Java naar FAMIX parsen

Hans Stenten

Academiejaar 2001 - 2002
Promotor: Serge Demeyer

Universiteit Antwerpen
Universitaire Instelling Antwerpen
Departement Wiskunde - Informatica

Abstract

Software Engineers can use a diversity of Computer Aided Software Engineering-tools to re-engineer object-oriented software. As an interchange format between these tools, the FAMIX model supports multiple object-oriented programming languages.

This document handles the ways to model a Java project in FAMIX. Java projects can appear as bytecode or as sourcecode. Different ways to parse Java are investigated and compared on performance, scalability and maintainability. The existing Java plug-in is evaluated and extended or adapted where necessary.

The thesis concludes that Java bytecode is very suited to parse a FAMIX model from.

Acknowledgements

I'd like to thank everybody who helped me writing this thesis. First of all my promotor Serge Demeyer, who helped me explore the subject and corrected me if I explored in the wrong direction. He also taught me some very usefull guidelines for giving a good presentation. I also want to thank all colleagues present during all those presentations, giving me feedback on my performance. Thank you Andy, Bart, Filip, Kris, Luc and Christophe.

This paper would contain much more spelling-mistakes without my dad, who also gave me feedback on the readability of this paper, so I also have to thank him. And Bart who also gave me comments on the readability.

And of course my girlfriend, who was always there to remind me there are other important things in life, next to writing this document.

INLEIDING

Java naar FAMIX parsen

Software engineers beschikken over verschillende CASE tools om object ge-orienteerde software te gaan re-engineeren. Tools die het volledige software development process ondersteunen, zijn zeldzaam. De meeste tools zijn gespecialiseerd in een onderdeel van het software ontwikkelingsproces.

FAMIX is een metamodel om software te modelleren en het is een uitwisselingsformaat tussen verschillende CASE tools. FAMIX is een model dat software modelleert op het niveau van de programma entiteiten, zodoende kan het verschillende object ge-orienteerde talen ondersteunen. Deze thesis onderzoekt verschillende manieren om Java te parsen naar FAMIX modellen.

Het onderzoekt zowel bytecode als broncode als basis om een model van op te stellen. De voor- en nadelen van beiden worden onderzocht, en de prestaties van enkele parsers worden in detail besproken. Zo wordt de performantie onderzocht, de schaalbaarheid van de parsers naar grote projecten, de precisie van de modellen die gegenereerd worden, de onderhoudbaarheid van bepaalde parsing technieken en tenslotte de bruikbaarheid van het FAMIX model.

De reeds bestaande Java plug-in wordt kritisch besproken, en op verschillende plaatsen aangepast en uitgebreid.

CHAPTER 1

Introduction to software reengineering and FAMIX

Reengineering of software is becoming more and more important. Software systems grow, because requirements change and the user asks new functionality. The original design may have been tidy and strict, but over time a system will lose its original clean structure and turn into a Big Ball of Mud [FY00].

This is because we can't build the perfect system. Manny Lehman and Les Belady wrote down some fundamental laws of software evolution quoted by Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz [DDN02]. Two of these laws are:

- *The Law of Continuing Change*: A program that is used in a real-world environment *must* change, or become progressively less useful in that environment
- *The Law of Increasing Complexity*: As a program evolves, it becomes more *complex*, and extra resources are needed to preserve and simplify its structure.

So the system we build can't be perfect. But we can try to build a system that can keep up with the changing environment it is used in. And that can be extended when new functionality is demanded. The systems we are talking about are so called *legacy* systems. A legacy system is a system that is in use, and very valuable in his current form, that you as software engineer have *inherited*. They tend to be very large. It may have been inherited many times and contain signs of different modifications and adaptations. But a *legacy system* doesn't mean necessarily *old system*. With the rapid changing business environments and turnover of personnel, software systems can turn into legacy systems very quickly. Without losing any of their *value*.

So we don’t want to just throw them away, but adapting legacy systems to new standards or make them benefit from new technologies tends to be very hard and time consuming.

This is where reengineering techniques come in the picture. The reason we want to reengineer can be very different, but to make these changes and adaptations possible we can use some general techniques in all cases. We might want to *unbundle* a monolithic system so that we can use the separate parts in combination with others. We might want to improve *performance*. If we want to *port the system to a new platform*, we need to separate the platform-dependant code from the rest. If we want to make a new implementation we might *extract the design* first. We might want to *exploit new technologies*, libraries or standards to cut maintenance costs and preserve compatibility in the future.

To implement these goals, we have to solve some problems:

- *Improper layering*: a good layering is needed for portability and adaptability.
- *Lack of modularity*: strong coupling between modules hampers evolution
- *Duplicated code*: lazy “copy paste” behaviour makes maintenance a very complex task
- *Duplicated functionality*: leads to unnecessary code size, making it again hard to maintain
- *Missing inheritance*: duplicated code and case statements are symptoms of this
- *Misplaced operations*: operations outside instead of inside classes, result in unexploited cohesion
- *Violation of encapsulation*: explicit type cast, over usage of public attributes
- *Class abuse*: lack of cohesion, using classes as namespace

We can solve them by *restructuring*[1] and *refactoring*[2] our software system. Restructuring comprehends source code transformations to kill “spaghetti” code.

1. ***Restructuring*** is the transformation from one representation form to another at the same relative abstraction level, while preserving the system’s external behaviour. [DDN02]
2. ***Refactoring*** is the process of changing a software system in such a way that it does not alter external behaviour of the code yet improves its internal structure. [DDN02]

Refactoring is restructuring in an OO context. If we think of legacy systems as old systems, then we have to think in the “waterfall” model. Reengineering is then the last step of the waterfall. If we think of legacy systems as recent OO software systems that have to be maintained, then we have to think in an iterative model, where reengineering is a part of the software development process.

Software development tools

To solve the problems we need tools, most CASE tools do not cover the whole software development process, they focus on a specific part or problem. Some tools are for designing, others will calculate metrics to express the coupling and cohesion in our project etc.

Legacy systems also tend to be very large (millions of lines of code), so whatever solution we use, it should be scalable and extendable/flexible.

A metamodel for our software development tools

Reengineering large software systems is impossible without appropriate tools. A good reengineering tool should be scalable and it should support different techniques for the different tasks in the reengineering process. They should enable us to analyze and re-model our system, to detect design faults. And they should allow us to “measure” the system through software metrics, to discover the problem points in our system, and to give it some quality measure. To analyze the results the tools should need a common information format, so we can cumulate the information we generate with different tools.

That common information format is influenced by the metamodel used. The metamodel describes what information is modeled, and how it is modelled. The metamodel determines if the right information is available for all our reengineering tasks we want to perform. It also influences the scalability and the extensibility, and the way we can exchange information between different tools.

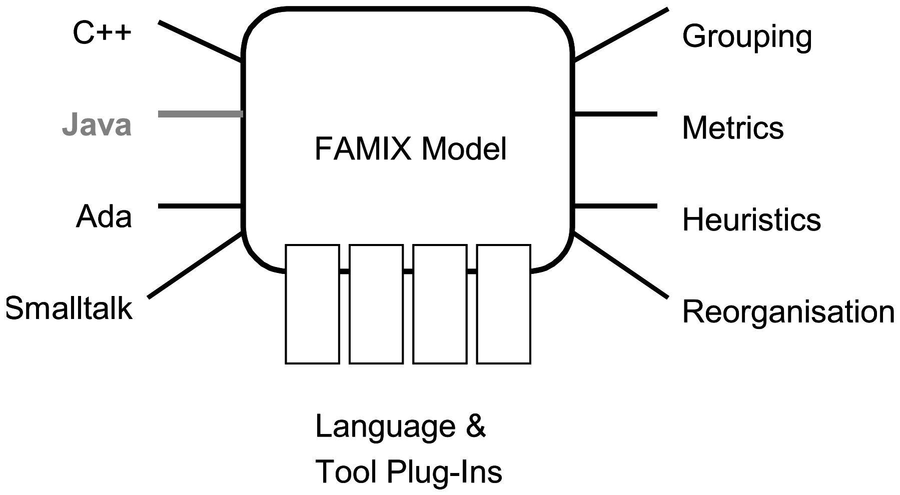
In this document we use a metamodel for object-oriented software called FAMIX. But there are many others. A very well known meta model representing software is the *Unified Modelling Language* (UML) [OMG99]. UML is the most famous metamodel aiming at object-oriented analysis and design (OOAD). However these metamodels represent the software at design level. Reengineering requires information about software at source code level. A precise mapping between the implemented software system and the model is needed. UML only gives a design model, so the information we need to reengineer our system isn't available.

Other metamodels exist, that do model the software itself, a listing of them can be found in *Modelling Object-Oriented Software for Reverse Engineering and Refactoring Chapter 2.3.2* [ST01]. The metamodels miss a discussion of the relevance of the represented information for reengineering and a discussion about the trade-offs of modelling alternatives.

What we need is a metamodel that can model our software and supports different reengineering tools. In this paper we use the FAMIX metamodel.

The FAMIX model is a metamodel for OO languages like Ada, C++, Java or Smalltalk. FAMIX was designed to be language independent, support the whole software-reengineering life cycle and to be scalable for multi-million lines of code projects. It is also an interchange format between different tools. Due to this general nature, some language specific features aren't covered. FAMIX is also made to be extensible and cope with this problem. It can be extended by language- and tool-specific "plug-ins".

FIGURE 1. The FAMIX model



Position of this document

Java is one of the languages supported by FAMIX, the language specific features of Java are covered in the Java plug-in. The complete Java plug-in can be found in Appendix A.

FAMIX was created in the FAMOOS project, and until now it was mainly used for SmallTalk and Ada projects. Languages as C++ and Java were also supported, but little experience was available to map FAMIX to them. This document focusses on Java, but will make references to C++ where possible.

This document evaluates the Java plug-in, its usability, and the process of creating FAMIX models by parsing Java.

- Is the existing Java plug-in sufficient to model the whole Java language? We give some features of the Java programming language that aren't present in the current plug-in.
- How can we create FAMIX models from Java? We examine different forms a Java project can exist in: a textual sourcefile or a binary classfile. What are the consequences and benefits of using the one or the other. Some important criteria are given to evaluate the parsing process.
  1. Correctness
  2. Precision
  3. Performance
  4. Scalability
  5. Maintainability
  6. Usability

  A framework was made so that different parsers can be evaluated and compared on all these criteria.

Structure of this document

Sparingly this chapter tries to give a short introduction to reengineering and the needs encountered when reengineering software systems, to create a context in which FAMIX can be located. Chapter 2 then discusses FAMIX in more detail, explaining what FAMIX is, and why we use it. Then the structure of FAMIX is explained, using the Java plug-in as an example. Chapter 3 then focusses on the language features that are not present in the current Java plug-in. They are explained and then suggestions are made to resolve these holes in the model.

Chapter 4 goes into the second subject of this paper: parsing of Java. The different parsers are presented, and the difference between the Java sourcecode and the Java bytecode as source for creating FAMIX models is given. Some basic introductions are given on the classfile format, to understand the conclusions drawn at the end of the chapter. Chapter 5 then gives an analysis of the different parsers and the parsing techniques. It introduces the criteria used to compare the different techniques, then describes the experiment for each criterion and discusses the results.

Chapter 6 summarizes the findings of this document and points out the work that still has to be done regarding FAMIX and creating models from software.

Finally in the Appendices the complete Java plug-in with documentation can be found, and some graphs visualizing the results found in chapter 5.

CHAPTER 2

The FAMIX modelling language

In this document the FAMIX model is used to model software. One could wonder why we would use FAMIX, why make a new standard? This chapter explains the choices made when constructing FAMIX, and why it suits our needs. Then the core FAMIX model is explained, and the way information is stored.

Finally, some suggestions are made to extend FAMIX, to fill up the holes in the model. But first let us introduce FAMIX...

And then there was FAMIX

What is FAMIX?

FAMIX stands for FAMOOS Information Exchange Model. To understand this we need to introduce the FAMOOS[1] project first, which aimed to develop a reengineering method to transform object-oriented legacy code into frameworks. The reengineering method itself is defined around a life cycle model.

1. Requirements Analysis: identifying the concrete reengineering goals

1. FAMOOS homepage: http://www.iam.unibe.ch/~famoos/

2. Model Capture: documenting and understanding the software system
3. Problem Detection: identifying flexibility and quality problems
4. Problem Resolution: selecting new software architectures to correct the problems
5. Reorganization: transforming the existing software architecture for a new release
6. Change Propagation: ensuring that all client systems benefit from the new release

Different CASE tool prototypes were made, to perform different tasks in the steps of this life cycle model. A single information exchange format was needed, to exchange information between the different tools, so they could benefit from the results of other tools.

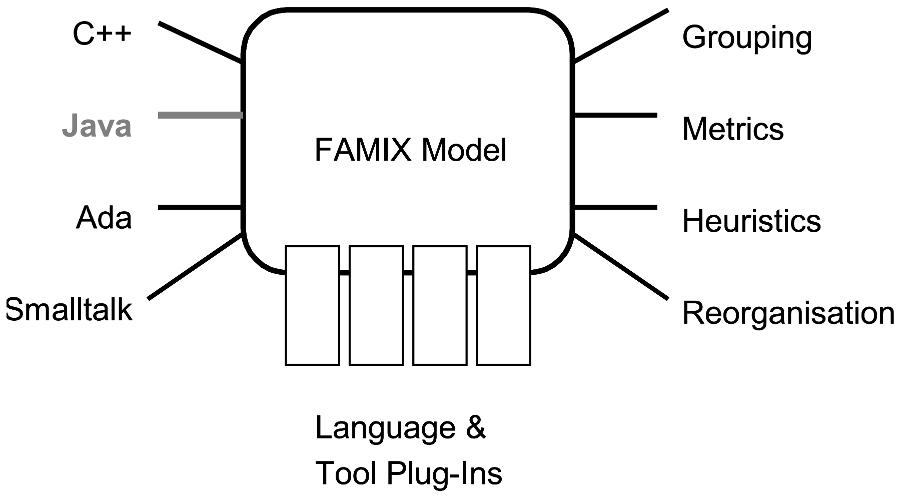
FAMOOS aimed to transform object-oriented legacy code. In the first place the Ada, SmallTalk, C++ and Java languages were targeted. This implied that the interchange format also had to be language independent.

More information on FAMOOS and the FAMOOS project can be found at the homepage of FAMOOS. We'll only focus on the interchange format FAMIX from now on. The basic FAMIX concept is given in figure 1.

FAMIX is a metamodel to model object-oriented software. It wants to support different reengineering tasks in a *language independent* way. FAMIX wants to capture the common features needed for the different reengineering tasks in *the whole reengineering lifecycle*, rather than cover all aspects of a language. Furthermore FAMIX should be *extensible*, to support language-specific features or tool-specific extensions. Last but not least it should be *scalable* to support large industrial multi-million lines of code.

One could ask why FAMIX was created. Why not use an existing modelling language? UML is widespread, and other models already exist. Sander Tichelaar [ST01] looked into this, and found that none of the existing models are suited for the task we prescribed it to do. They focus on other language paradigms or are not suited to support reengineering tasks in the whole life cycle model.

FIGURE 1. **The FAMIX concept**



Let us now look at how FAMIX was constructed. The source code is represented at program entity level, this means no information on control flow is stored in our model. So reconstructing the sourcecode from our model isn't possible. However, the program entity level allows us to model the structure and the dependencies of different classes. It also allows us to make an abstraction from language specific details, so a clean language-independent metamodel can be created.

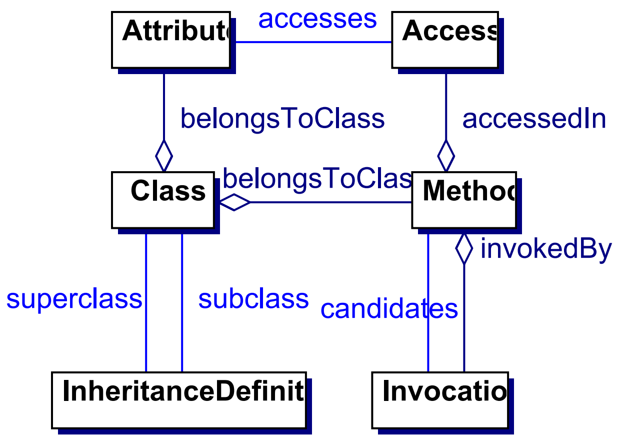
All basic elements of an object-oriented language are present: Classes, Method and Attributes. The dependency between different classes is modelled by the InheritanceDefinition (modelling the inheritance tree), the Invocation (method invocations) and Access (field accesses). You can find them in figure 2.

Figure 2 only represents the core of the model, the complete model has many more elements and can be found in Appendix A, but this illustrates of the basic idea behind FAMIX.

Every element in the metamodel has a set of attributes, and can inherit from other elements. For example a `Method` element, inherits from `BehaviouralEntity`, which is an `Entity`, which is in its turn an `Object`. All object have a sourceAnchor linking them to a specific location in the sourcefile. Attributes of a FAMIX entity can be optional or mandatory.

All objects can also be annotated by properties. A `Property` has a name and a value and is linked to a FAMIX `Object`.

FIGURE 2. The core of the FAMIX model



In this paper a FAMIX element can be called a FAMIX entity. If we want to define a new FAMIX element, or want to adjust an existing one, we will use following notation:

<entity>
<attribute> : `<type>; <optional|mandatory>` ...

Followed by a list describing the attributes.

To better understand the examples given in this chapter, let's look at the different abstract super classes in the FAMIX model.

- All elements in the model with a name (methods and variables) have a common superclass *`Entity`*, which has a name and a uniqueName. The `uniqueName` attribute is calculated by a given formula and serves as a unique external reference to that element.

- All elements in the model representing an association have a common superclass *`Association`*. It has no attributes itself, but it is the superclass of `Invocation`, `Access` and `InheritanceDefinition`.
- A last superclass in our metamodel is *`Argument`*. This represents the passing of an argument when invoking a `BehaviouralEntity` (subclass of `Entity`) or accessing a `StructuralEntity` (another subclass of `Entity`).

For a more detailed description of the different elements in the FAMIX metamodel, we refer to Appendix A.

Examples are given in the CDIF standard, for more information on the standards the FAMIX metamodel can be represented can be found in Appendix C.

To create some kind of hierarchy in the metamodel, levels of precision were defined. Stéphane Ducasse proposed to let go of this principle in the FAMIX standard, but in this document we will still use them, to be able to distinguish different FAMIX models. They will also make it easier to locate the suggestions we make in the metamodel.

More information on the 4 levels of precision can also be found in Appendix A.

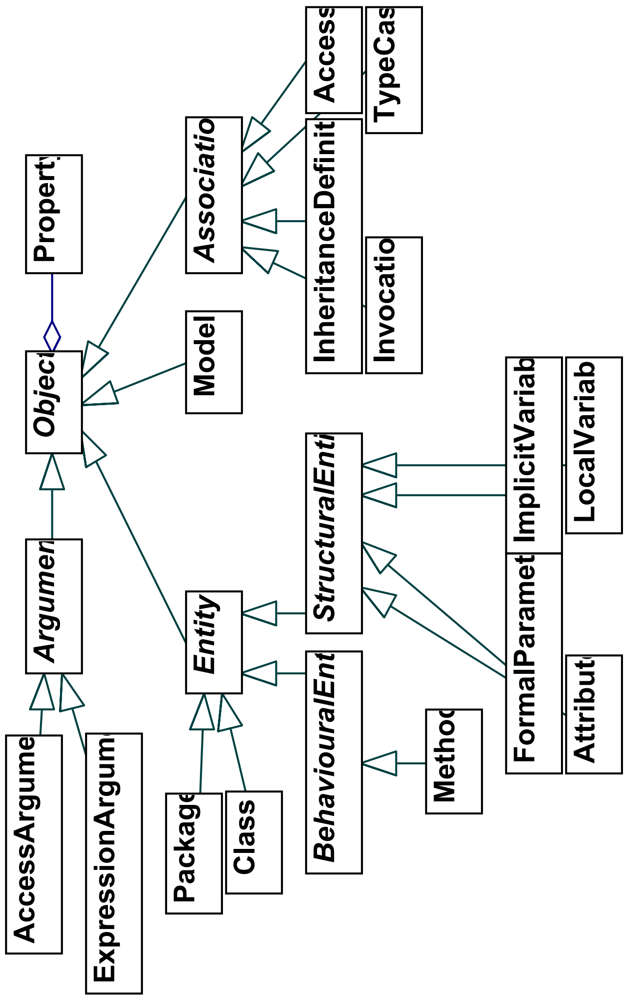
As we saw earlier, the FAMIX concept provides the possibility for plug-ins. Plug-ins for language specific features, and for tools specific additions are possible.

The next subsection handles the Java plug-in, and explains the extensions and the interpretations made to map the FAMIX model to the Java language. This chapter ends with a critical reflection on this plug-in, and suggests some extensions to the FAMIX model and to the Java plug-in. In the next chapter we'll see an example of a tool plug-in.

Extending FAMIX: The Java plug-in

We'll now describe the Java extension to FAMIX. It is important to note that Appendix A only formulates the complete Java FAMIX plug-in. The complete basic FAMIX specification is not included in this document. It can be found in [ST01]. The current Java plug-in looks like this:

FIGURE 3. The current Java plug-in



Extending the FAMIX model can be done in three manners.

1. Adding new classes to the exchange model, to model entities and associations unique to Java (or any other language).
2. Add new attributes to existing classes of the basic FAMIX model.

3. Make the descriptions of the attributes more specific or modify them, to better correspond to Java.

Some elements from the basic model such as `Function` and `GlobalVariable` do not have a counterpart in Java. They will never occur in a model of a Java system.

Other features such as the `TypeCast` isn't covered in the basic model, so a new subclass of association is added. A typecast belongs to a behavioural entity, so that's one of the attributes of the new element. The other two are, logically, fromType and toType.

Information added to the basic model includes the access flags of all Entity's, such as the final flag, the volatile, transient, native and synchronized flag. They are added to the elements they belong to.

Almost all attribute descriptions are modified to describe the more precise conditions in the Java source code that determine their value.

These few steps form the Java plug-in and extend the basic model to a metamodel capable of describing a Java system. The complete plug-in with description can be found in Appendix A.

As you will see the plug-in discussed in this document has some more changes to the basic model, this is because we found the existing Java plug-in, first defined on the FAMIX website, later updated by Sander Tichelaar in his paper, to be insufficient. In the next section we will go into some, what we think, holes in the FAMIX model and the plug-in.

CHAPTER 3

Extensions and suggestions for FAMIX and the Java plug-in

Exception

This isn't really just an extension for the Java plug-in. Exceptions is a missing concept in the FAMIX model. Exceptions are a concept present in different OO languages, C++, Java and SmallTalk. But are exceptions something we should model in FAMIX for re-engineering purposes?

The existing C++ plug-in[1] doesn't model exceptions, but in the Section "Excluded features of C++" a statement on exceptions is made:

"*Exceptions.*

If exceptions prove to be of interest we could model them as follows:

- *A class for every exception. Exceptions are classes in fact anyway.*
- *Ignoring the* `try` *block.*
- *The throwing (*`throw`*) of an exception is modelled as a call to the constructor of the exception.*

1. The C++ and Java FAMIX plug-in can be found at http://www.iam.unibe.ch/~famoos/FAMIX/Plugins/

- *The* `catch` *statement results in a definition of a local variable with the type of the catched expressions.*“.

