Recycling Trees: Mapping Eclipse ASTs to Moose Models

Daniel Langone
Software Composition Group,
University of Berne, Switzerland
http://scg.unibe.ch

April 6, 2009

Abstract

Reverse engineering tools are a great help in the process of adapting an existing software system to novel contexts. Current implementations use *models of software systems* to keep themselves language independent. This also implies that the models have to be built before a software system can be analyzed by such a tool. A common approach is to build *language specific parsers* to extract the information from the source-code. But: Manually building parsers is tedious work. This calls for a new approach.

In our approach we rely on the fact that there are already pre-parsed ASTs of software systems. Such parsing applications can be found in applications which host software systems of several languages. These ASTs are needed by the host application to provide auto-completion or syntax highlighting for example. In order to minimize the required knowledge about a certain language when extracting models we *recycle* those *ASTs*.

We hook into a specific *host application*, *Eclipse*, and the ASTs generated by such *parsing applications*, namely *language plugins*. Such a language plugin is responsible for building the AST from a software system written in that language and thus provides support for that language to Eclipse. The basic idea relies on extracting the information of such an AST to build a model which can be used by reverse engineering applications.

Acknowledgments

I want to thank my two supervisors Oscar Nierstraz and Toon Verwaest for letting me dive into the field of modeling software systems, for their support, guidance and review of my work. In addition my thanks go to Sandro De Zanet for letting me use the Moose Brewer. I also want to thank Jorge Ressia, mainly for his support on the Java threatenings system.

Contents

Chapter 1

Introduction

Software system[1] analysis tools target different objectives. Formerly they were often defined with a specific programming language in mind. In such a case information about the entire platform dependent model of the software system was available for analysis. In practice however only a fraction is used. This fact sparked interest in the definition of an abstract representation with less platform dependent but more domain specific information. This so-called domain specific model, as opposed to platform dependent model, is not bound to a specific programming language anymore. It is rather targeted to the objective of the reverse engineering tool. In consequence, tool builders no longer have to deal with language specific information. Domain specific models belonging to the reverse engineering domain may include metric information such as NOL (number of lines). In this way tool implementation is kept easier, as the model contains several kinds of pre-computed data. Besides that, the meta-model tower approach allows the same model to be used by several tools, as long as they all use the same meta-model. These meta-models define what information is required in a model. The purpose of the tools implementing the meta-model decides the selection criteria of the information that has to be included. An example of such a domain specific model is FAMIX[8].

However, the models have to be extracted from the software system first. This requires a tool which extracts a FAMIX compliant model from a software system of an arbitrary language. Once we are able to extract models, any tool supporting FAMIX for analysis and browsing can operate on the extracted model. The model does not have to be extracted from the reverse engineering tool, as models can be moved around freely. The only requirement is that we have a tool which can extract a model of a software system written in that language. The reverse engineering tool itself is language independent, but meta-model dependent. As mentioned before, the meta-model depends on the purpose of the reverse engineering tool. This also implies that in later iterations the tool may require meta-model adaptation or another meta-model.

[1]We define the software systems to be the programs from which we want to extract a model. Also see figure 2.1

The additional flexibility of meta-models requires a new approach in model design. Meaning that it should also be possible to not only import and export models from tools, but also the corresponding meta-models. This allows these tools to support all models for which a meta-model is defined or can be imported. Therefore a hardwired higher level (minimal) self-describing meta-meta-model has to be set up. If we look at figure 1.1 then this implies that the meta-model and model can be reused by any tool implementing the minimal meta-meta-model.[2] These observations have led to implementations such as MOF, EMOF[1] and Fame[5].

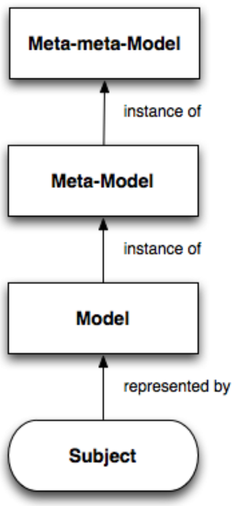

Figure 1.1: This figure shows the level of a tower.

Up to this point we notice that reverse engineering tools have been made flexible in two ways. Firstly, because of the domain specific model, the tools are not restricted to a specific programming language anymore. Secondly, because of the meta-meta-model, tools are not restricted to one meta-model anymore, as meta-models can be moved from one tool to another. The remaining question is, where the actual models come from. A common way of extracting models is by automatically mapping the platform dependent model onto a domain specific model. This requires expertise in the field of parsing technologies if done directly from the source-code. Otherwise familiarity with an implementation of an existing tool which sufficiently parses the given language is required.

In this paper we take a novel approach, which allows models from source-code in an arbitrary language to be extracted. We do not want to provide a novel way of extracting arbitrary intermediate representations by providing a novel

[2]We could also import/export the meta-meta-model for reuse, though this would result in a bootstrapping problem.

parsing approach. We want to extract FAMIX models directly from software systems. Another aim of the approach is to hide implementation details of the language by not having to generate a custom parser nor having to analyze an existing platform dependent model first. We also want a flexible tool which is as automatic as possible without restricting the control of the user. For this to be possible, we need applications which sufficiently parse source code first. Our aim is to have several similar parsing applications we can automatically reengineer to output a FAMIX model instead of an AST. We decided to use the Eclipse IDE as it has applications which parse code and generate ASTs from several languages. In the Eclipse environment these parsing applications are called language plugins. New language support can be added by defining a new language plugin. The underlying language of this IDE is Java, which implies that the format of the information of the software system has to be handled in Java. The fact that Java is *one* single language means, we know how to reengineer the this. Another nice property of Java is its support for reflection.

The paper is structured as follows. In section 2 we will explain in short how to extract a model from a software system using a three-phase approach. Section 3 covers the implementation issues of the first and the second phase. Details will be covered in section 4, where we will show some difficulties that arose during implementation. In the last sections of chapter 5 we will give an overview of how our approach was intended to be used to extract models.

Chapter 2

Model Extraction in a nutshell

In order to extract a model from a software system, current approaches conceptually use a two-step mapping. The first step consists of a source code to platform dependent model mapping. This results in a kind of intermediate representation, in most cases in an AST. This intermediate representation is still language-specific but more structured than the plain source code. In the second step, a mapping between this intermediate representation and the domain specific model is defined. Using a visitable AST, mapping can be done by linking nodes to the respective meta-model elements. Earlier implementations consisted of fully handcrafted parsers which extracted a useful intermediate representation to then extract the model.

