

TOOL SUPPORT FOR OBJECT ORIENTED FRAMEWORK REUSE

Toacy C. Oliveira¹, Paulo Alencar², Don Cowan³

¹*PESC/COPPE Federal University of Rio de Janeiro, Brazil*
toacy@cos.ufrj.br

^{2,3}*David Sheriton School of Computer Science, University of Waterloo, Canada*
(palencar,dcowan)@csg.uwaterloo.ca

Abstract: Object oriented Frameworks became a popular manner to improve software development lifecycle. They promote reuse by providing a semi-complete architecture that can be extended through an instantiation process, to integrate the needs of the new software application. Instantiation processes are typically enacted in an ad-hoc manner, what may lead to a tedious and error-prone procedure. This work leverages on our previous work on the definition of RDL, a language to facilitate the description of instantiation process, and exposes the ReuseTool, which is an extensible tool to execute RDL programs and assist framework reuse by manipulating UML Diagrams. The ReuseTool integrates an RDL Compiler and a Workflow engine to control most of the activities required to extend a framework's design and therefore incorporate the application needs. This work also exposes how the tool can be extended to incorporate new reuse activities.

Keywords: UML, Object Oriented Framework, Software Process, Software Reuse

1. INTRODUCTION

With the popularity of object oriented languages such as Java and C++, and the pressure for improving software development productivity, the concept of object oriented frameworks gained momentum and became the de facto approach to develop complex software systems. In essence, frameworks are reusable assets in the form of quasi-complete and flexible software constructions, specially assembled to reduce (Bosch, 1999) the effort to develop new application within a specific domain. An object oriented framework defines a set of permanent features known as frozen-spots (Pree, 1995) to deliver unchangeable functionality. It also uses typical object-oriented techniques to incorporate flexible features, the hot-spots (Pree, 1995), which must be appropriately extended to

integrate application specific needs (Markiewicz, 2001) (Mattsson, 2000). Examples of major current object oriented frameworks are: J2EE (J2EE, 2010), DotNET (DotNET, 2010), Hibernate (Hibernate, 2010), JUnit (JUnit, 2010), Eclipse (Eclipse, 2010), Struts (Struts, 2010) and MooTools (MooTools, 2010).

In contrast to Software Product Lines (Atkinson, 2001), where reuse occurs from pulling together a set of pre-defined software components, developers reusing object oriented frameworks need to engage on a more difficult reuse procedure that typically involves understanding the framework design rationale and programming. For example, a developer must be knowledgeable of the order in which hotspots are refined as they may have an implicit sequencing. Considering frameworks may contain several hotspots, such reuse order should be clearly specified to avoid faulty reuse processes (Fayad, 1999) (Kirk, 2005) (Hou, 2005). Moreover, it's expected that reuse processes of complex frameworks are supported by tools to assist handling all the constraints that are present when reusing a framework.

In this work we expose the ReuseTool as a way to assist object-oriented frameworks reuse. The tool rationale is to orchestrate reuse actions within a process that is specified by the framework developer and executed, interactively, by the framework reuser. The reuse process is specified using the RDL (Reuse Description Language) (Oliveira, 2004)(Oliveira, 2007), which is a special language that allows the specification of process flows such as sequencing, loops and branches, and also admits commands to manipulate UML-based framework design.

This work is organized as follows....

2. THE APPROACH OVERVIEW

As pointed out by Krueger (Krueger, 1992), most reuse processes hovers around four dimensions: Abstraction, Selection, Specialization and Integration. The Abstraction and Selection dimensions deal with the cognitive aspects of identifying a set of possible reusable assets and selecting the most suitable to implement the requirements associated with the new application. The cognitive nature of

these two dimensions lays on the fact that requirements are typically expressed with natural language, which demands human interpretation. The Specialization dimension appears once the reusable artefact is chosen and needs to be tailored as to incorporate the application specific increments. Last but not least, the Integration dimension deals with combining several reusable artifacts into a single system.

In this scenario, the Reusetool aims at facilitating the Specialization dimension with possible effects to Selection and Integration. The Specialization dimension for object oriented frameworks deals with adding new design elements and code to the original framework design, where each new element connects to a framework's hotspot. The connection between the new design element and the hotspots extends the framework with information that is specific to the application under development.

In order to orchestrate the framework specialization, the ReuseTool takes an RDL Program and the Framework UML Model as input and produces the Application UML Model. The Framework UML Model represents framework's classes and relationships and also provides useful information on how code can be obtained. The RDL program details how the UML Model should be manipulated to accommodate new design elements related to the new application and the Application UML Model is the final application design.

As illustrated in Figure 1 the process of executing the ReuseTool is quite straightforward. In Figure 1.1 the Framework UML Model and the RDL Program are passed to the tool. The framework model has classes *Text* and *Style*, to indicate the hypothetical framework is capable of applying styles to a given text such as in a word processor. The framework developers understand they can't provide all types of styles so they have decided to leave the style feature as a hotspot that must be configured by the application developer.

Figure 1.2 shows how the framework developers have exposed the *Style* hotspot and how to extend it. The rationale is based on the fact that the framework design has the class *Style* and this class is responsible for the style feature. Consequently the specialization of hotspot *Style* requires the specialization of the *Style* class, what is represented in the RDL Program by the line "CLASS_EXTENSION (Style,myPack,?);".

The `CLASS_EXTENSION` command indicates the class *Style* must be specialized with a new class that will represent the new style. The question mark (?) passed as parameter is a RDL feature called *Reuser Interaction* and indicates a placeholder for a name that should be given at runtime (i.e. reuse-time) by the framework reuser. Reuser interaction is actually exposed at Figure 1.3 when the reuser indicates the new style will be called *Header*.

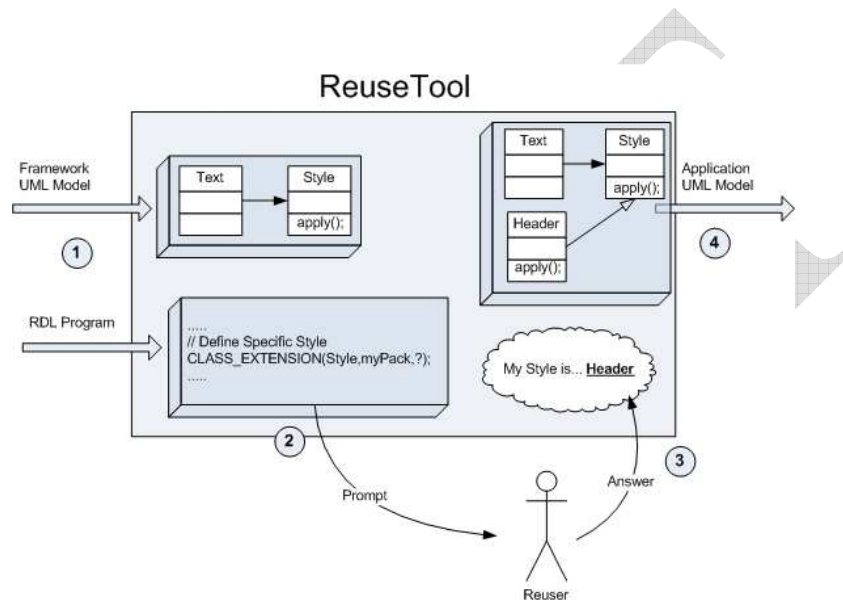


Figure 1 – ReuseTool operation

The result of the reuse process is an extended model called the Application UML Model (Figure 1.4), which contains the design elements to represent application specific requirements. In our example, the new model has the class *Header* as a sub-class for class *Style*, satisfying the reuser requirements from Figure 1.3.

Besides orchestrating reuse actions along the Specialization dimension, the ReuseTool can also impact the Selection and Integration dimensions. Selection can take into consideration the effort needed to reuse a framework measured in terms of the complexity to execute RDL programs. Frameworks with complex RDL programs may be avoided as the complexity may indicate more reuse effort. Assisting with Integration can be more straightforward as some integration actions may be incorporated into the RDL language as a new type of reuse action. The impact on the Selection and Integration dimensions are still under investigation.

3. THE REUSE TOOL

The ReuseTool aims at helping developers to follow a strict process when working on the Specialization Dimension as discussed in Section 2. By a strict process we mean a process that guides the framework reuser to extend all the required hot-spots in a sequence that is pre-defined by the framework engineer, thus following the same rationale used in the framework implementation.

In order to create a process based execution environment we have established a set of requirements for the tool.

Adopt current trends. Besides the use of RDL, the tool should follow current trends in the Information Technology scenario to avoid a steep learning curve and allow further integration with other approaches.

Handle object oriented frameworks. Object-oriented programming paradigm is used everywhere and has vast framework examples we can leverage on. It also provides a potential market for our approach.

Provide ways to pause and resume the process. A reuse process can be lengthy so the ability put the process in a suspension mode and resume with no loss of context is an important feature.

Provide infrastructure for a multi-“reuser” scenario. Nowadays frameworks have to tackle multifaceted scenarios so they are typically complex and require different types of expertise during customization. As a result the framework reuse process became multi-user by nature and our tool should provide means to support it.

Extensibility. RDL provides the building blocks to represent reuse process and manipulate UML Models. However it's important to leave space for improvements, such as handling forthcoming UML characteristics and new programming paradigms such as Aspect Oriented Programming. In this context, the tool must be open for extension, as becoming itself a framework that can be extended to handle other reusable assets.

In order to cope with all requirements we have designed a software architecture that integrates a workflow engine, a RDL compiler and Java object-oriented framework as described in Section 3.1.

3.1 Architecture

Reusing a framework is a human-oriented process by definition. Each framework hotspot must be extended with application specific information that is essentially known by the application developer at reuse time. According to (Georgakopoulos, 2005), a suitable way to represent a human-oriented process is by using the workflow technology. In summary, a workflow organizes a set of human-tasks, system-tasks and control flow structures to enact a given process [WorkflowThesis]. For a reuse process, human-tasks are those where a reuser introduces application specific information. In the same way, system-tasks are those devoted to manipulate UML models based on the information presented by the reuser, and control flow structures allow the representation of loops, decisions, and other workflow patterns (Workflow Patterns, 2010).

It is important to mention that our process is represented with RDL, an imperative language specially created to specify object oriented frameworks reuse. For that reason, our workflow is represented neither as set of icons such as in YAWL (YAWL, 2010) and JPDL (JPDL, 2010) nor with XML as in xPDL, but with an algorithm-like script.

In order to create a tool that is capable of executing a RDL-based workflow that takes into consideration the list of requirements exposed previously, we managed to design a software architecture that uses a workflow engine, a parser generator, a UML model manipulation library and some helper classes.

As illustrated in Figure 2 the architecture has three main components: the Transformer, the JPDL Process Language and the Process Execution Machine.

