Coding Standards and Development Guidelines

Project: FileSearhUtility

|  |
| --- |
| **Document Purpose and Usage**  This document describes the leading practices and standards for configuring the FileSearchUtility and Java applications. It is intended to be a living document and should be periodically updated to reflect the latest changes in best practices. Use of this document outside the Program is at the risk of the reader as these standards might not satisfy what is required for other projects and technologies. |

|  |  |
| --- | --- |
| Prepared by: | Anil Adusumalli |
| Date last updated | 14/03/2020 |
| Document location : | Confluence: xxxx |

# 

Contents

[1. Introduction 3](#_Toc342370119)

[2. Data model extensions 3](#_Toc342370120)

[2.1. Data model best practice checklist 3](#_Toc342370121)

[2.2. Entity extensions 3](#_Toc342370122)

[2.3. ClaimCenter and ContactManager integration 4](#_Toc342370123)

[2.4. Typelist extensions 4](#_Toc342370124)

[3. Configuration standards for the PCF Configuration 6](#_Toc342370125)

[4. Gosu Coding Standards 7](#_Toc342370126)

[4.1. Naming Conventions for Rules 7](#_Toc342370127)

[4.2. Naming of variables 7](#_Toc342370128)

[4.3. Comments 7](#_Toc342370129)

[4.4. Catching exceptions 7](#_Toc342370130)

[4.5. Type coercion 7](#_Toc342370131)

[4.6. Rule condition 7](#_Toc342370132)

[4.7. Comments 7](#_Toc342370133)

[4.8. Test cases 8](#_Toc342370134)

[4.9. Disabling rules 8](#_Toc342370135)

[4.10. Activity Patterns 8](#_Toc342370136)

[5. Integration Standards 10](#_Toc342370137)

[5.1. Package Structure 10](#_Toc342370138)

[5.2. General Coding Standards 10](#_Toc342370139)

[5.3. General Coding Standards 12](#_Toc342370140)

[5.3.1. Class 12](#_Toc342370141)

[5.3.2. Fields 12](#_Toc342370142)

[5.3.3. Methods 12](#_Toc342370143)

[6. Coding Checklists 14](#_Toc342370144)

[6.1. Gosu/PCF 14](#_Toc342370145)

[6.2. Integration 16](#_Toc342370146)

[7. CONTROL SHEET 19](#_Toc342370147)

[Document Information 19](#_Toc342370148)

[Document Filename 19](#_Toc342370149)

[Document Location 19](#_Toc342370150)

[Document History 19](#_Toc342370151)

[Distribution 19](#_Toc342370152)

# Introduction

The purpose of this document is to describe the Java and FileSearchUtility coding standards and development guidelines. A common set of conding standards and development guidelines helps improve the quality of the coding done by allowing configurers to apply consistent patterns when configuring the application.

The coding standards and development guidelines in this document is based on the organization coding standards and development guidelines documentation.

# Data model extensions

## Data model best practice checklist

* Observe general entity naming conventions
* Add the prefix to new extensions to avoid name conflicts
* Use singular for fields names. Use plural for arrays.
* Add ID as a suffix to column names for foreign keys.
* Observe typelist naming conventions

## Entity extensions

* Use camel notation for the entity names.

Example: <column desc="City Kana" name="**D**I\_**C**ity**K**ana" nullok="true" size="30" type="varchar"/>

* To avoid naming conflicts with Guidewire, add the prefix DI\_ to beginning the extension name to show it as a custom development-based extension and not Out-of-the-box.

Example: **DI\_**CityKana

* Add the name of the developer, the requirement number (JIRA reference number), and the date in the comments for the change. Use the month name, not the number, for clarity.

Example: <!— Dave Veloper 23-Apr-2009: Created the field XXXXXXX for requirement JIRA GOCLAIMS-40 -->

* Add the same comment line while checking in entity extensions to the version control system.
* While creating an array, the word array should be mentioned in the variable name in addition to the Ext wording.

e.g.: <array arrayentity="LossNoticeType" desc="**Array of** Loss Notice Type" name="DI\_LossNoticeTypes"/>

* Use a singular word for single value fields, for example: Claim.DI\_CatSevType. Use a plural word array fields reference a list of objects, for example: Claim.DI\_Recoverable**s**
* If a field is foreign key. The Variable name should have the following format [Field Name]ID:

e.g: ExposureID

* The field name in the data model should not exceed 25 Characters.
* Add a small description to all variables or sections of variables explaining why you are adding the fields to an entity.

