MetaProg*

Quality in Software

Rosetta – a dialect neutral framework for Smalltalk source code exchange

---

Version 1.0

November 10, 2002

MetaProg*
Quality in Software

Bachlettenstrasse 41
CH-4054 Basel
Switzerland
Email jpelrine@metaprog.com

Rosetta

Introduction

Rosetta is a dialect-neutral, XML-based format for Smalltalk code interchange. It is based upon a semantic model of Smalltalk that allows dialect-specific concepts to be mapped to a cross-dialect form, thus enabling transformation between different file-in formats.

Rosetta supports namespace annotations in dialects that do not implement namespaces, thus aiding code porting from a non-namespaced to a namespaced dialect and vice-versa. Also, the use of custom transformation blocks for method source code allows for powerful source code manipulation using the Refactoring Browser's rewrite tool.

Rosetta was designed to file into a vanilla base image, with no add-ons, compatibility layers etc. It is a lossy transformation format, which means it will install as much as it can, but probably leave out constructs (like namespaces) that cannot be mapped in the receiving dialect.

Rosetta is not yet finished. It will keep getting better, though.

Availability

The Rosetta export framework is currently available for VisualAge, Dolphin, and VisualWorks 3 (with ENVY) and 7 (parcels). Exporters for other dialects will become available as soon as either we or someone else port them.

The Rosetta export framework operates at the level of a "Package" – an ENVY (Sub)Application, Dolphin Package, or VisualWorks Package or Parcel. It also requires the package to be in-image when exported. Support for exporting "Clusters" (i.e. ConfigurationMaps or Bundles etc.) will be included in the next release.

Other extensions, such as support for batch processing of a complete ENVY repository without the necessity of loading applications into the image, are available from MetaProg for commercial use at reasonable rates.

Licensing

As a service to the Smalltalk community, Rosetta is available free for non-commercial or in-house development use. For redistribution or use in a commercial product, please feel free to contact MetaProg for licensing information.

Getting Started

Installation

1. Get the Rosetta distribution. If you're reading this, you probably have it already.
2. From the ./dist directory, get and load the distribution for your dialect. Tweak ModuleFacade class>>#dtdUrl to point to where you have the Rosetta.dtd if you're doing validation (you'll find it in ./DTD for now).
3. Export some code, using the examples in the next section as a starting point.

Export

Here are some examples of how to export code into Rosetta format.

```
"VisualWorks"
| package stream |
package := Parcel parcelNamed: 'MyPackageName'.
"or with Store"
package := (Store.Registry packageNamed: 'MyPackageName') asParcel.
[stream := '<MyPackageName>.xml' asFilename writeStream.
package facade writeRosettaXmlOn: stream] ensure: [stream close].

"VisualAge"
VA has a menu item in the ApplicationManager menu. To export
programmatically, do something like the following:"
| package stream |
package := Smalltalk classAt: #MyPackageName.
stream := CfsWriteFileStream openEmpty: '<MyPackageName>.xml'.
[package facade writeRosettaXmlOn: stream] ensure: [stream close]

"Dolphin"
| package stream |
package := PackageManager current packageNamed: 'MyPackageName'.
stream := FileStream write: '<MyPackageName>.xml'.
[package facade writeRosettaXmlOn: stream] ensure: [stream close]
```


Namespace Support

To enhance code portability, Rosetta offers namespace annotation support for dialects which do not yet implement namespaces. Using this support will enable you to notate in advance the namespace you wish a class to belong to, and to have this information used during transformation.

The support is relatively simple, and consists of two methods. To declare a namespace for a class, define a class method named **#rosettaNamespace**. This method should return a String describing the fully qualified namespace path for the class.

Here's an example:

```
TestCase class>>#rosettaNamespace
    ^'XProgramming.SUnit'
```

As you can imagine, such methods are best written as class extensions contained in a separate package.

To define a namespace itself, one defines the method **#rosettaNamespaces** on the package containing the namespaces. This method should return an hierarchically-ordered Collection of the namespaces defined in the package.

Here's an example:

```
RosettaTestCases>>#rosettaNamespaces
    ^Array with: (
            (PseudoNamespace named: 'MetaProg' parent: 'Smalltalk')
                    addImport: 'Smalltalk.*' private: true;
                    comment: 'This is a comment ';
                    yourself)
            with: (
            (PseudoNamespace named: 'Rosetta' parent: 'MetaProg')
                    addImport: 'Smalltalk.*' private: true;
                    addImport: 'XML.*' private: false;
                    comment: 'This is another comment ';
                    yourself)
```

You can add any number of imports to a namespace, and any number of namespaces to a package. As you can imagine, such methods are also best written as class extensions contained in a separate package.

