

## **OpenCDS Guvnor 5.4 and DSL Best Practices**

## Contents

|  |    |
|--|----|
| Change Log.....  | 4  |
| Introduction .....   | 5  |
| I. Pot-Holes.....  | 6  |
| A. Known “pot-holes” in Guvnor 5.4.....                        | 6  |
| B. <i>Work-Arounds for the Pot-Holes</i> .....                 | 8  |
| II. Best Practices for Writing DSLs .....                      | 13 |
| A. Plan the Logical Pieces of a Rule First .....               | 13 |
| B. Write and Test your DSLs.....                               | 14 |
| C. Build the Rules using DSLs.....                             | 18 |
| D. Export the Package as a KnowledgeModule for OpenCDS .....   | 18 |
| III. Common Patterns for OpenCDS Rules.....                    | 19 |
| A. Events in General .....                                     | 19 |
| B. Entities in General.....                                    | 20 |
| C. Encounters .....  | 21 |
| D. AdverseEvents.....  | 21 |
| E. Observations .....  | 22 |
| F. Problems .....  | 23 |
| G. Procedures .....  | 24 |
| H. SubstanceAdministrations – Medications .....                | 25 |
| I. SubstanceAdministrations – Immunizations .....              | 26 |
| J. Supplies.....   | 27 |
| IV. Best Practices for Creating Responses from OpenCDS .....   | 28 |
| A. Use Templates for Output .....                              | 28 |
| B. One Inference Wrapper per KnowledgeModule .....             | 28 |
| C. Include Both Coded Response and Text (if appropriate) ..... | 28 |
| D. Include Proposals (if appropriate) .....                    | 29 |

|    |  |    |
|----|--|----|
| E. | How to Create Return Messages from your Rules..... | 30 |
| V. | Common Patterns for Creating Responses .....       | 31 |
| A. | Write me... ..                                     | 31 |

## ***Change Log***

| Date       | Author        | Notes   |
|------------|---------------|---|
| 12-14-2011 | David Shields | Initial document without best practices section |
| 12-15-2011 | David Shields | Finished first draft of best practices section  |
| 12-20-2011 | Ken Kawamoto  | Minor edits                                     |
| 12-20-2011 | David Shields | Added screen shots that were omitted earlier    |
| 1-12-2012  | David Shields | Added one new “pot-hole” and work-around / fix  |
| 6-28-2012  | David Shields | Edited for Drools Guvnor v5.4                   |
| 7-5-2012   | David Shields | Added more “pot-holes”, worked on DSL Patterns  |

## ***Introduction***

The objective of this document is threefold:

- to list “pot-holes” (aka “bugs”, “flaws”, or “annoyances”) in Drools Guvnor 5.4,
- to furnish techniques to avoid these “pot-holes,” and
- to provide patterns and best practices for creating re-useable DSLs in Guvnor for use in writing clear and reviewable rules for OpenCDS.

Many of the “pot-holes” that were present in Drools Guvnor v5.3 have been fixed in this version, but some of the fixes have introduced a new “gotcha.” It is our hope that the pot-holes will eventually all be fixed in future versions of Guvnor, and that the “pot-hole” portion of this document will no longer be needed.

This document will also furnish some common patterns to accomplish certain tasks in building rules for OpenCDS in Guvnor.

## ***I. Pot-Holes***

### **A. Known “pot-holes” in Guvnor 5.4**

This list may not be comprehensive, but it represents the things that we bumped into repeatedly. If you have found other “pot-holes”, or have developed work-arounds, please either update this document with what you have learned, or let us know so that we can update this document.

#### **1. No Warning on Close without Save**

##### ***still an issue in 5.4***

All of the “Assets” that are part of a package can be opened for editing. They will all show a gray “X” on the right-hand side of the tab at the top of the screen when you hover your mouse over the tab. You need to think of this “X” as meaning “revert” or “abort,” because it will happily throw away all the work you have just done on the asset, with no warning.

#### **2. Inconsistent Refresh of Elements on Screen**

##### ***still an issue in 5.4, but improved***

There are not as many cases where the screen does not match what has been saved, but there is still at least one case where this happens. If you rename an asset such as a rule, it will remain on the screen with the former name, even though the new name has actually been saved.

#### **3. Possible to Add Invalid DSL to a Guided Rule**

##### ***no longer an issue in 5.4***

#### **4. Possible to Permanently Damage a Guided Rule**

##### ***no longer an issue in 5.4***

#### **5. Possible to put a Package into Inconsistent State**

##### ***no longer an issue in 5.4***

#### **6. Possible to Archive Global Area**

##### ***no longer an issue in 5.4***

It is possible to archive the Global Area. If you did this in 5.3, it destroyed the Global Area, permanently. In 5.4, it merely archives the contents of the Global Area, and the contents can be retrieved using the Administration tools, if necessary.

#### **7. Possible to Add an Asset to Multiple Packages**

##### ***still an issue in 5.4***

It is possible to create a new DSL or Enumeration (and possibly other types of assets) and have them become an unintended asset of more packages than the single package you specified when you created it. This sometimes happens when you have more than one package open as you create the element, and other times it happens for no apparent reason. For example, if you create a new Enumeration that you intend to be a part of a specific package, but you have the Global Area (and possibly other packages) open when you create the Enumeration, it will be added to all the other packages that are currently open.

## **8. Saving the Package and Signing Out**

### ***still an issue in 5.4 (was present in 5.3, but not documented)***

When you save a package and “Sign Out”, Guvnor will happily throw away all of any unsaved work you have done on any open assets within that package, with no warning.

## **9. Some Assets from Global Area Randomly Appear or Disappear from Active Package**

### ***still an issue in 5.4 (was present in 5.3, but not documented)***

This may be just a different view on Problem # 7. Although any DSL you have specifically copied from the Global Area to your current working package will remain an asset in that package, other DSLs from Global will randomly appear or disappear from the working package. While they are present, they can be used. If they don’t show at a particular point, they also cannot be added to a rule, and a build of the package will generate errors.

