Implementation of a Recoder Front-End

Yuanjun Song
Bachelor Thesis

Implementation of a Recoder Front-End

Yuanjun Song

27th of May 2009

Department of Computer Science

Växjö University

Supervisor:

Rüdiger Lincke, Phil. Lic.
Abstract

VizzAnalyzer is a program analysis tool that can be used for analyzing software programs. It relies on so called Front-Ends for information extraction from various sources like C or Java source code. It allows analyzing different source code by mapping the language specific front-end meta-model onto a common meta-model, on which analysis are defined. Right now we use the Eclipse Parser for parsing Java source code. This requires an Eclipse installation involving a large number of dependencies in order to work with Java source code. Yet, this is not always feasible; we want to be independent from Eclipse using an alternative parser.

Recoder is a Java meta-programming application program interface (API) that can be used to write Java programs that manipulate and analyze other Java programs. The Recoder framework provides over an application programming interface detailed access to the source code in form of an abstract syntax tree (AST). It has a small footprint and no external dependencies.

We create the Recoder Front-End as alternative to the existing Eclipse front-end. This includes the definition of a mapping between Recoder Front-End Meta-Model to Common Meta-Model. The mapping result will be used by VizzAnalyzer to do further analysis work. This Bachelor thesis documents relevant theory regarding Recoder Front-End and discusses its development and implementation.

Keywords: Recoder, Front-End Meta-Model, VizzAnalyzer, Common Meta-Model
Table of Content

1 Introduction ............................................................................................................... 1
  1.1 Motivation .......................................................................................................... 1
  1.2 Problem .............................................................................................................. 1
  1.3 Goals and Criteria ............................................................................................... 1
  1.4 Outline ................................................................................................................ 2
2 Background ............................................................................................................... 3
  2.1 Program Analysis Process ................................................................................... 3
  2.2 Information Extraction Process and Meta-Model Theory ................................. 4
  2.3 VizzAnalyzer ...................................................................................................... 5
  2.4 GRAIL ................................................................................................................ 6
  2.5 yEd ..................................................................................................................... 6
3 Evaluation of Java parsing tools ................................................................................. 7
  3.1 Candidates .......................................................................................................... 7
    3.1.1 Recoder ........................................................................................................ 7
    3.1.2 JavaCC + JJTree ........................................................................................... 7
    3.1.3 Jikes ............................................................................................................. 8
  3.2 Discussion ........................................................................................................... 8
    3.2.1 Criteria ......................................................................................................... 8
    3.2.1 Mapping table .............................................................................................. 8
  3.3 Details of Recoder ............................................................................................... 8
  3.4 Conclusion .......................................................................................................... 9
4 Requirements ........................................................................................................... 11
  4.1 Overall .............................................................................................................. 11
  4.2 Functional Requirements ................................................................................... 11
  4.3 Non-functional Requirements ............................................................................ 13
5 Architecture ............................................................................................................. 14
6 Design and Implementation ..................................................................................... 15
  6.1 The mapping between Recoder Meta-Model and Common Meta-Model ......... 15
  6.2 The Backend Listener API ................................................................................ 16
  6.3 Implementation of the Recoder Front-end ......................................................... 17
    6.3.1 The Big Picture .......................................................................................... 17
    6.3.2 Traversing the Base Model ......................................................................... 18
7. Case study .............................................................................................................. 20
8. Conclusion and Future Work .................................................................................. 24
  8.1 Conclusions ...................................................................................................... 24
  8.2 Future work ....................................................................................................... 24
References .................................................................................................................. 25
List of Figure

Figure 2.1: Program Analysis Process [6] ............................................................... 3
Figure 2.2: Information Extraction Process [2] ...................................................... 4
Figure 2.3: A front-end specific, a common, and a view model[2] ......................... 5
Figure 2.4: VizzAnalyzer Framework[6] ................................................................. 6
Figure 3.1: The Recoder Architecture [5] ............................................................. 9
Figure 5.1: Recoder Front-End Architecture ....................................................... 14
Figure 6.1: Traversal Principle ............................................................................ 19
Figure 6.2: Mapping the MethodDeclaration Element ........................................... 19
Figure 6.3: Mapping the MethodReference relation ............................................. 20
Figure 7.1: integrated screenshot ....................................................................... 22
Figure 7.2: local zoom ....................................................................................... 22
1 Introduction

Many studies show that software maintenance is expensive today – estimations are the lion’s share of the total costs of ownership for a software system. In all maintenance tasks, systems need to be comprehended first, and the effort for comprehension even dominates the total maintenance effort. Here, estimations can reach 90% [1].

These are the reasons that there are more and more analysis tools supporting the assessment and prediction of software quality. Although the principle tasks of these analysis tools are similar; they incorporate an information extraction and analysis process: extracting information, analyzing, and, finally, modifying the program [2]. Usually some changes of the programs and new programming languages’ using cost the refactoring of the analysis tools. In order to reuse the analysis, the research at Växjö University focuses on an extensible meta-model for program analysis: Common Meta-Model that is independent of specific programming languages. The proof-of-concept implementation—VizzAnalyzer—is a standard- and metric-based software quality model program analysis tool based on the Common Meta-Model. It allows analyzing Java source code by mapping the Java language specific front-end meta-model onto a common meta-model, on which analysis are defined. Right now it uses the Eclipse Parser for parsing Java source code.

