

FuegoBPM Studio Standards and Best Practices

Version 1.2

FUEGOBPM

FuegoBPM Studio Standards and Best Practices

PART NO. Studio-Best-Practices.doc





Date February 3, 2006

This document is subject to change without notice. This document and the software described in this document contains proprietary trade secrets and confidential information of Fuego, Inc. and is also protected by U.S. and other copyright laws and applicable international treaties. Use of this document and the software is subject to the license agreement between you and Fuego, Inc. If no such license agreement exists, you may not use this document and software in any manner whatsoever. Unauthorized use of the document or software, or any portion of it, will result in civil liability and/or criminal penalties.

Fuego, Component Manager, Process Designer, Work Portal, Orchestration Engine, Execution Console, Process Analyzer, FuegoBPM Studio and FuegoBPM Designer are trademarks or registered trademarks of Fuego, Inc.

All other trademarks, trade names, and service marks are owned by their respective companies.

Table of Contents

1 INTRODUCTION.....	13
1.1 Purpose	13
1.2 About this guide.....	13
Best Practices	13
 Errors	14
 Testing and Debugging Tips	14
 Performance Tips.....	14
 Critical notes.....	14
2 PROJECT SETUP AND ORGANIZATION.....	15
2.1 Define Studio Preferences	15
2.1.1 Save Your Work Automatically.....	15
2.1.2 Keep Generated Java Source.....	15
2.2 Decide What Graphical Notation Will be Used in the Process Models.....	16
2.2.1 Classic Notation	16
2.2.2 UML Notation.....	16
2.2.3 Business Analyst Notation	16
2.2.4 BPM Notation (BPMN)	17
2.3 The Project Name	17
2.3.1 Use <i>Business Area Verb</i> Syntax for Project Name	17
2.3.2 Extract Project Name Acronym	17
2.4 The Project Organization	18
2.4.1 Use the <i>Project Name Acronym</i> when naming a new process	18
2.4.2 Use folders to contain process artifacts.....	18
3 FUEGO BUSINESS LANGUAGE.....	19
3.1 Change is the Rule.....	19
3.2 Project Team Meeting – Day 1 – Ground Rules.....	19
3.2.1 Code Walk-Throughs	19
3.2.2 FBL Syntax Preferences	19

3.2.3 Switch FBL Syntax Views As Needed.....	20
3.2.4 Logical Operator Syntax Consistency	20
3.3 Comments	21
3.3.1 When documenting a method, clarity is key	21
3.3.2 Adopt the Project Report as your formal project documentation.	21
3.3.3 Use the “Document” tab in FuegoBPM Studio	21
3.3.4 Develop a Documentation template for use in your project	21
3.3.5 Change Log Belongs in the FBL, not in the Documentation Tab	22
3.4 Length of Coded Business Rules.....	22
3.5 White Space	22
3.5.1 Use White Space to Make FBL Readable	22
3.5.2 Spaces Around Operators.....	22
3.6 Case Sensitivity.....	23
3.7 Adding Variables to FBL.....	23
3.8 Indenting	23
3.8.1 Indent Conditional and Loop Statements	23
3.8.2 Reusable FBL and Fuego Objects Simplify an Activity’s FBL	23
3.9 “end” Statements	24
3.9.1 Add a comment to lengthy <i>end</i> statements.....	24
3.9.2 Do not add a comment to short <i>if-end</i> statements.....	24
3.10 “case” (or “switch” in Java) Statement vs. “if / elseif”	24
3.11 Use Parentheses.....	25
3.12 One FBL Statement per Line.....	25
3.13 Lengthy Statements	25
3.14 Testing for Equivalence.....	26
3.14.1 Use the “=” Operator for Equivalence AND Assignment.....	26
3.14.2 Eliminate Case Differences When Comparing Strings.....	26
3.14.3 Comparing Real Values	27
3.15 Division by Zero Still Doesn’t Work	27
3.16 Converting Variable Values.....	27
3.16.1 Integer Division Truncates	27
3.16.2 Casting Numeric Types to Integers Truncates the Value	28
3.17 Arrays.....	28
3.17.1 Index Out of Bounds Problems.....	28
3.17.2 Reinitializing an Array	29
3.17.3 Adding New Entry to an Array.....	29
3.17.3 Removing entry in an Array.....	29

3.17.4 Large Instance Variable Arrays Degrade Engine Performance	29
3.17.5 Use “Separated” Instance variables in Excess of 10K	30
3.18 Split-N Copied Instance Variables	30
3.19 Release Resources	31
3.20 Hard Coded Values.....	31
3.20.1 Enumerations	31
3.20.2 Fuego Server Property Component.....	31
3.20.2 An XML Property File	32
3.20.3 Business Parameters	32
3.20.4 Database Table and Presentation	32
3.21 Log Messages	32
3.21.1 Include the Process and Activity Names in Log Messages	32
3.21.2 Log Message in Fuego Objects and Screenflows	33
3.21.3 Begin and End FBL with Log Messages.....	33
3.21.4 Concatenate Meaningful Variable Information.....	33
3.21.5 Set Severity in Most Log Messages to <i>DEBUG</i>	34
3.21.6 Add INFO messages to show SQL Access Time	34
3.22 Conditional Transitions.....	35
3.22.1 Conditional Transition Evaluation Order	35
3.22.2 With Many Conditional Transitions Use the Unconditional to Catch the Unexpected	35
3.23 Due Transitions	35
3.23.1 Use Due Transitions With Subprocess, Termination Wait and Notification Wait Activities	35
3.23.2 Setting Due Transitions	36
3.24 Reading Text Files into a Variable	36
3.25 Use the Predefined “result” String Variable.....	37
3.26 Keep FBL Tasks That Communicate With Other Processes Simple	37
3.27 Finding Fuego Object Artifacts Quickly	37
4 VARIABLE NAMING RECOMMENDATIONS	39
4.1.1 Capitalization	39
4.1.2 Variable Name Choices.....	39
4.1.3 Avoid One Character Variable Names	39
4.2 Local Variables.....	39
4.2.1 Never Name a Local Variable the Same Name as an Instance Variable	39
4.2.2 Local Variables Need no Special Treatment	40
4.3 Argument Variables	40
4.3.1 Use <i>arg</i> . Syntax in Front of All Argument Variable Names.....	40
4.4 Begin and End Variables	40

4.4.1 Append <i>arg</i> to Argument Names	40
4.4 Business Parameters, Business Variables and External Variables	41
4.4.1 Business Parameters	41
4.4.2 Business Variables	41
4.4.3 External Variables	41
5 FUEGO OBJECT RECOMMENDATIONS	43
5.1 Method Reuse	43
5.1.1 Limit Methods to a Single Task	43
5.1.2 Avoid Recreating Methods	43
5.2 Method Names	43
5.2.1 Concise Descriptive Names	43
5.2.2 Method Naming Conventions	43
5.3 Use a Fuego Object to Aggregate Input and Output Arguments	44
5.4 Constructor Method	44
5.5 <i>get</i> and <i>set</i> Methods	44
5.6 Method Length	45
5.7 Method Arguments	45
5.7.1 Use <i>arg</i> . Syntax in Front of Argument Variable Names	45
5.7.2 Small Number of IN / OUT Arguments	45
5.7.3 Make Method a Function If It has Only One Outgoing Value	45
5.8 Testing Equivalence of Two Fuego Object Instances	45
5.9 Referring to a Fuego Object from FBL	46
5.9.1 Use “this.” Syntax to Refer to the Current Fuego Object	46
5.9.2 Using Instance Variables to Refer to a Fuego Object	47
5.10.3 Using Local Variables to Refer to a Fuego Object	47
5.10.4 Using Fuego Objects by their Class Names Can be Convenient	47
5.10.5 Naming Fuego Objects	47
5.11 Fuego Object Instance Variable Size	47
5.12 Use Fuego Objects to Invoke Components	48
5.13 Fuego Object Testing	48
5.14 Database Considerations	48
5.15 Fuego Object Comments	49
5.15.1 Use the Documentation Tab	49
5.15.2 Put the Change Log in the Method	49

6 SCREENFLOWS AND PRESENTATIONS.....	51
6.1 Always Use Screenflows	51
6.2 Standardize Screenflow Design.....	51
6.2.1 Screenflow Example.....	52
6.3 Screenflow Tips and Techniques.....	52
6.3.1 Use <i>result</i> variable to control transitions	52
6.3.2 Use <i>result</i> variable to Cancel a Screenflow	53
6.4 Fuego Presentations.....	53
6.4.1 Template Preferences.....	53
6.4.2 Consistent Template Preferences Across Project Team	54
6.5 Decide on Project-Wide Presentation Approach	54
6.5.1 Advantages and Disadvantages To Using Custom JSP's.....	54
6.5.2 Advantages and Disadvantages To Using Fuego Presentations.....	55
6.6 Organize Your Screenflows and Presentations	55
6.7 Use Cascade Style Sheet	55
6.7.1 Define CSS on each presentation.....	55
6.7.2 Define the CSS Filename	55
6.7.2 Prototyping With the Stylesheet in Studio.....	56
6.8 Populating Combo Boxes.....	56
6.8.1 Do Not Populate Combo boxes from a Database in the Presentation	56
6.8 Canceling Screenflows.....	56
7 PORTAL ADMINISTRATION	57
7.1 Always Use Work Portal Views	57
7.2 Define External Variables for Display in the Work Portal.....	57
7.3 Design Work Portal Views From Left to Right.....	57
7.4 Use Portal View <i>Folders</i>	58
7.5 Always review the “default” Presentations	58
7.6 Create Role-Based Presentations.....	59
7.7 Use Portal Console to Limit Toolbar Options	59
8 EXCEPTION HANDLING	61
8.1 Plan for Exception Handling from the Start.....	61

8.1.1 Decide on Centralized or De-centralized Exception Handling.....	61
8.2 Develop Common Exception Handling	62
8.2.1 Create An Exception Fuego Object for the Common Catalog.....	62
8.2.2 Create two different Exception Presentations	62
8.2.3 A Generic Process Error Presentation May Not Always Do the Job	63
8.2.4 The Runtime Exception Presentation should Show Critical Information.....	63
8.2.3 Stack Specific Exception Handling Logic.....	63
8.2.4 Log Messages in Exception Handling	64
8.3 Exception Handling Process	64
8.4 Use <i>on exit</i> to Release Resources.....	65
8.5 Try to Fix Exceptions Locally	65
8.6 To Throw or Not to Throw Exceptions in Fuego Objects.....	65
8.7 Exception Email	65
8.8 Exceptions in End Activity Arguments.....	66
8.8.1 Don't Forget to Map the Exception in the End Activity.....	66
9 DATABASE ACCESS	67
9.1 Separate database objects from other Fuego Objects	67
9.2 Create database table objects as heir to Catalog tables.....	67
9.3 Override the accessDatabase default?	67
9.4 Decide What Type of SQL Syntax is Appropriate, Case by Case.....	68
9.4.1 Use Static SQL for Simple, Single Table Queries	68
9.4.2 Use Dynamic SQL for Complex or DBMS Specific Queries.....	68
9.4.3 Log Dynamic SQL Statements and Elapsed Time	68
9.4.4 If Logging SQL, Don't Parameterize Dynamic SQL Statements	69
9.4.5 Implement SQL Query as a Stored Procedure as a last resort	69
9.4.6 Do Not Write Cross-Database Queries.....	69
10 PROCESS DESIGN CONSIDERATIONS	71
10.1 Procedures or Sub-processes?.....	71
10.1.1 Procedures are Atomic	71
10.1.2 Concurrent Executions Property Irrelevant in Procedure	71
10.2 Split-N Performance vs. Concurrency.....	72
10.3 Other Split-N Factors to Consider	72
10.3.1 Turn Off Generating Events.....	72
10.3.2 Throw Exceptions With Caution	72
10.3.4 Establish a Split-N Upper Bound.....	73

10.4 Dynamic Process Invocation	73
10.4.1 The Sub-process Properties Dialog.....	73
10.4.2 A Dynamic Process Invocation Example	74
10.5 Synchronizing Activities	75
11 BUSINESS PARAMETERS	77
11.1 Define Meaningful Business Parameters.....	77
11.2 Use Business Parameter to distinguish DEV, UAT or PROD.....	78
12 PROJECT VERSION CONTROL.....	79
12.1 Checking Out a Project.....	79
12.1.1 Use the “Checkout files <i>Read-Only</i> ” Option	79
12.2 Checking in Project Changes	80
12.2.1 Check-In Changes When They Have Integrity	80
12.2.2 Always Checkout (“Update Local”) before Checking-In (“Commit”).....	80
12.2.3 Backup Your Project Before Check-In.....	80
12.2.4 Check-In At the Lowest Level in the Project	80
12.3 Request “Edit” Permission At the Lowest Level.....	81
13 THE COMMON CATALOG	83
13.1 How A Catalog Component Becomes a Common Catalog Component	83
13.2 The Common Catalog is visible to all.....	83
13.2.1 A Project Team should own their Common Catalog component	84
13.2.2 Be Careful What Components Are Moved to the Common Catalog	84
13.3 Standard Project and Common Catalog Directories.....	84
13.3.1 Do Not Use Default Directory Names.....	84

Revision History

Version 1.01	September 6, 2005	Removed section 12.2.1 from document as it duplicated but contradicted the standard recommended in section 13.3.1.
Version 1.2	February 3, 2006	The graphics were reformatted.
		Added an example in section 3.13.
		Removed section 3.23.2 because it duplicated the information in 3.23.3.

Section**1**

Introduction

1.1 Purpose

This guide provides an encompassing set of recommendations and best practices when using FuegoBPM Studio to develop and deploy processes into production:

- Defining and organizing a project: Organizing a project so that processes, procedures, screen flows and Catalog components are well organized and easy to find.
- Fuego Business Language (FBL) coding recommendations: Many of the recommendations are common sense ideas that might have occurred to you after using FuegoBPM a short while. Others ideas contained here help reduce programming errors, testing and debugging time and improve performance.

This guide is only as good as you make it. Use it as a base to develop your own standards and recommendations. Adherence to standards will only be achieved through strong project management. During the project kick-off, talk about the standards. At every project meeting, encourage their use. Have this document at hand during peer code walk through sessions. Following “post mortem” discussions, update this document to customize it to your environment and development approach.

1.2 About this guide

This guide provides recommendations based on these four categories explained below. Graphical icons are used to help identify all but best practices, as this is a best practices document.

Best Practices

When we teach or mentor others how to use FuegoBPM, one of the key areas of emphasis is to ensure everything is clear, understandable, easy to debug and easy to maintain.



Errors

When first using FuegoBPM, certain errors are commonly made. These standards may help you avoid these errors and build more stable production processes your first effort.



Testing and Debugging Tips

These tips reduce the likelihood of bugs showing up later and simplify the testing and debugging effort.



Performance Tips

Some of these observations might seem minor. Our experience has been that processes in development and QA stages of development often appear to perform perfectly. Once put into production, with many end users creating thousands of instances, or running batch processes, problems sometimes need to be resolved. These tips provide some suggestions to help ensure processes going into production will perform as expected.



Critical notes

Most of these are related to best practices that are serious enough to warrant identifying with this rather ominous icon. The note doesn't necessarily mean your process will die a gruesome death but it does mean bad things could happen.

Section

2

Project Setup and Organization

2.1 Define Studio Preferences

Before creating or importing any projects to a new installation of FuegoBPM Studio, you should set the Studio defaults according to best practices defined by your organization. The Studio preferences covered in this section can be found in Studio at “File...Preferences”.

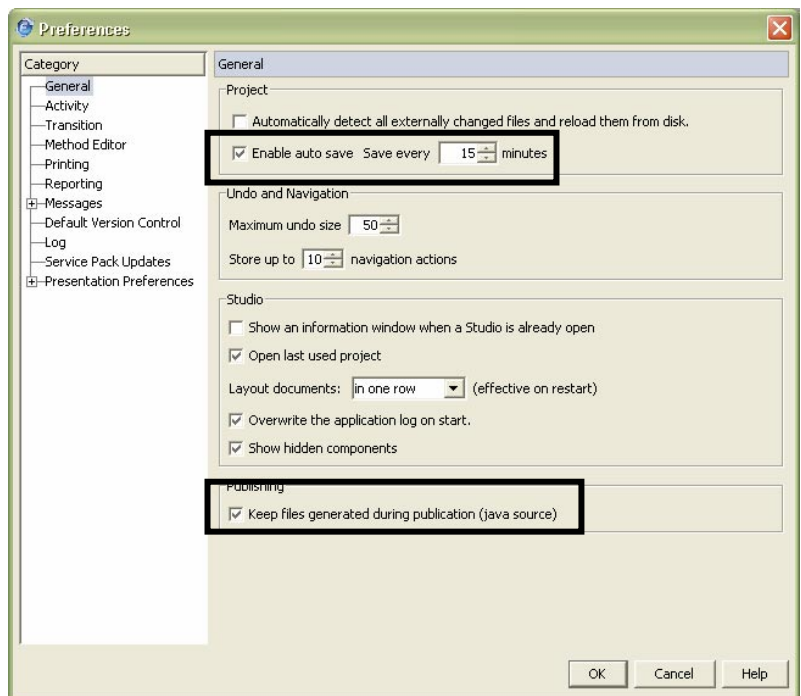
2.1.1 Save Your Work Automatically

We have all had experience forgetting to save our work and then some error occurs that results in all your work being lost. The answer is to let Studio save it periodically so you don't have to remember to do it.

2.1.2 Keep Generated Java Source

Check this option to have the Java source generated by Studio during publication so the source code can be referenced should a process exception occur. Even the best Fuego developers will periodically run into cases where a *NullPointerException*, for example, occurs because some process variable was not initialized.

When the process is published and deployed in Studio the generated **.java** source files can be found in sub-directories under



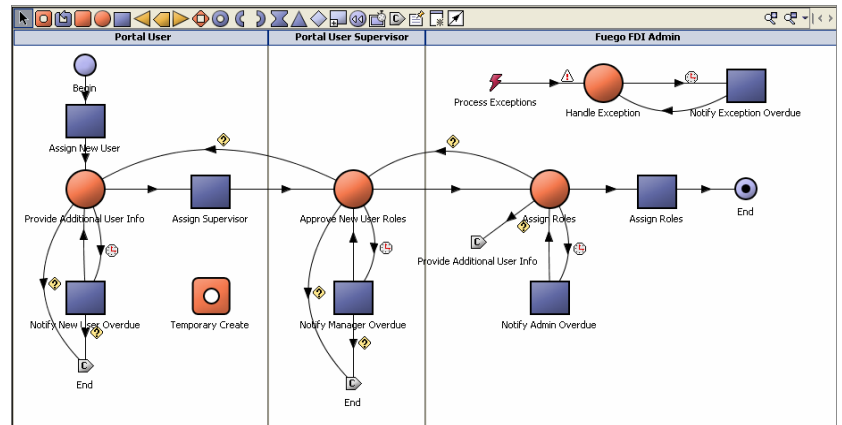
“C:\WINDOWS\Temp\your_project.fpr\build”, where “your_project” represents the filename of your Fuego project.