## **10. Using Techniques to “refresh” Lists or Views may Refresh Modified Assets Elsewhere**

### ***new issue in 5.4***

They resolved most of the issues around disparities between what is “saved” and what shows in a list of assets, or even the contents of an asset. However, this has introduced a new problem that is apparently a side-effect of the “refresh” fixes. A modified asset that has not been saved may get “refreshed” from disk if you do anything that causes a package list of assets to be refreshed (such as clicking on a button named “refresh list”).

## **11. Enumerations Do Not Allow Comments**

### ***new issue in 5.4***

In Guvnor 5.3 it was possible to imbed comments in Enumeration assets. In Guvnor 5.4, this will cause the enumerations to fail, except in one specific structure: You can add a block comment to the last line of the enumeration (e.g., `/* comment... */`). A comment in any other location will cause the enumeration to fail to be processed at all by Guvnor. Since OpenCDS has a tool to export enumerations which placed a comment at the beginning of the enumeration list, this will have to be removed, or your enumerations will not show up in DSLs.

## **B. *Work-Arounds for the Pot-Holes***

The following approaches can help you avoid the pot-holes, or recover from falling into them. Of course, it is always better to avoid them in the first place... ☺

### **1. Develop Habit of Save before Changing to a Different Tab EVERY TIME**

Before you click on another tab, use the File menu, and select “Save changes.” Some things you do in another tab can cause the screen to be refreshed from the last saved value, thereby losing all your updates in any open tabs.

### **2. Develop Habit of Save before Close EVERY TIME**

Instead of using that “X” to close the asset, use the File menu, and select “Save and Close.” Developing that habit will avoid a lot of lost work.

There are certain cases where a Save and Close does not work, such as when you have just imported a new version of your data Model. In that particular instance it is best to close the screen for uploading a new model (after you have clicked the “Upload” button), validate and build the package, and chose “Save” from the File menu for the entire package.

If there are errors shown at this point (and you didn’t have errors before importing the new model), the most common problem is that your imported model is missing items (or has renamed them) that were in the previous model. Guvnor does not regenerate your “Configuration: Imported types” and “globals” when you import a new model. The most it will do is add new types and globals to their respective lists. Items you dropped from the model will still show in the list unless you remove them.

If you are working with complicated models that contain a large number of types, you may find it useful to click the “advanced view” of the configuration, then copy the contents and paste them into an external text file. Once you have the external text file, sort it, and save it with your project.

This makes it much easier to find the import statements that have been renamed or deprecated, and to remove them. Guvnor will happily allow you to paste the sorted list into the “advanced configuration”, and will keep it in the order you created. Any renamed or new types will be added to the end of the list after you import an updated model, and you can then cut and paste them into sorted locations in the list as well.

NOTE: We maintain a text copy of all the import statements for vMR version 1.0 in the OpenCDS Maven project, as a text resource in the vMR internal module. In most cases, you can probably use that list without changes.

### **3. Develop Habit of Exporting the Guvnor Repository FREQUENTLY**

Because it is possible to get Guvnor packages and rules into inconsistent or “locked” and unopenable states, it is critical to export the package to create backups. You will avoid a lot of grief if you make it a habit to always do an export frequently and especially **before** you do any of the following:

- Rename anything
- Remove a DSL or an Enumeration (no longer as big an issue as it was in 5.3, but still a good idea)
- Import an updated Model



## 4. Techniques to Refresh Screen

There are three common techniques to refresh the screen. These were more useful in 5.3 than they are in 5.4, which is a lot better about refreshing the screen. Note that all of these techniques may refresh assets that have been changed, thereby losing any changes you have not saved. ***Save First!***

- Click the “Refresh List” button in the Assets view of the package. This will often make an asset that you just created appear in the list, or an asset that you just deleted/archived disappear from the list.
- Click the icons which change the view of Knowledge Base Packages from nested to listed. This will close the group of package listings and reopen them, and tends to refresh all the lists in all packages.
- When all else fails, and you know that you have something saved in the package, but it doesn’t appear appropriately (or the contents seem to be from the previous version), you can refresh everything, including the element lists that show when you are adding things to a Guided Rule, by logging out and logging back in.
- NOTE: All of the above methods may refresh assets that are currently open (and possibly have changes!). ***Save all your changes first!***

## 5. Avoid Damaging a Guided Rule

Guvnor 5.4 no longer appears to have a problem with “damaged rules,” but the following suggestions are still probably good ones...

The primary method I found in Guvnor 5.3 to damage a Guided Rule was to change a dependency, such as renaming or removing an Enumeration that is referenced in a DSL, and then adding the DSL to the Guided Rule. You could also damage a rule by making changes to a DSL without testing them, and updating the rule with the broken changes.

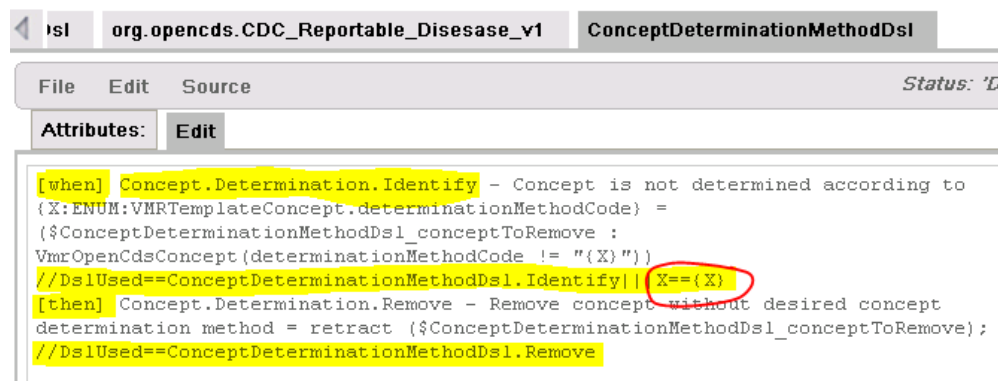
You can avoid ever having damaged rules by following this advice:

- Never rename a DSL. If you want it renamed, create a new DSL with the new name, and copy and paste the content of the original rule into the new rule. Once you have tested that new DSL, you can then go through **all** of the rules that used the old DSL, and delete the references to the old DSL, and add in the new DSL. Once you have completed the changes, then you can delete the old DSL.
- Never rename an Enumeration. If you want it renamed, create a new one with the new name. Go through **all** the DSLs that reference that Enumeration, and correct them to the new name. Once you have completed the changes, then you can delete the old Enumeration.
- Always create your rules in a package. This allows you to use recovery techniques if you damage the rule or package. In fact, I have personally decided that it is safer never to have a Guided Rule in the Global Area, although I use the Global Area for all DSLs, and for common enumerations.
- Always test your DSL files in throw-away rules before you start using the DSL in a complex rule. Throw-away rules are guided rules that you never save. You add DSL elements to them, and you check to see if the rule validates. You can even add a block for directly entering LHS and RHS rule content, which is

useful for testing DSLs. If the validation fails, you can always delete the rule, fix the problem, and try again.

