FAMIX Java language plug-in 1.0

Author	Sander Tichelaar (tichel@iam.unibe.ch)
Version	1.0
Last modified	1999-09-07

1 Abstract

This document describes the language plug-in to the FAMIX 2.0 model [Deme99] for the Java programming language [Gosl96]. It handles interpretation issues concerning Java in FAMIX and the extension of the FAMIX model for Java specific features.

2 Notation

The basic FAMIX model is modified in three different ways to handle Java sources:

- New classes are added to the common exchange model to model entities and associations unique to Java. These classes are marked as new entities resp. associations.
- New attributes are added to existing classes of the basic FAMIX model. In this case the class is marked "extended" and only the new and modified (see below) attributes are listed in the definition of the modified class.
- The definition of attributes of existing classes are modified or are made more specific. In this case the corresponding class is marked "interpreted" and the interpreted attributes are listed in the definition of the modified class. To discriminate new from interpreted attributes, new attributes are explicitly tagged as being new and interpreted attributes are listed without any type information since that information hasn't changed anyway.

3 Modified classes

3.1 Model (interpreted)

Model
SourceLanguage SourceDialect

Figure 1: Model

The new or modified attributes are:

- `SourceLanguage`
  For Java models this attribute always contains the string "Java".
- `SourceDialect`
  The Java language doesn't really have dialects, but it has versions. If known, this version can be stored in this attribute. The possibly interesting issues for FAMIX on the language-feature-and-syntax level (as opposed to added libraries) between the different versions are:

1.0.x -> 1.1.x:	- Addition of inner classes (including anonymous ones)
	- Final method parameters and local variables
1.1.x -> 1.2.x:	- Addition of a new keyword (strictfp)

3.2 Package (interpreted)

Package

Figure 2: Package

A Package maps in Java to the Java package construct. Packages in Java have the following properties:

- packages contain classes and packages. Both classes and packages can belong to only one package.
- package names should be unique within their encapsulating package.

Normally packages in Java map directly to the directory structure of source code, i.e. the source code for a certain class in a certain package appears in a directory with the same name as the package. Nested packages appear as subdirectories of the directory with the source code of the encapsulating package.

3.3 Class (interpreted and extended)

Class	
isInterface (): Boolean	# new
isPublic (): Boolean	# new
isFinal (): Boolean	# new
isAbstract	
belongsToPackage	

Figure 3: Class

Both classes and interfaces in Java are mapped to the FAMIX entity Class. Interfaces differ from classes in that they can only define abstract methods and final static variables. Interfaces cannot inherit from classes (for a full discussion, see InheritanceDefinition, p.6).

The new or modified attributes are:

- `isInterface: Boolean; optional`
  Is a predicate telling if the entity is an interface as opposed to a normal class.
- `isPublic: Boolean; optional`
  Is a predicate telling if the class is defined public or not. Public (as opposed to default) visibility means the class is visible outside its containing package.
- `isFinal: Boolean; optional`
  Is a predicate telling if the class is defined final or not. Final classes cannot be subclassed (and subsequently its methods cannot be overridden). Interfaces cannot be final.
- `isAbstract`
  In Java a class is abstract if the class is declared abstract. This is obligoraty if one or more of its methods are abstract. Even if the class does not contain any abstract methods, it can be declared abstract, implying that the class is not allowed to be instantiated. Interfaces are always abstract, but don’t have to be declared as such (although you may if you want to).
- `belongsToPackage`
  The package to which a class belongs is defined by the package statement at the beginning of a Java source file that also contains the class definition.

3.4 BehaviouralEntity (interpreted)

BehaviouralEntity
declaredReturnType
declaredReturnClass

Figure 4: BehaviouralEntity

The following attributes are interpreted as follows:

- `declaredReturnType`
  In Java this attribute can contain any primitive types, array types or classes (and interfaces).
- `declaredReturnClass`
  This attribute contains the unique name of the FAMIX class entity (which is a Java class or interface) if the declaredReturnType denotes such an entity.

3.5 Method (interpreted and extended)

Method	
isFinal (): Boolean	# new
isSynchronized (): Boolean	# new
isNative (): Boolean	# new
accessControlQualifier	
signature	
isPureAccessor	
hasClassScope	
isAbstract	
isConstructor	

Figure 5: Method

Each definition of a method in source code constitutes this entity.

The new or modified attributes are:

- `isFinal: Boolean; optional`
  Is a predicate telling if the method is defined final or not. Final methods cannot be overridden.
- `isSynchronized: Boolean; optional`
  Is a predicate telling if the method is defined synchronized or not. Only one of the synchronized methods of an instance of a class can be accessed at once at runtime.
- `isNative: Boolean; optional`
  Is a predicate telling if the method is defined native or not. Native methods are implemented in an external language (for instance, C++) and therefore do not have an implementation in the Java side of the code.
- `accessControlQualifier`
  The allowed access specifiers for methods are: `public, protected, private.` An empty specifier means default visibility, which denotes that the method is visible for all classes within the same package.

