examples in this document have been taken from the IssueTrack project.

Scope of the document

This document is only for writing comments in the best possible way. It does not cover the following points

- Code conventions. It aims to prescribe rules for writing comments which will explain what is coded not why or how a code is written. This issue of coding conventions and standards will be addressed in a separate document.

- Guidelines for writing software documentation.

Desired outcome

- Writing comments that are easy to maintain and enhance.
- Increased developer's productivity by having a common set of guidelines for comments.

Targeted recipient

Developers currently working with CCS' existing projects as well as new developers who are going to start coding either in existing or new projects.

Prerequisites

Developers with knowledge of coding in Java, JSP, XML and any other associated technology being used in CCS and that requires proper commenting.

Importance of Comments

The importance of comments can be summed up in one line – *"If your code is not worth commenting, then it's not worth running"*.

Commenting is a very vital part of writing highly maintainable, flexible and robust software applications. Programmers rarely work on the same project all the time. Also code rarely remains same throughout its life time. New functionality, production bugs, efficiency improvements etc. all these call for change in codes. A new programmer exposed to an existing code normally finds hard to understand the code just by looking at the code at the very first time. Worst if the original developer is not available, which is not a rare case, and then the situation turns into a development nightmare. Also it is not abnormal for the original developer to forget about the code if he has been away from it for a while.

Comments helps us to recollect instantly why a particular code was written the way it is written, how is the particular block is executed, what special techniques has been used etc.

Developers generally complain that commenting codes takes a lot of time and hence they either ignore it completely or write sub-standard comments. The price of which has to be paid later when it comes to maintenance and other bug fixing activities.

The way that you comment your code has a huge impact both on your own productivity and on the productivity of everyone else who later maintains and enhances it. By documenting your code early in the development process you become more productive because it forces you to think through your logic before you commit it to code. Furthermore, when you revisit code that you wrote days or weeks earlier you can easily determine what you were thinking when you wrote it – it is documented for you already.

Types of comments

Java has three styles of comments:

- **Documentation comments** start with /** and end with */.
- **C-style comments** start with /* and end with */.
- **Single-line comments** that start with // and go until the end of the source-code line.

In the chart below is a summary of suggested use for each type of comment, as well as several examples.

Comment Type	Usage	Example
Documentation	Use documentation comments immediately before declarations of interfaces, classes, member functions, and fields to document them. Documentation comments are processed by javadoc.	/** Customer – A customer is any person or organization that we sell services and products to. */
C style	Use C-style comments to document out lines of code that are no longer applicable, but that you want to keep just in case your users change their minds, or because you want to temporarily turn it off while debugging.	/* This code was commented out on Aug 8, 2010 because it was replaced by the preceding code. Delete it after two years if it is still not applicable. . . . (the source code) */

Single line	Use single line comments internally within member functions to document business logic, sections of code, and declarations of temporary variables.	// Apply a 5% discount to all invoices
		// over $1000 as defined by the Sarek
		// generosity campaign started in
		// Aug. of 2010.

General Instructions for writing comments

The following table list down some common instructions that should be followed in general while commenting codes.

#	Instruction	Rationale	How to do	Example[1]	Output in Generated Javadoc
1	Separate blocks of comments into separate paragraphs	Improves readability in the generated Javadoc	Start the block with and end with tags.	/** * Purpose : * The CreateTicketDataBean is a JSF *Managed Bean that is responsible for * ---------- * */	Purpose: The CreateTicketDataBean is a JSF Managed Bean that is responsible for …
2	Make the main headers in a new block bold	-do-	Start the header with and end with tags	/** * Purpose : * The CreateTicketDataBean is a JSF *Managed Bean that is responsible for * ---------- * */	**Purpose:** The CreateTicketDataBean is a JSF Managed Bean that is responsible for…
3	If a block of comment contains more than a single point then always represent them as a list.	-do	Start the list with and end with and all the listed items should be preceded by within the and .	/** * * Use Case supported - * Creation of a * new ticket Saving of attachments * Sending email of the created tkt * */	**Use Case supported** - 1. Creation of a new ticket 2. Saving of attachments 3. Sending email of the created tkt
4	Remove unwanted comments generated by Netbeans IDE	Unnecessary comments makes code unreadable and also waste someone's time reading the code	Whenever a new file like Java class, Managed bean etc are created then NetBeans puts some	/** * This method is called when this *bean is removed from request scope. .. …….. */ The above method appears above each	N/A