- Always select the dropdowns and populate the variables in a DSL that you have added to a rule before you save the rule. If you don't resolve the macro substitution values, the rule will not generate successfully, and you may end up with a damaged rule.
- Always Validate every rule before you save it. If you have added a bad DSL to the rule, and it caused the display of a part of the rule to disappear, you are going to have a problem if you save the rule. Simply close the rule using the "X" on the tab (or click on File : Delete if the rule is new and has never been saved), and the bad stuff you added will just go away. Once the rule is saved with the bad stuff in it, it may be permanently damaged.

TIP: Put a comment at the end of your DSLs which includes the name of the DSL, and references all of the Enumerations in the DSL. Then, in the event of a problem, looking at a Source | View Source on a Guided Rule which has this comment imbedded in the source code, will show where you have a bad DSL or a missing Enumeration. For Example:



```
[when] Concept.Determination.Identify - Concept is not determined according to
{X:ENUM:VMRTemplateConcept.determinationMethodCode} =
({ConceptDeterminationMethodDsl_conceptToRemove :
VmrOpenCdsConcept(determinationMethodCode != "{X}")})
//DslUsed==ConceptDeterminationMethodDsl.Identify|| {X}={X}
[then] Concept.Determination.Remove - Remove concept without desired concept
determination method = retract ({ConceptDeterminationMethodDsl_conceptToRemove});
//DslUsed==ConceptDeterminationMethodDsl.Remove
```

Note that the two comments highlighted above (one at the end of the [when] clause, and the other at the end of the [then] clause) will show in the generated source code, and the macro replacement of the DSL will include the selected drop-down value of the Enumeration named "VMRTemplateConcept.determinationMethodCode". If the displayed source simply shows X=={X}, then the enumeration was not selected, or there was no Enumeration list by that name.

TIP: You can look for Enumerations all in one place by opening the package and clicking on the Edit tab. Then select the URL for package source, and save the source to an editor such as Notepad++. You can then use the search facilities in the editor to look for the Enumeration names. If you followed the previous TIP about putting a comment in the DSL with the name of the DSL, you will know exactly which DSL is referencing the Enumeration.

## 6. Recovering from a Single Damaged Rule

This information no longer seems to be pertinent in Guvnor 5.4, but is kept here "just in case."

If the rule is damaged, but you can open it, you can do the following:

- Go to Source | View Source and copy the contents of the generated rule. *NOTE* that any DSL generated information in the rule will have resolved the macro elements to the extent possible. If some of the macro elements were not resolved, look for missing or renamed enumerations.

If you forgot to check the Validity of a rule before you saved it, and you can't open it to fix it or delete/archive it, you may (or may not) first want to try to recover the contents of the rule by doing one of the following.

- Select the package and click the Edit tab to show the configuration information. Click on the "URL for package source" and save the source to a file or an editor such as Notepad++. This source file will contain all the "DRL" source code in the package, including import statements, functions, and all of the rules. However, it does not contain the source of the DSL files. We have never had the DSL files lock and fail to open, so this is not normally a problem.
- If you have installed Drools in Eclipse and/or you are comfortable changing source code with a text editor, you can try to fix the problem by manually updating the rule source code. Access the source code file by connecting to Guvnor's working version of the repository using webdav (<http://localhost:8080/drools-guvnor/org.drools.guvnor.Guvnor/webdav/> -- substitute the hostname and port of your Guvnor installation, and add the package and filename that you need to mess with, e.g.: [globalarea/AssertionDsl.dsl](#) ).

If you were unable to fix the rule by this point, you then have two options to get rid of the damaged rule:

- Archive and Restore the package. Select the package, and select File | Archive. Go to Administration | Archives and delete the damaged rule. Restore the entire remaining package. Go to the package and recreate the rule, fixing the bad DSL or Enumeration before you save the new rule.
- The Nuclear option: import a backup of the repository. Go to Administration : Import – Export, and import your most recent backup (Guvnor is nice enough to number them). *NOTE*: this will destroy all the work you have done since you exported the backup. You did backup frequently, didn't you? ☺

## 7. Recovering from an Inconsistent Package

I haven't seen this happen in Guvnor 5.4. But if it does, here are some things to try.

In some cases you can recover by doing the Archive and Restore technique described above. This is the most desirable approach, because it doesn't lose any of your work.

In most other cases you can recover from one damaged package among a long list of packages by archiving and deleting the damaged package, and then recreating it from scratch. You may want to use the technique described above to save the source code for the package first, because once you delete the package, you will lose everything in it.

If you know what asset is damaged within the package, you can try to fix the problem by manually updating the asset using a text editor or Eclipse. Access the source code file by connecting to Guvnor's working version of the repository using webdav (<http://localhost:8080/drools-guvnor/org.drools.guvnor.Guvnor/webdav/> -- substitute the hostname and port of your Guvnor installation, and add the package and filename that you need to mess with, e.g.: [globalarea/AssertionDsl.dsl](#) ).

In the worst cases, your only choice is the nuclear option. This will lose all of the work you have done in all packages since you last made a useable backup. Which option you choose will partly depend on how recently you created a backup. Having a recent backup is always a good thing... ☺

## **8. Recovering from Archived Global Area**

It was impossible to recover from an archived global area in Guvnor 5.3, but is relatively simple to do in Guvnor 5.4. Unlike Guvnor 5.3, Guvnor 5.4 archives the assets in the Global Area, but not the Global Area itself. Simply go to Administration | Archive, and restore the assets.