1.1 Motivation

The Eclipse Parser is part of the Eclipse IDE, and it is currently not designed to run outside of the context of the Eclipse framework. It depends on a large number of components and is therefore not suitable if a light-weight solution is required. For example when integrating the analysis in a build process. For this reason, we want to be independent from Eclipse using an alternative parser.

1.2 Problem

The problem of this thesis can be summarized as:

The problem tackled by this thesis is to find and integrate an alternative parser to analyze the Java programs. It shall be integrated into the information extraction and analysis process as implemented by the VizzAnalyzer tool.

This is a complex problem and can be divided into three parts. First understand the elements of the complex Java programming language. Second define a mapping between the Java front-end specific Meta-Model and the Common Meta-Model. And finally, implement this mapping as font-end and integrate it into the VizzAnalyzer. However the later is not straight forward. Additionally, in order to get access to the Java Meta-Model, a suitable compiler/parser needs to be identified. The Common Meta-Model and Java language specification will be used as two major inputs to decide which nodes and edges need to be created.

1.3 Goals and Criteria

This section describes the goals pursued by this thesis in order to solve the problem and the criteria used for validating the goal:
• The first goal is to find and evaluate a suitable Java front-end providing access to the Meta-Model according to Java language specification. The goal is reached when a Java program has been parsed by the front-end in form of a prototype program allowing access to the essential elements of a Java program. Additionally, the elements of the front-end need to be understood.

• The second goal is to define mapping between Java front-end Meta-Model elements and the elements of the Common Meta-Model. This mapping is crucial step for the information extraction process. We define a set of mapping functions to ensure the “safe” transformation between java specific Meta-model and Common Meta-Model. This goal is reached when Java front-end Meta-Model has been formally defined.

• The third goal is to implement the previously defined mapping as a visitor for traversing the AST structure of a Java program. This goal is reached, when relevant information has been gathered from the front-end specific meta-model and a new abstract AST conforming to the Common Meta-Model using event-based architecture has been created. The second and third goal might be achieved in incremental and iterative steps.

The outcome of this thesis is a plug-in for the VizzAnalyzer, which will use the information extracted from the Recoder front-end plug-in for its further analysis.

1.4 Outline

The structure of this thesis is as follows. Chapter 2 provides background knowledge which is essential to understand methods and technologies used. It covers several theoretical aspects and explains the concepts and tools mentioned often in this thesis. Chapter 3 describes the tools which can be used to create Java front-end. We did some research into several candidate tools and made a decision which one to use. Chapter 4 indicates the requirement specification, including functional requirements and non-functional requirement. Chapter 5 introduces architecture of the front-end used for data extraction and its position in the information extraction process implemented by the VizzAnalyzer. Chapter 6 covers several design aspects from mapping between Java specific Meta-Model and Common Meta-Model and implementation aspect, describes implementation detail of setting working environment and setting up the model. Finally, in Chapter 7 we draw conclusion of the thesis and describes future work.
2 Background

Before we go into details of the selection of a suitable front-end, and the requirements, analysis, design and implementation of the proposed solution, we explain the concept of the information extraction and program analysis process. This section begins with the theoretical issues needed to understand the thesis. First, we introduce in general the program analysis process. Then, we focus on the information extraction and abstraction representing a part of the program analysis process. In connection to this, we explain the role of the meta-models used. Furthermore, we discuss the VizzAnalyzer Framework implementing the described processes. Finally, Grail and yEd, which are essential tools for the implementation and testing of our solution are described separately briefly.

2.1 Program Analysis Process

Figure 2.1 shows the whole process of program analysis [6]. We distinguish two separate domains, namely Software Analysis and Information Visualization, which are connected by the Software Visualization specific use. The Software Analysis Domain deals with information extraction and information analysis. For the first step, we can take advantage of parsers which create a Base Model from source documents. In other words, the Base Model is another representation of the source documents which allows operations usually not available in the source documents, like navigation, search, and random element access. Then, a further step needs to be done since the information in the Base Model cannot be used directly by software visualization. Information tailoring is necessary here which means we need to work on the transformation from Base Model to Model. What a Front-end does is to accomplish part of the transformation from Base Model to Model. Firstly, it gathers information from Base Model and then after some information processing, it creates an intermediate model (between Base Model to Model) which will be used in further analysis [6]. After this, a view is defined on the Model, which depicts the model elements as visual elements on the screen, e.g., as tables or 2D or 3D pictures. The mapping from a Model to a View using software specific view elements defines Software Visualization. How this view is visualized, navigated and manipulated is part of the Information Visualization Domain. It includes view mapping and image creation.

Figure 2.1: Program Analysis Process [6]
2.2 Information Extraction Process and Meta-Model Theory

The problem to solve in this thesis focuses on information extraction and analysis part, which is part of the software Analysis Domain. In detail we need to create an alternative way to extract information from Java source documents, and to map (analyze) the associated Base Model to a suitable Model.

This information extraction process is based on model transformations. It uses the abstractions as seen in Figure 2.2. A meta-meta model uses tree grammars and relations to describe a less abstract meta-model. This in turn outlines the mapping process and related architecture for information extraction and mapping in general. This process is then instantiated as model for a particular programming language, allowing to extract and transform any programs of this language.

In detail, model and meta-model are described according to information processing from source documents to metrics analysis which is divided into four steps: Information extraction, mapping to common representation, view abstraction and analysis [6].

