Importing Java Code into FAMIX

Alexandre Bergel
ADAM Project, INRIA, Lille, France
alexandre.bergel@inria.fr

1. INTRODUCTION

Moose is a collaborative platform for software analysis and information visualization[1]. Moose promotes the FAMIX language independent meta model in which program source codes may be imported. Analysis may be conducted over an imported program using the large set of available tools[2].

A typical FAMIX importers embodies two distinct parts: (i) a parser intended to convert a program source code into a set of abstract syntax tree nodes, and (ii) a function that maps those nodes into proper FAMIX abstractions. A number of importers are available for Moose: Lisp, Python, Smalltalk, C++, JSP and MSE are probably the most used one. To this big picture, Java has been shrugged off.

Several attempts have been made to increase the affinity between Java and FAMIX. External tools such as Penumbra and Moose Brewer directly operates as an Eclipse plugin to populate a FAMIX metamodel or to generate MSE files. However, these solution necessitate the use of Eclipse which might not always a wanted thing. This paper is about reconciling FAMIX/Moose with Java without the adjunction of external converter that rely on a foreign technology.

This paper describes Java4Moose, an extension of Moose to import Java source code. Input is simply .java source files that are parsed then translated into FAMIX, Moose's meta model. Dependencies between classes are extracted from class and type references contained into method body, instance variable declaration, method signatures, and interface implementation. This paper is organized as follows: Section 2 describes the general schema used by Java4Moose to import Java code into FAMIX. Section 3 present the critical points of the implementation. Section 4 offers a brief overview of the related work. And finally, Section 5 concludes this paper.

2. IMPORTING JAVA CODE INTO FAMIX

Java4Moose is embodies two different components: (i) a parser that produces a representation of the textual source code, and (ii) a conversion function that maps elements of this representation into FAMIX elements.

Code representation. Inherently tools built on top of Moose operates on a rather high level view of a program code including class structure and dependencies.

- JavaClassNode describes a Java class. It contains its name, the name of its superclass, the variables, a set of methods (instances of JavaMethodNode described below), the name of the package that define it, a set of references that may be contained in static part, and the set of implemented interfaces (instances of JavaInterfaceNode described below).
- JavaInterfaceNode represents a Java interface. It contains a name, the names of its super interfaces, a set of methods (described below), a package names, and a set of static variables.
- JavaMethodNode contains a name, a return type, a reference to the JavaClassNode that defines the method, its source code, and a set of types that are referenced by this method.

Note that a type is represented by a simple string at that stage. After the parsing phase, instances of the three classes mentioned above are mapped into FAMIX elements. As an illustration, let us consider the following Java class definition extracted from the AWT Java library:

```
// File Checkbox.java
package java.awt;
public class Checkbox extends Component
            implements ItemSelectable, Accessible {
    static {
        Toolkit.loadLibraries();
        if (!GraphicsEnvironment.isHeadless()) {
            initIDs();
        }
    }
    private static final long serialVersionUID =
        7270714317450821763L;
    void setStateInternal(boolean state) {
        this.state = state;
        CheckboxPeer peer = (CheckboxPeer)this.peer;
        if (peer != null) {
            peer.setState(state);
```

[1] http://moose.unibe.ch
[2] http://moose.unibe.ch/tools

```
        }
    }
    protected class AccessibleAWTCheckbox
        extends AccessibleAWTComponent
        implements ItemListener, AccessibleAction,
            AccessibleValue
    {
        private static final long serialVersionUID =
            7881579233144754107L;
    }
}
```

Checkbox is represented as an instance of JavaClassNode having the variables[3]:

className	set to 'Checkbox'
superclassName	set to 'Component'
variables	refers to a collection with 'serialVersionUID' as its unique element
packageName	set to 'java.awt'
typeReferences	refers to a collection with 'Toolkit' and 'GraphicsEnvironment' as elements
interfaces	refers to a collection containing 'ItemSelectable' and 'Accessible'
methods	refers to a collection containing an instance of JavaMethodNode as its unique element. This instance has the name 'setStateInternal', the 'void' return type, and the collection #('boolean' 'CheckboxPeer') as referenced types.
innerClasses	refers to a collection containing an instance of JavaClassNode to represent the inner class AccessibleAWTCheckbox.