[1] Examples are taken from the *IssueTrack* application.
For details on Javadoc, refer to Annexure - I

#	Instruction	Rationale	How to do	Example[1]	Output in Generated Javadoc
			instructional comments above the class and some methods, created based on the situation. Simply select and delete them	"destroy" method of a JSF managed bean. Since this is supposedly known to every programmer and doesn't convey any useful information it should be deleted.	
5	Comments for TODO tasks[2]	• Comments for tasks that need to be reworked later like unfinished tasks etc should be preceded with a single line comment starting with the word "TODO". • NetBeans can output a list of all those tasks in a list.	• Use single line comment[3] starting with the word "TODO"	//TODO the following line //should be removed in production //release System.out.println("Inside create method");	N/A

6	Simple Getters and Setters should not be commented	• Simple Getters and Setters that doesn't have any manipulation inside are self describing[4] and hence need not be commented.
		• This is basically done to avoid
			• N/A	public String getStatusFinal() {
				return statusFinal;
				}
				
				public void setStatusFinal(String statusFinal) {
				this.statusFinal = statusFinal;
				}
					N/A

[2] This is applicable only for code written in NetBeans IDE
[3] To know about different types of comments in Java, refer to the chapter *Types of Comments*
[4] Since the variables are commented separately

#	Instruction	Rationale	How to do	Example[1]	Output in Generated Javadoc
		unnecessary			
7	Use *"editor-fold"* comment to group related blocks like all user attributes, methods related to the same tasks etc. with a proper description	• This helps to fold the whole code. • The developer can quickly browse and find the block related to a specific use case.	• Use the single line comment followed by "<editor-fold defaultstate="collapsed" desc="*User Defined Description*[5]">" • As an illustration all the methods related to a single functionality, say data persistence can be grouped together.	// <editor-fold defaultstate="collapsed" desc="Getters and setters method">	N/A

[5] Replace *"User Defined Description"* with the description that you want to appear in the collapsed block

Commenting Class

Commenting of a class should be done with outmost care as they are the major artifacts of your source code. They should be placed immediately after the import statements and preceding the class declaration

The points to be included while commenting a class is given in the following table and has been broadly categorized as compulsory and special comments. It is then followed by an example to further illustrate the points.

Compulsory comments should be included in every class declaration whereas special comments are to be included in special conditions as need arises.

Type	C	Rationale – Why comment	How to comment	Jd	Jd Kw	Example[1]
Purpose	Y	A general description states the purpose of its existence so that other developers can quickly determine its functionality and whether or not it meets their needs	• The general purpose • Any other important information like is it a part of any pattern or is it special type of class like container Managed Bean or a Backing Bean etc.	Y	N/A	/** * Purpose : * The CreateTicketDataBean is a JSF *Managed Bean that is responsible for * handling all the tasks related to * the creation of a new ticket. It is a JSF * ManagedBean * */
Use Case supported	Y	A class exists to implement a particular use case or it may support multiple use case. Describing the use cases for which it is designed gives clarity to the reader.	• Description of the Use Cases implemented • Any other supported use cases if any i.e. if this class is being used by any other class	Y	N/A	/** * CreateTicketDataBean.java * ---------- * ---------- * * Use Case supported -

[1] The examples given under this column makes use of some HTML tags whose effects can be seen in the generated JavaDoc. For details on these tags please refer to the chapter titled *"General Instructions for writing comments"*.