2.2 Decide What Graphical Notation Will be Used in the Process Models

Arguably the most heated debate between FuegoBPM developers is what notation should be used by the project team to communicate the process models to the business users. The graphical notation is available from the “View...Themes” menu.

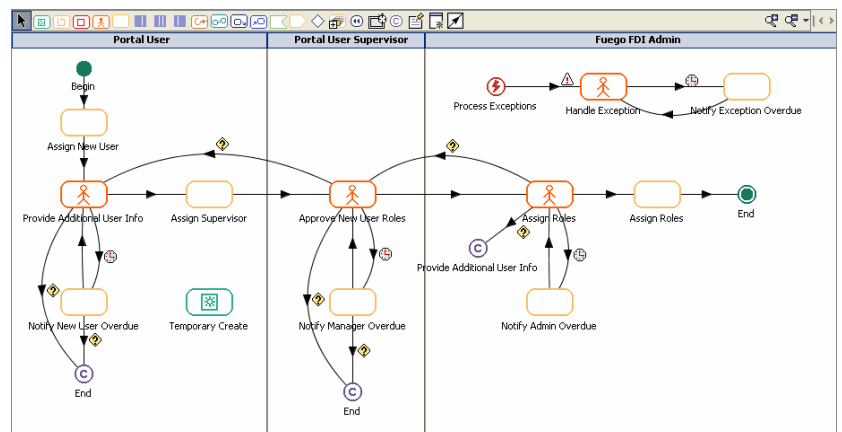
2.2.1 Classic Notation

Also referred to as the “Lucky Charms” notation, it has a long and distinguished tradition in BPM projects. From a new business user perspective however, it may not stack up to other graphical notations.



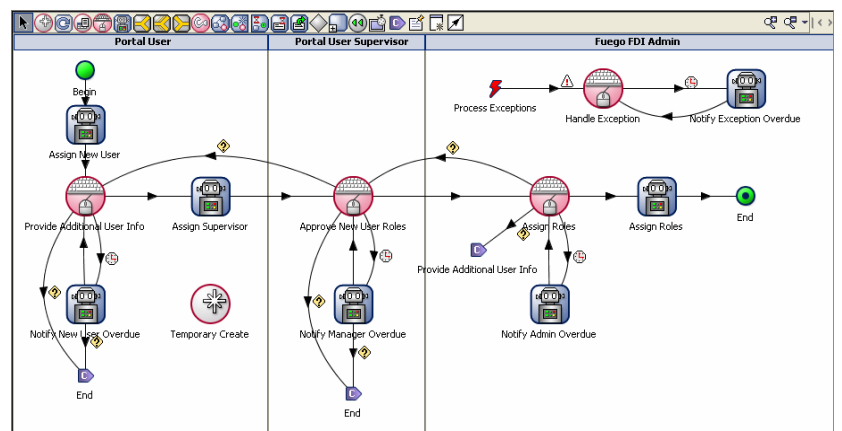
2.2.2 UML Notation

There really isn't UML notation for BPM but the little stick figure in the Interactive activities would lead one to that conclusion.



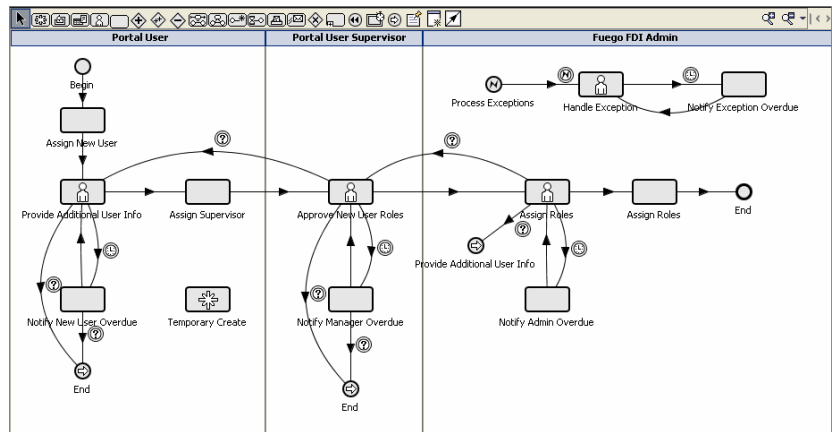
2.2.3 Business Analyst Notation

In recent new project startups this notation has been a favorite among business users.



2.2.4 BPM Notation (BPMN)

This notation comes in two flavors, uncolored (“BPMN”, as shown) and “ColorBPM” which is not very colorful at all. It is notations recommended by various BPM standards organizations and simple enough notation to become a standard way to represent processes in your organization.



2.3 The Project Name

The project will encapsulate one to many processes, procedures and screenflows representing the processing performed in a certain part of your business (or a “business area”). The project name should

2.2 The Project Name

The project will encapsulate one to many processes, procedures and screenflows representing the processing performed in a certain part of your business (or a “business area”). The project name should succinctly describe the overall processing to be done in the business area.

2.3.1 Use *Business_Area_Verb* Syntax for Project Name

The name selected should be three or more words in length, the last word ideally would be a verb (or noun) that summarizes the processing contained in the project. Some examples are:

- Warehouse_Inventory_Tracking
- EDI_Request_Matching
- Provider_Claims_Synchronization

2.3.2 Extract Project Name Acronym

Processes, procedures and screenflows to be developed in the project will be prefixed with a short acronym that represents the first letter in each word of the project name. From the previous project names examples you could derive the following acronyms to be used in the naming of processes, procedures and screenflows within the project:

- wit
- erm
- pcs

← The project name acronym should be 3-5 letters. Upper- and lower-case versions of the acronym will be used in naming standards.

2.4 The Project Organization

In large projects it can be difficult to easily locate process artifacts unless they are organized in a logical and consistent fashion. This section prescribes as standard for organizing the process artifacts.

2.4.1 Use the *Project Name Acronym* when naming a new process

Although processes are contained within a project they are sometimes accessed by other external applications and processes. By adding the project name acronym to the process name it makes the project in which it is defined easier to identify.

The process name should be a combination of “*ACR Process Name*” where

- “ACR” represents an upper-case *project name acronym* defined in *Section 2.3.2*
- “Process Name” consists of a noun-verb combination that describes the process.

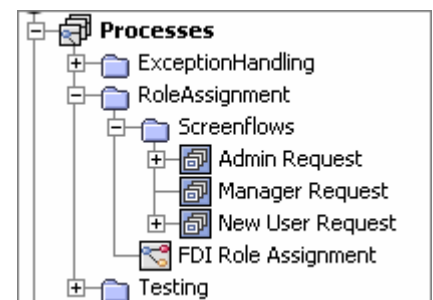
The “Description” section can be left blank as long as the process is documented in the Documentation tab (see *Section 3.3.1*).

The “Generate events for interactive activities” is the preferred default in most cases but there may be audit reasons for selecting the “Generate events for all activities” option. In a batch-style process with few interactive activities, “Do not generate events” made the best choice.

2.4.2 Use folders to contain process artifacts

Folders are available in the *Processes* section of the project as a vehicle to help organize process artifacts. The folder structure should not be overly complex. If the folder depth is too deep it becomes equally difficult to locate process artifacts.

Create one or more process folders (by business function, e.g. “Role Assignment”) in which processes and procedures will be located. Within the business process folder create a folder to contain the screenflows associated with that business function.



A “Testing” folder is sometimes used where procedures and methods have been created only for testing and experimentation purposes.

Section

3

Fuego Business Language

This section provides standards and recommendations for writing clear and concise Fuego scripting also referred to as the Fuego Business Language (FBL).

3.1 Change is the Rule

Write FBL that is easy to understand and maintain. Change is the rule. Always anticipate that this FBL will be modified at some point later.

3.2 Project Team Meeting – Day 1 – Ground Rules

The following are some ideas to keep in mind when you a project is initiated.

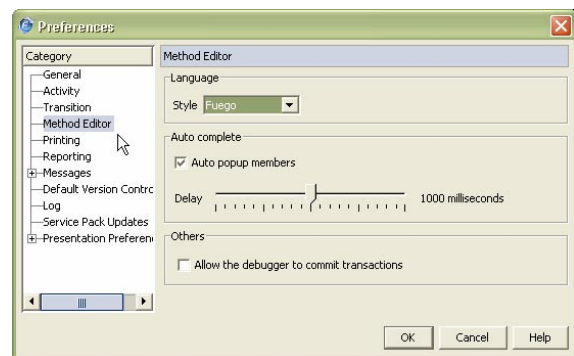
3.2.1 Code Walk-Throughs

Establish a policy for peer review code walk-through sessions. These should take place once the code is close to being completed.

Code walk-throughs are the quickest and simplest way to uncover FBL semantic and business rule errors early on. As one team member explains the logic, both are able to catch problems not thought of when originally reviewing the specification or coding the FBL. Problems caught early in development are much less costly to fix than those caught after a process is in production.

3.2.2 FBL Syntax Preferences

Decide the FBL preference setting (as shown below) the first day of the project. Everyone writing FBL on the project should use the same setting. This will help ensure the invocation of components have the same syntax throughout the project's FBL. The FBL will become



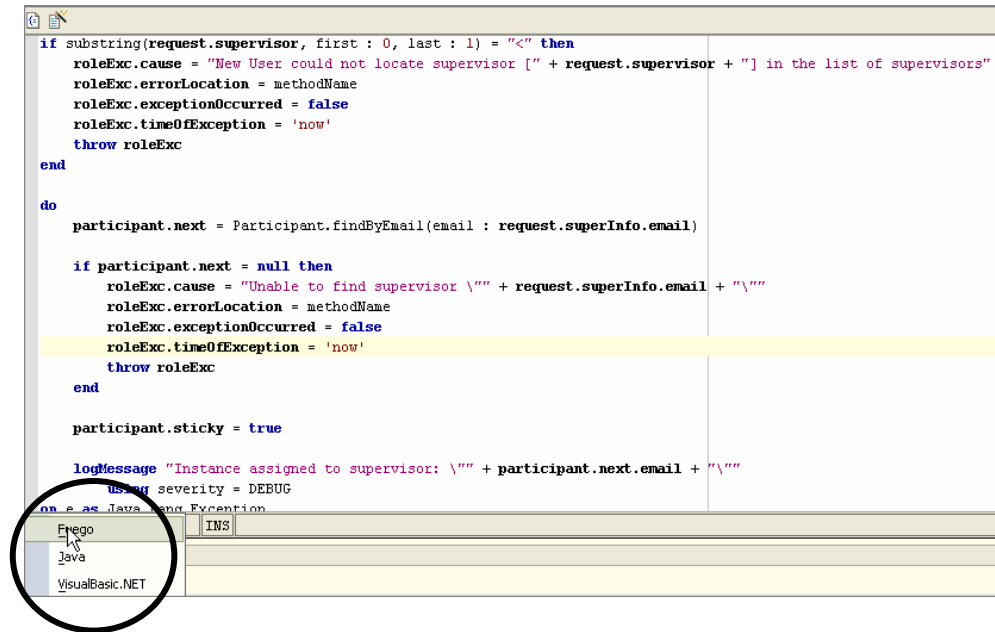
easier to maintain and code walk-through sessions will be facilitated.

3.2.3 Switch FBL Syntax Views As Needed

Across the project you should agree on what syntax to use. Recognize however that it can be switched on fly, within the FBL dialog.

In the lower right corner is the language preference represented in the FBL. It can be selected however and changed to “Java” or “VisualBasic.NET”.

In some cases, switching the syntax in select FBL may make development easier for certain developers on the project.



Switching between syntax here will change your preferences for all FBL that you review. Be aware also switching between the language syntax preferences may alter the formatting slightly, including spacing.

3.2.4 Logical Operator Syntax Consistency

Before starting the project, decide how logical operators will consistently be written in FBL by the entire project team. Decisions to make include:

Logical Operator	Example 1 (English syntax: <i>and</i> , <i>or</i> , <i>not</i> or <i>false</i>)	Example 2
<i>and</i> - &&	if (name == fName) and (type == 123)	if (name == fName) && (type == 123)
<i>or</i> -	if (name == fName) or (type == 123)	if (name == fName) (type == 123)
<i>false</i> - !	if (boolean == false)	if (! someBoolean)

Non-technical people may find the logical operator symbols &&, || and ! (example 2 above) obscure and prefer the standard shown in example 1. Others may not be used to using the verbose logical operator text *and*, *or* and *false* in example 1.

The point here is to ensure that everyone on the project uses the same syntax for all logical operators. The FBL will be easier to maintain and code walk-through sessions will be facilitated.

3.3 Comments

3.3.1 When documenting a method, clarity is key

Use comments to clarify difficult concepts. FBL can be somewhat self-documenting if variables are named correctly and the FBL is properly indented. Do not document the incredibly obvious.

3.3.2 Adopt the Project Report as your formal project documentation.

The FuegoBPM Studio “Project Report” is an HTML report created via “File...Project Report” and will include all of the content entered in the *Documentation* tab.

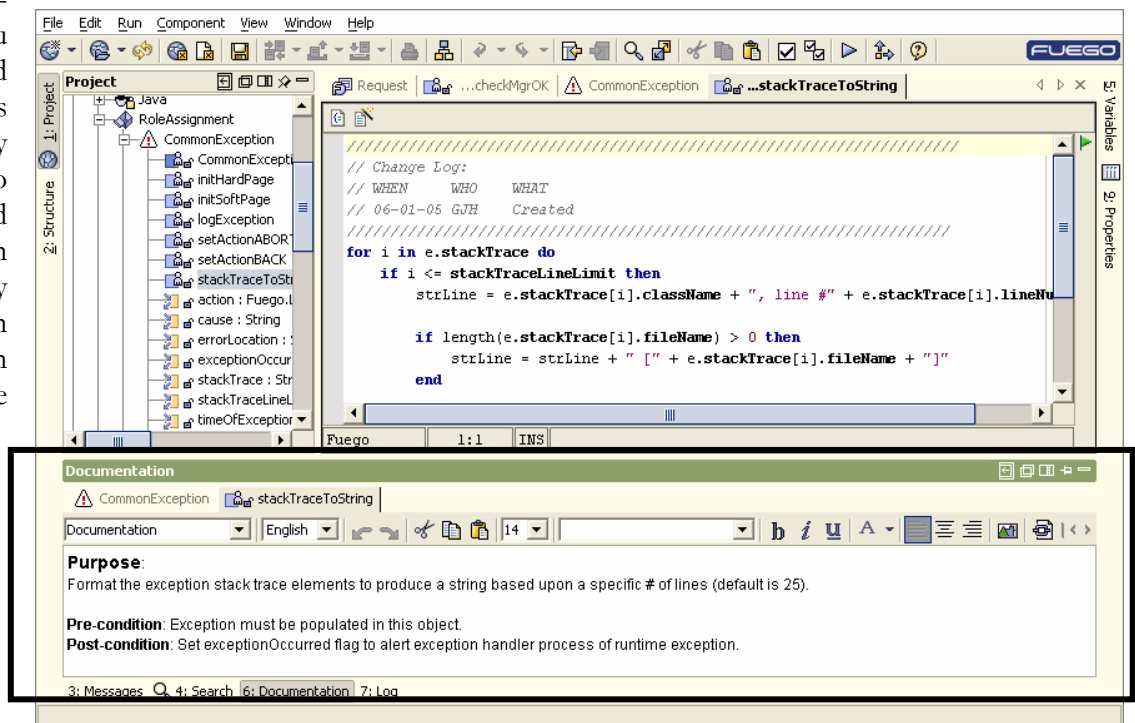
3.3.3 Use the “Document” tab in FuegoBPM Studio

The “Documentation” tab provides a rich-text editor for Project Report documentation. It is available for all process artifacts except methods. Method documentation needs to be done in the FBL for each method.

3.3.4 Develop a Documentation template for use in your project

The template should attempt to make documenting Fuego Objects and methods easy rather than overly time-consuming. As you develop the standard ask basic questions like is it really necessary to document input and output arguments in a method? If they have been named in an intuitive way then perhaps this can be omitted.

A common template that can be applied to both Fuego Objects and methods consists of three sections: “Purpose”, “Pre-



Conditions” and “Post-Conditions”.

3.3.5 Change Log Belongs in the FBL, not in the Documentation Tab.

After a process has been put into production, include a brief explanation of any changes made to the FBL, in the FBL and not in the *Documentation*. The comments can get rather mundane and is generally not desirable in the Project Report.

```

////////////////////////////////////
// Change Log:
// When      Who      What
// 06-01-05  JD       Created
////////////////////////////////////

```

This is useful to keep track of corrections and the people who modified the FBL. It is also to keep track of when the code was last modified.

3.4 Length of Coded Business Rules

Try to keep FBL business rules very short. They should normally be no longer than the number of lines shown in the FBL editor. FBL tasks longer than 80 lines are difficult to understand and maintain. It is easy to lose the context of what you are trying to understand if you have to scroll up or down long sections of FBL in the FBL editor.

(The 80 lines recommendation is somewhat arbitrary and really depends on individual experience. It is the spirit of the recommendation, not the law that is important.)

Complex FBL can be called from either reusable FBL in the same process or a Fuego Object method. Small methods promote reusability and are easier to maintain and understand.

3.5 White Space

3.5.1 Use White Space to Make FBL Readable

Use blank lines, space characters and tab characters to increase FBL readability. Blank lines should be used to separate different blocks of code and not to separate different lines of code. Too many blank lines will force you to scroll too often in the FBL editor and lose your concentration.

3.5.2 Spaces Around Operators

Place space characters on both sides of operators as shown below. This makes the operator stand out and makes the FBL easier to read.

```
orderAmt = orderAmt + freightCost
```

3.6 Case Sensitivity

FBL is case sensitive. Not using the proper upper and lowercase letters will result in syntax errors.

3.7 Adding Variables to FBL

There are any number of ways to reference variables in the FBL without having to type them directly:

- Use the <CTL-space bar> key to view instance and local variables
- Use `this` or `arg` (where appropriate) dot notation (e.g. `arg.firstName`)
- Use Class Name dot notation (e.g. `Customer.firstName`)

3.8 Indenting

Indenting FBL improves its readability.