## **9. Recovering from Assets that Appear in Wrong Package**

This will still sometimes happen in Guvnor 5.4, and if it does, here is what I found that sometimes would resolve it in Guvnor 5.3:

If you have created an Enumeration, for example, that has somehow appeared in other packages where it doesn't belong, one fix is to archive and delete the Asset, close all packages, and then recreate the Asset in the correct package.

Another technique to fix the problem is to manually update the asset using a text editor or Eclipse. Access the source code file by connecting to Guvnor's working version of the repository using webdav (<http://localhost:8080/drools-guvnor/org.drools.guvnor.Guvnor/webdav/> -- substitute the hostname and port of your Guvnor installation, and add the package and filename that you need to mess with, e.g.: [globalarea/AssertionDsl.dsl](#) ).

## **10. Recovering from Unintended Refresh**

Unsaved changes are gone. There is no recovery.

## **11. Recovering from Non-Functioning Enumerations**

Remove all comments from the Enumeration file.

## ***II. Best Practices for Writing DSLs***

The following suggestions represent the approach that we have learned to use in writing Guided Rules using DSLs for Guvnor. Although we write most of our rules in Guvnor, the same general steps probably apply to writing technical rules, and developing rules in Eclipse. Please add (or let us know about) any further suggestions or comments that you come up with as you work with OpenCDS and Guvnor DSLs.

### **A. Plan the Logical Pieces of a Rule First**

Use a whiteboard, or Notepad, or scratch paper to think about the different logical pieces that you need to complete the rule. It may be helpful to build a flowchart. If you plan to use Drools Flow / BPMN2 / Oryx for your rules, then you can use this tool to build the flowchart. Stay in this process until you can isolate elements that you need in the LHS of a rule, and elements that you need in the RHS of a rule. You should probably stay in this process until you have a first-cut idea of how many rules you need, and what each rule should do.

#### **1. Think in Terms of Concepts, rather than Data**

In all stages of this process you should be thinking in terms of “concepts” about the data, and not in terms of specific values or codes within the data. This means that you think about the disease, problem, medication as a concept, such as “diabetes mellitus, asthma, hypertension, ACE inhibitor, NSAID, specified age group, etc.

#### **2. Develop the Structure to Map Concepts to Data**

This process is described in a separate document named “Using OpenCDS Concepts.doc.” You will need to begin the process of creating concept instances before you start writing DSLs. These concept instances that you identify will have a name in an Enumeration that you will need to create before you can write your DSLs.

Think of OpenCDS Concepts as the interface between the way a Clinician thinks about what to do, and the ugly world of data values that are overlapping, coarse-grained or fine-grained, and sometimes even proprietary to a single institution.

You will not normally need to worry about the actual mapping of data values to the concept instances until you are ready to deploy the rules in OpenCDS and do the final integration testing.

#### **3. Create Enumerations in Guvnor**

You will need to have defined and named “Concept instances” that are in your Enumerations before you start writing DSLs.

We have found it useful to put most of our concept instances in a global enumerations list, kept in the Global Area. These are created by the OpenCDS tool named “GuvnorEnumerationCreator” and pasted into Guvnor Enumerations as plain text. Refer to the “Using OpenCDS Concepts.doc” and JBoss Drools documentation for guidance in this process.

These enumerations are the “concept instances” against which you will write your DSLs and your rules.

## B. Write and Test your DSLs

Before you start writing your final rules, develop re-useable Domain Specific Language (DSL) elements for as much of them as possible.

These DSLs should be written in terms of “concepts”, so that they are abstracting actual data and code values. For example, write your rules against a concept of “asthma”, instead of writing your rules against an ICD9 or SNOMED code that represents asthma. OpenCDS is designed to support the flexible mapping of codes at run-time to the logical concepts that you use to write the rules.

### 1. Start a DSL as a SandBox rule

A single named DSL can have zero to many [when] clauses (aka the left-hand-side or LHS of a rule), and zero to many [then] clauses (aka the right-hand-side or RHS of a rule). The purpose of the multiple elements is that each individual element is optional, and can be individually chosen or rejected when you build the rules. Until you are adept at creating DSLs, it is probably a good idea to keep your DSL simple, and avoid multiple elements.

Create a “sandbox rule” by creating a new Guided Rule in Guvnor, and name it “Sandbox” or some other throw-away name, because you are **not** going to keep this rule. Once the rule appears in your editor, add elements that you want your DSL to perform, and validate (but do NOT save) what you have added.


Repeat this process in the sandbox rule until it both validates, and includes the logical element of a rule that you want to turn into a DSL. Here is an example of a sandbox rule with the logic that you might want to turn into a DSL:

The screenshot shows the Guvnor rule editor interface. At the top, there's a tab labeled "sandbox". Below it, a menu bar contains "File", "Edit", and "Source". Under the "Edit" menu, there are two sub-tabs: "Attributes:" and "Edit". The main area is titled "WHEN" and contains two numbered conditions:

1. There is an EncounterTypeConcept [**\$encTypeConcept**] with:
  - openCdsConceptCode equal to Outpatient encounter
2. There is an EncounterEvent [**\$encEvent**] with:
  - id equal to \$encTypeConcept.conceptTargetId. Choose...
  - subjectIsFocalPerson equal to true
  - encounterEventTime --- please choose ---
  - IVLDate.high less than or equal to evalTime. Choose...

Go to the Source | View Source tab, and view the text of the generated sandbox rule. This text is going to become your DSL:

Viewing source for: sandbox



Viewing source for: sandbox

```

1. | rule "sandbox"
2. |   dialect "mvel"
3. |   when
4. |     $encTypeConcept : EncounterTypeConcept( openCdsConceptCode == "C44" )
5. |     $encEvent : EncounterEvent( id == $encTypeConcept.conceptTargetId ,
6. |     subjectIsFocalPerson == true , encounterEventTime.high <= evalTime )
7. |   then
8. | end

```

Create a new DSL, and copy and paste the text from the [when] clause of the generated sandbox rule into the new DSL. Since there are two lines in the [when] clause, and Drools implies an “and” connection between separate lines in the LHS of a rule, we will need to insert an “and” between the two lines, and put them all on one line. We will also parenthesize the entire thing.