The beginning step is information extraction which takes source files as input. After parsing the source file (information extraction), a special model reflecting the original structure of source file will be created. This step is described in Figure 2.1 as transformation from source file to base model. Each front-end meta-model understands a specific program representations [6]. This is currently handled by the Eclipse Parser, and we are looking for an alternative tool.

The second step is language mapping. A language mapping is a set of defined functions specifying how elements belonging to one meta-model are mapped to elements of another meta-model. Particularly, in this thesis, language mapping exists between Java front-end specific meta-model and Common Meta-Model. Common Meta-Model is an abstraction from front-end specific details. It is designed to ensure analyses are language independent. Because Common Meta-Model will be used by analysis, it should always evolve according to the requirements from analyses [6]. Since the mapping depends on the font-end used, we need to provide a new mapping for the new front-end we select.

The next two steps are view abstraction and analysis. View abstraction provides an intermediate between Common Meta-Model and analysis which make it possible for analysis operated on some certain views instead of directly on Common Meta-Model. Due to the architecture and common meta-model, these steps remain unchanged, and are not of concern for our problem at hand.

Figure 2.2: Information Extraction Process [2]
Figure 2.3 gives us an example to show how front-end specific model, common model and view model work to abstract the same information [2]. The first diagram (leftmost) contains more information than the other two because it reflects the original structure of source file and has not been tailored. In the second diagram (middle), it can be seen that several nodes and edges have been discarded from the first diagram to adapt the front-end specific model into common model. The last diagram (rightmost) is a view model which selects some nodes and edges from common-model according to the requirements of certain metrics [6].

The task of this thesis is to create the mapping between Java front-end specific model and Common Meta-Model. This is the second step of information processing explained above. It will take a Base Model generated by a suitable Front-End as input and create a new tree according to Common Meta-Model.

2.3 VizzAnalyzer

The VizzAnalyzer is a stand-alone tool developed at Växjö University based on the software analysis process and Common Meta-Model which have been explained in the two previous sections.

Three sub-processes “Retrieval”, “Analysis” and “Visualization” exist, see Figure 2.4. They map to the stages of the program analysis process as follows; “Retrieval” can also be called as “Front-end” which provides prerequisites for further analysis. It covers the whole “information extraction” and part of “analysis” in Figure 2.1. “Analysis” in VizzAnalyzer is corresponding to the rest part of “analysis” in Figure 2.1. And the last sub-process “Visualization” focuses exactly on “Information Visualization” Domain.

VizzAnalyzer has a flexible architecture. Each of the sub-processes is defined by specific components which are plugged via wrappers into respective extension points. In Figure 2.4, “Front-end”, “Analyzer”, “Metrics”, “yEd”, “Vizz3d” all play a plug-in role in VizzAnalyzer.
2.4 GRAIL
GRAIL is the abbreviation of GRAph ImpLementation. It is a graph library created by
development team of VizzAnalyzer which is regarded as an internal data representation
for VizzAnalyzer. In particularly, Common Meta-Model is stored using data structure of
Grail.

This representation consists of an annotated graph, where each graph entity (nodes,
edges, and the graph itself) has a data object and a set of predicates attached to it.

The main usage of Grail in this project is to provide node and edge type definitions
(grail.properties.TypeValues), and to store the generated graphs in the GML file format
for further usage.

2.5 yEd
yEd is a graph editor which is completely written in Java. yEd supports generating
graphs and it can be used to lay out complex graph structures in an efficient and
automatic manner. Several different kinds of layout are available at present, each of
which has its own features and advantages.

Grail can export graphs into GML files (graph markup language) which can be
displayed by yEd. Thus, graph of Grail can be easily viewed for debugging and other
purposes.
3 Evaluation of Java parsing tools

This section describes several tools which could be used to parse Java programs and to provide suitable Abstract Syntax Tree for further information extraction and tree-traversing mechanism. First, we give three candidate tools with some detail description. And several criteria for selecting the suitable tool are raised and a mapping between criteria and candidates is created. In the last section, we give our selection and reasons for selecting that tool and the detail of the tool.

3.1 Candidates

During our initial research we identified three candidate tools, which seemed promising for serving our purpose. Each one has a name, vendor, website and brief description.

3.1.1 Recoder

<table>
<thead>
<tr>
<th>Name:</th>
<th>Recoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor:</td>
<td>free</td>
</tr>
<tr>
<td>Description:</td>
<td>RECODER is a Java framework for source code metaprogramming aimed to deliver a sophisticated infrastructure for many kinds of Java analysis and transformation tools. It is a very fast Java source code frontend, including name- and type-analysis as well as providing cross-reference information. Also it has some Abstract Syntax Tree (AST) transformations, including undo functionality and (partially) incremental model updates.</td>
</tr>
</tbody>
</table>

3.1.2 JavaCC + JJTree

<table>
<thead>
<tr>
<th>Name:</th>
<th>Java Compiler Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor:</td>
<td>free</td>
</tr>
<tr>
<td>Website:</td>
<td><a href="https://javacc.dev.java.net/">https://javacc.dev.java.net/</a></td>
</tr>
<tr>
<td>Description:</td>
<td>Java Compiler Compiler is the most popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions, debugging, etc.</td>
</tr>
</tbody>
</table>
3.1.3 Jikes

<table>
<thead>
<tr>
<th>Name:</th>
<th>Jikes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vender:</td>
<td>free</td>
</tr>
<tr>
<td>Website:</td>
<td><a href="http://jikes.sourceforge.net/">http://jikes.sourceforge.net/</a></td>
</tr>
<tr>
<td>Description:</td>
<td>Jikes is a high-performance, highly compatible Java compiler that can be used on almost any computing platform makes it an interesting program and worth investigating for almost any Java programmer. It is no longer being updated.</td>
</tr>
</tbody>
</table>

3.2 Discussion

To decide which tool to use, we need to give selection criteria and to have a look at whether the tools meet our criteria.