Transformation

Once you've exported code out of Smalltalk into Rosetta format, you'll probably want to transform it into another dialect. To do this, though, you need an XSLT processor.

Non-Smalltalk Transformers

There are several XSLT processors to choose from. The three main ones are: Saxon, Xalan, and Microsoft's MSXML3. All of these can be downloaded free of charge.

Saxon is an open source XSLT processor developed by Michael Kay. It is a Java application, and can be run directly from the command prompt; no web server of browser is required. It is available at http://saxon.sourceforge.net/.

If you are running Windows, the simplest way to use Saxon is to download Instant Saxon, which is packaged as a Windows executable. You will need to have Java installed, but that will be there already if you have any recent version of Internet Explorer. On non-Windows platforms, you will need to install the full Saxon product and follow the instructions that come with it.

Xalan is another open-source XSLT processor, available from the Apache project at http://xml.apache.org. Xalan was originally derived from an IBM product called LotusXSL, but it has since developed an open-source life of its own. Xalan is available in both Java and C++ versions. Like Saxon, Xalan-Java is a Java application that can be run from the command prompt. Also, similar to Instant Saxon, the Xalan/C++ version is available as a native Windows executable.

Saxon and Xalan both implement the same Java interface, known as TrAX. Both are highly conformant to the XSLT 1.0 specification, so stylesheets are highly portable.

Alternatively, you can use Microsoft's processor, but you need to process the translation through Internet Explorer, which is quite clumsy to do. We recommend you avoid it.

Configuring and Transforming with Saxon or Xalan

These instructions presume you're going to use the compiled, executable version of Xalan or Saxon. If you use the Java versions of either of these, or use another tool. you'll probably have to correct these instructions. Good luck.

1. In the base Rosetta directory, edit setXSLTCmd.bat to point to the location of the Saxon or Xalan executable.
2. Execute the following from the command line (adjusted for your filename, of course):

```
Ros2<Foo> yourFileName
```

where <Foo> is one of the following:

a. App – transform to an ENVY .app file
b. Chunk – transform to a generic chunk format file. This transform is not complete yet, and will definitely need a bit of tweaking
c. Pac – transform to a Dolphin .pac file
d. Sif – transform to a generic SIF format file. This transform is not complete yet, and will definitely need a bit of tweaking
e. VW – transform to a VW7 (XML format) parcel file-in.

Smalltalk Transformers

VisualWorks provides an XSLT processor with the base distribution. In order to use it, you'll have to load the XSL parcel. Unfortunately, it's broken right now, and we'll have to wait for the bug fix to be able to use it. If you're interested, ask Cincom engineering about the status of AR 45123.

Dolphin wrappers the MSXML control, and we'll document how to access the XSLT mechanism as soon as we can figure out how to do it ourselves.

We are not currently aware of native XSLT support in VisualAge.

Transforming with VisualWorks XML

Use this code fragment as an example for how to transform a Rosetta file using VisualWorks' built-in XSLT processor.

```
| xslFile xmlFile parser doc transDoc |
xslFile := 'F:\XML\Smalltalk\Rosetta\XSL\Ros2Vw.xsl'.
xmlFile := 'F:\XML\Smalltalk\Rosetta\src\RosettaTests.xml'
xslRules := ( XSL.RuleDatabase new ) readFileNamed: xslFile.
parser := XMLParser new validate: false.
doc := parser parse: xmlFile asFilename readStream.
transDoc := xslRules process: doc.
transDoc inspect
```


Customizing the Transformation

By using XML and XSL technologies, we are not only buzzword-compliant <grin>, we get a lot of flexibility in how and what we transform. This flexibility comes at a price, though – XSL is just plain ugly. Don't expect us to explain it here.

If you want to start customizing your stylesheets, or maybe writing your own, we suggest you first get a tool such as XMLSpy (http://www.xmlspy.com), which has the option of plugging in different XSLT processors, and which we enjoy using, or possibly Xeena from AlphaWorks, (http://www.alphaWorks.ibm.com/tech/Xeena), which is free, but whose built-in XSLT processor has some problems.

We'd also highly recommend getting a good book on XSLT. We've found Michael Kay's XSLT Programmer's Reference (Wrox Press, Chicago, 2001) to be one of the best.

How it works

Not yet finished.

Encyclopedia of Classes

This section contains the public protocol for the wrapper classes contained in the Rosetta distribution. An architectural description of these classes can be found in the section “System Architecture”. In addition, we document the extensions to the standard system classes.