Our first effort might look like this (and it is all on one logical line):

```

($encTypeConcept : EncounterTypeConcept( openCdsConceptCode == "C44" ) and $encEvent :
EncounterEvent( id == $encTypeConcept.conceptTargetId, subjectIsFocalPerson == true,
encounterEventTime.high < evalTime ))

```

If we then clean up the variable names (the elements beginning with “\$”) to make them globally unique in a long and complex rule with many DSLs, this text might look like this (and it is still all on one line):

```

($PatientEncounterEventDsl_encounterConcept_OutpatientEncounter :
EncounterTypeConcept(openCdsConceptCode == "C44") and EncounterEvent(id ==
$PatientEncounterEventDsl_encounterConcept_OutpatientEncounter.conceptTargetId,
subjectIsFocalPerson == true, encounterEventTime.getHigh()< $evalTime ))

```

This is now a working DSL, but it only knows how to do the one thing you hard-coded into it. You are going to do several more steps to turn this text into a very useful and re-useable DSL:

1. Write a clear plain-language statement of what the DSL is supposed to describe, follow it by an equal-sign (“=”) and place it in front of the text you copied from the sandbox rule. This might be something like

```
Patient has previously had an outpatient encounter =
```

followed by the text of the rule that you wrote to implement this, as shown above.

2. Abstract out the concept instances into enumerations that you identify by {X} – any variable name surrounded by curly braces, and with some additional information that will be demonstrated in examples below.

Also abstract out any quantities you might want to change into variables that you identify by {n} – any variable name surrounded by curly braces.

The left-hand side of the DSL has then become something like

```
Patient has previously had a {X:ENUM:EncounterTypeConcept.openCdsConceptCode} =
```

3. Replace the specific concept on the right-hand side of the = with the variable name[s] in curly braces, and you get this for the entire DSL:

```
Patient has previously had a {X:ENUM:EncounterTypeConcept.openCdsConceptCode} =  
($PatientEncounterEventDsl_encounterConcept_{X} :  
EncounterTypeConcept(openCdsConceptCode == "{X}") and EncounterEvent(id ==  
$PatientEncounterEventDsl_encounterConcept_{X}.conceptTargetId,  
subjectIsFocalPerson == true, encounterEventTime.getHigh() < $evalTime ))
```

4. Finally clean it up by adding a list name at the beginning, and a comment at the end.

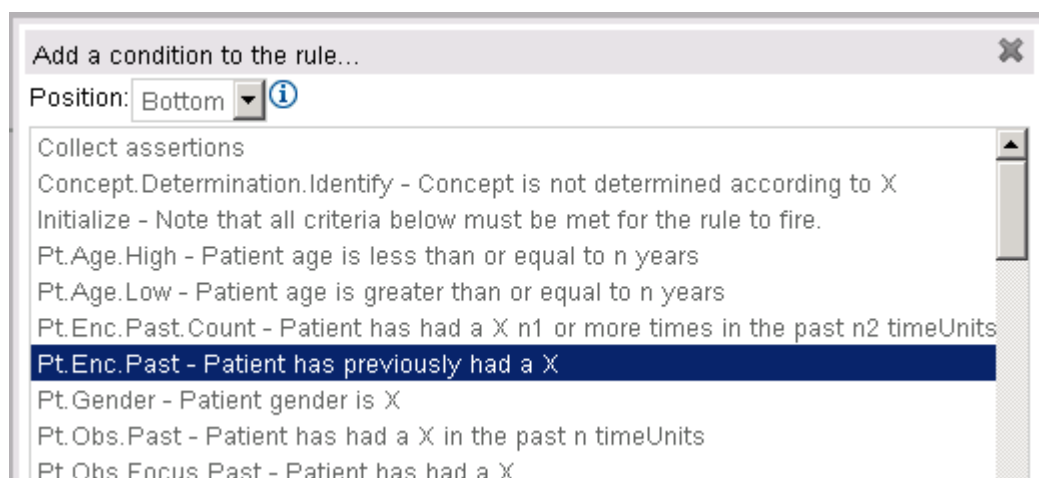
The list name is the first thing you will see in the Guvnor list of elements that you can add to a Guided Rule, and helps you choose the right rule. It does not become a part of the final source code for the rule.

The comment at the end of the DSL helps us identify the DSL when it is one among many DSLs contained in a single rule. We include the name of the DSL, and show the selected values for all the variables that are being substituted into the rule. This step is invaluable for debugging.

The final DSL named “PatientEncounterEventDsl” contains exactly the following (with no carriage returns in it anywhere – it is all on one logical line):

```
[when] Pt.Enc.Past - Patient has previously had a  
{X:ENUM:EncounterTypeConcept.openCdsConceptCode} =  
($PatientEncounterEventDsl_encounterConcept_{X} :  
EncounterTypeConcept(openCdsConceptCode == "{X}") and EncounterEvent(id ==  
$PatientEncounterEventDsl_encounterConcept_{X}.conceptTargetId, subjectIsFocalPerson  
== true, encounterEventTime.getHigh() < $evalTime ))  
//DslUsed==PatientEncounterEventDsl||X=={X}
```

If we click the plus sign on our sandbox rule to add this DSL to the sandbox rule, the drop-down list that Guvnor gives us to select from shows the rule like this, which makes it clear exactly what rule we are selecting:



Once we add it to the sandbox rule, and select the option we want from the drop-down list, the rule will look like this:



sandbox

File
Edit
Source

Status: 'Draft'

Attributes:
Edit

**WHEN**

1. Pt.Enc.Past - Patient has previously had a Outpatient encounter

**THEN**

(show options...)

+
+
+
+

If we look at the Source | view Source, the source will look like this (and you can see the comment at the end of the DSL has been populated with the selected concept instance ID for “Outpatient encounter” that we selected in the drop-down):

Viewing source for: sandbox
✕

Viewing source for: sandbox

```

1. | rule "sandbox"
2. |   dialect "mvel"
3. |   when
      ($PatientEncounterEventDsl_encounterConcept_C44 :
      EncounterTypeConcept(openCdsConceptCode == "C44") and EncounterEvent(id ==
4. | $PatientEncounterEventDsl_encounterConcept_C44.conceptTargetId, subjectIsFocalPerson ==
      true, encounterEventTime.getHigh()<= $evalTime ))
      //DslUsed==PatientEncounterEventDsl|||X==C44
5. |   then
6. | end
          
```

## 2. Study the Sample Rules Provided

They include examples of more complicated rules that combine multiple concepts and settable values in useful and re-useable ways.