- `signature`
  In Java is a method is uniquely distinguished by its name and the number, the types and the position of its formal parameters. Therefore, the signature string takes the form `methodname(T1, ...,Tn)` where $T_{1..n}$ are the types of the formal parameters of the method (see also the section about unique naming conventions in the FAMIX 2.0 Specification [Deme99]). Note that parameters can be declared `final`, but that this finalness is *not* part of the method signature. A subclass can override a method and add or drop any final parameter modifiers you wish. You can also add or drop final modifiers in a method's parameters without causing any harm to existing compiled code that uses that method[Gosl96].
- `isPureAccessor`
  A pure reader accessor in Java normally looks like (accessing a variable `name`):
  ```
  String getName {
       return name;
  }
  ```
  A pure writer accessor normally looks like:
  ```
  void setName(String name) {
       this.name = name;
  }
  ```
- `hasClassScope`
  A method in Java has class scope if it is defined static.
- `isAbstract`
  A method is abstract, if it is declared abstract with the abstract keyword. An abstract method in Java doesn't have an implementation.
- `isConstructor`
  A constructor in Java has the form of a method with no declared return type and a name identical to the name of the class it belongs to.

3.6 StructuralEntity (interpreted)

StructuralEntity
declaredType
declaredClass

Figure 7: StructuralEntity

The following attributes are interpreted as follows:

- `declaredType`
  In Java this attribute can contain any primitive types, array types or classes (and interfaces).
- `declaredClass`
  This attribute contains the unique name of the FAMIX class entity (which is a Java class or interface) if the declaredType denotes such an entity.

3.7 Attribute (interpreted and extended)

Attribute	
isFinal (): Boolean	# new
isTransient (): Boolean	# new
isVolatile (): Boolean	# new
hasClassScope	
accessControlQualifier	

Figure 8: Attribute

The new or modified attributes are:

- `isFinal: Boolean; optional`
  Is a predicate telling if the attribute is defined final or not. Final attributes are set only once and cannot be changed afterwards.
- `isTransient: Boolean; optional`
  Is a predicate telling if the (non-static) attribute is defined transient or not. Transient indicates that an attribute is not part of an object's persistent state and thus needs not to be serialized with the object.
- `isVolatile: Boolean; optional`
  Is a predicate telling if the attribute is defined volatile or not. Volatile specifies that an attribute is used by synchronized threads and that the compiler should not attempt to perform optimizations with it.
- `hasClassScope`
  An attribute in Java has class scope if it is defined static.
- `accessControlQualifier`
  The allowed access specifiers are: `public, protected, private.` An empty specifier means default visibility, which denotes that the attribute is visible for all classes within the same package.

3.8 ImplicitVariable (interpreted)

ImplicitVariable

Figure 9: ImplicitVariable

Implicit variables in Java are `this`, `super` and `class`. `this` is an implicit instance variable which refers the current object a method is executing in. `super` refers to the superclass of the current object. `class` is not an implicit variable in the strict sense of the word (as it is also a keyword in Java). An expression like `String.class` evaluates to a reference to the String class object. This works for all types, including the primitive types. It is close enough, however, to an implicit static variable to be modelled as an implicit variable. Normally implicit variables will only appear in a transfer when they are explicitly referenced by other entities.

3.9 LocalVariable (extended)

LocalVariable
isFinal (): Boolean   # new

Figure 10: LocalVariable

The new or modified attributes are:

- `isFinal: Boolean; optional`
  Is a predicate telling if the attribute is defined final or not. Final local variables are set only once and cannot be changed afterwards.

3.10 FormalParameter (extended)

FormalParameter
isFinal (): Boolean   # new

Figure 11: Attribute

The new or modified attributes are:

- `isFinal: Boolean; optional`
  Is a predicate telling if the attribute is defined final or not. Final parameters cannot be changed within the methodbody of the method it is a parameter of. Note that the finalness of a parameter is not part of the method signature - it is simply a detail of the implementation. A subclass can override a method and add or drop any final parameter modifiers you wish. You can also add or drop final modifiers in a method's parameters without causing any harm to existing compiled code that uses that method.

3.11 InheritanceDefinition (interpreted)

InheritanceDefinition
accessControlQualifier
index

Figure 12: InheritanceDefinition

In Java classes always inherit from a single class (except the root class Object that doesn't inherit from any class). A class can *implement* multiple interfaces, which simulates some kind of multiple inheritance, but as interfaces do not have any implementation, resolving which method needs to be executed, is not a problem. Interfaces can inherit from multiple interfaces. In FAMIX classes and interfaces are treated similarly, as shown by the fact that they are both represented as classes, therefore both class inheritance and interface implementation is represented by an InheritanceDefinition in FAMIX.