3.2.1 Criteria

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>The tool should be free.</td>
</tr>
<tr>
<td>C2</td>
<td>The tool should be light weight and easy to be installed and used on a variety of platform.</td>
</tr>
<tr>
<td>C3</td>
<td>The tool could access to the Java Abstract Syntax Tree (AST).</td>
</tr>
<tr>
<td>C4</td>
<td>The tool should easily be integrated in the Meta-Model Theory/Information Extraction Process.</td>
</tr>
<tr>
<td>C5</td>
<td>The tool should be up-to-date and have a long-term perspective.</td>
</tr>
</tbody>
</table>

3.2.1 Mapping table

<table>
<thead>
<tr>
<th>Criteria</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recoder</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>JavaCC</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Jikes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

From the mapping table, we selected Recoder as the tool used to develop the front-end. The reasons can be stated as following again: Recoder is an inhouse tool, we can access to the code via its API. Besides, it is light weight, and easy to integrate in the information extraction process.

3.3 Details of Recoder

Recoder is a Java meta-programming API that can be used to write Java programs that manipulate and analyze other Java programs. The Recoder framework provides over an application programming interface detailed access to the source code in form of an AST. Figure 3.1 shows is the process of meta-programming, where a program manipulates data representing another program [5].
The following gives a short description of the different layers of Recoder features:

**Parsing and unparsing of Java sources codes:**
- In addition to abstract model elements, Recoder also supports a highly detailed syntactic model - no information is lost. Comments and formatting information are retained. The pretty printer is customizable and will be able to reproduce the code (possibly improving upon it, but retaining given code structures) and to embed new code seamlessly.

**Name and type analysis for Java programs:**
- Recoder can infer types of expressions, evaluate compile-time constants, resolve all kinds of references and maintain cross reference information.

**Transformation of Java sources:**
- Recoder contains a library of small analyses, code snippet generators and frequently used transformations.

**Incremental analysis and transformation of Java sources:**
- Transformations change the underlying program model; for incremental and iterative use, this model has to be updated accordingly. Transformations have to take care of dependencies by updating their local data and setting back matching positions when necessary; however, recoder will analyze change impacts for its model and perform updates automatically.[5]

In this thesis, we focus on the first and second features.

### 3.4 Conclusion
Recoder is a Java meta-programming API that can be used to write Java programs that manipulate and analyze other Java programs. The Recoder framework provides over an application programming interface detailed access to the source code in form of an AST. VizzAnalyzer and Recoder now both maintained some teachers of MSI in Växjö
University. Based on the above factors, a Recoder front-end which defined mapping between Java front-end meta-model and Common Meta-Model is desired by us.
4 Requirements

This chapter describes requirements for the development of the Recoder-based VizzAnalyzer Front-end. Requirements are divided into two sections. The first section discusses functional requirements and the second section covers the non-functional requirements.

4.1 Overall
RECODER front-end is a program will be used by VizzAnalyzer to analyze Java programs. This section gives detailed description about product perspective to help understand it from several aspects.

User Interfaces:
RECODER Front-end will be used by VizzAnalyzer, so there is no user interface required.

Hardware Interfaces:
RECODER Front-end does not need to be interactive with hardware. We do not need to take care of hardware interfaces.

Software Interfaces:
RECODER front-end uses RECODER as the compiler of Java programs and is built as a project of Eclipse. RECODER front-end will be used by VizzAnalyzer, interface of VizzAnalyzer is needed.

Memory Constraints:
We want Java front-end to be able to run fast. The time of the process (parsing and traversing are the main parts) depends on the size of the program. Since the programs need to be analyzed are in large scale, the large memory will be required.

4.2 Functional Requirements
In this section, we list the function requirements on the Recoder front-end, i.e., focusing on what should be done by Recoder front-end. Each of the functional requirements has a brief description, Pre-condition, post-condition, related FR and priority. We refer to the Recoder front-end with ‘the system’.

<table>
<thead>
<tr>
<th>FR1:</th>
<th>Get source files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>The system should be able to get all source files from open projects in a work bench window.</td>
</tr>
<tr>
<td>Pre-condition:</td>
<td>There are some Java source files in currently opened WorkBenchWindow.</td>
</tr>
<tr>
<td>Post-condition:</td>
<td>Source files has been recognized, read and prepared to be traversed.</td>
</tr>
<tr>
<td>Related FR:</td>
<td>FR2: Traverse source files</td>
</tr>
<tr>
<td>Priority:</td>
<td>Essential</td>
</tr>
<tr>
<td><strong>FR2:</strong> Traverse source files</td>
<td><strong>FR2.1:</strong> Get information from source files</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td><strong>Description:</strong> It should have a mechanism to traverse source files.</td>
<td><strong>Description:</strong> When it is traversing source files, it should be able to get information needed to map Java front-end Meta-Model to Common Meta-Model.</td>
</tr>
<tr>
<td><strong>Pre-condition:</strong> FR1 has been processed.</td>
<td><strong>Pre-condition:</strong> A source file is under traversing.</td>
</tr>
<tr>
<td><strong>Post-condition:</strong> Source files have been traversed, useful information has been gathered, process of mapping Java front-end Meta-Model to Common Meta-Model has been finished, and a new AST tree has been constructed.</td>
<td><strong>Post-condition:</strong> Gathered useful information and prepared to use the information to create a new AST</td>
</tr>
<tr>
<td><strong>Related FR:</strong> Contain FR2.1 Get information from source files Contain FR2.2 Create a new AST</td>
<td><strong>Related FR:</strong> FR2: Traverse source file FR2.2: Create a new AST</td>
</tr>
<tr>
<td><strong>Priority:</strong> Essential</td>
<td><strong>Priority:</strong> Essential</td>
</tr>
</tbody>
</table>