The existing Java plug-in doesn’t mention exceptions, although exceptions are present in the Java language. Apparently exceptions are not proven to be of interest in a model. Let us therefore go into the question: “*Are exceptions of interest to model?*”. First of all we have to ask ourselves the question, why would we use exceptions in the first place. Then we have to find out what the consequences of using exceptions in our program are. Then we can determine whether we should add exceptions to our model or not.

Exception, what about them

Let’s begin with stating why we use exceptions, The Java Tutorial[2][TJT00] states three main reasons why one should use exceptions.

1. **Separating Error Handling Code from “Regular” Code**
2. **Propagating Errors Up the Call Stack**
3. **Grouping Error Types and Error Differentiation**

Exceptions separate the error handling code from the main code, making it easier to locate bugs. The normal logical flow of the program code doesn’t have to be broken with error handling code, making it harder to understand the code. Exceptions thus enhance the *Maintainability* and the *Readability* of our code.

The Java Tutorial provides a good illustration of this; two versions of a simple pseudo-code program are given, with full error handling. One without the use of exceptions, the other with exceptions.

- Without exceptions the bloat of the error handling code is about 400% of the original code. The original code is lost in the error handling code.
- When using exceptions the bloat of the error handling code is still 250% of the original code, but the error handling code is completely separated from the original code.

Exceptions also enhance the *Robustness* of a program, by providing means to support and handle out-of-the-ordinary events. Such an event is called an exception. In

2. http://java.sun.com/docs/books/tutorial/essential/exceptions/definition.html

Java all exceptions are objects, the type of those objects is `java.lang.Throwable`, or some subclass of it. There are two standard subclasses of `Throwable`, `Error` and `Exception`.

There are two types of exceptions in Java: *checked exceptions* and *unchecked exceptions*. Unchecked exceptions are objects that are an instance of `Error`, these are `OutOfMemory` exceptions and other unrecoverable exceptions. Other unchecked exceptions are instances of `java.lang.RuntimeException` (subclass of `java.lang.Exception`). This could be a `NullPointerException`. An unchecked exception is an exception that can be thrown at any time at any place in a program, because there is no way to predict the occurrence of an unchecked exception. The Java Virtual Machine can always run out of memory, and a passed object can always be `null`. Such things can be predicted.

Checked exceptions are all other exceptions, they occur in well-defined circumstances, and can thus be predicted. E.g. when reading a file, we can expect a `FileNotFoundException`.

Java expects a different handling for checked and unchecked exceptions. Checked exceptions must be handled locally in the method by a `catch` clause, or they should be declared in the method signature by including the exception in the `throws` list.

Let's consider next example:

TABLE 1. The testExceptions example

```
public class testExceptions{
   public testExceptions() throws MyException{
      // in the signature of this method a throw of
      // MyException is declared, t.i. because
      // there is no catch for the thrown MyException
      try{
         FileInputStream fis =
                  new FileInputStream("abc");
      }catch(FileNotFoundException fnfe){
         System.err.println("gotcha!");
         // unchecked exception
         throw new NullPointerException();
      }
      // checked exception
      throw new MyException();
   }
}
```

Two checked exceptions occur in this example, the two ways Java deals with them are illustrated. The throw of the `NullPointerException`, which is an instance of `RuntimeException` throws an undeclared exception. Unlike checked exceptions, unchecked exceptions can be caught, but no catch clause has to be provided, nor must they be included in the throws list of a method.

It's clear that exceptions are part of the signature of a method in Java. A method calling another method has to deal with the exceptions in the `throws` list of the called method. The invoking method has to provide suitable error handling or has to re-throw the exception further up the call stack.

If we want to model a method from a software project, we can't skip this information, it's more than an implementation issue because it tells something about the behaviour of the method. So if we want to reverse engineer the design of the software project, to examine its properties and its usability, the exceptions a method can throw are relevant information. Methods throwing exceptions also tell us something about the encapsulation of the object containing the method.

It is also possible to use the throw mechanism for other purposes than exception handling. In C++ the `exception` class in the STL can be extended, and in Java any class implementing `Throwable` can be thrown. This mechanism can be used to provide a more flexible return mechanism in methods.

To conclude we can say that exceptions can be very useful in an Object-Oriented software engineering tool environment. As we saw earlier they help to reduce the code bloat, and thus make the sourcecode more readable and better maintainable. The use of exceptions also allows to group different kinds of errors together in classes, which will result in a cleaner design. This makes exceptions especially of interest in the refactoring and the design phase.

TABLE 2. The usage of exceptions in some packages

project	throws	exception handling	#methods
eclipse	4.827	7.743	71.162
javax	765	999	14.370
java	2.448	2.038	12.380
scob	429	1	2.803

The packages used in this table are used elsewhere in this document, to analyze different parsers. They form the benchmark we use in this document, more information on the packages and the benchmark can be found in Table 4, “The performance and scalability benchmark,” on page 74.

The number of throw clauses encountered inside the package is given in the second column. The third gives us the number of methods with error handling code in its body using exceptions. Finally the total number of methods in the package is given. We can easily see that about 10% of the methods throw an exception. If we look at the percentage of methods that have exception handling code, we get again 10%. If we take into account that in the number of methods, all accessor functions and constructors are included, we see that exceptions are a very common feature in Java.

Exceptions should be included in our FAMIX model. Exceptions are more than just an advanced control flow feature. It’s an OO technique that allows interrupting the normal control flow, handing control over to another entity without necessarily returning. They could be regarded as a sort of return value, since exceptions are instances of classes that can contain any data.

Adding exceptions to the model

To adapt the FAMIX model to include exceptions, some adjustments have to be made, and a new FAMIX element is needed.

First of all we have to add the exceptions contained in the throws list of a method to the Method entity. Secondly to keep track of the throw statements in our method body, we need an Entity similar to the Invocation entity keeping track of method invocations. We'll introduce the Throw element.

Extending `Method`

Extend the FAMIX Method entity in the Java plug-in.

Method
exceptions(pos Integer) : `String; Mandatory` `#NEW`

- `exceptions: 0..N Strings; Mandatory`

  For methods the exceptions attribute holds all exceptions declared in the `throws` list. It is a list of the `uniqueNames` of the classes of the exceptions.

CDIF example; based on Table 1, "The testExceptions example," on page 17

```
(Method FM3
   (sourceAnchor #[file "testExceptions.java"
                                 start 10 end 23|]#)
   (name "testExceptions")
   (uniqueName "test::testExceptions.testExceptions()")
   (belongsToClass "test::testExceptions")
   (signature "testExceptions()")
   (accessControlQualifier "public")
   (declaredReturnType "")
   (declaredReturnClass "")
   (hasClassScope -false-)
   (isAbstract -false-)
```

```
    (isConstructor -true-)
    (isFinal -false-)
    (isNative -false-)
    (isSynchronized -false-)
    (isPureAccessor -false-)
    (exceptions #["MyException"|]#)
)
```

Adding a `Throw` element

The FAMIX standard itself should be extended with the *Throw* element.

Throw
thrownBy() : `Name; mandatory`
throws() : `Qualifier; mandatory`
base(): `Name; mandatory`
hasArguments() : `Name; optional`
isDeclared() : `Boolean; optional`

A throw represents the definition in the source code of a BehaviouralEntity throwing an exception. This exception can be any class inheriting from exception in C++, or any class implementing throwable in Java. For each throw a separate throw-association should be generated.

We chose to make `throw` a direct subclass of Association, because a throw represents a different kind of association between two classes, it associates code with its error handling code.

Another possibility was to make throw a subclass of `Invocation`. Since most attributes of the `Throw` element are analogous to their counterpart in `Invocation`, this would be an option. In that case a Throw would just be a specific case of an Invocation, but because throw is a language independent feature and because a throw clause will always call the constructor of the error class we thought it was better to model it as an independent element. This doesn't limit the extension possibilities of the Invocation element.

If one would decide for some specific reasons that a throw is just an invocation of the constructor of an exception class, the isDeclared attribute specific for the `throw` element wouldn't be needed, and one could still just model the throw clause as an `invocation`, and ignore the `throw` element.

Throw is a concrete class inheriting from Association. Besides inherited attributes, it has the following attributes:

- `thrownBy: Name; mandatory`

  Is a unique name referring to the BehaviouralEntity doing the throw. It uses the `uniqueName` of the entity as a reference.
- `throws: Qualifier; mandatory`

  Is a qualifier holding the `signature` of the constructor of the class thrown.
- `base: Name; mandatory`

  Is the unique name of the entity where the thrown entity is defined on. Null means unknown. Together with the `thrown` attribute, this attribute constitutes the unique name of a behavioural entity.
- `hasArguments: 0 .. N Name; optional`

  A throw can have arguments. The `hasArguments` attribute denotes the uuid's of the arguments.
- `isDeclared: Boolean; optional`

  An exception can be declared for a BehaviouralEntity, telling the invoking BehaviouralEntity that this exception wasn't handled and it should be handled by the invoking BehaviouralEntity. In Java this means a thrown exception is declared if it appears in the throws list of a method. Otherwise it isn't declared.

CDIF example; based on Table 1, "The testExceptions example," on page 17

```
(Throw FM12
   (sourceAnchor #[file "testExceptions.java" start 22
end 22|]#)
   (throws "MyException()")
   (base "MyException")
   (thrownBy "testExceptions.testExceptions()")
   (hasArguments "null")
   (isDeclared -true-)
)
```

```
(Throw FM13
   (sourceAnchor #[file "testExceptions.java" start 17
end 17|]#)
   (throws "NullPointerException()")
   (base "java::lang::NullPointerException")
   (thrownBy "testExceptions.testExceptions()")
   (hasArguments "null")
   (isDeclared -false-)
)
```

Inner Classes

Inner classes are currently not covered in the Java plug-in. Some forms of inner classes are also available in C++, so we might consider adding them to the FAMIX standard rather then to the Java plug-in. First of all let's figure out what inner classes in Java are. Then we can consider adding them to our model or not.

A good description of inner classes in Java can be found in Java In a Nutshell 3rd edition, Chapter 3. [NUT99] There are four types of inner classes in Java.

- Static member classes
- Member classes
- Local classes
- Anonymous classes

The four types of inner classes in Java

- A *static member class* is a class that is a static member of another class. It behaves as an ordinary class except it can access all static variables of it's containing class. A nested interface must always be a static member class.

  Example:
  ```
  public class A{
     public static class B{ ...implementation }
     ...
  }
  ```

- A *member class* is a class that is an instance member of another class. It is always associated with an instance of the enclosing class. It has access to all fields of the enclosing class, private, static and non-static.

  Example:
  ```
  public class A{
     public class MyClass{ ...implementation }
     ...
  }
  ```

- A *local class* is a class that is defined within a block of code. It is only visible inside that block of code (e.g. a method). As member classes, local classes are contained in an enclosing class, so they can also access all fields of the enclosing class. Additionally a local class can access any final local variable or formal parameter within the enclosing code block.

  Example:
  ```
  public MyInterface methodA(){
     class MyClass implements MyInterface {...implementation}
     return new MyClass();
  }
  ```

  `MyClass` is a local class implementing an interface `MyInterface`.

  `methodA` returns and object implementing `MyInterface`. So although the class `MyClass` doesn't exist anymore outside its scope block, its instances can exist outside the scope block as implementations of `MyInterface`.

- An *anonymous class* is a local class without a name. An anonymous class definition is part of an expression in Java, it is no statement as other inner classes. An anonymous class combines the syntax of a class definition with the syntax for an object allocation.

  Example:
  ```
  public MyInterface methodA(){
     return new MyInterface(){ ...implementation };
  }
  ```

  This example does the same as the previous example, only this time the class implementing `MyInterface` has no name. Again the instance can be referenced outside the scope block it is created in as an implementation of `MyInterface`.

In fact we can divide these inner classes in two groups: the one's that can be referenced outside the enclosing class, and the one's that can't. Static member classes, and member classes, when declared public or protected, can be referenced and even overloaded outside the class. Local classes can only be referenced in their scope block, and an anonymous class has only one instance that is used immediately after it is defined. But as we saw in the examples, it is possible to bypass this restriction.

A local class can implement an interface, and its instances can be passed on outside its own scope as implementations of that interface. They could also be returned as an Object, but this would cause loss of most of the instances functionality.

Nice to know is that it's even possible to call the `getClass()` method on these instances. This results in a class with a name corresponding the classfile created by the compiler for the local class of anonymous class the object is an instance of.

The way the compiler handles these different innerclasses is interesting to note. All innerclasses are compiled as if it were top-level classes. The compiler takes care of all the implications this has on the bytecode, it adds extra parameters and methods where necessary. The resulting class files look like this:

```
<enclosing classname>$<inner classname>
```

In case of an anonymous class `<inner classname>` a number is used, creating all the $x files encountered when dealing with Swing applications.

Let's now look into how inner classes are and can be used, and what their benefits are to determine their relevance.

Use and relevance of inner classes

Inner classes are new to Java, before Java 1.1 there was no such thing in Java. The Inner Classes Specification of Sun[3], gives us some examples for using inner classes, and they are a good illustration of how they are used in reality.

- First of all inner classes help to clarify source code, since they allow you to declare classes closer to the object they manipulate. Many classes are used solely in combinations with other classes anyway, because they were created for that reason. Inner classes are a technique that can be used to resolve such situations in a more tidy way. To clarify this position we'll give an example where it wasn't used.

3. http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html

The JFC[4] library (`javax.*`) uses the Event - Listener model. Where event objects are fired when some action occurs, e.g. a button click, events that are handled by specific listener objects. Such an event or listener is always linked to a component, such as a table, a textfield or any other widget. When we are for example working with a `javax.swing.table.TableModel` we have to provide an implementation for the `javax.swing.event.TableModel-Listener` interface, to handle all `javax.swing.event.TableMod-elEvent` objects created when the data in the TableModel is changed. Important is to note that the last two are only used in combination of a Table-Model, yet they are defined in another package, which is illogical. Since they are only used by the TableModel class, one could use inner classes.

They could be declared as a nested interface Listener, and a nested class Event in TableModel, resulting in:

```
javax.swing.table.TableModel
javax.swing.table.TableModel.Listener
javax.swing.table.TableModel.Event
```

TableModel would then be a fully self-contained class with all necessary support classes and interfaces, instead of having it all spread out over several packages and three files.

- A second use of inner classes, the anonymous classes, is very common used to link individual GUI components to application methods. Again the Event-Listener model triggers this use of anonymous classes. We define customized Listeners for each GUI component, e.g. the “OK” button needs another listener then the “Cancel” button.

```
public class MyFrame extends Frame{
   public MyFrame(){
      Button ok = new Button("Ok");
      ok.addActionListener(
         new ActionListener(){
           public void actionPerformed(ActionEvent e){
              // do something
              System.out.println("ok clicked");
```

4. Java Foundation Classes

```
                }
              }
          );
          ... add a Cancel button in the same way
      }
  }
```

These classes are never referred to later on by any other class. Giving these classes a name would only bloat our model, increasing the risk for name conflicts. What name should we have given this class? OkButtonActionListener? Since they are never referenced further on, there is no point in naming this class.

Analogous anonymous classes are often used to provide a local implementation of an interface, such as the `Runnable` interface to define a thread object.

- A third example of the use of inner classes is providing a default implementation of an interface. An interface can have inner classes as well, so we can define a member class implementing its enclosing interface.

  Example:

```
interface MyInterface{
   public class defaultMyInterface
                              implements MyInterface{
      public void implementMe(){
       System.out.println("a default implementation");
      }
   }
   public void implementMe();
}
```

We can conclude that inner classes are a valuable addition to the Java language that can make code more readable and efficient. Making inner classes something valuable for our CASE tools.

Inner classes in other languages

Now we know what inner classes are in Java and how they are used, let's look at inner classes in C++. Inner classes in C++ are fairly similar to inner classes in Java.

Static member, member classes and local classes can be declared in C++ with the same syntax used in Java. The only differences are the anonymous classes, there is no such thing in C++. There are however anonymous unions and structs in C++. An example of an anonymous struct:

```
// Example of an anonymous structure
struct phone{
   int  areacode;
   long number;
};

struct person{
   char   name[30];
   char   sex;
   int    age;
   int    weight;
   struct phone;    // Anonymous structure; no name needed
} Jim;
Jim.number = 1234567;
```

This example comes from the Microsoft Developer Network website[5]. This is an important remark, because in the C++ standard only anonymous unions are defined. Anonymous structs are not in the standard, on the programming ground this means it is a Windows feature, supported by very little Unix compilers. Anonymous union are low-level features that can hardly be called some sort of “class”.

Conclusions and remarks

Member classes, static member classes and local classes are features that can be modelled in the FAMIX model itself, rather then in a language specific plug-in. Member and static member classes behave much like any other top-level class, and should be modelled as an ordinary class. Nested classes have a nesting tree next to the inheritance tree. A nested class has at least one enclosing class, and this affects the scoping rules inside the nested class. Because it has access to all fields and methods of the enclosing classes, also the inherited one’s.

5. http://msdn.microsoft.com/

The problem here is the implicit “`this`” variable. Consider next example:

TABLE 3. The InnerClass example

```
package examples;
public class OuterClass{
   int a;
   public class InnerClass{
      public InnerClass(){
           a = 2;
           OuterClass.this.a = 2;
      }
   }
}
```

Both statements are equal. “`a`” is an instance field of `OuterClass`, so to access it we need to access the “`this`” variable of our `OuterClass` object. “`this`”, however points to an `InnerClass` object at the moment. So we need a new notation for this special case of an implicit variable. By specifying the name of the enclosing class we can access the right “`this`”. In case of name conflicts in nested classes this is the way to access a class attribute or method. This implies that the name attribute of an ImplicitVariable can be more than “super”, “this” and “class”.

Again it is interesting to note how the Java compiler solves this problem. We saw that all inner classes are compiled as top-level classes. The compiler add variables to all inner classes, representing the `this` variables of the enclosing classes. These variables are called `this$0`, `this$1` ... .

Since we are encountered with the whole local class and anonymous class problem again, the solution used by the Java compiler seems a good one, although rather cryptic at first. However, since we store the nesting information in NestingDefinition elements, if we come across an ImplicitVariable called `this$i` we have to follow i NestingDefinition elements in our model to find the class corresponding to `this$i`.

In this case however the solution provided in the Java language specification is more useful in our FAMIX model. In case of nested classes, we can just prepend the uniqueName of the class whose “this” variable is referenced. This would result in:

```
(ImplicitVariable FM30
   (name "OuterClass::this")
   ...
)
```

Adding inner classes to FAMIX: adjust Class

Adding support of member classes and static member classes can be achieved by some minor changes to our model. Adding an extra attribute to the Class entity and changing the description of the `belongsToPackage` attribute to cover nesting.

Class	
isNested(): `Boolean; optional`	#NEW
belongstToPackage() : `Name; optional`	#Interpreted

- `isNested: Boolean; optional`

  A predicate telling if the class is nested. I.e. is the class a member class of another enclosing class? The class can be static or not.

- `belongsToPackage: Name; optional`

  This attribute refers to the `uniqueName` of the package to which the class belongs, defined by the package statement at the beginning of a Java source file defining the class. In case of a nested class, this attribute refers to the `unique-Name` of the enclosing class. The enclosing classes are considered "packages". Because there is no difference in notation between an enclosing package and an enclosing class in Java either, a nested class cannot have the name of a package at the same depth. So given a class ClassA.ClassB, ClassA can be a package containing a class ClassB. Or ClassB can be a nested class in ClassA, but never both.

CDIF example based on Table 3, "The InnerClass example," on page 28:

```
(Class FM2
   (name "InnerClass")
   (uniqueName "examples::OuterClass::InnerClass")
   (isNested -true-)
   (belongsToPackage "examples::OuterClass")
)
```

Adding inner classes to FAMIX: add the NestingDefinition element

When adding inner classes to our FAMIX model, we need to store information on the nestingstree, just as we did for the inheritance tree before. So we need an entity similar to an InheritanceDefinition to express the nesting relations between two class entities. Such an entity can express the access control of the nested class, and can note whether it is a static member class or an instance member class.

Therefore we introduce the NestingDefinition to the FAMIX model.

NestingDefinition
innerclass(): `Name; mandatory`
enclosingclass(): `Name; mandatory`
accessControlQualifier(): `Qualifier; optional`
isLocal(): `Boolean; optional`
isStatic(): `Boolean; optional`

- `innerclass: Name; mandatory`

  Is a unique name referring to the class that is nested. It uses the uniqueName of that class as reference.
- `enclosingclass: Name; mandatory`

  Is a unique name referring to the enclosing class. It uses the uniqueName of that class as a reference.
- `accessControlQualifier: Qualifier; optional`

  A string determining the access control for the nested class, just as it would for a method of the class.
- `isLocal: Boolean; optional`

  A predicate telling if the nested class is a local class, thus only visible inside a certain scope block.
- `isStatic: Boolean; optional`

  A predicate telling if the nested class is a static member class of the enclosing class or an instance member class.

CDIF example based on Table 3, “The InnerClass example,” on page 28:

```
(NestingDefinition FM15
   (innerclass "examples::OuterClass::InnerClass")
```

```
    (enclosingclass "examples::OuterClass")
    (accessControlQualifier "public")
    (isStatic -false-)
)
```

Now we have added nested top-level classes (member and static member classes) to our model. We haven’t covered local classes yet. Since these classes can’t be referenced outside their scope block, their relevance to our model is less. For anonymous classes in Java this holds even more, because they can’t be referenced at all.

But we can’t skip them because they are, although usually small, valid classes, with member methods and fields, invoking and accessing other classes. There are several problems with Local classes that do not occur with member classes. First of all local classes can occur in any code block. This doesn’t have to be a method, it can also be a static or instance initializer. The name of a local class is only unique in its code block. So different local classes with the same name can occur in a single method.

The problem will be how to give those classes a qualified `uniqueName` we can use to reference it by. Consider this situation: we’ll use anonymous classes but a similar situation with local classes would be possible.

```
public class MyFrame extends Frame{
   public MyFrame(){
      Button ok = new Button("Ok");
      ok.addActionListener(
         new ActionListener(){
           public void actionPerformed(ActionEvent e){
               // do something
               System.out.println("ok clicked");
           }
         }
       );
      Button nok = new Button("Cancel");
      nok.addActionListener(
              new ActionListener(){
                public void actionPerformed(ActionEvent e){
                    // do something
                    System.out.println("cancel clicked");
                }
              }
            );
...
```

What we have here are two methods actionPerformed in two separate anonymous classes. To be able to model these methods and the accesses and invocations they include in a non-ambiguous way, we have to find a formula to create a unique name.

One possible solution would be to treat them as ordinary nested classes, with the `isLocal` attribute set in the NestingDefinition. This would result in possible different class entities in the same package with the same name.

Then our model wouldn't be unambiguous any more. Imagine two local classes each named LocalClassA. Our model would then contain two class entities with the same `uniqueName`. If we would model a method inside one of the classes, its belongsToClass attribute would be set to the same value. But we wouldn't be able to determine whether this method is a method of the first local class or the second.