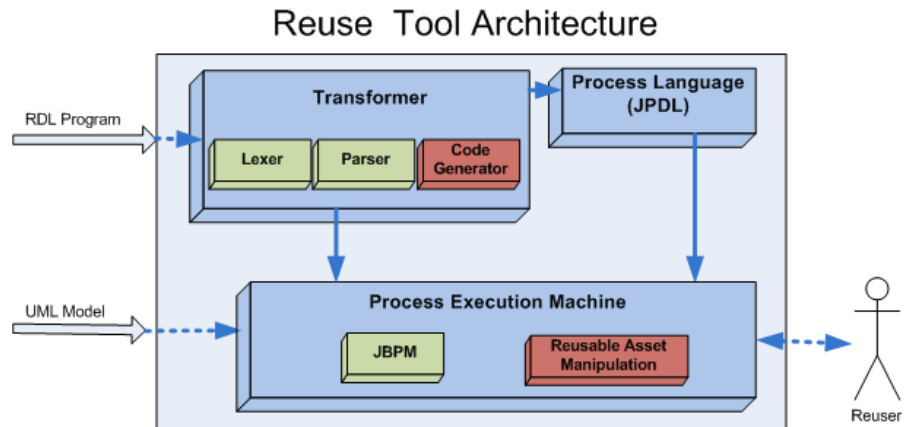


Figure 2 – ReuseTool Architecture

The Transformer

The Transformer aims at converting a RDL Program into a JPDL process. It has three sub-components: the Lexer, the Parser and the Code Generator. The Lexer and Parser assure the RDL Program is well formed. We have used the ANTLR parser generator program to create the Lexer and Parser based on the RDL Grammar as indicated in Figure 3.

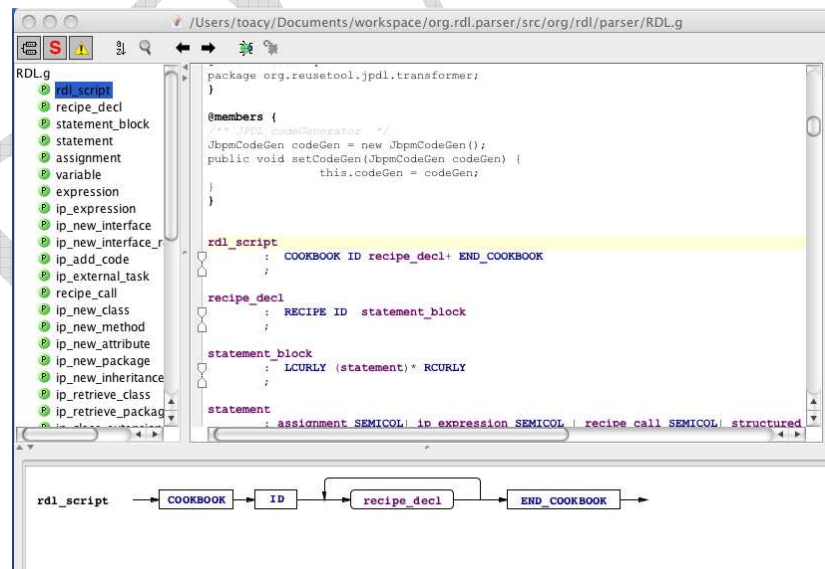


Figure 3 – RDL BNF and ANTLRWorks

The *CodeGenerator* is a set of Java classes that are capable of writing JPDL code according to the RDL command semantic. For example Figure 4.a illustrates the `CLASS_EXTENSION` command

and how it is converted as a JPDL custom node (b). Note the parameters *Shape*, *packA* and *?*, passed to the RDL command were represented as arguments passed to the constructor of the custom node.

```
// Define Shape SubClass
shapeClass = CLASS_EXTENSION(Shape, packA, "?");
```

(a)

```
</custom>
<custom class="org.reusetool.jbpm.pem.reuseactions.ClassExtensionHandler"
  g="281,211,80,40" name="7">
  <constructor>
    <arg>
      <string value='Shape' />
    </arg>
    <arg>
      <string value='packA' />
    </arg>
    <arg>
      <string value='?' />
    </arg>
  </constructor>
  <transition g="-46,-18" name="to 8" to="8" />
</custom>
```

(b)

Figure 4 – Code Generation

The Lexer, Parser and Code Generator work synchronously. After the Lexer converts the RDL text into meaningful tokens, the Parser tries to identify a sequence of tokens that matches a non-terminal symbol described in the RDL Grammar. Once the Parser matches a valid sequence, it calls the Code Generator to spit JPDL code accordingly. We managed to achieve this level of integration by inserting Java commands into the RDL Grammar, which is a feature supported by ANTLR. Figure 5 illustrates the main classes for the Transformer component.

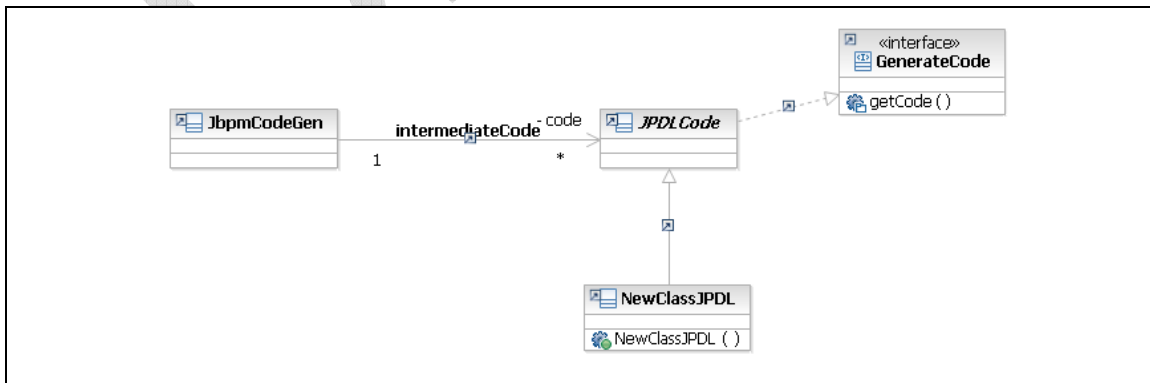


Figure 5- Main Classes for the Transformer component

The Process Language

The next ReuseTool component is the JPDL Process Language. JPDL is an executable process language that allows the declarative definition of workflows. JPDL uses XML technology to represent process elements such as system tasks, human tasks, workflow patterns, swimlanes and subprocesses. A JPDL file can be deployed and executed with the jBPM engine. Together, JPDL and jBPM are integrated in the jBPM Platform, which is a JBOSS (JBOSS, 2010) community product for process representation and execution. We have chosen the jBPM Platform as it fulfils some of the ReuseTool requirements such as process orchestration in a multi-user environment, process persistency and extensibility.

JBPM is oblivious to the nature of the workflow tasks it executes. Tasks can be either human-oriented such as signing a document or system-oriented such as performing a complex computation, but internally JBPM treats them as an executable item that fits within the workflow graph. In fact, JBPM is itself based on the framework approach where tasks are hotspots that can be extended to allow the introduction of new executable items. We have benefited from such extension mechanism to smoothly integrate all RDL reuse-oriented tasks and leave space for further additions.

RDL tasks were mapped to JPDL *Custom Nodes*. We have decided to use JPDL Custom Nodes as they allow binding a Java class to implement the desired node behaviour. An instance of the java-class will be later created by the underlying process execution machine and initialized with the parameters passed under the *constructor* tag. For example, in Figure 4.b the Custom Node with name

7 (seven) is bound to the class *org.reusetool.jbpm.pem.ClassExtensionHandler*, which is responsible for implementing the CLASS_EXTENSION task.

The Process Execution Machine

The last but most important component of the ReuseTool is the Process Execution Machine. The Process Execution Machine, or simply PEM, is responsible for executing the reuse process described with JPDL that will extend the given framework design and eventually turn it into the application design. To accomplish such a goal, the PEM component extends the jBPM platform with a set of Java classes capable of handling reuse-oriented tasks¹. Reuse tasks are those executed by the framework reuser when specializing the frameworks' hotspots.

The PEM internal structure depends on the type of node used in the JPDL workflow. According to the JPDL reference manual (JPDL, 2010), Custom Nodes are place holders for objects implementing the *org.jbpm.api.activity.ExternalActivityBehaviour* Java interface provided by the JBPM library. As a result, when the JBPM workflow engine needs to execute a Custom Node, it actually instantiates the user-defined Java class that implements the associated behaviour, which in our case is a type of reuse action. Figure 6 illustrates how PEM classes are structured to implement the NEW_CLASS reuse action used in Figure 4. It's important to observe the whole PEM implementation contains more than 50 classes to support the implementation of all RDL reuse actions (More on the PEM's internal rationale will be discussed at Section 6.2-Extending the Architecture).

¹ We have used the terms "reuse action" and "reuse task" indistinctively

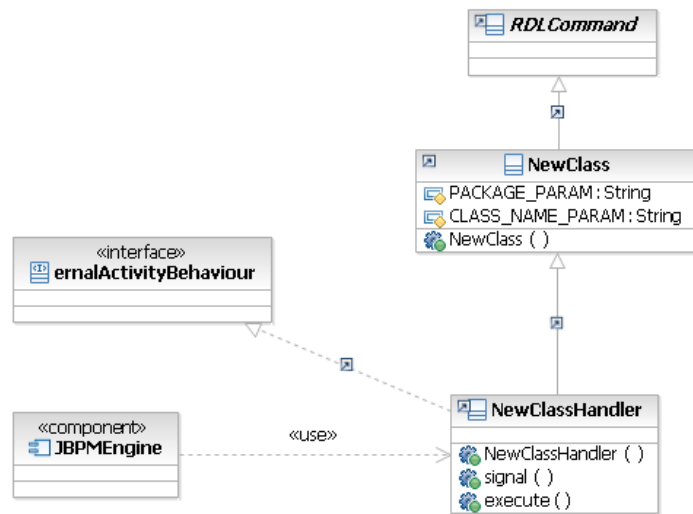


Figure 6 - PEM Classes for the NEW_CLASS Reuse Action

It's important to point out that PEM operates on UML Models specified as “.uml” files. Figure 7 brings an illustration for the UML File: on the right there is a tree-like rendering with classes and associations; on the left there is the actual file content for the class *Activity*.

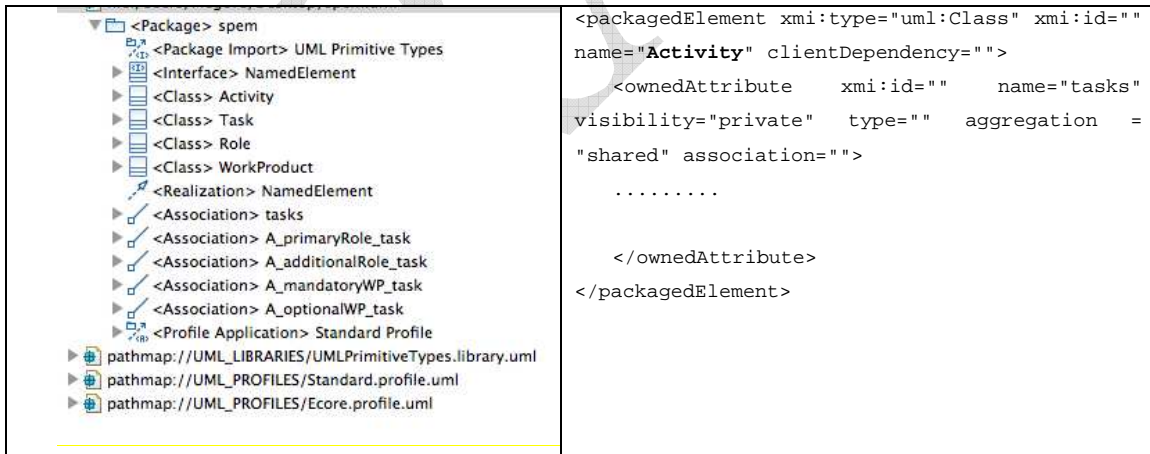


Figure 7 - UML File Example

3.2 Software Process for Framework Reuse

A software process organizes the actions needed to achieve a given goal; specifies the individuals that participate in an action; and also describes the manipulated information. We believe using a software process to expose the ReuseTool functionalities would be a valuable contribution and also facilitate understanding the approach. For that reason we have devised the Software Process for Framework Reuse. The process is specified with the Software Process Engineering MetaModel (SPEM), which is a standard notation for representing software processes.