12
### 4.3 Non-functional Requirements

<table>
<thead>
<tr>
<th>NFR</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFR1:</td>
<td>Based theory Its design should be based on existing theory, such as Meta-Model Theory, event-based architecture.</td>
<td>Essential</td>
</tr>
<tr>
<td>NFR2:</td>
<td>Suitable for VizzAnalyzer It should be suitable for VizzAnalyzer to use.</td>
<td>Essential</td>
</tr>
<tr>
<td>NFR3:</td>
<td>Result should be repeatable Result got from the system should be repeatable.</td>
<td>Essential</td>
</tr>
<tr>
<td>NFR4:</td>
<td>Design in a clear way Its design and implementation should be easily understood.</td>
<td>Desirable</td>
</tr>
<tr>
<td>NFR5:</td>
<td>Run fast It should be as fast as possible (related to memory constraint)</td>
<td>Desirable</td>
</tr>
</tbody>
</table>
5 Architecture

This chapter describes the architecture of Recoder Front-end and its connection to the VizzAnalyzer. The first section explains how Recoder front-end adapts itself to Meta-Model theory and how event-based architecture is used.

Figure 5.1 shows how the Recoder front-end relates to the information extraction process we discussed in Section 2.2. It also shows the event-based architecture used in the Recoder front-end.

In Figure 5.1 we can see that the Recoder front-end covers the first two of the total four steps in of the information extraction process, i.e., Information Extraction and Language Mapping; mapping to a language-independent representation, the Common Meta-Model.

In the first step, we take advantage of Recoder API. This provides access to Java source files which are interpreted as an AST structure with predefined ASTNodes and “contains” relationships between nodes. This representation conforms to the front-end specific (Recoder specific) Base Model.

The second step reuses an Event-based architecture which is defined by Lincke el. al. [2] and has already been used with Java and UML front-ends. As depicted in Figure 5.1, the language mapping step is divided into two parts: MapsA and MapsB. In MapsA, a “visitor” traverses the Recoder specific data structure (AST) and sends events to a common backend listener which gathers information from MapsA and constructs a new data structure which corresponds to the Common Meta-Model.

The Recoder front-end component is added as plug-in to the front-ends extension point of the VizzAnalyzer, as described in Chapter 2. The selection of source files is provided by the VizzAnalyzer, the Grail graph conforming to the Common Meta-Model is returned to the VizzAnalyzer for further processing, including analysis and visualization.

![Figure 5.1: Recoder Front-End Architecture](image-url)
6 Design and Implementation

This chapter transfers the abstract architecture defined in the previous chapter into a more detailed design. Related design issues are described. In particular, we present the mapping between the Java specific Meta-Model elements and the Common Meta-Model (CMM) elements. We also explain how the mapping between Recoder Meta-Model and Common Meta-Model is constructed, i.e., the API of the backend listener, which is used by the traversing visitor to generate the CMM elements.

6.1 The mapping between Recoder Meta-Model and Common Meta-Model

This section lists the defined elements of the Recoder specific Meta-Model in the right and the counterparts from the Common Meta-Model in the left side of Table 6.1. The upper part of the Table 6.1 shows the mapping for the entities, and the lower part for the relations.

Table 6.1: Mapping of elements

<table>
<thead>
<tr>
<th>Element of Recoder Meta-Model</th>
<th>Element of Common Meta-Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompilationUnit</td>
<td>File</td>
</tr>
<tr>
<td>PackageDeclaration</td>
<td>Directory</td>
</tr>
<tr>
<td>ClassDeclaration</td>
<td>Class</td>
</tr>
<tr>
<td>InterfaceDeclaration</td>
<td>Interface</td>
</tr>
<tr>
<td>MethodDeclaration</td>
<td>Method</td>
</tr>
<tr>
<td>ClassInitializer</td>
<td>Initializer</td>
</tr>
<tr>
<td>ConstructorDeclaration</td>
<td>Constructor</td>
</tr>
<tr>
<td>FieldDeclaration</td>
<td>Field</td>
</tr>
<tr>
<td>ParameterDeclaration</td>
<td>FormalParameter</td>
</tr>
<tr>
<td>LocalVariableDeclaration</td>
<td>Variable</td>
</tr>
<tr>
<td>AnnotationPropertyDeclaration</td>
<td>Method</td>
</tr>
<tr>
<td>ArrayInitializer</td>
<td>Initializer</td>
</tr>
<tr>
<td>Operator</td>
<td>Operator</td>
</tr>
<tr>
<td>Do</td>
<td>Do</td>
</tr>
<tr>
<td>For</td>
<td>For</td>
</tr>
<tr>
<td>EnhancedFor</td>
<td>EnhancedFor</td>
</tr>
<tr>
<td>If</td>
<td>If</td>
</tr>
<tr>
<td>Switch</td>
<td>Switch</td>
</tr>
<tr>
<td>While</td>
<td>While</td>
</tr>
<tr>
<td>EnumDeclaration</td>
<td>Enumeration</td>
</tr>
<tr>
<td>FieldReference</td>
<td>FieldRef</td>
</tr>
<tr>
<td>Implements</td>
<td>ImplementsRef</td>
</tr>
<tr>
<td>Extends</td>
<td>ExtendsRef</td>
</tr>
<tr>
<td>TypeReference</td>
<td>TypeRef</td>
</tr>
<tr>
<td>VariableReference</td>
<td>AccessRef</td>
</tr>
<tr>
<td>MethodReference</td>
<td>MethodRef</td>
</tr>
<tr>
<td>other elements are currently omitted</td>
<td>NotYetDefined</td>
</tr>
</tbody>
</table>