In an effort to reduce the amount of knowledge of parsers needed, other work reuses existing intermediate representation by reengineering existing tools to output models instead of intermediate representations. This can be done by manually defining the mapping between the AST and the meta-model.[2] The mapping from the intermediate representation to the meta-model has to be redefined for every intermediate representation as the format of it depends on the language and the tool hooked in. This approach is used by the Eclipse plugin called Moose Brewer [10]. It extends Eclipse with the capability of extracting FAMIX models from Java and JSP projects. The idea is that Eclipse sufficiently parses the code for Java and JPS projects. This implies that all information for model extraction is available and thus only the mapping from the AST elements to the FAMIX elements has to be defined.

We can reuse the previously mentioned approach. The main problem is that the mappings from the AST elements to the FAMIX elements have to be defined manually for each language. In our approach we want to automate it. Eclipse provides its language support through language plugins. When a software system is imported as project the corresponding language plugin generates an intermediate representation. Looking at the approach used by the Moose

Brewer, it reengineers the Java language plugin to extract a FAMIX model from a Java project instead of the intermediate representation. Our approach will automatically reengineer language plugins to extract FAMIX models instead of intermediate representations. Being a very active open source project, the Eclipse IDE already provides support for many major languages and some dialects. New languages are added regularly by the supporting community, which means that we considered taking this *language-independent* environment.

The language plugin defines the internal representation of the software system written in that language and hosted in the Eclipse IDE.[1] The tree representation of the software system should be visitable, as Eclipse needs to access the information to provide features like syntax highlighting. The underlying language of Eclipse is Java and thus the AST of a software system has to be a Java object. All software systems hosted in this IDE, for which such a language plugin is loaded, will be automatically parsed and mapped to an intermediate representation. The language plugin defines the format of the resulting intermediate representation with Java classes. As Java is one language, we know how to reengineer it. However, we want to automatically reengineer this a parsing application, language plugin, to output a FAMIX model. We consider a three-phase approach.

The three conceptual phases in short:

1. Reengineering the language interpreter written in Java to create a custom extractor. (*Inference phase*)

2. Extracting relevant information from the software system using the reengineered tool from step one. (*Extraction phase*)

3. *Model Mapping by Example* to a domain specific model such as FAMIX.

In the next sections we will explain all the phases in detail.

We combine the pre-parsed AST with the knowledge about its format provided by the language plugin. In the first phase we extract an FM3[6][2] compliant meta-model from the language plugin. We map all Java elements to the corresponding FM3 elements. Each of them corresponds to an element represented by the language AST, which in Eclipse are all instances of Java. Concretely if we find an AST node in the language AST describing an attribute, we map it onto the FM3 attribute. This is iteratively done until we have all the AST nodes mapped onto FM3 elements. The resulting meta-model defines a mapping between the Java based language AST elements to its corresponding FM3 elements. Software systems hosted in Eclipse will be instances of such language ASTs. In order to clarify the different conceptual levels in the rest of the paper we will specifically talk about the Ruby language plugin. This is only to differentiate the underlying language of Eclipse, which is Java, from the language of

[1] *Hosted* in our case does not necessarily mean that the software system has to be imported into Eclipse. It is only important that the language plugin generates the intermediate representation from the software system. However, the easiest way to do this is importing it into Eclipse.

[2] FM3 is the meta-meta-model layer of the FAME meta-model tower. See fig. 1.1

the software system, which will be Ruby. Nevertheless, Ruby can be replaced by any other language. In a second phase we extract a model from the Ruby AST of the software system. This model is still platform specific and needs to be mapped onto a domain specific model. In this paper we will use FAMIX as domain specific model.

2.1 Inference Phase

In the *inference phase* we infer the implementation of an AST of a certain language which is defined in Eclipse. We will use the extracted model as the meta-model in the next phase. As mentioned, the architecture of Eclipse is expandable through plugins. Furthermore, the plugin of an arbitrary language also defines the structure of the intermediate representation of the software system. Eclipse uses Java as its underlying language and therefore all plugins are written in Java or at least the part describing the AST. As such, software systems hosted in Eclipse with the correct language plugin loaded, are internally represented as instances of Java classes which define the AST. These Java classes can be found in the language plugin.

In order to extract the meta-model we created a plugin called *FM3 Extractor*. This plugin extracts a model from any Java project hosted in Eclipse by using the minimal self describing FM3 meta-meta-model as meta-model. The resulting model will be FM3 compliant and can therefore be used as an FM3 compliant meta-model.

Eclipse, as an open source project, encourages developers to provide the source code of their language plugins. We can use them to extract the model. As mentioned earlier, Eclipse creates an internal representation of any software system it hosts as a project. For this setup to be possible Eclipse must have the associated language plugin of the language of the software system loaded. For Java projects the responsible plugin is JDT. Concretely we import the source code of a Java project, Eclipse generates an internal representation according to the AST description provided by the JDT plugin. JDT also provides us with a visitor. When importing Java source-code into Eclipse the JDT plugin generates a visitable AST from it. This means that by importing the Ruby language plugin as Java project in Eclipse a visitable JDT AST representation of the Ruby language plugin is generated. This limits our project only to those languages for which an open-source plugin is available.

With the help of the integrated JDT AST visitor we can extract an FM3 compliant model representing the AST nodes for Ruby software systems hosted in Eclipse. In order to do so we have to select the JDT AST subgraph corresponding to the AST nodes of the internal representation of Ruby software systems. When loading the source code of the Ruby language plugin into Eclipse, it will be represented as a JDT AST. Using the Java Fame library an the JDT AST visitor we can extract a model from it. In order to do so, the Fame library requires a mapping from the Java class elements to the FM3 elements to be defined. The Java classes correspond to the nodes of the Ruby AST. The

Fame library will, using the mapping and JDT AST nodes of the Ruby language plugin automatically extract the (meta-)model for us.

In Eclipse the project view corresponds to the JDT AST graph. All the user has to do, is select the packages(s), subgraph(s), containing the Ruby AST nodes and with the help of FM3 Extractor annotate the Java classes. This model will then be used as a meta-model in the next phase. As FM3 is self describing it can be used as meta-meta-model, meta-model or model. In this first phase the FM3 is used as a meta-model. See the meta-model tower 1.1. The quality of the model of the software system extracted in the next phase highly depends on the extracted FM3 compliant meta-model of the Ruby language plugin in this phase.

In order to avoid the implementation of an "all-in-one" tool, we implement this phase independently from the next ones. This also corresponds to the approach of the environment-independent model setup defined in the introduction. We export the resulting FM3 compliant (meta-)model to an MSE file. This file is a portable exchange format for FAME models, which allows reuse of (meta-)models in other tools.

2.2 Extraction phase

In the previous phase we extracted the meta-model of the Ruby language plugin AST. We will now use this to extract a platform dependent model of a Ruby software system. We have to consider that there might be two different implementations of Ruby language plugins. If this is the case we have to ensure that Eclipse loads the Ruby language plugin we have extracted the meta-model from.