3.8.1 Indent Conditional and Loop Statements

Indent consistently using the Tab key. FBL code inside of an *if* or loop statement should be indented one level of indentation. This makes the body of the structure stand out and enhances readability. If there are several levels of indentation, each level should be indented by the same additional amount of space (as shown below).

Consistently indent using the tab key

```

if (item is not null) then
    complete = false
    reqRecord = item.reqRecord
    // find out the first request number
    if (firstReqMbr == 0) then
        firstReqMbr = dsetRequest.iREQUESTNO
    end
end
end

```

3.8.2 Reusable FBL and Fuego Objects Simplify an Activity's FBL

If for some reason, you believe you need more than three levels of indentation, consider moving the code to either a reusable FBL in the same process or a Fuego Object method. This will make the Activity's FBL easier to read, reducing the need to scroll to the right and left while in the FBL Editor.

3.9 “end” Statements

3.9.1 Add a comment to lengthy *end* statements

When there is a long block of FBL before the *end* statement, add a comment beside the *end* statement that indicates the beginning statement it relates to. It is easy to lose sight of where the block of code associated with the *end* began. This practice helps prevent semantic errors and improves the readability and the ease of maintenance.

```
// Check to see if the customer number is valid
if (CustomerNumber is not null) then
    . . . (many lines of FBL here)
end // end for "if (CustomerNumber is not null)"
```

If the original condition changes, be sure to update this comment.

3.9.2 Do not add a comment to short *if-end* statements

In the case of shorter *if-end* statements Studio is quite handy at helping the developer to identify what *if – end* statements are associated with each other. In the example below, by clicking immediately after the *end* statement Studio highlights the associated *then* line to show their relationship. (Conversely clicking on *then* would highlight the *end* line.)

```
if substring(request.supervisor, first : 0, last : 1) = "<" then
    roleExc.cause = "New User could not locate supervisor [" + request.supervisor
    roleExc.errorLocation = methodName
    roleExc.exceptionOccurred = false
    roleExc.timeOfException = 'now'
    throw roleExc
end
```

3.10 “case” (or “switch” in Java) Statement vs. “if / elseif”

Use a *case* (or *switch*) statement in lieu of an *if-elseif* statement when:

1. testing the value of one variable (e.g., *city* in the example below), and
2. there would be three or more *elseif* statements under an *if* statement.

This will improve readability and ease maintenance.

```
case toLowerCase(city)
    when "addison" then
        zipcode = 75001
    when "allen" then
        zipcode = 75002
    when "bedford" then
```



```

        zipcode = 75037
    when "plano" then
        zipcode = 75023
    else
        zipcode = 75051
    end
end

```

3.11 Use Parentheses

Always use parentheses in both simple and complex expressions. This makes the expression easier to read, improves debugging efforts, and eases maintainability. Parentheses explicitly and clearly define the boundaries of different parts of a FBL statement.

```

orderAmt = ((price * quantity) + (taxRate * price * quantity))

// Shipping is free for large credit orders
if ((paymentType == "Credit") && (orderAmt > largeOrderAmount)) then

```

3.12 One FBL Statement per Line

Place only one statement per line in FBL. This improves readability.

```

// Do not do this (two statements on one line)
orderAmt = (price * quantity); orderAmt = orderAmt + (taxRate * orderAmt)

// Instead do this (each statement is on its own line)
orderAmt = (price * quantity)
orderAmt = orderAmt + (taxRate * orderAmt)

```

3.13 Lengthy Statements

A lengthy statement should be spread over several lines to improve readability. If a statement is split, choose breakpoints that make sense. As shown below, keep the expression operators (“&&” in this example) at the end of the line that you are splitting.

```

if ((paymentType == "Credit") && (orderAmt > LARGE_ORDER) &&
    localDelivery) then

```

Indent all subsequent lines until the end of the statement.

3.14 Testing for Equivalence

3.14.1 Use the “=” Operator for Equivalence AND Assignment

This rule really only applies to the Fuego or .NET FBL syntax. If you are using Java FBL syntax then you must distinguish between the two. What this rule is saying is that for Fuego FBL syntax you can use the “=” operator for both equivalency comparisons and as an assignment operator.

In the Java syntax example below, “==” (equivalency) and “=” (assignment) operators must be used .

```
if (participant.next == null) {
    roleExc = CommonException();
    roleExc.cause = "Unable to assign participant \""
        + newUser.email + "\"";
    roleExc.errorLocation = methodName;
    throw roleExc;
}
```

Toggle to Fuego syntax and the same example would be shown differently.

```
if participant.next = null then
    roleExc = CommonException()
    roleExc.cause = "Unable to assign participant \""
        + newUser.email + "\"";
    roleExc.errorLocation = methodName
    throw roleExc
end
```

3.14.2 Eliminate Case Differences When Comparing Strings

Whenever case insensitive string comparisons are done use the `toLowerCase` or `toUpperCase` methods to ensure the string comparison does not fail due to case differences alone.

```
case toLowerCase(city)
    when "addison" then
        zipcode = 75001
    when "allen" then
        zipcode = 75002
    when "bedford" then
        zipcode = 75037
    when "plano" then
        zipcode = 75023
    else
        zipcode = 75051
end
```

3.14.3 Comparing Real Values



Do not test Real type variables for exact equality or inequality. Rather, test the absolute value of the difference between two Real type variables and then check to see if the result is less than a specified small error acceptance value. For example:

```
result = abs(real1 - real2)

if (result < .0000002) then
    // do something here since the two variables are roughly equivalent
end
```

3.15 Division by Zero Still Doesn't Work



When performing division by a variable whose value could be zero, explicitly test this variable for equivalence to zero and handle it appropriately rather than allowing the division by zero to occur.

3.16 Converting Variable Values

When converting variables to different types, keep the following points in mind.

3.16.1 Integer Division Truncates

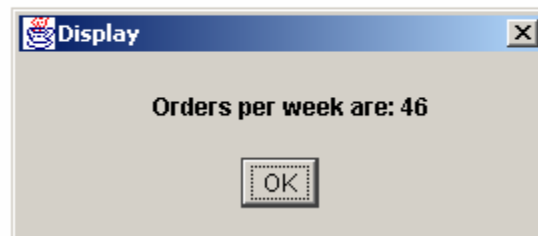


Assuming that integer division rounds (rather than truncates) can lead to incorrect results. Use a decimal variable to return the result of integer division.

In the example below, the variable *ordersPerWeek* is an Integer

```
ordersPerWeek = totalOrders / 52

display "Orders per week are: " + ordersPerWeek
```



If *ordersPerWeek* was a Decimal variable the answer would have instead been the correct 46.8.

3.16.2 Casting Numeric Types to Integers Truncates the Value



Casting a Real or Decimal variable to an Integer truncates the value and can lead to incorrect results.

```
orderTotal = 1234.56

display Int(orderTotal)
```



3.17 Arrays

3.17.1 Index Out of Bounds Problems



Before referring to an element in an array, test to ensure the array contains the specified number of elements. Referring to an element outside the array will result in a runtime “Index out of bounds” exception. The FBL logic below will detect this exception before it occurs.

```
// Check to see if the item referred to in the index "someIndex" is
// in the bounds of the array
if (someIndex_L < 0) || (someIndex_L >= someArray.length) then
    display "out of bounds"
else
    display someArray[someIndex_L]
end
```

This array index exception usually takes place inside loops. Always be careful to only loop through the number of elements contained in the array.

In the sections to follow assume the following array definitions:

```
supervisors    String[]
adminRoles     bool[String]
```

3.17.2 Reinitializing an Array

The preferred way to use arrays is via their encapsulated methods. Both statements below will work, however, on all types of arrays.

```
supervisors.clear()
adminRoles=[]
```

3.17.3 Adding New Entry to an Array

Use the “extend” method to add to an array. Any of the following statements will work.

```
supervisors.extend("new entry")
extend supervisors
    using arg1 = "new entry"
```

The “extend” method is not available for associative arrays. Use explicit array syntax to add an associative array entry:

```
adminRoles ["Administrator"]=true
```

3.17.3 Removing entry in an Array

Use the “delete” method to remove an array in any type of array. All of the statements below can be used to remove the 2nd entry in an array, for example.

```
supervisors.delete(1)
delete supervisors [1]
delete adminRoles ["key"]
```

3.17.4 Large Instance Variable Arrays Degrade Engine Performance

Storing many items in an instance variable array can degrade engine performance and unnecessarily consume large quantities of memory. For example, customers can have many orders. If a process needs to access all the orders for a customer, do not keep the orders as an instance variable. If a customer has many orders, too much memory will be consumed. A better approach is to store the list of orders for a specific customer in a relational database and read them when they are needed.

Large arrays degrade performance because a larger engine memory cache is needed to store instance information. Also, when the instance is serialized to be stored in the engines database on disk, the process of serializing it and writing it to the BLOB in the database will take more time as well.

Keep this in mind when deciding what to make an instance variable.

3.17.5 Use “Separated” Instance variables in Excess of 10K

Separated variables are stored in a special binary (BLOB) column in the FuegoBPM server database to optimize I/O access. If large documents must be attached to an instance consider creating a “separated” instance variable for the documents.

Separated variables are available only as instance variables and not as Fuego Object data members. This consideration may have an impact on the process design as it may mean the “separated” instance variable may have to be added to argument mapping, wherever the object it is associated with is mapped.



Be aware that separated variables cannot be copied via the clone method nor will the Split-N copies have separated instance variable copies.

Instance	
admOverdueInterval	Interval
email	Mail
exceptionHolder	Any
fdiNewUserEmail	String(255) [External]
fdiNewUserId	String(20) [External]
managerOverdues	Int
newUser	NewUserInfo
newUserOverdues	Int
request	Request
roleExc	CommonException
superInfo	SupervisorInfo
superOverdueInterval	Interval
userOverdueInterval	Interval
amazinglyLargeObject	Binary
Name: amazinglyLargeObject Type: Binary Description: Category: Normal	
Arguments : Normal, Separated , External	
Predefined : Local :	

3.18 Split-N Copied Instance Variables

As copies of an instance are created in Split-N logic, all of the instance variable information is also copied. If many copies are created using the “clone()” method, immediately set the unnecessary cloned instance variable(s) to null as shown below:

Category	
Activity Id	
Advanced	
Image	

Advanced	
Generate copies The activity will generate a copy for each transition. <input checked="" type="checkbox"/> Generate copies	
Generate Events If enabled, the Server will generate events for this activity. <input checked="" type="radio"/> Default <input type="radio"/> Generate Events <input type="radio"/> Do Not Generate Events	

OK Cancel Help

```
copy.someLargeInstanceVariable = null
```

This frees the memory that otherwise would have been wasted in the copied instance. The parent instance remains unchanged. This improves performance by reducing the memory and persistent database storage required by the copies.

Optionally you can adjust the Split-N activity property to generate a separate copy of the process instance if you don't need to distinguish the results within the Split-N circuit. The “Advanced” dialog is shown below.

3.19 Release Resources

Always release resources at the earliest possible moment when the resource is no longer needed. For example, after reading a text file and obtaining the information needed, use the “close()” method to free the resource. This also improves code clarity.

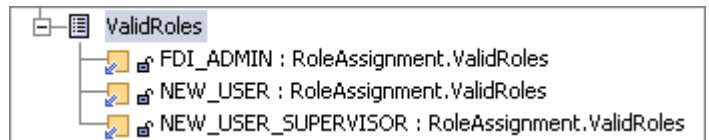
3.20 Hard Coded Values

Hard coding text or numbers in FBL is a common source of problems. This reduces maintainability of processes and Fuego Objects as these hard coded values change. Techniques for externalizing these static values are to use the techniques below, based on how static the values are.

- Enumerations (static and never change)
- Fuego server properties component (very static; seldom change)
- XML (or Windows INI) property file (static; may be changed on occasion)
- Business Parameters (static and changed on Web Console)
- Database table and Fuego presentation to update the values (static but changes with annoying regularity).

3.20.1 Enumerations

If the values never change then one option is to describe them as an enumeration in the FuegoBPM Studio catalog. They can then be referenced directly within the FBL using the class name.



Keep in mind the enumeration may need to be cast to String or Integer depending on how they are defined. In the above example they are defined as String and so to compare them with a String attribute it must be cast as string.

```

if toUpperCase(request.role) == String(ValidRoles.FDI_ADMIN) then
    ...
end
  
```

3.20.2 Fuego Server Property Component

An approach for storing global values used across all processes tied to the directory service is to use the Fuego *Server* component's *storePropertyFor* and *retrievePropertyFor* methods.

```
// set the value for the global server variable "ftpFiles"
storePropertyFor Server using
    name = "ftpFiles",
    value = ftpFileLocation

// retrieve the value for the global server variable "ftpFiles"
retrievePropertyFor Server using
    name = "ftpFiles" returning ftpFileLocation
```

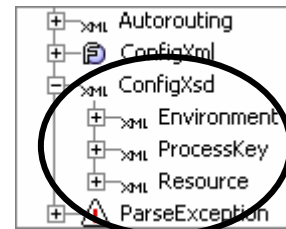


There are methods for storing objects as server properties but there can be performance issues depending on the frequency the object is retrieved from the FuegoBPM server.

3.20.2 An XML Property File

Consider using an XML properties file with the values change rarely and the person changing the information is comfortable changing an XML file.

Introspect the XML's XSD to access the data easily using the SAX parser.



3.20.3 Business Parameters

If the property is truly of a business nature then implementing it as a business parameter makes sense.

3.20.4 Database Table and Presentation

If the supposed static values change with annoying regularity and/or are changed by non-technical users then consider using a global activity in a Fuego process to provide a Fuego presentation to allow the values to be changed and stored in a table used by the process. It's a little more work than the previous options but it shifts the onus to the users.

3.21 Log Messages

3.21.1 Include the Process and Activity Names in Log Messages

Including the name of the process and the activity in log messages makes it much easier to search for the information in the log file you are looking for. Do not hard code either the process or activity names since these can change through the life of a process. Instead use FBL code like that shown below:

```
logMessage process.name + ": " + activity.name + ": " + instId + ": entry"
using severity = DEBUG
```


where *process.name* returns the current process name, *activity.name* returns the current activity name and *instId* returns an integer value for the Instance Id. As shown below, the *instId* is helpful to easily differentiate the different instances flowing through the process.

Using the *instId* in `logMessage` statements within a Split-N is especially useful. This keeps track of what each copy is doing.

3.21.2 Log Message in Fuego Objects and Screenflows

The process and activity reserved words used to prepend log messages previously mentioned obviously only works in processes. In Fuego Objects as well screenflows it is necessary to create an attribute that can be used by the `logMessage` method.

In a Fuego Object method define a “methodName” local variable and assign it a value in “Fuego Object.methodName” format:

```
methodName = "Customer.retrieveDetails()"
```

In a screenflow activity define a “methodName” local variable and assign it a value in “Screenflow Name, activityName” format:

```
methodName = "Customer Details Screenflow, Initialization()"
```

The “methodName” variable can then be prepended on the `logMessage` statements.

3.21.3 Begin and End FBL with Log Messages



Begin and end FBL with *logMessage* statements (like the one shown above) indicating whether it is the entry or exit. This technique is very useful inside activities that invoke components. In initial testing, this will help debug exceptions not caught that sometimes get thrown from components. If there is a begin in the log file, and there is no corresponding end log message in the log file an exception probably has occurred.

There is also a feature (“Trace Components”) of FuegoBPM Enterprise server that provides a **Begin-End** trace of all component methods.

3.21.4 Concatenate Meaningful Variable Information



Concatenating meaningful variable information in *logMessage* statements makes it easier to debug processes and monitor what is going on.

3.21.5 Set Severity in Most Log Messages to *DEBUG*



In almost all log messages, ensure that the severity is set to *DEBUG*. Log messages in batch process activities may be executed thousands of times each day. By setting the severity to *DEBUG* in FBL, you are able to easily turn off *DEBUG* level logging in the Execution Console and improve process performance once in production.

3.21.6 Add *INFO* messages to show SQL Access Time



In methods where complex SQL is performed it is useful to log the access time as a way of tracking how well the query works in different environments and external variables. In the example below a `logMessage` now only records the elapsed time ('now' – `beginTime`) but shows the actual SQL that ran.

```
beginTime='now'

sqlStmt = "SELECT a, b"
        +" FROM AUDITLOG with (nolock)"
        +" WHERE (file_primarykey = '" +provId +"')"
        +" AND    (a >= '" +strDateTime +"'"
        +" ORDER BY a DESC"

result = executeQuery(DynamicSQL, sentence : sqlStmt,
                      implname : "Some Database",
                      inParameters : [])

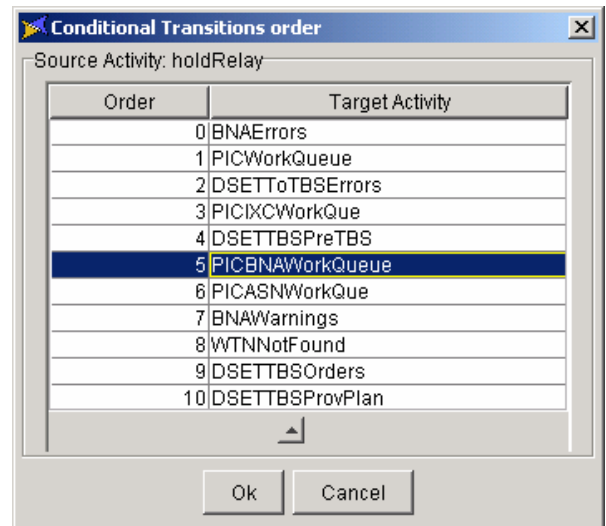
logMessage methodName +", ELAPSED time: " +('now'-beginTime)
        +" to execute query: \"" +sqlStmt +"\" using severity=INFO
```

3.22 Conditional Transitions

3.22.1 Conditional Transition Evaluation Order



If an activity has more than one outgoing conditional transition, the **first** conditional that evaluates *true* will be used (even though more than one would have evaluated true). As shown below, always check the order that these conditional transitions are evaluated whenever entering or changing logic behind any of these transitions.






3.22.2 With Many Conditional Transitions Use the Unconditional to Catch the Unexpected

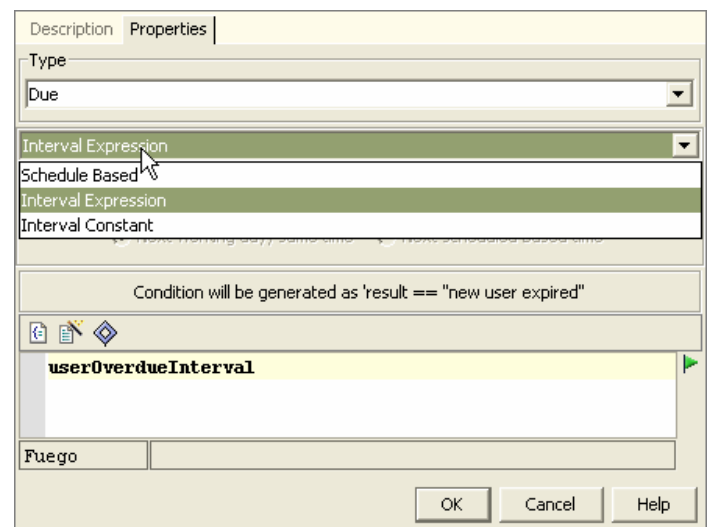
When you have many conditional transitions leaving a single activity and you believe each of the possible flows are handled by these conditional transitions, use the one unconditional transition to catch unexpected results.