SPEM allows the specification of process elements namely roles, artifacts and tasks. Roles define a set of characteristics and attributes an individual must possess to execute tasks in a software process. Artifacts enclose the information manipulated in a task in one or more documents. Finally, tasks represent a set of actions that must be handled by an individual playing a role. A task can also indicate the list of preceding and succeeding tasks to define the sequence in which tasks should be handled.

The Software Process for Framework Reuse has two phases, the Framework Development and the Framework Reuse that are executed in a sequential order. The Framework Development phase is the first to be handled and it attempts to create a working object-oriented framework, its associated UML-based design documentation and the RDL reuse program. We do not detail the tasks executed in this phase as we believe most approaches to modern software development are capable of developing a framework. Nevertheless, the Framework Development phase includes one role and two artifacts that will be used in the Framework Reuse Phase and are described in the Table 1.

Model Element	Name	Description
Task	Develop Framework	Umbrella task to represent all tasks required to develop the object-oriented framework.
	Develop Framework Model and RDL Program	Focus on developing the artifacts related to the reuse process.
Role	Framework Developer	Responsible for defining the UML Model and RDL Program that will be used to assist the framework reuse. A person playing this role must deeply understand the

		framework underlying concepts and hotspots, and also be knowledgeable in RDL and UML
Artifact	Framework UML Model	The UML Model must specify the classes and relationships that describe the object-oriented framework implementation.
	RDL Program	The RDL Program organizes the reuse actions that will extend the framework hotspots.

Table 1 - Process Elements for the Framework Development Phase.

Once the framework is developed and its associated UML Models and RDL Programs are finished the Framework Reuse phase can occur. This phase aims at tailoring the framework design to include the application specific extensions informed by the framework reuser. In order to achieve its goals the Reuse phase uses the artifacts Framework UML Model and RDL Program defined previously but also defines its own process elements as described in Table 2.

Model Element	Name	Description
Task	Load Framework Model and RDL Program	Inform the ReuseTool the Framework UML Model and associated RDL Program.
	Execute Reuse Process	Execute the Reuse Process. This task is an umbrella task to represent all possible reuse actions according to the RDL Language. Typical reuse actions that require reuser interaction and related RDL command are: <ul style="list-style-type: none"> - Extending a Class -> CLASS_EXTENSION - Creating a package -> NEW_PACKAGE - Iterating on loop -> LOOP - Confirming an human task -> EXTERNAL_TASK
Role	Framework Reuser	Responsible for executing the framework reuse process. Must be knowledgeable about the application needs to extend the framework at the proper hotspots. Must also understand the framework documentation.

Artifact	Application UML Model	The Application UML Model contains the new classes and design elements that represent the application specific needs.
-----------------	-----------------------	---

Table 2 - Process Elements for the Framework Reuse Phase.

Figure 8 illustrate the Software Process for Framework Reuse using the SPEM notation. At the top of the figure are represented two phases, the Development on the left and the Reuse on the right. Each phase brings a sort of workflow diagram that specifies the order in which each task should occur. At the bottom of the same figure we represented the relationships between the model elements. For example at the bottom-right, we show the task Develop Framework Model and RDL Program is performed (<<performs>> symbol) by the role Framework Developer to create (<<output>> symbol) the Framework UML Model and RDL Program artifacts.

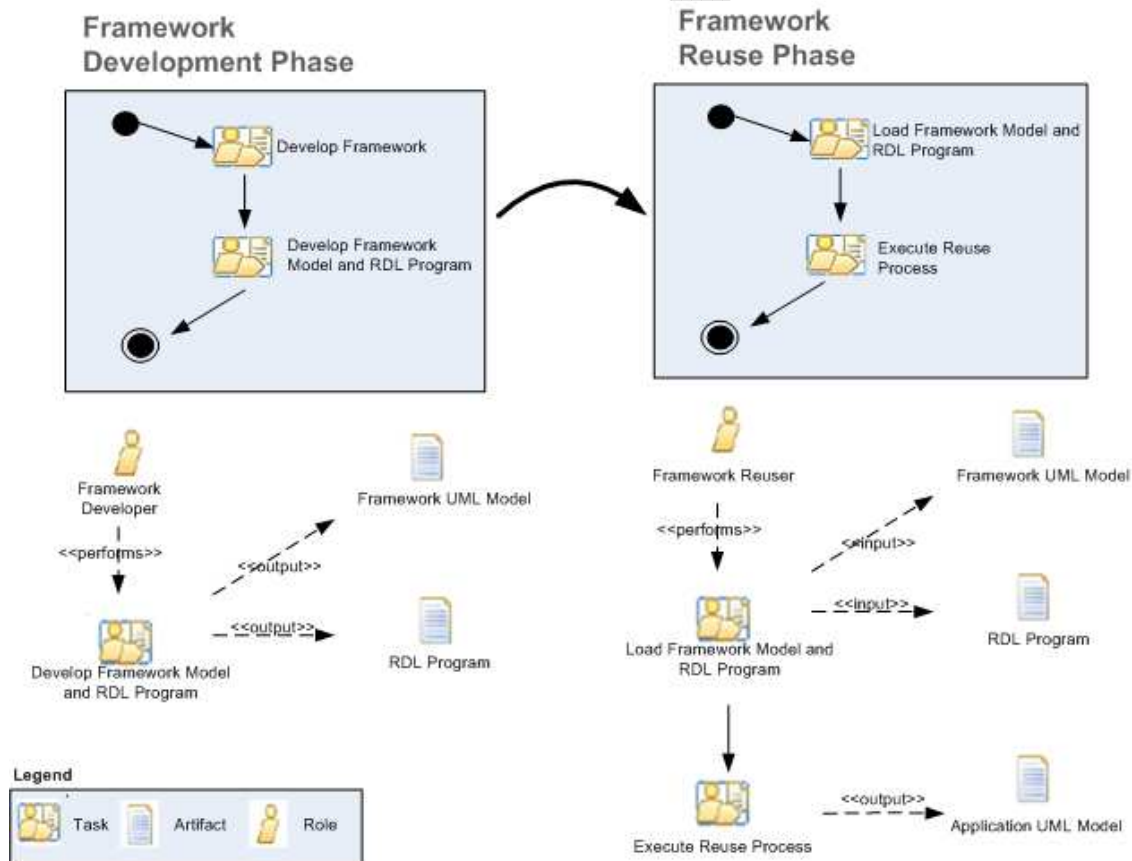


Figure 8- Process Elements

Although the process specification is presented at once, its actual execution takes place in two distinct environments, Development and Reuse. The two environments may differ in time, location, development team and in fact they are typically part of two separate software projects. It's also important to point out that modern software process are iterative by nature thus tasks are orchestrated according to the project's constraints such as time-to-market, budget, team allocation, etc. As a result, the tasks described in the Software Process for Framework Reuse may be interleaved by other tasks.

4. AN ILLUSTRATIVE EXAMPLE – THE SHAPE FRAMEWORK

This section illustrates how the ReuseTool can be used to extend an existing framework by exploring the artifacts that are manipulated in the process. We have chosen the Shapes Diagram Editor (GEF, 2010) to be our example as it has a comprehensive documentation and it is part of the Eclipse Graphical Editing Framework, what makes it extremely available to general public. It's also important to mention the Shapes Editor is a 3rd party implementation thus indicating the ReuseTool is not biased towards our frameworks but can be used in a broader context.

4.1 The Shapes Framework

The Shapes Diagram Editor is an Eclipse Plug-in developed as an extension to the Graphical Editing Framework (GEF) (GEF, 2010). Shapes Framework facilitates development of diagram editors made out of nodes and connections that can be customized by a developer to represent all sorts of icons and connecting lines respectively. A typical Shapes instance is shown in Figure 9 with a diagram customized to handle circles and squares that can be connected by solid or dashed lines.

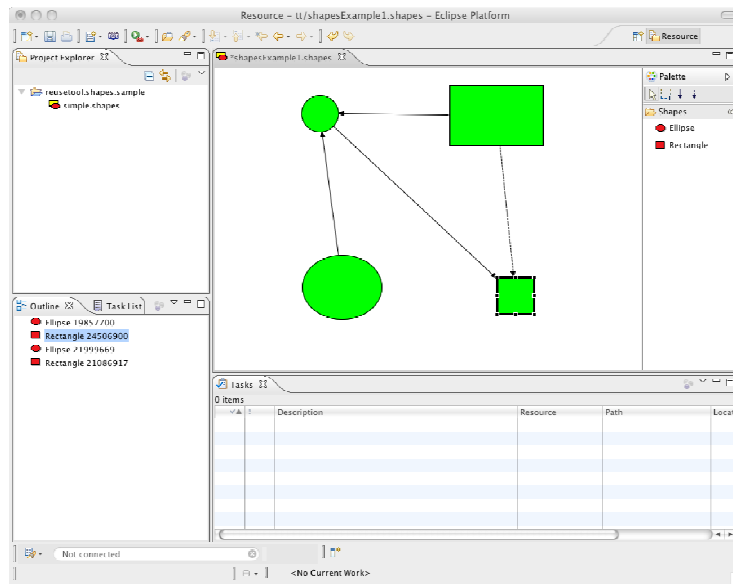


Figure 9 - Shapes Look and feel.

A diagram within a Shapes instance has ordinary drawing features such as creating nodes and connections, selecting nodes, resizing nodes, deleting nodes, moving nodes and deleting connections. Connections can also use a special feature to restrict the link between two undesired nodes. For example, it's possible to represent squares and triangles cannot be connected by dashed lines.

In order to provide all its features, Shapes blends itself with GEF, which is Eclipse's building block for creating graphic editors. Together, GEF and Shapes have several classes that represent an intricate design, allowing the reuser to represent diagrams based on their model, view and edit parts (controllers), such as in a typical Model-view-controller pattern. Model represents the semantics behind each node; View specifies how nodes are rendered and; Edit parts connect the Model with the Views. Our goal is to demonstrate how to reuse Shapes using the ReuseTool. Therefore we will not explain the Shapes' design rationale but explore its main hotspots to develop a typical diagram.

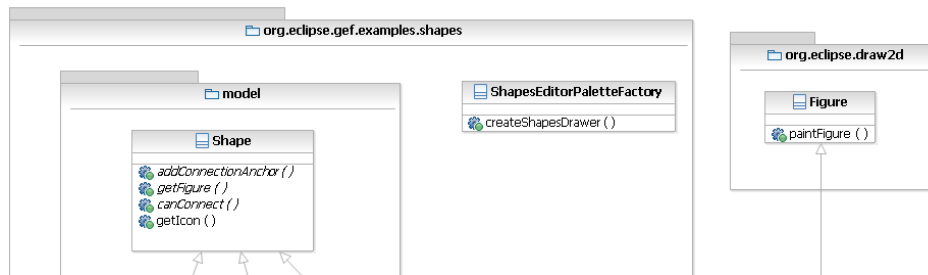


Figure 10 - Shapes Hotspots

Shapes main goal is to facilitate creating new nodes so we have chosen this feature to demonstrate the ReuseTool. In order to create a new node the reuser has to partake three reuse actions:

- Define the new shape model
- Define the new shape look and feel
- Insert the new shape in Shapes palette.

