UML2Ruby Translator (RML)

Term Project for CS428

Professor: Darko Marinov
Spring 2014

RML Team
Kevin Boice (boice3)
Holly Decker (hdecker2)
David Overgaard (overgaa2)
Jonathan Gill (jagill2)

Staff Supervisor:
Cosmin Radoi (cos)

Table of Contents

Project Description

There are several programs that translate from UML to source code for other languages but very few exist for Ruby. Ruby was introduced in 1995 and is now one of the top 10 most popular languages and new tools are needed to support its development. As a way to learn both Ruby and UML, this team decided to develop the UML2Ruby Eclipse plugin that translates UML diagrams into Ruby.

The source code for the UML2Ruby project can be found on its GitHub page and is released under the BSD License. The software is also available as a plugin for end user installation through the Eclipse Marketplace.

Process

The team followed a modified XP process during development of our software. The general process was agreed upon before development began, and further fleshed out after the first development iteration. The details are explained below.

Iterative development

The user stories that comprised the project were broken down into six iterations. Each story was assigned a T-Shirt size (S,M,L,XL, etc) which represented the estimated effort and mapped back to units of effort. The actual quantity of effort was recorded at the end of each iteration. After the first two iterations, we began to measure our iteration velocity. This was helpful in planning future iterations, especially when we were planning a three week iteration over Spring Break. The team met twice a week (Wednesday evenings and Sunday afternoons) to discuss progress, help others out with issues, and review one another's code.

Refactoring

The entire team was new to coding in Acceleo's Model to Text Language (MTL). In the beginning, the Acceleo code reflected that. However, we often found that one of the team members discovered a different mechanism for parsing or accessing the same data in a more succinct manner. As these tidbits were shared with the team during our meetings, the rest of the team often adapted their code to follow this new code standard. Thus, the code was refactored over time to be more consistent and easier to read.

Testing

Development of each user story was a three-stage process. First, a UML diagram was created that included the set of components that we wished to turn into auto-generated Ruby code. Second, a Ruby unit test was written using Ruby's Test::Unit framework and TUXML. The unit tests usually contained a combination of running the code that was generated and looking for certain lines in the output to match expectations. The unit tests also ensured that the code was valid. Lastly, Acceleo code was written to parse the UML model and generate Ruby code. Once the unit tests were passing, we ensured that there were no regressions through the use of Jenkins, a continuous integration server. Only after passing all tests in Jenkins would a pull request be submitted for the code. The following activity diagram describes the development and testing process followed by all team members:

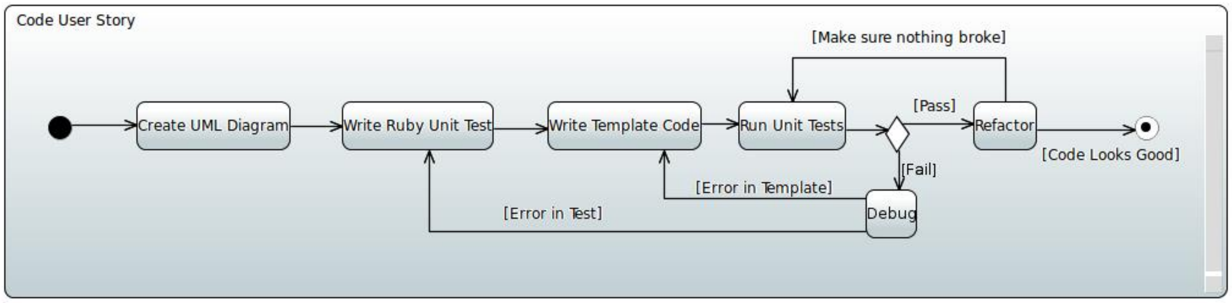

Figure 1: User Story coding process

Collaborative Development

For this project, we had four team members spread across the United States, so the team paired as much as schedules would allow. Extensive use was made of GitHub's pull request system to review code developed in feature development branches before merging it to the master branch. When pairs did pair-program together, Google Hangouts was used as the medium to share screens and communicate over the Internet. The process for pull requests and reviews can be seen in the following activity diagram:

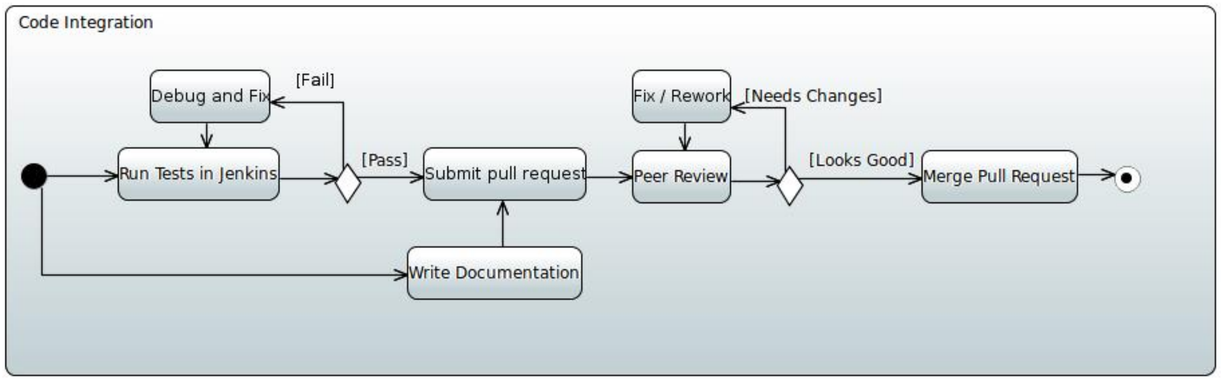

Figure 2: Code integration

Requirements & Specifications

Use Cases

The overarching use case of our project was to convert UML Diagrams to Ruby code. We implemented the main functionality of both Class and Activity diagrams as described in the use case diagram in Figure 3.