3.23 Due Transitions

3.23.1 Use Due Transitions With Subprocess, Termination Wait and Notification Wait Activities



Add Due transitions leaving Subprocess , Termination Wait  and Notification Wait  activities. In these activities, the parent process instance is held there until either a child instance reaches the End activity or a notification occurs. If the child instance does not reach the End activity or the notification does not occur, instances in these activities remain stuck where they are. Since instances in these activities are not shown to end users in the Work Portal, no one will be able to track them. Adding a Due transition will keep the instance moving through the process.



3.23.2 Setting Due Transitions

The manner of expressing a due transition is generally via a time interval constant, e.g. '1h' (1 hour), '3d' (three days), etc. If it needs to be evaluated to a specific time of day (e.g. 1730 or 5:30 PM) then use the Schedule Based option. If it needs to be calculated based on external factors then use the Interval Expression option and calculate it prior to the due transition.

3.24 Reading Text Files into a Variable

If all the contents of a text file are first being read and placed into a single string variable, it is three to four times faster to first read the file as a binary and then cast it to a string. Reading a text file into a string is a common prelude to parsing an XML file.

For example, the FBL code shown below is a typical (and slow) way to read a text file and place its contents into a string variable:

```

////////////////////////////////////
//////// SLOW WAY TO READ A TEXT FILE INTO A STRING VARIABLE //////////
////////////////////////////////////

// set the variable used to open the XML file
handleForTextFile = TextFile()

// open the file
open handleForTextFile
    using name = LOCATION_OF_FILE,
        lineSeparator = "\n"

// build the string from the file just opened
xmlString = ""
for each line in handleForTextFile.lines do
    xmlString = xmlString + line
end
// close the file
close handleForTextFile

```

It is much quicker to first read the text file as a binary file and then cast the returned value into a string variable as shown below.

```

////////////////////////////////////
//////// FAST WAY TO READ A TEXT FILE INTO A STRING VARIABLE //////////
////////////////////////////////////
// Read an XML file
readToBinaryFrom BinaryFile using
    name = LOCATION_OF_FILE
    returning binaryFile

// build the string from the file just opened
xmlString = String(binaryFile, "ascii")

```






3.25 Use the Predefined “result” String Variable

Use the predefined *result* string variable in an activity’s FBL that sets values for conditional transitions. Although this technique cannot be used for all conditional transitions, keep it in mind since it will reduce the number and size of the process instance variables. This will help improve engine performance.

The result variable is very powerful when first defining processes. When used by the process designer to describe due transition descriptions it can later be used by the process developers with little or no modification.

Do not use the *result* predefined variable in conditional transition logic for transitions leaving a join activity. Many activities coming into the join activity might each have set the value of the result variable making its use in a conditional transition coming out of the join questionable.

3.26 Keep FBL Tasks That Communicate With Other Processes Simple

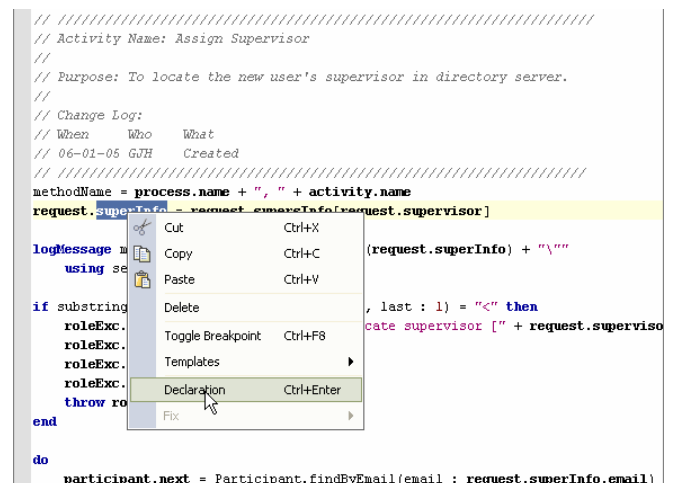
Subprocess , Process Creation , Termination Wait , Process Notification  and Notification Wait  FBL tasks should be used only to pass information to and receive information from argument variables. If any other FBL logic is required, add it to a previous or to the next activity.

3.27 Finding Fuego Object Artifacts Quickly

In the FBL there can be many references to objects and methods. Finding these process objects in the Project Catalog can be tedious. Fortunately there is a shortcut to navigate directly to them from the FBL. Simply highlight the object and press <control-enter>.

In this example, an attribute of the “request” object was selected so the properties panel associated with the attribute would be shown.

Alternatively the object itself, “request” could have been highlighted and viewed.



Section

4

Variable Naming Recommendations

FuegoBPM Studio will insist of certain convention such as when to capitalize the first letter of an object or variable name. They are mentioned here only as reminders of the standard in place across all Fuego projects.

4.1.1 Capitalization

Except for the first character, capitalize every new word in the variable (e.g., “orderNumber”).

4.1.2 Variable Name Choices

Variable names should be short, yet meaningful. The choice of a variable name should indicate to the casual observer the intent of its use. This helps FBL become self-documenting.

4.1.3 Avoid One Character Variable Names

One-character variable names should be avoided except for temporary “throwaway” local variables. Common names for local variables used as an integer counter in loops are i, j, k, m, and n.

4.2 Local Variables

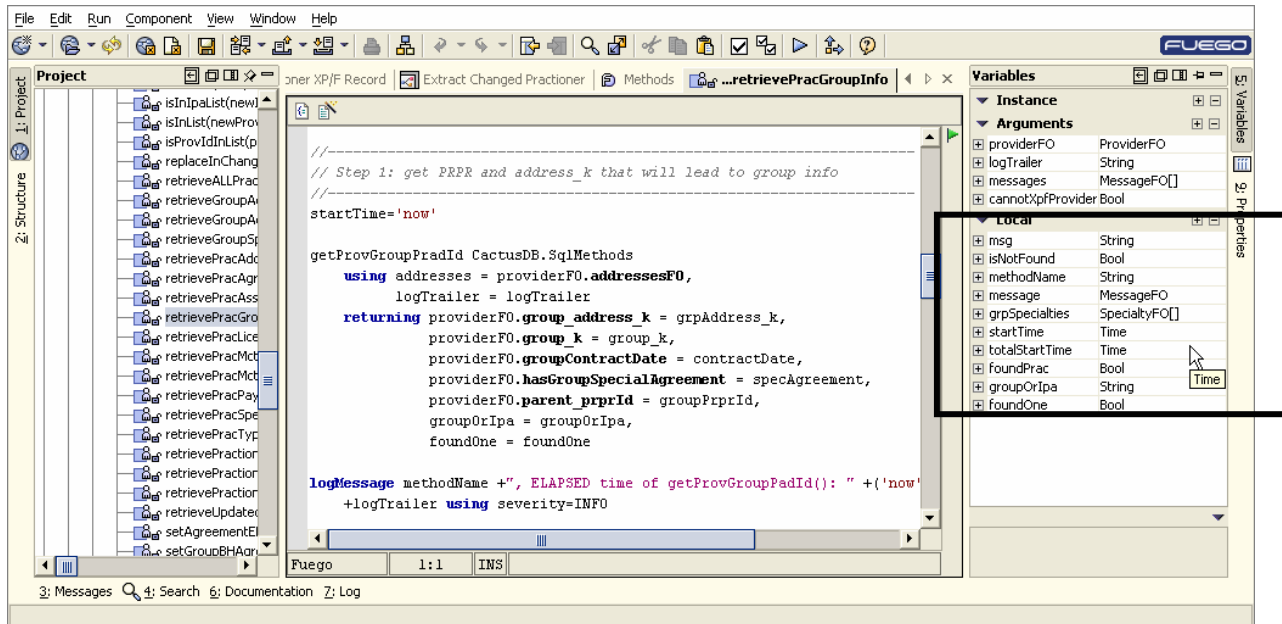
4.2.1 Never Name a Local Variable the Same Name as an Instance Variable



Never name a local variable the same name as an instance variable. A common question is “Why doesn't the instance variable I set in this activity reflect the change once it leaves the FBL?”. If an instance variable has the same name as a local variable in a FBL task, the variable set / used in the resulting FBL is the local variable and not the instance variable. This concept is called variable shadowing.

4.2.2 Local Variables Need no Special Treatment

The creation and display of local variables in earlier versions of Studio was handled much differently than the version in use today. Local variables should not be named any differently than other argument or instance variables.



It is the effortless way that argument, instance and local variables are shown in the Studio alongside the FBL that renders any special naming standards moot.

4.3 Argument Variables

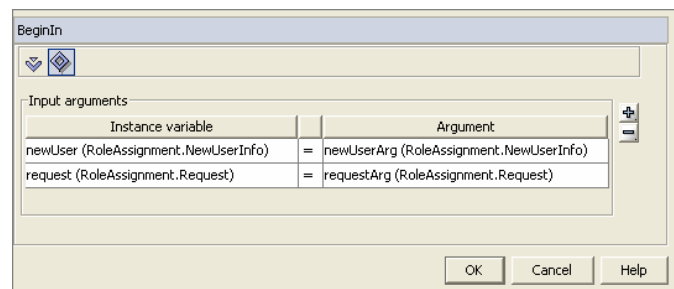
4.3.1 Use *arg.* Syntax in Front of All Argument Variable Names

Always begin argument variables with “*arg.*”. This makes it easy to tell argument variables from local or instance variables. For example, the correct way to refer to the *paymentType* argument variable in the FBL is *arg.paymentType*.

4.4 Begin and End Variables

4.4.1 Append *arg* to Argument Names

Although argument variables are limited in scope they are used extensively in argument mapping. Consequently a standard has evolved to append “Arg” to the names of *Begin* and *End* variable to help render the argument mapping displays more distinct.



4.4 Business Parameters, Business Variables and External Variables

Any process variable that is to be externalized should observe a rigid naming. This naming standard should be enforced or confusion will reign supreme.

4.4.1 Business Parameters

Business parameters are externalized to the FuegoBPM Enterprise Web Console where their values can be adjusted at the request of business users. You can imagine the confusion an administrator might encounter when asked to adjust a particular business parameter on an Enterprise server where 30-50 processes deployed, most using business parameters.

The Business Parameter should have a project abbreviation (e.g. “fdi”) pre-pended to the front of the variable name to differentiate it from the business parameters of other projects.

When published to FuegoBPM Enterprise server these business parameters appear in a list of variables for other projects and so some differentiation is helpful in knowing where they are used.

Variables	
▼ Business Parameter	
FDI_ID	String [default]
FDI_PWD	String [password]
FDI_UID	String [fuegoDir]
▼ Business	
▼ External	
fdiNewUserEmail	String<255>
fdiNewUserId	String<20>
▼ Instance	
+ admOverdueInterval	Interval
+ email	Mail
+ exceptionHolder	Any
+ fdiNewUserEmail	String<255> [External]
+ fdiNewUserId	String<20> [External]
+ managerOverdues	int
+ newUser	NewUserInfo
+ newUserOverdues	int
+ request	Request
+ roleExc	CommonException
+ superInfo	SupervisorInfo
+ superOverdueInterval	Interval
+ userOverdueInterval	Interval

4.4.2 Business Variables



Business variables have a special purpose related to exporting of process variables to the FuegoBPM Data Warehouse. They should never be used unless they are to be used for this purpose. When they are used, the Business variable names must be pre-pended with the project prefix to help distinguish them from the Business variables of other projects.

4.4.3 External Variables

External variables are generally used as a means of externalizing process variables to the Work Portal. As a matter of consistency then, the External variable names should be pre-pended with the project prefix (“fdi”) as shown in the example.

Don’t forget to give the external variable a friendly user names (e.g. “New User Email”).

External variable

Name

fdiNewUserEmail

Type

String

Size

255

Localize

OK

Cancel

Help

Localize label

Language	Message
English	New User Email

OK

Cancel

Help

Section

5

Fuego Object Recommendations

5.1 Method Reuse

5.1.1 Limit Methods to a Single Task

To promote reuse, each method should be limited to performing a single, well-defined task, and the name of the method should express that task effectively.

Large complex methods have little reuse and are difficult to maintain.

5.1.2 Avoid Recreating Methods

Avoid reinventing the wheel. Study the methods already created. As tempting as it might be to create your own, always attempt to find an existing Fuego Object method or component that already does what you want to do. In some cases, FBL can be split apart to increase its reuse. Remember that you can build complex methods from multiple small methods.

5.2 Method Names

5.2.1 Concise Descriptive Names

If you cannot choose a concise name that expresses a method's task, it is possible that your method is attempting to perform too many diverse tasks. It is usually best to break such a method into several smaller methods.

5.2.2 Method Naming Conventions

Method names start with a lowercase character and each following word should be capitalized. An exception to this is if you attempt to create *get{Attribute name}* (retrieve information) or *set{Attribute name}* (set information) methods for an attribute. In the java that FuegoBPM generates, methods with the

`get{Attribute name}` and `set{Attribute name}` are automatically created by FuegoBPM. Following `get` or `set`, use an underscore character (“_”) to avoid conflicting with the automatically generated FuegoBPM methods.

5.3 Use a Fuego Object to Aggregate Input and Output Arguments

Instead of defining processes with many input and / or output argument variables, define Fuego Objects to aggregate all the input and another to aggregate the output arguments. For example, a process may have many individual variables representing different information about a customer (name, number, street address, ...). Create a Fuego Object called Customer and add all the input arguments as attributes of the Fuego Object. The benefit is that you then only need one variable as an input argument. This simplifies the interface of the process.



Consideration must be made as to whether the process will be accessed via the FuegoBPM PAPI (Process API). If this is the case then a Fuego Object cannot be used as a process Begin argument.

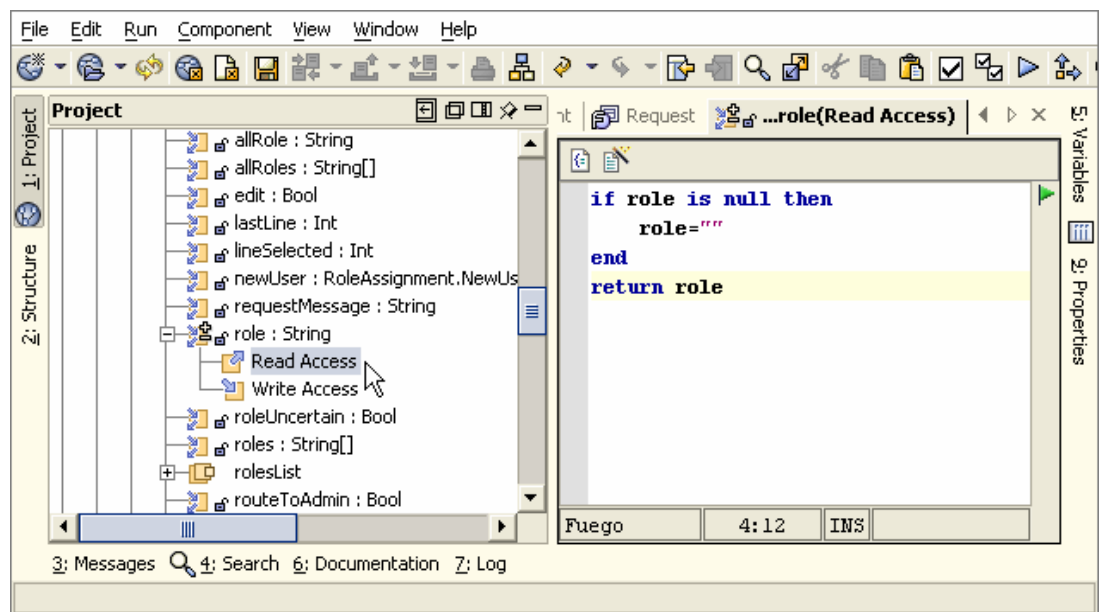
5.4 Constructor Method

The `<constructor>` method should only contain statements that are directly related to the one-time initialization of a Fuego Object. Never include any interactive (e.g., `display` or `input` statements).

5.5 `get` and `set` Methods

“set: and “get:” methods do not need to be encapsulated in Fuego Objects. Instead, refer directly to the Fuego Object attribute. (If this offends your Object Oriented sensibilities, know that Fuego will generate set and get methods for the attributes and uses them under the covers.)

If additional logic is required for when a Fuego Object attribute is referenced or assigned, then override the attribute’s read-write methods.



5.6 Method Length

A method should only do one task, and should normally be no longer than 80 lines. Small methods promote reusability and are easier to maintain and understand. Again, it is the spirit of the recommendation, not the law that is important. In some cases, a larger method may be warranted and not necessarily difficult to maintain.

5.7 Method Arguments

5.7.1 Use *arg.* Syntax in Front of Argument Variable Names

Some developers feel that prefixing incoming or outgoing argument variables with “*arg.*” leads to more understandable FBL. This makes it easy to differentiate incoming and outgoing argument variables from local or Fuego Object attribute variables (see section 4.3.1 *Use **arg.** Syntax in Front of All Argument Variable Names* for more information).

5.7.2 Small Number of IN / OUT Arguments

A method requiring a large number of incoming and/or outgoing arguments may be performing too many tasks. Consider using the Fuego Object’s attributes before using incoming and/or outgoing arguments. If many arguments are needed, consider creating another Fuego Object to aggregate these values.

5.7.3 Make Method a Function If It has Only One Outgoing Value

Avoid using an *OUT* argument when there is only one outgoing value is returned. Methods with one *OUT* argument should instead be defined as functions.

5.8 Testing Equivalence of Two Fuego Object Instances



Never compare two variables of the same Fuego Object type using either `==` or the *equals* method. This does not test the two for equivalence (that each attribute in the two are equivalent). You should create a method in the Fuego Object to test equivalency.

For example, it is tempting (and incorrect) to test the equivalence of two objects directly using `==` as shown below (**aCust** and **bCust** are Fuego Objects of the type **xyz**).

```
// Directly testing two XObjects for equivalence always results in false
if (aCust == bCust) then
    // This is never true because the two variables
    // would have to be the same area of memory
```

The above “`if (aCust == bCust)`” FBL always evaluates to false. The only way this is ever true is if the two object variables are actually referring to the same object. Using either this test for equivalence or the `equals()` method does not do an object attribute-by-attribute comparison.