Type	C	Rationale – Why comment	How to comment	Jd	Jd Kw	Example[1]
						* Creation of a * new ticket Saving of attachments * Sending email of the created tkt * */
Known bugs	N	• If there are any outstanding problems with a class they should be documented so that other developers understand the weaknesses/difficulties with the class. • If a bug is specific to a single method then it should be directly associated with the method instead.	• Description of the bug and the expected behavior. • Specific methods which include the bug • Reason for not fixing the bugs • Workaround adopted, if any	Y	N/A	/** * CreateTicketDataBean.java * ---------- * ---------- * * Known Bugs – Even if a * user don't upload any file it *persists null values * to the database whereas it should * persist only when files are selected. * Its not a fatal bug and hence not * fixed but needs to be done. */
Applicable preconditions and postconditions (Invariant)	N	• A precondition is a constraint under which a method will function properly, and a postcondition is a property or assertion that will be true after a member function is finished running. Specifying these will help a developer who intends to use it to have a clear picture. • Also referred to as an invariant it is a set of assertions about an instance or class that must be true at all "stable" times.	• Describe the invariants • Describe the state of the invariant before and after the object and its method is invoked.	Y	N/A	/** * CreateTicketDataBean.java * ---------- * ---------- * *Invariant details -Ticket * Number - The value of the ticket * number should be one more then * the existing ticket number *

Type	C	Rationale – Why comment	How to comment	Jd	Jd Kw	Example[1]
		• A stable time is defined as the period before a method is invoked on the object/class and immediately after a method is invoked. • Documenting the invariants of a class provides valuable insight to other developers as to how a class can be used				
Multithreading details (Concurrency strategy)	N	• Concurrent programming is a complex topic that is new for many programmers; therefore you need to invest the extra time to ensure that people can understand your work. • It is important to document your concurrency strategy and why you chose that strategy over others.	• Describe the method that implements all the work to be done in a separate thread and what all methods it contains.	Y	N/A	/**MailerDataBean.java *---------- *---------- * Multithreading *details - The work of *sending mail is being separated *out as separate thread of work * to avoid system hang up */
Details of implementing interfaces	N	• Since interfaces define behavior hence it becomes crucial to mention why a particular interface(s) is being implemented in the class.	• List of implemented interfaces with their needs	Y	N/A	/** * MailerDataBean.java * *Interfaces Implemented - *Runnable. This *interface is being *implemented for concurrent behavior of *some tasks of this class, mainly *the mail send task */
Details of	N	• Information regarding the parent	• Describe the extended class needs.	Y	N/A	/**

Type	C	Rationale – Why comment	How to comment	Jd	Jd Kw	Example[1]
parent/base classes		class is necessary because knowing this will help the developer know what all behavior this class already possesses through inheritance.				* CreateTicketDataBean.java * ---------- * ---------- * *Extended Class- *AbstractPageBean. This class is being *extended to inherit JSF managed bean *characteristics. */
Overridden methods	N	• In case of inheritance, if any method(s) is overridden then it should be clearly listed down so that any one reading the code doesn't expect default behavior. • Only list down the names of the methods overridden. Details should be mentioned in method commenting	• List down the overridden methods name and the reason behind overriding.	Y	N/A	/** * CreateTicketDataBean.java * ---------- * ---------- * *Overrides- init method of *AbstractPageBean */
Object initialization details (or static block)	N	• When an object of a class is initialized then sometimes we also do some initialization. This is very crucial information to document as this will help anyone using the class. • This initialization info should not be limited to only constructors since many classes, for example static blocks and init methods of JSF Managed Beans.	• List down the initialization block tasks if it is within a static block, since a static block doesn't have any name. • If it is an init block of JSF ManagedBean then list down the block name (mostly init) and the tasks it does.	Y	N/A	/** *HibernateUtil.java * *Initialization Block - *Creation of a new session *factory object * */

