xored software

xored software, Inc.
6/1 Lavrentieva street
Novosibirsk, Russia 630090

Phone +7 383 330 2957
Fax +7 383 332 4054
http://www.xored.com

Eclipse and Dynamic Languages

Dynamic Languages Toolkit for Eclipse Platform

Chapter 1

Dynamic Languages Toolkit

Generic Model and Common Infrastructure for Dynamic Languages

Background

Eclipse Platform is used as a base for development environments for many languages. For the moment of this writing there are 55 plug-ins/products registered in Languages category on http://www.eclipse-plugins.info. Some of them are related to strongly typed languages, others are related to functional and dynamic languages.

Eclipse community understands a need in Refined Language Model which could be used as a base for concrete language models. For instance, such a model was described in Language Development Toolkit Proposal but has been withdrawn. Quote: *"Shared Editing Infrastructure and Reified Language Modeling. Exploring potential reuse among the JDT, CDT…"* We can find the same ideas in Photran Proposal: *"…make the CDT code-base more language neutral (i.e. progress towards a UDT or XDT)…"* Also we can see successful movement in this direction with LTK Refactoring plug-ins, which provide an abstraction layer and UI for refactoring support for different languages.

While Refined Language Model looks reasonable for some groups of languages, it looks useless when targeted to cover all possible languages. The aim of this paper is to try to argue reason for generic language model and common infrastructure for a subset of modern programming languages: dynamic (scripting) languages like PHP, Python, Ruby, and Perl.

Here and below under "Language Model" we mean representation of code structure along with related services like type hierarchy, search functionality, type evaluation, etc.

Current IDEs for Dynamic Languages

We have been tracking several available IDEs for dynamic languages for almost 3 years. All of them develops by the following scenario:

- Simple text editor with syntax highlighting.
- External interpreter used for several language aware tasks: (syntax checking, code analysis and refactoring with tools available for that language). (EPIC - Perl, PyDev - Python).
- Internal source code parsers for code outline and syntax check tasks.
- Language Model as a base for most tasks required by modern IDEs.

It looks to be common for most Eclipse Platform-based IDEs that JDT Core model is used as a starting point and "adopted" for a new language. The same approach selected by most of dynamic language plug-in providers, which went more or less far in JDT Core "adaptation" for concrete dynamic language.

Our experience in JDT Core adaptation

We have also passed this way in our TruStudio Foundation project. Target of TruStudio is to provide a base for dynamic language IDEs and for the time of writing PHP IDE and Python IDE have been built on top of it.

Right now we have clear understanding of Generic Language Model for Dynamic Languages, and below are problems we met during JDT adoption.

Dynamic vs. Static Languages

Models for static languages are "declarative", opposite dynamic language model should be less "declarative", lightweight, and … "dynamic".

While model elements for static languages hold all information about element declaration like references to super classes and parameter types, code in dynamic languages in most cases doesn't provide such information at all or this information needs to be evaluated in specific context (for example: a Python class can be derived from class which is evaluated during some function execution, or less specific example: method declarations can be "included" in context of the class declaration from another piece of source code, or even binary module).

Due to dynamic nature of these languages some tasks which could be done trivially with static languages (for example parameter type evaluation) became rather complex in dynamic language environments and need additional support from model (for example call graph and function invocation analysis is often required to guess possible parameter types of that function).

JDT Core misusing for dynamic languages

Since adoption of JDT Core model is the approach we can get some things working fast it looks not correct in long perspective as an approach to develop model for dynamic language, even more - not suitable as a generic model for many dynamic languages.

JDT Core model is tightly coupled with Java language conventions, reflects Java language specification very closely, doesn't care about model element and parameter type evaluation due to static Java nature, and so on.

Conclusions

Refined Language Model for *any* language is not reasonable.

Static Language Model is not suitable for Dynamic Languages.

Generic Dynamic Language Model and Common Infrastructure looks promising.

Generic Dynamic Language Model and Common Infrastructure

Since Refined Language Model for all language types looks useless, and static language model looks not very suitable for dynamic languages, Generic Language Model for dynamic languages looks very promising:

Reasons for Common Infrastructure

Common execution environments

The idea to build common execution environments for virtually any scripting languages is not new and implemented in several projects. One of the famous is Parrot VM – Perl Foundation's virtual machine for Perl6 and other dynamic languages. At the moment Parrot VM executes Perl, Python, and other dynamic languages. Ideally code in these languages would become interoperable inside Parrot VM, with the ability to extend Python class in Perl or pass Python objects as parameters to the Perl function.