The reason is because there is a default java.lang method called `equals()`. This default method looks like:

```
boolean equals(Object o) {
    return this == o;
}
```

The example below shows the correct way to test the equivalence of two Fuego Objects. Note that the Fuego Objects each contain attributes called **name** and **address**. Note also that the **address** attribute is actually another Fuego Object that contains the attribute **city**.

```
// But checking to see if the two objects (aCust and bCust)
//      have equivalent values is a valid and appropriate way
//      to test equivalence.
if ((aCust.name == bCust.name) &&
    (aCust.address.city == bCust.address.city)) then
    display "The two XObjects are equivalent"
end
```

To test equivalence of two Fuego Objects, it is useful to create a Fuego Object `testEquivalence()` method (you will get an error if you attempt to override the `equals()` method automatically generated by FuegoBPM for the Fuego Object). This method should:

- Receive a variable of this Fuego Object’s type
- Define the method as a function and return a *Boolean* (*true* – equivalent, *false* – not equivalent)
- Add logic in this method to do an attribute-by-attribute comparison between the attributes of the current Fuego Object and the one passed in as an argument.

5.9 Referring to a Fuego Object from FBL

5.9.1 Use “this.” Syntax to Refer to the Current Fuego Object

When referring to a method or attribute in the Fuego Object FBL you are currently working on, always use the **this.**{attribute name} and **this.**{method name} syntax. Using *this.* improves clarity and avoids conflicts with any identically named local or argument variables.

5.9.2 Using Instance Variables to Refer to a Fuego Object

If you intend to use a Fuego Object throughout a process, declare an instance variable of the Fuego Object's type in the process and use this variable name to call the Fuego Object's methods or refer to its attributes.

5.10.3 Using Local Variables to Refer to a Fuego Object

When a Fuego Object is only needed during the execution of one FBL task, create Fuego Object local variables instead of a Fuego Object instance variable.

Fuego Object instance variables on the other hand persist through the life of the instance as it flows through the process. Engine performance will be improved by using local variables for Fuego Objects whenever possible.

5.10.4 Using Fuego Objects by their Class Names Can be Convenient

Using a Fuego Object explicitly can be convenient in the sense that it does not have to be declared (and instantiated) within the FBL. In the example below the Mail object is used with the same result.

```

if newUserOverdues < OVERDUE_LIMIT then
  send email
    using from = "Admin@mycompany.com",
          recipient = newUser.email,
          subject = "!!! Important !!!",
          message = "Please supply supervisor information..."
else
  send Mail
    using from = "Admin@ mycompany.com",
          recipient = newUser.email,
          subject = "!!! Important !!!",
          message = "Please supply supervisor information..."
end

```

5.10.5 Naming Fuego Objects

Avoid confusion by never naming a variable the same name as a Fuego Object.

5.11 Fuego Object Instance Variable Size

Be aware of the size of Fuego Object instance variables. If a Fuego Object contains all the information for one order – consuming 1k of memory, and someone wants a list of all the orders for a customer (1,000 orders), a megabyte of memory is consumed. This ties up memory unnecessarily. Be aware of the size of the Fuego Object and be very cautious about creating an array of Fuego Objects. Fuego Object variables not needed in Split-N / Join constructs can be set to *null* to free memory (see section 3.18 *Split-N Copied Instance Variables*).

Read from relational database tables instead of tying up megabytes of memory in instance variable information. To do this, keep a key attribute that references the Fuego Object's content. For example, instead of storing all of an order's information in one Fuego Object will consume more memory than storing only the order number (key value in a relational database). If the total order amount is needed for this order, the database order table could be read using the order number as the key.

5.12 Use Fuego Objects to Invoke Components

Invoke components using Fuego Objects. Always avoid invoking underlying components from process FBL. If the component changes, the change is made in one place instead of changing every process that calls the component.

Another advantage to wrapping component calls using Fuego Object methods is that errors can be handled inside the Fuego Object method. This error handling logic in the Fuego Object can then return a single value to the calling process that indicates the problem. Having this error handling logic in a Fuego Object simplifies the business rules in the calling process.

Always keep the process FBL logic free of statements that access components directly.

5.13 Fuego Object Testing

Simple unit testing from the Fuego Object debugger is fine for initial testing. Fuego Objects must also be tested from the processes. If the Fuego Object developer skips this step, it is a burden to the person doing the process design system integration testing later. Too much time will be spent too late in the development cycle debugging Fuego Object problems.



It is not uncommon to have a “Testing” folder defined in the project's processes where test processes can be created as a way of testing them in a “scratchpad” process without modifying the real process. When the objects are tested and working then they can be incorporated into the real process.

5.14 Database Considerations

Be aware that there is sometimes a balance between reuse and performance. For example having one component that retrieves the customer information and having a separate component retrieve the order information for that customer will result in two database queries. This would be equivalent to having a database query inside of another database query.

Instead, consider having one component that does a join on the tables using either a primary or foreign key of the tables. This will result in faster and more scaleable queries.

For the join to work efficiently, the two tables should be in the same database and the columns joined should be indexed.

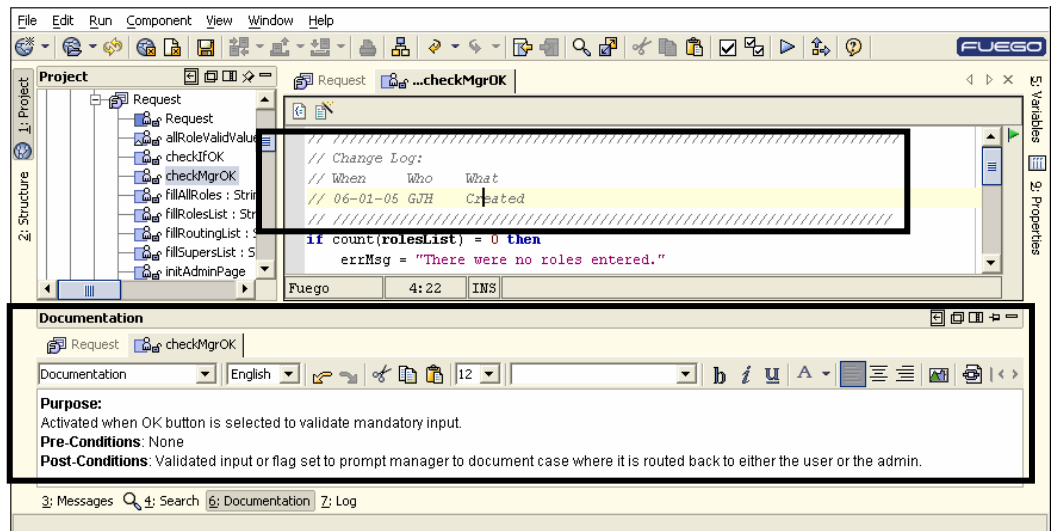
5.15 Fuego Object Comments

5.15.1 Use the Documentation Tab

The Doc. tab should be used to document both the Fuego Object as well as its encapsulated methods. In the example, the “Request” Fuego Object and the “checkIfOK” method are documented inside the Doc. tab.

5.15.2 Put the Change Log in the Method

Each Fuego Object method should begin with a comment describing its purpose, arguments and template as shown here.



Section

6

Screenflows and Presentations

6.1 Always Use Screenflows

Although you can write *input* and *display* statements directly into an interactive activity as a quick way to test, it should not be done in production for performance reasons. A thread on the FuegoBPM Enterprise server will be held for the life of each end user session. Always use screenflows.

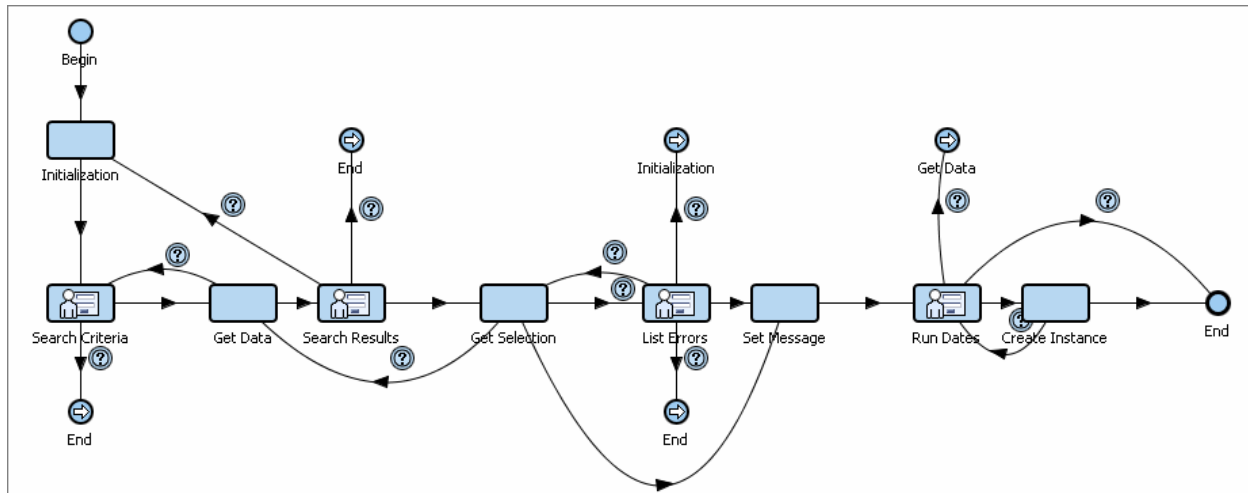
6.2 Standardize Screenflow Design

Good screenflow design incorporates elements of the “model-view-controller” design paradigm. The table below summarizes the screenflow elements that implement what aspects of “model-view-controller”.

MVC	Activity Type	Description
View	Presentation	Contain only logic to control user interface behavior
Model	Automatic	Data contained in databases is represented by the Model layer. Fuego objects that inherit from the database tables contain the data access methods of the Model layer. The Model methods are used in automatic activities before and after the presentation.
Controller	Automatic, Transitions	Most presentation activities are followed by an activity that will review interface results and determine what needs to be done next. Controller logic is not centralized as in a traditional MVC model but implemented via transitions and automatics that often follow the presentation elements.

6.2.1 Screenflow Example

Illustrated here is a screenflow example that is representative of good “model-view-controller” design. The screenflow provides a way for clients to be listed and from that list the user can add, update or remove a specific client.



The view activities (“Search Criteria” and “Search Results”) do not perform any control functions. Controller activities immediately after the presentations (or sub-screenflows) do this (“Get Selection” and “Set Message”).

There is no database access embedded in the presentation activities. Model activities (“Initialization”, “Get Data” and “Create Instance”) perform this function as well as performing controller functions.

Use sub-screenflows judiciously as there is argument mapping that must be done between the screenflows.

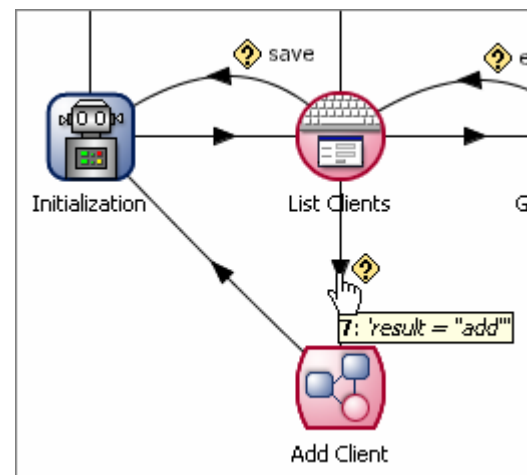
6.3 Screenflow Tips and Techniques

6.3.1 Use *result* variable to control transitions

In this screenflow example, result is being set by a button in the “List Clients” presentation that will cause the transition to be taken based on “result=add”. The button (type “submit”) has a method invoked that contains but one line:

```
submit("add");
```

Fuego will set the *result* variable to the value supplied in the submit action.



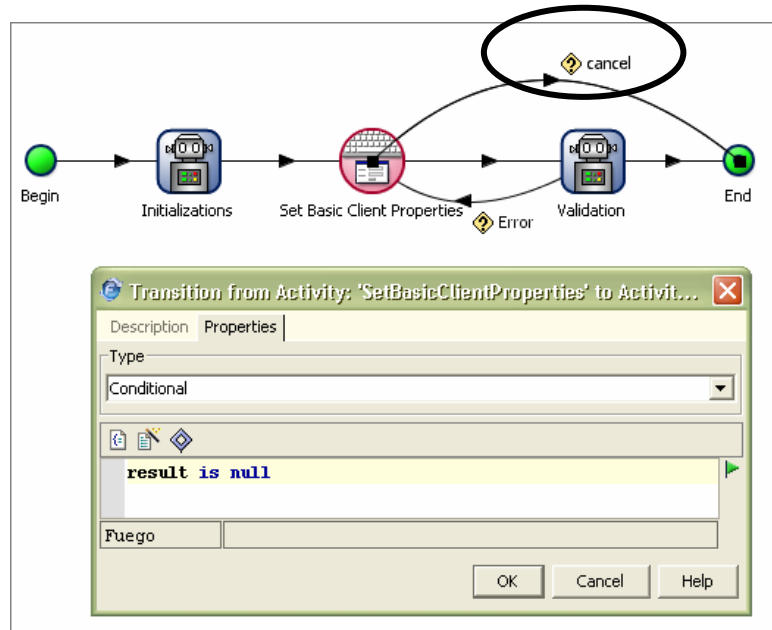


Be aware the result variable is not available when using custom JSP pages for the presentation so a designer-added variable would need to be created in lieu of result.

6.3.2 Use **result** variable to Cancel a Screenflow

Nearly all presentations have a “Cancel” button. Be sure to map the presentation’s “selectedButton” variable to the screenflow’s “result” variable. Then “result is null” will be true in the screenflow if the user selected the “Cancel” button in the presentation.

Another good practice is illustrated in this example. Notice the description of “cancel” in the screenflow. It is defined in this dialog in the “Description” tab and provides a nice way to document transitions.



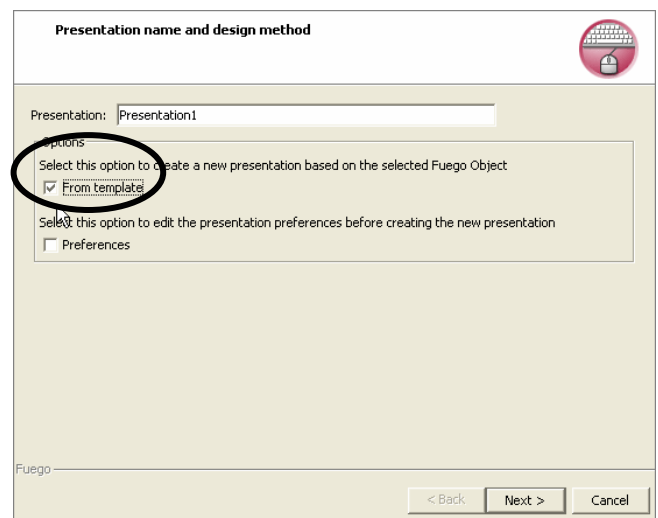
6.4 Fuego Presentations

This section provides some lessons learned using Fuego presentations and is not intended to teach you how to create them.

6.4.1 Template Preferences

For consistency and ease of use, build Fuego Object presentations using the *From template* option as shown in the example.

Since a Cascade Style Sheet is recommended the “Preferences” option provides little value.



6.4.2 Consistent Template Preferences Across Project Team

To ensure a common look and feel of Fuego presentations, the project team should use the same settings for the Fuego Template Preferences. For consistency, some recommendations for these settings are:

Template Preferences Tab	Property	Setting
General	Presentation Background	<i>White</i>
Table	Padding	<i>10</i>
Cell	Alignment	<i>Left</i>
Text	Column Count	<i>20</i>
	Alignment	<i>Right</i>
Combo	Alignment	<i>Left</i>

6.5 Decide on Project-Wide Presentation Approach

When developing presentations for Fuego screenflows there are two approaches available: Fuego presentations and presentations developed externally as custom JSP (Java Server Pages). A mixture is supported but generally one or the other is used to enforce consistency in the user interface.

6.5.1 Advantages and Disadvantages To Using Custom JSP's

Advantages: A richer user interface can be developed using custom JSP in most cases. If Javascript is used extensively in the custom JSP's then a performance benefit will be derived using client- rather than server-side validation. Further, the Fuego supplied tag libraries (JSTL) provide a means of easily accessing and formatting Fuego objects in custom JSP's.

Disadvantages: Additional HTML and JSP developer resources must be added to the project. If Javascript is not used to any great extent in the custom JSP's then there will be little performance advantage using custom JSP's rather than Fuego presentations.

The Fuego supplied tag libraries (JSTL) provide a means of easily accessing and formatting of Fuego objects in the custom JSP's.

6.5.2 Advantages and Disadvantages To Using Fuego Presentations

Advantages: Sophisticated presentations can be developed within FuegoBPM Studio, nearing what can be done externally in custom JSP's, without the need for the project team to use non-Fuego skilled development resources.

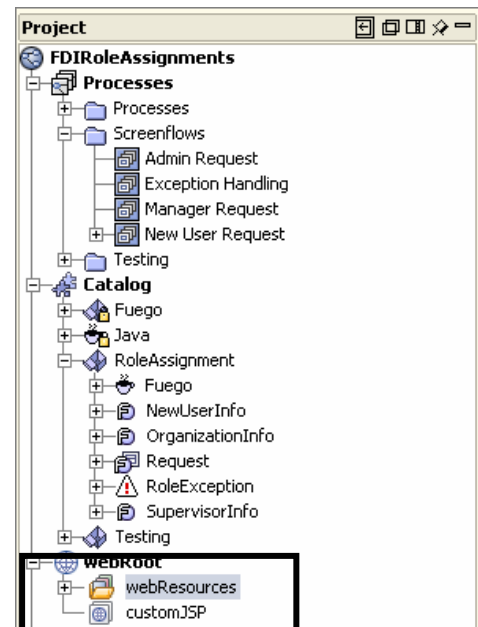
Disadvantages: All validation is done server-side and so there is a small performance “hit” taken using Fuego presentations. Added up across hundreds of web pages and thousands of users however, this may mean considering a custom JSP's rather than Fuego presentations approach.

6.6 Organize Your Screenflows and Presentations

Screenflows should be readily distinguishable from other process artifacts. In a large project this may mean creating a “Screenflows” folder to contain them.

It may also require the creation within this folder of other folders to further separate one business area's screenflows from another.