The new or modified attributes are:

- `accessControlQualifier`
  The access control in Java is always "public". It means that all public and protected attributes and methods are inherited by the subclass and keep their declared visibility.
- `index`
  The index is always "null" as Java has single inheritance and therefore name collisions cannot appear. Java classes can implement multiple interfaces, but as interfaces do not implement any behaviour name collisions do not cause any problems. Interfaces can

contain constants, but a class cannot implement multiple interfaces that contain constants with the same name with possibly different values.

3.12 Invocation (interpreted)

Invocation
base
candidatesAt

Figure 13: Invocation

The new or modified attributes are:

- `base`
  In Java this attribute contains the statically determinable class of the expression receiving the invocation. For example:
  ```
  MyClass r = new MyClass();
  …
  r.m();
  ```
  Then MyClass is the receiving class (and thus the base) of this invocation.
- `candidates`
  For invocations the candidates attribute holds either all methods overriding the method `base::invokes`, or if base is a Java interface it holds all methods with the same signature in classes that implement that interface

4 New classes

4.1 TypeCast

TypeCast
belongsToBehaviour (): Name
fromType (): Name
toType (): Name

Figure 1: TypeCast

This **new association** models type cast like `(MyClass)variable`.

Type casts are interesting for reengineering as they often point to problems in the design of a system. There will be an instance of this class for every type cast occuring in the source code, even if the cast is between the same types, because we are interested in all the places where casts occur.

The attributes of TypeCast are:

- `belongsToBehaviour: Name; mandatory`
  Refers to the BehaviouralEntity the cast appears in.
- `fromType: Name; optional`
  Refers to the unique name of the type the casted expression has. This is the declared type of `variable` in the above example.
- `toType: Name; optional`
  Refers to the unique name of the type the expression is casted to (`MyClass` in the above example).

5 Miscellaneous

Java does not have functions or global variables, thus those entities will never appear in a FAMIX model of a Java system. Next to that, arrays and primitive types are not handled explicitly in this FAMIX extension either.

Then there is a minor issue about file visibility. Normally a class with default visibility is visible within its package. However, when such a class is defined in the same file of another class *and* the name of the file is the same as the name of the other with the .java extension *and* theses classes are not defined in the default package, then the class is *not* visible outside the file, even to classes in the same package that are defined in other files. This issue is not dealed with in this Java language plug-in, because it’s a minor issue and in model transfers we assume a compilable system anyway.

6 Pending issues

Issues not yet covered in this plug-in are:

- Nested classes, inner classes, anonymous classes. A solution for this needs to be synchronized with other language plugins (most notably C++).
- Implicit methods. In Java there are certain methods defined implicitly. These are the default constructors and the methods this(..), super(..) (with or without parameters), which are some kind of aliases to constructors of either the current class or its superclass. If introduced in the plug-in, these implicit methods should only appear in a transfer when they are explicitly referenced by other entities.
  Implicit methods could be introduces with an isImplicit attribute for methods and interpreting this(..) and super(..) calls as calls to the respective constructors instead. But using an ImplicitMethod is consistent with the ImplicitVariable. However, this causes problems on the language independent level, as entities and associations on the language independent level (such as in Invocations) may reference this language specific entity. For that to work, ImplicitMethod should be defined on the FAMIX level rather than the Java Plug-in level.
- Static and instance initializers

7 References

[Deme99]	FAMIX 2.0, technical report, University of Berne, 1999.

[Flan97]	David Flanagan, Java in Nutshell: 2nd edition, O'Reilly, 1997.

[Gosl96]	James Gosling, Bill Joy and Guy Steele, The Java Language Specification, Addison Wesley, 1996.

Cover Pages

Achievement 2.4.1b

FAMIX Java language plug-in 1.0

1) Identification

Project Id:	Esprit IV #21975 “FAMOOS”
Deliverable Id:	D 2.2 – FINALFHB Final FAMOOS Methodology Handbook
Date for delivery:	31.08.99
Planned date for delivery:	31.08.99
WP(s) contributing to:	1
Author(s):	Sander Tichelaar

2) Abstract

This document describes the language plug-in to the FAMIX 2.0 model [Deme99] for the Java programming language [Gosl96]. It handles interpretation issues concerning Java in FAMIX and the extension of the FAMIX model for Java specific features.

3) Keywords

Object-oriented, reengineering, reverse engineering, code repository, round-trip engineering, FAMOOS, FAMIX, Java.

4) Version History

Ver	**Date**	**Editor(s)**	**Status & Notes**
0.3	24.08.99	Sander Tichelaar	First draft version released for public review.

5) Issues for future releases

See Pending issues, page 8.

6) Table of Contents

7) List of Figures

8) List of Tables