Mapping into FAMIX elements. The second phase of importing Java files task consists in translating the code representation described above into FAMIX elements. Because of the inherent mutual dependencies between elements of the code representation, this translation has to be performed in two steps. An example of such mutual dependencies occurs when importing the Object and String Java classes: the class String is a direct subclass from Object and the class Object defines the method toString() that has String as a return type. None of these classes can be fully imported into a FAMIX model without the presence of the other classes. This simple situation exhibits the need of having a translation of Java code representation into FAMIX in two steps.

First, an instance of FAMIXClass is created for each JavaClassNode and JavaClassInterface. These instances are stored into a globally accessible dictionary with the name of the corresponding Java element as the key. These instances are almost empty at that stage: a FAMIXClass has only a name, a flag saying whether it is an interface or a class.

Then, a second iteration over JavaClassNode's and JavaInterfaceNode's instances is performed to "fill" all the FAMIXClass's instances. At that stage, methods (instances of JavaMethodNode) are mapped into instances of FAMIXMethod. It is very likely that when this mapping is being realized some type references are absent from the set of classes and interfaces that have been imported. All dependencies external to the imported code defines libraries that may be part of the runtime libraries (e.g., the class Object, the collection libraries). The FAMIX class that represents such a type is set as stub.

3. IMPLEMENTATION

Java files are imported by triggering the method JavaImporter»importFile: aFileName on an instance of the JavaImporter class. When the name of folder is provided, a recursion is performed. Note that only files having a suffix .java are imported. This help preventing non java files such as package description (stored as HTML files) to be processed. The importFile: methods only create a first representation of the recursively attainable files. The import is complete when a FAMIX model is effectively created. This is achieved by invoking createModel on an importer.

SmaCC[4] is in charge of parsing Java. One drawback of SmaCC is to not produce abstract syntax tree. Java4Moose builds the code representation using rules associated to the Java 1.5 grammar production. Thanks to the excellent parser generator SmaCC, Java4Moose offers satisfactory performance and scales up nicely. As an example, importing the 652 classes and 4947 method of the whole Java GUI library AWT takes only 8.8 seconds on a MacBook with 1 Gb of memory.

4. RELATED WORK

A number of related works exist and are listed in this section. Dependencies between Java files are explicit in binary Java class files. A simple analysis over .class files may recreate the graph of dependencies [1]. Java4Moose operates directly on source code, independently whether the program under analysis may be compiled or not.

There has been a lot of work on bridging Moose with the Java world. The most two relevant projects are Moose Brewer and Penumbra. Moose Brewer is an Eclipse plugin to generate MSE file from an Eclipse project. *Penumbra*[5] is a Visualworks application that makes the Eclipse application steerable within Smalltalk. Queries toward Eclipse may be directly formulated in Smalltalk. Java4Moose is a pure Smalltalk solution to directly import Java files.

5. CONCLUSION

Having left Java out of the range of supported languages by Moose has probably repelled a number of potential users. This paper aims at filling this gap by proposing a native Java importer. The import of Java source code relies on two distinct steps: first a representation of the source code is created, then a mapping to FAMIX element is realized. As a future work, we envision a set of different visualizations of Java source code centered on Java concepts such as generics, dissociation of class and types, inner classes. Java4Moose is freely accessible from http://moose.unibe.ch/tools/Java4Moose, its official website.

6. REFERENCES

[1] H. Melton and E. Tempero. An empirical study of cycles among classes in java. *Empirical Software Engineering*, 12(4):389–415, 2007.

[3] We recall that the Smalltalk style of writing strings (*i.e.,* ordered set of characters) makes use of delimiting quote (') as in 'this is a string'.

[4] http://www.refactory.com/Software/SmaCC/index.html

[5] http://www.info.ucl.ac.be/~jbrichau/penumbra.html