Of course each of the modelled classes is still unique by its `uuid` attribute. But integrating the uuid in the `uniqueName` attribute of a class, but that attribute would become much too bloated, and would affect the readability of our model. To find a solution to our problem we have to think in this direction though. We have to find a way to keep the `uniqueName` of a class entity "unique" in a model.

We could use a variable that keeps track of the number of local classes encountered in our model, and append that variable after each name of a local class. This would give all local classes a unique uniqueName that can be referred unambiguously. Anonymous classes in Java would then be a class with the value of that variable as name. Although this solution isn't ideal, it has proven to work for the Java compiler, since it has the same problem of finding a way to reference the classes unambiguously.

Another possibility would be to use the uuid (completely or partially) in the uniqueName, but this would make the uniquename even less readable than the first solution we gave. Solutions replacing the uniqueName completely by the uuid violates the uniqueName formulae we defined for our model, in an even greater way then our proposition. Although it's not ideal, the Java compiler solution looks like the optimal solution.

When using bytecode parser there is another option: we can retrieve the filename of the classfile created for an inner class by the compiler, and use the name created by the compiler to denote the inner class in our model. This solution is similar to the first solution we suggested, but has the advantage that we will always create the same "uniqueName" for an innerclass with different parsers. When using the first solution it's possible that some innerclass gets two different names when using two parsers.

InheritanceDefinition

A Java class always extends its superclass, and can implement different 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, no lookup procedure in the inheritance tree is needed. Interfaces can inherit from multiple interfaces. In the Java plug-in classes and interfaces are both Famix Class Entities, but the attribute isInterface allows us to differentiate between interfaces and classes. This should be extended for an InheritanceDefinition.

If the implemented interface is not available in our FAMIX model, the only information we have on the relation between the interface and its implementing class are the FAMIX Class entity modelling the class, and the InheritanceDefinition entity. All information that it involves an implementation and not an extension is lost. This information is important to include in our model, because an `extends` clause implies inherited functionality. While an `implements` clause does not, it forces implementation.

By adding an attribute isImplementation to the InheritanceDefintion entity, we can point out whether the inheritance is an implementation or an extension.

InheritanceDefinition
isImplementation(): `Boolean; optional`

- `isImplementation: Boolean; optional`

  Is a predicate telling whether the inheritancedefinition is an implementation or an extension. I.e. is the superclass an interface, or a class.

CDIF example the `Integer` class in the `java.lang` package, extending `Number` and implementing `Comparable`.

```
(InheritanceDefinition FM20
   (subclass "java::lang::Integer")
   (superclass "java::lang::Number")
   (accessControlQualifier "public")
   (index 1)
   (isImplementation -false-)
)
(InheritanceDefinition FM21
   (subclass "java::lang::Integer")
   (superclass "java::lang::Comparable")
   (accessControlQualifier "public")
   (index 1)
   (isImplementation -true-)
)
```

Static & instance initializers

All class and instance fields of a class are initialized automatically to the values shown in Table 1.

TABLE 4. Java Data Type Default Values

Type	**Default value**
boolean	false
char	\u000
byte / short / int / long	0

TABLE 4. Java Data Type Default Values

Type	Default value
float / double	0.0
reference type	null

These values are guaranteed by Java. Often this default value is not appropriate, Java has three techniques to initialize class and instance fields. Some better known than others.

- field initialization expressions
- constructors
- static and instance initializers

A field initialization expression looks like this:

```
public static double Pi = 3.14159;
```

We can have class fields or static fields that are initialized only once for a class, and we can have instance fields that are initialized for each instance of a class.

The second technique is well known, the constructors of a class are methods called to initialize an object.

The third one is more obscure. Static initializers are a standard technique in Java, and since Java 1.1 also Instance initializers are allowed. Initializers are arbitrary blocks of code for the initialization of fields. A static initializer is the keyword `static` followed by a block of code between curly braces.

```
public class Initializers{
   private static double PI;
   static{
      double temp = 0.0;
      // some difficult calculation
      PI = temp;
   }

...
```

Static initializers can appear anywhere a field or method definition can occur. Multiple static initializers can appear within a single class. They can be used as suggested in the example to perform a non-trivial calculation to initialize the static fields of a class.

Instance initializers are like static initializers, they look just like them, without the leading `static` keyword.

```
...
   private double[] data = new double[100];
   {
      for(int i = 0; i < 100; ++i){
          data[i] = /* some non trivial calculation */;
      }
   }
}
```

They too can appear anywhere in the code where a field or method definition can, and can be used for the same purpose as static initializers. In practice the use of instance initializers is rare. One place it can be, and is, used is in combination with anonymous classes. Since these can't have a constructor, because a constructor requires a class name, one can use instance initializers to initialize some fields in an anonymous class.

Both initialization expressions and initializer blocks are not easy to model in FAMIX. Since method invocations and field accesses have mandatory attributes invokedBy and accessedIn that require some containing Method. These initialization techniques have no containing method.

To solve this problem, we should look at the Java compiler again. Because the JVM faces a similar problem as we do. When should the JVM execute these initializers? The answer is for each instantiation for the instance initializers and once for the class initializers (expression and block).

The compiler solves this problem by creating a `<clinit>` method for the static initializers. That method is executed once when the class is loaded into the JVM. All static initialization expressions and initializer blocks are concatenated in that method in the order they appear in the source code.

The instance initializers are treated in the same way, but the compiler adds them in the beginning of each constructor or to one general `<init>`[6] constructor that is called by other constructors.

We can use the same strategy to solve our problem. When faced with field initializers, we add a clinit method for the static initializers, and an init method for the instance initializers, both "constructors" that can contain our accesses and invocations.

```
public class Initializers{
   private static double PI = 3.14159;
   private double result = 0.0;
}
```

In CDIF this would result in:

```
(Method FM2
   (name "<clinit>")
   (uniqueName "Initializers::<clinit>()")
   (signature "<clinit>()")
   (isConstructor -true-)
)
(Access FM3
   (accessedIn "Initializers::<clinit>()")
   (accesses "Initializers::PI")
   (isAccessLValue -true-)
)
(Method FM4
   (name "<init>")
   (uniqueName "Initializers::<init>")
   (signature "<init>()")
```

6. all constructors are renamed to <init> by the compiler

```
)
(Access FM4
    (accessedIn "Initializers::<init>()")
    (accesses "Initializers::result")
    (isAccessLValue -true-)
)
```

Import

In the C++ plug-in, an Include entity is added. It is an association added to create a rough overview about the dependencies between different source files in a system. This is an interesting addition for two reasons:

- It is very easy to extract this information from our model
- It gives a good overview of the general dependencies between source files

In Java we could use the import statements in the beginning of a source file, to get an overview of the dependencies between different classes. This could be added to the entities supported for Level 1 precision.

When using source files to create our FAMIX model, we can encounter `import java.io.*;` statements. In that case we can only model a dependency between a class and another package (`java.io`). When using byte code files, this information is lost, because no packages are used inside the bytecode.

Bytecode is much more detailed than the sourcecode, all classes referenced inside a class are contained in the constant pool by their full names. This is done to avoid any name lookup procedure, which make sourcecode parsers slow. So when constructing a FAMIX model from bytecode, we would only have dependencies between two classes, never between a class and a package.

We suggest adding an Import entity to the Java plug-in.

Import
importingClass(): `Name; mandatory`
importedEntity(): `Name; mandatory`
isPackage(): `Boolean; optional`

Import is an Association, it models the classes or packages a class imports. It gives a rough overview of the dependencies between different classes in the system, when refactoring this information can be used to locate the heavily used classes.

- `importingClass: Name; mandatory`

  Refers to the uniqueName of the class importing another class
- `importedEntity: Name; mandatory`

  Refers to the uniqueName of the package or class imported. This attribute is called importedEntity, because in Java we can import a specific class or a whole package.
- `isPackage: Boolean; optional`

  A predicate telling whether the importedEntity is a package or a class, by default this is false.

CDIF example:

```
package examples;
import java.io.File;
public class ImportExample{
}

(Import FM4
   (importingClass "examples::ImportExample")
   (importedEntity "java::io::File")
   (isPackage -false-)
```

)

Conclusion

We presented FAMIX in this chapter, and pointed out that it is easy to extend it by a plug-in to support the Java language. We also argumented that the existing plug-in, and even the basic model have some holes in them. Features such as exceptions and inner classes are widespread in object-oriented programming. And although they aren't necessarily useful to add to our model, a standard that aims to be flexible enough to be an interchange standard between different tools, supporting reengineering tasks throughout the whole lifecycle, should include all common features of that can influence such a task. Because situations where they are of importance can always occur.

The suggestions made aims to be a constructive contribution to the discussion on the further development of FAMIX. Some of the suggestions on exceptions and innerclasses could become part of the basic FAMIX definition, because they are concepts also present in other OO languages such as C++ or SmallTalk.

CHAPTER 4

Parsing Java

Now we know what FAMIX is, and what it tries to do, we have to know how to build our FAMIX models. We need to extract information from our OO software projects. In this document we describe this process for projects written in Java.

But what is the meaning of the term “Java”? “Java” has two meanings: on the one hand, Java as a programming language. On the other hand, the Java Virtual Machine (JVM) is meant. The latter is not necessarily targeted by the Java language exclusively, but may be used for other languages as well (e.g. Eiffel [CCZ97], Ada [Taf96], Python, SmallTalk...). In the first case Java is represented as “source code”, in the second case Java is represented as a Java Class file. We assume the reader to be familiar with the Java language, and to have a general understanding of the Virtual Machine.

In the first section we handle the most straightforward case, where we have the sourcecode of a project. And we want to build FAMIX models of that project for our CASE tools. We discuss some techniques, and the problems we encounter.

But as software engineers building or supervising large software projects, we have to face the fact that we can’t build everything ourselves. We have to buy third party components. Buying implies a commercial action, so it’s fair to suppose we only obtain the bytecode of that component for copyright matters. But we still want to produce FAMIX models of those components, to use them in CASE tools. We might want to measure and verify a component. The second section handles this case. First it gives an introduction to the classfile format, then we describe some techniques. Next we'll see that the problems we encounter in bytecode are completely different from the ones we handled in the source code.

The end of this chapter summarizes the conclusions we can draw.

Source Code

Many of the tasks we want to do when reengineering need the sourcecode of the system we're reengineering. The complexity of methods is calculated from the control flow of the system, and is best calculated from the sourcecode. If we want to refactor our system, to improve it, or to adapt it to the new requirements, we have to alter the source code, so without it this wouldn't be possible.

The source code contains too much information for what we need. The FAMIX model is specified at the program entity level, because it is language independent. So the key question for our source code parser will be: what is the cost of the superfluous information the parser provides. Most parsers will generate an abstract syntax tree. This gives us much more detail than we need for FAMIX. An *AST* is a representation of the source code, containing information on the control flow and is thereby language specific. Given a complete AST of a project, one can reproduce the source code.

In this section we'll discuss two parsers tested in this document: ANTLR and Barat.

ANTLR

The ANTLR is an open source project, it has its roots in the PCCTs project, a parser-generator project at Purdue University in 1988. Terrence Parr a PhD at Purdue university is the driving force behind ANTLR. Antlr stands for ANother Tool for Language Recognition. It is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing C++ or Java actions.[1] For this paper two Java grammars, one for Java 1.2 and one for Java 1.4 were tested.

1. Quoted from "What's an ANTLR?" http://www.antlr.org/about.html

ANTLR creates an AST of the source code that can be represented in an AST. Even graphical representations of this tree are provided.

None of the Java grammars sufficed for generating FAMIX models, the parsers generated by ANTLR from the given grammars are syntax parsers. This means incorrect Java source files can be parsed and represented. The next example illustrates this.

TABLE 1. A syntactically correct file

```
class Test{
   int i;
   method(){
      i = "test";
   }
}
```

This file contains two errors, `method()` has no return type, and the assignment `i = "test";` is not valid. ANTLR doesn't detect this, because there are no errors against the Java syntax. Of course ANTLR also accepts correct Java source files, but in our FAMIX model we need more than just syntactic information; we need semantic information. To create the uniqueName of a FAMIX entity, information on the type of the entity and the package nesting is needed. This information is not present in the AST created by the ANTLR parser. The import statements are not processed, so it is impossible to find the correct `uniqueName` of types.

The ANTLR parser can't be used to generate FAMIX models, since it can't even parse correct entities for Level 1 extraction.

Other parsers, similar to ANTLR are the CUP[2] parser generator for Java and the well known JavaCC[3] parser generator. Both parser generators have the same defects as ANTLR, they are pure syntax parsers.

2. More information on CUP can be found at: http://www.cs.princeton.edu/~appel/modern/java/CUP/
3. More information on the JavaCC can be found at: http://www.webgain.com/products/java_cc/

Barat

The second sourcecode parser used in this document is Barat[4][BS98]. Barat is a front-end for Java that supports static analysis of Java programs.

Barat is class oriented, meaning that it takes a class name as argument when invoked, and does a name lookup procedure on this classname, in the current classpath. So the source files can be located anywhere in the system, even inside a zip of jar archive.

Once loaded, Barat builds a complete abstract syntax tree from Java source code files, enriched with name and type analysis information. Barat is structured as a framework that supports traversals of abstract syntax trees using visitors and attributes, and provides additional features such as parsing comments as tags, access to parent nodes in the abstract syntax tree, and regeneration of source code. For users of Barat, there is no explicit distinction between phases of loading, parsing, and analyzing Java source code: All actions that need to be performed for building the AST of a Java program are transparent to clients of Barat and are triggered on demand.

Barat is an open source project of Boris Bokowski and André Spiegel of the Freie Universität Berlin.

It is written using the GJ generic Java compiler. And uses the BCEL parser for parsing class files.

More information can be found at the web site:

http://sourceforge.net/projects/barat

This is of course much more information than we need for parsing FAMIX, Barat gives us a complete annotated AST. The visitor pattern can be used to visit the AST, by implementing our own visitor we can visit the nodes of the AST that contain information we need for FAMIX, and skip the others.

Since Barat creates a tree with more information than we need, creating FAMIX from and Barat AST is very simple. Most FAMIX entities have a corresponding visit-method. For LocalVariable we have visitLocalVariable, for Invocation visitMethodCall... An interesting property of the Barat parser is that it also provides visitors for InnerClasses and AnonymousClasses.

4. ‘Java Barat’ (West Java) is one of the three parts of the island Java. The other two are called ‘Java Tangeh’ (Central Java), and ‘Java Timor’ (East Java).

Byte Code

In the introduction we noted that sourcecode isn't always available. And parsing of bytecode should be faster by intuition. These two things make it interesting to look into the potential of bytecode parsers. Bytecode is always available, otherwise there would be no component to use. In the next chapter the performance advantage of bytecode parsers will be examined.

This section will show how information can be extracted. First it gives an introduction to the classfile format used to store the java bytecode. The key question with byte code will be: can the program entity information put in by the compiler, be extracted? All essential information (e.g. no comments) in the source code is compiled into the bytecode by the compiler. As you know extracting information on the program entities from compiled code is not self-evident. In other languages (e.g. C++) this would be very hard, to impossible. In the second sub-section we will explain why it is possible to do that with Java bytecode, and why it is useful to use java bytecode as information source for re-engineering tools.

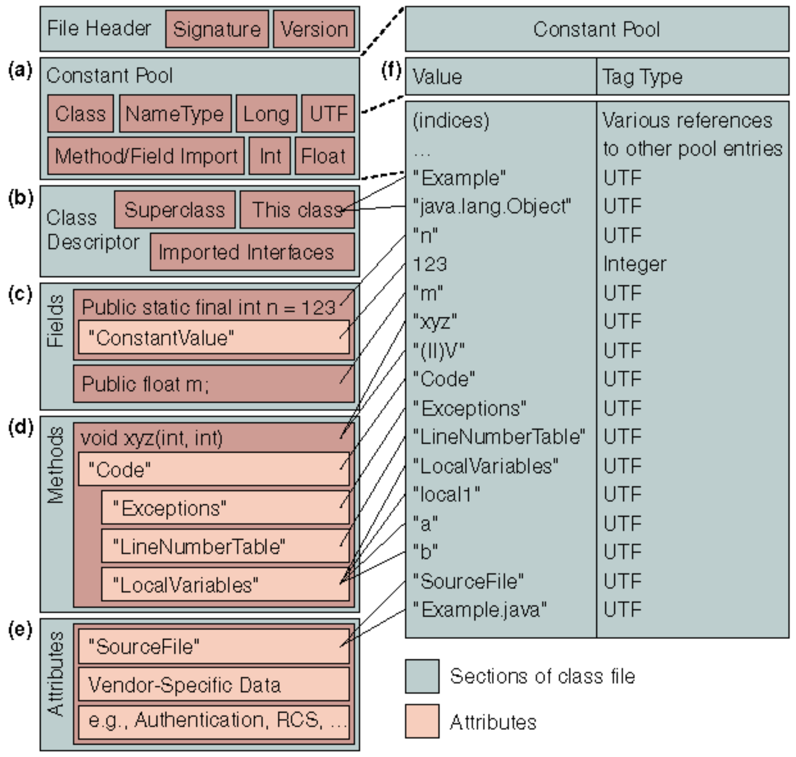
Then it explains the two parsers used on bytecode in this document. Next the role of the compiler is investigated. Is bytecode unique? Does it matter how source code is compiled?

Finally the encountered problems are given, and the questions to be answered in the next chapter are posed.

The class file format

It's not in the scope of this paper to give a complete overview of the design issues of the Java class file format and the byte code instructions. We will just give a brief introduction on the subject, needed to understand the findings in the rest of this document. A more complete and detailed description of the byte code instructions and the class file format can be found in “The Java Virtual Machine Specification Second Edition”[5] [LY99].

FIGURE 1. The classfile layout[a]



a. Figure from Matt Yourst’ article in Dr. Dobb’s Journal [DRD98]

5. Also available online: http://java.sun.com/docs/books/vmspec/index.html

Figure 1 shows a simplified example of the contents of a Java class file: It starts with a header containing a signature or "*magic number*" `0xCAFEBABE`[6] and the *version number*, followed by the *constant pool* (Figure 1 (a)).

There are two version numbers: a major V and a minor v. The numbers are used to determine the version of the class file format as V.v. A Java virtual machine implementation can support the classfile if it lies between Mi.0 <= V.v <= Mj.m. The current 1.2 JVM's accept all versions in [45.0, 46.0]. The earlier 1.1.x JVM's only accepted versions in [45.0, 45.3][7].

In the constant pool, various structures represent String constants, class and interface names, field names and other constants. In the rest of the classfile, references are made to the constant pool. Then a class descriptor follows (Figure 1 (b)), containing the access rights of the class encoded by a bit mask, the `this` class, the `super` class and a list of interfaces implemented by the class. Next lists containing the fields (Figure 1 (c)) and methods of the class (Figure 1 (d)), all referencing the constant pool. And finally the *class attributes* (Figure 1 (e)), e.g. the `SourceFile` attribute telling the name of the source file. Attributes are a way of putting additional, e.g. user-defined, information into class file data structures. For example, a custom class loader may evaluate such attribute data in order to perform its transformations. The JVM specification declares that unknown, i.e. user-defined attributes must be ignored by any Virtual Machine implementation.

The methods contain a description of the signature of the method, and the `Code` attribute (Figure 1 (d)). In the code attribute the bytecode instructions are contained. The sourcecode statements are compiled into bytecode instruction as shown in Table 2 on page 50. The Code attribute is thus an essential part of the class file format. It contains the functionality of a class. The code attribute has three optional attributes: *`Exceptions`*, *`LineNumberTable`*, *`LocalVariables`*. Two of them, LineNumberTable and LocalVariables, are for pure debugging purposes. If exceptions are present in the method, the code attribute has an Exceptions attribute.

6. The first four bytes form the hexadecimal number 0xCAFEBABE, tell the JVM it is receiving a classfile. There are some interesting facts and rumors about CAFEBABE, first the fact: the MAB files on the NeXT platform begin with the same 4 bytes. MAB files are the compiled and executable form of programs on the NeXT platform. Then the rumor: it would be in honor of a waitress in a nearby café.
7. An interesting remark here is that the problems between Microsoft and Sun over Java are not about this level. At the bytecode level the Microsoft and Sun JVM's are still compatible, the problems are about different standard classes.

The LineNumberTable is an optional attribute of the Code attribute, it contains a mapping of the bytecode instructions to the lines in the sourcecode they represent. Typically one line in the sourcecode will be mapped on different bytecode instructions, as we will see in the example explained in Table 2 on page 50. But empty lines or comment blocks in the sourcecode won't be represented at all. So there is no one-to-one mapping between the LineNumberTable entries and the source code.

One use of the LineNumberTable is for debuggers. Lets compare the presence or absence of the LineNumberTable by a little example:

The execution of this class will create an `ArrayIndexOutOfBoundsException`.

```
public class CrashMe{
   static public void main(String[] args){
       int[] array = new int[20];
       for(int i = 0; i < 40; ++i)array[i] = i;
   }
}
```

When a LineNumberTable is present, execution gives:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsEx-
ception
        at CrashMe.main(CrashMe.java:4)
```

When no LineNumberTable is present, execution gives:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsEx-
ception
        at CrashMe.main(Unknown Source)
```

The difference is that in the second case the JVM has no idea of the source file, and can't map the exception back to the source file. Which makes debugging a lot harder. Fortunately the Java compiler includes the LineNumberTable by default.

An other optional attribute of the Code attribute used for debugging purposes is LocalVariables. The local variable table contains the names and types of all variables used in a method body. And information on the scope of the variable. It can be used by debuggers to determine the value of a local variable during execution of a method. There are some remarks to be made. The LocalVariables not only contain the names and types of the local variables, but also of the formal parameters of a method. And in case of an instance method (non-static) it also has an entry "this" for the implicit `this` variable.

Because all of the information needed to dynamically resolve the symbolic references to classes, fields and methods at run-time is coded with string constants, the constant pool contains in fact the largest portion of an average class file, approximately 60% [AP98]. The byte code instructions themselves just make up 12%. The right upper box shows a "zoomed" excerpt of the constant pool (Figure 1 (f)).

TABLE 2. **Source code vs. byte instructions**

public HelloWorld(){	Method HelloWorld()
System.out.println("Hello World!");	0 `aload_0`
}	1 `invokespecial` #11
	4 `getstatic` #15
	7 `ldc` #18
	9 `invokevirtual` #23
	12 `return`
	Constant Pool entries:
	#12 NameType name #13 type #14
	#13 Name "`out`"
	#14 Type "`java.io.PrintStream`"
	#15 Field class #16 type #12
	#16 Class "`java.lang.System`"
	#18 String "`Hello World!`"
	#20 NameType name #21 type #22
	#21 Name "`println`"
	#22 Type "`void (java.lang.String)`"
	#23 Method class #24 type #20
	#24 Class "`java.io.PrintStream`"

Table 2 shows the straightforward translation of the well-known statement: System.out.println("Hello, world");

In the next paragraph we will translate the byte code step by step, to show the role of the constant pool when constructing FAMIX models.