We will use the internal Eclipse representation of the software system provided by the Ruby plugin to extract the model. However we are only interested in the (sub)graph representing the nodes of the software system. Due to the fact that Eclipse might require an extended graph of the software system, because of caching for example, the resulting intermediate representation might also include non-AST subgraphs. The meta-model of the Ruby language plugin we extracted in the previous phase defines which Java classes are AST nodes of Ruby software systems.

We define the Ruby meta-model to be the extracted meta-model of the Ruby plugin in phase one. By exporting the Ruby meta-model into the MSE format, we lost the mappings between the Ruby meta-model element and the corresponding Java class. In order for us to extract a model from a Ruby software system, mappings between the Ruby meta-model and the Java classes have to be reset. Because of Java reflection we can load any class by using a String representation of its name. Intentionally we made the names of the Ruby meta-model elements to correspond to the names of the Java classes to be loaded. The aim of this step is to extract a model from the software system written in Ruby. The information is contained in the instances of the Java classes, which are nodes of the AST of the software system. The previously restored mapping allows us to map Java instances to the meta-model instance elements. The result of this phase will be a model of the Ruby software system. As the meta-model is FAME compliant, also the model will be. Information about the ClassLoader, an instance responsible for loading Java classes, is covered in section 4.7.

The Fame framework automatically extracts a model from an instance of an Eclipse AST. This instance corresponds to the instance of the Ruby plugin AST representing the software system. Automation is possible due to the fact that we defined the higher level mapping. Figure 2.1 reveals the need for the higher mapping to be restored. In order to implement the next phase independently from the environment used in this phase we extracted the model into an MSE file.

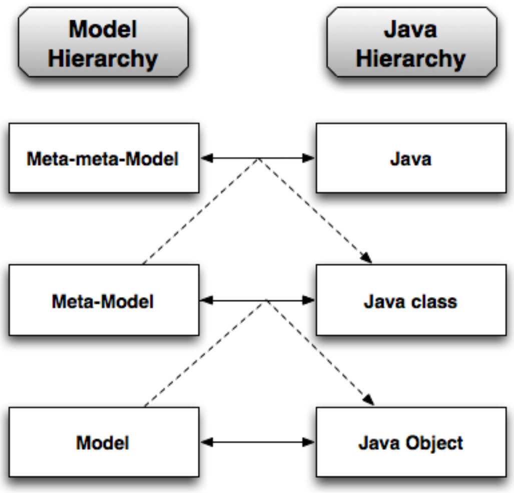

Figure 2.1: The dotted lines correspond to the extraction of a (meta-)model. The other arrows between the model and the software system hierarchy show which Java tower level corresponds to which meta-model tower element.

2.3 Model Mapping by Example phase

Our previously extracted model still contains platform dependent information and thus is not a domain specific model for reverse engineering. It could be used by very specific or very generic reverse engineering tools, as it contains the concrete syntax tree of the software system only. However such tools do not exist. In the Model Mapping by Example phase we want to transform the model into a FAMIX compliant one. With such a format it will be easier to use in several software engineering tools.

The idea behind this approach relies on Parsing by Example [3]. Parsing by Example presents a semi-automatic way of mapping source code to domain specific model mapper. Its implementation is a mix between the well-known *LALR(1) parser*[3][3], *fuzzy parsing*[4][4] and *island grammar*[5] [7].

The Parsing by Example approach provides us with the possibility to extract models from source-code by mapping them to FAMIX meta-model elements. This approach still requires knowledge about the programming language of the software system. Therefore we added the two previous phases during which we obtain a model which is expected to be easier to map than source code.

The Model Mapping by Example implementation is still ongoing work. In the next section, where we will cover the implementation, the Model Mapping by Example phase is not covered. Because of the flexibility of models, the phase does not necessarily have to be implemented in the host application. Nevertheless we must assure that the implementation uses a fame-enabled language.

[3]**L**ook **A**head **L**eft to right parser producing a **R**ightmost derivation with a look ahead of **1** input symbol

[4]Partial extraction of a source code model based on syntactical analysis

[5]Translation of source code into parts of interest (island) and irrelevant information (water).

Chapter 3

Current Implementation

The implementation of the first phase uses the Eclipse JDT plugin, the Java plugin, to extract a meta-model of the Ruby language plugin as Java project hosted in Eclipse. As mentioned we have to ensure that Eclipse loads the Ruby source-code using the same Ruby language plugin we extract the meta-model from. The *FM3 Extractor* is a plugin itself and is used as a language extractor. The second phase will have as a result a library which will reengineer Ruby plugins, called *MSE Exporter*. The user has to add a few lines of code to an existing Ruby plugin and extend this library. Using the Ruby meta-model and the *reengineering-library* the reengineered Ruby plugin will become a model extractor for Ruby software systems.

3.1 Inference Phase Implementation

To not have to implement from scratch we have taken an existing model extraction tool for Eclipse: the Moose Brewer. The main reason is that we wanted to start directly implementing without first having to deal with the plugin specifications of Eclipse. Nevertheless, Moose Brewer was not a random choice. We reengineered the Moose Brewer to extract FM3 compliant meta-models instead of FAMIX models from Java projects.

Internally Eclipse Java projects are represented as JDT ASTs, JDT being the language plugin for Java. As mentioned in the theoretical approach, the Ruby language plugin source-code has to be imported as an Eclipse Java source-code project, be hosted in Eclipse, in order to use our plugin. To activate the FM3 Extractor the user has to select the package(s) or classes describing the nodes of the AST of the Ruby language plugin which are then passed as an object to our plugin.

The main work consisted in re-defining the mapping of the Moose Brewer, which went from JDT AST nodes to FAMIX elements, to map JDT AST nodes to FM3 elements. The core architecture of the meta-model extraction was undergoing some changes. Nevertheless it is kept very simple and basically consists of the following classes:

- `FM3Builder`: Builds the FM3 compliant meta-model of the selected Java classes. These Java classes correspond to a subgraph of the Ruby language plugin representing the nodes of the Ruby AST of a software system hosted in Eclipse. These node classes are selected by the user. This class has a JavaToFM3Visitor and an FM3Registry and will also parse unparsed Java source code, obtaining JDT ASTs.

- `FM3Registry`: Registers all classes and packages and the corresponding FM3 compliant meta-model elements. This is necessary to guarantee unambiguity and consistency in the meta-model. It contains the Fame repository.

- `JavaToFM3Visitor`: When importing the Java source-code of the Ruby language plugin into Eclipse, the JDT plugin will generate a visitable JDT AST. This class extracts the facts by visiting the JDT AST nodes from the selected subgraph of the JDT AST to build the meta-model.