Fortunately the representation for the hotspot to develop new nodes using Shapes is straightforward. Figure 10 exposes the UML class diagram containing the framework classes *org.eclipse.draw2d.Figure*, *org.eclipse.gef.examples.shapes.model.Shape* and *org.eclipse.gef.examples.shapes.ShapesEditorPaletteFactory*. The Shape class is the main participant in the hotspot as it defines a set of abstract methods that must be extended by the application reusing the framework. In our scenario the most important methods are: *getIcon()*, *getFigure()* and *canConnect()*. The *getIcon()* method should return the icon that represents the node in the diagram palette to represent the node; the *getFigure()* method should return the graphical elements that represent the node in the diagram canvas and; the *canConnect()* method should return if the current shape can connect with the target shape that is passed as parameter.

The abstract class *Figure* is defined in the Eclipse Draw2d library as a general representation for two dimensional visual elements. It declares several fields to customize the look and feel for graphical elements. It also declares the method *paintFigure()*, which is responsible for rendering the image. In order to extend this class the reuser must redefine the *paintFigure()* method and add the required visual details. It's important to point out that Eclipse has several out-of-the-box visual elements such as *Ellipse*, *Polygon* and *Triangle* that can be use as a subclass for *Figure*.

The last class we need to handle is the *ShapesEditorPaletteFactory*. This class organizes the entries in the palette found in the diagram editor interface. Reusing *ShapesEditorPaletteFactory* requires a precise incision to modify its existing java file at the method *createsShapeDrawer()*, without damaging the whole system integrity.

4.2 The Shapes RDL Program

Assisting the reuse of the Shapes framework requires an RDL Program as the one represented in Code 1. In RDL, programs are described as cookbooks that contain recipes. Recipes list RDL commands (shown in bold) in the same way a procedural language such as Java exposes its commands inside procedures and methods. Code 1 also declares the cookbook *ShapesProducts* (Line 1) and the *main* recipe (Line 2). The *main* recipe starts by creating the package *org.reusetool.example.myshapeseditor* (Line 4) to group the application-related design elements in a single place. It's important to observe that variable *packA* was declared in Line 2 to store the just created package model element. This was required allow reference to the package element later in the program.

```

1. COOKBOOK ShapesProducts
2. RECIPE main{
3.
4.     packA = NEW_PACKAGE(FrameworkModel,"org.reusetool.example.myshapeseditor");
5.     LOOP ("Create another shape?")
6.     {
7.         // PREPARE Images
8.         EXTERNAL_TASK("Define 16x16 icon");
9.         EXTERNAL_TASK("Define 24x24 icon");
10.
11.        // Define Shape SubClass
12.        shapeClass = CLASS_EXTENSION(Shape, packA, "?");
13.
14.        // Refine Abstract Methods
15.        m = METHOD_EXTENSION(Shape, shapeClass, addConnectionAnchor);
16.        ADD_CODE(shapeClass, m, "return new ChopboxAnchor(iFigure);");
17.        m = METHOD_EXTENSION(Shape, shapeClass, getFigure);
18.        ADD_CODE(shapeClass, m, "return new IFigureSubclass();");
19.        m = METHOD_EXTENSION(Shape, shapeClass, getIcon);
20.        ADD_CODE(shapeClass, m, "return createImage(\"NEW_SHAPE.gif\");");
21.
22.        // Configure Palette
23.        ADD_CODE(ShapesEditorPaletteFactory, createShapesDrawer,
24.            "component = new CombinedTemplateCreationEntry(
25.                \"NEW_SHAPE\",
26.                \"Create a NEW_SHAPE shape\",
27.                NEW_SHAPE.class,
28.                new SimpleFactory(NEW_SHAPE.class),
29.                ImageDescriptor.createFromFile(ShapesPlugin.class, \"icons/crazy16.gif\"),
30.                ImageDescriptor.createFromFile(ShapesPlugin.class, \"icons/crazy24.gif\");
31.            componentsDrawer.add(component);
32.        ");
33.

```

```

34.      // Redefine Method to treat How Connections can be handled
35.      IF ("Add Feature - Restrict Connections?") THEN {
36.          m = METHOD_EXTENSION(Shape,shapeClass,canConnect);
37.          ADD_CODE(shapeClass,m,"Code that checks if the connection can be made.");
38.      }
39.      IF ("Add Feature - New Figure?") THEN{
40.          // Define Shape SubClass
41.          figClass = CLASS_EXTENSION(Figure, packA,"?");
42.          m = METHOD_EXTENSION(Figure,figClass,paintFigure);
43.      }
44.  }
45.  }
46.  END_COOKBOOK

```

Code 1 - RDL Program

Creating the RDL requires acknowledging development and reuse occur in different times. Therefore and by the time the Shapes RDL program is implemented by the framework developer, the number of shapes the framework reuser will create is unknown. To mitigate such issue, Line 5 declares a loop that will iterate several times according to the framework reuser's need. The loop will surround all reuse actions related to shapes creation, from Line 5 to Line 44.

Lines 8 and 9 declare two external tasks. RDL uses an external task to indicate the reuser has to perform an action that is impossible to be represented as a model element manipulation. When the ReuseTool reaches an `EXTERNAL_TASK` it stops its execution and wait for the reuser feedback acknowledging the task was completed. For example, the Shapes framework requires an icon to be shown in the palette. An icon is a visual element that cannot be represented by a design element such as a class, which means neither RDL nor UML can represent it. However, the action of defining the icon and placing it in the proper folder needs to be executed by the reuser so that the framework can be properly extended. Line 8 uses the external task command to resemble the reuser he needs to *"Define a 16x16 icon."*

One of the most important aspects of the ReuseTool is creating the design elements that extend the framework hotspots. Lines 12 through 42 bring several RDL commands that are responsible for extending the framework design at specific places to include the application specific requirements. Line 12 uses the `CLASS_EXTENSION` command to indicate the class *Shape*, shown as a hotspot in Figure 11, needs to be sub-classed. This command refers to the package variable *packA* defined at Line 4 as the container for the new class. Note the presence of a question mark as the third parameter in the `CLASS_EXTENSION` command. RDL uses this character to indicate the parameter is unknown at compile time and must be discovered at runtime (reuse-time). Line 12 also declares a

variable to represent the just created class for later reference. Following the execution of the CLASS_EXTENSION command the resulting model should contain a new class that extends the Shapes class as shown in Figure 11.

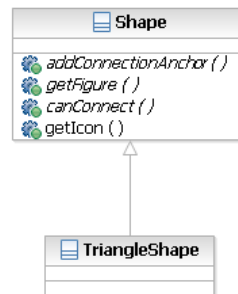


Figure 11 - CLASS_EXTENSION with “?” = TriangleShape

Line 15 shows the METHOD_EXTENSION command, which also introduces a new design element. The METHOD_EXTENSION command indicates a method from the super-class needs to be refined by a method in the sub-class, which in terms of modelling means a method with equal signature must be declared in the sub-class. In Line 15, the method *addConnectionAnchor* found in the class *Shape* is refined in the sub-class referred by the variable *shapeClass*. Figure 12 illustrated the model after executing Line 15.

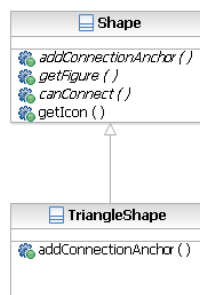


Figure 12 – METHOD_EXTENSION with *addConnectionAnchor*.

The ReuseTool assists framework reuse by manipulating design elements according to the hotspots it needs to extend. Nevertheless, the gap between the application design and the application implementation, the actual code, can be still considered a hurdle the reuser must overcome. In order to facilitate implementing the hotspots with application specific code, RDL introduced the ADD_CODE command. Line 16 illustrates how the ADD_CODE command is used to “add the code

snippet” *“return new ChopboxAnchor(iFigure);”* to the method referred by the variables *shapeClass* and *m*, which point to the just created class and the refined *addConnectionAnchor* method respectively. The text “add the code snippet” was quoted in the previous sentence because the ReuseTool does not attach plain code to the method but instead creates a comment that is linked to the method. The reason for creating the model element comment is twofold: i) RDL and UML are programming language agnostic; ii) the new code depends on the application information that is unknown when the RDL program is created. Although the pre-programmed code fragment may not fit exactly in the application, we believe it gives valuable information on a direction to be pursued by the framework reuser during application implementation.

Lines 17 through 34 repeat the METHOD_EXTENSION and ADD_CODE commands to extend the methods related to the icon, figure and palette hotspots. Line 35 however, exposes another RDL feature, the conditional command IF. The conditional command works in the same way as in an imperative programming language such as Java but the expression being specified as interaction with the reuser. For example, the Shapes hotspot that restricts how shapes can be connected is optional so the reuser has to be asked if he wants to extend it or not. Line 35 declares *“IF (“Add Feature - Restrict Connections?”) THEN”* to request if the reuser needs to add the feature that restricts shapes connections to the application. If the response is yes the ReuseTool will execute lines 36 and 37, otherwise jumps to Line 38. Lines 39 through 42 also use the conditional command but this time to extend the figure hotspot.

4.3 The Application Scenario

Reusing any asset requires some planning to determine what hotspots will be extended and what information will be provided. In this section we provide the rationale of the application we have developed reusing the Shapes framework. We have decided to go for a very simple application as our intention was to demonstrate the ReuseTool without overcomplicating the text.

The application requirements can be specified as:

“an application to create a diagram editor that exposes three types of shapes: triangles, labelled rectangles and rectangles with a red cross in the middle. The application also needs to restrict the connection between triangles and labelled rectangles.”

After browsing the Shapes hotspots documentation, the reuser identified the need to create the classes *TriangleShape*, *LabelledShape* and *RedCrossShape* to represent shapes and the class *RedCrossFigure* to render the rectangle containing a red cross in the middle. Table 3 contains a trace with the reuse actions executed in the process considering those classes were created in the order *TriangleShape* => *LabelledShape* => *RedCrossShape* => *RedCrossFigure*. The first column contains the corresponding line from the ShapesProduct cookbook and the second indicates the reusers response for each new shape class. For example to add the shape *LabelledShape* the reuser should answer first answer “Yes” to indicate the need of a new shape and then confirm the two actions that define the icons by pushing the enter button. After that the reuser should enter “*LabelledShape*” as the name of the class, following to “No-s” to decline creating restrictions rules and new figures. In this case, Line 41 is not executed (not applicable).

RDL Command	Reuser Response		
	<i>TriangleShape</i>	<i>LabelledShape</i>	<i>RedCrossShape</i>
5. LOOP("Create another shape?");	yes	yes	yes
8. EXTERNAL_TASK("Define 16x16 icon");	enter	enter	enter
9. EXTERNAL_TASK("Define 24x24 icon");	enter	enter	enter
12. shapeClass = CLASS_EXTENSION(Shape, packA, "?");	<i>TriangleShape</i>	<i>LabelledShape</i>	<i>RedCrossShape</i>
35. IF ("Add Feature - Restrict Connections?") THEN {	No	No	yes
39. IF ("Add Feature - New Figure?") THEN{	No	No	yes
41. figClass = CLASS_EXTENSION(Figure, packA, "?");	N/A	N/A	<i>RedCrossFigure</i>

Table 3 - Reuse Actions Trace

Figure 13 shows a UML class diagram containing the final application model. The new shape classes are shown as sub-classes of class *Shape* and the *RedCrossFigure* class as a sub-class of *Figure*. The comment associated to each refined method does not appear in the diagram but can be found in the XML file that represents the UML diagram.