The first instructions `aload_0` and `invokespecial` load this and invoke the Object() constructor because our HelloWorld() method is a constructor.

The next instruction (getstatic #15) loads the contents of the field `out` of class `java.lang.System (#16)` onto the operand stack. This is an instance of the class `java.io.PrintStream (#14)`.

The `ldc` ("Load constant") pushes a reference to the string ”Hello world!” (#18) on the stack.

The next instruction (invokevirtual #23) invokes the instance method `println` (#21) that takes both values on stack as parameters (Instance methods always implicitly take an instance reference as their first argument).

If we look at this last instruction a bit closer we see the way information is stored in the constant pool. #23 in the constant pool contains a Method entry. This entry points to the class containing the method (#24) "`java.io.PrintStream`" and points to the type of the method (#20), there we find the name of the method "`println`" and its type which is an encoded form containing the parameters and the return type of the method "`void (java.lang.String)`". With this information we can form the famix uniqueName of that method:

```
java::io::PrintStream.void println(java::lang::String)
```

Everything has its consequences

As we saw in the previous sub-section, the classfile format is a very structural format, where program entities like Methods or Fields can be found in a straightforward way. This is because Java is designed to be portable over different operation systems. Therefore the class file format should be relatively simple to understand, so it can be interpreted and executed on different operating systems. We can think of classfiles as being analogous to object files in other languages as C++, waiting to be linked and executed. This is very important, all linking, name resolution and loading of other classes is done at run time in the JVM, and not at compile time as in C++. This makes Java class files more portable over networks since dynamic loading is possible.

Of course this makes Java very prone to decompilation, but also very suited for extracting FAMIX information. Since it is possible to map the information back to the source code if there is a LineNumberTable present, the class file is a useful source for creating FAMIX models. Adding information specific for Metrics might be a bit harder, although the control flow is present in the bytecode, and could also be used to calculate metric values. Even for refactoring, the classfiles could be used to generate FAMIX models.

The options in bytecode can have an influence on the usability of the class file as a source for generating FAMIX models.

The compiler options & obfuscators

Now that we know parsing java bytecode is useful and possible, we have to ask ourselves: is bytecode unique? If we have two compiled versions of the same project, will this always produce the same FAMIX models? The answer is not always yes, and not always no. The two major players in this case are the compiler and the use of obfuscators.

First of all some entities in the class file format are optional, the LineNumerTable and the LocalVariableTable are optional attributes for a Method. And the Sourcefile attribute for the class is also optional. The `javac` compiler has the option -g to turn these options on or off, by default javac only generates the LineNumberTable for methods, for debugging purposes.

If the class file was compiled with the line number table and sourcefile options on (`-g:source,lines`) then all entities contained in the method bodies, can be mapped back to the source code. Thus everything but the Attribute, Package, Class and Inheritance entities have a begin and end line and a sourcefile. This information isn't always 100% correct, the begin and end line of a method aren't always correct, since we only have information on the method body. If a method begins or ends with a comment block or whitelines, the given begin and end line in the FAMIX entity will point to the method body, but not at the real begin and end line of the method. But it always is an accurate guess.

If the class file was compiled with the local variable table option on (`-g:vars`) then we have LocalVariable elements in our FAMIX model, and we can give the FormalParameters of our model a correct name. Formal parameters are also stored in the local variable table, if there is no local variable table the name of the formal parameter is not known. And we use a default name (argX).

But all this information is debug information, most release versions are compiled without this debug information (`-g:none`). And optimized, this last option skips unused variables, but has no real effect on our FAMIX model, since local variables aren't available anyway in most cases.

But even without debug information it is still very easy to decompile the bytecode to a very readable sourcecode with exception of the native methods, but those weren't available in the original Java sourcecode either, and make the code less portable, one of the reasons one would choose Java as development language.

For component vendors decompilation is unacceptable because they have to protect their copyrights. So since the first decompiler was written in 1996 by Hanpeter van Vliet[8], Mocha, there was also a tool to prevent a classfile from being decompiled into readable and thus usable source code. This tool, called an obfuscator, intends to alter the classfile in such way that it preserves its functionality, but is more difficult to understand. The obfuscated code will produce the same output as the original code, but might be a bit slower due to added or altered code. This is a trade-off between security and performance.

What does an obfuscator do? There are three main obfuscation strategies that can be used in combination. *Layout obfuscation*, *data obfuscation* and *control obfuscation*. Most obfuscators are freeware, shareware or commercial, rather then academic publications, most of the obfuscators for Java on the market only use layout obfuscation.[DL98]

1. *Layout obfuscation* is altering or deleting information that isn't necessary to the execution of a program. This means all identifiers in the program are scrambled, sometimes they are replaced by random generated identifiers. Another technique used is switch the entries in the constantpool. These tools don't protect against decompilation, but they make the decompiled code unreadable. Next to altering the constant pool, the line number table is also an easy target. The information in the line number table is changed, so that mapping the bytecode back to sourcecode becomes much harder.

   Of course we don't want to decompile the code, but the obfuscator also affects our FAMIX model, since the names of the classes, methods and attributes are

8. Hanpeter van Vliet released the beta version of his Mocha decompiler in 1996, and he also wrote a obfuscator to prevent decompilation: Crema. Unfortunatly he died in December 1996 of cancer, and never completed his decompiler. But since Mocha many more and better decompilers have appeared, Mocha itself was improved and included in Borland JBuilder.

affected by the obfuscator. And the begin and end lines extracted from the bytecode are meaningless. But next to the naming and location information, our FAMIX entities remain unaffected. For modelling a system, or applying metrics to the system, the resulting FAMIX models will suffice.

2. *Data obfuscation* affects the data structures used by a program, e.g. changing the iterating integer in a loop. Other examples are using a hash function to access an array, or converting two dimensional arrays to one dimensional arrays. This kind of obfuscation is not widely used, since it helps very little against decompilation, and makes the program slower.
3. The last technique is *Control Obfuscation*. Some obfuscators (the better ones) use control obfuscation. The idea here is to disguise the real control flow in a program. This can be iterating loops backwards, inlining called procedures. Add irrelevant statements to hide the real control flow, adding redundant termination conditions in loops etc. But the most promising technique is high-level language breaking. This is something especially useful for Java. Since the Java programming language has no `goto` statement. While the Java byte code has a `goto` statement. So Java byte code can contain non-reducible control flow graphs. Inserting a goto in the middle of a while-loop in the byte code, cannot be transformed back into a standard while-loop without difficulty. And all control structures can be re-written using only goto statements, making it impossible to translate the modified bytecode back to valid Java source code.

Until now this last technique is very useful against decompilation[9], but since it only affects the control flow, it has little to no effect on the extraction of FAMIX models. In the next chapter we'll discuss the results supporting this claim.

Bytecode Parsers

The two bytecode parsers we examine in this document are the BCEL parser and the JCF parser. The main difference between them is the way they approach the classfile. BCEL requires a classname as Barat did, and performs a name lookup procedure to find the classfile somewhere in the classpath. JCF demands the path of a certain classfile. It can be a relative or an absolute path to the classfile, but JCF performs no lookup procedure, since it asks the precise location of the classfile from its user.

9. There is a "chicken and the egg" relation between obfuscators and decompilers, so this might change in the future.

BCEL

BCEL stands for Byte Code Engineering Library, a library by Markus Dahm. In this document we use the JavaClass API of the BCEL library. It's the "static" component of the BCEL API, all binary components and structures defined in the class file format are mapped to classes in this package.

Thus the top-level data structure: `JavaClass`, consists of fields, methods, a superclass and a list of implemented interfaces. The actual byte code instructions are available as the `Code` attribute of a `Method`. BCEL provides a method to get the code as a bytestream, but nothing more. So to analyse the bytecode, one has to translate the byte code instructions and their parameters using the constant pool.

BCEL supports the visitor design pattern, so we can write our own visitor to traverse the contents of a class file, and generate our FAMIX models on the way.

For each class parsed by BCEL a `JavaClass` object is created. JavaClass represents a Java class, i.e. the fields and methods contained in a Java .class file. All these data structures can be visited by passing the visitor to each data structure. The Famix `Class` element and the Famix `Package` element can be directly drawn from this object. The name of the source file is contained in the Sourcefile attribute of the classfile if present. Next the different `InheritanceDefinitions` are extracted from JavaClass, in bytecode there always is a superclass defined. Since in Java all classes are an implicit extension of `java.lang.Object`, this is set as superclass in the bytecode when there is no explicit superclass given in the sourcecode.

BCEL is an open source project of Markus Dahm of the Freie Universität Berlin.

It has three parts:

1) a package to reflect the class (the former JavaClass), this is the package used in this document

2) a package to generate and modify JavaClass objects

3) some utilities

More information can be found at the web site:

http://jakarta.apache.org/bcel/

All classes in Java implicitely inherit from `Object`, it is a language specific feature. The only exceptions in Java on this rule are the primitive types: `boolean, byte, char, double, float, int, long and short`, but these types aren't really classes. Therefore adding a InheritanceDefinition for each class inheriting from java.lang.Object doesn't add any information to our model, it only makes our model larger, and thus less scalable.

Next the attributes and methods in the classfile are visited. All Famix elements of extraction levels 1 and 2 can be determined from the JavaClass object itself. The other levels of extraction are a bit less straightforward to extract. Since all other elements are contained in the Code attribute of a Method.

This means parsing the different bytecode instructions to find the representation of Accesses and Invocations in the sourcecode. This isn't too difficult because a field access and a method invocation in the source code, are easy to recognize as we will see. The entities of level 4 are more complex, the LocalVariable famix entity corresponds with an entry in the optional LocalVariableTable. As we have told above, we cannot be sure to get the LocalVariables out of the bytecode. FormalParameters will always be present, since they are part of the method signature, but as stated above, the names of the Parameters are also stored in the LocalVariableTable.

The ImplicitVariable entity in Java comprehends "`this`", "`super`" and "`.class`". The first two are real implicit variables, `this` is an instance variable that refers the current object. `super` refers to the superclass of the current object. In the source code it is optional to place `this` in front of a instance field.

Consider the next example:

```
public void setA(int i){
    a = i;
    this.a = i;

}
```

Assuming setA() is a method in a class with a field `a` of the type `int`, the two statements in this method are identical. In Java there is an implicit "`this`" in front of each instance field access or instance method invocation, putting it there explicitly doesn't change the statement.

```
   0 aload_0
   1 iload_1
   2 putfield #2 <Field int a>
   5 aload_0
   6 iload_1
   7 putfield #2 <Field int a>
  10 return
```

Once compiled it is impossible to determine the difference between those two statements. Since an instance access or invocation also has arguments, storing an implicit `this` as a Famix `ImplicitVariable` does not spoil our model, because an `AccessArgument` can refer to this `ImplicitVariable` element.

The second implicit variable encountered in Java is “`super`”. As with `this`, `super` can also be optional. When we want to call a method of the super class instead of the overwriting method of an object itself, Java allows us this, by using the implicit `super` instance variable. Without this implicit variable we would be forced to cast. In this case, when the called method is also a member method of the class, we use `super` explicit. When the called method is not overwritten and only present in the super class, the `super` implicit variable is presumed implicit.

Consider next example:

```
public void method(){
   int hash1 = hashCode();
   int hash2 = super.hashCode();
}
```

Assume this method is in a class with `java.lang.Object` as super class, that does not overwrite the `hashCode()` method. Thus again the two statements in the example are equal. Both calling the `hashCode()` method of `java.lang.Object`. In the second statement `super` is optional, however when we compile this method we find this:

```
   0 aload_0
   1 invokevirtual #2 <Method int hashCode()>
   4 istore_1
   5 aload_0
```

```
 6 invokespecial #2 <Method int hashCode()>
 9 istore_2
10 return
```

Both statements are compiled into different bytecode instructions. If we look at the bytecode instructions, we see that twice this is loaded (`aload_0`). And twice method #2 in the `constantpool` is called. But for the first statement `invokevirtual` is used, for the second statement `invokespecial`. The difference is the use of the implicit variable super in the source code. As we saw previously, `invokespecial` is used in three cases: private methods, instance initialization methods (constructors) and handling superclass methods. `Invokevirtual` is the general bytecode instruction to handle a invocation. The difference lies is the way the actual method to be invoked is selected in the lookup procedure at run-time. `Invokevirtual` will look for the method in the class of *objectref*, in this case `this`. `Invokespecial` will look for the method in the direct superclass of `this`.

The last implicit variable in the Java plug-in is the `class` implicit variable. As we can read in the plug-in, this is not really an implicit variable in the strict sense of the word. A `String.class` expression evaluates to a reference to the String class object. It's not really an implicit variable, but it is close enough to be modelled as one.

But since it isn't one in the strict sense of the word, this doesn't look that simple in bytecode instructions. The `.class` expression is in fact not as simple as it looks. Table 3 on page 59 gives a simple example using the `.class` expression, and the bytecode instructions needed to execute the statement. As we see, there are two separate methods needed. The `class$` method is a static method generated by the compiler to return the class object of the given class. This is called an `synthetic` method, because it is not represented by itself in the sourcecode. All entities in the Java classfile can have the attribute "`synthetic`" meaning that it was generated by the compiler, an other example is the `<clinit>` method. When a class has static member fields, the `<clinit>` method initializes all static member fields when the class is first loaded. An entity in the classfile marked synthetic should be skipped in our FAMIX model, since they only handle language specific features. They would model information on the compiler rather then on the system itself.

TABLE 3. The bytecode representation of the .class notation

```
public void getClassObject(){
   Class c = String.class;
}
```

```
Method void getClassObject()
   0 getstatic #9 <Field java.lang.Class class$java$lang$String>
   3 ifnonnull 18
   6 ldc #10 <String "java.lang.String">
   8 invokestatic #11 <Method java.lang.Class class$(java.lang.String)>
  11 dup
  12 putstatic #9 <Field java.lang.Class class$java$lang$String>
  15 goto 21
  18 getstatic #9 <Field java.lang.Class class$java$lang$String>
  21 astore_1
  22 return
Method java.lang.Class class$(java.lang.String)
   0 aload_0
   1 invokestatic #1 <Method java.lang.Class forName(java.lang.String)>
   4 areturn
   5 astore_1
   6 new #3 <Class java.lang.NoClassDefFoundError>
   9 dup
  10 aload_1
  11 invokevirtual #4 <Method java.lang.String getMessage()>
  14 invokespecial #5 <Method java.lang.NoClassDefFoundEr-
ror(java.lang.String)>
  17 athrow
```

In BCEL we didn't implement support for Arguments, this would be easy to add, but we chose not to. We implemented support for Arguments in JCF, and since the implementation would be almost the same for BCEL, we skipped it. This gives us the chance of observing the differences between a Level 3 (BCEL) and a Level 4 (JCF) parser.

JCF

JCF stands for Java Class File. In the article Inside Java Class Files in Dr. Dobb's Journal of January 1998, Matt T. Yourst presents two tools he wrote to explore the Java class file format. They both use the JCF class file manipulation library. JCF allows us, just as BCEL did, to access the defined parts of the class file format, through the top-level data structure: `JcfClassFile`. Just as in BCEL we can access the actual byte code instructions through a array of bytes.

JCF was freely available online, in the Dr Dobb's archive. But that archive isn't free anymore.

Matt Yourst wrote the article when he sat in High School. Currently he has a PhD at MIT.

JCF does not come with a visitor pattern. So the entities have to be accessed one by one to generate FAMIX. We implemented JCF to support all elements in the Java plug-in, except the `fromType` attribute of the TypeCast element.

Conclusion

The FAMIX model is based on the program entities, the source code parsers give us an AST. Of course we can extract the information needed to create a FAMIX model from the AST, but the AST gives us much more information, information we don't use and don't need either. The bytecode parsers don't build an AST from the classfile. They offer us a way to access the structures in the class file format, which are basically the program entities. The only information we cannot access directly through the bytecode parsers, are the level 3 and 4 FAMIX entities, with the exception of Formal Parameters, which are included in description of a method. The other FAMIX entities, Access, Invocation, Argument, LocalVariable and ImplicitVariable are included in the Code attribute of a method.

So bytecode parsers have the advantage over sourcecode parsers that they don't waste time in generating a far more detailed model of our software then we need. The drawback is, that they don't give us all information we need in a straightforward way.

The fact that bytecode can serve to produce very detailed models, is very important. For Java projects we will always have at least the bytecode available, and even if the bytecode is obfuscated and stripped we can still extract enough information to calculate e.g. the coupling between the different components or the design of the project.

One remark that has to be made is that the Java compiler is allowed to make some changes to the bytecode, to optimize it. This could be switching statements, which wouldn't affect our model, but the compiler can also drop unused local variables, or even skip pieces in the code that can never be reached at run time (e.g. `if(false){...}`). This would of course change the output of our parser, and would introduce some differences between a model created from bytecode and one created from sourcecode. This doesn't change the value of bytecode as a source for our FAMIX models, but it is important to note it, the compiler can remove dead code in some cases. Since the code is never executed it has no effect on the behaviour of a system, but it wouldn't be possible to detect such pieces of dead code in our model since they aren't modelled at all.

CHAPTER 5

Analysis of parsing techniques

Introduction

In the previous chapter several techniques were given. We can take bytecode and sourcecode as basis to build our model on. As we have seen, this choice can have some implications on the resulting model. We have also seen in Chapter two, what FAMIX is, and how it is structured. The goal of this document is to compare different parsing techniques used to parse Java systems into FAMIX models.

How can we compare different parsing techniques? We need some criteria to differentiate and analyze the techniques. We need to analyze the resulting model, the process to create the model and also more in general we have to briefly analyze FAMIX itself.

First of all we need criteria to test the resulting model with. Of course our model has to be correct. *Correctness* is an absolute priority, we want to create a model of our system, so we expect that our resulting model is indeed a faithful representation of our software system. A model has to be syntactically correct, since we use FAMIX as our repository format, our resulting model should indeed be a consistent FAMIX model. E.g. the uniqueName formulas should be respected.

But even more important, out model must also be semantically correct. Meaning that our FAMIX entities have to be valid. E.g. the containing method of an invocation should be an existing method in our model. And there shouldn't be any ambiguities in our model.

Our model can contain little information and still be correct, so correctness doesn't tell us enough to analyze a parsing technique. FAMIX has different levels of precision, we need to verify which constraints our model satisfies, to determine the level of *precision* it achieves.

TABLE 1. The levels of precision in FAMIX given the Java plug-in

Level	Entities needed to satisfy to this level
Level 1	Package, Class, InheritanceDefinition, (NestingDefinition), Method
Level 2	Level 1 + Attribute
Level 3	Level 2 + Access, Invocation, (Throw)
Level 4	Level 3 + AccessArgument, ExpressionArgument, FormalParameter, LocalVariable, ImplicitVariable

To measure the precision of a parsing technique, a default benchmark is provided, with several files focussing each on an FAMIX entity, and taking the differences between sourcecode and bytecode into account.

Scalability is a more general criterion, more for FAMIX than for specific parsing results. We need a description of the influence of the level of precision on the size of the model. And the influence of the system we model, in LOC or in complexity, on the size of the resulting model.

Scalability can also be seen as a measure for the parsing technique. The memory use of a particular technique is also something we have to take into account. A technique can have a high performance with small systems, but explode on larger projects; or it could be rather slow, but increase its memory use linear with the systems size. In this chapter we'll look for ways to evaluate this criterion.

Scalability brings us to the process of parsing rather then the result of this process. Indeed we cannot only look at the output of a parsing technique, other aspects such as performance, maintainability and usability also have to be taken into account.

Java is a moving target, not a sitting duck. We just all adapted to JDK1.3, now there is JDK1.4, and with that new development kit, an extra element is added to the syntax: `assert`. Not to mention the plans to include generics in the next Java version (adding templates to Java). So the *maintainability* criterion is very important when examining Java parser techniques. It should be easy to adapt and update an existing parser to changing language specifications.

Another straightforward criterion concerning the parser is *performance*. How does the compiler perform, what is the relation between the size of the software system, and the time to parse it? Is there a connection between the complexity and the speed?

A last criterion we should take into account again focuses more on FAMIX in general. We should investigate the usability of our model. In this document we'll look into the possibilities to use the FAMIX repository for a fictitious metric tool.

We conclude with this short list summarizing the criteria we will use.

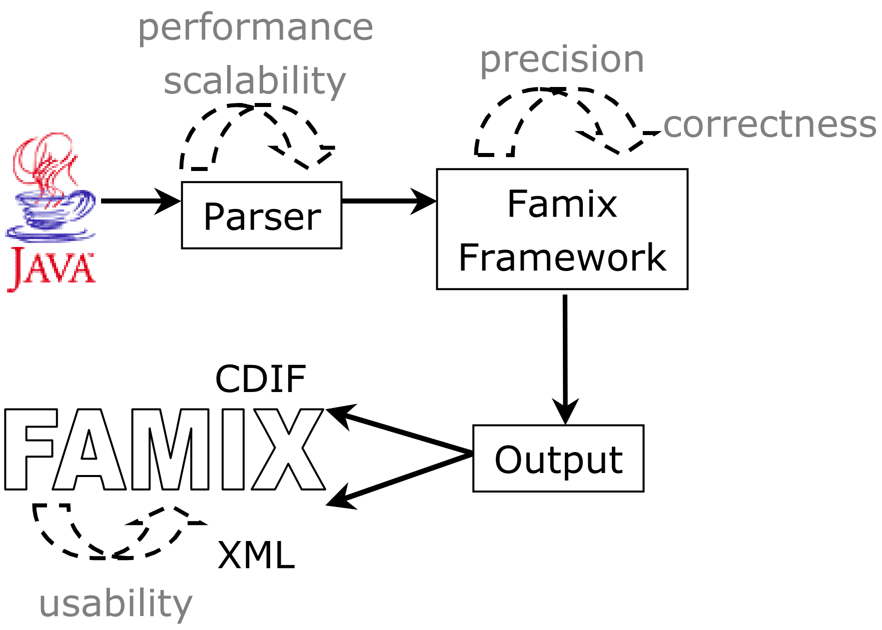
TABLE 2. The criteria used and the way to evaluate them

• Correctness	Syntactic
	Semantic
• Precision & Performance	Benchmark
• Scalability	Memory use
• Maintainability	Parser properties
• Usability	Apply FAMIX on a Metrics tool

The experiment

In the previous subsection, we stated 6 criteria that were of importance. Let's now have a look at the process used to test the parsing techniques. We begin with our Java source (source code or byte code) then we choose a parser accordingly. All parsers implement a common parser interface, so they are interchangeable in the testing framework.

FIGURE 1. A scheme of the experiment



Performance and scalabilities were always done on the parsing process, by timing the parsing process, and investigated further by the JProbe tool[1]. This allowed us to analyse the behaviour of the parser in memory, the time use and the memory use.

All parsers generate objects for the intermediate Famix Framework. In this phase all famix entities are represented by a corresponding class, and are stored in a tree-like structure, making it possible to check the generated famix entities on precision and correctness. It is also possible to compare the resulting Famix model of different parsers.

Since famix is an interchange format, our famix model has to be saved to disk. The famix framework provides an output interface that allows the famix model to be stored in CDIF or XML.