E.g: // Dave Veloper 04 Apr 09: JIRA GOCLAIMS-56 - Adding these three variables to track the start and end dates and status of the claim when a D&O policy is outside of policy term.

## Java and FileSearch Utility integration

* When any new field is added to ClaimCenter contact entity, corresponding field should be updated in the ContactManager side entity as well or if the field is not required in the ContactManager side we need to ignore the field in the contact-sync-config.xml file.
* Also AB Integration WSDL needs to be refreshed and committed.

# Configuration standards for the PCF Configuration

* Recommended Directory Structure for PCF Files: It is recommended that NEW related PCF files are placed in the same directory.

For E.g: let’s say you are creating a new Administration View that will hold reference data for Service Providers. During the course of your development, you end up creating 5 PCF files. These PCF files should all be placed in an appropriately named folder (i.e. use the PCF naming convention illustrated above) under the main Admin folder. For example: ../config/web/pcf/admin/DI\_ServiceProviderAdmin/

* Use the DI\_ prefix to name new PCF files, for example DI\_PaymentDetailsDV
* Add the name of the developer, the requirement number (JIRA reference number), and date in the comments for the change.

e.g.: <!— Dave Veloper 23 Apr 2009: Created the field for JIRA GOCLAIMS-78 -->

* Add a similar comment line while checking configuration to the version control system.
* Comment every widget that is being modified or created keeping the above mentioned format in mind.

e.g: <!-- Dave Veloper: Fixed the column issue for JIRA GOCLAIMS-1024 -->

* PCF file should have only display logic. Business logic should specified in Gosu code and rules.
* Do clutter the properties section with lengthy logic in PCF Files.

Example: OnChange=... , ValidationExpression=...

* All the business logic should be handled in separate supporting file. Any specific validation on the inputSet should be handled in the code block section
* Unless required do not set postOnChange attribute as true, while there is no onChange functionality required.
* When retiring or deleting any field in studio, make sure the field is not referenced in the total configuration.

# Java/Gosu Coding Standards

## Naming Conventions for Rules

For more information on Rules, please reference the ClaimCenter/ContactManager Studio Guide, ClaimCenter/ContactManager Rules Guide and Gosu Reference Guide.

To avoid problems when configuring and testing rules, it is recommended that each Guidewire Studio rule have the following naming convention:

8 character alphanumeric identifier + space + - + space + description (where at least the last 3 characters in the 8 character alphanumeric identifier are numbers that are unique for a particular rule set).

Eg: 

## Naming of variables

* Use Camel notation for the variables that are being declared

e.g: var todayDate = gw.api.util.DateUtil.currentDate();

## Comments

* Comment the name of the developer, date and the reason for the Function that is being written

e.g: **//** 18 Dec 08 Dave Veloper: Created the rule for bug 2 from sprint 1 for checking the time to be in future when a person calls.

or

**/\*** 18 Dec 08 Dave Veloper: Created the rule for bug 2 from sprint 1 for checking the time to be in future when a person calls. **\*/**

## Catching exceptions

* Always use try, catch block syntax for catching exceptions
* Write User readable exception messages and make sure it has equivalent displaykey if shown on the user interface.

## Type coercion

* Follow the type coercion. Type coercion is assigning the right variable to the right data type.

e.g: var x:Boolean= Claim.CheckNumber as Boolean

## Rule condition

* While writing any new rule in Studio, put a code filter condition in "Condition" part of Rule tree instead of just a true value & business logic in "Action" part of Rule tree. This will help a lot in Performance Enhancement by executing rules conditionally & also will add modularity to rule code for ease of maintenance.

## Comments

* Comments, including owner name, shall be included for any new Gosu Class, Libraries or any piece of code. Add additional comments for each new Method/Function & wherever needed in code. Assume the audience has limited Gosu/ClaimCenter knowledge.