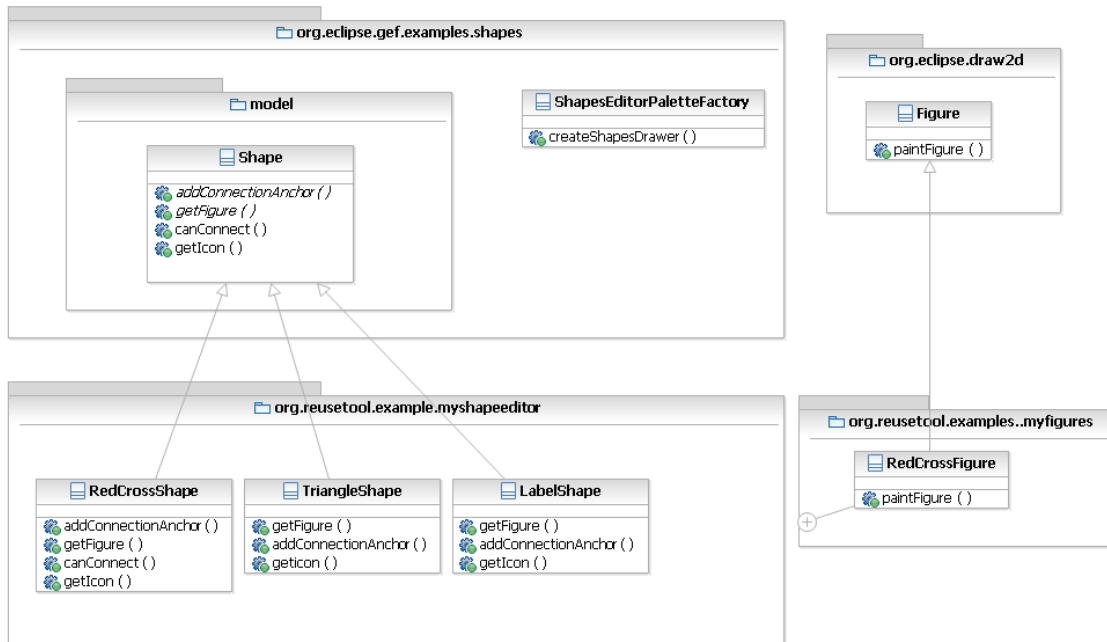


Figure 13- Application Model

4.4 Considerations

Although the Shapes framework has a straightforward reuse process that deals with few hotspots and design elements, we believe using the ReuseTool brought some improvements. Initial analysis about the tool experimentation indicates the sequence of reuse actions follows a seamless path that is carefully designed by the framework developers to facilitate navigating through the framework models. As a result the reuser does not waste time dealing with several classes at once to discover what needs to be done but instead focus on the model elements related to the hotspot he is interested in extending.

Another benefit is also related to the sequence. Some reuse actions must occur before others as they can have impact on the process flow. For example, what's the point of extending the class *Figure* unless the reuser is creating a visual element that is not present in the library, such as in the *RedCross* case? Without the ReuseTool, an unattended reuser may create such class inadvertently.

Another point that is worth exposing in this paper is with regards the Shapes framework design. Although the original Shapes framework has an object-oriented approach and some of its hotspots are

reused through class extensions, there were hotspots that required modifying the existing classes' code. For example, the method *CreateShapesDrawer* from class *ShapesEditorPaletteFactory* needs to have its code modified to add a new palette entry. We believe this is an invasive way to reuse that may expose the framework code to possible misuse so we have modified the Shapes' design and code to avoid such practice. For instance the class *org.eclipse.gef.examples.shapes.parts.ShapeEditPart* implemented the methods *createFigureForModel()* and *getConnectionAnchor()* with code that was dependent on the new shape classes as shown on the left cell of Table 4. We have implemented the same behaviour by delegating such code to the sub-class itself via polymorphism.

Original Code	Modified Code
<pre> private IFigure createFigureForModel() { // if (getModel() instanceof EllipticalShape) { // return new Ellipse(); // } else if (getModel() instanceof RectangularShape) { // return new RectangleFigure(); // } else if (getModel() instanceof CrazyShape) { // return new Label("Testing"); // }{ // // if Shapes gets extended the conditions above must be updated // throw new IllegalArgumentException(); // } } </pre>	<pre> private IFigure createFigureForModel() { // the Shape class knows how to return its associated figure Shape sh = (Shape) getModel(); return sh.getFigure(); } </pre>

Table 4 – Modified code for the method *createFigureForModel*.

4.4.1 Experience

We have also conducted some informal experiments to evaluate the applicability of the proposed approach. The underlying rationale of such experiments was to determine if the ReuseTool was capable of instantiating frameworks from different domains that were created by different development teams, thus avoiding peculiarities related to ReuseTool the developers. The experiments were conducted by 3 people with good object oriented programming and modelling skills with both academic and industry backgrounds. The experiment procedure can be summarized as:

- Choose a particular object-oriented framework from literature or self-made.
- Create the associated UML file, either by hand or reverse engineering from code.

- Create the RDL script files by analyzing the framework documentation and known instances.
- Execute the script using the ReuseTool to create the design for at least two different framework instances.
- Report the experience as either successful or challenging.

Table 5 summarizes the experiment information by exposing the Frameworks' Name and Domain, the Number of Classes, Number of Hotspots and the Number of Lines for the RDL Script.

Framework Name	Domain	#Classes	# hotspots	# RDL Lines
Shapes (Shapes, 2010)	Graphical	13	6	46
JHotDraw (JhotDraw, 2010)	Graphical	90	12	55
Ecommerce Product Line (Gomma, 2004)	Ecommerce	16	10	16
Arcade Product Line (APL, 2010)	Games	16	5	21
REMF (REMF, 2010)	Infrastructure	53	7	27

Table 5 - Experiment Information

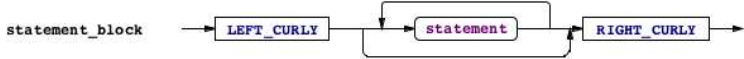
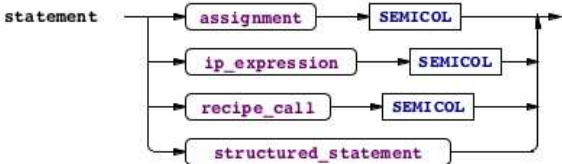
Although the experiments were conducted in an informal and academic environment, rather than using techniques from Experimental Software Engineering (Juristo, 2001), we believe the results are encouraging. All three subjects managed to successfully create framework instances without interference from the frameworks' developers, which indicates the underlying technology is sound and valuable. Most of the reported issues were related to installing the tool as we didn't provide a detailed step-by-step procedure.

5. RDL ENHANCEMENTS AND IMPLEMENTATION

Our paper (Oliveira, 2007) thoroughly describes the Reuse Description Language and how its execution modifies a UML model but briefly discusses about implementation. On the other hand, the course of this work focused on the development of a mature tool for framework reuse based on an established workflow engine and a fully fledged RDL compiler, a scenario that brought an excellent

opportunity to upgrade RDL itself. As a result, we have managed to refactor the RDL BNF to make it more flexible and ready for future extensions. We have also introduced new commands and upgraded old ones to facilitate the reuse experience.

Table 6 describes RDL enhancements. We have divided the table in four groups as follows: BNF Refactoring, Syntactic Sugar, Refactored Commands and New Commands. The BNF refactoring group exposes the adaptations made at the language BNF to make it more flexible to future adaptations and cope with how UML identify design elements. The Syntactic Sugar group reflects some cosmetic changes in the language and basically replaced lengthy command delimiters by curly brackets with no semantic effect. On the other hand the groups Refactored Commands and New Commands unveil broader changes that impact how RDL programs are developed and executed. The former group updated the CLASS_EXTENSION and LOOP commands while the latter introduced commands to deal with packages, interfaces, conditions and human-interaction. Most rows in Table 6 thoroughly describe the modification accompanied by the BNF syntax diagram.

BNF Refactoring	
Blocks	<p>Blocks define a set of commands surrounded by a <i>begin-end</i> pair to facilitate code organization and the introduction of structured commands.</p>  <pre> graph LR statement_block --> LEFT_CURLY[LEFT_CURLY] LEFT_CURLY --> statement statement --> RIGHT_CURLY[RIGHT_CURLY] RIGHT_CURLY --> end </pre>
Structured commands	<p>Statements were modified to accommodate structured commands such as <i>loops</i> and <i>ifs</i>. Combined with blocks, structured statements allow declaring typical code found in most imperative languages such as Java and C++.</p>  <pre> graph LR statement --> assignment statement --> ip_expression[ip_expression] statement --> recipe_call statement --> structured_statement assignment --> SEMICOL1[SEMICOL] ip_expression --> SEMICOL2[SEMICOL] recipe_call --> SEMICOL3[SEMICOL] structured_statement --> SEMICOL4[SEMICOL] SEMICOL1 --> end SEMICOL2 --> end SEMICOL3 --> end SEMICOL4 --> end </pre>
Qualified ids	<p>An ID can be named according to its enclosing module. For example a Java id can contain several package manes connected by dots followed</p>

	<p>by a name.</p> <pre> graph LR qualified_id --> ID1[ID] ID1 --> DOT[DOT] DOT --> ID2[ID] ID2 --> qualified_id </pre>
Ids and Strings	Strings are now surrounded by quotes to differentiate from IDs
Syntactic Sugar	
Removed END_*	END_LOOP and END_RECIPE were replaced by pairs of left and right curly brackets to improve compatibility with current programming languages.
Refactored Commands	
CLASS_EXTENSION with packages	<p>The CLASS_EXTENSION command has a new parameter to indicate the package where the new sub-class should be placed.</p> <pre> graph LR ip_class_extension --> CLASS_EXTENSION[CLASS_EXTENSION] CLASS_EXTENSION --> LP1[LEFT_PARENTHESIS] LP1 --> ID[ID] ID --> COMMA1[COMMA] COMMA1 --> dots1[...] dots1 --> qualified_id[qualified_id] qualified_id --> COMMA2[COMMA] COMMA2 --> STRING[STRING] STRING --> RP1[RIGHT_PARENTHESIS] RP1 --> dots2[...] </pre>
Loop with condition	<p>The LOOP command has a condition that is implemented as a string. The string text is used to prompt the need of another iteration.</p> <pre> graph LR loop_statement --> BEGIN_LOOP[BEGIN_LOOP] BEGIN_LOOP --> LP[LEFT_PARENTHESIS] LP --> condition[condition] condition --> dots1[...] dots1 --> RP[RIGHT_PARENTHESIS] RP --> statement_block[statement_block] statement_block --> dots2[...] </pre>
ADD_CODE	This command had its semantics changed to indicate it actually attaches the code as a comment to the model element that represent the method in UML.
New Commands	
new_package	<p>Creates a model element of type UML::Package with name <i>qualified_id</i>.</p> <pre> graph LR ip_new_package --> NEW_PACKAGE[NEW_PACKAGE] NEW_PACKAGE --> LP[LEFT_PARENTHESIS] LP --> qualified_id[qualified_id] qualified_id --> COMMA[COMMA] COMMA --> STRING[STRING] STRING --> RP[RIGHT_PARENTHESIS] RP --> dots[...] </pre>
new_interface	<p>Creates a model element of type UML::Interface in package <i>qualified_id</i> with name <i>string</i>.</p> <pre> graph LR ip_new_interface --> NEW_INTERFACE[NEW_INTERFACE] NEW_INTERFACE --> LP[LEFT_PARENTHESIS] LP --> dots1[...] dots1 --> qualified_id[qualified_id] qualified_id --> COMMA[COMMA] COMMA --> STRING[STRING] STRING --> RP[RIGHT_PARENTHESIS] RP --> dots2[...] </pre>

new_interface_realization	<p>Creates n interface realization (inheritance) between ids.</p> <pre> ip_new_interface_realization → NEW_REALIZATION → LEFT_PARENTHESIS → → ID → COMMA → ID → RIGHT_PARENTHESIS → </pre>
external_task	<p>Presents a note to the reuser showing a message described in the <i>string</i>. Stops the reuse process until reuser hits the <enter> key.</p> <pre> ip_external_task → EXTERNAL_TASK → LEFT_PARENTHESIS → STRING → RIGHT_PARENTHESIS → </pre>
if statement	<p>Represents an if-then-else conditional statement. The condition has a text to prompt which path should be executed. Answering Y means the then path and N the else path.</p> <pre> if_statement → IF → LEFT_PARENTHESIS → condition → RIGHT_PARENTHESIS → → THEN → statement_block → ELSE → statement_block → </pre>

Table 6 - RDL enhancements

6. EXTENDING THE REUSETOOL

One of the features we aimed at when developing the ReuseTool was extensibility. Our past experiences with software projects demonstrate modelling and programming evolve over time and having a tool that can cope with such evolution would be a valuable contribution. As a result, instead of creating an immutable piece of software, we ended up using the concept of object-oriented frameworks throughout the ReuseTool's architecture to define a set of hotspots and an associated RDL Program. In this section we explain the rationale for the ReuseTool Framework and expose its major hotspots, the associated RDL Program and a reuse scenario based on handling UML Enumerations.