1. Homepage of JProbe : http://www.sitraka.com/software/jprobe

If we run the whole experiment, these three phases correspond to the invocation of the methods `parse()`, `test()`, `compare()`, `getExtractionLevel()`, `print()`. These are the methods of the `parser` interface, all parsers used in the experiment implement this interface.

Each of the criteria in Table 2 on page 64 will be tested

- *Correctness* and *precision* are tested on a benchmark. A benchmark containing 23 classes, each written to focus on a specific element in the Java language or focusing on a specific construction in the Virtual Machine bytecode language.

  For correctness `test` is called, for precision `getExtractionLevel` is called. To be able to compare different parsers, `compare()` is called. It compares the resulting models of two parsers.
- For *Performance* and *scalability*, we also use a test on a benchmark. Several packages, mostly open source and student projects, but also two JDK's from Sun, form a benchmark. All parsers will process these packages, and their behaviour in time and memory use will be recorded and analyzed. We will look for connections between the packages and the results (complexity of the classes, lines of code...). The packages we will use are eclipse, java, javax, junit, scob and sun.

  For performance the parse method is timed, when parsing one of the packages in our benchmark, the parsing time of all classes are summed. For the scalability the JProbe tool is used, that allows us to analyse the time and space usage of the parser.
- *Maintainability* is tested by trying to extend the parser to accept the new assert statement added to the Java syntax since JDK 1.4.
- For *Usability,* we look at its extension abilities for tools and languages. By creating a FAMIX plug-in for a hypothetical metrics tool, we test the possibility of writing a tool plug-in. To investigate the language plug-in capabilities the quality of the Java language plug-in is discussed.

Correctness & Precision

Not all parsers generate output that is “correct”. The ANTLR parser for example, and similar techniques as JavaCC and CUP, can’t generate FAMIX models that are correct. They cannot extract enough information from their datasources, the sourcecode in their case. The reason for this is, as we mentioned in chapter 2, that the resulting AST of these parsers doesn't contain enough semantic information.

For FAMIX we need a complete analysis of the type-information in our system. E.g., when we have a variable of the type `String` and we have a variable of the type `java.lang.String`, they are both of the same type. However, to recognize that, we have to know that `String` is in fact `java.lang.String`, except in some special cases. Anyway, we will have to parse or process all import statements in the Java code, and take the scoping rules into account, to be able to draw such a conclusion and define the `uniqueNames` of the class of a variable. When using bytecode parsers we are not encountered by this problem, because the compiler already did all this work for us. In the constantpool, all classes are given by their `uniqueName`[2].

So we can conclude that ANTLR and most other parser based on JavaCC or something similar, don't hand us enough information on their own to create FAMIX models. They produce a pure syntax tree from the source code. Since it wasn't possible to create correct FAMIX models, it wasn't useful to determine a level of precision.

The other parsers: Barat, JCF and BCEL all produce FAMIX models that are semantically correct. This means that all `uniqueNames` are conform to the given formula's, and that all referrals in the model are valid. To model the precision of the generated FAMIX models, we used a matrix: containing the FAMIX entities as rows, and the different parsers as columns.

TABLE 3. The precision levels attained by each parser

FAMIX	Barat	BCEL	JCF
Package level1	✓	✓	✓
Class level1	✓	✓	✓
InheritanceDefinition level1	✓	✓	✓

2. more information on the class file format to store the bytecode can be found in Table on page 46.

TABLE 3. **The precision levels attained by each parser**

FAMIX		Barat	BCEL	JCF
Method	level1	✓	✓	✓
Attribute	level2	✓	✓	✓
Accesses	level3	✓	✓	✓
Invocation	level3	✓	✓	✓
ImplicitVariable	level4	✓	✗	✓
LocalVariable	level4	✓	☑	☑
FormalParameter	level4	✓	☑	☑
TypeCast	level4	✓	☑	☑
Argument	level4	✓	✗	✓
✓ supported ☑ partially supported ✗ not implemented				

Barat

As we can see the only parser that achieves complete Level 4 precision is the *Barat* parser. This is as we could expect, because Barat does a full semantic analysis of the sourcecode. It goes even deeper then what we need for FAMIX. Yet some remarks have to be made.

First of all Barat can only give us the begin line of an entity. This is quite unexpected, certainly if we see that the bytecode parsers give us a very accurate guess of the begin and end line. We would expect a source code parser to give us very precise information about the begin and end line of the entities of our system. Barat is basically an LL(k) parser building an AST, it does not approach the sourcecode at a program entity level. The end line information is simply not stored.

Furthermore the iterator variables used in loops (`int i = 0;` style) are not modelled as `LocalVariables`. This is a parser implementation issue rather than a parser feature, but it is important to notice, because bytecode parsers can't make this difference. If needed the Barat parser implementation can be easily altered to include these variables as local variables. (by setting a flag)

The `accesses` attribute in the FAMIX Access entity, and the base attribute in the FAMIX Invocation entity contain `Names`. The FAMIX definition states that these are the uniqueNames of the accessed field, or the entity the invoked method is invoked on. Consider next example:

```
...
MyClass mine = new MyClass();
mine.invokeMe();
...
```

With `MyClass` a class implementing an interface called `MyInterface`. The `invokeMe` method is a method of `MyInterface`, implemented by `MyClass`. In this case, the base attribute of the FAMIX Invocation entity modelling `mine.invokeMe()` would be "MyInterface" and not "MyClass" because `invokeMe` is a method of `MyInterface`. One would expect that it would be MyClass because that is the class of the method that is actually invoked. Although this is not in conflict with the FAMIX definition, it is important to note this. Because the `candidates` attribute of the Invocation entity will only contain methods overriding `MyClass.invokeMe()`, not all methods overriding `MyInterface.invokeMe()`.

Analogous, if we are accessing a field declared in an interface, that interface field would be referred, rather than the class implementing the interface.

Another shortcoming of the Barat parser, although this is maybe more a Maintainability issue, is that Barat is based on the Java 1.1 syntax. So it provides very limited support for inner classes and no support for initializers. If these Java features would be added to the FAMIX Java plug-in, as we propose in this document, Barat would fail to support them. Inner classes can be declared, but the naming of InnerClasses is not fully supported. As seen in the previous chapter, Barat analyses all classes it needs, this includes the parsing of all classes and packages stated in the import list of the class.

But Barat fails to translate the import clause in case of an inner class, causing the parsing process to abort, even on a completely valid Java source file.

E.g: `import MyPackage.MyClass.MyInnerClass;`

With MyClass the class containing MyInnerClass. This clause would be accepted by Barat, but the equal clause `import MyPackage.MyClass.*;` would not. Barat presumes MyPackage.MyClass to be a package, and it doesn't recognize that MyClass is a class enclosing inner classes. The consequences and solutions for this will be discussed later when discussing the Maintainability of Barat.

BCEL

The next parsing technique in Table 3 on page 67 is the *BCEL* parser. Let us now discuss some general consequences of using bytecode as source medium for our FAMIX model, before we go into further detail on BCEL.

First we will consider fully verbose bytecode, with line number table and local variable table. The consequences of stripped bytecode will be discussed later. The begin and end lines in the linenumber table are an accurate guess of the real begin and end line in the source code. Since only the statements are compiled into bytecode, the begin line given by the bytecode, will be the first line of the method containing a statement. Comments or code layout may cause the real method (in the source code) to begin some lines before the first statement in its body. E.g.:

```
09 void
10 methodA()
11 {
12 /*
13 some comment telling us this method is in fact empty
14 */
15 }
```

Would result in a Method entity in our FAMIX model telling us methodA begins at line 14 and ends at line 15. Because that's where the implicit return statement is located. If using the Java code conventions[3], and placing comments in Java-

3. http://java.sun.com/docs/codeconv/

doc clauses outside the method body, the begin and endline given by the bytecode parsers will be very accurate, with a possible fault of 1 line.

The line numbers given for invocations and accesses will always be correct. A problem are the Attribute entities. Class fields are compiled as part of the class layout rather then as a part of the class implementation. So the line number of the class field definition is not stored in the bytecode.

A second consequence of using bytecode is the `isFinal` attribute in LocalVariable and FormalParameter. It's fairly simple why bytecode parser can't retrieve this information, it simply isn't compiled into the bytecode. The `final` keyword is used by the compiler to perform a semantic check, so when trying to assign a final variable after initializing it, the compiler will give a compile time error. However, in the bytecode instructions generated by the compiler, this information isn't included anymore.

A third consequence encountered in the set up we use in this paper, is the `fromType` attribute in the TypeCast entity. It isn't impossible to derive this information from the bytecode, since at run time the object casted to another type, will have an original type. Since this object can be the result of a complex expression, determining this information from the bytecode instructions itself is only possible if we also keep track of the stack. The resulting type will be influenced by the return types of the method invocations and the operations on those return types that occur in the expression.

E.g.

```
String s = (String)(b.method(
                         a.someMethod().callingAnother()));
```

To determine the type of the object casted to a String requires information on the return and parameter type of a couple of invoked methods. This information can be extracted from the bytecode but it isn't easy.

The presented bytecode parsers in this document do not have an implementation that allows doing this.

They do however, support the `fromType` attribute for casts between primitives (`int` to `long`, `double` to `float` etc.).

We also notice that `ImplicitVariables` and `Arguments` are not supported in BCEL. As we stated before, the bytecode parsers only give us the functionality to access the program entities, such as method and attributes. For more detailed information such as Accesses and Invocations, we need to access the byte code instructions ourselves. This is the case in both JCF and BCEL. So the implementation to add support for Level 3 and 4 entities to our model, is the same for all bytecode parsers, given the differences the bytecode is presented (stream vs. array). So the fact that a parser using BCEL doesn't support `Argu-`

`ments` and `ImplicitVariables`, is a choice we made, not a lack of functionality in BCEL.

Since the performance of both parsers is almost identical, as we will see in the next section, we chose not to support those entities. To add arguments and implicits to our model, we have to keep track of the status of the stack. Skipping this construction in BCEL allowed us to calculate the difference in cost to parse Level 3 or 4 models.

So far, we assumed fully verbose bytecode. Unforunatley bytecode will most generally not contain a local variable table, implying no LocalVariable entities, and no name information on FormalParameters. It's also possible that there is no LineNumberTable present, implying that there is no information about the begin and end lines of classes, methods and invocations and accesses. Fortunately, by standard this information IS included in the bytecode, only by stripping it, or by explicitly excluding it at compile time, the LineNumberTable won't be present.

It is also possible that bytecode is obfuscated, as we saw in the previous chapter. In that case the names of the entities will be meaningless or not present in most cases. All other information will remain correct, but all naming information is mangled or randomized.

JCF

We can be rather short on *JCF*, because as we stated in the previous subsection, all bytecode parsers are very similar. Their difference lies in the way they present us Level 1 and 2 information. JCF was implemented to be a full Level 4 parser, except the isFinal attribute of `LocalVariables` and `FormalParameters`, since they are not present in the bytecode. The `fromType` attribute is not supported for the same reasons as we saw for BCEL.

The JCF parser also supports Arguments, to model arguments correctly we need to keep track of the stack. Consider next example:

```
public void methodA(){
   int a = 22;
   doSomething(a);
}
```

In bytecode instructions this look like:

`7 iconst_22`	(load constant 22)
`8 astore_1`	(store it in `a`)
`9 aload_1`	(load `a` on stack)

```
10 invokevirtual #22 <Method myClass.doSomething(int)>
11 return
```

As we can see the arguments of the invocation are pushed on stack before it is invoked. By keeping track of the stack we can add exact information on the Arguments to our model. Intructions of the form `aload_x` imply an access of a local variable or a formal parameter. `getfield` and `getstatic` imply an access of an attribute. In those cases we model an AccessArgument. ExpressionArguments can be recognized when we encounter bytecode instructions that denote an operation, like `mult`, `div` or `add`.

Performance & scalability

These two criteria are simple: what we want is a parser that is **fast** and doesn't choke itself on **large projects**.

Measuring time is easy, we note the time our parser starts its job, and when our parsers finishes, we note the time again. The difference of these two timestamps gives us the parsing time. It's important to note that all values given in this chapter always only include the parsing process. The time needed to export the model to CDIF or XMI is never included, since this time is independent from the parser used.

Now we have the time needed to parse a project / package / class, how does this help us to differentiate the parsers? To determine whether a certain parser is fast or slow we will compare the times of the different parsers we examine on a benchmark.

The benchmark we will use consists of different projects. To give a general idea of the size of the projects, the *lines of code* and the *number of classes* metrics will be given for each project.

TABLE 4. The performance and scalability benchmark

Project	LOC	NOC	Description
junit	3.405	52	JUnit is a regression testing framework for Java. http://www.junit.org
org.omg	6.884	247	Some packages from OMG, containing support for CORBA and IDL. http://www.omg.org
scob	23.749	104	A student project of a colleague of mine, Conny Van den Bossche. She wrote a graphical user interface for a stock exchange database.
sun	64.422	1.533	The sun package in the JDK 1.3 distribution from Sun for the windows platform. This package is in constrast to the other packages in the JDK's platform dependent.
java	107.748	1.132	The java package from JDK 1.3.
javax	118.544	1.441	The javax package from JDK 1.3.
org.eclipse	668.037	8.959	The eclipse project from IBM. This is an open-source IDE. http://www.eclipse.org

For scalability, we will use the same benchmark, focusing on the changes in behaviour of the parser for the different projects. We'll look for explanations behind the performance of the parsers, are there relations between certain properties of the projects and the time use behaviour of the parser. For example we expect a clear relation between the LOC and the parsing time.

This will give us a view on the scalability in time. Of course we also have to look into the scalability in space. What is the memory use of the different parsers, and again we'll look into the relations between properties of the project and the memory use behaviour of the parser. To do this we used the JProbe tool.[4]

4. The JProbe website : http://www.sitraka.com/software/jprobe/

Performance

Figure 1 gives us a good idea of the proportions between the parsing times of the different parser, most of all between our sourcecode parser and the bytecode parsers.

All performance tests were done on a Pentium III machine with 320 MB RAM, running Windows 2000 and Java JDK 1.3 installed. The virtual machine was initialized with a maximum heap size of 512 MB. This was to prevent OutOfMemory Exceptions. All tests were conducted at night, so the processor load was minimal.

On Figure 1 in Appendix D we clearly see that the Barat parser is much slower than the bytecode parsers. For the eclipse project, Barat takes about 2 hours and 45 minutes to completely parse the eclipse project, while the bytecode parsers do it in about 3 minutes, that's a difference of factor 60. It's already clear that our Barat doesn't really scales up to larger projects.

We see the dotted line in the graph clearly shows the relation between the lines of code of (a) package(s) and the time Barat needs to parse the package(s). So given the LOC of a project we can approximate the time needed to parse the project.

When look at the results a bit more closely, we see that BCEL and JCF always have values that are very near to each other. This can be easily explained because most work when parsing the classfile is located in parsing the bytecode instructions. And that's something we have to do ourselves, so the implementations to parse the bytecode instructions are almost identical for BCEL and JCF.

When analysing the parsing process of the bytecode parsers in more detail, we see that parsing the bytecode instructions, meaning generating all level 3 and 4 FAMIX elements, takes 60% of the parsing time. So if we would for example only want to extract the class diagram from the eclipse project, this would take about 1 minute. The cost of generating level 4 FAMIX elements only takes about 25% of the total parsing time. This can be explained because generating ImplicitVariables and TypeCasts costs nothing more, since they are extracted from the same bytecode instructions that are already evaluated for the Invocations and Accesses. The LocalVariables are contained in a separate attribute of the classfile, so we only have to translate the entities of this attribute from the ConstantPool to generate a LocalVariable FAMIX element, which isn't a really great cost either. The only real extra cost are the Argument elements (ExpressionArgument and AccessArgument). They require an extra analysis of the elements present on the stack.

Object passed as arguments to Invocations of Accesses are first pushed on the stack. There we have to keep track of them, are they used in an expression? Or are they a reference to an Attribute or a LocalVariable, or some other Entity modelled in our FAMIX model.

When looking closer to the parsing process of the Barat parser, we notice something similar. Barat spends 58.5% of its time parsing expressions, this is only necessary for level 3 and 4 FAMIX elements. So the same reasoning can be made as for bytecode parsers.

An explanation for the gigantic time difference between Barat and the bytecode parsers lies in the semantic analysis. For each type Barat encounters, it does a name lookup. Barat looks up the type in an internal cache, if Barat didn't encounter the type before then the type isn't present and Barat parses the class of the type. This procedure is done for all types occurring in a sourcecode file, before Barat actually starts parsing the sourcecode file itself. Barat needs to do this to determine the correct type of each typed entity in the AST. This namelookup process causes enormous delays (up to ten minutes) for each class. In bytecode this analysis isn't needed, since each type is given by its complete name.

This also makes Barat not very well suited to parse a single class. Since the overhead Barat generates by doing the name lookup procedure for each type it encounters in the code, can cause an initial delay of several minutes. For example when parsing the scob package a first time, the total time is over 3 minutes. We parsing it again directly after the first time, all name lookup procedures can be skipped, since all classes are already in cache, it only takes half a minute. The overhead of doing all the name lookup procedures, takes up to 6 times the actual parsing time.

Let us now go deeper into the performance of the bytecode parsers. Figure 2 (page 134) shows us the graphs of BCEL and JCF. Again a dotted line is given, this time we plotted the number of classes metric (multiplied by 20). For bytecode parsers, the number of classes influences the parsing time more then the lines of code of the project.

We can explain this because in the bytecode all code is compiled into bytecode instructions, which we can access in a byte array, or on a stream. Iterating over an array, or reading from a stream is many times faster than parsing source code into a syntax tree. The complexity of iterating an array or reading from a stream is linear with the number of bytecode instructions. The complexity of parsing an AST from source code is rather quadratic with the lines of code.

For bytecode parser the number of classfiles it has to parse will have a greater influence on the total parsing time, than the size of the method bodies. These assumptions are only true when working with “normal styled” Java code. If we would consider a project written in Java, using a procedural style, so very long methods in some gigantic classes, the number of classes wouldn’t give us a good approximation of the performance of the bytecode parsers anymore.

Normally functionality in Java is divided over different classes, all containing a few methods (classes containing more than 20 methods are very rare) and most methods medium sized (methods larger than 30 lines of code or exceptional). Only in that case the number of classes gives a good approximation of the parsing time.

So for a project written in Java, but using a procedural approach, the number of classes will be small and won’t provide a good approximation of the total parsing time. In that case the lines of code will provide a better view.

To explain the small difference in performance for the eclipse project, we should note that JCF generates level 4 elements, and BCEL does not in our testing configuration. For larger projects this difference begins to notice.

Scalability

The time scalability is already discussed above, the time to parse a project with Barat increases linearly with the lines of code, but since it increases with a factor 14 the parsing time attains longer times rather quickly. A project of 250.000 lines of code will take about an hour to parse with Barat. A project of a million lines of code would take about 4 hours to parse.

Since the parsing time for bytecode parsers scales with the number of classes, rather then the lines of code, it also increases linearly but at a lower pace. If we would extrapolate the graph, a project of a million lines of code would take about 3 to 6 minutes, depending of the number of classes. This is a shrill difference with the 4 hours Barat would need.

So although they both scale up linearly, the bytecode parsers BCEL and JCF a much more scalable since they scale up with the number of classes rather than the lines of code.

Now lets look at the memory use of our parsers, since the memory use also has to be scalable, for a parser to be useful. Again we see a major difference between the bytecode parsers and Barat. We saw in the previous subsection that Barat uses a cache where it stores all the types it has encountered in the parsing process. This implies that when we parse a larger project with Barat the “namelookup” cache grows considerably, stuffing the Java Virtual Machine heap with String and Tokens.

On out testing machine (PIII 320 MB RAM) setting the maximum heap size to 256 MB wasn’t enough to prevent the JVM of throwing an OutOfMemory exception when parsing the java package, which is about 100.000 lines of code. This is because the heap keeps growing with each class that is parsed by Barat, and never decreases in size.

The solution we used to be able to use Barat for larger projects, is shutting the parser down after 10 or 100 classes, and restarting it. This allows us to keep the allocated heap within limits, but adds extra overhead of loading the classes into the VM repeatedly. And Barat isn’t able to benefit from its cache.

In fact the cache mechanism is the reason why Barat isn’t useful for our purposes.

The bytecode parsers are much cleaner in memory use. No extra semantic analysis is needed as in Barat, since all information needed is located in the constant pool of the classfile. We might expect the bytecode parser to have a very low memory usage, but both parsers behave very differently for memory usage.

JCF can parse the whole eclipse package with a maximum stack size of 9 MB. This is because classes are parsed one at a time as we expected. Most classes can be parsed with a heap allocated to a maximum of 4 MB. When parsing a classfile with JCF the class is first loaded into memory as a JcfClassFile object, then our parser processes the JcfClassFile object. When the parsing of an object is done, the whole process starts over for the next class, nothing of the previous run remains on the heap.

BCEL uses a class Repository, we can compare with the cache Barat used. BCEL uses a name lookup procedure given a class name in its Repository, if the class name is not present it is loaded into the Reposity, and a JavaClass object is returned. The JavaClass object is then parsed, when the parsing of an object is done, the parser begins the next run, but the JavaClass object remains in the repository. Since we only parse the classfiles once for generating a model, the Repository doesn’t give us any speedup, it only incrementally grows the heapsize of the parsing process. When parsing the eclipse project, the maximum heap size keeps increasing up to 220 MB.

We can conclude that JCF scales up the best, since it has a constant memory use over time and lines of code. It is very cheap in memory use, BCEL's memory use increases according to the number of classes in the project. With the current large amounts of memory, projects of some million lines of code won't be a problem however.

In the next section we'll go deeper on the way the parsers are constructed themselves, and how they can be maintained.

Maintainability

Maintainability is not a very straightforward criterion to measure. How can we determine whether a given technique is easy to maintain or not? It all depends on different aspects. The FAMIX specification could change forcing us to adapt our parsers, the Java language could change, something that happens very regularly. The parser has to be adapted to the new Java standard then. Another aspect important in maintainability is the parsing technique underlying the parser we built to generate FAMIX models.

So to evaluate this criterion we will use two approaches.

1. Evaluate the parsing technique itself, is it still being maintained, what techniques are used to implement it, how easy is it to add functionality?
2. Suppose we want to add support for the assert statement in the parser. How much effort would it take to achieve this goal? So far all parser were constructed to support Java 1.3. Recently however, Java 1.4 became the new Java standard. Java 1.4 includes an assert statement adding assertions to the Java syntax. It is a feature that can be turned on or off at compile time, and at load time.

Evaluation of the parsing techniques

Since ANLTR and most JavaCC based parsers aren't fit to create FAMIX models from source code, we will skip them in this section. The first parser we will examine is the Barat parser. Barat is also based on a JavaCC parser, but surrounded with a whole system providing the information needed to build a good FAMIX model, i.e. a complete semantic analysis of the code.

- Barat has some very promising features from a maintainability-point-of-view. It provides the visitor pattern [EG95], making it very easy to visit the AST with all its nodes. So if changes occur in the FAMIX standard, it should be very easy to implement them. For specific subtasks required in our model, such as determining all `candidates` attributes for an `Invocation`, or the `isPureAccessor` attribute of a `BehaviouralEntity`; we can construct a specific visitor, making our parser tidier and cleaner and thus easier to maintain.