## 3. Test your Sandbox Rule with just One DSL in it

Make sure your sandbox rule validates, and save it.

Create a Test Scenario to test it. Follow the examples in the supplied samples for ideas on how to do this. The Test Scenario will tell you whether or not the rule “Fires”, even if you don’t have anything in the RHS of the rule yet.

Test each DSL you create in this same way before you start putting them in a rule you want to keep.

## 4. Use Meaningful Names for Variables

You may have noticed that the example rule we built above has variable names that can be quite long. The variable “\$PatientEncounterEventDsl\_encounterConcept\_{X}” would work in a simple rule if it was only “\$x1”.

However, by combining the name of the DSL with the substituted concept, we have a variable name that can be referenced elsewhere in the rule containing the DSL, and it should be perfectly clear what the variable is referring to by its name. Also, using unique names ensure that duplicate variable names will not be declared within the same rule (which may include many DSLs), since such duplicate declarations will cause compile errors.

## C. Build the Rules using DSLs

As you can see from the Sample Guided Rules we included, you can often build a quite complex rule from an assortment of generic DSLs.

Write Test Scenarios for each rule, and test all of the following thoroughly:

- Absence of expected data
- Unexpected data values
- Boundary conditions for ranges, including dates, times, and counts

Remember that you can create a copy of a Test Scenario and modify or add to it to create another Test Scenario. You don't have to create each test from scratch. The goal is to achieve 100% testing coverage of all aspects of every rule.

## D. Export the Package as a KnowledgeModule for OpenCDS

You have two choices: You can either export the compiled binary package as a \*.pkg file, or you can export the source code for the package as a \*.drl file. The **PKG** file will run without a lag-time the first time it is called, but is completely opaque, will only run on the version of Drools that it was built in, and you can't modify it. The **DRL** file on the other hand can be modified with a text editor while it is sitting in a running OpenCDS Knowledge Repository, and that can be very useful for debugging.

Note that a **PKG** must be used when a Guvnor package includes processes in addition to rules. If processes are used, OpenCDS expects you to configure the setup file that **knowledgeModules.xml** with the name you assigned to the primary process.

Note also that when you use a **PKG** in OpenCDS, you must include the classPath of the correct version of the DroolsAdapter in OpenCDS as an attribute of the correct entry in the **knowledgeModules.xml** configuration file.

### ***III. Common Patterns for OpenCDS Rules***

In building OpenCDS, we carefully created a structure to separate generalized “clinical concepts” that a clinical domain expert would use to describe the “logic” of the rule from the actual “concept descriptor” codes used in the data. This makes it possible to keep the logic of the rules separated from the mappings between the specific concepts used in the actual data, and the generalized concepts used in the rules.

It is useful to keep the logic and the mappings separate, because they tend to need to be updated on different schedules, and sometimes even by different experts (clinical domain experts versus terminology experts).

One component of the technology we use in writing rules are Domain Specific Languages (DSLs). DSLs are created by programmers to provide a way for the clinical domain experts to express the generalized clinical concepts in language they are familiar with, and preferably in the same language they would use to describe what the rule should accomplish.

OpenCDS furnishes a number of sample DSLs, and we add more frequently.

Study the DSLs in the Guvnor Binary download, along with the suggestions below. These suggestions will help you understand why the rule is written the way it is written, and how we separate the technical details of Drools from the clinical descriptions of what the rule should do.

Our goal in writing the DSLs has been to: 1.) make the final rule perfectly understandable by a clinician, and 2.) produce the correct results. It may be possible to do things differently, and to improve on either or both of those aims. We welcome feedback and suggestions in that regard.

We also welcome and encourage additional patterns to add to the following list. Please note that the suggested prefix, terminating comment, and robust naming that we describe above is omitted for conciseness in these patterns. You should always add them to your final DSL to make the DSL re-useable.

#### **A. Events in General**

##### **1. [Event] has happened**

```
[when] Patient has previously had a
{X:ENUM:<conceptType>.openCdsConceptCode} =
($x: <conceptType>(openCdsConceptCode == "{X}")
and <clinicalStatement>(id == $x.conceptTargetId, subjectIsFocalPerson ==
true, <clinicalStatementEventTime>.high <= $evalTime ))
```

Definitions of elements in the above pattern:

\$x should be expanded to a meaningful name,

<conceptType> represents the name of an OpenCDS Concept Type, such as “EncounterTypeConcept”

<clinicalStatement> represents the name of an OpenCDS vMR clinical statement Type, such as “EncounterEvent”

<clinicalStatementEventTime> represents the name of the relevant event time element in the particular clinical statement Type.

2. [Event] has happened at least {n} times
3. [Event] has happened in last {n} [timeUnits]
4. [Event] has happened at least {n1} times in last {n2} [timeUnits]
5. [Event] happened between {n1} and {n2} [timeUnits] ago
6. [Event] as tag {idTag}...

then uses all patterns shown above under Events in General

7. [Entity] has role {targetRole} to {clinicalStatementClass} identified as tag {idTag}
8. [Entity] has role {targetRole} to {clinicalStatementType} identified as tag {idTag} during {relationshipTimeInterval}...
9. [Event] is {targetRelationshipToSource} of {clinicalStatementClass} identified as tag {idTag}...

then uses all patterns shown above under Events in General

## B. Entities in General

1. [AdministrableSubstance] {substanceCode} [of type {entityType} ][strength {strength} ][form {form} ][brand {substanceBrandCode} ][generic {substanceGenericCode} ][mfg {manufacturer} ][lot# {lotNo} ]

Only the substanceCode is strictly required, but some use cases may require some or all of the additional elements. More specific patterns will be developed as the need arises.