For simplicity’s sake, we have left out accessor methods and export methods.

RosettaPreLoad

The RosettaPreLoad package is dialect-specific, and consists mainly of wrapper classes used to provide a consistent meta-object protocol across dialects. This protocol is based on ideas described by Kent Beck in [Beck95]. If you want to port Rosetta to a new dialect, you’ll have to implement these classes and messages. It’s pretty easy though – the RosettaPreLoadTests package (which requires SUnit 3.1) provides description strings for the tests which will guide you as to what to do next.

Behavior

Instance methods

metaObject

Return an instance of MetaObject as a wrapper around the receiver

CompiledMethod

Instance methods

metaObject

Return an instance of CompiledMethodMetaObject as a wrapper around the receiver

CompiledMethodMetaObject

A CompiledMethodMetaObject is a wrapper around a CompiledMethod instance.

Inherits from

MetaObject Object

Named Instance Variables

method

Class Instance Variables

transformBlock

Instance methods

basicSourceString
return the raw source string

categories
return the categories/protocols the wrapped object belongs to

category
return the main category/protocol the wrapped object belongs to

comment
return the method's comment field, if any

fullyQualifiedClassName
return the namespaced class name of the wrapped object's method class

isClassMethod
is the method a class method?

IsPrivateMethod
is the method a private method?

method
return the wrapped object

method:
set the wrapped object

methodClass
return the wrapped object's method class

rosettaNamespace
return the method's namespace, if supported

selector
return the method's selector

sourceString
return the wrapped object's source string after running it through the class-side transformBlock

Class methods

defaultTransformBlock
return a one-argument block, taking the instance of CompiledMethodMetaObject as argument, and returning the transformed source code

namespacedTransformBlock
return a sample transform block using the RefactoringBrowser's rewrite tool to do a namespace conversion.

onMethod:
return a new instance wrappering a CompiledMethod

resetTransformBlock
set <transformBlock> to nil

MetaObject

A MetaObject is a wrapper around a Class.

Inherits from

Object

Named Instance Variables

object

Instance methods

category
return the class category, if supported

classComment
return the class comment

classInstanceVariableString
return a space-separated string of class instance variable names

classVariableString
return a space-separated string of class variable names

instanceVariableString
return a space-separated string of instance variable names

instVarTypeString
return a string describing instance variable type

isDefinedIn:
return whether the wrappered object is managed by the package argument

isMetaclass
return whether the wrappered object is the metaclass

isPrivate
return whether the wrappered object is private

name
return the wrappered object's name

sharedVariableString
return a space-separated string of pool (shared) variable names

superclass
return the wrappered object's superclass

theNonMetaclass
return the wrappered object's instance behavior instance

Class methods

asciiValueOf:
return the ascii value of the Character argument

characterWithAsciiValue:
return the Character with the Integer's ascii value

on:
return a MetaObject wrappering the Behavior argument

ModuleFacade

ModuleFacade is an abstract class for wrappers around packages and clusters.

Inherits from

Object

Named Instance Variables

object

Instance methods

exportingDialect
return a String designating the exporting dialect. This String can be one of: (VA | VW | D | OS | GNU | STX | MT | GS | VSE | AOS)

exportingDialectVersion
return a String designating the version of the exporting dialect

moduleSpec
return an instance of ModuleSpecification defining the receiver

moduleTimestamp
return a String with the receiver's timestamp in Zulu format (YYYYMMDDHHMMSS)

moduleType
return a String with the module type (cluster|package)

moduleVersion
return a String designating the module's version

packageName
return the name of the module

prereqModuleSpec
return a "prerequisite" or incomplete ModuleSpecification designating the receiver

Class methods

defaultDtdUrl
return a String with the location of Rosetta.dtd

on:
return a ModuleFacade wrappering the argument

PackageFacade

A PackageFacade is a wrapper around a dialect's class representing a Package – a VisualAge (Sub)Application, Dolphin Package, VisualWorks Parcel or Package etc. This class serves as the unit of granularity for export. As a consequence, although imports for all dialects are planned, export support for dialects without an adequate module concept (Squeak, for instance) will probably not be coming for a long time.