Enumerations are UML model elements to represent user-defined data types based on a set of literals (ANTLR, 2010). Each literal represents a concept from the application domain and can hide complicate computer based information. For example, computers typically represent a color as a number that indicates the amount of RGB (Red, Green and Blue) that should be combined to reproduce its appearance. Thus the color black can be seen as (0,0,0), while red as (255,0,0). In order to make the color information more palatable for developers not eager to work with numbers it's possible to define an enumeration can represent the set of possible colors as, Color = (BLACK, WHITE, RED), where each literal maps to a RGB number. Table 7 illustrates the look and feel of an UML Enumeration and its associated XML representation.


 <pre> classDiagram class Color { <<enumeration>> BLACK WHITE RED } </pre>	<pre> <packagedElement xmi:type="uml:Enumeration" xmi:id="_a" name="Colors"> <ownedLiteral xmi:id="_a1" name="BLACK" classifier="_a"/> <ownedLiteral xmi:id="_a2" name="WHITE" classifier="_a"/> <ownedLiteral xmi:id="_a3" name="RED" classifier="_a"/> </packagedElement> </pre>
---	--

Table 7 - Enumerations in UML

6.1 Extending the Language

RDL represents the major front-end for the ReuseTool given that RDL is used to describe the reuse process that is later translated into a process description. As a result, the first step towards making the ReuseTool extensible is to make RDL extensible so that framework developers can make use of new commands programmatically. Allowing new commands in RDL programs implies modifying the RDL grammar to represent the command's syntax and how it will be connected to the JPDL Code Generator. Fortunately the RDL grammar specifies a light-weight extension mechanism that allows defining new commands that must be executed without affecting the process flow. The restriction regarding the side effects on the process flow means the new command cannot introduce a way to bypass the course of execution already defined by the RDL program.

Figure 14 exposes the production rule for the non-terminal *ip_expression*. Recalling the Table 6 on RDL Enhancements (line Structured Commands), an *ip_expression* can take part in any statement, what makes it the perfect place to add new reuse commands with no workflow side-effects. As a result the first hotspot specified by the ReuseTool Framework is “Refining the non-terminal *ip_expression*.”

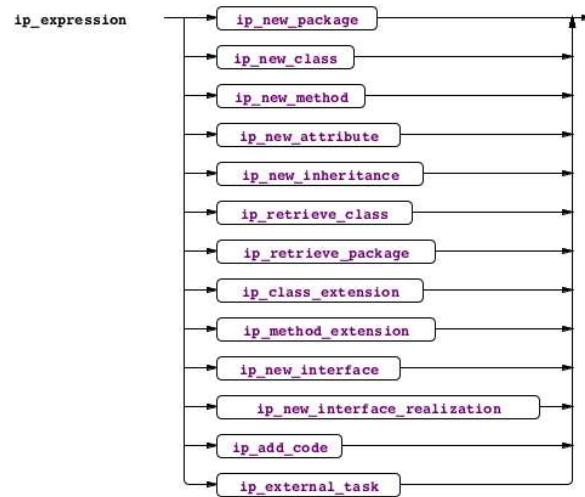


Figure 14- Non-terminal *ip_expression* and its associated production rule.

In order to extend the hotspot related to *ip_expression* the reuser must:

- Add a new entry in the *ip_expression* production rule to introduce the new non-terminal symbol.
- Define the new non-terminal symbol production rule to represent its syntax. Sometimes it's required to add several terminal and non-terminal symbols to properly represent the new command.
- Define the inline Java code to connect with the JPD L Code Generator.
- Re-generate the RDL Parser and RDL Lexer.

```

1. ip_expression : ... | ip_new_enumeration
2. ;
3.
4. ip_new_enumeration
5. : NEW_ENUMERATION LEFT_PARENTHESIS qualified_id COMMA STRING RIGHT_PARENTHESIS
6. {
7.     codeGen.addNewEnumeration($qualified_id.text , $STRING.text);
8. }
9. ;

```

Code 2- RDL Grammar for a new command.

Code 2 illustrates the piece of grammar required to define a new command to create UML Enumerations. Line 1 specifies the new entry with the *ip_expression* production rule to introduce the new non-terminal symbol *ip_new_enumeration*. Lines 4 through 9 specify the *ip_new_enumeration* production rule that will allow declaring an RDL code such as “*NEW_ENUMERATION(id,”label”*)”, where *id* is the enclosing package and *label* is the name for the new enumeration. It's also

important to observe how the non-terminal symbol is linked to the code generator using a snippet of Java code from lines 6 through 8. The *addNewEnumeration* method (Line 7) should be created at class *org.reusetool.jpdl.transformer.JbpmCodeGen*, which is part of the ReuseTool *CodeGenerator* component (see Section 3.1).

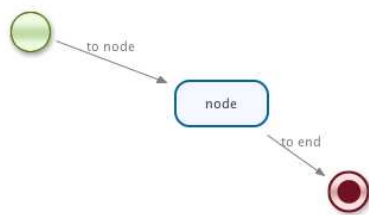
The last action to extend RDL is developing its associated lexer and parser. Fortunately we have adopted the compiler generator ANTLRWorks (ANTLR, 2010), which is capable of generating parsers and lexers automatically based on the language grammar.

6.2 Extending the Architecture

The ReuseTool's architecture is divided in two components: the Transformer and the Process Execution Machine. The former aims at generating JPDL code based on the RDL parser output while the latter executes the JPDL code to manipulate the UML Model. In order to make these two components flexible we have developed a set of classes to define the hotspots Extending the JPDL Code Generation and Extending Process Execution Machine.

HotSpot - Extending JPDL Code Generation

The Code Generator aims at transforming the RDL program into a JPDL process graph that is executable by the JBPM workflow engine. The process graph consists of a series of nodes that are connected to define the sequence in which the reuse actions occur. Table 8.a brings an image for a process graph containing the nodes *Start*, *Custom* and *End* to define a process graph that follows the sequence "*Start* -> *Custom* -> *End*". However the process graph cannot be manipulated graphically but by the use of JPDL (JBoss Process Definition Language), which is an XML representation as illustrated in Table 8.b.



(a) Process Graph

```

1. <process name="rdl_script"
   xmlns="http://jbpm.org/4.0/jpdl">
2.   <start name="start">
3.     <transition name="to node" to="node"/>
4.   </start>
5.   <custom class="JavaClass" name="node">
6.     <transition name="to end" to="end"/>
7.   </custom>
8.   <end name="end"/>
9. </process>
  
```

(b) JPDL Representation

Table 8 - JPDL Sample

In order to make the JPDL code generation flexible we have developed the interaction strategy described by the UML Sequence Diagram at Figure 15. The diagram illustrates how to combine instances of the classes *RDLParser*, *JBPMCodeGen* and *NewEnumerationJPDL* to generate JPDL code for the RDL command *NEW_ENUMERATION*. Note the diagram has two loop fragments. The top loop fragment starts when the *RDLParser* matches a valid command in the RDL program input stream. It then triggers the *addNewEnumeration()* method found in the *JBPMCodeGen* class to store an instance of the *NewEnumerationJPDL* class in the intermediate code array. Once all commands are parsed, the bottom loop iterates through the intermediate code array, retrieves all *NewEnumerationJPDL* instances and then generates JPDL code specific to the reuse action type.

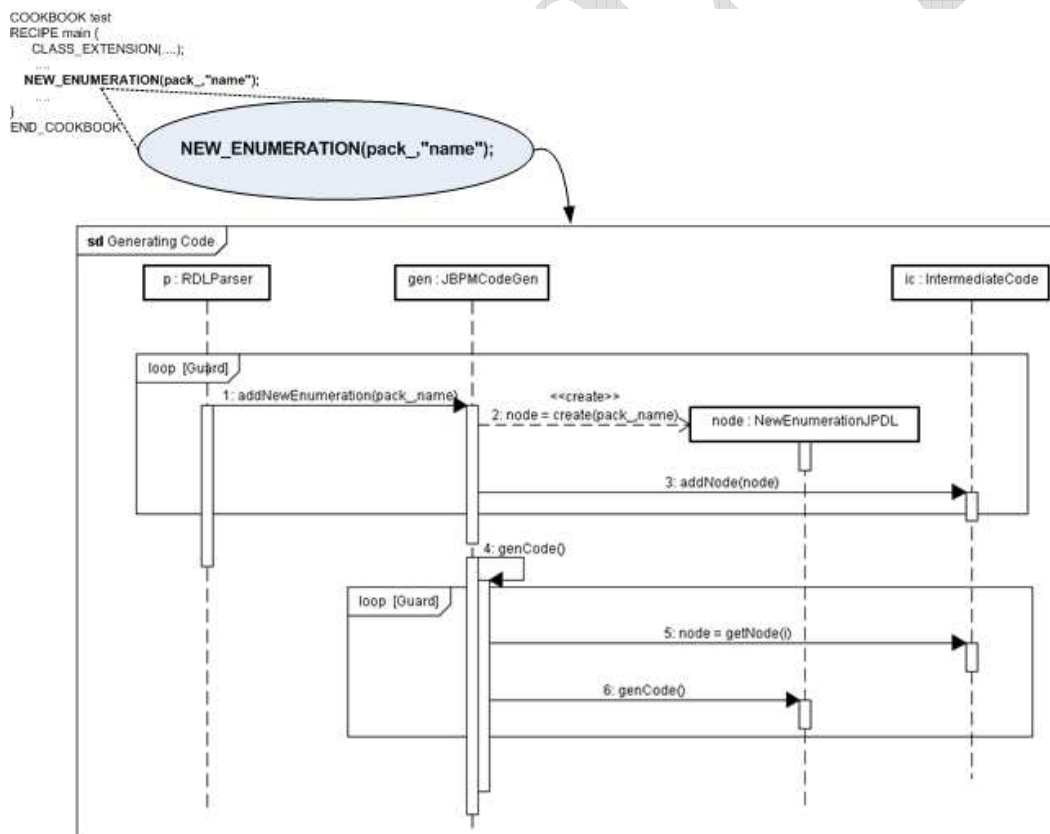


Figure 15- Sequence Diagram to Generate Code

The aforementioned flexibility rests on the fact that the class *NewEnumerationJPDL* and the method *addNewEnumeration()* used in the Figure 15 Sequence Diagram are actually placeholders to more generic representations, and they can be replaced by other pairs that correspond to any other RDL command. The class *NewEnumerationJPDL* is part of a hierarchy based on the class *JPDLCode*, which means any instance of a *JPDLCode* sub-class can be stored within the intermediate code array instead. Furthermore, the method *addNewEnumeration()* can be replaced by the method specified in the Java code snippet defined within RDL grammar that binds the RDL Parser and the Code Generator (see Code 2 Line 7). As a result, extending the Code Generation implies defining a new class following the *JPDLCode* hierarchy and implementing the related “*add*()*” method to add an instance of the new sub-class to the intermediate code array.