Type	C	Rationale – Why comment	How to comment	Jd	Jd Kw	Example[1]
Last modified	Y	• This is to track down last when the last modification was done. • Importance of this is that one can have an idea about the stability of the class. If the last modified date is too recent then it means that some issues have cropped up in the recent past.	• @since – *last modified date in dd-mm-yyyy format*	Y	@since	/** * CreateTicketDataBean.java * ---------- * ---------- *@since 26/06/2010 */

Example

```
/**
 *
 * <p><strong>Purpose : </strong>
 * The CreateTicketDataBean is a JSF Managed Bean that is responsible for
 * handling all the tasks related to the creation of a new ticket. It is a JSF ManagedBean
 *</p>
 * <p><strong>
 * Use Case supported </strong> -<ol><li>Creation of a new ticket<li>Saving of attachments
 * <li> Sending email of the created tkt </ol>
 * <p><strong>
 * Known Bugs</strong> – Even if a user don't  upload any file it persists null values
 * to the database whereas it should persist only when files are selected.Its not a fatal bug and hence not
 * fixed but needs to be done. </p>
 *<p><strong>
 *Invariant details </strong>-Ticket Number - The value of the ticket number should be one more than
 * the existing ticket number
 *Extended Class</strong>- AbstractPageBean. This class is being extended to inherit JSF managed bean
 * characteristics.</p>
 *Overrides</strong>- init method of AbstractPageBean. This is done for all the initialization work</p>
 *@since 26/06/2010
 */
```

Generated Javadoc

issuetrack.src.dataBean

Class CreateTicketDataBean

```
java.lang.Object
  └com.sun.rave.web.ui.appbase.FacesBean
      └com.sun.rave.web.ui.appbase.AbstractRequestBean
          └issuetrack.src.dataBean.CreateTicketDataBean
```

```
public class CreateTicketDataBean
extends com.sun.rave.web.ui.appbase.AbstractRequestBean
```

Purpose : The CreateTicketDataBean is a JSF Managed Bean that is responsible for handling all the tasks related to the creation of a new ticket. It is a JSF ManagedBean

Use Case supported -

1. Creation of a new ticket
2. Saving of attachments
3. Sending email of the created tkt

Known Bugs – Even if a user don't upload any file it persists null values to the database whereas it should persist only when files are selected.Its not a fatal bug and hence not fixed but needs to be done.

Invariant details -Ticket Number - The value of the ticket number should be one more than the existing ticket number Extended Class- AbstractPageBean. This class is being extended to inherit JSF managed bean characteristics.

Overrides- init method of AbstractPageBean. This is done for all the initialization work

Since:

26/06/2010

Commenting Methods

Methods are the main action areas of your objects. Hence it is of vital importance to comment them properly. The important thing is that you should comment something only when it adds to the clarity of your code. You should not comment all of the factors described below for each and every method because not all factors are applicable to every method. You would however comment several of them for each method that you write.

In the following table lists the specific areas to be addressed while commenting methods. There are two tables; the first one describes the *External Commenting* rules. These should be included before starting the method i.e. on the top of the method.

In addition to the member function documentation, you also need to include comments within your member functions to describe your work. The goal is to make your member function easier to understand, to maintain, and to enhance. These come under *Internal Commenting* rules. For Internal commenting use single-line comments.

Internal Commenting Rules

Type	C	Rationale – Why comment	How to comment	JD	Jd Kw	Example[1]
Purpose	Y	• Describing the purpose make it easier for others to determine if they can reuse your code. • It also makes it easier for others to put your code into context. •	The general purpose	Y	N/A	/** * Purpose *Saves a new ticket *---------- */
Return type	N	• You need to document what, if anything, a member function returns so that other programmers can use the return value/object appropriately.	• Use *@return* tag followed by the return type • Note that the *@return* tag should not be used in methods with void return type. Doing so will generate a warning statement by the java doc	Y	@return	/** * @return A reference of *WorkTicketTransaction */ public WorkTicketTransactions createNewTicket(String sub, String …) {

[1] All these examples are taken from the *IssueTrack* application.