2. [Entity] [of type {entityType}]

**3. [Facility] [of type {entityType}]**

**4. [Organization] [of type {entityType}]**

**5. [Person] [of type {entityType}]**

**6. [Specimen] [of type {entityType}]**

**7. [Entity] as tag {idTag}...**

then uses all patterns shown above under Entities in General

**8. [Entity] has role {targetRole} to {entityClass} identified as tag {idTag}**

**9. [Entity] has role {targetRole} to {entityClass} identified as tag {idTag} during {relationshipTimeInterval}...**

## **C. Encounters**

**1. [Encounter] as tag {x}...**

then uses all patterns shown above under Events in General

**2. [Encounter] is {targetRelationshipToSource} of {clinicalStatementClass} identified as tag {idTag}...**

then uses all patterns shown above under Encounters and Events in General

## **D. AdverseEvents**

**1. [AdverseEvent] due to {adverseEventAgent}...**

then uses all patterns shown above under Events in General

**2. [AdverseEvent] due to {adverseEventAgent} with {criticality} criticality...**

then uses all patterns shown above under Events in General

**3. [AdverseEvent] due to {adverseEventAgent} with {severity} severity...**

then uses all patterns shown above under Events in General

**4. [AdverseEvent] due to {adverseEventAgent} with {criticality} criticality and {severity} severity...**

then uses all patterns shown above under Events in General

**5. [AdverseEvent] as tag {idTag}...**

then uses all patterns shown above under Adverse Events and Events in General

**6. [AdverseEvent] is {targetRelationshipToSource} of {clinicalStatementClass} identified as tag {idTag}...**

then uses all patterns shown above under Adverse Events and Events in General

**E. Observations**

**1. [ObservationResult] has {observationValue} (where observationValue may be of any supported datatype)**

**2. [ObservationResult] is interpreted as {interpretation}**

**3. [ObservationResult] has {observationValue} interpreted as {interpretation}**

**4. [ObservationResult] has {observationValue} using specimen from {targetBodySite}**

5. **[ObservationResult]** using specimen from **{targetBodySite}** is interpreted as **{interpretation}**

6. **[ObservationResult]** has **{observationValue}** using specimen from **{targetBodySite}** interpreted as **{interpretation}**

7. **[ObservationResult]** as tag **{idTag}**...

then uses all patterns shown above under Observations and Events in General

8. **[ObservationResult]** is **{targetRelationshipToSource}** of **{clinicalStatementClass}** identified as tag **{idTag}**...

then uses all patterns shown above under Observations and Events in General

## **F. Problems**

1. **[Problem]** was present on admission to encounter **{x}**

2. **[Problem]** was present on admission as **{priority}** to encounter **{x}**

3. **[Problem]** was a discharge diagnosis for encounter **{x}**

4. **[Problem]** was **{priority}** discharge diagnosis for encounter **{x}**

5. **[Problem]** is currently **{problemStatus}** and **{importance}**

6. **[Problem]** was **{problemStatus}** at least **{n}** times

7. **[Problem]** was **{problemStatus}** in last **{n}** **[timeUnits]**

**8. [Problem] was {problemStatus} at least {n1} times in last {n2} [timeUnits]**

**9. [Problem] was {problemStatus} between {n1} and {n2} [timeUnits] ago**

**10.[Problem] as tag {idTag}...**

then uses all patterns shown above under Problems and Events in General

**11.[Problem] is {targetRelationshipToSource} of {clinicalStatementClass} identified as tag {idTag}...**

then uses all patterns shown above under Problems and Events in General

## **G. Procedures**

**1. [Procedure] on {targetBodySite}...**

then uses all patterns shown above under Events in General

**2. [Procedure] by way of {approachBodySite}...**

then uses all patterns shown above under Events in General

**3. [Procedure] on {targetBodySite} by way of {approachBodySite}...**

then uses all patterns shown above under Events in General

**4. [Procedure] as tag {idTag}...**

then uses all patterns shown above under Procedures and Events in General

**5. [Procedure] is {targetRelationshipToSource} of {clinicalStatementClass} identified as tag {idTag}...**

then uses all patterns shown above under Procedures and Events in General



## H. SubstanceAdministrations – Medications

Not all possible patterns are shown. The ones shown cover the use cases listed below. It is expected that some elements in all patterns may not be needed in particular constrained use cases, and that more specific patterns will be created to meet this need.

### 1. Current medication list

- a. *[Medication] {substanceCode} since {administrationTimeInterval} [reported {documentationTime} ][by {dataSourceType} ][attested by {informationAttestationType} ]*

### 2. Medication order (prescription)

- a. *[Medication] {substanceCode} [strength {strength} ][form {form} ][dose {doseQuantity} ][route {deliveryRoute} ][frequency {dosingPeriod} [interval matters {dosingPeriodIntervalsImportant} ]][sig {dosingSig} ][# fills {numberFillsAllowed} ][{criticality} ][ordered on {orderEventTime} ]*

### 3. Medication administration (in clinic or hospital)

- a. *[Medication] {substanceCode} [type {doseType} ][strength {strength} ][form {form} ][dose {doseQuantity} ][route {deliveryRoute} ][site {targetBodySite} ][frequency {dosingPeriod} [interval matters {dosingPeriodIntervalsImportant} ]] from {administrationTimeInterval.low} [thru {administrationTimeInterval.high} ]*

### 4. Medication dispensation by clinic or pharmacy

- a. *[Medication] {substanceCode} [type {doseType} ] [strength {strength} ][form {form} ][dose {doseQuantity} ][route {deliveryRoute} ][frequency {dosingPeriod} [interval matters {dosingPeriodIntervalsImportant} ]][ not to exceed {doseRestriction} ][days supply {daysSupply} ][# {dispensationQuantity} ][ fill# {fillNumber} ][fills left {fillsRemaining} ][dispensed {dispensationTime} ]*

### 5. [Medication] ...

then uses many patterns shown above under Events in General

## 6. [Medication] as tag {idTag}...

then uses many patterns shown above under Medications and Events in General

## 7. [Medication] is {targetRelationshipToSource} of {clinicalStatementClass} identified as tag {idTag}...

then uses many patterns shown above under Medications and Events in General

## I. SubstanceAdministrations – Immunizations

Not all possible patterns are shown. The ones shown cover the use cases listed below. It is expected that some elements in all patterns may not be needed in particular constrained use cases, and that more specific patterns will be created to meet this need.