The *JPDLCode* class defines a set of helper services to emit the XML representation for a JPDL Custom node that will be part of the reuse process graph. A JPDL Custom node defines an action in the process graph that can be carried out by a Java class, which in our case implements the manipulation of a UML model element related to the reuse action. Code 3.a exposes the *NewEnumerationJPDL* class used to implement the *NewEnumeration* command while Code 3.b demonstrates the generated JPDL Custom node. Note the *NewEnumerationJPDL* class extends the *JPDLCode* class and uses the method *setJavaClassName* at Code 3.a Line 6, to indicate the qualified name *NewEnumerationHandler* (see Code 4) for the java class that will handle the UML model manipulation at runtime. The *JPDLCode* sub-class also defines two parameters to indicate the UML Package that will contain the enumeration (Code 3.a Line 4) and the enumeration name (Code 3.a Line 5). These parameters are used in the custom node as arguments passed to the constructor for the *NewEnumerationHandler* class (Code 3.b lines 2 through 9).

```

1. public class NewEnumerationJPDL extends
    JPDLCode {
2. public NewEnumerationJPDL
3.     (String package_, String enumName) {
4.     params.add(package_);
5.     params.add(enumName);
6.     setJavaClassName
7.         ("NewEnumerationHandler");
8.     setJavaMethodName("newEnum");
9. }

```

(a)

```

1. <custom class="NewEnumerationHandler" name="X"
2.   <constructor>
3.     <arg>
4.       <string value='package_' />
5.     </arg>
6.     <arg>
7.       <string value='enumName' />
8.     </arg>
9.   </constructor>
10. <transition name="to Y" to="Y" />
11.</custom>

```

(b)

Code 3 - Code to Configure the NewEnumeration command.

Hotspot – Extending the Process Execution Machine

Once the JPDL code is generated it's time to add new features to the Process Execution Machine component so that it becomes capable of manipulating the new UML Model element properly. It's important to remind the Process Execution Machine is developed around a workflow engine that has been tailored with some Java classes to deal with UML model elements. Thus, using the same rationale of the Code Generation component, we have devised a set of classes that facilitate extending the process machine through sub-classing.

The class diagram in Figure 16 illustrates some model elements that compose the Process Execution Machine component. We have developed the class *RDLCommand* to implement a set of services to facilitate executing the RDL command within a process environment. For instance, it allows accessing a shared symbol table that contains runtime elements such as program variables, UML model elements, etc. In addition the *RDLCommand* class declares an abstract method named *internalExecute()* that should be redefined in a sub-class to provide specific implementation to the command behaviour. On the other hand, the interface *ExternalActivityBehaviour* is provided by the workflow engine library to promote integration between a Java object and the engine.

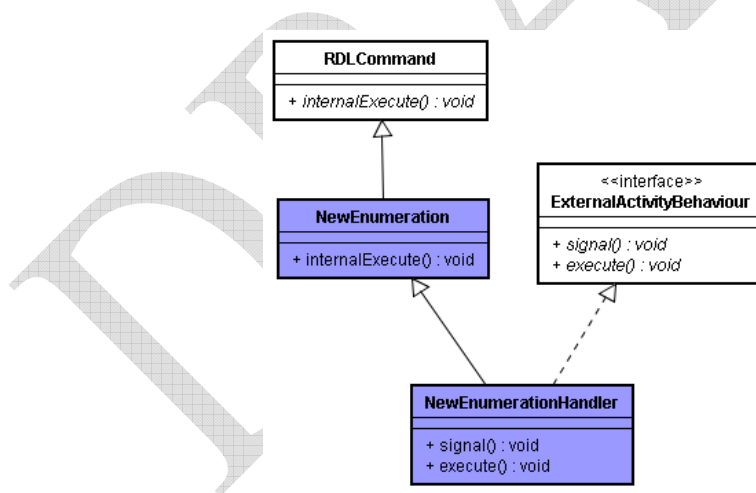


Figure 16 - Process Execution Machine Component.

The classes *NewEnumeration* and *NewEnumerationHandle* (darkened) are also shown in Figure 16 to represent classes that implement the command behaviour and the integration with the workflow engine respectively. Furthermore these two classes illustrate how the Process Execution Machine hotspot is extended through redefining proper methods in sub-classes. For example the method

internalExecution() at the *NewEnumeration* class redefines the behaviour of the equivalent method declared in super-class *RDLCommand*. Code 4 illustrates the final implementation for the class *NewEnumeration* containing its *constructor* and the *internalExecute* methods. The constructor is described from lines 6 through 10 and it essentially stores the parameters in the class current instance. It is important to remind these parameters are the same used in the JPDL custom node configuration (see Code 3.b lines 2 through 9).

The method *internalExecute()* described from Line 11 through Line 25 is programmatically aware of the command syntactic and semantic and therefore can implement the sheer UML element manipulation to realize the behaviour of the RDL command. For instance, at Line 12, the method uses that symbol table object to gain access to a reference to the UML package element which name matches the *PACKAGE_PARAM* parameter. This method also uses a library that is responsible for manipulating the XML structure that represents the UML model in memory (see Code 4 Line 23).

```

1. public class NewEnumeration extends RDLCommand {
2.
3.     protected String PACKAGE_PARAM ;
4.     protected String ENUM_NAME_PARAM ;
5.
6.     public NewEnumeration(String pName, String name) {
7.         super("NEW_ENUMERATION");
8.         PACKAGE_PARAM = pName;
9.         ENUM_NAME_PARAM = name;
10.    }
11.    protected Object internalExecute(){
12.        Package umlPackage = (Package) symbolTable.get(PACKAGE_PARAM);
13.        String enumName = ENUM_NAME_PARAM;
14.        // Trim excess quotes and spaces
15.        enumName = SupportLib.cleanString(enumName);
16.        if (enumName.startsWith("?")) {
17.            enumName = SupportLib.promptUser("Type the Enumeration Name : ");
18.        }
19.        if (!SupportLib.isValidID(enumName)){
20.            System.out.println("Enumeration Name Invalid.");
21.            return null;
22.        }
23.        return UMLModelRAM.newEnumeration(umlPackage,enumName);
24.    }
25. }

```

Code 4- Class *NewEnumeration*.

Figure 16 also contains the class *NewEnumerationHandler*. This class extends the *NewEnumeration* class and implements the *ExternalActivityBehaviour* interface to provide a smooth integration between the code that implements the NEW_ENUMERATION command and the workflow engine. The integration happens in two levels. Firstly the name

“*NewEnumerationHandler*” was used when extending the JPDL Code Generation hotspot at Code 3.a Line 6 to indicate the name of the class that will realize the command at runtime. Secondly, the implementation for methods *signal()* and *execute()* handle the exact moment (Code 4 Line 9) when the JPDL custom node is executed and re-route the execution flow to the *NewEnumeration.execute()* method (Code 4 Line 13) respectively.

```

1. public class NewEnumerationHandler extends NewEnumeration implements
   ExternalActivityBehaviour {
2.
3.     public NewEnumerationHandler(String pName, String name) {
4.         super(pName, name);
5.     }
6.
7.     public void signal(ActivityExecution execution, String signalName,
8.         Map<String, ?> parameters) throws Exception {
9.         execution.take(signalName);
10.    }
11.
12.    public void execute(ActivityExecution execution) throws Exception {
13.        this.execute();
14.    }
15. }

```

Code 5 - NewEnumerationHandler implementation.

6.3 The RDL Program

Sections 6.1 and 6.2 have illustrated the rationale for the ReuseTool extension process. However, in order to demonstrate the tool’s bootstrap approach, this section exposes the commented RDL Program that is capable of extending the ReuseTool itself (Code 6).

```

1. COOKBOOK ReuseTool
2.
3. RECIPE main
4. {
5.     /// Extending Hotspot - Refining the non-terminal ip_expression
6.     EXTERNAL_TASK("Modify the ip_expression production rule in the RDL.g grammar.");
7.     EXTERNAL_TASK("Re-generate the RDL Parser and RDL Lexer.");
8.
9.     /// Extending Hotspot - JPDL Code Generation
10.    // Add CodeGenerator Method to JBPMCodeGenClass
11.    NEW_METHOD(JbpmCodeGen, "?");
12.
13.    // Create JPDL Class to handle code generation at package
    org.reusetool.jpdl.transformer;
14.    jpdlClass = CLASS_EXTENSION(JPDLCode, org.reusetool.jpdl.transformer, "?");
15.    // Define the constructor
16.    NEW_METHOD(jpdlClass, "?");
17.
18.    /// Extending Hotspot - Process Execution Machine
19.    // Create Class to execute the command at package
    org.reusetool.pem.reuseactions.
20.    cmdClass = CLASS_EXTENSION(RDLCommand, org.reusetool.pem.reuseactions, "?");
21.    // Define the constructor
22.    NEW_METHOD(cmdClass, "?");
23.    // Define method internalExecute()

```

```

24.         m = METHOD_EXTENSION(RDLCommand,cmdClass,internalExecute);
25.
26.         // Create Class to handle JBPM Call the command at  at package
org.reusetool.jbpm.pem.reuseactions;
27.         engineClass = CLASS_EXTENSION(cmdClass,org.reusetool.pem.reuseactions,"?");
28.         // Define the constructor
29.         NEW_METHOD(engineClass,"?");
30.         // Implement ExternalActivityBehaviour
31.         NEW_REALIZATION(cmdClass,ExternalActivityBehaviour);
32.         // Define method signal()
33.         m = METHOD_EXTENSION(ExternalActivityBehaviour,engineClass,signal);
34.         ADD_CODE(engineClass,m,"execution.take(signalName)");
35.         // Define method execute()
36.         m = METHOD_EXTENSION(ExternalActivityBehaviour,engineClass,execute);
37.         ADD_CODE(engineClass,m,"this.execute()");
38.         END_COOKBOOK

```

Code 6 - ReuseTool Bootstrap Instantiation Program

7. RELATED WORK

In this section, we describe existing work on software reuse related to framework instantiation, product lines and model-driven architectures.

In (Johnson, 1992) the authors have proposed a structured specification to support framework instantiation. They have introduced a template called Cookbook applied to a reuse document expressed in natural language, which can be difficult to follow because of the lack of a formal underlying approach. Moreover, Cookbooks cannot be processed automatically, possibly leading to inconsistencies in their definition. As an extension to Cookbooks, Hooks (Froehlich, 1997) also provided a template for framework instantiation assistance which shares similar problems. Fred (Hautamäki, 2006) (Viljamaa, 2002) can be seen as a revamped version of CookBooks and Hooks where the authors define a Pattern based approach to specify goals that are related to framework instantiation.

In the area of framework instantiation guidance, Active Cookbooks (Ortigosa, 1999) advocate the use of Software Agents to execute instantiation plans. The main issue with the HiFi (Ortigosa, 1999) approach, which is an extension of Active Cookbooks is that it introduces non-standard notations such as TOON (Ortigosa, 1999) into the application development process. Such non-standard approaches can add an extra burden to framework development. In OBS (Cechticky, 2003) the authors used a generative approach for framework instantiation that shares some of the features of our previous work (Oliveira, 2002). However, the OBS approach is based on ready-to-use blackbox

frameworks, which restricts instantiation processes to component configuration, and does not focus on customization.