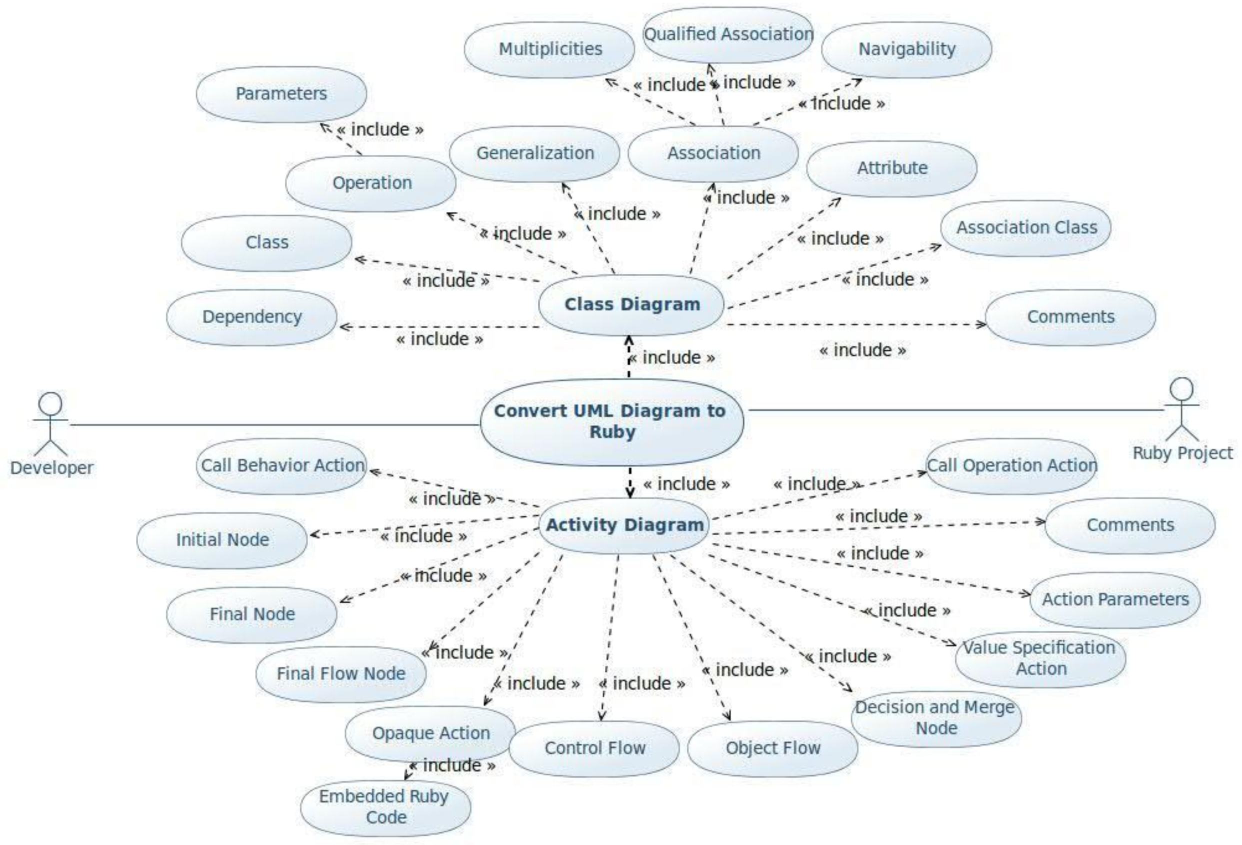

Figure 3: Use Case Diagram

Implemented User Stories

Please refer to the closed issues on the uml2Ruby GitHub page and the product backlog for details on the implemented stories.

Architecture & Design

Overview

The workflow for using UML2Ruby as an Eclipse plugin has three components; drawing the UML model, extracting the model data, and translation into Ruby. The first two components are implemented with third party tools. Drawing UML models is done with an Eclipse plugin named Papyrus which saves the user's diagram as a .uml file. Extracting the data from the .uml file is done with an open source code generator named Acceleo. The final component of formatting this information into Ruby code is the focus of this project.

The translation of model data to Ruby code is organized by UML diagram type. How to format the code for each diagram type is defined by a 'model to text language' (.mtl) file. Each of these files use a combination of Acceleo as well as service calls to Java called queries. After the Ruby code has been generated, Java is used to do final cleanup formatting.

Currently UML2Ruby supports class and activity diagrams. Each of the .mtl files focus on the generation of one Ruby file at a time. Class diagrams create a new file for each class. Activity diagrams create a new file for each associated class as well as a file for each activity and sub-activity. The starting point for the generation of a given file is defined in generate.mtl which then calls the template corresponding to the correct diagram type. The templates for Class Diagrams are in class.mtl. The templates for Activity Diagrams are in activity.mtl.

Supporting Java queries are used for more powerful data manipulation than is easily done with Acceleo. These Java methods are defined in the file AcceleoHelperQueries.java and are associated with the templates in queryTemplates.mtl. See Acceleo documentation on queries in order to regenerate the queryTemplates.mtl file if any interface changes are made to methods or new methods are added.

The post generation formatting to remove duplicate requires can easily be extended to perform other simplistic 'code clean-up' is a method named `formatFile` defined in Generate.java. This method is called on each of the generated files from the main method and is the last step in creating the source files.

Unit and system testing is done on the generated Ruby code to validate the supported features. These tests are all included in the 'testing' folder and make use of Ruby's test framework 'test-unit.' In order to automate and view these tests with Jenkins (continuous integration) a third party tool named 'tuxml' is used to generate a standard .xml file format. To generate all of the test data from a single command, each test is defined in run_all_tests.sh which is executed by automation after each build.

Class Diagram

The class diagram below, in Figure 4, describes the structure of the project source code.