These classes are needed by the FM3 Extractor. When initializing our plugin an `FM3Builder` object is created. It holds all the classes of the Ruby language plugin previously selected by the user. Iteratively the `JavaToFM3Visitor` visits these classes and creates FM3 compliant meta-model elements. These are added to the `FM3Registry`. The central task of the `FM3Registry` is analyzing these to avoid duplications, perform type updates, among others. The last step consists of passing the FM3 compliant meta-model objects to the Fame repository. The repository is part of the Fame framework which provides an MSE file exporter. The approach is visually explained in figure 3.1.

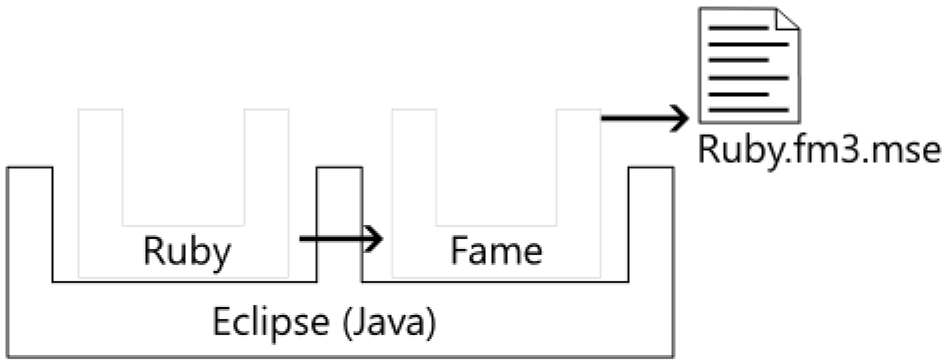

Figure 3.1: FM3 Extractor approach. The Ruby language plugin is hosted in Eclipse. FM3 Extractor extracts information through Eclipse from the Ruby language plugin and with the help of Fame generates a meta-model. Fame finally exports the obtained meta-model to an MSE file (Ruby.fm3.mse).

From the user's perspective, automated approaches often mean less effort. However, developing automated tools normally implies considering several special cases. The basic idea seems to be straightforward: Java classes are mapped onto FM3 Class, Packages onto FM3 Package and Java fields onto FM3 Property. We presumed that these are the most important Java elements to be extracted. Filters have to be applied in order to extract the most important information.

In a basic approach all relevant information should be available through fields. In order to filter irrelevant fields we might only extract information from fields which are of a primitive Type[1] or of any Type in the selected subgraph. However, language plugins in Eclipse do not have other conventions than regular Java projects. This of course makes the reengineering task harder as we have to look at language plugins as regular Java projects. It would be helpful to have a convention which requires that AST nodes can only be contained in a container of a specific type. Because of this missing convention any class can be an AST node container. This means that also `java.util.ArrayList` is a possible container for relevant classes. In such a case, the type of the object containing the information is neither primitive nor of one in the selected subgraph. Therefore information is not reachable following this approach.

Another problem is that there is no convention saying that information must be available through fields. A language plugin developer might also consider using other implementation patterns, such as hashes. Taking the example of the `java.util.HashMap`, keys and values are only available through methods and not through fields. This implies that we had to reconsider the previously considered fact of all information being accessible through fields. As mentioned earlier, we have to look at language plugins like normal Java projects because of missing conventions.

This suggests that we also have to include fields whose type is neither primitive nor in the selected subgraph. Besides that we have to take into account the information only accessible through getters which should also be included in our meta-model. These are not fields anymore but methods. However, getters should also be mapped onto FM3 properties. This results in a filter applied to methods, which could be defined as: *Getter methods are methods that contain **get** in its name.* Considering a sample class like `java.util.HashMap`, this is not an appropriate definition of getters. None of the relevant information is accessible through getters containing *get* in their name. In this case information is accessed through `java.util.HashMap.values()` and `java.util.HashMap.keySet()`. Therefore we opted for a heuristic getter definition:

- The method has to return an object or primitive other than void.
- The method does not require any arguments.

This is pretty close to the definition of relevant getters, without using heuristics. Unfortunately we will include information which is not of our interest, like `toString()` methods.

[1]By primitive we mean those which are defined as primitives by the Fame Framework

Up to this point we have only taken a closer look at which elements to include and left the type-filtering open. When adding new elements to the meta-model this question arises again. Consider the following possibilities.

- Follow all getters and all fields.

  Pro: No false negatives.

  Contra: Too many false positives.

- Only follow getters and fields with return type in selected subgraph or primitives.

  Pro: Fewer false positives.

  Contra: We might lose information. Consider the `java.util.HashMap` example.

- Follow the smallest subgraph so that all types occur as types. Only getters and fields of those types or of primitive types are included.

  Pro: Fewer false positives.

  Contra: Hard implementation.

We have opted for the first option, even while the third option seems to be the most appropriate one, because false positives can be manually excluded in the Model Mapping by Example phase. Indeed false positives can not be avoided, as primitive typed getters and fields will be included anyway.

As shown, filtering already presents a hard topic by itself. However this approach seemed to extract a meta-model containing all relevant information. During implementation we were confronted with some more problems treated in section 4.

3.2 Extraction Phase Implementation

Model extraction is done using the internal representation of the software system generated by the language plugin. We opted to only implement a library which can then be extended by a plugin, as uniformity in implementation of language plugins can not be granted. We called this library *MSE Exporter.* In order to provide a working example we have implemented a sample plugin, *Java AST Extractor,* which can be used on Java projects. Because of implementation problems[2], the library is not yet finished and therefore can not be used in productive systems. The implementation of the model extractor for any other language using the library should not be complicated. See section 5 for further information.

Our aim is to extract a subgraph of the graph provided by the Ruby language plugin of the software system to obtain a graph representing the software

[2]see section 4

system. In our case, a platform dependent model in a portable format conforming to the previously exported platform dependent meta-model. We use the Fame Framework for Java for the model repository. The implementation can be developed in two conceptual steps, a setup step and an extraction step.

In the first conceptual step we have to, as mentioned in section 2.2, reconnect the extracted language plugin description, the meta-model, with the runtime Java classes. The names of the elements of the meta-model correspond to the names of the Java classes, which makes this task easy. Therefore we decided to use Java reflection, with which we can locate classes by using the `Class.forName(String className)` method. We then map the loaded class to the corresponding FM3 compliant meta-model element. Remapping Java classes is straightforward, unlike Java fields and Java methods. This is more problematic, because methods and fields are mapped onto the same FM3 element. The result of this step will be a repository with FM3 as meta-meta-model, the previously extracted meta-model as meta-model and the mapping between meta-model elements and Java classes.