Such environments and projects prove structural compatibility of these dynamic languages and another reason for Generic (Common) Language Model.

Proof-of-concepts

As I wrote before, our JDT adoption for dynamic languages is far from ideal, but is "proof-of-concepts": it's definitely possible to build common infrastructure for dynamic language IDEs. Such infrastructure drastically decreases IDE development time and focuses implementer on language-aware components only.

Some facts: in our implementation (TruStudio) Language-independent plug-in sizes are 3.8 MB, language-aware PHP and Python plug-in sizes are 0.9MB and 0.8MB respectively. So we have about 4:1 ratio.

Major components of Common Infrastructure

Common Infrastructure includes following major components:

- Dynamic Languages Core.
- Debug Infrastructure.
- Common UI Components.

Dynamic Language Core

So JDT adoption seems to be a wrong way for building generic language model for dynamic languages, but there are a some things to reuse (carefully), which should increase GDLM development time as well as code quality, and also (potentially) could be pushed down to LTK (language toolkit) along with refactoring support.

As it seems to us, some code and/or concepts must be borrowed from JDT for the following reasons:

- Follow brilliant JDT Core concepts which are mature and proven to work.
- Provide well-known architecture for persons, already familiar with JDT Core.
- Explore possibility to push parts of JDT Core down to LTK.

Generic and Language-aware parts

Dynamic Language Models will consist of Generic parts working together with Language-aware parts. Language-aware parts are built around source/binary code parsers, language conventions and other components like identifier type evaluators, element visibility rules, etc. At the same time, contributed language-aware parts may extend model with language-specific elements like annotations (introduced in Python 2.4 and similar to Java 1.5 annotations), which are ignored by Common Infrastructure and used by language-aware components on its own.

Common UI Components

Generic Language Model provides services on top of which Common UI Components can be built. These components includes but not limited to:

- Structural Code Explorer (like Package Explorer in JDT).
- Code Outline.
- Search and Navigation UI.

Debug Infrastructure

PHP, Perl and Python interpreters do not offer any network-based debug protocol. However, different third-party tools offer network-based debug functionality. For example DBGp protocol from Xdebug project serves as a common protocol for dynamic languages, and is used as de-facto standard in several IDEs.

Implementation of common debugging protocol among with IDE-side debugger over this protocol could be a great step forward to standards in scripting world and reduce debugger implementation efforts for target languages.

Relation to other Eclipse Projects

Web Standard Tools

Dynamic Languages are widely used in web development: PHP, Python with Zope application server, Ruby with Ruby on Rails, etc. Eclipse is also an excellent platform for building web development tools (WST). Strong dynamic languages support, provided by Dynamic Languages Toolkit will attract a lot of non Java-centric developers, plays JDT role for dynamic world and will focus tool developers on concrete web tools implementation and technology support instead of spending time on languages.

Modeling Tools and Technologies

Users of dynamic languages can potentially benefit from power of various Eclipse Projects related to modeling. Starting from EMF (generating dynamic language sources from Ecore models) up to future projects like UML editors, MDA projects, etc.

Mobile Development Tools

Dynamic languages become used in mobile world (Nokia Python SDK). Combined with mobile development and deployment tools, Dynamic Languages Toolkit would be good add-on to mobile and embedded software development tools.

Chapter 2

Eclipse Scripting

Eclipse Scripting with Dynamic Languages

Important notes

I decided to include this chapter in paper, because talks about dynamic languages support in most cases triggers talks about usage of these languages for Eclipse scripting. By Eclipse scripting I mean adding some functionality to Eclipse using scripting languages by development of Eclipse extensions with scripting languages and in other ways.

Nevertheless, I included this chapter in the paper, because I think that Eclipse Scripting and Dynamic Languages Toolkit are very different issues (only one thing which makes them similar is the word “script”). I will try to argue below.