When used, Custom JSP's will be visible within the “webRoot...customJSP” folder. The JSP resources will be stored in the source repository when using Fuego's version control system (VCS).



6.7 Use Cascade Style Sheet

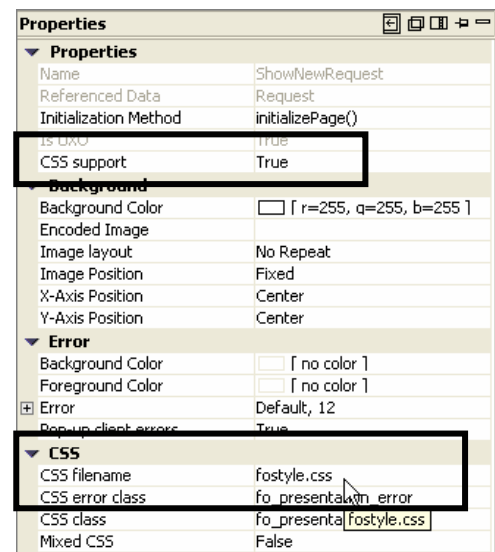
Regardless of the presentation approach (Fuego or JSP presentations), use a Cascade Style Sheet (CSS) to enforce a common-look and feel.

6.7.1 Define CSS on each presentation

At the presentation level in the Fuego Presentation designer indicate that CSS will be used for the page.

6.7.2 Define the CSS Filename

By default, the “fostyle.css” filename will be used in each new presentation.



It may simplify matters to use this stylesheet filename as well as the default CSS class names (e.g. “fo_text”, fo_label) in your project’s presentations.

6.7.2 Prototyping With the Stylesheet in Studio

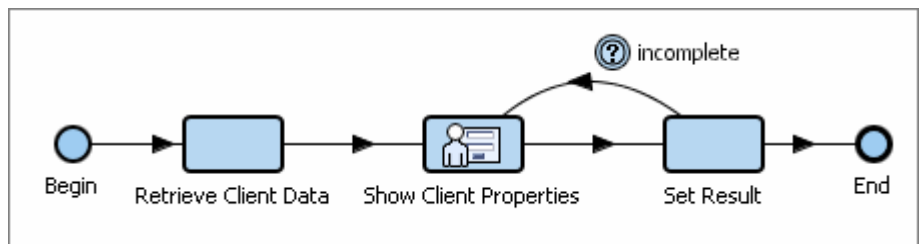


The stylesheet must be copied to “C:\Documents and Settings\your_windows_id\Local Settings\Temp” to use the HTML prototyping feature. When publishing to FuegoBPM Enterprise it must be copied to “FUEGO_SERVER_HOME\webapps\portal\css”.

6.8 Populating Combo Boxes

6.8.1 Do Not Populate Combo boxes from a Database in the Presentation

An earlier topic covered good screenflow design which articulated the use of *model* elements to contain the logic the database access logic. For combo (or dropdown) boxes in a Fuego presentation this would be done in an automatic activity (“Retrieve Client Data”), prior to the execution of the “Show Client Properties” presentation.



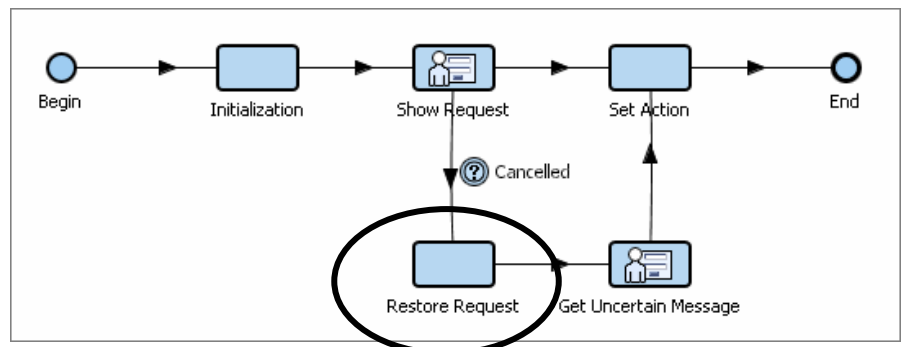
Do not embed SQL inside

presentations. In the above example each time the “Show Client Properties” is re-executed in the case of an “incomplete” transition the SQL will run. If the combo box is populated once then the traversal back to “Show Client Properties” results in no additional database access.

6.8 Canceling Screenflows

A whitepaper is available on the Fuego website (“Cancelling Screenflows” by Pablo Victory) which describes how to cancel changes made in a screenflow to the presentation object. The key points made in the whitepaper are:

1. Clone the presentation object (“Initialization”) in the first activity (before changes are made to it)
2. If the user selects the Cancel to restore the presentation object to the cloned object (“Restore Request”.



Section

7

Portal Administration

7.1 Always Use Work Portal Views

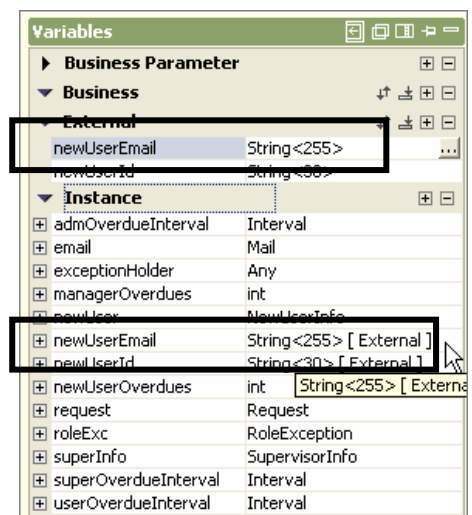
The default view in the Work Portal lacks the business information the process users should see when viewing their processes. Generally speaking there is too much information provided in the default view. There are toolbars and other options present as a default, regardless of the user's role in the process. For these reasons role-based views for the project should be designed early on in the project.

7.2 Define External Variables for Display in the Work Portal

Create external variables for the project to add a meaningful identifier and descriptor of process instances shown in the Work Portal.

In the following example, the user's email address is an important fact to display in the Work Portal. An *external* variable of “newUserEmail” was defined with a property of “use in the process”.

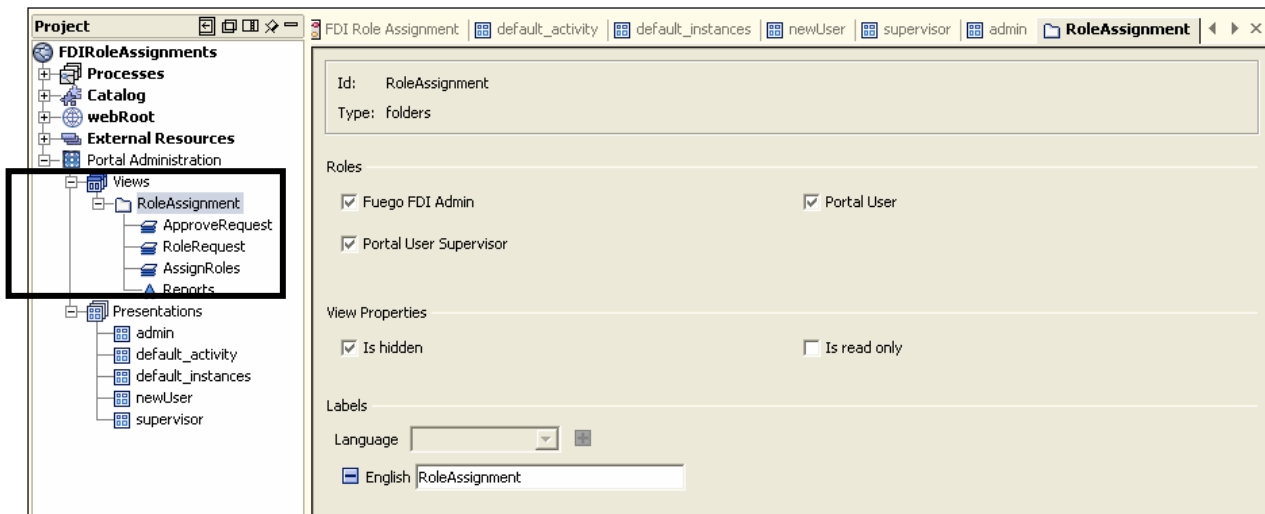
Obviously the label “newUserEmail” is not one you would want shown in the Work Portal. For this reason when you create the external variable select “Locale” in the variable creation dialog and give it a user-friendly name like “Email Address”.



7.3 Design Work Portal Views From Left to Right

From the left side of the Work Portal, that is. As a process designer consider how at a high level, how the inbox instances, applications and attachments be organized. In other words, *how should the left side of the Work Portal be organized?* Do you want all instances for all processes to be in the *Inbox* and the global functions all listed in the *Application* folder? Or alternatively, should these process artifacts be put into a

folder by process name or business function? In the example below, process views are defined and added to a view folder called “Role Assignment”.



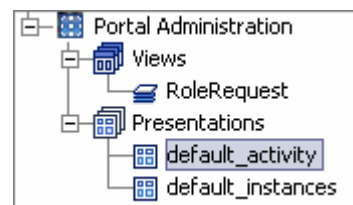
7.4 Use Portal View Folders

In most cases an organization will deploy tens, perhaps hundreds of processes. The Work Portal user must be presented with an organized presentation of his or her work to be done in the organization.

7.5 Always review the “default” Presentations

Presentations are the layout with which a view will be shown in the Work Portal. Two presentations, *default_instances* and *default_activity* (shown below) are provided. Many times there is too much detail available in the default presentations so it's a good practice to always review the presentations to perhaps remove information that may confuse rather than clarify what is shown in the Work Portal.

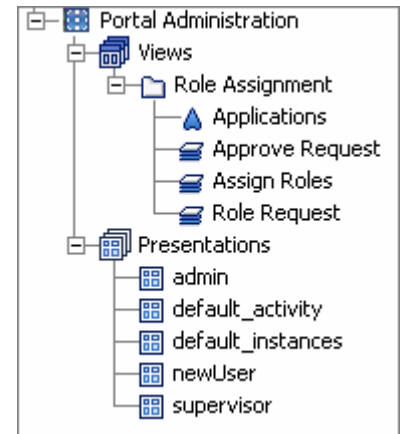
New presentations can be created in Portal Administration but to be sure all presentations by default make use of the Work Portal views and external variables modify both defaults (*default_activity* and *default_instances*).



7.6 Create Role-Based Presentations

Consider customizing the presentations by process role. If needed, external variables can be defined that are meaningful to the Work Portal user. More times than not there is different information that should be shown in a process based on the Work Portal user's role.

In the “Role Assignment” example, the views make use of presentations designed around the roles in the request approval process. A “newUser” presentation is implemented by the “Role Request” view. A “supervisor” presentation is implemented by the “Approve Request” view.



7.7 Use Portal Console to Limit Toolbar Options

In most cases there are toolbar options displayed in the Work Portal that are not relevant to each role in the process. Although the Portal Console is not within the scope of this best practices document it is mentioned here as a “to-do” for the process designer.

In the example the checked toolbar options will appear in the Work Portal for the “User” view.



Section

8

Exception Handling

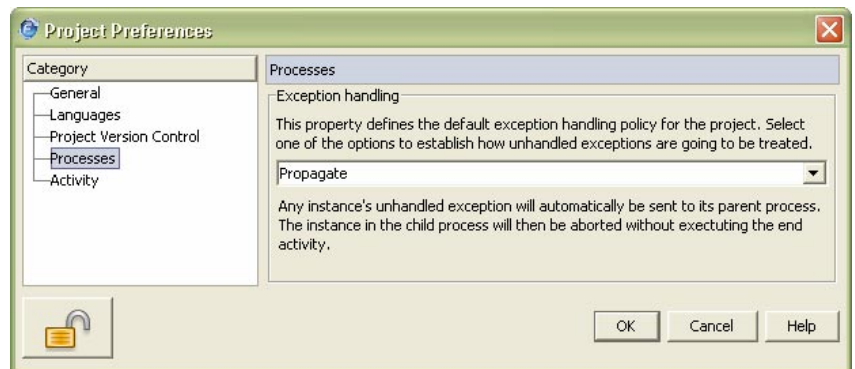
8.1 Plan for Exception Handling from the Start

Deal with exception handling from the inception of development. It is difficult to add effective exception handling during implementation. If you know a component method may throw an exception, include appropriate exception handling logic.

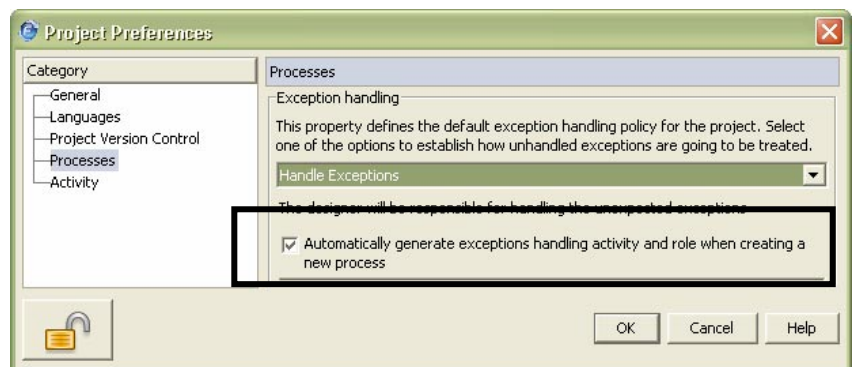
8.1.1 Decide on Centralized or De-centralized Exception Handling

The “File...Project Preferences... Processes” option decides how exceptions are to be handled within the project. The process designer typically sets a guideline that all exceptions are to **Propagate** from sub-process to the top process where they are handled by a centralized exception handler.

The “propagate” choice can be overridden with in each sub-process by implementing an exception handler in the sub-process so the value exists as a preference.



If the **Handle Exceptions** option is elected by the process designer then an additional option can be selected to allow Fuego to generate a role and activity in every newly



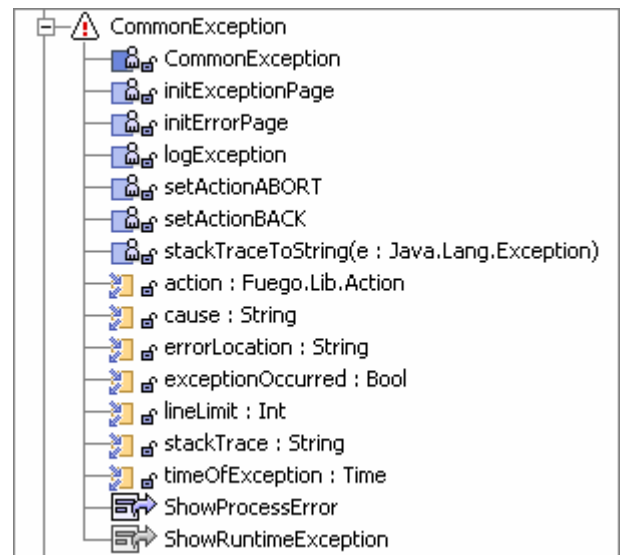
create sub-process to provide de-centralized exception handling.

8.2 Develop Common Exception Handling

Although specific requirements will vary for each process, many processes can get by with common generic components to handle exceptions. Broadly speaking there are two types of errors:

- Errors raised as a result of an explicit condition checking in the Fuego FBL (i.e. “process” error)
- Unexpected runtime errors captured by the process (i.e. “runtime” exception)

A generic Exception component would consist of the attributes to handle both process errors and runtime exceptions as well as containing the presentations to display the appropriate error.

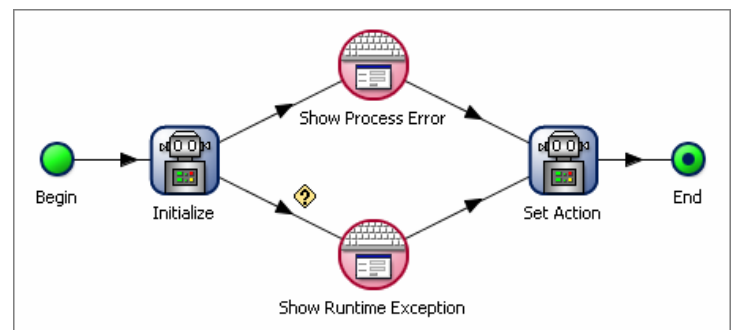


8.2.1 Create An Exception Fuego Object for the Common Catalog

In most cases a common exception handler component will accommodate most process exception requirements. Move this component into the common catalog to share it with other projects. Specific exception handlers would be contained within the projects that need them.

8.2.2 Create two different Exception Presentations

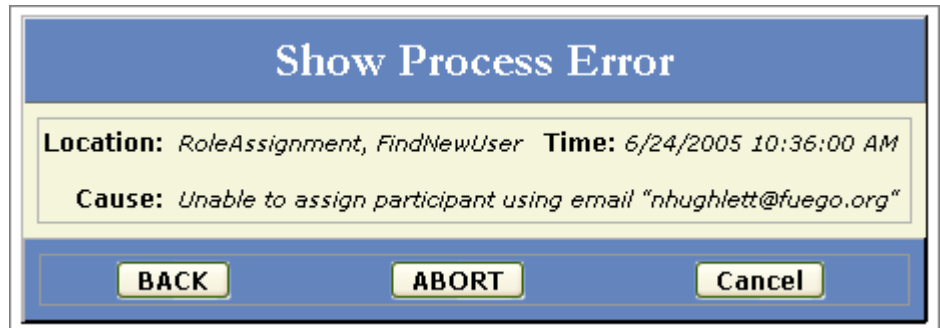
The sub-process or screenflow that implements the Exception Handling should contain at least two separate presentations to differentiate between *process errors* and *runtime exceptions*.



8.2.3 A Generic Process Error Presentation May Not Always Do the Job

A process error occurs when a specific condition is found in the process that violates assumptions or rules required by the process.

This generic presentation shows where, when and why the process error occurred. In specific cases, however, a custom presentation may be better suited and could be included in the same exception handling process.



8.2.4 The Runtime Exception Presentation should Show Critical Information

In case of a runtime exception more information needs to be shown, like the location of the error, the *time* it occurred and the cause.

The Java stack trace output should be shown (limit the # of lines shown in this presentation). The entire stack trace should be written to the Fuego log.



8.2.3 Stack Specific Exception Handling Logic



Some FBL invoking components can throw several different exceptions. Stack these using FBL logic shown below. As shown below, the exceptions are delegated to the process exceptions *badArgument* and *dbDown*.

```

on IllegalArgumentException do
    logMessage process.name + ": " + activity.name + ": Illegal Arg exception was: " +
        IllegalArgumentException.localizedMessage
        using severity = WARNING
    throw badArgument
end
on SQLException do
    logMessage process.name + ": " + activity.name + ": Sql exception was: " +
        SQLException.localizedMessage
        using severity = WARNING
    throw dbDown
end

```

Process Exception

Process Exception

The advantage to using process exceptions is that the operation that failed can be correctly handled in a process activity.