In a second step we want to extract information from a selected object, which is the internal representation of the software system we want to extract the model from. In a first approach we added the element to the tower and let Fame do the work for us. But as we cannot assume that the object will have the Eclipse AST of the software system as a root node, we have to search for the correct subgraph to add. This is because Fame always starts from the root node and tries to include all child nodes. Considering that the Ruby language AST of the software system is a subgraph in the graph we want to analyze and therefore the root node does not have a corresponding meta-model element, inclusion of the graph in the Fame repository fails.

We have to search for the entry point in the graph to add to the Fame repository. The type of the entry point should have a corresponding meta-model element and not be primitive. We use the Breadth First Search algorithm to search for the correct subgraph entry node of the Ruby language AST of the software system. Our implementation of the Breadth First Search slightly differs from the original, to perform memoization, as we know that we only have to visit a node of type A and its children once. If later a node of type A is reached, and the previous search was not successful, we can also ignore its whole subgraph. This considerably increases the speed of the Breadth First Search algorithm. The algorithm terminates its search if an object node is found with its type having a corresponding element in the meta-model not being primitive. For further information we refer to section 4.1. The steps are visually explained in figure 3.2

As mentioned earlier, the mapping of the properties in the first conceptual step was not straightforward. This is because the Fame framework requires accesses to be set for properties. With the help of these accesses, Fame is able to extract information from Java elements and visit the children of the Java objects.

Again, automation was a handicap. We were not able to decide whether the information had to be extracted from a Java field or from a Java method

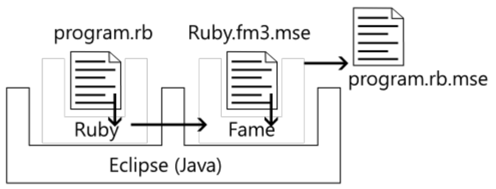

Figure 3.2: MSE Exporter approach. Fame resets the meta-model with the help of Ruby.fm3.mse, which is the meta-model from phase one in the MSE format. The Ruby language plugin (Ruby) generates the internal representation from the software system saved in program.rb. MSE Exporter extracts information of the software system through Eclipse and with the help of Fame generates the model of the software system. This is saved as an MSE file, program.rb.mse.

just by looking at a meta-model property. This ended in a trial and error algorithm, which tries to first pretend the meta-model property metadescribes a Java field; if loading fails, then most probably it metadescribes a Java method. Due to class-loading problems it was possible that no access was found for a meta-model element, so these were simply ignored. However, this approach caused some difficulties later. In a later implementation we had to disable those elements from the meta-model instead of only ignoring them.

Even if this simple approach would theoretically be enough for extracting a model from a software system written in Ruby, we were confronted with several problems. We describe the most important ones in section 4. Most of these problems were caused by bugs encountered in the implementation of the library, but also because we depend on several other tools.

MSE Exporter not only depends on Fame, but also on the quality of the extracted meta-model of FM3 Extractor and is sensitive to possible bugs[3] in the JDT and the Ruby language plugin. Besides that, understanding in detail the JDT and Eclipse architecture, the Fame modeling hierarchy, testing various different language plugins and realizing various case-studies to minimize implementation errors, would go far way beyond the scope of this project.

[3]We are not aware of any bugs in the JDT plugin having an impact on our approach. However, implementation failures in the JDT plugin might have as consequence that informations are not or only partially extracted.

Chapter 4

Implementation Challenges

In this chapter we will cover the most important difficulties we had during implementation. Not only considering several Eclipse and plugin development specific cases caused problems but also the Fame Framework. We used the Fame framework beyond its initial purpose. Each subsection will treat in short the problem and provide solutions if we found or used any. In the future we might consider a different implementation approach considering some of these problems.

4.1 Eclipse representation of a software system

Problem: When adding objects to the MSE Exporter's repository, an exception was thrown. The problem was that the root node of the objects graph did not have a corresponding meta-model element.

This problem arises in the extraction phase, the second phase in our approach. The MSE Exporter library adds the internal representation of the Ruby project as an object to the Fame repository. Such an object might also include location of source files or project settings, as additional information for Eclipse. If in the first phase we selected the correct subgraph to be our meta-model of the Ruby language plugin, we should not have a compliant meta-model element for the root node of the internal representation. Indeed this was the case in our first attempt, which forces us to think again which filters to apply on the objects to add to the repository of the MSE Exporter.

Fame automatically follows all nodes of the objects in the repository using the defined accesses of the corresponding meta-model element. If a non-primitive node does not have an access defined, an exception is thrown. We considered two different procedures to deal with that. The first would be a primitive approach in which we ask the user to extend the current meta-model with the missing elements. The problem of this approach is that the extraction can be of the whole Eclipse representation and not only the AST of the software system. In a second approach we try to find the correct subgraph representing the software system's Eclipse AST in the object graph of the provided object.

When using this approach, we have to define a few conditions. We part from the idea that all possible root elements are part of the meta-model. The first condition to be fulfilled is that nodes to be added to the repository should not be of primitive type. This has to be done, as Eclipse handles its information mostly using primitive types and this would end up in the inclusion of Eclipse specific information. A second condition is that the Java class of the object needs a compliant FM3 element in the meta-model. As described in section 2.2, this can be easily found out, by comparing the names of the FM3 element with the object's class name. We only add those element to the repository which fulfill both prerequisites.

The current implementation slightly differs from the described approach in section 3.2, as we do not only limit the number of added nodes to one. This is due to the possibility of the AST graph being split into smaller subgraphs with no connections between them. We will explain the problem with an example.

Consider the following scenario: We have one software system written in the Ruby language, which only contains two packages. We assume that the Ruby language AST does have two separate package nodes for Ruby package A and Ruby package B and that these do not have a link between them. With the first used approach the Breadth First Search algorithm stops if an non-primitive object, which has a compliant meta-model element, is found. This means that either node A or node B was found, depending on the Ruby language AST at runtime. By only adding the found object to the repository of MSE Exporter, we would also only include one package for model extraction. Therefore the entire Ruby language AST has to be visited. Added objects, and therefore added subgraphs, do not have to be re-visited.

4.2 Properties of Type Array

Problem: When trying to extract a model from a software system containing fields or methods of type Array, an unknown type exception was thrown.

It is important to know that the Fame Framework is not supposed to be used for an automated approach like ours. Therefore it was obvious from the beginning that needed to adapt some parts for it to work correctly. This can be seen in the implementation of multivalued FM3 properties. In Fame multivalued properties have to be flagged as *multivalued.* In the *Inference Phase* we set the multivalued flag to every property whose type is supposed to be multivalued. Besides all objects implementing the `java.util.List` interface, also `Array`s are considered to be multivalued. In Fame on the other hand, all multivalued properties were treated as `java.util.List`. This makes sense, as these are easier to handle than `Array`s.