Product Line Architectures also lead to approaches to support generation of applications from pre-defined software components that can be configured based on a decision model (Atkinson, 2001). In Gomaa (Gomaa, 2004) the author has presented a technique to develop Software Product Lines using UML diagrams and Feature Models. However, the lack of tool support causes difficulties for systematic adoption. Another initiative based on Features model is the Generative Programming (GP) approach described in (Czarnecki, 2004) (Antkiewicz, 2009), where the author proposes the development of Domain Specific Modeling Languages (DSML) as a means to generate programs from high-level specifications and transformations. Alfama (Santos, 2008) also leverages on DSMLs to foster framework instantiation. Although we share some underlying principles with the Generative approaches, we believe DSMLs should be anchored on a robust notation such as UML to avoid using a multitude of languages to define a single system.

WebML (WebML, 2010) and WebRatio (WebRatio, 2010) are approaches based on UML-based Domain Specific Language that guides developers through a set of pre-defined forms to collect application-specific information to create Web Applications. In the WebML/WebRatio approach the configuration knowledge is hard-coded into the tool, which complicates its extension to incorporate new types of reusable assets and extension point types. The work on Design Fragments (Fairbanks, 2006) encapsulates the framework extension points as part of goals the reuser must fulfil. Goals describe all the code elements that should be manipulated during instantiation with examples and also expose some related framework underlying concepts. Design Fragments is a code-based approach to framework instantiation whilst ours is model-based.

OMG's Model Driven Architecture (MDA) (MDA, 2010) is also an attempt to systematize reuse using UML (Unified Modelling Language), OCL (Object Constraint Language) (OCL, 2010) and software transformations. The MDA approach can use modelling languages such as UML to represent software in a platform-independent manner and have transformation programs to generate code or platform dependent models. Our approach shares many of the MDA characteristic but instead of using a strict deterministic (Czarnecki, 2003) approach based on transformation languages such as ATL (Atlas Transformation Language) (ATL, 2010), we use an interactive process in which the reuser can leverage on.

8. CONCLUSIONS AND FUTURE DIRECTIONS

In this paper we present the ReuseTool, which is an infrastructure to facilitate the instantiation of Object Oriented Frameworks. The tool is based on the interactive model-to-model transformation (Czarnecki, 2003) approach, where the framework model is tailored with application specific increments to create the application model. The interactive transformation process is specified with RDL (Reuse Description Language) (Oliveira, 2007) and executed by a bespoke extension to JBPM workflow engine (JBoss, 2010). By an interactive transformation we mean the instantiation process can request the reuser information on how to adapt an extension point at reuse-time.

The ReuseTool operates on UML models and was implemented using the Eclipse UML2 plugin (UML2, 2010). We have also used ANTLR (Antlr, 2010) to implement the RDL-to-JPDL (JPDL, 2010) compiler. It is important to mention the ReuseTool is based on an extensible architecture that is itself an object oriented framework and can be adapted to handle other types of reusable artifacts than UML models. In fact, as part of our future work, we plan to devise a concept of Reuse Contracts that integrates a Reusable Artifact Model, a Reuse Process Specification and Reuse Constraints as an extension to the Reusable Asst Specification (RAS) (RAS, 2010). Conceptually, the Reuse Contract will allow the representation of reuse process for reusable assets such as Aspects (Santos, 2007) and other Software Process concepts. We also plan to conduct an experiment based on (Juristo, 2001) to collect valuable feedback from end-users.

Acknowledgements

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the National Brazilian Research Council (CNPq).

REFERENCES

(Santos, 2007) A. Santos, A. Lopes, K. Koskimies, Framework specialization aspects, Proceedings of the 6th international conference on Aspect-oriented software development, AOSD 2007, 14-24.

- (Bosch, 1999) Object-Oriented Frameworks: Problems & Experiences J. Bosch, P. Molin, M. Mattsson, PO Bengtsson, M. Fayad. in Building Application Frameworks, M. Fayad, D. Schmidt, R. Johnson (eds.), John Wiley, ISBN 0-471-24875-4, pp. 55-82, 1999
- (Hautamäki, 2006) F. Hautamäki and K. Koskimies, "Finding and documenting the specialization interface of an application framework," *Software: Practice and Experience* 36, no. 13 (11, 2006): 1443-1465.
- (Viljamaa, 2002) A. Viljamaa and J. Viljamaa, "Creating framework specialization instructions for tool environments," in *Proceedings of the Nordic Workshop on Software Development Tools and Techniques*, 2002.
- (Hou, 2005) D. Hou, K. Wong, and H. James Hoover. What can programmer questions tell us about frameworks? In *IWPC*, pages 87–96, 2005.
- (Fayad, 1999) M. Fayad, D. Schmidt, and R. Johnson, *Building Application Frameworks*. John Wiley, 1999.
- (Kirk, 2005) D. Kirk, M. Roper, and M. Wood. Identifying and addressing problems in framework reuse. In *IWPC*, pages 77–86, 2005.
- (Pree, 1995) W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison Wesley/ACM Press, 1995
- (Markiewicz, 2001) M. E. Markiewicz, C. Lucena, Object oriented framework development, *ACM Crossroads*, Volume 7 Issue 4, June 2001
- (Mattsson, 2000) M. Mattsson, J. Bosch. Stability Assessment for Evolving Object-Oriented Frameworks, *Journal of Software Maintenance*, Vol. 12, No. 2, pp. 79-102, 2000.
- (J2EE, 2010) Documentation at <http://www.oracle.com/technetwork/java/javase/overview/index.html>, accessed July 2010.
- (DotNet, 2010) Documentation at <http://www.microsoft.com/net/>, accessed July 2010.
- (Hibernate, 2010) Documentation at <http://www.hibernate.org/>, accessed July 2010.
- (JUnit, 2010) Documentation at <http://www.junit.org/>, accessed July 2010.
- (Eclipse, 2010) Documentation at <http://www.eclipse.org/>, accessed July 2010.
- (Struts, 2010) Documentation at <http://struts.apache.org/>, accessed July 2010.
- (MooTools, 2010) Documentation at <http://mootools.net/>, accessed July 2010.
- (Atkinson, 2001) Atkinson et. al., *Component-Based Product Line Engineering with the UML*, Addison-Wesley, September 2001
- (Oliveira, 2004) T. C. Oliveira, P. Alencar, C. Lucena, D. Cowan, Software Process Representation and Analysis for Framework Instantiation. *IEEE Transactions on Software Engineering* 30(3): 145-

159 (2004).

(Oliveira, 2007) T. C. Oliveira, P. Alencar, C. Lucena, D. Cowan, RDL: A language for framework instantiation representation. *Journal of Systems and Software* 80(11): 1902-1929 (2007)

(Krueger, 1992) C. Krueger, Software Reuse, *ACM Computing Surveys (CSUR) Surveys Homepage* table of contents archive Volume 24 Issue 2, June 1992.

(Georgakopoulos,1995) D. Georgakopoulos, M.F. Hornick, A.P. Sheth: An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases* 3(2): 119-153 (1995)

[WorkflowThesis]

(Workflow Patterns, 2010) Documentation at <http://www.workflowpatterns.com/>, accessed July 2010.

(YAWL, 2010) Documentation at www.yawlfoundation.org/, accessed July 2010.

(JPDl, 2010) Documentation at <http://docs.jboss.org/jbpm/v3/userguide/jpdl.html>, accessed July 2010.

(JBoss, 2010) Documentation at jboss.org, accessed July 2010.

(GEF, 2010) Documentation at <http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html>, accessed July 2010.

(Shapes, 2010) Documentation at <http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html>, accessed July 2010.

(Jhotdraw, 2010) Documentation at www.jhotdraw.org/, accessed July 2010.

(Gomaa, 2004) H.Gomaa, *Deseigning Software Product Lines with UML*, Addison Wesley, July 2004.

(APL, 2010) Documentation at <http://www.sei.cmu.edu/productlines/ppl/>, accessed July 2010.

(REMF, 2010) Documentation at www.cs.uwaterloo.ca/research/tr/2005/25/CS-2005-25.pdf, accessed July 2010.

(Juristo, 2001) N. Juristo, A. Moreno, *Basics of Software Engineering Experimentation*, Springer; 1 edition (February 28, 2001).

(Antlr, 2010) Documentation at www.antlr.org, accessed July 2010.

(Johnson, 1992) Johnson, R., *Documenting Frameworks Using Patterns*, Proceedings of OOPSLA'92, ACM/SIGPLAN, New York, 1992.

(Froehlich, 1997) Froehlich, G., Hoover, H.J., Liu L. and Sorenson, P.G. Hooking into Object-Oriented Application Frameworks, Proc. 19th Int'l Conf. on Software Engineering, Boston, May 1997, 491-501.

(Ortigosa, 1999) Ortigosa, A., Campo, M., Smartbooks: A Step Beyond Active-Cookbooks to Aid in Framework Instantiation, Technology of Object-Oriented Languages and Systems 25, IEEE Press, June 1999.

(Cechticky, 2003) V. Cechticky, E. Chevalley, A. Pasetti, W. Schauffelberger, A Generative Approach to Framework Instantiation, Erfurt, Germany, Proceedings of GPCE, pp. 267-286, LNCS 2830 OOPSLA, 2003.

(Oliveira, 2002) Oliveira, T.C., Alencar, P., Cowan, D. : Towards a declarative approach to framework instantiation Proceedings of the 1st Workshop on Declarative Meta-Programming (DMP-2002), September 2002, Edinburgh, Scotland, p 5-9

(Gomaa, 2004) Gomaa, H. ,Designing Software Product Lines with UML, Addison Wesley Object Technology Series, July 2004.

(Czarnecki, 2004) Czarnecki, K.. Overview of Generative Software Development. In Proceedings of Unconventional Programming Paradigms (UPP) 2004, 15-17 September, Mont Saint-Michel, France, Revised Papers, volume 3566 of Lecture Notes in Computer Science, pages 313–328. Springer-Verlag, 2004.

(WebML, 2010) Documentation at www.webml.org. accessed in July 2010.

(WebRatio, 2010) Documentation at <http://www.webratio.com/>, accessed July 2010.

(Antkiewicz, 2009) M. Antkiewicz, K. Czarnecki, M. Stephan, Engineering Framework-Specific Modeling Languages, IEEE Transactions on Software Engineering 35(6): 795-824 (2009).

(Santos, 2008) A. Santos, K. Koskimies, and A. Lopes, “Automated Domain-Specific Modeling Languages for Generating Framework-Based Applications,” in 2008 12th International Software Product Line Conference , 149-158.

(Fairbanks, 2006) G. Fairbanks, D. Garlan, and W. Scherlis, “Design fragments make using frameworks easier,” in Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, 2006, 88.

(MDA, 2010) Documentation at www.omg.org/mda, accessed July 2010.

(OCL, 2010) Documentation at <http://www.omg.org/technology/documents/formal/ocl.htm>, accessed July 2010.

(Czarnecki, 2003) K. Czarnecki and S. Helsen, “Classification of model transformation approaches,” in Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, 2003.

(ATL, 2010) Documentation at <http://www.eclipse.org/atl/>, accessed July 2010.

(UML2, 2010) Documentation at www.eclipse.org/uml2/, accessed July 2010.

(RAS, 2010) Documentation at <http://www.omg.org/technology/documents/formal/ras.htm>, accessed July 2010.

DRAFT