Inherits from

ModuleFacade Object

Named Instance Variables
(None)

Instance methods

declarations
return a properly-ordered collection of shared Pools and Pool variables

definedClasses
return a collection of all classes defined by the receiver's wrapped object

definedMethods
return a collection of all CompiledMethods defined by the receiver's wrapped object

initializerDefinitions
return a collection of all initializers defined by the receiver's wrapped object

moduleType
return 'Package'

prerequisites
return a Collection of the wrapped object's prerequisites

rosettaNamespaces
return a Collection of RosettaNamespace instances describing the namespaces owned by the receiver. This only needs to be implemented this way in dialects which don't support namespaces. VisualWorks et al. directly return the namespaces they contain.

Class methods

onPackageNamed:
return a PackageFacade wrappering the package name <argument>

packageNamed:
delegating to the dialect's package manager, return the package named <aString>

PseudoNamespace

A PseudoNamespace is a placeholder, in a package, for a namespace declaration, for dialects not supporting namespaces

Inherits from
Object

Named Instance Variables
comment
imports
name
parentNamespace
private

Instance methods
addImport:private:
add <aNamespaceString> as an import to the receiver, with visibility <aBoolean>

Class methods
named:parent:
return a PseudoNamespace named <aString>, contained in parent namespace <parentString>

RosettaModuleSpec

A RosettaModuleSpec contains relevant information about a package, such as creation time, exporting dialect etc. See the instance variables.

Inherits from
Object

Named instance Variables
exportingDialect
exportingDialectVersion
inclusionContexts
moduleName
moduleTimestamp
moduleType
moduleVersion
repositoryURL

Instance methods
formattedTimeStamp
Return the time stamp, formatted in ISO Zulu format (YYYYMMDDHHMMSS)

timeStampFormatter
Return the formatter used to format the timestamp

Class methods

onModuleFacade:
return a wrapper on a package

onPrereq:
return a minimal wrapper on a package, usable for defining it as a prereq

ScopedPoolFacade

A ScopedVariableFacade is a wrapper around a pool (or shared) variable

Inherits from

Object

Named Instance Variables

container
object

Instance methods

comment
Return a comment string, if any

isConstant
is the receiver constant

isPublic
is the receiver public

name
return the name by which the wrapped object can be accessed in it's containing Dictionary

poolName
return the name of the containing Dictionary

rosettaXmlElementTag
return 'ScopedPoolFacade'

Class methods

on:
return a ScopedPoolFacade wrappering the argument

ScopedVariableFacade

A ScopedVariableFacade is a wrapper around a pool (or shared) variable

Inherits from
ScopedPoolFacade Object

Named Instance Variables
(None)

Instance methods
rosettaXmlElementTag
return 'ScopedVariableFacade'

valueExpression
return a String which, when evaluated, will give the value the receiver is set to

String

Instance methods
rosettaCompactString
return a copy with leading and trailing blanks removed

writeHtmlEncodedOn:
write an html-escaped version of the receiver to <aStream>

WriteStream

Instance methods
crtab:
write a carriage return, followed by <anInteger> tabs, to the Stream

ROSETTA BASE

The RosettaBase package consists largely of extensions to dump components onto a Stream in Rosetta format. We've left these methods out here, and only documented the few remaining methods that are relevant in the context of namespace support.

ClassDescription

Instance methods

rosettaNamespace
Return an emptyString. Hook method

MetaObject

Instance methods

fullyQualifiedName
return a String containing the namespace-prefixed class name

fullyQualifiedSuperclassName
return a String containing the namespace-prefixed superclass name

rosettaNamespace
in dialects not supporting namespaces, return a String with the namespace the class should be in

References

[Beck95] Kent Beck, “A Modest Meta Proposal”, *Smalltalk Report*, July-August 1995
[Kay01] Michael Kay, *XSLT Programmer’s Reference,* Wrox Press, Chicago, 2001

Acknowledgements

Joseph Pelrine wishes to thank John Sarkela and Paul McDonough, whose ideas during the ill-fated Stable Squeak project helped build the basis for the semantic model behind Rosetta, Don MacQueen, whose email in the summer of 2002 prompted me to start the project, Reinout Heeck, my pair-programming partner at the ESUG 2002 iteration of Camp Smalltalk, Dave Simmons, who suggested Rosetta as a cool and appropriate name, Alan Knight, who ripped apart the code with me, and gave me the idea for supporting multiple binary resources at method level, Roger Whitney, who gave us the idea of having a web repository of code in Rosetta format, which could be converted to a specific dialect by the (Smalltalk) web server upon download, John O’Keefe and Eliot Miranda, who were willing to consider the Rosetta DTD as the start of a grass-roots standard for code exchange, and last but not least Ralph Johnson, probably the world’s greatest supporter of Smalltalk.

Appendix A. The annotated Rosetta DTD

Not yet finished