Type	C	Rationale – Why comment	How to comment	JD	Jd Kw	Example[1]

Exception thrown	N	• You should document any and all exceptions that a member function throws so that other programmers know what their code will need to catch.	• Use *@exception* or *@throws* tag followed by the description of the exception	Y	@exception or @throws	/**
						*@exception MessagingException
						*/
						
						public int sendMail() {
						try {
						Properties props =
						---------- 	

Parameters	N	• Indicating what parameters, if any, must be passed to a method makes it clear how they will be used. • This information is needed so that other programmers know what information to pass to the method.	• Use *@param* tag followed by the parameters description	Y	@param	/** *@param *sub,desc,createdUser,finalStatus,de * ptId,appType,….. */
Date of last modified	Y	• Date of last modified is necessary to know for how long this method has been in stable stage.	• Use *@since* tag followed by the date	Y	@since	/** *@since 31/07/2010 */

Variables initialized	Y	• This describes what the variables that are initialized by this method.
		• This is required for the developers to take care of those variables while using this method.
		• You should also mention why these variables are initialized, if necessary.
			• Write a one line describing the variables initialized and why	Y	N/A	/**
						* Variables Initialized -
						* A reference variable of
						*Mailer data bean because it is
						*needed to send mail
						*/
						
						private void sendMail(int ticketNo,
						String mailSubject) {
						MailerDataBean mdb =
						(MailerDataBean)
						getBean("src$dataBean$MailerData
						Bean");
						mdb.setTicketNo(ticketNo);
						
						…….

Applicable preconditions and postconditions	N	• A precondition is a constraint under which a method will function properly, and a postcondition is a property or assertion that will be true after a member function is finished running. Specifying these will help a developer who intends to use it to have a	• Describe the pre conditions • Describe the post conditions	Y	N/A	/** * Precondition – *The ticket no value should be the *current latest value in the db * * Post Condition-

Type	C	Rationale – Why comment	How to comment	JD	Jd Kw	Example[1]
		clear picture.				* The new ticket number *value should be one more than the *current one after the method is *executed */ public void saveWorkTicketTrans() {…..

Concurrency issues	N	• If the method is designed such that it will run in a separate thread then it should be properly documented.	• One liner describing why this method's actions are required to run in a separate thread.	Y	N/A	/**
						* This actions in this method is
						*to be run in a separate thread as
						*they are resource intensive
						*/
						
						public void run() {
						sendMail();
						}

Internal Commenting Rules

Internally, you should always document the following

Type	C	Rationale – Why comment	How to comment	JD	Jd Kw	Example

Control structures	N	• Commenting it doesn't require the reader to read all the code in a control structure to determine what it does, instead he just have to look at a one or two line comment immediately preceding it.	• Describe what each control structure does, such as comparison statements and loops.
			• Control structures includes the if-else, for, while, do-while etc.
				N	N/A	//The following code loops through the
						//list and extracts each WTA object
						
						for (Iterator<WorkTicketAttachments> it = l1.iterator(); it.hasNext();) {
						WorkTicketAttachments wt = it.next();
						
						}

Business Rules	Y	• One can always look at a piece of code and figure out what it does, but for code that isn't obvious one can rarely	• Describe clearly the logic behind the business rule	N	N/A	//Check that department id and work
						//classification type id is not null. If null
						//then return false

Type	C	Rationale – Why comment	How to comment	JD	Jd Kw	Example
		determine why it is done that way. Hence non obvious business rules should always be commented.				if (deptId == 0 || workClassType == 0) { return false;
Local Variables	N	• Non obvious local variables should be properly commented so that it clearly indicates its purpose.	• Use an end of line comment[2] to comment local variables	N	N/A	private boolean disableDeptDD; // This //variable controls the enabling-disabling //status of the department drop down box