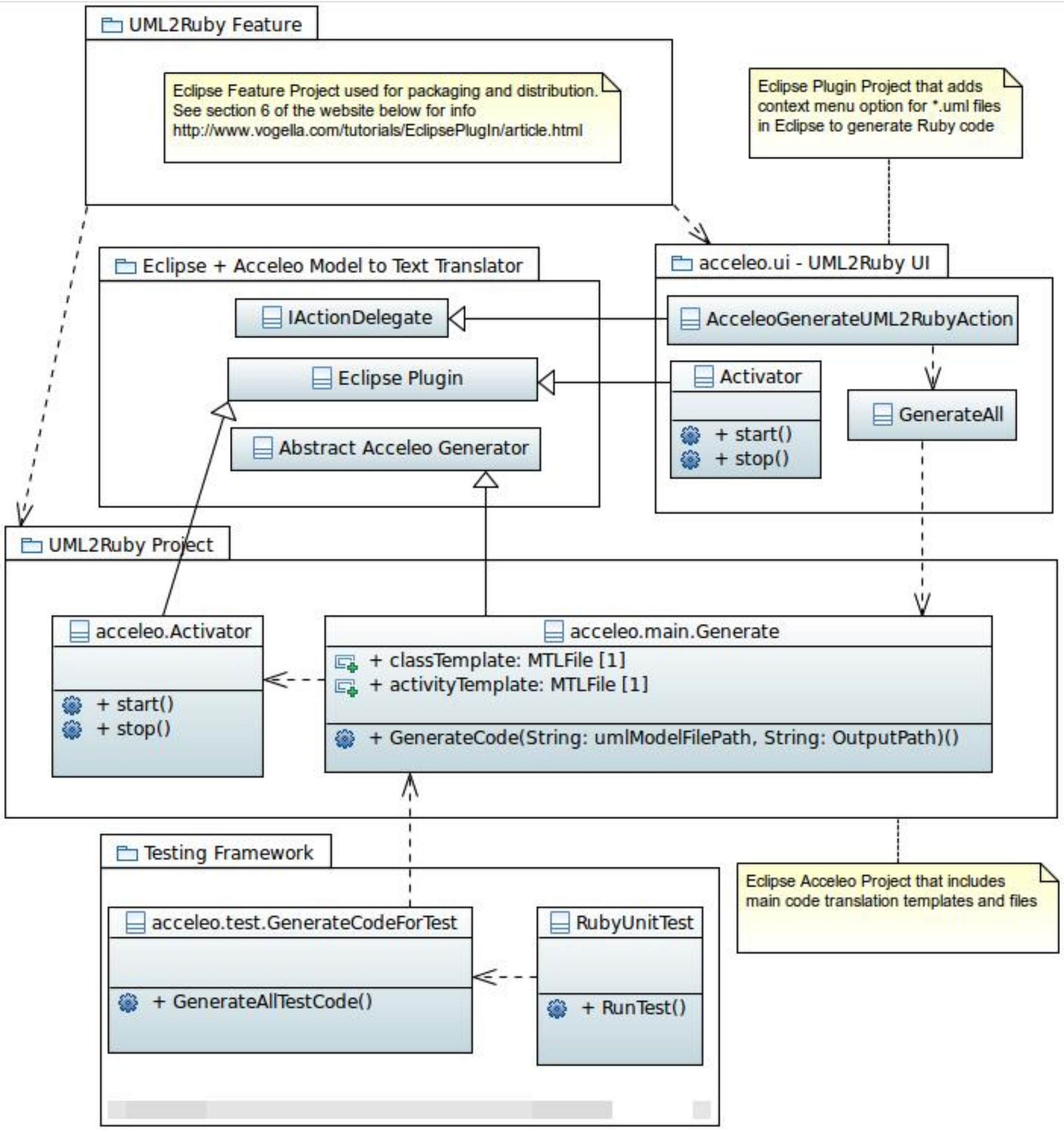
UML2Ruby Feature
Eclipse Feature Project used for packaging and distribution.
See section 6 of the website below for info
http://www.vogella.com/tutorials/EclipsePlugIn/article.html
Eclipse Plugin Project that adds context menu option for *.uml files in Eclipse to generate Ruby code
Eclipse + Acceleo Model to Text Translator
IActionDelegate
Eclipse Plugin
Abstract Acceleo Generator
acceleo.ui - UML2Ruby UI
AcceleoGenerateUML2RubyAction
Activator
+ start()
+ stop()
GenerateAll
UML2Ruby Project
acceleo.Activator
+ start()
+ stop()
acceleo.main.Generate
+ classTemplate: MTLFile [1]
+ activityTemplate: MTLFile [1]
+ GenerateCode(String: umlModelFilePath, String: OutputPath)()
Testing Framework
acceleo.test.GenerateCodeForTest
+ GenerateAllTestCode()
RubyUnitTest
+ RunTest()
Eclipse Acceleo Project that includes main code translation templates and files