### 1. Immunization history

- a. *[Immunization] {substanceCode} [mfg {manufacturer} ][lot# {lotNo} ][dose# {doseNumber} ][route {deliveryRoute} ][site {targetBodySite} ][valid? {isValid} ] on {administrationTimeInterval.low}*

### 2. Immunization administration

- a. *[Immunization] {substanceCode} [mfg {manufacturer} ][lot# {lotNo} ][dose# {doseNumber} ][route {deliveryRoute} ][site {targetBodySite} ] on {administrationTimeInterval.low}*

### 3. Immunization forecast

- a. *[Immunization] {substanceCode} [dose# {doseNumber} ] not before {administrationTimeInterval.low} ideally on {administrationTimeInterval.high}*

### 4. [Immunization]...

then uses many patterns shown above under Events in General

### 5. [Immunization] as tag {idTag}...

then uses many patterns shown above under Immunizations and Events in General

**6. [Immunization] is {targetRelationshipToSource} of {clinicalStatementClass} identified as tag {idTag}...**

then uses many patterns shown above under Immunizations and Events in General

**J. Supplies**

**1. [Supply] on {targetBodySite}...**

then uses all patterns shown above under Events in General

**2. [Supply] as tag {idTag}...**

then uses all patterns shown above under Supplies and Events in General

**3. [Supply] is {targetRelationshipToSource} of {clinicalStatementClass} identified as tag {idTag}...**

then uses all patterns shown above under Supplies and Events in General

## ***IV. Best Practices for Creating Responses from OpenCDS***

### **A. Use Templates for Output**

Check the OpenCDS website for output templates that are already defined, and select an appropriate one. Use the `templateId(s)` assigned to that template in your output response message. This furnishes information to external software to define how your output is structured.

If an appropriate output template does not already exist, create a definition of one or more response message(s) that will suit the needs of your KM(s), and ask OpenCDS to assign new `templateId` values. This will be done quickly, and will be posted with the OpenCDS documentation so that it can be used by others. Write your definition using the *TemplateId Request Form* available on the OpenCDS website.

### **B. One Inference Wrapper per KnowledgeModule**

Create an `ObservationResult` as a “wrapper” for all of the response messages from each `KnowledgeModule` (aka Drools “Package”, or a “ruleset”) that OpenCDS was requested to apply to the current dataset. Use one of the DSLs whose name begins with “`OutputRoot...`” as this wrapper.

Imbed any additional `ObservationResults` and other clinical statements within this wrapper by adding a `ClinicalStatementRelationship` that relates the statement you wish to imbed to the wrapper statement. This puts the responses in the proper context when you are running multiple rules against a single dataset. Sample DSLs to imbed additional clinical statements have names that begin “`OutputNested...`”.

You are not limited to imbedding just `ObservationResults` in a wrapper. You can imbed any clinical statement(s) in whatever order is useful and meaningful. You can imbed all or portions of the submitted data by creating a `ClinicalStatementRelationship` to the wrapper element, in the same way that the “`OutputNested...`” DSLs do it.

### **C. Include Both Coded Response and Text (if appropriate)**

Note that you can return both “typed” values and text values in an `ObservationResult`, and it is sometimes useful to include both a code, and explanatory text designed for humans (for validation and debugging purposes, even if the output is intended solely for machine processing). For example, you might return something like the “`originalText`” in the following examples to express the full contextual meaning of the `code/codeSystem` returned in an `observationValue`:

```
<observationValue>
  <concept code="..." codeSystem="..." originalText="Patient is overdue for an Hba1C test."/>
</observationValue>
```

```
<observationValue>
  <physicalQuantity value="3.14" units="ml"/>
</observationValue>
<interpretation code="..." codeSystem="..." originalText="Extremely high value, further treatment should be considered"/>
```

## **D. Include Proposals (if appropriate)**

Response messages most commonly include one or more of the following:

- ObservationResults inferred by the rules from the submitted data,
- Any of the "...Proposal" clinical statements,
- Bounced input data, sometimes with additional elements populated, or ObservationResults nested within them.

<Include an example of a response message that includes all of the above elements:>

## E. How to Create Return Messages from your Rules

There are two techniques for creating return messages from your rules. They can be combined in rules to create quite complex and specific output patterns.

### 1. Flag Input in FactLists as “toBeReturned”

Any of the input *clinicalStatement* and *entity* data that is present in Drools fact lists can be flagged to be returned in the output message from OpenCDS. By default, the Patient ID and Demographics (and the same thing for OtherEvaluatedPersons) are the only thing that will be returned, but you can add to that from your rules, if required.

You will undoubtedly need to return additional information. In some cases you might want to set some value on an input clinical statement, and return that clinical statement as output. For example, you might want to populate an Interpretation on an ObservationResult and return the input ObservationResult with that one additional element as an output or response. This is easily done by setting the *toBeReturned* flag to true on that particular *clinicalStatement*.

### 2. Create New Output Elements in Global NamedObjects

The basic technique for returning a result is to do the following:

- create a new ObservationResult or other clinical statement,
- set appropriate values in it,
- set the flag *toBeReturned* (present in all clinical statements) = true (because you can also use the “namedObjects” as temporary storage for working variables and intermediate results in your rules without flagging them *toBeReturned*...), and
- add the clinical statement to the global element “namedObjects”.

Note that other output elements (e.g., ClinicalStatementRelationship, Entity, etc.) can also be returned as a part of the output. These (and any other output elements you create from scratch) must also be added to “namedObjects” in order to appear in the response message from OpenCDS. Examples of how to do this appear in the sample “OutputNested...” DSLs. Remember:

- a. You can create an object (of any type), and add it to “namedObjects,” and you can reference the contents of the object in a rule, and/or use it as part of the output response from OpenCDS, but Drools will not fire a rule based on a change to the contents of a global variable (which is what “namedObjects” is). Drools only fires rules based on changes to facts in fact lists.
- b. You can create new facts while running a rule, and reason on them. Changes to these new facts can cause rules to fire (unlike changes to globals). However, new facts will not be returned to OpenCDS to be used as response messages unless they are structured as ClinicalStatements and you also add them to “namedObjects.”

See the sample DSLs whose names begin with “Output...” for examples.

## ***V. Common Patterns for Creating Responses***

### **A. Write me...**

asdf