Usually, we create edges for the References, and create nodes for the rest elements by calling the appropriate methods in the backend listener API described in the next section.
6.2 The Backend Listener API

Backend is a simple and flexible interface which actually can connect with several “traversing visitor” in MapsA, Figure 5.1. The major usage of it is to create a tree with nodes and reference relations between nodes. Five methods exist in Backend, they are:

1. Called upon entering a node which has no children during AST traversal.

```java
public void addLeaf(TypeValues nodeType, Object nodeKey, String label, Attribute[] attributes);
```

2. Called upon entering a node during the AST traversal. None of the children have been visited yet. A new node is created and connected with the previous node in the tree. Filtering is applied.

```java
public void enterNode(TypeValues nodeType, Object nodeKey, String label);
```

3. Called upon exiting a node during the AST traversal. All of the children have been visited. Attributes are attached to the node, if any.

```java
public void exitNode (TypeValues nodeType, Object nodeKey, Attributes[] attributes);
```

4. Adds cross edges between the nodes of the constructed containment tree. It is called as soon as a outgoing edge is resolved during the AST traversal. The "from" node is therefore always existing in the containment tree. But it is possible that the "to" node has not yet been processed and is therefore not contained in the containment tree. Therefore the edge needs to be buffered until the complete containment tree is build. It will be resolved upon calling completeGraphs().

```java
public void addCrossRefenceEdge (TypeValues edgeType, Object srcKey, Object tgtKey, Attribute[] attributes);
```

5. Completes the graph construction, adds the cross reference edges and cleans up buffers and caches.

```java
public void completeGraphs();
```

A back-end has already existed. We can reuse it directly in the Recoder front-end calling the appropriate methods from the Visitor.
6.3 Implementation of the Recoder Front-end

This section covers two implementation details of the Recoder Front-End. The first explains the big picture, including on how to get the Java source files to traverse, setting up the Visitor and Backend, receiving the completed Common Meta-Model conforming data graph, and on how to store the GML file for further usage. The second discusses implementation details of the Visitor, that is traversing the Recoder specific Base Model from which the Common Meta-Model conformant nodes and edges are created according to the mapping defined in the previous chapter.

6.3.1 The Big Picture

First, download the latest version of RECODER and GRAIL. Then, import them with cmm and backend into Eclipse project. The next step is to get the source files and some configuration.

```java
CrossReferenceServiceConfiguration crsc = new CrossReferenceServiceConfiguration();

specify input path, containing all source (.java) and libraries. In this example, the path is "F:\eclipse3.5\project\AssignmentTask1\src".

crsc.getProjectSettings().setProperty(PropertyNames.INPUT_PATH, "F:\eclipse3.5\project\AssignmentTask1\src");

create a graph

DirectedGraphInterface graph = new SetBasedDirectedGraph();

set type of graph

graph.setProperty(GraphProperties.TYPE, CommonMetaModel20.COMMON_META_MODEL);

set name of graph

graph.setProperty(GraphProperties.LABEL, "Test Graph");

create root node of graph and set type

DirectedNodeInterface root = graph.createNode(Integer.valueOf(0));
root.setProperty(GraphProperties.TYPE, CommonMetaModel20.Solution.getType());
root.setProperty(GraphProperties.LABEL, "Test Root");

add root node to graph

graph.addNode(root);

initialize backend as event listener, provide graph, root node and

VizzBackEndListener backend = new DefaultBackEnd(graph, root,new
create the visitor to be called for each program element

for each compilation unit (.java file) parse the AST

traverse all elements of the AST in DFS order

Complete the graph

Create GML file

6.3.2 Traversing the Base Model

In this section, we explain the traversal and mapping process, as implemented in the Visior. Since covering all elements in detail would exceed the detail feasible in this thesis, we ask the reader to look at the source code for details on all mapped elements. We explain the general concept on an representative example: we choose the element MethodDeclaration and relation MethodReference as two representative cases, which we explain in detail. But first, we explain how the traversal works in general. Each element will be traversed two times. "isVisit" is the field to distinguish the first time the element be traverse or the last time. Figure 6.1 shows the traverse principle.
Mapping the MethodDeclaration element from the Recorder Meta-Model to the Common Meta-Model. As we can see in Figure 6.2, first, it identify that it is the first or second time the MethodDeclaration element being visited. If it is the first time, call the backend listener to create a node with the Method type (as node type), the full name (as label) and the ID (as key) of the element. Then continue visit the children nodes. Otherwise add the attributes to the node which was created when it was visited the first time. Then leave the element and visit back to the parent node.