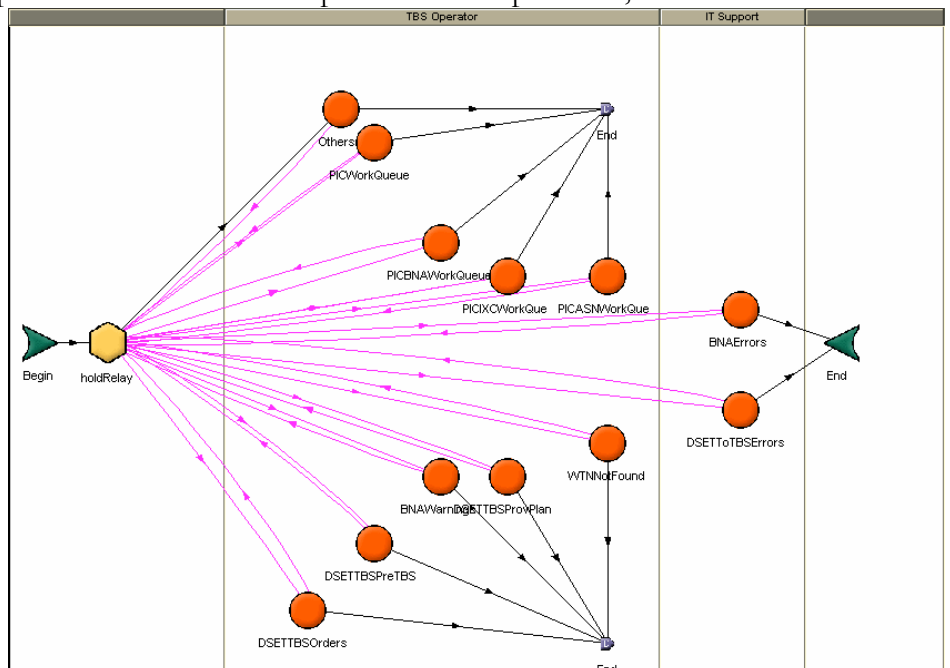
8.2.4 Log Messages in Exception Handling

Include logMessages in all exception handling logic as shown above. Set the severity level to at least WARNING.

8.3 Exception Handling Process

Rather than throwing process exceptions for non-critical exceptions in batch processes, instead create a new instance in a separate exception handling process. Catching a process exception in the typically stops execution of batch processes. By creating an instance in a separate process, the batch process can continue processing.

Exception handling process typically looks like the one shown below. The various processes create instances into this process. One of the input argument variables passed in is the type of problem that occurred. The problem type determines the conditional transition flow to the



correct Interactive activity.

8.4 Use *on exit* to Release Resources



Component problems that result in a thrown exception may lock a resource (e.g., threads). Monitor resources closely and consider “clean up” FBL to release them. The *on exit* syntax (shown below) can be used for this purpose. This section of FBL is always executed as the FBL completes.

```
on IllegalArgumentException do
  throw CaughtIllegalArgumentException
end

on exit do

  // Some "clean up" logic here

end
```

8.5 Try to Fix Exceptions Locally

If an error can be handled inside the same activity’s FBL, do so. Only “throw” user-defined exceptions when necessary. If problems can be fixed in the context of the FBL task fix it there. The “try/catch” syntax shown below is sometimes useful for handling exceptions locally. This will improve execution speed.

```
// Try
do
  badInteger = "SomeNonIntValue"
  someInteger = Int(badInteger)
// Catch
on IllegalArgumentException
  // fix the problem here
end

// Enter CIL here after the Try / Catch
```

Consider moving this try/catch logic to reusable FBL if the same logic is being processed from more than one FBL task.

8.6 To Throw or Not to Throw Exceptions in Fuego Objects

While it is technically feasible to create components that throw exceptions good process and component design would dictate that exceptions are only thrown in a process context. In the many contexts in which components can be used in context an exception might be the proper response but in other cases it might not. It depends on the process!

8.7 Exception Email

Include the process name, activity name, server and environment (e.g., QA, production) in emails sent to alert users of where problems are occurring. As discussed in 3.21.1 *Include the Process and Activity Names in Log Messages* do not hard code either the process or the activity names. Instead use the variables **process.name** and **activity.name** for the process and activity names respectively.

8.8 Exceptions in End Activity Arguments

8.8.1 Don't Forget to Map the Exception in the End Activity

Any sub-process (or screenflow) that throws an exception that is not handled within the sub-process must be defined as an output argument of the End activity.

EndOut

Output arguments

Argument		Instance variable
requestArg (RoleAssignment.Request)	=	request (RoleAssignment.Request)
resultArg (String)	=	result (String)
roleExcArg (RoleAssignment.CommonException)	=	roleExc (RoleAssignment.CommonException)

roleExcArg (RoleAssignment.CommonException)

OK Cancel Help

Section

9

Database Access

9.1 Separate database objects from other Fuego Objects

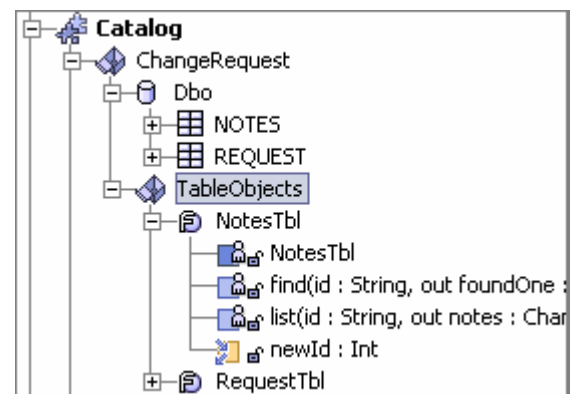
Observing a “Model-View-Controller” design paradigm, the database access (Model) is separated from other aspects of the process. The performance and concurrency of database access in a Fuego process warrants this separation.

9.2 Create database table objects as heir to Catalog tables.

Follow the following steps to create a Catalog structure for each database you will be accessing in the process. (These steps assume you have already defined the necessary external resources for the database.)

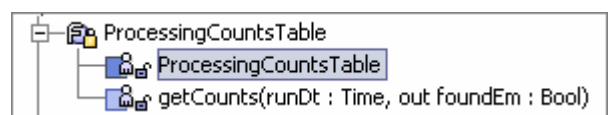
- In the project Catalog define a module for your database (e.g. “ChangeRequest”).
- Inside of this module create another module called “TableObjects”.
- Catalog the database tables (e.g. “NOTES”) in the top level database module (*ChangeRequest*).
- Create an heir for each catalogued table (e.g. “NotesTbl”) with a destination of *TableObjects*.

Adding a “Tbl” (or “Table”) suffix to the *TableObjects* members may provide more clarity in the FBL of any method referencing the database objects.



9.3 Override the accessDatabase default?

The inherited “load” method will automatically execute when the table’s primary key attribute is populated (“accessDatabase=true” is the default). In some cases



this behavior may be inappropriate or confusing. The good news is that the behavior can be easily changed.

Add the “accessDatabase=false” statement to the table object’s constructor method (e.g. “Processing CountsTable”).

9.4 Decide What Type of SQL Syntax is Appropriate, Case by Case

There are a variety of ways of running SQL in a FuegoBPM process (static, dynamic SQL and stored procedures). You should use the appropriate SQL technique case by case rather than using one way for all of the SQL. For example, stored procedures will always outperform static and dynamic SQL. However, stored procedures require specialized expertise to write them and are harder to change than implementing the SQL as static or dynamic SQL in the FBL.

9.4.1 Use Static SQL for Simple, Single Table Queries

When database tables are cataloged in Studio it is extremely easy to write “static” SQL on the tables. For simple single table queries this is many time the best path to follow. It allows you to switch DBMS vendors without having to rewrite the SQL in the FBL. (The database External Resource must be modified to point to a different database.)

9.4.2 Use Dynamic SQL for Complex or DBMS Specific Queries

Switching DBMS doesn’t usually happen however and most SQL isn’t simple. Furthermore you may want to take advantage of features offered by the DBMS SQL syntax and this is only available when you use dynamic SQL.

Another excellent reason for expressing a query dynamically is the query can be written to the log and then executed verbatim in a query tool. The inverse of this is true as well. SQL can be developed, tested and prototyped outside of FuegoBPM and then easily implemented in dynamic SQL.

9.4.3 Log Dynamic SQL Statements and Elapsed Time

In the example below not only is the SQL logged but so is the time of execution. Marking this type of statistic as INFO in the log allows the performance statements to be reviewed for performance assessment.

```
beginTime = 'now'
sqlStmt = "SELECT top 1"
        +" PRAD_TYPE_CHECK AS checkind"
        +", PRAD_TYPE_PRIM AS primind"
        +" FROM CMC_PRPR_PROV with (nolock)"
        +" WHERE (PRPR_ID = '" +prprId +"')"
```

```
result = executeQuery(DynamicSQL, sentence : sqlStmt,
                      implname : "Some Database",
                      inParameters : [])
```

```

for each col in result do
    ...
end

logMessage methodName +", ELAPSED time: " +('now'-beginTime)
    +"for query: \"" +sqlStmt using severity=INFO

```

9.4.4 If Logging SQL, Don't Parameterize Dynamic SQL Statements

In the preceding example the SQL is being logged for later review and possibly to re-run the query in an external SQL query tool. If the variables constituting the query had been parameterized the statement would not have been able to be logged in a manner that would allow it to be recreated easily in an external query tool.

```

sqlStmt = "SELECT top 1"
    +" PRAD_TYPE_CHECK AS checkind"
    +" , PRAD_TYPE_PRIM AS primind"
    +" FROM CMC_PRPR_PROV with (nolock)"
    +" WHERE (PRPR_ID = ?) "

params[]=prprId

result = executeQuery(DynamicSQL, sentence : sqlStmt,
    implname : "Some Database",
    inParameters : params)

logMessage methodName +", ELAPSED time: " +('now'-beginTime)
    +"for query: \"" +sqlStmt using severity=INFO

```

The logMessage statement would not correctly record the query necessary for it to be recreated externally.

9.4.5 Implement SQL Query as a Stored Procedure as a last resort

The general advice for most SQL written for a FuegoBPM process is to not implement the queries as stored procedures as they are DBMS specific, harder to implement and requires a database migration when they change. There are cases however, when a complex query can be best optimized when implemented as a stored procedure.

9.4.6 Do Not Write Cross-Database Queries

Although it is technically feasible to write dynamic SQL joins tables from different databases they are difficult to manage from as an external JDBC resource, not to mention the special permissions needed for a database login to more than one database.

Section 10

Process Design Considerations

10.1 Procedures or Sub-processes?

As a process designer there exist several alternatives available when adding an automated activity to a process model. One of the implementation options available for an automated activity is “procedure”. This implementation type allows you to implement the automated activity as a series of automated activities and transitions.

A procedure is distinguished from a sub-process in that you cannot have interactive activities and roles in a procedure. There is however another factor before using procedures.

10.1.1 Procedures are Atomic

Even though a procedure will consist of a number of automated activities, they are executed as one transaction (or unit of work). A good way to think of a procedure is if all of the procedure activities are all implemented inside one method; the procedure implementation simply allows you model the procedure’s tasks in a graphical manner.

10.1.2 Concurrent Executions Property Irrelevant in Procedure

Although this property is available for each activity in a procedure, assume that it doesn’t. Since procedures are atomic this property does not apply.

The screenshot shows a software interface for configuring an activity. On the left, a 'Category' list contains 'Activity Id', 'Advanced', and 'Image'. The 'Advanced' tab is active, displaying two main sections. The first section, 'Unlimited concurrent executions', includes a text description and a checkbox labeled 'Unlimited concurrent executions' which is checked, with a value of '100' in an adjacent field. The second section, 'Generate Events', includes a text description and three radio button options: 'Default', 'Generate Events', and 'Do Not Generate Events', with 'Do Not Generate Events' being the selected option. At the bottom right, there are 'OK', 'Cancel', and 'Help' buttons.



If an activity needs to have a restricted Concurrent Executions property value (most often it is set to “1” when not “Unlimited”) then you need to use a sub-process not a procedure.

10.2 Split-N Performance vs. Concurrency

Split-N circuits can be useful in many cases for improving process performance but in other cases it must be adjusted, for example, if there are many database access and updates occurring in the Split-N circuit then performance could be adversely affected.

By adjusting the “Maximum number of simultaneous copies” advanced property of the Split-N activity and reducing the number of copies pushed through the circuit at one time concurrency may be improved. Obviously performance is affected by this adjustment but it may be an acceptable trade-off.

10.3 Other Split-N Factors to Consider

There are other using Split-N circuits lessons learned.

10.3.1 Turn Off Generating Events

To limit the amount of I/O performed by the FuegoBPM Enterprise server, turn off the “Generate Events” property on activities in the circuit.

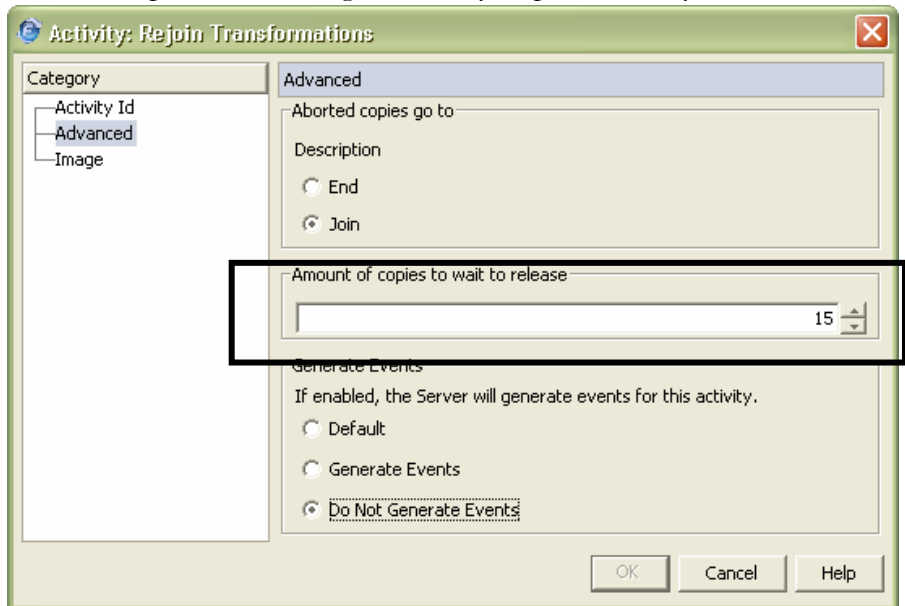
10.3.2 Throw Exceptions With Caution

An administrator charged with handling process exceptions can be overwhelmed by hundreds or perhaps thousands of exceptions thrown from a **split-join** circuit. Furthermore, if the exceptions are need to handled at the **copy** level then the exception handling process should be implemented within the circuit and not in a top level process which may be the default.

10.3.4 Establish a Split-N Upper Bound

Technically there is no limit to the number of copies that can be generated by a Split-N activity. Consider creating a business parameter or server property that establishes a limit beyond which some notification action occurs to let someone know the limit has been reached.

Another escape valve is to set the “Copies to wait to release” property of the join activity to some number beyond which the main instance will be released.

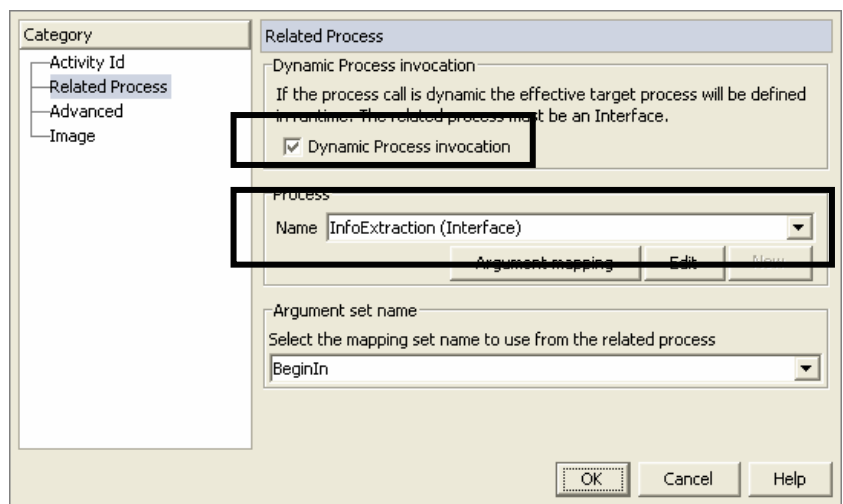


10.4 Dynamic Process Invocation

10.4.1 The Sub-process Properties Dialog

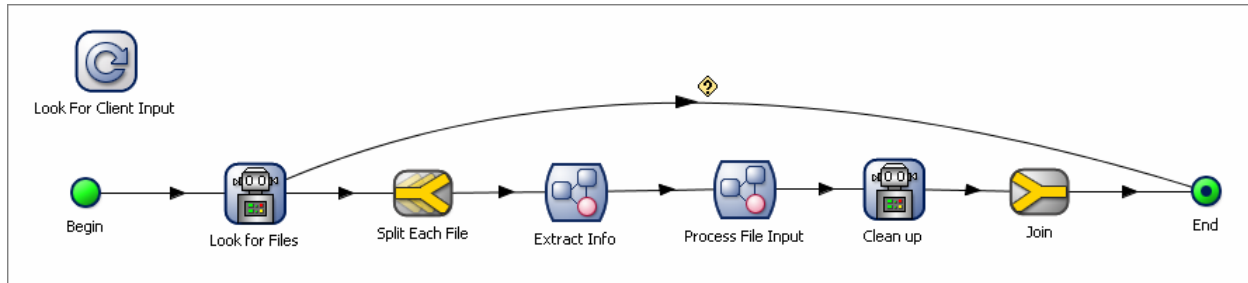
FuegoBPM provides the ability to model process interfaces, an abstract sub-process layer that is invoked dynamically at runtime to invoke a specialized sub-process. The dynamic process invocation feature requires that the sub-processes that execute dynamically at runtime observe the arguments defined by the process interface.

As a way of introducing the subject, shown is the sub-process properties dialog. Notice the dynamic process invocation checkbox. The process name makes reference to a process *interface* and abstract process that must be implemented by a real sub-process. Clearly, an example is needed to fully explain how dynamic process invocation works.