In our approach we used the Java Fame modeling Framework beyond its initial purpose. We try to extract an *arbitrary* meta-model. Fame is supposed to be used to extract a model using a specific meta-model, defined in terms of the FM3 meta-meta-model. Therefore the meta-model should not be of arbitrary types. In the first phase of our approach we use a meta-meta-model as meta-model in the Fame repository to extract a meta-model, which also implies the usage of arbitrary types, such as Arrays. This caused problems, as *multivalued* Array objects are not supported by the FM3 meta-meta-model, because the type of such an object was expected to be a `java.util.List` implementations.

We flagged Array objects as multivalued, causing them to be cast to `java.util.List`. In a first approach we set the *multivalued* flag to *false*. As a consequence the object of type Array were treated as singlevalued, and the information could not be extracted. In the current implementation we set the multivalued flag to *true* for properties in our meta-model of type Array. However, we will have to convert Array typed objects to Lists in the MSE Exporter library.

At the time we were confronted with this problem, the implementation of the Fame framework for Java forced us to change the code in Fame itself. We chose to use `java.util.ArrayList` as List representation for objects of type Array.

Another approach would change the code in the MSE Exporter instead, which also implies that we change the type of an object of type Array to type `java.util.ArrayList`. As a consequence the newly defined type has to be added to the meta-model. However, the meta-model should not be changed in the library. If we do not adapt the meta-model, Fame will not find a compliant element for an object of type `java.util.ArrayList` in the meta-model and therefore not extract the information from the array object. The easiest adaption is to change Fame itself to handle array objects correctly.

In a later Fame version, the `ch.akuhn.fame.internal.Access` interface was introduced. This allowed a much cleaner approach. The interface can be used to define how to invoke/get and set methods and fields in Fame. When adding an object to the repository all of the child-objects are iteratively initialized using this definition from the meta-model property. The current Fame implementation uses this interface for FM3 compliant properties to define the accessors and setters. Previous versions used `java.util.reflect.AccessibleObject` instead. The `Access` allows us to define our own Access for objects of type Array instead of manually changing the code in Fame. In the current implementation we extract all objects of an Array and, by using this `Access`, add them to a `java.util.ArrayList` which can be handled by Fame.

The example below shows the resulting MSE file of the meta-model containing a String Array field or method called `prop`.

```
...
(FM3.Property (id: 1)
   (name: 'prop')
   (multivalued: true)
   (type (ref: String)))
...
```

Considering the instance `String[] prop = {''Hello'', ''World''}`, the resulting MSE file of the model would look like:

```
...
(prop 'Hello' 'World')
...
```

4.3 Properties of Type Dictionary or List

Problem: Failing to extract information from fields and methods of type dictionary or list.

Even if this problem seems to have the same source as the previous one, the Fame framework is not the problem in this case. Fame needs a multivalued flag to be set for lists to work correctly. Immediately the question arises which types have to be flagged as multivalued. This means that the problem relies on automated recognition. The Ruby language plugin is written in Java and therefore treated as a Java project. We visit all its nodes with the JDT visitor. But the visitor does not provide us with a method which would help us to recognize lists or dictionaries.

We assume that classes which are used as lists either implement the `java.util.List` or the `java.util.Collection` interface. So if a type whose class implements one of these two interfaces is found, then the multivalued flag is set to *true*. For this purpose we defined an interface list containing String representation of the qualified name of these two list interfaces. This approach is not the perfect solution, but the most appropriate one we found as we can easily extend the existing list if new list interfaces are defined. Nevertheless we can not assume that every list implemented in the Ruby language plugin implements the lists provided by the `java.util` package. However, if a new multivalued interface is found, then the qualified name can be added manually to the interface-list.

The example below shows the resulting MSE file of a String list field or method called `prop`. It is not surprising us that it is of the the same format as the example discussed in section 4.2. This is due to fact that from a model view we treat lists and arrays the same way.

```
(FM3.Property (id: 1)
   (name 'prop')
   (multivalued true)
   (type (ref: String)))
```

When considering a similar approach for dictionaries, we have to bear in mind that dictionaries internally contain two lists. Besides that also a new dictionary interface list has to be defined, with the same constraints as list interface lists. In our current implementation we only include `java.util.Map`. When extracting a meta-model element whose type implements an interface contained in the dictionary interface list, we have to add a new instance of an FM3 class for every Map type. These classes contain two multivalued properties. One to access the values, named `values`. And one to access the keys, named `keySet`. Keys and values can be of an arbitrary type. Nevertheless, the names of the properties are not a random choice, but comply to the names of the methods used to access the lists defined by the `java.util.Map` interface.

We take the following example to clarify what happens. Consider a field or a method called `testProp` of type `java.util.HashMap<String, Object>`. The extracted meta-model in MSE format of this property will look like:

```
...
(
 FM3.Property (id: 1)
    (name 'testProp')
    (type (ref: 5))
)
...
(
 FM3.Package(id: 3)
    (name 'java.util')
       (classes(FM3.Class (id: 5)
       (name 'HashMap<String,Object>')
    (attributes
       (FM3.Property (id: 4)
          (name 'values')
          (multivalued true)
          (type (ref: String)))
       (FM3.Property (id: 2)
          (name 'keySet')
          (multivalued true)
          (type (ref: Object)))))
    )
)
```

In the implementation of the MSE Exporter library we depend on the names of the meta-model elements. If these names would not comply to the Java class names, we would have a problem when using the `ClassLoader`. By using this approach we do not have to treat the naming of lists and dictionaries as special cases in the MSE Exporter library as we use the Java naming.

4.4 Methods throwing Exceptions

Problem: Fame calls methods on objects using the defined access. When doing this, arbitrary exceptions were thrown.

Java allows the usage of Exceptions to handle unexpected object behavior or when the class invariant is violated. This exceptions are normally thrown when a method of an object is called and not when accessing fields. When searching for a solution to this problem we were confronted with two different types of exceptions. It is up to the programmer to advise if an exception can be thrown by a method or not.

The next example will show the difference between “advised throw” and “non-advised throw”.

```
public void advisedThrow() throws SomeException{
....
}

public void nonAdvisedThrow(){
....
if(element == null){
   throw new SomeException();
}
....
}
```

The reason why we differentiate between these two types has its root in the way the JDT AST is built. Namely, advised Exceptions flag the JDT AST node of a method, while other methods do not have this flag. Therefore we could easily catch advised ones in the first phase implementation, the FM3 Extractor, and exclude those methods from the resulting meta-model. However, when using this approach, we might lose important information. The main reason would be ensuring the consistency of the resulting model extracted by the MSE Exporter. In the current implementation we dismissed this idea again. Firstly, because of the non-advised exceptions we will never be able to guarantee that no method will be included that can throw an exception. The second reason was that a method does not always throw an exception. Normally if an exception is thrown, there will not be any relevant information.