Also for changes in the Java syntax Barat looks promising. Because the Barat parser comes in 4 packages: `barat`, `barat.collections`, `barat.reflect` and `barat.parser`. The last package contains the implementation that is normally hidden for users of Barat. The interface of the nodes is completely separated from their implementation. Each AST node has a public interface in `barat.reflect`, and a class implementing the interface in `barat.parser`. Implementation details are completely hidden from the user, all types defined in the `barat.reflect` classes, only reference other interfaces in the package.

Therefore, implementation changes to the parser should be possible without affecting the top-level implementation. This is unfortunately not true. First of all the parser itself is generated by JavaCC, and a BNF syntax is given. But given BNF is the Java 1.1 syntax, explaining the poor support for inner classes (the last addition to the Java language before asserts). The parser generator JavaCC used is version 0.7.1, so more recent releases could give compatibility problems. More important however, adding new elements to the syntax would also require new AST nodes, this would imply changes to the other packages as well. Making it incompatible with the original Barat version, and very hard to incorporate bug fixes and improvements to “our” Barat.

This brings us to another weak point for Barat. It is an open source project, but it isn’t supported very actively anymore. It was a project of a Ph.D. student at the university of Berlin, but now there isn’t any active development done anymore with Barat. Making it very unlikely bug-fixes or updates will be released in the future.

Another downside of Barat is that it is written in Pizza rather then Java. Pizza is an alpha-version of what could become Java 1.5 with generic typing. But it isn’t a standard at all, so making changes to the Barat packages, isn’t very straightforward because it is written in a not well known language.

We can conclude that Barat is very nice as-is, but its score on maintainability is very low. Since there is little to none support for the parser, and it is very hard to modify it ourselves.

- The second parser we investigate is the BCEL parser. It also has a visitor pattern, but this provides less possibilities then the visitor used in Barat, because of the use of bytecode instead of sourcecode. The visitor pattern only allows us to visit the program entities, like methods, attributes and classes. Not the real “body”. No visitor is available to access the real code, the method bodies. Although the tokens used in the bytecode instruction make it easier to comprehend the bytecode instructions, altering the entities in our model would require altering and refactoring the existing implementation. Because there is no abstracting layer, adapting the parser to changes in the FAMIX model would be more complicated than for Barat.

Changes to the Java language and/or bytecode would be a far lesser problem than when using Barat. Because bytecode tends to change very little, most new features in the Java language are translated by the compiler to existing bytecode instructions, rather than new bytecode instructions are added. So a bytecode parser would be less influenced by that kind of changes.

But BCEL is also a very actively supported library. It has recently been added to the Apache project. So we can expect that bug fixes and new releases will include support for new elements in the Java programming language.

We can conclude that BCEL scores very good on maintainability. The fact that there is no pattern like the visitor pattern to access the bytecode instructions, is only a minor disadvantage, because it would be very hard to model bytecode instructions in a tree, and to be of any use it should keep track of the stack in some way too. BCEL offers us functions to access the different instructions and their operands. Since we are working with a stream of instructions, rather than the more complex source program, this approach is very flexible.

- The last parser we used in this document is the JCF parser, also a bytecode parser. JCF provides no visitor pattern, but it provides an Object for each entity present in the classfile format, making it very easy to access all different program entities. The code itself can be found the same way one would expect looking at the classfile format: as an attribute of the method entity. The bytecode instructions are not accessible by a stream as in BCEL, but in an array. Making JCF a very flexible framework to access bytecode and that was exactly what JCF was created for. It was created as an educational tool, used as a guide through the classfile format in Matt Yourst’ article.

  In comparison to BCEL, JCF requires much more knowledge of the classfile format from the software engineer. It provides a much more flexible environment, the main difference is the use of an array to represent the bytecode instructions rather than a stream. This requires a bit more knowledge about the operands of the different instructions. But is also gives us more liberty to scan through the instructions.

Changes to the FAMIX model are very easy to incorporate in the JCF parser, when you have a good knowledge of the bytecode instructions and the classfile format. Without this knowledge, adding functionality would be very hard.

Changes to the Java languages have little impact on this parser, just as with BCEL. Again the engineer needs to have knowledge about the structure of the.class file. He has to know how the new feature in the Java programming language is compiled into bytecode instructions. So he can adapt the parser to recognize this construction and model it if necessary.

As long as changes do not influence the class file format in a drastic way, adding a new attribute wouldn't cause any problems, JCF will be a very useful parser. Unfortunately, the JCF parser isn't supported at all. It was just documentation in the context of the article, and hasn't been developed since.

Conclusion is that JCF is a very valuable parser, very light and flexible to use. As long as the bytecode instruction set doesn't change in a drastic way, a software engineer familiar with the classfile format will find JCF very useful. But when major changes would take place in the bytecode instruction set, JCF would loose much of its value. It comes with the sourcecode, but it isn't very well documented, so refactoring it to add functionality would be hard.

Adding support for `assert`

In JDK1.4 a new keyword was added to the Java language. JavaSoft says: *"An assertion is a statement in the Java^{TM} programming language that enables you to test your assumptions about your program. For example, if you write a method that calculates the speed of a particle, you might assert that the calculated speed is less than the speed of light.".*

We will presume we're all familiar with assertions, so we won't go further in on what assertions are, we'll focus on "how are assertions implemented in Java?".

First let's look at the Java programming language. The assertion mechanism in Java is fairly simple and basic. An assert expression will look like this:

```
assert <Boolean expression>;
```

or

```
assert <Boolean expression>:<expression>;
```

The assert statement evaluates the boolean expression, if true nothing happens and the execution of the program continues. If the expression evaluates to false, an `AssertionException` is thrown (extension of `Error`).

The second form is a variation of the first one. The expression given behind the ":" contains the message of the thrown exception. The first case throws an exception with an empty message, the second form can contain some useful information in the message: e.g. the cause of the exception.

Asserts are only supported by the compiler when compiling them explicitly with the JDK 1.4 compiler.

```
javac -source 1.4 testAsserts.java
```

Without the -source 1.4 attribute, the code is compiled as by a 1.3 compiler, and generates an error because a syntax keyword (assert) is used as an identifier.

If compiled correctly, one can turn the assertions on or off at load time. When invoking the JVM with the `-ea` (enable asserts) flag, the asserts are evaluated. Without they are skipped at run time.

```
java -ea testAsserts
```

Let's now look at how asserts are handled by the virtual machine. No new instructions were added, so the compiler has to translate the assert statement into a certain construction that can be skipped at run time, without any noticeable loss in performance.

When compiled with the `-source 1.4` flag, the compiler generates some extra static synthetic methods. For each class a boolean variable `$assertionDisabled` is created, telling a class and all its instances whether the assertions are turned on or off. It uses the `desiredAssertionStatus()` method in `java.lang.Class` to determine its status when it is first loaded into the virtual machine.

Each assert statement is then translated into a sequence of bytecode instructions allowing them to be skipped if the `$assertionDisabled` variable is set.

```
assert a == 2;
```

is translated into

```
getstatic $assertionDisabled
ifne <jump to end of assertion code>
/* translation of a == 2 and then check for true or false, if
true skip the assertion */
if_icmpeq <jump to end of assertion code>
new <Class java.lang.AssertionError>
/* possible message is loaded, and throw the assertion */
invokespecial Method java.lang.AssertionError(...)
...
```

The ifne instruction will skip the assertion code when the assertions aren't turned on at load time.

Now that we know how asserts are dealt with in Java, and which effects they have on source / bytecode, we can discuss the consequences adding them to our parsers will have.

Adding support to the Barat parser is very hard. Many classes have to be updated to fully support the assert expressions in the AST created by Barat. The poor documentation and the use of Pizza make this a very tedious job. There is also very little support to guide you in this process.

For JCF and BCEL we only have to recognize the assert construction, to skip it. As long as we won't add the assert expressions in our FAMIX model, all we have to do is recognize the `getstatic` instruction checking the `$assertionsDisabled` field. The boolean expression in the assert statement can be arbitrary long, since in an expression we can declare an anonymous class. However an assert statement will always have instructions like the bold ones in our example. They determine the layout of the "assertion handler".

Next to this construction, two methods are added to the bytecode, used to turn assertions on or off at load time for all instances of that class. One of them is synthetic so can be easily detected, the other one doesn't have a name. It is a static codeblock executed when the class is loaded into the VM.

Usability

The last criterion we proposed to evaluate was the usability of the FAMIX model, it doesn't really affect the parsers itself, or the output of one parser in specific. This criterion wants to investigate the ways FAMIX can be used. The FAMIX 2.0 specification[5] provides the possibility to extend the standard with plug-ins. First language specific plug-ins, such as the Java plug-in used in this document. Secondly the FAMIX standard can also be extended with tool plug-ins. So far no tool plug-ins have been proposed, in this document we are going to define a metrics plug-in, assuming a CASE Metrics tool that wants to use FAMIX as its information repository.

As a language plug-in

A language plug-in should allow the FAMIX model to support language specific features. From that point of view, we looked into the current Java plug-in. As we saw in chapter 2 there are some Java specific features that aren't modelled in the plug-in.

Without inner classes and exceptions, we wouldn't be able to construct a model of a Java system covering all aspects of that system. A more detailed discussion on this can be found in chapter 2. We can conclude that a more complete plug-in would have a better usability than the current one. A concrete proposition for a more complete Java plug-in can be found in Appendix A.

The FAMIX model should be broad enough to support different tools, and each tool should be flexible enough to cope with different systems, whose implementation style (e.g. patterns) are all heavily influenced by the programmers who wrote them. Therefore our model should be able to model all relevant relations between different methods and classes, these include extra information on the inheritance definitions, more information on the encapsulation of objects (inner classes), the coarse grained coupling of classes (import statements) and more fine grained coupling (invocations, accesses and throws).

Of course our model would become very bloated if we added all this information. Therefore, there also should be a filtering possibility. If for our current purposes e.g. inner classes are of no importance at all, we should be able to exclude them

5. http://www.iam.unibe.ch/~famoos/FAMIX/Famix20/Html/famix20.html

from the model we create. In addition, invocations and accesses to fields and methods in classes of the `java.lang` package do not really contribute any valuable information to our model, since they are a standard part of each Java system. Invocations of methods of String and Integer will occur in any system, and don't tell us much about its structure or behaviour. So most of the time they will only bloat our model, making it harder to see the real information in our model.

Therefore we suggest to include a filtering possibility to give the software engineer using the model more control, so he can decide which FAMIX entities should be included in the model, and so that he can formulate constraints on the attributes of these entities. This would make the model much more usable in different circumstances. Since FAMIX supports an incremental model, extra information can be added when needed.

As a tool plug-in

FAMIX is intended to be a language independent exchange format between different CASE tools. So far no real tool plug-ins have been defined, mostly information generated by tools is added as a property to the FAMIX entity they belong to. Of course this is very effective when used with one tool, and it allows us to add extra information to the model with a minimum overhead.

When using FAMIX as it is intended: an information repository for different tools, all adding their own extra information, the model would become a mess of FAMIX entities cluttered with properties. The risk of name collisions between the properties would increase, and the confusion would increase.

A neater approach would be to create a tool specific plug-in. In this subsection we'll look into the possibility of creating a plug-in for a Metrics tool. Since probably some new software metrics are invented while you're reading this, providing a good structure to model metrics would be very handy. Without that, our tool would only be able to use some standard metrics to add to the model. If we would allow our tool to support an arbitrary amount of software metrics, all stored by properties in our model, our model would become chaotic, and interchanging the model between different metrics tools would be difficult.

However since most software metrics can be categorized, we should consider a more organized approach.[FAMOOS99]

Most software metrics can be classified in these categories:

1. Complexity metrics

   They measure the complexity of an entity of a system. By complexity we mean the effort a software engineer has to spend to understand, write or modify a piece of software. If code is difficult to write or understand it is considered complex. Metrics are used to estimate the complexity, since it can't be measured directly.
2. Class-Hierarchy metrics or Inheritance metrics

   Inheritance is a central concept in object-oriented systems. Inheritance allows to reuse functionality of other classes, and it can model relationships between classes. These metrics will measure properties of the inheritance tree.
3. Coupling metrics

   Coupling between classes is measured by the way they depend on each other. I.e. does class A invoke methods of class B or accesses attributes of B? Coupling between classes should be rather low.
4. Cohesion

   The cohesion of a class expresses how close attributes and methods of a class are related. It expresses a level of encapsulation of an object. Cohesion of a class should be rather high.

It is logical to use this classification as base for our metrics plug-in.

Metric
description(): `String; optional`
scope(): `Qualifier; optional`

Metric is an abstract class inheriting from Property. Besides inherited attributes it has following attributes:

- `description: String; optional`

  A short description of the metric can be included, explaining what it calculates, and what it tries to express, because not all metrics are standard, and because the metric name is mostly a rather cryptic abbreviation. This attribute is purely optional.
- `scope: Qualifier; optional`

  If applicable the scope of a metric can be set to "*class*", "*method*" or if not set "*inapplicable*". It expresses whether the metric is calculated on the whole class or just on a single method.

According to the classification we saw, there are four entities inheriting from Metric:

- ComplexityMetric
- ClassHierarchyMetric
- CouplingMetric
- CohesionMetric

with no further attributes. The complete Metrics plug-in with examples can be found in Appendix C.

Conclusion

The sourcecode parsers presented in this document are all Java based parsers. They all fail to be used as a FAMIX parser. Barat is very successful but lacks support and further development, making it no reliable parser and it is not scalable for large projects because of its excessive memory use. All JavaCC based parsers, fail to parse correct FAMIX models, since they don't perform a semantic analysis on the code.

Bytecode parsers are the most promising. First of all when compiled verbosely, they contain almost all information needed to create Level 4 models. Next to that the bytecode instructions are less subject to changes in the Java language specification. They also have a better performance and scalability, and the BCEL parser is heavily supported and further developed.

For the FAMIX model to be fully functional as a interchange model between different CASE tools for Java systems, some extensions have to be made, as reported in chapter 3.

CHAPTER 6 Future work

In this document we looked at different ways to parse Java systems into a FAMIX model. We looked at some parsers, saw the differences between bytecode and sourcecode to build a model from, and we looked at FAMIX, investigated whether FAMIX succeeded in modelling a Java system.

Future work on FAMIX

FAMIX is a very valuable modelling language, this document showed that it is easy to extend the model, but to further achieve the goals of FAMIX, some work has to be done.

Interchange format

FAMIX is intended to be an exchange format between different tools, some tools exist that use FAMIX. CodeCrawler[1] a tool developed at the University of Berne, uses FAMIX. But most CASE tools on the market today never heard of FAMIX,

1. http://iamwww.unibe.ch/~lanza/CodeCrawler/codecrawler.html

they use other formats. For FAMIX to be an exchange format, good conversion tools should be developed, to be able to convert FAMIX models to other formats, and to import other formats into a FAMIX model. FAMIX is incremental by nature so it would be very suited to store information created by different tools.

Most importantly, mapping between UML and FAMIX should be possible, information in FAMIX can be mapped to Class diagrams and Collaboration diagrams and vice versa. This would help in making FAMIX a real interchange format between different tools, able to store information needed for re-engineering tools.

Usage of the format

The FAMIX model is a repository for different reengineering CASE tools. Different tools can add information to the model and the information in the model can be visualized by different tools, and metrics can be applied to the model. The real goal is to detect flaws and errors in the design of the system, that may hamper future evolutions of the software or that may influence the performances of the application or component. So that we can improve the system by refactoring it.

Algorithms and metrics to detect these flaws and errors given the model would greatly facilitate the software engineer given the task of reengineering a system. Visualizations are a powerful tool, but for large industrial software systems, they are unusable, the visualization becomes cluttered. And metrics are scalable, but the results of software metrics are harder to comprehend, a visualization is more intuitive. An algorithm or a metric combining the information in different visualizations and metrics to really detect a problem in the system, and propose a solution is something that should be further investigated.

Future work on creating a model

Parsing Java

The main conclusion of this paper is that for parsing Java to a FAMIX model, bytecode is the most suitable source. Of course there are some minor downsides when using bytecode. Therefore it might be necessary to use sourcecode parsers when the isFinal information on local variables and formal parameters, or the fromType in typecasts is essential or wanted in our model.

In this document only parsers written in Java were considered, but many other ways to parse Java source code exist.

- All parsers based on lex and yacc or some similar techniques may prove very efficient for parsing Java source code, since they are LR(k) parsers, rather than the LL(k) parsers examined in this document, making them much more powerful.
- Parsers for Java source code also exist in Smalltalk.
- Compilers construct a detailed AST when transforming the source code into bytecode, some compilers, such as IBM Jikes[2] and GNU gcj[3], allow the user to extract that AST. This AST could be used to create a FAMIX model.

Further work should be done to analyse the properties of these approaches.

The run-time approach

FAMIX is used as repository for the information we extract of the system. All techniques covered in this document model the system in a static way. The source we use to build our model is a static representation of the system, by using sourcecode or compiled code.

The behaviour of our system at runtime might be completely different. Using some monitoring mechanism to extract information from a running system, could give an interesting perspective on the system. It would also allow us to model the systems and component we use, but we don’t have the sources available. What we can model we can measure and analyse, so this could give us the opportunity to tell something about structure and the quality of compiled components and applications, independent from the language the are written in.

2. Jikes homepage: http://oss.software.ibm.com/developerworks/opensource/jikes/
3. The GCJ homepage: http://gcc.gnu.org/java/

APPENDIX A

The FAMIX Java plug-in

This appendix describes the FAMIX metamodel Java plug-in version 2.1. It starts with an overview of the metamodel before describing the metamodel in detail. Next it gives a clear description of all elements in the Java plug-in. Attributes of the FAMIX model that are interpreted for the Java plug-in will be marked by "(interpreted)". Attributes of elements new for the Java plug-in, as inlcuded in the FAMIX 2.1 standard will be marked by "(extended)". Elements and attributes added to the Java plug-in and suggested in this document will be marked by "(new Java)". The elements and attributes added to the FAMIX model itself will be marked by "(new Famix)".

Overview

Figure 1 on page 96 shows an overview of the core FAMIX metamodel. The colored entities are the elements proposed in this document. See "The concrete elements in the model" on page 103. for a detailed description of all the shown elements. In this section, we introduce some information that is necessary for the rest of the metamodel definition, such as some basic data types, unique naming conventions and extraction levels.

Basic Data Types

Besides the usual primitive data types (String, Integer, Boolean, …) we have a number of extra data types in our metamodel that are considered ‘basic’. These are `Name`, `Qualifier` and `Index`:

- Name vs. Qualifier

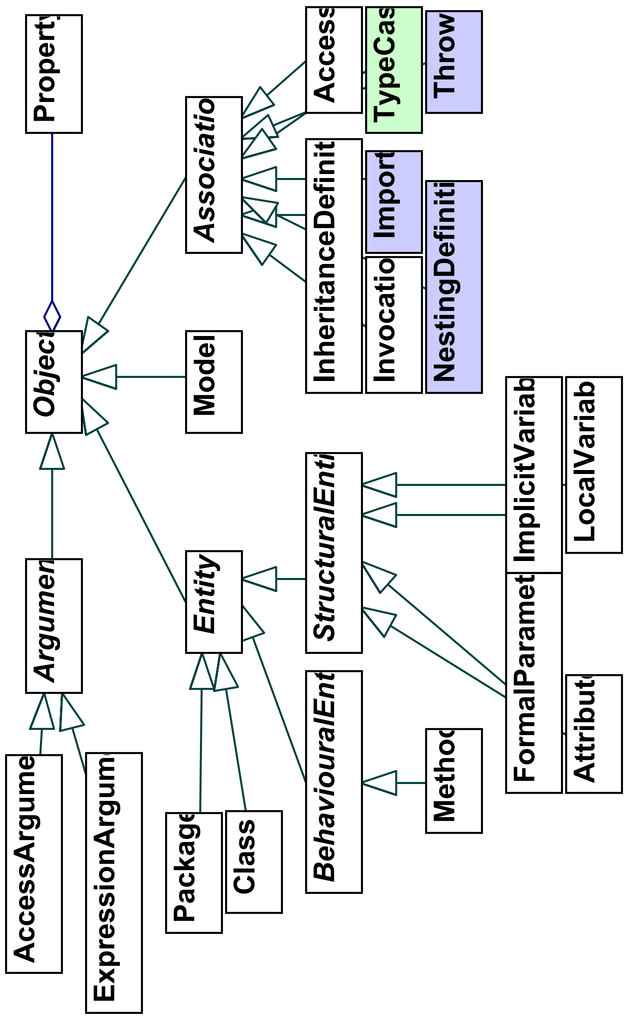
  A `Name` is a string that bears semantics inside the metamodel, while a `Qualifier` is a string that gets its semantics from outside the metamodel. A `String` does not bear any semantics.

  For instance, a `uniqueName` may be used to refer to another object, hence bears semantics inside the metamodel and is a Name. However, a `sourceAnchor` will store some information that must be interpreted by applications outside the metamodel, hence is a qualifier. Finally, a comment line is a string, since it does not bear any semantics understandable by a computer. In CDIF these types are simply represented by Strings, or TextValues if they are multivalued.

- Index

  An Index represents a position in some sequence. Indices always have a base of 1. In CDIF this type is represented by an integer.

FIGURE 1. The complete FAMIX model



Unique Naming Conventions

The naming conventions used in the FAMIX metamodel is as much as possible compliant with UML [OMG99]. This means that the following rules apply:

1. **Scoping via packages.** Global entities, such as classes, functions, global variables and packages themselves receive a unique name by concatenating with the containing package name using "::" as a separator. They will typically look like "`package::subpackage::classname`".
2. **Naming of variables.** Variables, such as attributes, local variables, etc. receive a unique name by concatenating with the containing entity using a "." as a separator. They will typically look like "`package::subpackage::classname.attributename`", "`package::subpackage::classname.method().localvariablename`".
3. **Naming of methods and functions.** Methods and functions distinguish themselves from variables because they have a parameter list. Therefore, they are named by concatenating their scope and their signature. For functions we follow the convention of package scoping, thus separate the scope and the signature via a "::". For methods we follow the convention of variable naming, thus separate the scope and the signature via a ".". The signature of a method and a function contains the name of the method or function, followed by its parameter list surrounded by parentheses. The return type is not part of the signature. They will typically look like "`package::subpackage::functionname(para1,para2)`", "`package::subpackage::classname.methodname(para1,para2)`".

   To achieve a normal form for signatures, parameter lists should not contain unnecessary spaces. Thus

   "`functionname(para1,para2)`"

   instead of

   "`functionname(para1, para2)`".

   However, sometimes languages include keywords in their parameter list, and then spaces cannot be avoided. For instance, the C++ const parameters will be represented like

   "`functionname(const para1,const para2)`".

Level of Extraction

The core metamodel contains entities that not all parsers may provide. Next to that, some tools do not always need all of this information (e.g., a metrics tool might not need Invocation and Access, because many metrics can already be gathered from Class and Method alone). To allow focused models, we introduce the *level of extraction*.