## Test cases

* Write the individual test cases for all the functionality of the developed code

## Disabling rules

* Whenever disabling an OOB rule:- add a short comment as to why is it disabled (so no one checks it back in, assuming it is disabled by mistake)
* Avoid using repeated shortcut operator rather group them in a list and do a contains check.
* Use of shortcut operators &&, || instead of AND, OR to optimise code performance

## Activity Patterns

* When using activity patterns.

Always use...

Activity.ActivityPattern.Code == "contact\_claimant"

The ActivityPattern code should be defined in a constant file.

DO NOT USE..

Activity.ActivityPattern == ActivityPattern( "actpattern:1003" /\* contact\_claimant \*/ )

or

ActivityPattern.finder.getActivityPatternByCode("task\_complete\_notification")

* If a check is required, if a particular ActivityPattern is allowed/disallowed, group the possible activity codes as contants array and do 'contains' check.

Eg:

Instead of the below approach

Activity.ActivityPattern.Code == "plan\_review" or

Activity.ActivityPattern.Code == "approve\_reserve\_change" or

Activity.ActivityPattern.Code == "approve\_payment"

Group the codes as string array in a constants file. Do an contains check, to see if the code is exist or not.

Constants.ActivityCode\_AllowedCodes.contains(Activity.ActivityPattern.Code.toLowerCase()) where ActivityCode\_AllowedCodes : String []= new String[]{"plan\_review","approve\_reserve\_change","approve\_payment"}

* If the type is a subtype of the original type, Gosu automatically downcasts after a typeis expression. Explicit casting to subtype is not required. Because Gosu confirms that the object has the more specific subtype, Gosu implicitly considers that variable’s type to be the subtype, at least within that block of code

E.g:   
var x : Object = "nice"  
var strlen = 0  
if( x typeis String ) {  
 strlen = x.length  
}

# Integration Standards

## Package Structure

The java plugins will follow the package structure descibed below:

* **di.cc.plugins** – All plugin classes invoked from ClaimCenter UI

eg. di.cc.plugins.DocumentContentSourcePlugin.java

* **di.cc.constants** – All constant values should be stored interfaces as public static variables.

eg. di.cc.constanst.CGUConstants.java

* **di.cc.exceptions** – All exceptions that may be thrown from plugin classess

eg. di.cc.exceptions.BasePluginException.java

* **di.cc.message** – All messaging implementations, MessageRequest, MessageTransport, MessageReply

eg. di.cc.message.ClaimFeedsTransportPlugin.java

* **di.cc.serviceadapter** – All classes that will be called by plugins to handle webservice calls

eg. di.cc.serviceadapter.GWSAdapter.java

* **di.cc.servicelocator** – All classes that will be called from adapters to encapsulate webservice lookup logic

eg. di.cc.servicelocator.GWSLocator.java

* **di.cc.transformer** – All classes called from adapters to do complex transformations and mapping logic

eg. di.cc.transformer.GWSTransformer.java

* **di.cc.utils** – All utility classes called that are encapsulate reusable logic

eg. di.cc.utils.DateUtil.java

## General Coding Standards

Review each recommendation and question below to ensure coding standards are followed for all interfaces developed.

* Each class should provide a description of the class purpose.
* Javadoc comments should be included with all methods. The exception to this rule is obvious get and set methods, which don’t require javadoc unless there is something special about the get or set method.

@param – for each method parameter

@return – for each method

@throws – for all exceptions thrown

* Check whether the starting brace start in the same line and the block statements should start in the next line with 2 space indent. End brace should be in separate line.

Eg.  
 If (I < 1) {  
 fStack = getList()  
 }

Note : Use open and close curly braces for if/else conditions even though has a single statement.

* 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 "{".
* Are brackets placed around blocks even when only one line of execution exists?
* Check Java Class members Sorting Order:

1. Static/class Fields

2. Static Initialises

3. Static Methods

4. Member/Instance Fields

5. Initialises

6. Constructors

7. Methods

• Use “IS” Prefix for getter methods which return Boolean value.

* Is the maximum line width less than 80 character.
* Two spaces should be used as the unit of indentation. Never place tab characters in source code files.
* When an expression will not fit on a single line, break it accordingly.
* Initialise all local variables. Make sure that any variable of Object type is initialised to null, unless otherwise required.
* All objects should be (including Strings) compared with “equals” and not “==”?
* Exceptions, catch block should take appropriate action.
* In a switch statement, are all cases should either break or return.
* Every variable or constant declaration should have a comment.
* Every function, class, and file should have an appropriate header comment.
* One declaration per line is recommended since it encourages commenting.
* Initialise local variables where they’re declared

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.