We decided to treat all Exceptions as non-advised exceptions. Therefore we have to catch them in the MSE Exporter library, when invoking a method of an object through reflection. As soon as an invocation throws an arbitrary exception we deactivate its corresponding meta-model element. Fame will then not try to extract information from this method anymore. In this way we ensure that the maximum available information is extracted.

However, one major problem remains. When a meta-model element is deactivated all following objects of that type will be erroneous. As an example take an arbitrary AST consisting of addition nodes, with a left and a right operator and an operation. Besides that we define a `getLeft()` and a `getRight()` method which throw an exception if the corresponding left or right summand is not defined. For some reason one of these nodes does not have a left summand. When invoking the method of that object an exception will be thrown. By deactivating this property, we will also lose the left summand of the other nodes. The original meta-model contains an FM3 class instance as addition node and at least two properties describing the `getLeft()` and `getRight()` methods. However after deactivation the meta-model element describing the `getLeft()` method, it will be missing in all the correct subgraphs too.

The previous example also provides us with the reason why we stick to this approach. Our example node would most probably also provide its information through fields. In short, we only lose information if the information is only accessible through a method and a call to it throws an exception.

4.5 Methods with Clone Behavior

Symptom: Arbitrary Eclipse specific exceptions were thrown.

Fame iteratively calls methods on objects. Those automated random method calls also were the reason for this problem. Mainly because we cannot control the behavior of methods. The selection criteria for *getter* methods is one reason. `Clone` methods are also interpreted as getters: they do not need any arguments and return an object or a primitive. A first solution would extend the current selection criteria by adding the condition: The return type might not be the same as the type of the caller-class. However, a method having the appearance of a *clone* method using the newly added definition can also just return the next element of an AST as in a linked list. Consider the possibility that these might not be available through fields. Even if this sounds impossible, optimization might cause an AST to only provide the next element through a method without having a field representation for the next element. Again we see how problematic the missing language plugin development convention can be.

The problem of *clone*-acting methods is that *clone*-acting methods return a new object any time they are called. This fact leads to an inconsistency which Fame can not handle. As our approach might also add multiple objects to the repository, the same object might be added twice, once as real instance and once as a *clone*. Also the Access linked to the *clone*-acting method initiates a new type every time it is called. Fame will interpret any *clone* instance as new instance. Needless to say that this is also the case if the information does not change, as in common `clone` methods. But if we disable them, we might also lose information, as clone-acting methods could provide crucial information.

Another approach directed us to create a special representation of objects, containing all relevant information. We could therefore notice if a method returns exactly the same information or not. The idea behind this is to create hash objects containing the information of the objects. This would mean that if two hashes are the same, then the objects are the same. A method returning an instance where the hash is the same as the hash of its parent object, would just link to the hash-object of the parent element. Else if no hash was found corresponding to the hash of the created object, then a new hash object is initialized. By using this approach we still have the same problems. This is because of the underlying architecture of Java. For comparison the `equals` method is used. However, the default `equals` method of the Object class only compares object references to determine equality. Our proposed approach only depends on the `equals` method. To exemplify, we assume the following setup: we have two classes, with no `equals` methods defined in `Dummy` nor in `someotherObject`.

```
public class Dummy{
   public someotherObject o;
   public Dummy next;
   ...
   public Dummy clone(){
      return new Dummy(new someotherObject(o.getInformation()), new Dummy(---
                                                            next information---));
   }
 public Dummy getNext(){
      return this.next;
   }
}
```

This means that we cannot distinguish the `getNext` and `clone` methods. The reason is that both are linked to new hash-objects. Our aim would be that the `getNext` method links to the hash of the next object and the `clone` method to the hash of itself. All AST nodes which do not implement an `equals` method would not be linked correctly. Our current solution is to disable *clone*-acting methods, which also implies that we might lose information. Unfortunately we cannot avoid this without analyzing the method itself. In a future implementation we could provide an interface and let the user choose if such a method should be disabled or not.

4.6 Looping Methods

Problem: The MSE Exporter library is caught in an endless loop.

A problem we did not consider when we started implementing the MSE Exporter was that a method could be caught in an endless loop. One possible reason might be missing information or wrong implementation. The basic idea of the solution consists of letting the methods run for a limited time only. This led to creation of threads when extracting the model element from a method. We need to find a realistic timeout, a balance between preserving most of the information and not needlessly increasing extraction time. After the timeout the thread terminates and provides the extracted information. The information might also be `null`, in which case the corresponding meta-model element is disabled. A `null` object does not contain any information and therefore execution of the method failed. We have opted for a high timeout, as the language plugin implementations will not always comply with the OO convention and can have long, almost procedural methods. The time needed for such methods is hard to estimate. We decided to use 1000 ms, ensuring that a method is not interrupted too early. Information loss is minimized this way.

To avoid having to wait 1000 ms for each method, we have opted for a wait-notify approach. A nice side effect was that we minimized the amount of concurrently started threads. In detail our approach starts a new thread, responsible for information extraction. To implement the wait-notify pattern we created an object with the function of a waiting-object. This object interrupts the current thread and waits for a notification from the previously started thread. If the thread successfully extracts facts, then the waiting-object is notified and the main thread continues. If after 1000 ms no notification occurred, then the waiting-object stops the extraction thread and the corresponding property in the meta-model is disabled.

4.7 Classloader problem

Problem: A good implementation is well-tested. In order to test our application we need to simulate a Java project hosted in Eclipse. The JDT plugin should generate an intermediate representation we need to perform the tests. Whenever we tried to run the tests the ClassLoader which initializes `IJavaElements`[1] was not loaded.

When testing our application, the ClassLoader[9] for the JDT classes could not be located. A Java program runs on a Virtual Machine and is composed of many individual Java classes. To avoid overhead they are not all loaded into memory at once; but rather on demand instead. Java uses a ClassLoader also written in Java to load them. The default ClassLoader is part of the Java Virtual Machine. However, self-defined ClassLoaders can be written in Java. This also implies that in the moment of execution the self-written ClassLoader has to be loaded in the Java Virtual Machine. As we do not know the architecture of a certain language plugin in advance, we cannot assume that it uses the default ClassLoader. And indeed, the JDT Plugin uses at least one self defined ClassLoader responsible for loading `IJavaElements`.

A basic approach would try to load the missing ClassLoader. However, this is not an easy task when targeting an automated approach. After reconsidering the problem we came to the conclusion that this will not be a realistic scenario. The main reason is that we still hook in the same application space in the Java Virtual Machine. All we have to assure to have the Classloader loaded is: when running the MSE Exporter the language plugin is also loaded.

4.8 Enumerators and Inner Classes

Problem: Eclipse does not enforce any convention on the internal representation of the software systems.