```java
public void visitMethodDeclaration(MethodDeclaration x) {
    String fullName = x.getFullName();
    if (isVisit) {
        System.out.println("Visiting MethodDeclaration: "+ fullName);
        backend.enterNode(CommonMetaModel20.Method.getType(),
                          ""+x.getID(), fullName);
    }
    else {
        System.out.println("End Visiting MethodDeclaration: "+ fullName);
        Attributes attrs = new Attributes();
        attrs.addAttribute(CommonMetaModel20.isAbstract, x.isAbstract());
        attrs.addAttribute(CommonMetaModel20.isStatic, x.isStatic());

        backend.exitNode(CommonMetaModel20.Method.getType(), ""+x.getID(),
                       attrs.getAttributes());
    }
}
```

Figure 6.2: Mapping the MethodDeclaration Element

Mapping the MethodReference relation from the Recorder Meta-Model to the Common Meta-Model. As we can see in Figure 6.3, first, like the process of mapping the element, it identify that it is the first or second time the MethodReference element being visited. But the follow steps are different. In the first time visiting process, call the backend listener to create a cross edge from the caller node to the callee node with the MethodRefEdge type (as edge type),
MethodRef type(node type), the full name of the element(as label), the key of the callee node and the attributes, if the callee node was or will be created. If the callee is the utility method which defined by Java itself, just create a MethodRef node and create a cross edge from the caller node to the newly created node. In the second time visiting process, there is nothing we should do.