Why Eclipse needs scripting

I don’t pretend to provide deep analysis here, just like to write down common reasons why scripting support may be useful for Eclipse. These reasons are the same for many applications – not only Eclipse. So scripting language would be good for:

1) Attracting non-Java developers and allowing them to develop Eclipse extensions in simple language without Java (and even OOP) knowledge (I don’t want to go deep into philosophy and discuss why for example PHP is “simpler” or “harder” than Java, but in fact there are a lot of programmers in the world who can program in PHP and can not (don’t want?) in Java. At least Eclipse scripting will attract those languages fans.

2) Many developers share an opinion that usage of scripting languages decreases development time and is good at least for fast prototyping. If this assumption is true, then fast development/prototyping is another reason for Eclipse scripting.

3) End-users will have lightweight approach to develop/change interpreted scripts to extend/tune application without heavy development process and just-in-time. Such feature may be very interested for many RCP application vendors (not only for Eclipse Technology or Tools projects).

Implementation scenarios

Let's assume that we have Java implementations of scripting languages which are interoperable with Java, like Mozilla Rhino (ECMAScript) or Jython (Python). Such implementations allow developer to extend Java classes, invoke methods, and so on. In this case it's simple to add scripting support to Eclipse Platform and allow virtually any Eclipse extension, plug-in, or application to be implemented completely in scripting language. Moreover, with Eclipse 3.1 extension factories we can do this in elegant way without any changes to Eclipse Core, considering:

```
<extension
      point="org.eclipse.ui.views">
   <category
         id="org.eclipse.scripting.example"
         name="Sample Category"/>
   <view
         category="org.eclipse.scripting.example"
         class=
   "org.eclipse.scripting.ScriptedExtension:/views/sample.js"
         icon="icons/sample.gif"
         id="org.eclipse.scripting.example.views.SampleView"
         name="Sample View"/>
</extension>
```

Above is correctly declared Eclipse extension, and *org.eclipse.scripting.ScriptedExtension* class implements *IExecutableExtensionFactory* with simple behavior: it executes script, which location is specified by initialization data ('/views/sample.js' in our case) and returns script execution result as executable extension instance. Of course script must respect extension point specification and must return an instance implementing *IViewPart* interface for our sample. Simple '/views/sample.js' (path relative to plug-in root):

```
var view = org.eclipse.scripting.example.MyJavaView()
view.setTitle("Created from JavaScript")
return view
```

Are any benefits there?

This approach definitely allows developing Eclipse applications in scripting languages, and very easy to implement (we assume that we using scripting engine that is interoperable with Java like Rhino). Let's check benefits we gain from this.

1) Users definitely can develop applications in scripting language. At the same time they must be familiar with Eclipse API, Java language, and OO concepts because developers will extend Java classes like *ViewPart* in JavaScript or extend them in Java and use JavaScript as a 'glue' for Java objects (compose controls, set attributes, etc). So newbie inexperienced developer will not be able to use such scripting environment.

2) Development/prototyping speed-up. Clearly no. Most scripting languages are interpreted and benefits we have from interpreter like no compilation time or possibility to change code at run-time is one of the reasons why development with scripting languages going faster. Eclipse JDT and PDE are enough powerful tools and among

with incremental compilation and hot code swap during debugging provides the same benefits. Dynamic nature of the languages … to be continued

3) End-user can hack/extend functionality. According to (1) & (2) there are no benefits to extend with scripting approach in comparison to extend in Java. Possibility to "hack" running application looks to be unreasonable and should not be considered as benefit.

Adding declarative language for Eclipse Scripting

It seems that we can not gain any benefits using imperative languages for Eclipse Scripting. Consider declarative language scenario, like Mozilla's XUL or Microsoft's XAML.

We at xored software have sample implementation of such language (please write to andrey@xored.com if interested in implementation). Let's start from looking at sample code, and follow with concepts and analysis.

Eclipse extensions are defined as other Java extension like in *plugin.xml* fragment above or in-place in *plugin.xml* since this language source is well-formed XML (not a good solution because PDE will report errors due to extension point schema violations).

```
<view xmlns="http://rcpml.org/ui"
      xmlns:core="http://rcpml.org/core">
   <core:script>
      function alert(message) {
         org.eclipse.jface.dialogs.MessageDialog.
            openInformation(null, 'title', message)
      }
   </core:script>
   <button text="Sample Button"
           onclick="alert('Hello from JavaScript')"/>
</view>
```

Let's look at same extension implemented in Java:

to be continued