Figure 4: Class Diagram

Sequence Diagram

The sequence diagram below, in Figure 5, describes the interactions of the classes when the UI plugin is called by selecting the 'Generate Ruby Code' context menu option for a *.uml file in Eclipse.

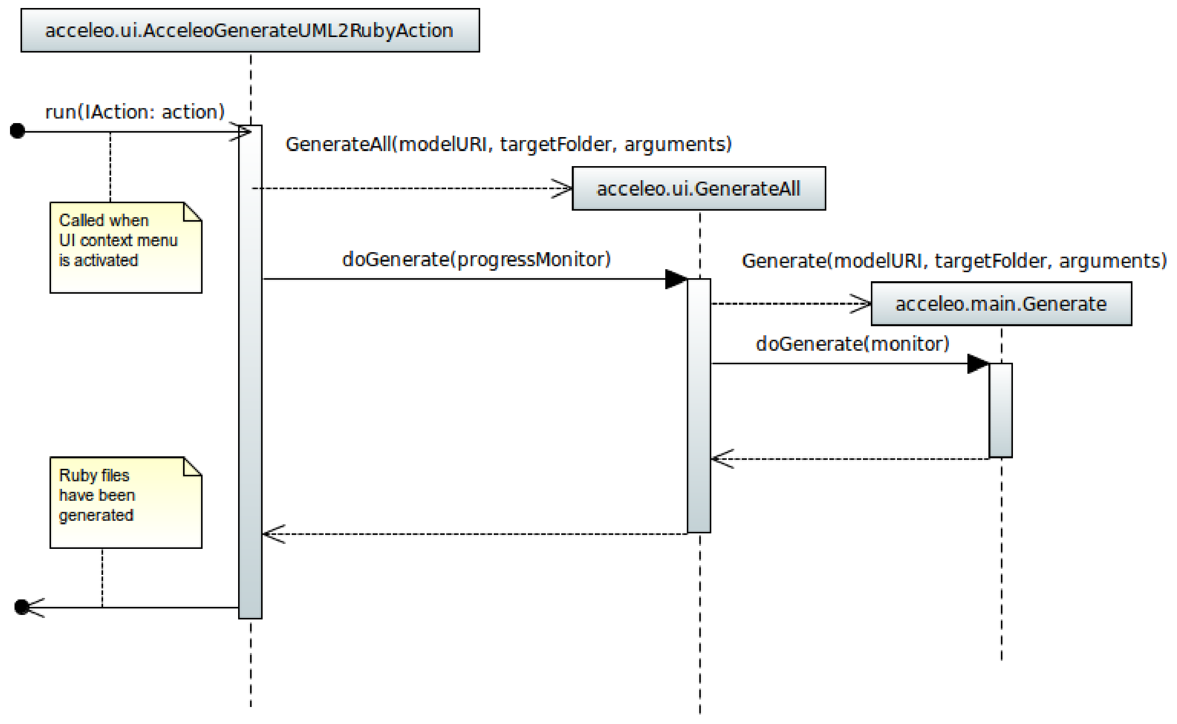
acceleo.ui.AcceleoGenerateUML2RubyAction
run(IAction: action)
Called when UI context menu is activated
GenerateAll(modelURI, targetFolder, arguments)
acceleo.ui.GenerateAll
doGenerate(progressMonitor)
Generate(modelURI, targetFolder, arguments)
acceleo.main.Generate
doGenerate(monitor)
Ruby files have been generated