```java
public void visitMethodReference(MethodReference x) {
    String fullName = x.getName();
    if (isVisit) {
        System.out.println("Visiting MethodReference: " + fullName);
        Attributes attrs = new Attributes();
        if(this.osrGraph.getSourceInfo().getMethodDeclaration(this.osrGraph.getSourceInfo().getMethod(x)) != null)
            this.addCrossReferenceEdge(
                CommonMetaModel20.MethodRefEdge.getType(),
                CommonMetaModel20.MethodRef.getType(),fullName , attrs
                .getAttributes());
    } else {
        this.addCrossReferenceEdge(
            CommonMetaModel20.MethodRefEdge.getMethodDeclaration(this.
                osrGraph.getSourceInfo().getMethod(x)).getID(), attrs
                .getAttributes());
    }
}
```

Figure 6.3: Mapping the MethodReference relation

7. Case study

This chapter uses a test to show the case study of analyzing a program with our front-end. It describes how long it tooks, number of the test files, the different elements in the graph, etc.

The following is the test log, integrated screenshot and local zoom:

Back-End Overview
-----------------
1962 ListenerEvents
844 Nodes
844 ContainsEdges
348 CrossRefEdges (resolved in graph: 132)

Graph name: Test Graph
Include Filter: null
Exclude Filter: null

Back-End detail
-----------------
844/0 nodes (unfiltered/filtered)
844 Enter node events total
### 348/0 Filtered cross-references

(filtered/unfiltered)

348 Add cross-reference events total

<table>
<thead>
<tr>
<th>References resolved</th>
<th>References aggregated</th>
<th>References not resolved</th>
<th>Error resolving references</th>
<th>References total</th>
</tr>
</thead>
<tbody>
<tr>
<td>132</td>
<td>0</td>
<td>216</td>
<td>0</td>
<td>348</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unfiltered/graph</th>
<th>filtered/!scope</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>------------------</td>
<td>-----------------</td>
<td>------</td>
</tr>
<tr>
<td>79/79</td>
<td>0/0</td>
<td>AccessRef</td>
</tr>
<tr>
<td>79/79</td>
<td>0/0</td>
<td>AccessRefEdge</td>
</tr>
<tr>
<td>12/12</td>
<td>0/0</td>
<td>Class</td>
</tr>
<tr>
<td>0/1</td>
<td>0/0</td>
<td>Common Meta-Model 2.0.10</td>
</tr>
<tr>
<td>3/3</td>
<td>0/0</td>
<td>Constructor</td>
</tr>
<tr>
<td>1/1</td>
<td>0/0</td>
<td>EnhancedFor</td>
</tr>
<tr>
<td>2/2</td>
<td>0/0</td>
<td>Enumeration</td>
</tr>
<tr>
<td>3/3</td>
<td>0/0</td>
<td>ExtendsRef</td>
</tr>
<tr>
<td>3/3</td>
<td>0/0</td>
<td>ExtendsRefEdge</td>
</tr>
<tr>
<td>16/16</td>
<td>0/0</td>
<td>Field</td>
</tr>
<tr>
<td>30/30</td>
<td>0/0</td>
<td>FieldRef</td>
</tr>
<tr>
<td>30/9</td>
<td>0/21</td>
<td>FieldRefEdge</td>
</tr>
<tr>
<td>7/7</td>
<td>0/0</td>
<td>File</td>
</tr>
<tr>
<td>1/1</td>
<td>0/0</td>
<td>For</td>
</tr>
<tr>
<td>22/22</td>
<td>0/0</td>
<td>FormalParameter</td>
</tr>
<tr>
<td>17/17</td>
<td>0/0</td>
<td>If</td>
</tr>
<tr>
<td>2/2</td>
<td>0/0</td>
<td>ImplementsRef</td>
</tr>
<tr>
<td>2/2</td>
<td>0/0</td>
<td>ImplementsRefEdge</td>
</tr>
<tr>
<td>3/3</td>
<td>0/0</td>
<td>Initializer</td>
</tr>
<tr>
<td>2/2</td>
<td>0/0</td>
<td>Interface</td>
</tr>
<tr>
<td>23/23</td>
<td>0/0</td>
<td>Method</td>
</tr>
<tr>
<td>63/63</td>
<td>0/0</td>
<td>MethodRef</td>
</tr>
<tr>
<td>63/10</td>
<td>0/53</td>
<td>MethodRefEdge</td>
</tr>
<tr>
<td>258/258</td>
<td>0/0</td>
<td>NotYetDefined</td>
</tr>
<tr>
<td>53/53</td>
<td>0/0</td>
<td>Operator</td>
</tr>
<tr>
<td>41/41</td>
<td>0/0</td>
<td>Reference</td>
</tr>
<tr>
<td>6/6</td>
<td>0/0</td>
<td>ReferenceEdge</td>
</tr>
<tr>
<td>0/1</td>
<td>0/0</td>
<td>Solution</td>
</tr>
<tr>
<td>1/1</td>
<td>0/0</td>
<td>Switch</td>
</tr>
<tr>
<td>165/165</td>
<td>0/0</td>
<td>TypeRef</td>
</tr>
<tr>
<td>165/23</td>
<td>0/142</td>
<td>TypeRefEdge</td>
</tr>
<tr>
<td>37/37</td>
<td>0/0</td>
<td>Variable</td>
</tr>
<tr>
<td>3/3</td>
<td>0/0</td>
<td>While</td>
</tr>
<tr>
<td>0/641</td>
<td>0/0</td>
<td>contains</td>
</tr>
<tr>
<td>0/3</td>
<td>0/0</td>
<td>containsConstructor</td>
</tr>
<tr>
<td>0/16</td>
<td>0/0</td>
<td>containsField</td>
</tr>
<tr>
<td>0/7</td>
<td>0/0</td>
<td>containsFile</td>
</tr>
<tr>
<td>0/22</td>
<td>0/0</td>
<td>containsFormalParameter</td>
</tr>
<tr>
<td>0/56</td>
<td>0/0</td>
<td>containsMember</td>
</tr>
<tr>
<td>0/23</td>
<td>0/0</td>
<td>containsMethod</td>
</tr>
<tr>
<td>0/23</td>
<td>0/0</td>
<td>containsStatement</td>
</tr>
<tr>
<td>0/16</td>
<td>0/0</td>
<td>containsType</td>
</tr>
<tr>
<td>0/37</td>
<td>0/0</td>
<td>containsVariable</td>
</tr>
</tbody>
</table>

### Graph overview

-------------

845 nodes
976 edges
2871 properties
Memory/time overview

4375 KByte heap memory used
4341 KByte heap memory used before resolving edges
1257 ms passed for extracting graph

Figure 7.1: integrated screenshot

Figure 7.2: local zoom
According to the above log, we can see the time and memory behavior meet the requirement defined in Chapter 4. In the screenshot, each shape stand for a kind of element; each colour of line stand for a kind of relation. We choose several kinds of elements to check whether our front-end succeed or not.

We choose the file, field, method nodes and methodRef relation. The number of them are 7, 16, 23 and 63 respectively. Compare with the log and graph, the first three number fit perfectly. The relation only showed 10 in the graph, because some are repetitive. But the number 63 match the number in log.

This test shows the positivity of our front-end. All the requirement are met basically.
8. Conclusion and Future Work

This last chapter draws conclusion of this thesis, and points out the directions of future work. First, in this conclusion section, we look back the whole thesis and give a result. In the future work section, we will give some ideas about how to improve current implementation of Recoder Front-end.

8.1 Conclusions

This thesis describes the design and implementation of Recoder front-end for VizzAnalyzer.

To solve the problem finding and integrating an alternative parser to analyze the Java programs for VizzAnalyzer, we explained the related theory, and give three major goals of this thesis and corresponding criteria for each goal used to check whether we have achieved the goal or not. The first goal is to find and evaluate a suitable Java front-end providing access to the Meta-Model according to Java language specification. This goal has been realized since we choose the Recoder to parse opened Java programs. Detailed description is stated in Chapter 3.

The second goal is to define mapping between Java specific front-end meta-model and Common Meta-Model. In Section 6.1, we give Recoder Meta-Model elements, Common Meta-Model elements and the mapping between them.

The third goal is to implement the previously defined mapping as a visitor for traversing the AST structure of a Java program. The incremental and iterative implementation of this step is achieved in Section 6.3.

As we can see, all goals stated in the introduction of this thesis have been successfully realized. We can make a conclusion that the Recoder front-end has been built successfully, thus solving the problem addressed by this thesis.

8.2 Future work

Our current implementation of Recoder Front-End provides us with the expected result and solves our problem in the first place. But there is still some room for improvement. First of all, this Recoder front-end is a plug-in of VizzAnalyzer. We should integrate them. Another thing needs to be taken into concern is that the Recoder Meta-Model needs to be extendable, evolved according to our research in metrics analysis. If a new metrics needs more information from Java Meta-Model, Recoder Meta-Model should be extended to provide more information.
References