In a sample implementation of an AST, enumerators were used to represent operators. This naturally raised the question whether enumerators will be correctly extracted. Enumerators in Java are internally handled as inner classes. When testing enumerators the FM3 Extractor ended in an endless loop. Since we previously tried to extract a meta-model from the JDT plugin itself, testing showed that the JDT language plugin did not use enumerators in the implementation. However, we cannot assume that all language plugins will have an enumerator-free implementation.

[1]JDT interface which represents JDT AST nodes.

The JDT AST visitor uses a separate node for enumerators. Therefore the existing FM3 Extractor implementation only had to be extended to also extract information from enumerator nodes. These are mapped onto FM3 classes and all of its fields and methods onto FM3 properties. As an example we consider the following AST node for an unary expression. In this example the `UnaryExpression` is part of the `AST` package.

```
package AST;

public class UnaryExpression{
...
   public enum UnaryOperation {
      MINUS("-") {
         public int eval(int a)   { return -a; }
      },
      NOT("!")   {
         public int eval(int a)   { return (a == 0) ? 1 : 0; }
      };

      private String name;

      UnaryOperation(String name) {
         this.name = name;
      }

      public String toString() {
         return name;
      }

      public abstract int eval(int a);
   }
...
}
```

The extracted meta-model in MSE format looks like:

```
(FM3.Package (id: 2)
   (name 'AST')
   (classes
      ((FM3.Class (id: 3)
         (name 'UnaryExpression')
         ...
         ))
       (FM3.Class (id: 9)
          (name 'UnaryExpression\$UnaryOperation')
             (attributes
                (FM3.Property (id: 8)
                   (name 'name')
                   (type (ref: String)))
                (FM3.Property (id: 1)
                   (name 'toString')
                   (type (ref: String)))))))
```

We mentioned earlier that enumerators in Java are internally managed as inner classes. Therefore these are located in the same package as the implementing class. Looking at its name, the qualified name of the enumerator `UnaryOperation` is separated from the implementing class `AST.UnaryExpression` with a $ sign. The same representation is valid for inner classes. The next step consisted of testing the FM3 Extractor on inner classes. Surprisingly, this time no endless loop was caused. This due to the fact that the JDT AST uses the same node for inner classes as for normal classes. Nevertheless, extraction was erroneous, as the FM3 Extractor mapped the fields and methods of the inner class onto the implementing class.

Unfortunately there are no JDT AST nodes for inner classes. However a special flag is set to *true* for this classes, called *Local* and/or *Member*. In the current implementation of FM3 Extractor we use this flags to differentiate classes from inner classes. This allows us to correctly link all fields and methods of inner classes. Using this approach the MSE Exporter will, using the ClassLoader, correctly load the corresponding classes into the Java Virtual Machine.

Chapter 5

Quick Start Guide

In this section we will describe how to use our application. However, the current implementation is not meant to be used in a productive system because of many of the previously mentioned problems. You can download all files from http://smallwiki.unibe.ch/daniellangone/.

5.1 Installation

1. Copy the jar file of *FM3 Extractor* into your Eclipse plugin folder.

2. If you want to export a Java Project, you might use Java AST Extractor. Else you may ignore this step. Copy the jar file of Java AST Extractor to your Eclipse plugin folder. You have finished.

3. Create a new plugin, with which you will extract the information from a software system written in an arbitrary language. You have to add the following lines of Java code to your plugin, being `Object astObject` the representation of the software system in Eclipse created by the language plugin, `String filenameOfMetamodel` being the URL of the folder to your meta-model of the language plugin extracted with *FM3 Extractor* and `String outputFilename` being the filename where the MSE file containing the model of the software system should be saved to:

   ```
   MseExporter exporter = new MseExporter(filenameOfMetamodel);
   exporter.addToRepository(astObject);
   exporter.createMseFile(outputFilename);
   ```

4. Create a jar file from your plugin project and copy it to your Eclipse plugin folder.

5.2 How To

1. Start Eclipse. Create a Java project in Eclipse hosting the source code of the language plugin. In the case of Java this is the JDT plugin. Right click on this project and select “Extract language meta-model”. Select a location and a filename where your meta-model of language plugin should be saved to.

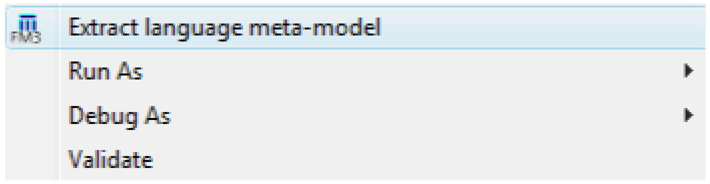

Figure 5.1: FM3 Extractor in action.

2. If you used our plugin, then right click on the Java project of the software system in the Eclipse IDE and select “Extract model using custom meta-model”. Otherwise, if you created your own plugin use it to extract the model.

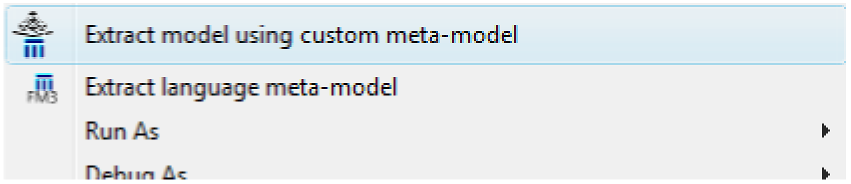
Extract model using custom meta-model
Extract language meta-model
Run As
Debug As

Figure 5.2: Our sample plugin implementing the MSE Exporter library used on a sample Java package.

Bibliography

[1] Stéphane Ducasse and Tudor Gîrba. Using Smalltalk as a reflective executable meta-language. In *International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006)*, volume 4199 of *LNCS*, pages 604–618, Berlin, Germany, 2006. Springer-Verlag.

[2] David Gurtner. Importing jsp into moose. Informatikprojekt, University of Bern, 2006.

[3] Markus Kobel. Parsing by example. Diploma thesis, University of Bern, April 2005.

[4] Rainer Koppler. A systematic approach to fuzzy parsing. *Software: Practice and Experience*, 27(6):637–649, 1996.

[5] Adrian Kuhn and Toon Verwaest. Fame - a polyglot library for metamodeling at runtime. *Models @ Runtime*, 2008.

[6] Adrian Kuhn and Toon Verwaest. FAME, a polyglot library for metamodeling at runtime. In *Workshop on Models at Runtime*, pages 57–66, 2008.

[7] Leon Moonen. Generating robust parsers using island grammars. In Elizabeth Burd, Peter Aiken, and Rainer Koschke, editors, *Proceedings Eight Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22. IEEE Computer Society, October 2001.

[8] SCG. Moose analysis technology: Famix, 2008.

[9] Greg Travis. Understanding the java classloader. *developerWorks*, 24 Apr 2001.

[10] Sandro De Zanet. Moose brewer, 2008.