Difficult or complex code	Y	• Documenting complex or non obvious code makes the logic clear to anyone reading it.
		• Its better than reading the code
			• Give a clear and concise description of the logic used	N	N/A	//Status-0 indicates that the ticket is still //Open and 1 indicates that the ticket is //closed
						
						if (statusFinal.equals("0")) {
						mdb.setStatus("Open");
						} else {
						mdb.setStatus("Close");
						}

Object instantiation	Y	• This helps to understand why a particular object reference was initialized here and in this way	• Give a clear and concise description of why the object is created and the way it is created.	N	N/A	private void sendMail(int ticketNo, String mailSubject) {
						
						//MailerDataBean object is used to send //mail. Since it is a JSF managed bean, //hence it is initialized using getBean //method.
						
						MailerDataBean mdb = (MailerDataBean) getBean("src$dataBean$MailerDataBean");

[2] *End of line comments* are single line comments that are written in the same line after the semicolon.

ANNEXURE-I

How to write Comments for Javadoc

Javadoc is a tool that parses the declarations and documentations in a set of Java source files and generates API documentation in HTML format (by default). This utility pulls out the important information concerning the classes, inner classes, interfaces, constructors, methods, and fields, and automatically generates good-looking documentations.

- Text intended for ***javadoc*** must be placed in doc-comments that begin with "/**" and end with "*/".
- Doc-comments for a class or method must be placed immediately before the class or method declaration.
- The first sentence for a doc-comment should be a one-sentence summary of the class or method being documented.

The following table lists down

Tag	Format	Description
@author	**`@author`** *name-text*	Creates an "Author" entry. A doc comment may contain multiple @author tags.
@deprecated	**`@deprecated`** *deprecated-text*	Adds a deprecated comment indicating that this API should not use the specified features of the class. Deprecated notes normally appear when a class has been enhanced with new and improved features, but older features are maintained for backwards compatibility.
@exception	**`@exception`** *class-name description*	The @exception tag is a synonym for @throws.
{@link}	**`{@link`** *name label***`}`**	This tag allows the programmer to insert an explicit hyperlink to another HTML document.
@param	**`@param`** *parameter-name description*	Adds a parameter to the "Parameters" section. The description may be continued on the next line.
@return	**`@return`** *description*	Adds a "Returns" section, which contains the description of the return value.
@see	**`@see`** *reference*	Adds a "See Also" heading with a link or text entry that points to reference.
@serial	**`@serial`** *field-description*	Used in the doc comment for a default serializable field.

Tag	Format	Description
		An optional *field-description* augments the doc comment for the field. The combined description must explain the meaning of the field and list the acceptable values.
@serialData	**`@serialData`** *data-description*	A *data-description* documents the sequences and types of data, specifically the optional data written by the `writeObject` method and all data written by the `Externalizable.writeExternal` method. The `@serialData` tag can be used in the doc comment for the `writeObject`, `readObject`, `writeExternal`, and `readExternal` methods.
@serialField	**`@serialField`** *field-name field-type field-description*	Documents an `ObjectStreamField` component of a `Serializable` class' `serialPersistentFields` member. One `@serialField` tag should be used for each `ObjectStreamField` component.
@since	**`@since`** *since-text*	Adds a "Since" heading with the specified since-text to the generated documentation. These notes are used for new versions of a class to indicate when a feature was first introduced. For example, the Java API documentation uses this to indicate features that were introduced in Java 1.0, Java 1.1 or Java 2.
@throws	**`@throws`** *class-name description*	The `@throws` and `@exception` tags are synonyms. Adds a "Throws" subheading to the generated documentation, with the *class-name* and *description* text. The *class-name* is the name of the exception that may be thrown by the method. If this class is not fully-specified, ***javadoc*** uses the search order to look up this class.
@version	**`@version`** *version-text*	Adds a "Version" subheading with the specified *version-text* to the generated docs when the -version option is used. The text has no special internal structure. A doc comment may contain at most one `@version` tag. Version normally refers to the version of the software (such as the JDK) that contains this class or member.