The level of extraction is an integer, representing how much of the core metamodel is available in a model. The higher the number, the more information is available. The levels are set up in such a way that no information is available on a level that needs information from higher levels (for instance, Access is not usable if there are no Attribute's available). Next to that, it is possible that on the higher levels parts of the information are not necessary for a certain task, or simply not computable by a certain tool. Therefore it is allowed to only provide parts of the information (designated by the "+/-"). Table B.1 gives an overview of the levels of extraction.

TABLE 1. The levels of Extraction

Level 1	• Class • InheritanceDefinition • BehaviouralEntity • Package • NestingsDefinition • Import
Level 2	• Attribute
Level 3	• Access • Invocation • Throw
Level 4	• ImplicitVariable • LocalVariable • FormalParameter

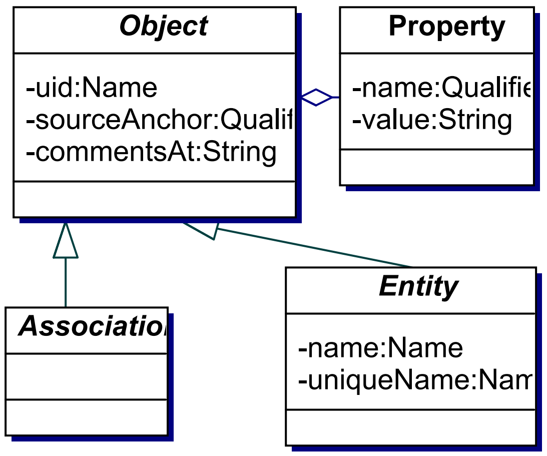
Definition of FAMIX

This part describes the various classes that together specify the FAMIX metamodel. The following subsections describe the different elements with their attributes, and give examples in the CDIF transfer format.

Mandatory attributes must always be present. Optional attributes may be omitted. Some optional attributes have a default value.

The abstract part: Object, Entity and Association

FIGURE 2. The basic classes Object, Entity and Association



The classes Object, Association, Entity and Property capture the commonalities in the design of FAMIX. Furthermore, they provide the hooks to extend the core meta-model with new elements and, through the Property element, with the ability to annotate any Object in a model. These are the different classes in detail:

Object

Object
uid() : `Name; mandatory`
sourceAnchor() : `Qualifier; optional`
commentsAt(pos Integer) : `String; optional`

Object is an abstract class without a superclass. The attributes of Object are:

- uid: Name; mandatory

  Denotes an identifier that is unique for every element in a model. FAMIX does not impose a scheme or format. It is recommended to use Universal Unique

Identifiers (UUIDs) [OG97], which is a standard way of constructing identifiers that are unique over space and time. The resources to store and compute UUIDs, however, might clash with scalability needs (see also section 3.10).

- sourceAnchor: Qualifier; optional

  Identifies the location in the source where the information is extracted. The exact format of the qualifier is dependent on the source of the information. Usually, it will be an anchor in a source file, in which case the following format should be used

  ```
  file "<filespec>" start <start_index> end <end_index>
  ```

  where `<filespec>` is a string holding the name of the source-file in an operating system dependent format (preferably a filename relative to some project directory). Note that filenames may contain spaces and double quotation marks. A double quotation mark in a filename should be escaped with a `\"`. `<start_index>` and `<end_index>` are indices starting at 1 and holding the beginning respectively ending character position in the source file. Extra position indices or whole source anchors may be added to handle anchors in files that may need to be displayed with external editors. For instance, the line and column of the character (`startline`, `startcol`, `endline`, `endcol`). Or the negative offset counting from the end of the file instead of from the beginning (`negstart`, `negend`). In CDIF a basic source anchor looks as follows (delimited with a '|', see section 2.3.1 for a description of multi-valued strings in CDIF):

  ```
  (sourceAnchor #[file "factory.h" start 260 end 653|]#)
  ```

- comments: 0..N String; optional

  Entities and associations may own a number of comments, where developers and tools store textual information about the object. In CDIF we represent this with a CDIF TextValue, where the blocks are delimited by a '|' (see section 2.3.1 for a description of multi-valued strings in CDIF):

  ```
  (comments #[commentLines|]#,#[commentLines|]#,...)
  ```

Property

Property
name() : `Qualifier; mandatory`
value() : `String; mandatory`
belongsToUid() : `Name; mandatory`

Entities and associations may own a number of properties where extensions of the core metamodel may be stored. Property has the following attributes:

- `name: Qualifier; mandatory`

  Is a string that identifies a Property within an Object. Thus, the name should be unique for all properties of a single Object.
- `value: String; mandatory`

  Contains the value of the property. The meaning of the value is not defined within this metamodel.
- `belongsToUid: Name; mandatory`

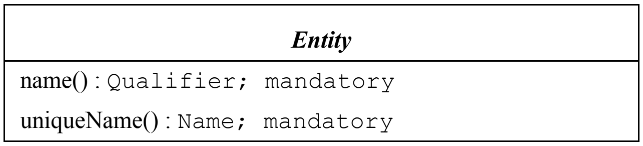
  Contains the uid that identifies the Object this Property is a property of.

CDIF example showing a class `Widget` with a `Property` containing the value 5 for the number-of-methods metric. They are related by the `belongsToUid` attribute.

```
(Class ENT001
   (name "Widget")
   (uid "c842bf06-d202-0000-0282-5c410d0000")
   ....
)

(Property PR005
   (name "metric_NOM")
   (value #[5]#)
   (belongsToUid "c842bf06-d202-0000-0282-5c410d00000")
)
```

Entity



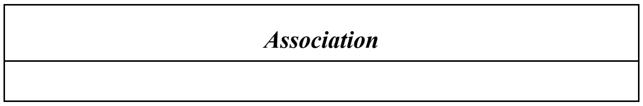
To enable a global referencing scheme based on names, the key classes in the meta-model should respect the minimal interface of Entity. Entity is an abstract class inheriting from Object. Besides inherited attributes, it has the following attributes:

- `name: Qualifier; mandatory`

  Is a string that provides some human readable reference to an entity.
- `uniqueName: Name; mandatory`

  Is a string that is computed based on the name of the entity. Each class of entities must define its specific formula. The `uniqueName` serves as an external reference to that entity and must be unique for all entities in a model.

Association



Association is an empty common superclass for all associations in the metamodel. Assocation is an abstract class inheriting from Object. It defines no new attributes itself.

The concrete elements in the model

Model

Model	
exporterName() : `String; mandatory`	
exporterVersion() : `String; mandatory`	
exporterDate() : `String; mandatory`	
exporterTime() : `String; mandatory`	
publisherName() : `String; mandatory`	
parsedSystemName() : `String; optional`	
extractionLevel() : `String; mandatory`	
sourceLanguage() : `String; mandatory`	(interpreted)
sourceDialect() : `String; optional`	(interpreted)

A Model represents information about the particular system being modelled. Extractors must ensure that there is only one instance of a Model in a model. Information exchange standards often provide means to exchange similar information in special sections of the exchange stream. However, having an explicit Model element in FAMIX allows us to transfer this information independent of the chosen exchange format.

Model is a concrete class inheriting from Object. Besides inherited attributes, it has the following attributes:

- `exporterName: String; mandatory`

  Represents the name of the tool that generated the information.
- `exporterVersion: String; mandatory`

  Represents the version of the tool that generated the information.
- `exporterDate: String; mandatory`

  Represents the date the information was generated.
- `exporterTime: String; mandatory`

  Represents the time of the day the information was generated.
- `publisherName: String; mandatory`

Represents the name of the person that generated the information. Provide an empty string if this information is not known.

- `parsedSystemName: String; optional`

  Represents the name of the system where the information was extracted from.

- `extractionLevel: String; mandatory`

  Represents the level of extraction used when generating the information (see Table on page 139).

- `sourceLanguage: String; mandatory`

  For Java models, this attribute always contains the string “Java”.

- `sourceDialect: String; optional`

  The Java language does not have dialects, but it has versions. If known, the 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 methods parameters and local variables
1.1.x => 1.2.x:	- Addition of the `strictfp` keyword.
1.3.x => 1.4.x:	- Addition of the `assert` keyword.

CDIF example of a Model instance for the Eclipse project.

```
(Model FM0
   (exporterName "barat2famix")
   (exporterVersion "1.0")
   (exporterDate "2002/05/10")
   (exporterTime "10:25:02")
   (publisherName "Hans Stenten")
   (parsedSystemName "Eclipse")
   (extractionLevel "4")
   (sourceLanguage "Java")
   (sourceDialect "1.3.x")
)
```

Package

Package
belongstToPackage() : `Name; optional`

A Package maps 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

These properties are in sync with the expected properties in the core FAMIX definition.

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 encapsulation package.

Package is a concrete class inheriting from Entity. Besides inherited attributes, it has the following attributes:

- `belongsToPackage: Name; optional`

  Is the unique name of the package containing this package. A null value represents the fact that there is no containing package.

Formula for uniqueName (See also “Unique Naming Conventions” on page 96):

```
if isNull (belongsToPackage(package)) then
   uniqueName(package) = name(package)
else
   uniqueName(package) = belongsToPackage(package) + “::” +
                         name (package)
```

CDIF example of a package parser:

```
(Package FM1
   (name "parser")
   (uniqueName "hs::parser")
   (belongsToPackage "hs"))
```

Class

Class
isInterface() : `Boolean; optional` (extended)
isPublic() : `Boolean; optional` (extended)
isFinal() : `Boolean; optional` (extended)
isAbstract() : `Boolean; optional`
isNested(): `Boolean; optional` (new Java)
belongstToPackage() : `Name; optional` (interpreted)

Both classes and interfaces in Java are mapped to the FAMIX entity Class. Interfaces differ from classes in the fact that they can only define abstract methods and final static variables. Interfaces cannot inherit from classes (See also "Inheritance-Definition" on page 108). Since version 1.1.x Java supports inner classes, so we need to store in a way the fact that a class is nested or not.

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: Boolean; optional`

  In Java a class is abstract if the class is declared abstract. This is obligatory 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).
- `isNested: Boolean; optional`

  A predicate telling if the class is nested. I.e. is the class a member class of another enclosing class? The class can be static or not. Nested classes have different scoping rules, they can access all attributes of their enclosing classes.

- `belongsToPackage: Name; optional`

  This attribute refers to the `uniqueName` of the package to which the class belongs, defined by the package statement at the beginning of a Java source file defining the class. In case of a nested class, this attribute refers to the `unique-Name` of the enclosing class. The enclosing classes are considered “packages”. Because there is no difference in notation between an enclosing package and a enclosing class in Java either, a nested class cannot have the name of a package at the same depth. So given a class ClassA.ClassB there are two possibilities: ClassA can be a package containing a class ClassB

  ClassB can be a nested class in ClassA

  It’s never possible to have both, since this would introduce ambiguity in Java.

Formula for `uniqueName`(See also “Unique Naming Conventions” on page 96) :

```
if isNull (belongsToPackage (class)) then
   uniqueName (class) = name (class)
else
   uniqueName (class) = belongsToPackage (class) + "::" +
                        name (class)
```

CDIF example of a non-abstract class `Parser` in package hs.`parser` (note the difference between `name` and `uniqueName`):

```
(Class FM1
   (name "Parser")
   (uniqueName "hs::parser::Parser")
   (isAbstract -FALSE-)
   (sourceAnchor #[file "parser.java" start 0 end 60|]#)
)
```

InheritanceDefinition

InheritanceDefinition	
subclass() : `Name; mandatory`	
superclass() : `Name; mandatory`	
accessControlQualifier() : `Qualifier; optional`	(interpreted)
index() : `Index; optional`	(interpreted)
isImplementation() : `Boolean; optional`	(new Java)

In Java classes always inherit from a single class (except the root class Object that does not 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.

InheritanceDefinition is a concrete class inheriting from Association. Besides inherited attributes, it has following attributes:

- `subclass: Name; mandatory`

  Is a unique name referring to the class that inherits. It uses the uniqueName of the class as a reference.
- `superclass: Name; mandatory`

  Is a unique name referring to the class that is inherited from. It uses the unique-Name of the class as a reference.
- `accessControlQualifier: Qualifier; optional`

  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: Index; optional`

  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.

- `isImplementation(): Boolean; optional`

  Is a predicate telling whether the inheritancedefinition is an implementation or an extension. I.e. is the superclass an interface, or a class. The Java interface the class implements isn't always present in our model, in that case this attribute stores the information that it was an interface and not a class.

CDIF example the `Integer` class in the `java.lang` package, extending `Number` and implementing `Comparable`.

```
(InheritanceDefinition FM20
   (subclass "java::lang::Integer")
   (superclass "java::lang::Number")
   (accessControlQualifier "public")
   (index 1)
   (isImplementation -false-)
)
(InheritanceDefinition FM21
   (subclass "java::lang::Integer")
   (superclass "java::lang::Comparable")
   (accessControlQualifier "public")
   (index 1)
   (isImplementation -true-)

)
```

NestingDefinition

NestingDefinition (new Java)
innerclass(): `Name; mandatory`
enclosingclass(): `Name; mandatory`
accessControlQualifier(): `Qualifier; optional`
isLocal(): `Boolean; optional`
isStatic(): `Boolean; optional`

This element is a new element to the Java plug-in. It is a level 1 precision element. Java has two types of classes. Top-level classes and nested classes, top-level classes are normal classes, standing alone or contained in a package. Nested classes are contained inside a class. Different sorts of nested classes exist, member classes, but also local classes and anonymous classes. The main difference is the way they can be referenced.

Inner classes can have multiple enclosing classes, the relation between the inner class and its enclosing class is represented by the NestingDefinition element in FAMIX. Only the relation with the first enclosing class is given, since if there are multiple enclosing classes, then the enclosing class is itself an inner class, which will have another NestingDefinition element in our model.

NestingDefinition is a concrete class inheriting from Association. Besides inherited attributes, it has following attributes:

- `innerclass: Name; mandatory`

  A unique name referring to the class that is nested. It uses the uniqueName of that class as reference.
- `enclosingclass: Name; mandatory`

  A unique name referring to the enclosing class. It uses the uniqueName of that class as a reference.
- `accessControlQualifier: Qualifier; optional`

  A string determining the access control for the nested class, just as it would for a method of the class.
- `isLocal: Boolean; optional`

  A predicate telling if the nested class is a local class, thus only visible inside a certain scope block.

- `isStatic: Boolean; optional`

  A predicate telling if the nested class is a static member class of the enclosing class or an instance member class.

CDIF example of a class OuterClass containing an inner class InnerClass:

```
(NestingDefinition FM15
   (innerclass "examples::OuterClass::InnerClass")
   (enclosingclass "examples::OuterClass")
   (accessControlQualifier "public")
   (isStatic -false-)
)
```

Import

Import
importingClass(): `Name; mandatory`
importedEntity(): `Name; mandatory`
isPackage(): `Boolean; optional`

Import is an Association, it models the classes or packages a class imports. It gives a rough overview of the dependencies between different classes in the system, when refactoring, this information can be used to locate the heavily used classes.

- `importingClass: Name; mandatory`

  Refers to the uniqueName of the class importing another class
- `importedEntity: Name; mandatory`

  Refers to the uniqueName of the package or class imported
- `isPackage: Boolean; optional`

  A predicate telling whether the importedEntity is a package or a class

Example:

```
package examples;
import java.io.File;
public class ImportExample{
}
```

CDIF example:

```
(Import FM4
   (importingClass "examples::ImportExample")
   (importedEntity "java::io::File")
   (isPackage -false-)
)
```

BehaviouralEntity

BehaviouralEntity
accessControlQualifier(): `Qualifier; optional`
signature(): `Qualifier; mandatory`
isPureAccessor(): `Boolean; optional`
declaredReturnType(): `Qualifier; optional`
declaredReturnClass(): `Name; optional`

A BehaviouralEntity represents the definition in source code of a behavioural abstraction, i.e., an abstraction that denotes an action rather than a part of the state. Subclasses of this class represent different mechanisms for defining such an entity.

The special conversion rules between the declaredReturnType and the declared-ReturnClass, the class represents a FAMIX element, the type represents the source code notation.

TABLE 2. Conversion from declared(Return)Type to declared(Return)Class

SourceCode	**declared(Return)Type**	**declared(Return)Class**
Point	Point	Point
Point[]	Point[]	Point
int	int	

BehaviouralEntity is an abstract class inheriting from Entity. Besides inherited attributes, it has the following attributes:

- `accessControlQualifier: Qualifier; optional`

  A specifier that defines who is allowed to invoke the BehaviouralEntity. In Java this can be ‘public’, ‘protected’, ‘private’ or empty. The last case represents package scope, meaning the method is visible for all classes in the same package.

- `signature: Qualifier; mandatory`

  In Java 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 T1..n are the types of the formal parameters of the method (See also “Unique Naming Conventions” on page 96). 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 [GJSB00].

- `isPureAccessor: Boolean; optional`

  Is a predicate telling whether the behavioural entity is a pure accessor. There are two kinds of accessors, a reader accessor and a writer accessor. A pure reader accessor is an entity with a single receiver parameter, only returning the value of an attribute of the class the method is defined on. A pure writer accessor is a method with one receiver parameter and one value parameter, only storing the value inside the attribute of a class.

  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;
  }
  ```

- `declaredReturnType: Qualifier; optional`

  In Java this attribute can contain any primitive types, array types or classes (and interfaces). For conversion rules between declaredReturnType and declareReturnClass as seen in Table 2 on page 112.

- `declaredReturnClass: Name; optional`

  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. For conversion rules between declaredReturnType and declareReturnClass as seen in Table 2 on page 112.

Method

Method	
belongsToClass(): `Name; mandatory`	
hasClassScope(): `Boolean; optional`	
isAbstract(): `Boolean; optional`	
isConstructor(): `Boolean; optional`	
isFinal(): `Boolean; optional`	(extended)
isSynchronized(): `Boolean; optional`	(extended)
isNative(): `Boolean; optional`	(extended)

A Method represents the definition in source code of an aspect of the behaviour of a class. Method is a concrete class inheriting from BehaviouralEntity. Besides inherited attributes, it has the following attributes.

- `belongsToClass: Name; mandatory`

  Is a name referring to the class owning the method. It uses the `uniqueName` of the class as a reference.
- `hasClassScope: Boolean; optional`

  A method in Java has class scope if it is defined static.
- `isAbstract: Boolean; optional`

  A method is abstract, if it is declared abstract with the `abstract` keyword. An abstract method in Java doesn't have an implementation.
- `isConstructor: Boolean; optional`

  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.
- `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.

Formula for `uniqueName` (See also “Unique Naming Conventions” on page 96):

```
uniqueName (method) = belongsToClass (method) + "." +
                      signature (method)
```

CDIF example (constructor for a class `Widget`. This method has no return type and therefore also no ‘return class’, hence both attributes are empty):

```
(Method FM2
   (name "Widget")
   (belongsToClass "gui::Widget")
   (sourceAnchor #[file "factory.java" start 123 end 143|]#)
   (accessControlQualifier "public")
   (hasClassScope -FALSE-)
   (signature "Widget()")
   (isAbstract -FALSE-)
   (isConstructor -TRUE-)
   (declaredReturnType "")
   (declaredReturnClass "")
   (uniqueName "gui::Widget.Widget()")
)
```

StructuralEntity

StructuralEntity	
declaredType(): `Qualifier; optional`	(interpreted)
declaredClass(): `Name; optional`	(interpreted)

A StructuralEntity represents the definition in source code of a structural entity, i.e., it denotes an aspect of the state of a system. The different kinds of structural entities mainly differ in lifetime: some have the same lifetime as the entity they belong to, e.g., an attribute and a class, some have a lifetime that is the same as the whole system, e.g., a global variable. Subclasses of this class represent different mechanisms for defining such an entity.

For the conventions of converting a declaredType to a declaredClass see Table 2 on page 112.

StructuralEntity is an abstract class inheriting from Entity. Besides inherited attributes, it has the following attributes:

- `declaredType: Qualifier; optional`

  In Java this attribute can contain any primitive type, array type or class and interface.
- `declaredClass: Name; optional`

  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.

Attribute

Attribute	
belongsToClass(): `Name; mandatory`	
hasClassScope(): `Boolean; optional`	
accessControlQualifier(): `Qualifier; optional`	
isFinal(): `Boolean; optional`	(extended)
isTransient(): `Boolean; optional`	(extended)
isVolatile(): `Boolean; optional`	(extended)

An Attribute represents the definition in source code of an aspect of the state of a class. Attribute is a concrete class inheriting from StructuralEntity. Besides inherited attributes, it has the following attributes:

- `belongsToClass: Name; mandatory`

  Is a name referring to the class owning the attribute. It uses the uniqueName of the class as a reference.
- `hasClassScope: Boolean; optional`

  An attribute in Java has class scope if it is defined static.
- `accessControlQualifier; Qualifier; optional`

The allowed access specifiers are: public, protected and private. An empty specifier means default visibility, which denotes that the attribute is visible for all classes within the same package.

- `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 optimize it.

Formula for `uniqueName` (See also "Unique Naming Conventions" on page 96):

```
 uniqueName (attribute) = belongsToClass (attribute) +
                                      "." +   name (attribute)
```

Example of an attribute:

```
public class testClass{
   private int counter;
}
```

CDIF example of a private attribute `wTop` in class `Widget`:

```
(Attribute FM22
   (name "counter")
   (belongsToClass "testClass")
   (sourceAnchor #[file "testClass.java" start 2 end 2|]#)
   (declaredType "int")
   (declaredClass "")
   (accessControlQualifier "private")
   (uniqueName "testClass.counter")
)
```

ImplicitVariable

ImplicitVariable (interpreted)
belongsToContext(): `Qualifier; mandatory`

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. Implicit variables will only appear in a transfer when they are explicitly referenced by other entities.

When adding inner classes to our model, these three variables do not cover all implicit variables that we can encounter in Java. We need a notation to refer the `this` variables of all enclosing classes. As we saw in Chapter 3 a solution is to denote the local variable of the first enclosing class with `this$0`. For innerclasses with multiple enclosing classes, a `this$i` variable will be provided, with `i+1` denoting the the height of the enclosing class.

ImplicitVariable is a concrete class inheriting from StructuralEntity. Besides inherited attributes, it has the following attributes:

- `belongsToContext: Qualifier; optional`

  Is a string with a language-dependent interpretation, that defines a possible scope of the variable. A null `belongsToContext` is allowed, it means that the variable has global scope. The `belongsToContext` concatenated with the `name` of the variable must provide a unique name for that variable within a model.

Formula for `uniqueName` (See also “Unique Naming Conventions” on page 96):

```
if isNull (belongsToContext (implicitVariable)) then
   uniqueName (implicitVariable) = name (implicitVariable)
else
   uniqueName (implicitVariable) =
                    belongsToContext (implicitVariable) +"."+
                    name (implicitVariable)
```

Example of an implicit variable `super`:

```
public class testClass{
   public void testMethod(){
      int hash = super.hashCode();
   }
}
```

In CDIF:

```
(ImplictVariable FM77
   (name "super")
   (declaredType "java::lang::Object")
   (declaredClass "java::lang::Object")
   (belongsToContext "Test::void testMethod()")
   (uniqueName "Test::void testMethod().super")
)
```

LocalVariable

LocalVariable (interpreted)
belongsToBehaviour(): `Name; mandatory`
isFinal(): `Boolean; optional` (extended)

A LocalVariable represents the definition in source code of a variable defined locally to a behavioural entity.

LocalVariable is a concrete class inheriting from StructuralEntity. Besides inherited attributes, it has the following attributes:

- `belongsToBehaviour: Name; mandatory`

  Is a name referring to the BehaviouralEntity owning the variable. It uses the `uniqueName` of this entity as a reference.
- `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.

Formula for `uniqueName` (See also “Unique Naming Conventions” on page 96):

```
uniqueName (localVar) = belongsToBehaviour (localVar) + "." +
                        name (localVar)
```

Example of a local variable `hash`:

```
public class testClass{
   public void testMethod(){
      int hash = super.hashCode();
   }
}
```

In CDIF:

```
(LocalVariable FM76
   (name "hash")
   (sourceAnchor #[file "testClass.java" start 85 end 85|]#)
   (declaredType "int")
   (declaredClass "")
   (belongsToBehaviour "testClass.void testMethod()")
   (uniqueName "testClass.void testMethod().hash")
)
```

FormalParameter

FormalParamter (interpreted)
belongsToBehaviour(): `Name; mandatory`
position(): `Index; mandatory`
isFinal(): `Boolean; optional` (extended)

A FormalParameter represents the definition in source code of a formal parameter, i.e., the declaration of what a behavioural entity expects as an argument.

FormalParameter is a concrete class inheriting from StructuralEntity. Besides inherited attributes, it has the following attributes:

- `belongsToBehaviour: Name; mandatory`

  Is a name referring to the BehaviouralEntity owning the variable. It uses the `uniqueName` of this entity as a reference.
- `position: Index; mandatory`

  Indicates the position of the parameter in the list of parameters. Language extensions should specify what the position of a parameter is and this should be consistent with the `position` attribute of Argument (See also “Argument” on page 128).

- `isFinal: Boolean; optional`

  Is a predicate telling if the attribute is defined final or not. Final parameters cannot be changed within the body 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.

Formula for `uniqueName` (See also "Unique Naming Conventions" on page 96):

```
uniqueName (formalPar) = belongsToBehaviour (formalPar)
                         + "." + name (formalPar)
```

Example:

```
public class testClass{
   public void testMethod(String s){
   ...
   }
}
```

In CDIF:

```
(FormalParameter FM41
   (name "s")
   (declaredType "java::lang::String")
   (declaredClass "java::lang::String")
   (belongsToBehaviour
            "testClass.testMethod(java::lang::String)")
   (position 1)
   (uniqueName "testClass.testMethod(java::lang::String).s")
)
```

Access

Access (interpreted)
accesses(): `Name; mandatory`
accessedIn(): `Name; mandatory`
isAccessLValue(): `Boolean; optional`
hasArguments(pos Integer): `Name; optional`

An Access represents the definition in source code of a BehaviouralEntity accessing a StructuralEntity. Depending on the level of extraction (See also "Level of Extraction" on page 97) , that StructuralEntity may be an attribute, a local variable, an argument, a global variable….What exactly constitutes such a definition is a language-dependent issue. However, when the same structural entity is accessed more than once in a method body, then parsers should generate a separate access-association for each occurrence.

Access is a concrete class inheriting from Association. Besides inherited attributes, it has the following attributes:

- `accesses: Name; mandatory`

  Is a unique name referring to the variable being accessed. It uses the `uniqueName` of the variable as a reference.
- `accessedIn: Name; mandatory`

  Is a unique name referring to the method doing the access. It uses the `uniqueName` of the method as a reference.
- `isAccessLValue: Boolean; optional`

  Is a predicate telling whether the value was accessed as Lvalue, i.e., a location value or a value on the left side of an assignment. When the predicate is true, the memory location denoted by the variable might change its value; false means that the contents of the memory location is read; null means that it is unknown. Note that LValue is the inverse of RValue.
- `hasArguments: 0 .. N Name; optional`

  An Access can have arguments. Typically this will be one, namely the receiving, argument. For instance, in the case of `x.a`, `x` is the receiving argument of the access of `a`. The `hasArguments` attribute denotes the uid of the argument.

Example:

```
public class testClass{
   private int counter;
   public void testMethod(){
      counter = Integer.parseInt("2");
   }
}
```

In CDIF:

```
(Access FM18
(accesses "testClass.counter")
(accessedIn "testClass.void testMethod()")
(isAccessLValue -TRUE-)

)
```

Invocation

Invocation (interpreted)
invokes(): `Name; mandatory`
invokedBy(): `Name; mandatory`
base(): `Name; optional`
candidatesAt(pos Integer): `Name; optional`
hasArguments(pos Integer): `Name; optional`

An Invocation represents the definition in source code of a BehaviouralEntity invoking another BehaviouralEntity.

When the same behavioural entity is invoked more than once in a method body, then parsers should generate a separate invocation-association for each occurrence. It is important to note that due to polymorphism, there exists at parse time a one-to-many relationship between the invocation and the actual entity invoked: a method, for instance, might be defined on a certain class, but at runtime actually invoked on an instance of a subclass of this class. This explains the presence of the `base` attribute and the `candidates` aggregation.

Invocation is a concrete class inheriting from Association. Besides inherited attributes, it has the following attributes:

- `invokedBy: Name; mandatory`

  Is a unique name referring to the BehaviouralEntity doing the invocation. It uses the `uniqueName` of the entity as a reference.

- `invokes: Qualifier; mandatory`

  Is a qualifier holding the signature of the BehaviouralEntity invoked. Due to polymorphism, the signature of the invoked BehaviouralEntity is not enough to assess which BehaviouralEntity is actually invoked. Further analysis based on the arguments is necessary. Concatenated with the `base` attribute this attribute constitutes the unique name of a behavioural entity.

- `base: Name; optional`

In Java this attribute contains the statically determinable class of the expression receiving the invocation.

For example:

```
MyClass r = new MyClass();
r.m();
```

Then r is the receiver and therefore MyClass the receiving class. If the receiving class also contains a method with the invoked signature, it is the base. Otherwise the class defining the inherited method with that signature is the base.

- `candidates: 0 .. N Name; optional`

  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 the class hierarchies that implement that interface.

- hasArguments: 0 .. N Name; optional

  An Invocation has arguments. The `hasArguments` attribute denotes the uids of the arguments.

Example:

```
public class testClass{
   private int counter;
   public void testMethod(){
      counter = Integer.parseInt("2");
   }
}
```

CDIF example:

```
(Invocation FM35
   (invokes "parseInt(java::lang::String)")
   (invokedBy "testClass.void testMethod()")
   (base "java::lang::Integer")
)
```

Throw

Throw (new Famix)
thrownBy() : `Name; mandatory`
throws() : `Qualifier; mandatory`
base(): `Name; mandatory`
hasArguments() : `Name; optional`
isDeclared() : `Boolean; optional`

A throw represents the definition in the source code of a BehaviouralEntity throwing an exception. This exception can be any class inheriting from exception in C++, or any class implementing throwable in Java. For each throw a separate throw-association should be generated.

Throw is a concrete class inheriting from Association. Besides inherited attributes, it has the following attributes:

- `thrownBy: Name; mandatory`

  Is a unique name referring to the BehaviouralEntity doing the throw. It uses the `uniqueName` of the entity as a reference.
- `throws: Qualifier; mandatory`

  Is a qualifier holding the `signature` of the constructor of the class thrown.
- `base: Name; mandatory`

  Is the unique name of the entity where the thrown entity is defined on. Null means unknown. Together with the `thrown` attribute, this attribute constitutes the unique name of a behavioural entity.
- `hasArguments: 0 .. N Name; optional`

  A throw can have arguments. The `hasArguments` attribute denotes the uid's of the arguments.
- `isDeclared: Boolean; optional`

  An exception can be declared for a BehaviouralEntity, telling the invoking BehaviouralEntity that this exception wasn't handled and it should be handled by the invoking BehaviouralEntity. In Java this means a thrown exception is declared if it appears in the throws list of a method. Otherwise it isn't declared.

Example:

```
public class testClass{
   public void testMethod(){
      throw new NullPointerException();
   }
}
```

CDIF example:

```
(Throw FM36
   (throws "NullPointerException()")
   (thrownBy "testClass.void testMethod()")
   (base "java::lang::NullPointerException")
)
```

TypeCast

TypeCast (extended)
belongsToBehaviour(): `Name; mandatory`
fromType(): `Name; optional`
toType(): `Name; optional`

TypeCast is a subclass of Association. It models Java type cast (e.g., `(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 occurring 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 besides inherited ones:

- `belongsToBehaviour: Name; mandatory`

  Refers to the BehaviouralEntity the cast appears in.
- `fromType: Name; optional`

  Refers to the unique name of the type the cast 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 cast to (MyClass in the above example).

Example:

```
public class testClass{
   public void testMethod(){
       Object o;
       ...
       String s = (String)o;
   }
}
```

CDIF example:

```
(TypeCast FM37
   (belongsToBehaviour "testClass.void testMethod()")
   (fromType "java::lang::Object")
   (toType "java::lang::String")
)
```

Argument

Argument
position(): `Index; mandatory`
isReceiver(): `Boolean; mandatory`

An Argument represents the passing of an argument when invoking a Behavioural-Entity or accessing a StructuralEntity.
Argument is an abstract class inheriting from Object. Besides inherited attributes, Argument has the following attributes:

- `position: Index; mandatory`

  The position of the argument in the list of arguments. Language extensions should specify what the position of a argument is and this should be consistent with the `position` attribute of FormalParameter (See also "FormalParameter" on page 121).

- `isReceiver: Boolean; mandatory`

  Is a predicate telling whether this argument plays the role of the receiver in the containing invocation. Knowing which argument plays the role of the receiver may help resolving polymorphic invocations.

ExpressionArgument

ExpressionArgument

An ExpressionArgument models an argument that is a complex expression. This expression is not modelled in further detail (at least in the context of arguments; any access that is part of the expression should be modelled anyway). Expression-Argument is a concrete subclass of Argument. It does not define any new attributes itself.

AccessArgument

AccessArgument
hassAccess(): `name; mandatory`

An AccessArgument models an argument that is a reference to a StructuralEntity.

AccessArgument is a concrete subclass of Argument. Besides inherited attributes, it has the following attributes:

- `hasAccess: Name; mandatory`

  Denotes the unique identifier (uid) of the Access instance that models the access by the argument of a StructuralEntity.

Example:

```
public class testClass{
   private String s;
   public void testMethod(){
      char a = s.charAt(2);
   }
}
```

CDIF example:

```
(Access FM38
   (uid "c842bf06-d202-0000-0282-5c410d0001")
   (accessedIn "testClass.void testMethod()")
   (accesses "testClass.s")
)
(AccessArgument FM39
   (position 1)
   (isReceiver -true-)
   (hasAccess "c842bf06-d202-0000-0282-5c410d0001")
)
```

APPENDIX B Performance graphs

FIGURE 1. **The performance of the different parsers**

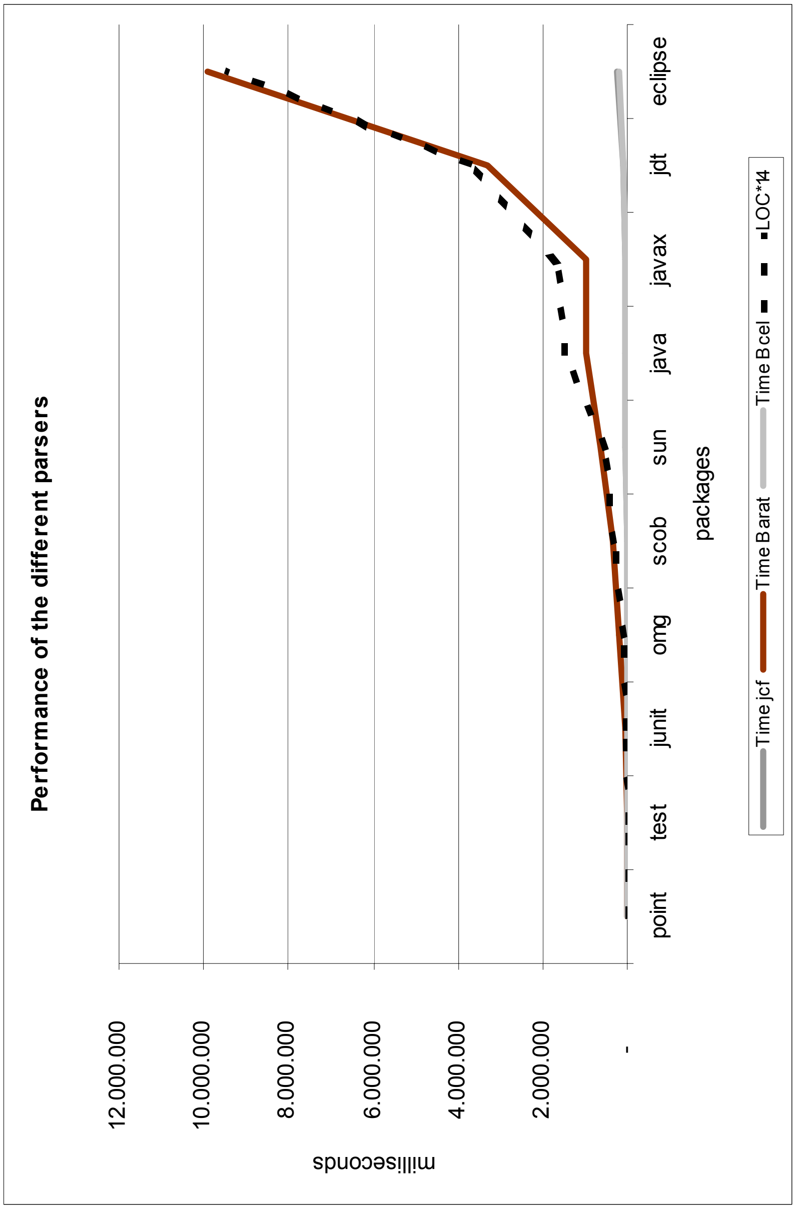
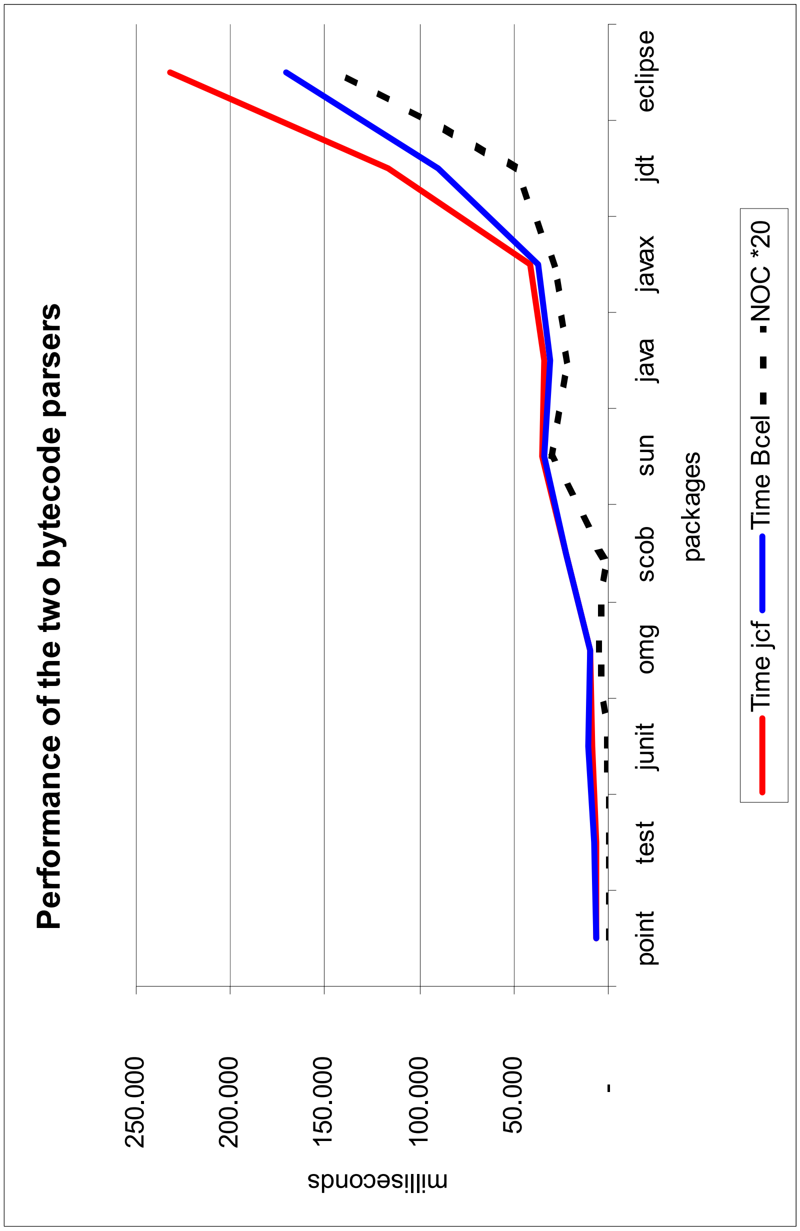


FIGURE 2. The performance of the two bytecode parsers



APPENDIX C

The FAMIX metrics plug-in

Software metrics

Most software metrics can be classified in these categories:

1. Complexity metrics

   They measure the complexity of an entity of a system. By complexity we mean the effort a software engineer has to spend to understand, write or modify a piece of software. If code is difficult to write or understand it is considered complex. Metrics are used to estimate the complexity, since it can't be measured directly.

2. Class-Hierarchy metrics or Inheritance metrics

   Inheritance is a central concept in object-oriented systems. Inheritance allows to reuse functionality of other classes, and it can model relationships between classes. These metrics will measure properties of the inheritance tree.

3. Coupling metrics

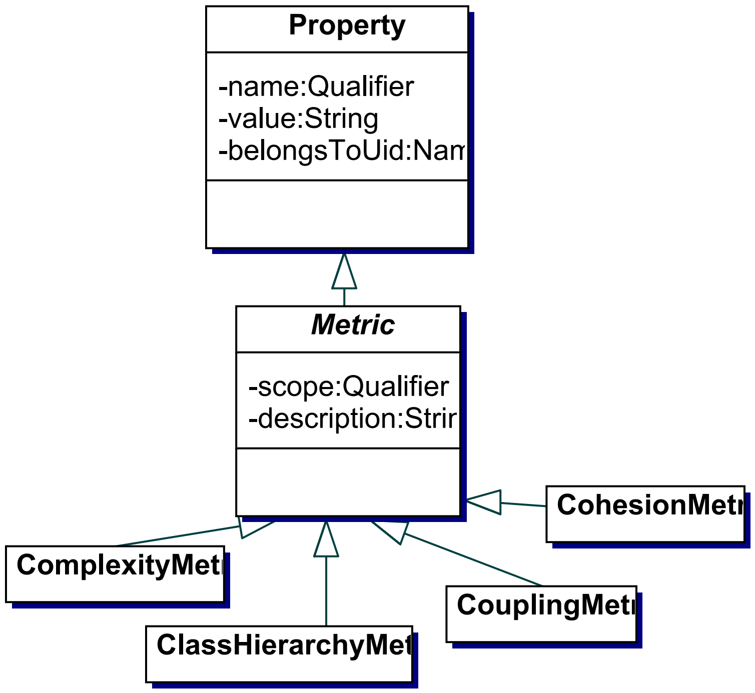
   Coupling between classes is measured by the way they depend on each other. I.e. does class A invoke methods of class B or accesses attributes of B? Coupling between classes should be rather low.

4. Cohesion

The cohesion of a class expresses how close attributes and methods of a class are related. It expresses a level of encapsulation of an object. Cohesion of a class should be rather high.

A Metrics tool plug-in

FIGURE 1. The elements of the Metrics plug-in



It is logical to use this classification as base for our metrics plug-in.

Metric
description(): `String; optional`
scope(): `Qualifier; optional`

Metric is an abstract class inheriting from Property. Besides inherited attributes it has following attributes:

- `description: String; optional`

  A short description of the metric can be included, explaining what it calculates, and what it tries to express, because not all metrics are standard, and because the metric name is mostly a rather cryptic abbreviation. This attribute is purely optional.
- `scope: Qualifier; optional`

  If applicable the scope of a metric can be set to "*class*", "*method*" or if not set "*inapplicable*". It expresses whether the metric is calculated on the whole class or just on a single method.

ComplexityMetric, Class-HierarchyMetric, CouplingMetric and CohesionMetric are concrete subclasses without any further attributes.

CDIF example:

```
(Method FM38
   (uid "c842bf06-d202-0000-0282-5c410d0001")
...
)
(ComplexityMetric FM39
   (name "LOC")
   (value #[50]#)
   (description #[Lines of code]#)
   (scope "method")
   (belongsToUid "c842bf06-d202-0000-0282-5c410d0001")

)
```

Bibliography

[AP98]	D. Antonioli and M. Pilz. Statistische Analyse von Java-Classfiles. In Clemens Cap, editor, Proceedings JIT'98. Springer, 1998.
[BS98]	B. Bokowski and A. Spiegel. Barat - A Front-End for Java. Technical report, Freie Universität Berlin, 1998.
[CCZ97]	Suzanne Collin, Dominique Colnet and Olivier Zendra. Type Inference for Late Binding. The SmallEiffel Compiler. In Proceedings JMLC '97, 1997.
[DL98]	Java Control Flow Obfuscation. Douglas Low. Master Science Thesis. University of Auckland 1998. Available online: http://www.cs.arizona.edu/~collberg/Research/Students/DouglasLow/thesis.ps
[DDN02]	Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz. Object-Oriented Reengineering Patterns. Morgan Kaufman, 2002
[DRD98]	Inside Java Classes. Matt Yourst. Dr. Dobb's Journal January 1998.

[EG95]	Design Patterns; by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; Addison-Wesley Pub Co; ISBN: 0201633612; 1st edition (January 15, 1995)
[FAMOOS99]	The FAMOOS Object-Oriented Reengineering Handbook; Stéphane Ducasse, Serge Demeyer et al.; 1999; http://www.iam/unibe.ch/~famoos/handbook/
[FY00]	Brian Foote and Joseph W.Yoder. Big ball of mud. In N. Harrison, B.Foote, and H.Rohnert, editors, Pattern Languages of Program Design, volume 4, pages 654-692 Addison - Wesley, 2000.
[LY99]	Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification Second Edition. Addison Wesley 1999.
[NUT99]	Java In a Nutshell 3rd Edition; David Flanagan; O'Reilly 1999
[OMG99]	The Unified Modeling Language User Guide; Addison-Wesley Pub Co; ISBN: 0201571684; 1st edition (October 30, 1998)
[ST01]	Modeling Object-Oriented Software for Reverse Engineering and Refactoring. Sander Tichelaar. Philosophisch-naurwissenschaftlichen Fakultät der Universität Bern 2001
[Taf96]	Tucker Taft. Programming the Internet in Ada95. In Proceedings Ada-Europe International Conference on Reliable Software Technologies, 1996.
[TJT00]	The Java Tutorial; Addison-Wesley Pub Co; ISBN: 0201703939; 3rd edition (January 15, 2000)