* Always list specific checked exceptions in the throws clause.
* Avoid changing the value of a method parameter in a method.
* Always Class Member variables should be private.
* Avoid System.out.println and System.err.

## General Coding Standards

Naming conventions are broken down for Classes, Fields, and Methods. Review each section and address each question and confirm each recommendation is adhered to in code.

### Class

* Value Object Class Names should end with Info E.g.: <X>Info
* Class names should be defined using intercap convention (E.g. ClaimFeedsPlugin) and begin with an upper case character?
* Class and interfaces should have named with descriptive nouns (If the class is a major participant in a well-known design pattern, name that class after that pattern. E.g. ConnectionFactory, RuleIterator)

### Fields

* Non-constant field names begin with a lower case letter and follow intercap and acronym conventions.
* E.g transformer : DMSTransformer = null
* Constants should be all uppercase with words separated by underscores ("\_")
* Final static field names should be defined in upper case

e.g. public static final int OU\_CATEGORY = 1;

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

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

* Counter fields should follow the convention of <x>Count?

E.g. reserveCount

* Default values and fields should follow the default<x> convention?

E.g. defaultPort, DEFAULT\_PRIORITY

* Boolean fields should follow the convention is<x> convention. E.g. isDebugEnabled

### Methods

* Method names should start with a lower case and follow intercap convention. Use verbs or verb phrases to name methods. E.g. getPolicyCoverage ()
* Validate the function parameters before using.
* Methods that return Boolean values should either have an is or has prefixed.

E.g. isEditable()

user.hasPermission(txnId)

* Method that does an update should have follow the convention update<x> or update<x>.

E.g. updateClaim(claimNo,value)

* Method that creates and add object should follow the convention addNew<x> convention.

E.g. addNewExposure(exposure)

* Methods that convert should follow the convention to<x>.

E.g. object.toString()

* Method that does the validation should follow the convention check<x>.

E.g. checkAddress()

# Coding Checklists

## Gosu/PCF

| S.No | Description | Yes/No | Remarks |
| --- | --- | --- | --- |
| **1** | Does the PCF file have only display logic? |  |  |
| **2** | Does the code in the PCF which is more than one line moved to classes / class /code block extensions? |  |  |
| **3** | Is the code well formatted? |  |  |
| **4** | Is the code commented? |  |  |
| **5** | Is the code clean? ( Remove unused variables / functions from the code. If we cannot remove for some reason atleast it should be commented.) |  |  |
| **6** | Confirm the code has not duplicate logic? |  |  |
| **7** | Does the code does proper null checks? |  |  |
| **8** | Does each levels of rules condition and have rules action, have a one line logger? |  |  |
| **9** | Have the newly created/modified function have been unit tested? |  |  |
| **10** | Have function created either in the valid entity if not in util? |  |  |
| **11** | Does the PCF working for all supported locale with proper labels without any mismatch? |  |  |
| **12** | Does the new extension added begins with the capital letter? |  |  |
| **13** | Does the new extension added has only allowable characters mentioned in coding standard document? (letters, numbers and underscores) |  |  |
| **14** | Does the new extension added has DI\_ as prefix? |  |  |
| **15** | Does the new extension added correctly represent whether it is a singular or plural? (for eg. Array should be plural) |  |  |
| **16** | Does the foreign key added ends with ID? (for eg. ClaimID) |  |  |
| **17** | Does new PCF added has DI\_ as a prefix? |  |  |
| **18** | Does the new PCF added in the correct directory structure? |  |  |
| **19** | Does the new rule name adhere to the coding standard? (8 character alphanumeric character + space + - + space + description) |  |  |
| **20** | Does the rule name begins with the initial characters of the rule set. (For eg, CV for claim validation.Pls refer coding std. document for valid prefixes) |  |  |
| **21** | Does the rule has only business condition in condition part and needed actions in actions part? ( For eg. The action part should not have entire code including condition part leaving the condition part blank / true) |  |  |
| **22** | Does the new class added has DI\_ as prefix? |  |  |
| **23** | Does the Try / Catch block used in Rule Actions, Classes and Class Extensions? |  |  |
| **24** | Does the entity been checked for NULL before it is used? (For eg. When a method is called using 2 arguments, both should be checked for NULL before calling that method) |  |  |
| **25** | Does the new function name prefixed with DI\_? |  |  |
| **26** | Does the variable name follow the naming convention? (For eg. strFName - Also refer coding standard document for valid prefixes) |  |  |
| **27** | Does the function marked as private if it is used only within that class? (The default access specifier is Public unless specified) |  |  |
| **28** | Have the developer used the break and continue statements to control processing within loops? |  |  |
| **29** | Are all class extension function logically related to the name and function of class extension? |  |  |
| **30** | Are transaction iteration used only once? (Multiple transaction iteration causes performance issues) |  |  |
| **31** | Does the verify command in Studio runs without any compilation errors? |  |  |
| **32** | Does the display keys (labels) changed only in appropriate places? |  |  |
| **33** | Does variable name in the database within 25 characters? ( should not exceed 25 characters) |  |  |
| **34** | Does the shortcut operators &&, || used instead of AND and OR to optimise the code performance? |  |  |
| **35** | Does the retiring / deleting field in the studio been properly modified in all referenced PCF as well? |  |  |
| **36** | Does the code adhere to the configuration and rules guidelines? |  |  |
| **37** | Confirm all the Activity pattern check uses only the pattern names and not the activity pattern code. |  |  |
| **38** | If any new field has been added to CC contact entity, has the respective change made in CoC entity too? |  |  |