Figure 5: Sequence Diagram

Future Plans

David's Personal Reflections

Since we're all working professionals following a set of rules for development, working together was not an issue. The majority of us took CS 427 together and were preconditioned to use an Agile development method. Following an XP like process we collaborated effectively. Continuous integration with Jenkins was crucial for our parallel development and caught several integration errors. Development seems especially worthwhile when creating a new product that is in an easy to deploy environment such as Eclipse.

I learned that a language expert such as one described by Keith Brooks can be an extremely useful resource to a development team since the majority of our time was spent figuring out how to do something in Acceleo not the logic behind it. I also learned that given the proper research even a daunting task can be made feasible. Extending existing projects was key to the success of our project.

Holly's Personal Reflections

Going into this project, yes I had previous experience with the XP development process from CS427, and had worked with several members of the team previously, but the project idea itself as well as the languages and tools it used (i.e. Ruby, Acceleo, Papyrus, mtl code) was entirely new. I learned a lot working on this project and feel that by following the XP practices we agreed to at the beginning of the term, and with such disciplined team members to begin with anyway, the team made great progress on supporting both the Class and Activity diagrams to be translated into Ruby.

If there was more time to work on this project, the Java code integration that was discovered and used in our last sprint would be beneficial to use throughout more of the code-base. This is due to the fact that there is a learning curve to working with the mtl code, and trying to apply algorithms with this code is difficult. Being able to transfer the needed objects over to Java and then to manipulate them in a familiar, higher-level language, and pass the results back to the mtl code, takes away much of the learning curve as well as provides the opportunity to do more, I think.

Jon's Personal Reflections

I came into this project familiar with coding in Ruby, and somewhat familiar with UML. Although I learned a bit more about each, I think that my major takeaways were related to testing. As a security professional, I am much more of a hacker (literally) than a developer. Most of my code is written in scripting languages for one-time use. I found that the unit test framework hooked up with Jenkins was easy to use and prevented many regressions. I had used GitHub in the past, but this project helped me gain the experience to use it collaboratively with other developers.

As far as the future of this project, I have some concerns with its viability. I personally found that using Papyrus as a UML editor was clunky and slow. I am not sure how many users would turn to a different editor, which may or may not export raw UML. Although I had some frustrations with Acceleo development initially, it grew on me over time. If the tool were to grow a user base, I would not have any issues contributing time to bug fixes or feature development.

Kevin’s Personal Reflections

I enjoyed working with the group on this project. Its impressive what a distributed team can accomplish now with modern tools and committed individuals. Combining Google Hangouts, GitHub, and Jenkins worked well for us. The project itself ended up being a good way to learn about UML and understand what it meant to actual code. I’ve started to use more of the diagrams at work, as we’re beginning a new project and are working to describe the system. As usual, the diagrams work well some of the time, and other times its easier to use a less structured method.

That is kind of the benefit and difficulty with UML tools. The well-defined structure enables applications like code generation, but sometimes the structure is too restrictive to concisely describe a system. The tools also tend to be built around the “System Model” versus the “User Model”. It would be interesting to work on some intro UML software that might make common use cases more accessible, versus providing an exhaustive interface to the UML standard.

In the near term, I will probably work on maintaining this software with updates for new Eclipse versions, bugs fixes, and investigations into other UML tool compatibility.