10.4.2 A Dynamic Process Invocation Example

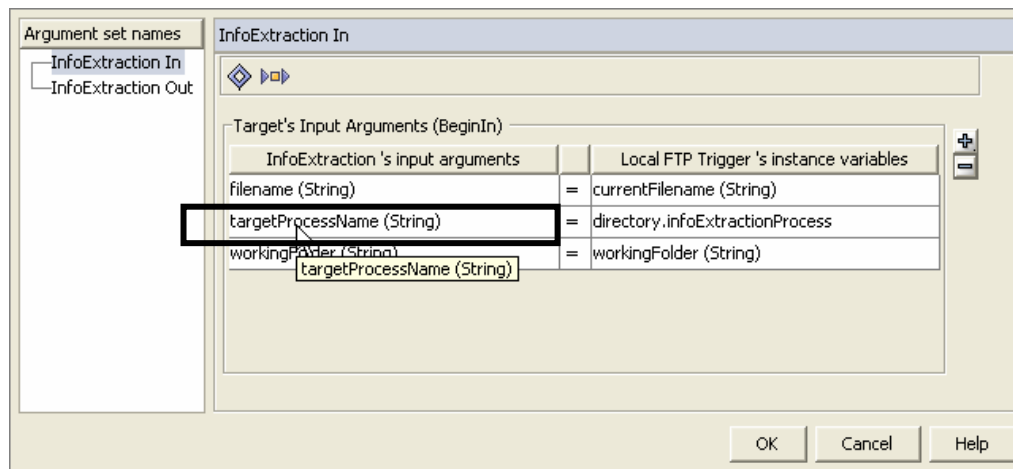
The process example is one where a company receives files from a known list of clients and the files have to be unpackaged according to whatever packaging rules each client has chosen to implement.



In this process an automatic global (“Look For Client Input”) checks the file system client directories for input every 30 seconds. When the global automatic find client files it creates an instance in the process. There can be 1-to-many files for each client (instance) in the process so a Split-N activity is equipped to handle this business rule.

The “Extract Info” sub-process must be customized for each client as each client may have specific packaging requirements. Some will send the files in a ZIP format while others will send the file encrypted using a PGP (Pretty Good Privacy) algorithm or any combination of the above (a PGP file in a ZIP file with a ZIP password, for example).

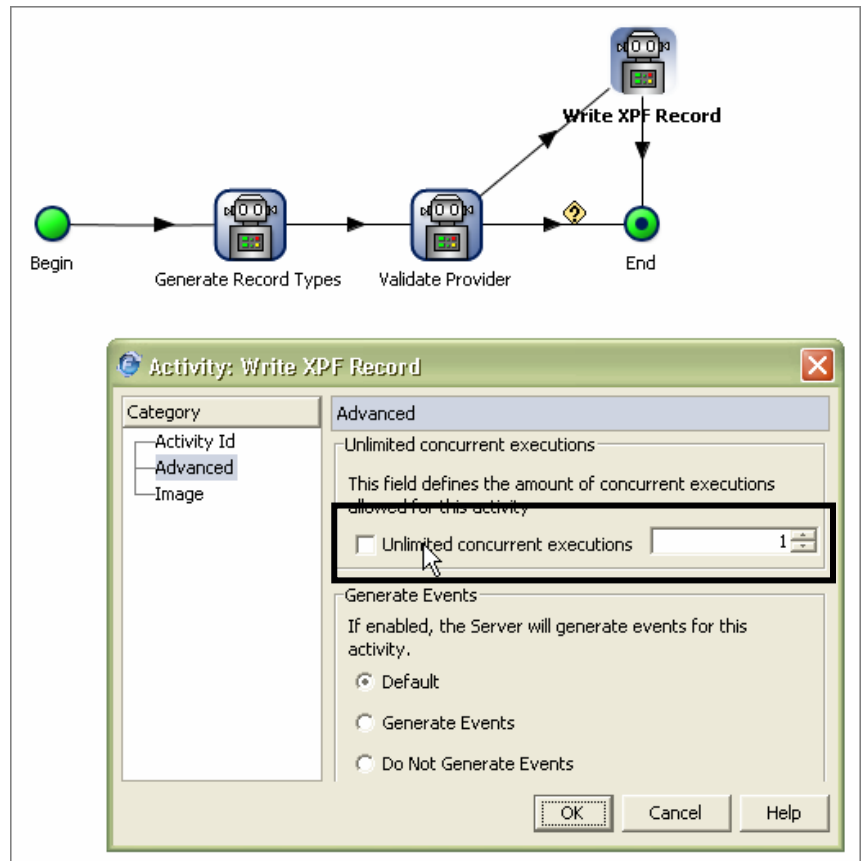
Extract Info is defined as a sub-process “interface” with the “dynamic process invocation” property set. At runtime the reserved variable “targetProcessName” must be provided with the name of an implemented sub-process that shares the same common input and output arguments.



10.5 Synchronizing Activities

There are cases in a process when an activity has to be “synchronized” or run in a serialized fashion to avoid issues with multiple threads writing to a file or updating a control table.

In sub-processes (not procedures) there is an ability to adjust the “concurrent executions” property to change it from “unlimited” (default) to some value. In the example, “Write XPF Record” writes specific output lines to a file and it must be run single-threaded so as to not result in multiple thread writing to the file at the same time, resulting in interleaved output.



Section

11

Business Parameters

Whenever the behavior of the process can be modified by the value of a process variable and that process variable needs to be externally controlled by the business users then Fuego business parameters are an effective way to do this.

11.1 Define Meaningful Business Parameters

Judicious use of business parameters should be the rule as they are visible and modifiable within the FuegoBPM Enterprise server.

The rule should be that only parameters meaningful to the business process users should be defined as business parameters and the rest configured via some other means.



Although business parameters can be modified within the process, they should not be. Reasonable default values should be assigned and changes to the business parameters should only be made from the FuegoBPM Enterprise Web Console.

Variables	
Business Parameter	
ADMIN_OVERDUE_HRS	Int [1]
DEFAULT_HRS	Int [4]
MGR_OVERDUE_HRS	Int [4]
OVERDUE_LIMIT	Int [4]
USER_OVERDUE_HRS	Int [4]
Business	
External	
newUserEmail	String<255>
newUserId	String<30>
Instance	
admOverdueInterval	Interval
email	Mail
exceptionHolder	Any
managerOverdues	int
newUser	NewUserInfo
newUserEmail	String<255> [External]
newUserId	String<30> [External]
newUserOverdues	int
request	Request
roleExc	RoleException
superInfo	SupervisorInfo
superOverdueInterval	Interval
userOverdueInterval	Interval

11.2 Use Business Parameter to distinguish DEV, UAT or PROD

In some cases the process may need to access services and data from different sources depending on whether the process is running in a development , QA or production environment. Rather than hard-coding values for variables that are environmentally dependent, use a business parameter to conditionally set the values. When the process is published to those environments, set the value in the Enterprise Web Console and you're ready to go! In the example below, *ENVIRONMENT_NAME* is retrieved and checked for "test", "uat" or "prod".

```
obj = BusinessParameter.getValue(name : "ENVIRONMENT_NAME")
envName = String(obj)

case substring(toLowerCase(envName), first : 0, last : 4)
  when "test" then
    comDocPath="\\\\server010.abc.net\\MS Files-DEV"
  when "uat" then
    comDocPath="\\\\server020.abc.net\\MS Files-QA"
  when "prod" then
    comDocPath="\\\\server020.abc.net\\MS Files"
  else
    logMessage methodName + ": envName [" + envName + "] is INVALID!"
    using severity=SEVERE
    exit
end
```

Section 12

Project Version Control

This section is limited to best practices surrounding the use of version control in Studio. Rather than version control, source control is more to the point. The examples presented in this section utilize the Concurrent Versioning System (CVS) open source product but would generally apply to other version control products supported within Studio (e.g. ClearCase).

Some Studio VCS terminology needs to be explained.

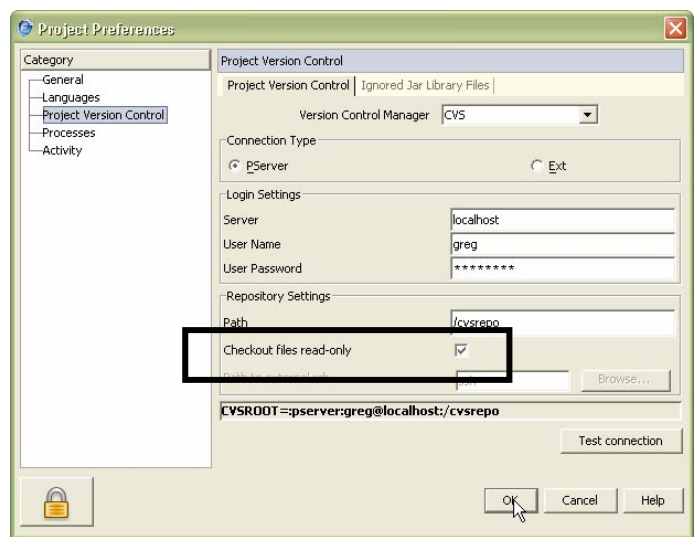
- “Update Local” means to *checkout* changes from the Version Control repository and update the local project files.
- “Commit” means to check-in changes from the local project to the Version Control repository.

12.1 Checking Out a Project

12.1.1 Use the “Checkout files Read-Only” Option

When the project is checked out of the version control system, all files will be read-only. The developer will not be able to change any project files within Studio unless they request “edit” permissions. If someone else is editing the object then the developer requesting edit will have to wait until it is committed to source control.

When a process artifact is changed and the changed checked into source control, the files just edited will again be marked as



“read-only”.

12.2 Checking in Project Changes

The check-in recommendations assume the project has been checked out with the “read-only” option.

12.2.1 Check-In Changes When They Have Integrity

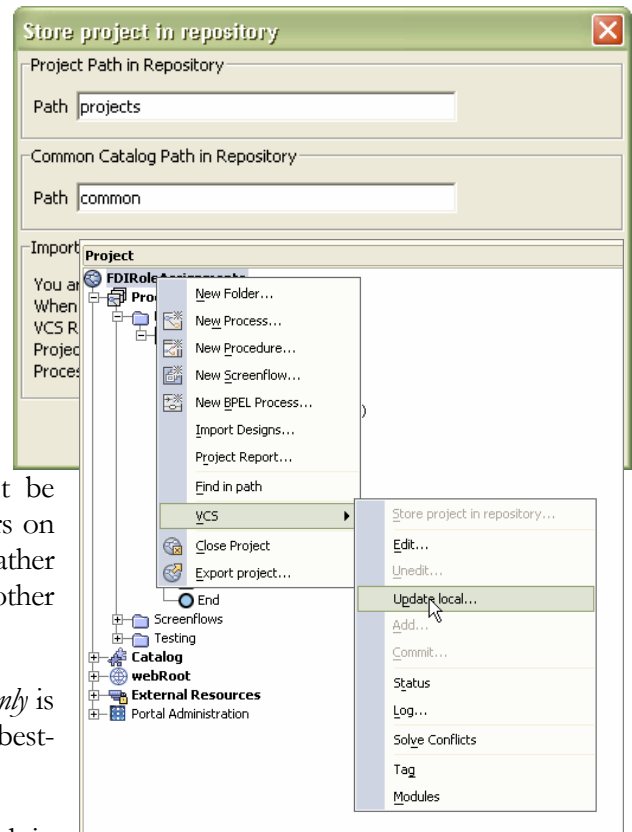
You are sure to make many enemies if you check in source components that have errors that will either cause compile errors in Studio or that will fail outright when the process runs. You should only check-in changes after you have tested the changes thoroughly or if someone else on the project team needs the changes to complete his or her work.

12.2.2 Always Checkout (“Update Local”) before Checking-In (“Commit”)

The changes made in one developer’s workspace must be reconciled with the changes made by the other developers on the team. (VCS source control systems use “optimistic” rather than “strict” locking there is a possibility of overlaying another developer’s changes.)

This is an absolute requirement if the *Checkout files Read Only* is not selected and even if it is selected it is still a good best-practice.

Always perform the “Update Local” at a level at least as high in the project for which you will be committing changes. If unsure, do it at the project level.



12.2.3 Backup Your Project Before Check-In

Export your project changes before checking into the version control repository.

12.2.4 Check-In At the Lowest Level in the Project

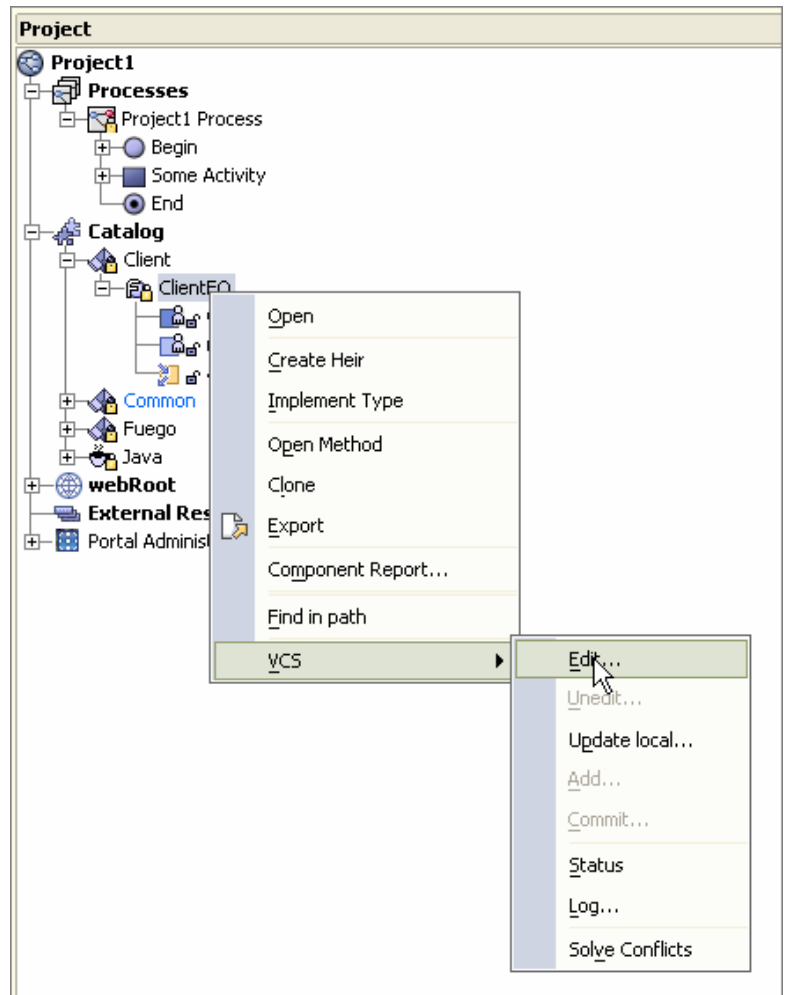
Based on the VCS Status command just performed; check-in only the “locally modified” source components.

12.3 Request “Edit” Permission At the Lowest Level

If you request “Edit” permission of a source component at a level too high in the project then other team members will not be able to make changes in the project.

In the provided example, the developer is requesting “Edit” permission on the “ClientFO” object component. Studio will check the Project Version Control system to ascertain whether this permission can be granted. If someone else beat you to the punch then you will not be able to edit the component.

If you change your mind about requesting “Edit” on a source component, they “Unedit” will release your hold on the file in the Version Control system. If you have already made changes you can still select “Unedit” and Studio will rollback your changes as though you hadn’t changed it.



Section 13

The Common Catalog

The FuegoBPM Common Catalog is a powerful feature for sharing development resources. It is visible to all FuegoBPM Studio projects that share the same VCS source repository.

The Common Catalog is only available when using Studio's Version Control System (VCS).¹

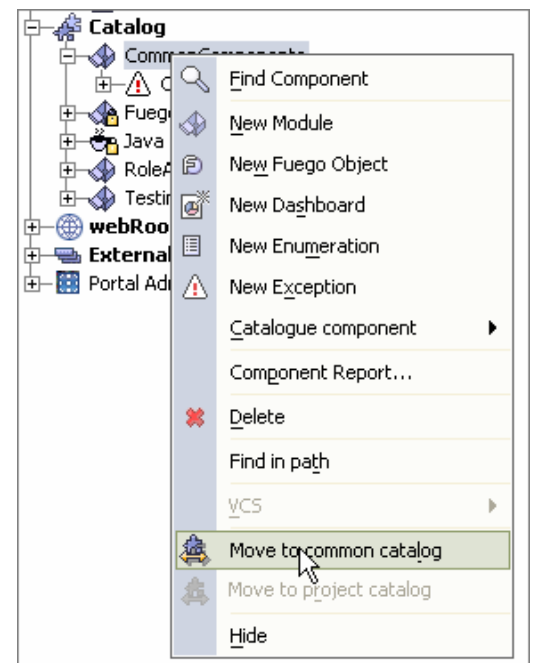
13.1 How A Catalog Component Becomes a Common Catalog Component

Any component in the Studio project can be moved to the Common Catalog. By right-clicking the Catalog module and selecting "Move to common catalog" the module will be "marked" as a common catalog component, however, it is not officially in the common catalog until the change is committed to the VCS repository.

The steps to move a Catalog component to the Common Catalog is actually too easy! Moving a component to Common Catalog could have a serious impact to all of the projects using the VCS repository.

13.2 The Common Catalog is visible to all

Once a Catalog component is moved to the Common Catalog and stored in the VCS repository it is seen by all projects stored in the repository.



¹ It is not practically feasible to implement outside of VCS due to the high-degree of potential risk to the Common Catalog's due to manual check-in or check-out errors.

13.2.1 A Project Team should own their Common Catalog component

As the Common Catalog is inherited by all projects that share the same VCS repository, the project that owns the component should request edit on the component and not relinquish control of it. This will ensure other project teams cannot alter a component that does not belong to them that may be in wide use in other projects.

13.2.2 Be Careful What Components Are Moved to the Common Catalog

Database, Messaging and other components that require External Resource definitions in Studio that are promoted to the Common Catalog will require that all projects in the VCS repository include external resource definitions in their project.

13.3 Standard Project and Common Catalog Directories

The VCS repository will prompt a new project stored in the repository for the name of the sub-directory where the project artifacts are to be stored. If a Common Catalog component is also defined in this new project then the name of the Common Catalog sub-directory will also be prompted.

13.3.1 Do Not Use Default Directory Names

When storing a project in the VCS repository for the first time, do not use the default directory names (the default names are shown in the example). This assures the project is stored in the repository in a different subdirectory than other projects.

The same is true for the Common Catalog location. Store your project's common catalog modules in a subdirectory different from others common catalog modules.

As a standard, consider appending a hyphenated Project Acronym to the directory names, “projects-fdi” and “common-fdi”.