## Integration

| S.No | Description | Yes/No | Remarks |
| --- | --- | --- | --- |
| **1** | Does the code have unused local variables/parameters or unused private methods or unused imports ? If yes, then it should be commented at least. |  |  |
| **2** | Does the code have javacdoc comments at class and method level? (Only exception to this are getters/setters) |  |  |
| **3** | Check whether the starting brace start in the same line and the block statements should start in the next line . End brace should be in a separate line.  Eg:  if(I < 1) {  c=2; }  Note : Braces must even if it is a single statement. |  |  |
| **4** | No space between method name and paranthesis '('. Open brace '{' starts in the same line as declaration. End brace '}' starts a separate line. |  |  |
| **5** | 'is' prefix needs to be used for methods returning Boolean value. |  |  |
| **6** | Two spaces should be used as the unit of indentation. Never place tab characters. |  |  |
| **7** | Initialise all local variables. Also, make sure that any Object type is initialised to null unless otherwise required. |  |  |
| **8** | Are all objects compared with 'equals' and not '==' |  |  |
| **9** | Avoid changing value of method parameter in a method |  |  |
| **10** | Class member variables should be private & accessed with get()/set() |  |  |
| **11** | Does the code have empty catch() blocks ? All exceptions should be logged. (Console output should be removed) |  |  |
| **12** | Avoid console output. (System.out.print) |  |  |
| **13** | If you need to iterate through any lists, make sure you iterate it only once. Redundant loops should be avoided. |  |  |
| **14** | Does the code have static variables? Avoid static variables in plugins if at all possible. ClaimCenter may call plugins from multiple process threads and in some cases this could be dangerous and unreliable. |  |  |
| **15** | Logging. Use gw.api.util.Logger.logXXXX(strMessage) class with appropriate category based on the plugin and:  a. use logInfo() for general information for tracking  b. use logError() for error conditions  c. Log error in the class that finally catches exceptions  d. use logDebug() for debug logging. Enclose the logDebug() in a Logger.isDebugEnabled() if statement. |  |  |
| **16** | Message Plugin :  The send(msg,payload) method should end with either :  a. msg.reportAck() if success  b. msg.reportError() and msg.setErrorDescription() if failure.  (The error description should not exceed 254 characters) |  |  |
| **17** | Use StringUtils API (apache) for string related operations as it is null safe. Usually used for checking blank. |  |  |

# CONTROL SHEET

Document Information

|  |  |
| --- | --- |
| Document Filename | Coding Standards and Development Guidelines |
| Document Location | Confluence: |

Document History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version | Change Description | Amended by | Owner Approved | Date |
| 1.0 | Initial Document |  |  | 15/03/2020 |

Distribution

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Review/Approve | Title | Date |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |