Master's Thesis

A Framework for Extracting Information from a Code Base

Love Johansson *D00* *& Johan Larsson* *MatNat*

Department of Computer Science
Lund Institute of Technology
Lund University, 2006

ISSN 1650-2884
LU-CS-EX: 2006-3

A framework for extracting information from a code base

Love Johansson
Johan Larsson

February 15, 2006

Abstract

This thesis examines the need for an information extraction tool which, when applied on a large software project, can aid the developers with information regarding the system structure of the actual implementation, as well as controlling architectural principles and guidelines.

The thesis has been written at the software architecture group of Ericsson AB in Lund, and the framework has been tailored for the needs of the architecture group in order to aid their work with maintaining the design integrity of the Ericsson mobile platform. The platform is written entirely in C for a real-time embedded system, and contains a large number of build variants. This poses a series of problems as the platform code must be processed by our information extraction tool without any form of pre-processing, macro expansions or modification, in order to guarantee that our tool can function on real production code.

Other problems not related to the structure of the code, are issues regarding the practical implementation of our tool prototype. What kind of information is important? Is it possible to extract the desired information? If so, how do we extract it? How should we store the extracted information? How do we use the stored data to provide the user with information that is related to a real problem? These are the kinds of questions that we have been working with when trying to solve the problems of the information extraction.

We have designed a fully functional tool framework prototype, using our conclusions from the research on both code related and non-code related problems. In the design of the prototype, we have aimed for ease of extensions, as new design problems arise all the time. During the implementation, additional functionality was added as more areas of applicability was discovered during discussions, which led to an extremely open structure in terms of parsing and output formats.

The prototype for the framework which we have developed handles information extraction from C code, in a non-preprocessed state, used in the mobile platform. The prototype handles parsing of the code, database facilities for storing parsed data, and a client-side application for producing relevant output based on the information stored in the database.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In large scale software development, a controlled architecture is necessary in order for the system to be manageable and extendable. Without specified architectural guidelines and rules, a large system would soon become impossible to maintain, and the structure of the system would depend on the individual developers. For this purpose, most companies today have architectural policies regarding their source code and documentation. However, these policies can be difficult to verify on a large code base, and often a way to extract information from the code to check how it really works is desired. This thesis will aim to find a way to simplify information extraction from source code, in order to be able to, among other things, verify architectural guidelines. The thesis is focused on the mobile platform software developed by Ericsson AB in Lund, and the architectural policies maintained by the *Software Architecture Group* (SwAG).

1.1 Problem Domain

At Ericsson AB, the Mobile Platform business unit (EMP) provides mobile communication companies with a hardware/software platform to be used in mobile units, such as mobile phones. The platform stretches from ASIC[1]-level component design, to a high-level programming interface called *Open Platform API* (OPA) covering all functionality the customers might need in order to incorporate the Ericsson platform into their product. The platform contains functionality covering everything from a Java VM to audio codecs and GSM/UMTS/EDGE signaling protocols. It is designed so that the same platform using different configuration variants should be able to be deployed in the customer's phones – there is no specific UMTS platform or a specific GSM platform, they are only different build variants of the same source. This leads to complex configuration dependencies which need to be managed.

The entire software platform is written in the C programming language, and contains a lot of seemingly complex solutions for handling real-time embedded execution, so the code itself is not trivial in any way, but rather difficult to understand at first. Finding information in the code can be a very time-consuming task, partly due to the sheer volume of the platform, and a lot of time can be

[1] Application Specific Integrated Circuit, a chip designed for a specific purpose (for instance data compression)

lost while looking for things that should be easy to find. Thus, trying to control architectural principles is, for the same reason, a task which is not undertaken lightly. The number of different build variants also poses a problem when trying to understand the code, as there are a countless number of configuration flags, which depend on each other, controlling large parts of the code.

For this purpose, this thesis will explore the possibilities for information extraction from the existing platform code and, using the extracted information, provide a tool for the developers as an aid in ensuring that their code behaves in such a way that the architectural guidelines and rules are followed, as well as producing high-level views of the system, enabling among other things tracking of dependencies on a system-wide level.

1.2 Method

Several meetings with the SwAG has been held, both for gathering information about the platform and the coding rules, and also for obtaining ideas and feedback on the prototype as it was being developed.

In this thesis, we will also cover the available research and tools in the field – do they all agree on a single solution, or are there tradeoffs to be made? If so, which proposed solution fits EMP's needs the most? Are there tools mentioned in research material that can be used to solve some of our problems, either as-is or slightly modified?

1.3 Goals of this Thesis

This thesis will describe our approach to the information extraction and architecture control problems, compared to the current research material and the tradeoffs that had to be made in order for a solution to work on the EMP platform. The result from this should be a prototype for extracting information, which can be extended and used by EMP as a way of obtaining information from the code and use it for various purposes.

The remainder of this thesis is divided into five major parts. First, a system overview, where a more detailed problem definition is presented, along with an overview of the Ericsson code and our solution. The following three chapters describe the three major parts of our information extraction tool – the parser system, the database system and the client system. Finally, the conclusions from the thesis work are presented.

Chapter 2

System Overview

The task of finding a way to extract design information from the mobile platform code is both a very large task, as well as a very vague one. In order to grasp the problem, we have performed an initial analysis of the problem. We have then decided to divide the problem, and our prototype, into smaller parts.

This chapter will discuss the problem at hand, and give an overview of the different parts of our system.

2.1 Problem

In order to create a tool that handles the problem of information extraction, we first needed to understand what was expected by such a tool, and what kinds of problems it was intended to solve. In conversations with SwAG, it became clear that no specified problem existed, but rather a mix of conceptions of what such a tool could be used for. Such conceptions could, for example, be tracing of signals, building dependency graphs, visualizing state machines, checking code-violations etc.

With this in mind, we tried to break down the main problem into smaller sub-problems, in order to identify the different parts of an information extraction tool.

- *Extracting relevant information from the source code* – What information is relevant? How do we extract the different design data from the code - a "raw" parsing of the code, or a more compiler-like approach using a lexer and a parser? How is EMP-specific data recognized? Can real-time information be gathered accurately without touching the code? Will a static analysis suffice, or should a dynamical analysis be considered – if so, how should this be performed without touching the code?

- *Storing the extracted data* – What is the most efficient way to store the design data so that the potential user can access it with ease? A tree-like structure, or perhaps a relation database? What are the pros and cons of the different approaches, and what fits the needs of EMP best?

- *Creating a user interface* - How should the available data be presented to the user? What levels of abstraction should the system be able to

handle? What is the user's role in our workflow - should the user just be able to configure the output settings, or take a more active role in designing the output from the data structure? How does the user interact with our system - a series of options in a user-friendly environment, or a script-like language, or a combination? What level of user input should be expected - general search criteria or code-level input? What kind of information output is expected? UML diagrams, XML trees, relation tables or something else?

As can be seen, the problem covers many questions that needs to be solved in order to create a prototype tool. These can be solved in many ways, and one consideration at this stage is how the prototype should be structured. The choice is between one monolithic system which handles everything, or a divided system where each part handles a specific task. We have chosen to focus on the latter, since it provides a flexibility in terms of extendability and small, replaceable subsystems. A few cornerstones for a working system was identified which gave us a list of potential subsystems.

- A parser subsystem, used to extract information from source code.
- A storage/database subsystem, which should be able to store arbitrary information extracted by the parser subsystem.
- A client subsystem, used to aid a user when working with the data stored.

Each of these subsystems can be designed and implemented independently, but require ways of communication between them. They each have their own difficulties and considerations that needs to be taken into account. Figure 2.1 shows an overview of our proposed prototype tool, which will be followed by a short presentation of the context for which the prototype has been written.

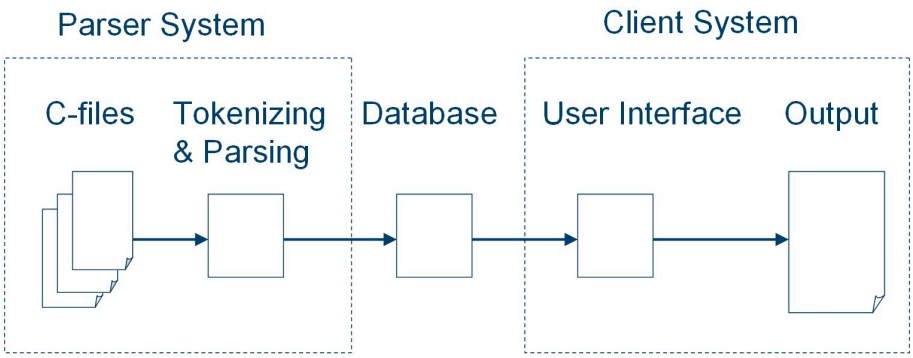

Figure 2.1: An overview of the system

2.2 Regarding the code

The source code for the mobile platform developed by Ericsson AB in Lund consists of several software modules. In turn, these software modules consist

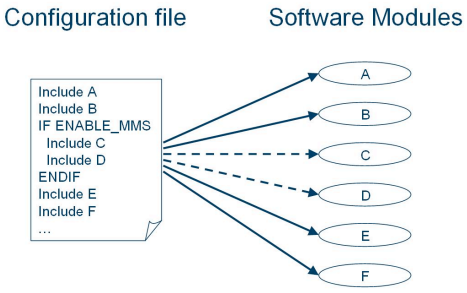

Figure 2.2: Variant management on a system level

Figure 2.3: Variant management on a file level

of several source files. The platform code is implemented mostly using the C language, and contains a real-time operating system for handling inter-process communication.

Since the software is designed to run on a number of different hardware platforms, the code contains configuration variables and flags which in turn controls large part of the software. For instance, a phone with hardware that doesn't handle MMS support, would have its platform software variant set up so that the software module (or modules) handling MMS would not be included on that particular build (figure 2.2). This, of course, leads to changes in modules which interact with the MMS module (figure 2.3), and so the number of code areas affected by each build variant increases, as does the number of configuration dependencies and flags. The variants are handled by a system of configuration files, which sets the configuration flags during build-time based on information given by an in-house developed build management tool.

Apart from the different build variant configuration flags and dependencies, a set of macros which handle OS-related tasks are used, which makes the OS communication interfaces simpler for the developers. Due to these design choices, the C preprocessing language is used heavily throughout the system.

2.2.1 The C preprocessor language

In C programming, some operations are handled by the C preprocessor. These operations are written in the C preprocessor language, and will be transformed into regular C code prior to the compilers parsing of the source file. According to [9], the C preprocessor has the following main functions:

- Inclusion of header files. The included files will be copied into the source file during preprocessing.
- Macro expansion. *Macros* can be defined, which are abbreviations for segments of C code. These abbreviations will be substituted for the code segments during preprocessing.
- Conditional compilation. Parts of the code can be included or excluded depending on various conditions.
- Line control. If a program is used to merge source files prior to compilation, the line control stores information on where each source line came from.
- Diagnostics. The preprocessor can detect problems at compile time and issue warnings.

2.3 Conclusions

As shown, the problems stated can be divided into three parts - a parser system, a database system and a client system. In the following chapters, we will discuss the three different subsystems of our prototype, with the considerations and choices made for each subsystem, starting with the parser system.

Chapter 3

The Parser System

This chapter will describe the parser system of our prototype tool. As a method of extracting information from source code is needed, different approaches to this problem will be presented and evaluated. The existence of and applicability of existing tools will be discussed, as well as the resulting design of the framework prototype. Finally, we will present our implementation, how it works, and a few examples of how to use it.

3.1 Analysis

In order to produce a tool which can extract information from source code, there is a need for understanding and defining the problems involved. In this chapter, we will focus on defining the information extraction problem, and present more detailed requirements for our tool.

Based on our initial problem description, the parser system of the tool should be able to do the following:

- Process the Ericsson Mobile Platform code.
- Gather information from the code.
- Store the information.

In order to perform these tasks, an analysis of existing methods and tools in this field must be done.

Extracting information from source code is a task with a large number of different solutions. First, we must decide whether we need a dynamic analysis of the system, or if a static analysis will suffice. A static analysis is performed by only observing system artifacts, such as source code, while a dynamic analysis is performed on a system during execution. In [5], a list of common approaches to information extraction is presented:

- Parsers.
- Abstract syntax tree (AST)-based analyzers.
- Lexical analysers.

- Profilers.
- Code instrumentation.
- Ad hoc.

Each of these methods approaches the problem in different ways. Parsers generate a representation of the source code, mostly for the purpose of generating machine code. AST-based analysers work in mostly the same way, but generates an explicit tree-like structure of the source code.

Lexical analysers process the source code as text, searching for specific text strings and patterns in the code in order to find the desired information. For instance, a lexical analyser could identify patterns such as

```
#include "time.h"
```

by matching it to a pattern like

```
<#> <include> <string>
```

One could also use ad hoc tools, such as perl, to match lexical patterns to specific occurrences in the source code.

These methods all work on a static level, producing static output. A profiler, on the other hand, works on a running system observing the inputs and outputs from the executing code without adding to or tampering with the source code. This method can be used to observe for instance the order of execution of different concurrent threads, or values of a specific memory address or function pointers. Code instrumentation works in a similar way, but allows the user to add for instance log prints to the code, and observe the output from these.

3.1.1 Tools

A number of tools designed for analysing and extracting information from source code exist. In this section a few of these, which have been evaluated for use in this thesis, will be presented. Though we may not have used some of them, mostly due to public availability and pricing issues, many of the ideas behind the programs have been very useful when designing our framework.

LSME

The *Lightweight Source Model Extraction* (LSME) tool [6] is a tool developed at the University of Washington for extracting source code information, such as call graphs and file dependencies. The tool aims for a *lightweight*, *flexible* and *tolerant* solution to source model extraction. Lightweight, as the specifications for the different extractors are small and easy to write, flexible, as there are few constraints on the information in the source code that can be extracted, and tolerant, as the code doesn't necessarily have to be compileable in order for the tool to work. The tool uses user-defined patterns and actions to be performed for each pattern to build a scanner, and then post-processes the output from the scanner into a source model.

The LSME tool is written in the Icon programming language, and its grammar and pattern definition files are also written in icon. It then uses an Icon runtime environment to generate a C based scanner.

As we will describe in 3.3, we aim to use Java 5 as the main language for constructing our prototype. Involving at least two more programming languages into the development is something which, in our opinion, should be avoided. However, the article presents a lot of useful ideas for information extraction.

DISCOVER

In [8], DISCOVER, a tool framework developed by Software Emancipation Technology for performing software management operations, like for instance reverse engineering, is presented. DISCOVER provides data gathering capabilities through the use of the Gnu compiler, which means that the extracted data contains everything which is processed by the compiler, and it also contains facilities for supporting profilers.

Even though the DISCOVER framework would have made parts of our system much easier to construct, the price tag is rather steep starting at $50.000 for a 10-seat license. Therefore, this tool has not been evaluated in practice during the prototype development.

JavaCC

When trying to construct a system which parses source code, the use of a compiler generator could easily reduce the amount of work that needs to be done. Java Compiler Compiler [11] is a widely used Java parser generator written in Java, which produces Java code. Using a specified grammar, JavaCC generates a lexer, which performs lexical analysis, and a parser which builds a representation of the system[1].

Since JavaCC is written in Java, and produces Java code, JavaCC would be a good candidate for an information extraction tool, or at least the construction thereof.

3.1.2 Static or dynamic analysis?

To make a proper information extractor, we must first decide if we want a static analysis of the code, a dynamic analysis of the code, or a combination of the two. Since the aim of our tool is to be flexible, facilities for combining both static and dynamic analysis would be valuable. Since adding or tampering with the production code is out of the question, a code instrumentation approach to the dynamic analysis was quickly discarded. As we would like to trace function calls among other things, a profiler would provide us with detailed valuable information on especially the concurrent parts of the system (like the real-time signaling), thus providing us with sufficient dynamic information which, in combination with a static analysis of the code, would provide a very detailed view of the mobile platform. However, due to memory constraints and a heavily optimized use of the mobile phone hardware, profiler facilities does not exist in the platform hardware. This leads to the conclusion that a dynamic analysis would not be either useful nor would it be necessary, as the primary target group for it never would use or extend such a facility. Also, a dynamic analysis would limit us to a single configuration, a design issue that will be discussed in 3.1.3.

[1] See [1] for more information on generating a compiler

3.1.3 Preprocess or not?

A question which arises on this low-level of information extraction is whether the code should be dealt with as it is structured in the code repository, or if the code should be preprocessed according to a specific variant configuration and then processed by our tool. Each path has its own advantages and disadvantages, described below:

Preprocessed code
Advantages

- Easy to parse (standard C language).
- Many C code parsers exist.
- More detailed function call tracing abilities, since macros are expanded.

Disadvantages

- Information loss in include file dependencies.
- Limited to one configuration.
- Code parsed in a state where no one will work with it.
- Not possible to parse configuration dependencies.
- Ability to parse macro calls for information is lost.

Non-preprocessed code
Advantages

- Code parsed as developer sees it.
- Not limited by different configurations.
- No information loss in include file dependencies.
- Possibility to parse configuration dependencies.
- Ability to parse macro calls for information.

Disadvantages

- Difficult to parse (due to preprocessor commands).
- Need for tool which handles preprocessor commands.
- Since there is no macro expansion, some information will be lost.

After reviewing the different approaches, our solution could go either way. One way is easier to implement, and more focused on a single configuration. This solution could rely on a series of existing code processing tools, and allow more time to be spent on other issues. The problem with loss of configuration dependencies could be somewhat fixed by combining information from different

build variants. The other solution is more difficult to implement, but keeps among other things configuration dependency information and macros. This solution would require more time to be focused on implementing a system for parsing the code. As we discussed the different approaches with the SwAG, more drawbacks to the first solution became apparent:

- The solution should be able to handle a lot of non-preprocessed information, such as include dependencies, as traceability in this field is desired.
- It is often easier to look for certain macros in order to obtain information, as the expanded macro code often is very difficult to understand.
- Due to the volume of the platform, about 65.000 possible build variants exists in theory[2].

When taking these issues into consideration, it becomes clear that although the solution in which we preprocess the code requires less time to implement, as it involves the use of existing tools, the amount of work required to make it handle the above stated problems would far exceed the amount of work involved to make a tool which can handle code which has not been preprocessed. In order to gather all the information required, a way to extract information from the non-preprocessed code would still be required, and the merging of about 65.000 different extractions to form a configuration-independent database does not seem like a sound task, partly due to the sheer size of all the extracted information from all variants. In light of this, our tool should be able to process code which has not been preprocessed as our main source of information extraction.

3.1.4 The tokenizer

The next problem is whether the source code should be parsed for information in its original shape, or if the code should be processed into an intermediate format to make it easier to understand for a parser. In the first approach, the source code would have to be parsed either manually, or by tools which utilizes some kind of regular expressions to match different patterns in the code. This approach would result in a very slow system, since it would rely heavily on string manipulation and comparison. In [6], the source code is tokenized whereafter a series of regular expressions is applied to the token stream in order for the information finding and extraction to take place. Another approach would be to lean towards a compiler-like system, where full parsing of the code is done to produce either an AST, or some other form of intermediate code. In either of these two cases, the source code is converted into tokens, and then processed. The tokenizing approach has several advantages. Mainly, it is faster, as token comparisons take place on an integer level, which is vastly faster than string comparison. Secondly, the expression patterns that would have to be written would operate on a token type level, which also reduces the required time to parse a file drastically – an identifier, for instance the name of a variable, would in a string comparison scenario be identified by ensuring that the string does not match a list of reserved words in the programming language, whereas in the token case, each token has its own type as an attribute. This has led to the decision that regardless of the parsing method, tokenization will have to be performed to ensure an easier and quicker parsing process.

[2] According to figures from an include file study performed by SwAG a few years ago

3.1.5 The parser

When we have extracted the token streams from all source files, it is time to parse these in order to extract the desired information. As stated in our analysis section, there are several ways of doing this. We have chosen to take a closer look at two of these:

- Build an AST from the tokens like most modern compilers, an approach supported by [11].
- Extract information by searching for specified patterns occurrences in the token stream, the approach used in [6].

Building an AST from our code is advantageous in many ways. Mainly, since the whole program structure is generated, finding information is simply a matter of traversing the tree until it is found. This approach would also give a higher abstraction level on our data extraction facilities – since one of the benefits of the tokenizer was to only have to tokenize the system once, the same argument goes for building an AST: once it is built, it wouldn't have to be rebuilt until the code changes. However, when trying to design this solution, a series of problems which hindered further development was found. The most severe problem involved the use of non-preprocessed code. Even though the preprocessor grammar was integrated easily into our tokenizer, building a structure from this information seemed to be a much harder task than initially expected. As the C preprocessor language allows macros for replacing any code and configuration flags for handling conditional compilation, there was no way of building a tree structure from the tokenized code, as shown here:

```
Example 1:
#define RETURN return 0;}
...
int main(){
  printf("Hello World");
  RETURN

Example 2:
...
int main(){
  #IFDEF A
  printf("A: Hello World");
  return 0;
  }
  #ELSE
  printf("Hello World");
  return 0;
  }
  #ENDIF
```

Without the macro expansion, the parser would not be able to build a proper AST from the code shown in example 1, as the *main*-method is never closed. In example 2, the opposite problem is found. Here, conditional compilation flags

(in this case A) generate a situation where the *main*-method is closed twice. A worst-case scenario would be with these two techniques combined, as it would rule out any possibility of building a structure from the code.

Another problem which surfaced when trying to adapt the AST-solution to the mobile platform code is that some information is generated during the system build process. This leads to a lack of information regarding for instance files to be included or other definitions, which in turn leaves holes in the structure making the AST generator fail.

The other approach to parsing which we have evaluated is by searching for patterns in the token stream which define different language constructs. We have solved this by writing a small language for finding and evaluating tokens which match a certain regular expression. The grammar is defined using constants generated by JavaCC. The layout of the parser is such that our main parser only invokes a kind of parser plugins, called parse units (see 3.2.3 for more info) which in turn parses the token stream looking for certain expressions. After finding such an expression, it then submits the relevant information to a database interface, depending on what kind of information storage solution is used.

We have chosen this approach, as it gives the user very good possibilities to extend the parser. For example, if the user wants to parse the code for a specific sequence of tokens, he would only have to write a parse unit which matches that specific sequence and then add the unit to the list of units which the parser invokes on runtime. Since the database interface is replacable without changing the parser system, several database solutions can be considered. This makes our prototype very extendable, which is one of our main goals.

Regular expressions

The use of regular expressions for finding patterns in source code has previously been successfully investigated and implemented in [6]. The approach has the advantage of being flexible and easy to extend, but has some drawbacks compared to an AST approach. It is especially hard when one needs to, for example, backtrack to where a variable found in the pattern is assigned or similar matters. Using an AST-based parser would simplify some of these issues since everything, such as assignments, is stored in the tree, but would instead be very specific and hard to use for anything else but for the language it was designed for. Building ASTs from a large system is both time and memory consuming, which is also a big drawback of this approach.

All in all, our conclusions from this is that it would be desirable to have a fast, flexible and powerful regular expression parser, with the option to build simple AST-structures from the information extracted by it, in order to manage some of the backtracking problems.

3.1.6 Problem definition

After reviewing this information, the different approaches to information extraction, existing tools, and a basic understanding of the system which should be processed, a more detailed definition of the problem and the tasks to be performed by our tool is made clearer.

The problem is to develop a tool which can perform the following tasks:

- Process code written in the C programming language, which has not been preprocessed. The code contains a lot of configuration dependency flags, as well as an extensive use of macros.
- Tokenize the code.
- Perform a static analysis of the token stream by using patterns to find information.
- Submit the parsed information to a database interface, which stores the data.

3.2 Design

The main goal of the parser system is to handle all processing and parsing of the code. As we have seen, there are a lot of different approaches to this task. In this section, we will discuss the design of the parser system based on the analysis results and the structure of the hardware and software platform.

3.2.1 Tokenization

According to the design decisions made so far, the base part of our system should perform a static analysis on non-preprocessed code and build tokens from the source code for further parsing. This resembles the function of a lexer used in a compiler.

Processing source code and building tokens is a task performed by various tools, for example JavaCC[11]. As this is one of the largest available Java parser generators, chances are big that a JavaCC-generated parser would aid us in the development of the tokenizer by easily providing such facilities. However, one problem still remains – the need for a tokenizer which handles non-preprocessed code. For this, a grammar input for JavaCC which handles C code interlaced with C preprocessor directives is needed. As JavaCC is largely meant for compiler construction, the C grammar files for JavaCC which we have found all make the assumption that the code has been through the preprocess step of the compilation, and the resulting code doesn't contain any preprocessor information.

Since writing our own grammar for the C language based on the language definition extended to handle preprocessor directives is a very large task, a decision was made to extend an existing C grammar file for JavaCC with the information required for tokenizing a source file containing both C code and preprocessor directives. This will then be used to generate a lexer which tokenizes the source files into streams of tokens (figure 3.1).

3.2.2 Storing tokens

After tokenization, the stream of tokens is stored in a list. In order for many files to be tokenized at the same time, one list is produced for each source file (figure 3.2). For storage, each list is stored in a hash table using the filename as the hash value. This can be done as each filename in the platform software must be unique. This table is then saved on the disk. The reason for doing this,

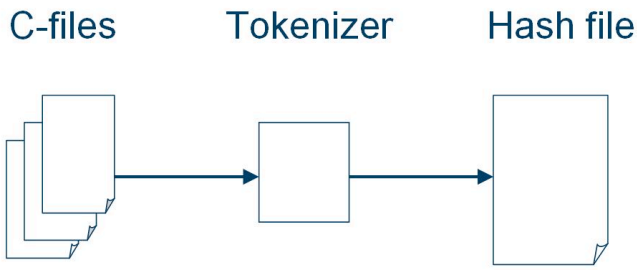

Figure 3.1: The tokenizer section of the base system

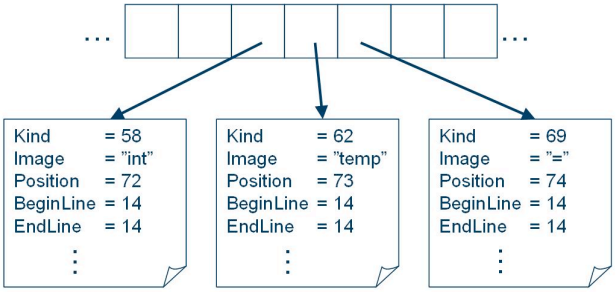

Figure 3.2: Tokens stored in a list.

is because the system only should have to be tokenized once, reducing the time needed if multiple parsings is necessary. The use of this technique also opens for different kinds of parsers, which could then operate directly on the generated token stream without having to tokenize the system for each parsing technique.

3.2.3 Parse units

In order to make an extendable parser system, the parsing is divided into smaller units which have their own area of responsibility. A parse unit, in its most abstract form, is a program that has as only task to extract information out of some input and send this information to a database interface (described in chapter 4.2.2), as shown in figure 3.3. In our prototype, the input is a tokenized C code or header file, but a parse unit is designed to be able to work on any type of input. Each parse unit can also be invoked separately in order to only extract the information wanted and to reduce time spent on parsing.

3.2.4 The parser system model

In this section, we will describe the differences between our parser system, and "traditional" parser systems like a JavaCC generated parser.

Normally when generating a parser, the purpose is to perform a lexical and syntactical analysis, in order to build a complete system model. In JavaCC, this is done by first reading a grammar file, which defines the lexer and parser

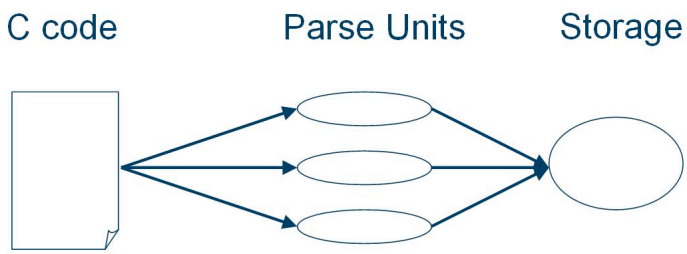

Figure 3.3: The parser section of the base system

grammar (see figure 3.4). A lexer and a parser is generated, and these elements form the front-end of the parsing system. Using JJTree, a JavaCC add-on, the parser output is then processed, and an AST covering the complete parsed system is generated.

Afterwards, the AST is traversed by using visitors, which perform a semantical analysis to extract information from each node in the tree. The AST traversal process by the visitors constitute the back end of the system.

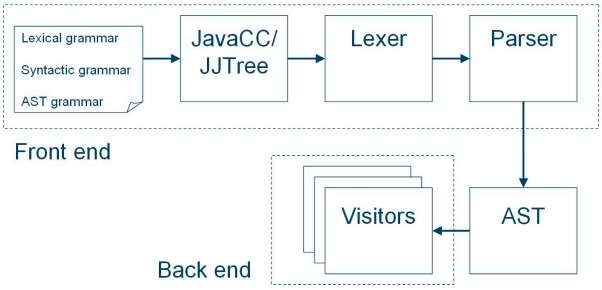

Figure 3.4: The JavaCC parsing model

In our case, we only want to perform a lexical analysis, i.e. tokenize the system. In our grammar file, only information regarding the lexer part is described. Using JavaCC, we then generate a lexer which tokenizes the platform code base, and stores this in token streams for each file that is tokenized (see figure 3.5). This constitutes our front-end equivalent of the JavaCC system. So far, there are not many differences, except that a syntactic analysis have not been performed.

Using these token streams, we then invoke our parse units to find specific token sequences (see 3.3.2). This is the equivalent of the JavaCC visitor back-end, with a few important differences:

- The code does not have to be syntactically correct, since our system only looks for specific patterns.
- Each parse unit contains only the necessary grammar for finding different language constructs in the code.

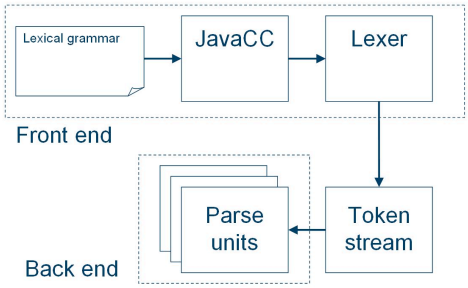

Figure 3.5: Our parsing model

3.2.5 User interface

The user interface of the parser system consist of two parts – one where decisions on which code to tokenize, and one where the user can define which parse units to invoke as well as which database transport interface is to be run(figure 3.6). This lets the user control the whole parsing and extracting process, from file level to which database format the information should be stored on.

The tool also contain a console, where information on the progress is printed (figure 3.7).

The user is also able to configure the tokenization in regards as to which file types it should process and tokenize, and if there are files containing errors that should be omitted from the tokenization.

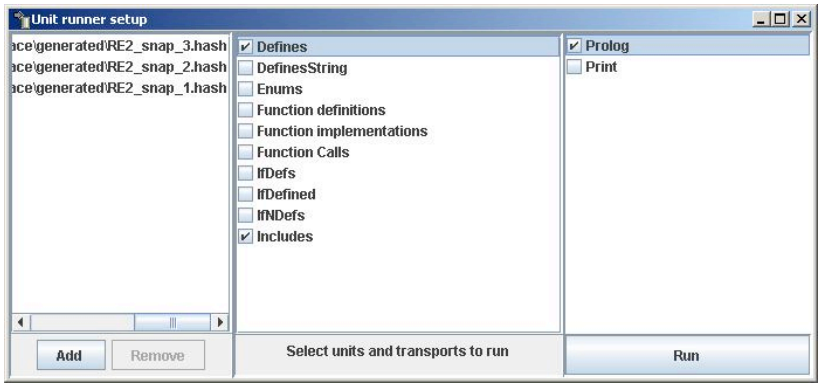

Figure 3.6: The parser system unit runner

3.3 Implementation

The parser system of our framework has been implemented in Java, using the eclipse platform. One reason for choosing Java, is because a lot of the tools developed at EMP is developed in Java. We have used Java 5, since it provides

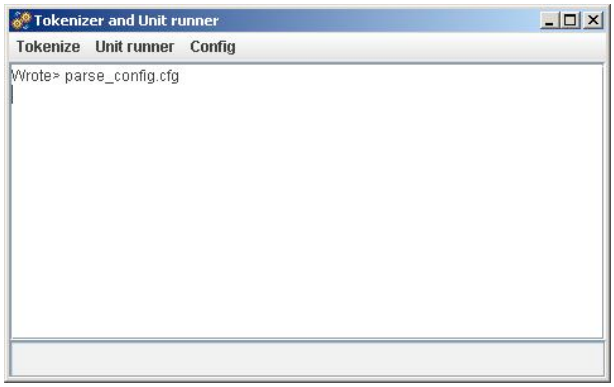

Figure 3.7: The parser system GUI

a lot of useful additions to the Java language[3], for instance generics and the possibility for having variable length argument lists, which has made the implementation of the system easier to both write and understand. In this section we will describe considerations and choices taken during the implementation phase.

3.3.1 Tokenizer results

After experimenting with different ways of writing Java objects to disk, and structuring the stored tokens, a series of tests were performed on a subset of the code base. A few ideas which surfaced during the implementation were tested, and compared to each other. The metrics presented in table 3.1 show the results for our tokenizer in three different stages:

- Its first fully functional state, using the Serializable-interface to implement disk writing and reading.
- A state in which all disk writing was rewritten using the Externalizable-interface.
- A state in which only simpler data structures was implemented using the Externalizable-interface and the rest using the Serializable-interface.

Stage	Time units	Size units
Serializable	100	100
Externalizable	115	76
Both	79	76

Table 3.1: Disk writing tests

As shown in the tests, we managed to save up to 21% in tokenization time using a combination of both the Externalizable-interface and the Serializable-interface which, when applied to a system-wide tokenization, means up to 5 minutes.

[3]For more information about Java 5, see [10]

Another experiment was to change the way the tokens store their value. Table 3.2 shows the results for this experiments, with the tokenizer in two different stages, each based on the most effective disk writing stage from above:

- A state in which the tokenizer stores the kind (for instance identifier, integer literal, plus) and image (for instance "temp", "7", "+") for each token in the tokens.
- A state in which the kind is stored in the token, but each unique image string is stored in a table, and only a reference to a position in the table is stored in the individual tokens.

Stage	Time units	Size units
Store string	100	100
Store reference	82	96

Table 3.2: Token structure tests

According to the test results, by using the second approach we manage to save additionally 18% of the total time used for tokenizing, which leads to a total of 35% saved.

3.3.2 Java parser grammar

The tokenizing of code gives us, in practice, a token stream where each token contains information about kind, for example `<integer>`, image, for example "myMethod", location, for example start line, and its position in the token stream list. The most simple form of using our parser grammar is to search for lists of tokens of various kinds. For example, the pattern used to search for "int myInt = 10;" and similar assignment constructs could look like:

```
Match assignment = new MatchList(INT, IDENTIFIER, ASSIGN,
                                 INTEGER_LITERAL, SEMICOLON);
```

This simple sequence would supply our parse unit with five tokens each containing the above described information (kind, image, location and token stream position). Since our pattern matching language is designed in Java, adding new functionality is done by defining a new Java class that inherits most of its structure. These classes handle pattern matching operations on the tokenstream and will hereby be called operation classes. For instance, we have created a class "Or", that allows us to do more advanced pattern matching A pattern for matching more general assignment constructs, such as "float myFloat = 10.3;" or "int myInt = 30;" could look like this:

```
Match assignment = new MatchList(new Or(INT, FLOAT, SHORT),
                                 IDENTIFIER, ASSIGN,
                                 new Or(FLOAT_LITERAL,
                                 INTEGER_LITERAL), SEMICOLON);
```

This could be rewritten in order to minimize recurrence for other similar patterns, for example:

```
Match TYPE = new Or(INT, FLOAT, SHORT);
Match INT_OR_FLOAT_LITERAL = new Or(FLOAT_LITERAL,
                                    INTEGER_LITERAL);
```

This enables us to rewrite the assignment sequence as:

```
Match assignment = new MatchList(TYPE, IDENTIFIER, ASSIGN,
                                 INT_OR_FLOAT_LITERAL,
                                 SEMICOLON);
```

In order to extract more complex information, such as functions etc, more advanced operation classes are required. Using the "Object..." construct in Java5, the Or and MatchList has a variable number of arguments to provide greater flexibility. The following list shows a few that have been implemented and found useful (also shown in figure 3.8).

- *MatchList* – A list class, which contains a series of tokens or other operation classes. This is used if an expression contains more than a single token, which is very often.
- *Interval* – An operation class initiated with a start token and an end token, that will create a list with every token found between those two tokens. The interval can also be set to handle nested intervals, for instance while-loops inside while-loops.
- *Or* – An operation class initiated with tokens or other operation classes. The matched sequence will correspond to one of them (shortest sequence first).
- *Optional* – An operation class that can be used where optional tokens exist.
- *Not* – An operation class used to make sure that a sequence or token is not part of the wanted sequence.
- *Rule* – An operation class which can be customized to match against the image, location or position of a token.

Since all tokens know their place in the token stream, it is also possible to do look-ahead (or look back) in the token stream. Another feature found useful is the ability to name the results found by the operation classes, as will be shown in the next example.

By combining the described operation classes, we can construct advance pattern matching structures. For example, consider the problem of finding function implementations within the code. The pattern used for searching for these could look something like this:

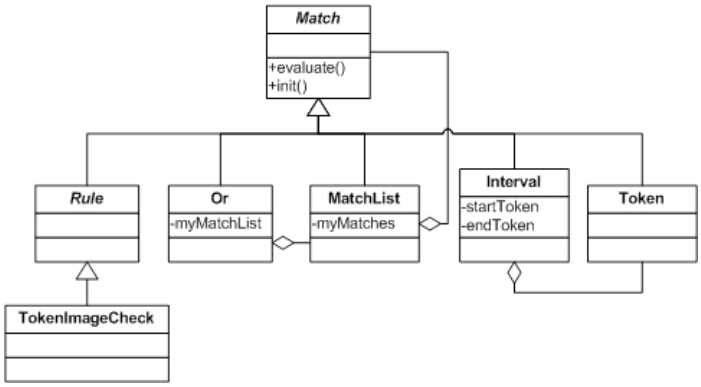

Figure 3.8: The Java parser grammar classes.

```
Match function = new MatchList(TYPE.setName("type"),
                               IDENTIFIER.setName("function_id"),
                               new Interval(LEFT_PAREN,
                               RIGHT_PAREN).setName("parameters"),
                               new Interval(LEFT_BRACKET,
                               RIGHT_BRACKET).setName("body"));
```

Now, if we apply this patten to a section of a C-program.

```
...
    return 1;
}

int myFunction(int parameter1, int parameter2)
{
    return parameter1 + parameter2;
}

int main()
{
...
```

Or, as seen by our parser after tokenizing.

```
...,<return>,<integer_literal>,<";">,<"}">,<int>,<identifier>,
<"(">,<int>,<identifer>,<",">,<int>,<identifer>,<")">,<"{">,<return>,<identifer>,
<plus>,<identifer>,<";">,<"}">,<int>,<identifer>,<"(">,<")">,<"{">,...
```

The parser would yield the result shown in figure 3.9.

As can be seen in figure 3.9, each part corresponds to the operation class that was used for parsing it, and the naming of each operation class result makes the parts accessible. If we want to, we could apply another search pattern to one of the parts. For example, if we are building a ”function call function”-graph, one

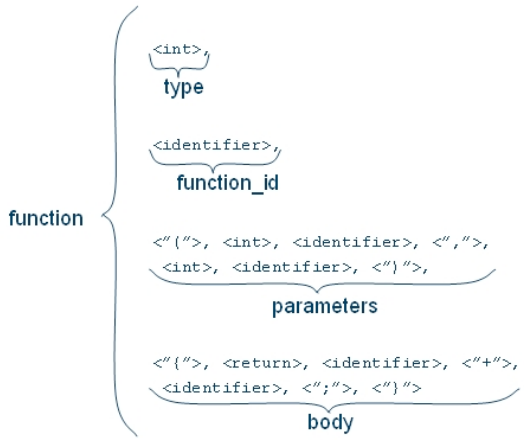

Figure 3.9: Tokens parsed and arranged into parts.

could apply a "function call"-pattern to the body part of a function. In general, by doing this kind of top-down iterative parsing, the result will be more refined and contain less errors.

3.3.3 Building simple ASTs

As said earlier, ASTs can simplify the task of backtracking if we, for example, want to know where a variable, found in a pattern, was last assigned a value. For this reason, some time was spent on trying to order extracted information, parsed with different search patterns, into an structure that resembles an AST, but without many of the details (often unnecessary for design extraction) one get when generating ASTs used to compile a program. For example, consider the following simple program:

```
int A(int var1)
{
    int a_var = var1 + var1;
    return a_var;
}

void B()
{
    int b_var = A(10);
}
```

One way of creating an AST-similar structure is to first parse all the functions, then, for each function, parse parameters, declarations, assignments and return statements. This would yield a model that would look like figure 3.10. However, if we are only interested in, for example, function calls made from a

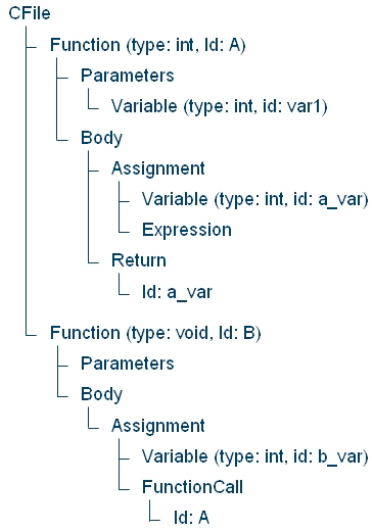

Figure 3.10: A detailed model

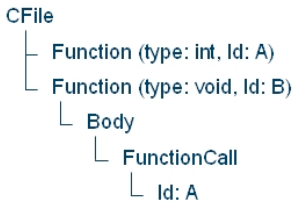

Figure 3.11: A reduced model

function, the model created in figure 3.11 would be preferable, since it, compared to figure 3.10, contains fewer details and ignores that the function call made from function B is part of an assignment.

Creating these kind of models can greatly aid extraction when there is a need for backtracking or getting a more abstract and manageable view of data.

3.4 Conclusions

The objectives for this part of the thesis were to analyse how a system which processes source code and extracts information from it should be constructed. The tool was to be constructed in such a way that it primarily focuses on processing the source code used in the mobile platform.

A dynamic analysis was ruled out early, due to the inability to make additions to the source code, and the lack of a profiler on the mobile hardware, so a static analysis approach proved to be the only reasonable solution to the method of analysis problem. When deciding whether the source code should be analysed before or after preprocessing, the preprocessing-approach was discarded due to the loss of configuration dependent information which can be obtained by analysing the code in a non-preprocessed state.

The processing of the code takes place by first tokenizing the source code, and storing the source code tokens in a file. By doing this, the system only needs to be tokenized once. Since the tokenizer which can handle non-preprocessed C code also contains the grammar for parsing regular C code, the ability for parsing preprocessed C code exists, and no modification to the system will have to be made for this to function.

After the tokenizing, the different parse units finds specified occurrences of patterns in the token stream. This leads to a system which is easy to extend, as the parse units are very small and easy to implement.

The transport units then produce arbitrary output based on which database interface that has been used. This leads to a open structure regarding the output format, if the end user should desire a different information storage approach than the one that will be described in chapter 4.

3.4.1 Future work

Several improvements can be made to the parser system. The main reason for not implementing these improvements in the first place, is time constraints. Due to the limited time for writing this thesis, a lot of ideas were left unfinished. In this section, we will describe a few of these.

The regular expression language could be extended. In its current state, it can handle basic operations, but could be extended to include more advanced features.

Another feature which could generate several more areas of use for the parser system is the possibility for supporting several grammars. Instead of just one C grammar tokenizer, perhaps a solution where the user could choose between different grammars depending on what types of files that are to be parsed is wanted.

Chapter 4

The Database System

After having parsed the platform software for different kinds of information, naturally an efficient way to store this information is needed. In this chapter, different approaches to data storing will be examined, along with existing solutions for doing this. Afterwards, the design of our data storage system will be presented, as well as the implementation and conclusions drawn.

4.1 Analysis

In order to decide which database system to use, a list of requirements for the data storage part of our tool must be generated. Based on the research, as well as wishes from the SwAG, the following list of requirements has been gathered:

- The system shall store relations between code elements.
- The system shall provide support for user interaction, either through an external scripting language, or integrated in the database system.

When storing data, there are several different approaches. One approach considered in the early stages of the analysis, was to store the extracted information in an object oriented graph-like structure (figure 4.1). Each node in the graph would describe a source code element, which would in turn contain references to other nodes used by that element. This would make the tracing of data through the system an easy task.

This data structure would probably work nicely on a smaller system, but after realizing the amount of memory required to store such a structure describing thousands of files containing a countless number of functions, this solution was considered to be unrealistic.

An approach used by [3] and [5] is to store source code elements in a relation database. This database would contain about the same information as the graph solution, but ordered in a relation table instead of a graph. This is described in table 4.1.

Regarding memory usage for storing the data structure, this solution seems to fit the needs of the database system of our tool nicely.

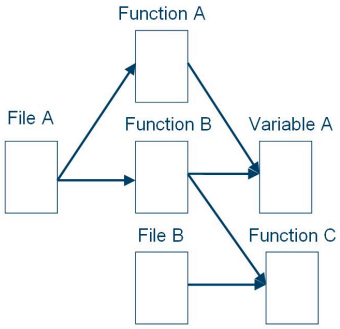

Figure 4.1: A graph representation of data storage

Source Element	Relation	Target Element
File	includes	File
File	contains	Function
File	defines_var	Variable
Function	calls	Function
Function	access_read	Variable
Function	access_write	Variable

Table 4.1: A relation table approach

4.1.1 Tools

So, how does the tools developed for the purpose of extracting information from source code handle the storing of data? In this section, a few existing tools and their solutions for data storage will be presented. In general, these tools will use either a SQL based database, such as in figure 4.2, or a database structure based on Prolog[1], such as in figure 4.3. The DISCOVER tool mentioned in 3.1.1, is a closed system with proprietary data management facilities, and is therefore not mentioned here since its inner workings are unknown.

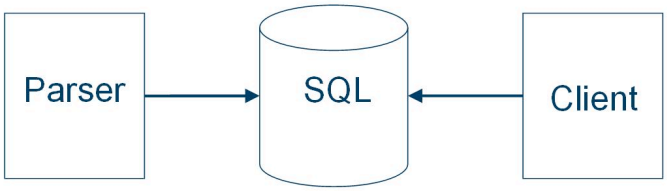

Figure 4.2: A SQL database system

[1] see chapter 4.2.1

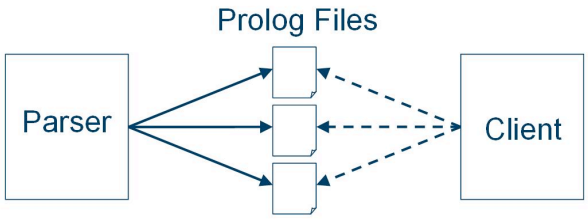

Figure 4.3: A Prolog database system

Dali

The Dali tool [3] is a reverse engineering workbench which uses a variety of tools in order to build a complete view of a system. For data storage, it first parses the source code into a format called Rigi[2] Standard Format, or RSF. A perl wrapper then handles communication between the Rigi system and a PostgreSQL database, and stores the information in a structure like that in table 4.1.

Dali is a tool which is maintained by the Software Engineering Institute at Carnegie Mellon University. From what we have gathered, use of this tool requires a representative of the institute who comes to your location in person, and who uses the tool for you in order to perform an architecture reconstruction. Since this is a somewhat inconvenient solution, we will not evaluate this tool further. We will, however, evaluate the concept of a SQL based data storage.

Riva

Riva, described in [7], is based on extracting design information from source code and loading it into a Prolog database. It uses Prolog to generate abstractions from the code, and then visualizes these using Rigi.

X-RAY

X-RAY, also mentioned in [7], is a tool designed for recovering the architecture of distributed software systems. X-RAY is implemented in a Prolog environment, and extracted information is represented as Prolog facts. Searching, pattern design and operations on the extracted information are implemented as Prolog predicates.

Both Riva and X-RAY use a Prolog based data storage solution. Unfortunately, they seem to have been discontinued, as there are no traces of them mentioned since circa 1998. Even though the tools might not exist anymore, the possibility of using Prolog will be evaluated.

4.1.2 Choosing a data storage approach

After having reviewed the different approaches mentioned to handle data storage, a decision must be made regarding which storage approach that should be used in our tool. Should it be a SQL relation database, or a database based on

[2]Rigi [13] is a tool developed for visualizing legacy systems, see 5.1.1

Prolog facts? In order to make this decision, the advantages and disadvantages of both systems must be compared to see which fits our solution best:

SQL
Advantages

- Fast, widely used database system.
- Good interface language.

Disadvantages

- Large back-end for handling databases.
- Portability (not easy to move databases).

Prolog
Advantages

- Very powerful function language.
- Small back-end for handling operations on relations.
- Easy to work with only a subset of files.
- Portability (database storage in text files).

Disadvantages

- Not as fast system as SQL
- Files must be compiled into the Prolog system.

As we can see, both systems have their advantages and disadvantages. Using a SQL database would allow fast access to the databases, at the expense of functionality in the language, while Prolog would provide this functionality, at the expense of speed. To make a decision, the future usage of our tool was examined. Would developers use it to such an extent that a central database, such as an SQL server, would be needed? What kind of facilities in the user interface language is needed? At this point, we chose to sacrifice speed for language capabilities and chose Prolog as our main data storage system. This decision was made based on the assumption that any future user would use the tool not for finding bugs or casual browsing of the system properties, but for performing somewhat complicated operations trying to analyse structures of the system and not do this very often (as in several times per day). This would allow a system which lacks in data accessing techniques and speed, but makes up for this by providing a very powerful interface language. This solution also allows the user to experiment with the databases, since it is easy to just copy the text files which the database consists of, and modify them.

4.2 Design

This chapter will present the design of our database system. As seen in 4.1.2, this design will focus on a Prolog based data storage system. In this chapter, we will describe the Prolog programming language, as well as an example of how prolog queries can be used to solve an information extraction problem.

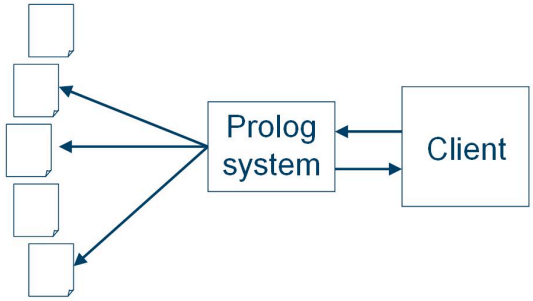

Figure 4.4: Loading of necessary files

4.2.1 Prolog

Prolog (Programmation logique) is a language used for computing logical expressions. A Prolog database work with facts and rules defined directly in the program, or external files. A Prolog fact (or predicate) has the form head(argument1, argument2, ...), where head is the name of the relation, and the arguments are the parameters of the relation[3].

Structure

The Prolog system operates on a set of files containing facts or rules, which is then compiled into the prolog interpreter in order for them to be accessible to the user. Since the entire database is not necessary for each user operation, we have designed the database interface to only load the parts of the database which is necessary for each task. The choice on which files to load are made in the client system, which will be described in chapter 5. Figure 4.4 shows an overview on how prolog accesses files.

Applicability example

Prolog provides a powerful language for processing data. This has been very useful when constructing a set of functions to generate abstractions of the system. To show how this language can be useful, imagine that we have the following set of facts:

```
include('file1','file2').
include('file1','file3').
include('file1','file4').
include('file2','file5').

functiondefinition('file5', 'hello').

functionimplementation('file1', 'hello').
```

File1 includes the files *file2*, *file3* and *file4*, while *file2* includes *file5*. *File5* contains the definition of the function *hello*, which is implemented in *file1*. A

[3]For more information on Prolog, see [15] or [2]

rule which traces a function implementation from a function definition could look something like this:

```
traceImplFromDef(File, Function, Resulting_file):-
  functiondefinition(File, Function),
  traceimpl(File, Function, Resulting_file).

traceimpl(File, Function, Resulting_file):-
  functionimplementation(File, Function),
  functionimplementation(Resulting_file, Function),
  Resulting_file==File.

traceimpl(File, Function, Resulting_file):-
  include(X, File),
  traceimpl(X, Function, Resulting_file).
```

The rule traceImplFromDef has three parameters, the file in which the function is defined, the function to trace, and the file in which the function is implemented. The rule takes each *Function* defined in *File* and checks the *traceimpl*-rule on each of them. The first instance of the *traceimpl* rule checks whether the function is implemented in the current file. If not, the second instance of the rule runs *traceimpl* recursively on every file which includes *File*. A call to *traceImplFormDef* could look like this:

```
 ?- traceImplFromDef('fil5', 'hello', Implemented_in).

    Implemented_in = fil1
    yes.

?- traceImplFromDef(File, Function, Implemented_in).

    File = fil5
    Function = hello
    Implemented_in = fil1
    yes.
```

4.2.2 Transport interface

In order to transport information from the parser to the database system, the parser needs some kind of transportation interface. For this purpose, we have designed the transport unit. A transport units task is to take the information sent from a parse unit (chapter 3.2.3), process it if needed, and then store it in some format or display it to a user. The use of transport units simplifies the task of switching between different output formats without having to change anything in the parse units. For instance, in our system two transport units have been designed: one which prints the output without storing it, and one which generates a Prolog file containing the parsed relations. If the user desires an alternate form of output, a new transport unit needs to be constructed, but nothing in the parse units need changing.

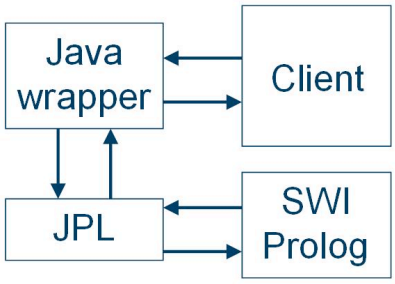

Figure 4.5: The database system implementation

4.3 Implementation

Implementation of the database system required an interface to be made between both the parser system and Prolog, as well as between the client system and Prolog. A Prolog transport unit was constructed (see 4.2.2) in order to store information from the different parse units on a Prolog format. For Prolog communication, several interpreter systems are available. For this task, we chose SWI Prolog, since it provides a fully functional Java interface.

4.3.1 JPL and SWI Prolog

SWI Prolog [15] is a popular freeware Prolog interpreter licensed under the LGPL[4], which focuses on fast compiling, small size[5] and scalability to handle large data structures.

JPL is a library using the SWI-Prolog foreign interface and the Java JNI[6] interface providing a bidirectional interface between Java and Prolog that can be used to embed Prolog in Java as well as for embedding Java in Prolog. We have used this to write a Java wrapper which handles all communication between the client system and the Prolog database. This is shown in figure 4.5.

4.4 Conclusions

The objective for this part of the thesis was to construct a database system which stores the data gathered from the parser system. The system should handle both storing of data, as well as accessing it and performing operations using it.

A graph-like storage solution was ruled out, due to the resulting size of the structure. As the research done in the field suggest storing the data in a relation table, this approach was examined. A Prolog based database system was chosen, due to a wider range of functionality in the language. As the system probably won't be used in time-critical situations, the loss of the speed benefits from SQL systems were considered acceptable.

[4]Lesser GNU Public License, `http://www.gnu.org/copyleft/lesser.html`
[5]About 15MB on disk
[6]Java Native Interface, an interface for native programs through Java

For populating the database, a Prolog transport unit stores the data in a format readable by the Prolog interpreter. The interpreter then compiles the database files, and operations on the database can be performed.

SWI Prolog was chosen as our Prolog interpreter, since it provides a Java interface. This interface is used to process queries from the client system and returning information from these queries back to the client. How this is done will be described in more detail in chapter 5.

4.4.1 Future work

For the database section, there is not that much future work that can be done. Perhaps an evaluation of other Prolog interpreters and Java interfaces can be done, since we only picked one that seemed to suit us.

Chapter 5

The Client System

After having constructed a system for storing data collected from the source code, a way to present and process this information must be constructed. In this chapter, we will discuss some of the approaches used by existing tools, as well as the solution we have implemented, and the motivation for choosing this approach. Finally, conclusions drawn from the client system work will be presented.

5.1 Analysis

In order to create a system which presents the collected data, we must first analyse how such a system shall behave. This chapter will analyse how a client system designed for this kind of problem can be constructed. In 1.2, a number of considerations regarding the client system are stated: How the data should be presented (the output from the system), which abstraction level the tool should provide, and which user input that could be expected. These considerations will now be analysed.

The first consideration is about what level of abstraction the system should be able to handle. A difficult question, as the desired level of the system depends on which information is desired. If the user wants to know for instance include dependencies, the system should handle such information on a file level. However, if the user wants dependencies between software modules, the system should process this on a module-level. While a system which could process all kinds of information on every imaginable level would be nice to have, it would also require a substantional amount of work in order to function. However, limiting the system to only a few modes of processing information would cripple the open and adaptable system we desire to create.

The second consideration is about how the system should output processed data. The first way that comes to mind is to have the system display the results directly on the screen. This solution would require a way to visualize different components, and possible relations between them. This problem is related to that of the level of abstraction – as more levels of abstraction is needed, as is a way to visualize them, which in turn increases the amount of work required for a flexible system to function. Another approach is to have the system produce other forms of output. Perhaps producing output on a format

readable by existing tools would reduce the amount of work needed to maintain an open and flexible system? This would, however, make the system dependent on external tools in order to function properly.

The final consideration is about how the user should interact with the system. Since we have a Prolog based database system, allowing the system to accept Prolog queries as input would be desirable. Since one of the motivations for choosing this language is its powerful query capabilities (described in chapter 4.2.1), it would seem natural that the client system utilizes these facilities. Another question is the amount of interactivity of the system. Should we limit the user to query operations, or should the user be allowed to experiment further with the database or a copy thereof? If the solution in which the system produces output readable by external tools is considered, perhaps the external tool can provide interactivity in regards to experimenting with different re-structurings of the database elements. This would allow for some interactivity in cases where it is needed, and a lack of it in cases where interactivity is not desired.

From this analysis, a requirement list can be constructed. The client system should be able to:

- present information on different levels of abstraction.
- access the prolog database system.
- produce output readable by other tools, for instance visualization tools.

5.1.1 Tools

In order to make a decision, we must first analyse existing tools, and see what kinds of facilities are available for visualizing and processing extracted information. In the following section, different tools whose specialty is displaying information regarding a software system will be described.

Lattix LDM

Lattix LDM[12] is a tool created by Lattix inc. which specializes in visualization of the design model for a Java system in a *Lightweight Dependency Model* (LDM). This gives the user quick feedback about the architecture of the system, as well as tools for an eventual restructuring of the system if it turns out that it is badly designed.

The LDM can be described as a matrix, where all classes are shown on both the x and y axis, and dependencies are visualized by numbers in the matrix (for instance, if the number "2" appears on the intersection element of *Test1.java* and *Test2.java*, it means that *Test1.java* uses two resources in *Test2.java*). The LDM also shows the package hierarchy. This in combination with the ability to expand and collapse the packages gives a good overview of the system at any level. Lattix inc. has also provided us with a plugin for reading XML-files, which means that we can specify output from our system to produce output readable by Lattix LDM.

When dealing with a very large system, such as the Ericsson platform, Lattix

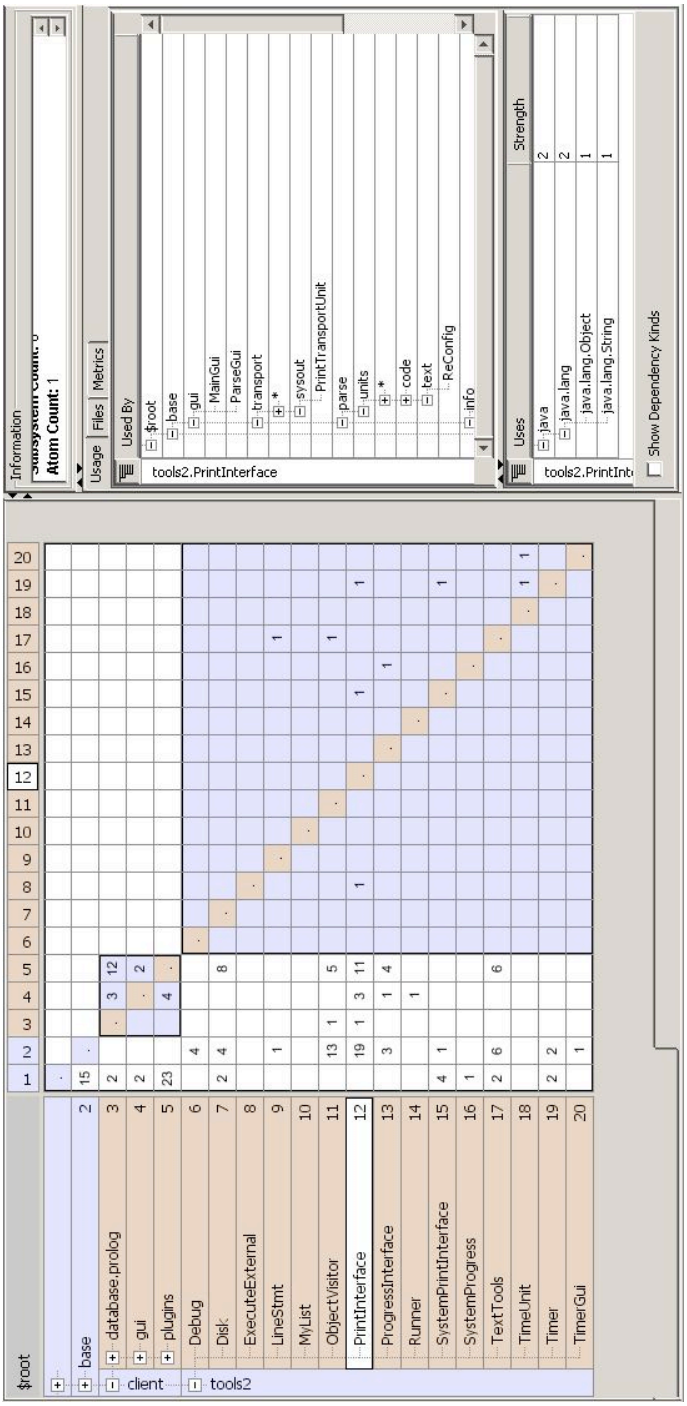

Figure 5.1: A Lattix LDM view of our prototype

LDM provides us with an intuitive way of visualizing dependencies[1]. Using the XML plugin, we categorized the modules and files of the platform using different design criteria ("faking" a Java package structure), and built an XML file for different kinds of dependencies (like an include-dependency file). When this XML file was imported into Lattix LDM, the dependencies where visualized and gave a good overview of the different dependencies throughout the platform.

Another feature in Lattix LDM is the possibility to define rules for dependencies, and have the program mark the occurrences of rule violations. For instance, a module might be defined as being on top of the include tree, which means that no other file should be able to include files from that module (however, there may be include dependencies inside the module). After creating a rule saying that no external file is allowed dependencies to the files in the specific module, all files which includes from this module will be marked red in the LDM matrix. This is an easy way to spot modules which violates certain architectural guidelines.

Although we have had much success using this tool, there are also many drawbacks. One major such is that it is impossible to process large files (size exceeding 25 MB), which locks up the system completely. It is also not recommended to save the LDM parsed from the XML-file, since it corrupts the file names of source files in the LDM.

Since it is a system designed for Java, it is also difficult to link a specific row in the matrix to a file (there is a "view source" command), since the program assumes that the package structure is intact, and the only input one is allowed to give is the location of the top-level package.

Rigi

Rigi (described in [13] and [7]) is a tool for visualizing and manipulating software information. It contains an interpreter for applying operations to the visualized information, as well as possibilities for manually manipulating the information. Rigi presents the extracted information in a graph-like structure (figure 5.2), and contains facilities for filtering different node and arc types. Additionally, Rigi provides the Rigi Standard Format (RSF), used by for example Dali and DISCOVER (see 3.1.1).

As the screenshot shows, even though the model only contain four different nodes, the view seem cluttered. Using this tool on views generated from the mobile platforms, perhaps containing one hundred nodes, would result in a graphical chaos.

SHriMP

SHriMP (Simple Hierarchical Multi-Perspective), described in [7] and [14], is a freeware information visualization and navigation system. When used for reconstruction, the tool can assist an user in generating high-level architectural views of a system by manually grouping elements in a graph (figure 5.3). It takes RSF files as input, and can be used together with Rigi to provide the user with more possibilities for generating views and architectural abstractions.

[1] A demonstration of our prototype interfaced with the Lattix tool was held, in which the strengths of the tool was shown

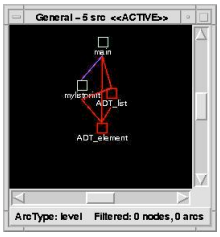

Figure 5.2: The Rigi tool

The SHriMP demos available on the web show promising results, with a visualization which looks much less cluttered than the one found in Rigi. As we will describe in 5.1.2, possibilities for extending the client system to produce input readable by SHriMP is possible.

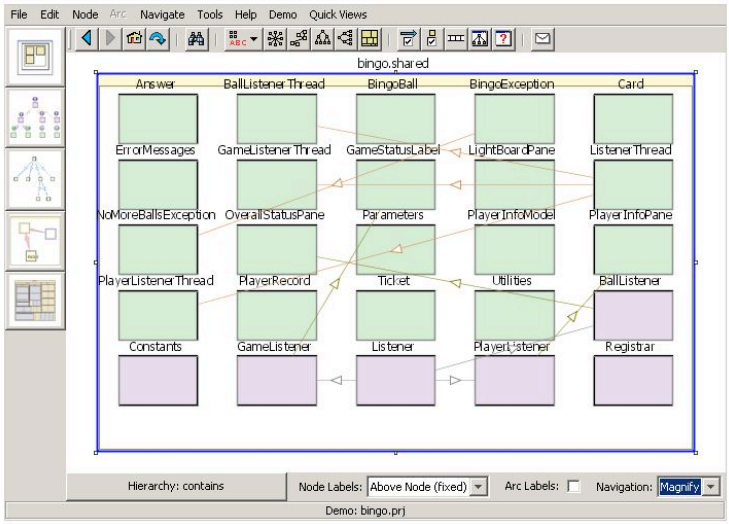

Figure 5.3: The SHriMP tool

5.1.2 Deciding on a program structure

This far, different approaches to client system construction have been discussed, as well as the properties of existing tools. As shown, there are several different ways of implementing the client system of the framework, and based on these, a solution has been developed. Our solution, in short, provides the user with facilities to easily construct programs to extract and process desired information. The programs are used as plugins to a client system, which contains some basic functionality, for instance accessing the database system.

The advantages of this solution are many. Mainly, the users could write plugins suited for their specific needs, without having to change the client system. The plugins can also produce any kind of output, which gives the users

an ability to integrate their plugins with existing tools without limitations from the client system.

Of course, there are also disadvantages with this solution. Perhaps the largest is that it forces the user to actually code a small program and compile it in order to perform operations on the database system. However, since the target group of our framework system is experienced developers, this shouldn't be a large problems, as they probably are used to this kind of approach.

5.2 Design

In this chapter, we will discuss the resulting design of the client system, based on results from the analysis. As stated in 5.1.2, the design will be based on an open plugin structure, allowing the user to decide the output format and plugin behaviour.

5.2.1 Plugins

Our system is based on a series of plugins. These are meant to function as stand-alone applications that can produce any kind of output. However, since many tasks performed by the plugins are similar in nature, a basic set of functions should be provided.

Each plugin is a subclass to an abstract plugin class containing some useful functionality. This functionality includes a *run* method which is invoked on plugin startup, *a SWI-prolog interface* for database interaction, a *JPanel* where user interaction prior to startup can take place, a progress bar, a print console, and information and description of the plugin.

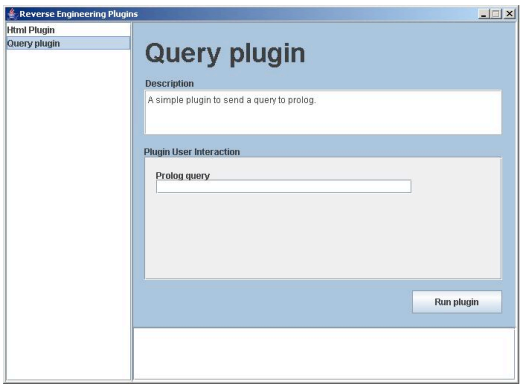

Figure 5.4: The client system GUI

One drawback of the plugin structure, is that the plugins in many cases depend on the Prolog interface. As the system should be flexible, limiting the design to only be able to access a Prolog system is undesirable. However, if the possibility of accessing the database using Prolog queries is removed, the greatest advantage of the Prolog solution, the powerful language, is lost.

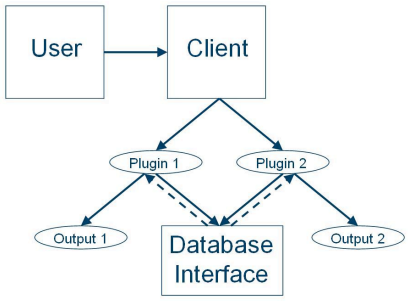

Figure 5.5: A detailed view of the client system

5.2.2 View fusion

In order to explain view fusion and its importance, a definition of the term *view* is in place. A view, in a database context, is a means of presenting information stored in a database. For instance, in an address database, a simple view could display the name and address of a person.

To create a complete view of a system requires that different kinds of extracted data can be *fused*. Fusing architectural views means establishing connections between them. This is important for several reasons, described in [4]:

- Different views provide complementary information. Combining data from different views can produce a more complex view.
- Users need to be able to analyse different views in order to obtain a complete understanding of the system.
- One view can be improved with information from another. One way of improving quality is by cross-checking information with other views, in order to find contradictions.

For this purpose, the plugins of our client system can access and combine different Prolog files, or other files containing system information, in order to perform advanced operations generating output which would be impossible to generate from one view alone.

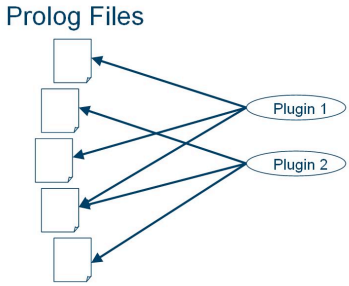

Figure 5.6: View fusion in the client system

5.3 Conclusions

The objective for this part of the thesis was to construct a client system which interacts with the database system described in chapter 4. The client system should then present the extracted information to the user.

As the Prolog language is a highly functional programming language, the problem of user interaction with the database may have reached an answer. Since Prolog contains powerful query operations[2] and the ability to define recursive scripts to gather information, the use of another scripting language for database interaction seems unnecessary.

We have ruled out the solution containing a large system which handles every kind of input and visualization, as this solution would not be as flexible as we wish, and would require a great amount of work in order to provide the user with sufficient functionality to perform operations on the stored data. Instead, an approach involving a minimal client system, which is then extended by a series of plugins was chosen.

The plugins have an open structure, and are designed to produce any kind of output. This leads to a structure where each plugin handles the task of view fusion and queries in the database files, and generation of suitable output based on what level of abstraction is desired. Extending the client system is also easy, since changes are made in either adding a plugin, or re-writing an existing.

Visualisation of the system on different levels can be done using either existing tools, such as Lattix LDM, or by constructing such functionality as a part of a plugin.

5.3.1 Future Work

Due to time limitations, not all external tools which showed promising features were integrated into the client system as plugins producing input for the tools. The SHriMP tool, for instance, seems to provide powerful visualization capabilities which could be used in combination with complicated view fusions.

Since the plugins depend on a prolog interface, perhaps another layer of database interface should be constructed, making the client side plugins independent of the type of database. However, this would rule out the possibility of the plugins using powerful Prolog queries.

[2] In fact, Prolog is a Turing-complete programming language

Chapter 6

Conclusions

Developing a framework for extracting information has been an interesting task, mostly due to the fact that we started out without anyone knowing what kinds of results to expect, but rather a notion that work in this field would provide useful results, and give the developers as well as the designers valuable aid in their everyday work.

While designing our framework, some of the choices made by the authors of the material we studied seemed strange at the moment, but after spending time trying to find better solutions, we soon realized that our initial ideas were simply not feasible. As a result, our findings and experiences made us converge with the views of for instance [5], in the case of the relation database usage instead of a object graph solution. In some cases, working solutions which were theoretically "better" designed than the state-of-the-art researchers failed since we did not fully realize the volume of the platform at first, and that our solutions would take days (if not weeks) to fully process the code base, and sometimes require massive amounts of memory.

We chose to divide our solution into three separate subsystems, in order to provide the future users and developers with an option to replace different parts, or easily extend the present ones. These three parts are the parser system, the database system, and the client system. Regarding our parser, it could be argued that we didn't spend enough time getting an AST solution to work. Such a solution would have made a lot of aspects of the framework simpler, but after encountering more and more difficulties involving the structure of C code which has not been preprocessed, such as conditional compilation flags and macro expansions, we realized that it would take too much time to get a working system. Instead, we decided on the simpler approach, tokenizing the code and then apply search patterns to find occurrences of specific data. This leads to a more open solution, where the parser easily could be extended to handle other forms of input, such as other programming or scripting languages.

Choosing Prolog as our main source of data storage depends on a number of benefits against ordinary relationship databases, such as SQL. First, storing and accessing data in a prolog database is a simple task. We use a Java interface to build and retrieve information from our databases, and we found no problems along the way. Also, since the database consists of text files which are compiled into a prolog system, users can easily generate their own copies of the database (or a subset thereof) and experiment with. However, the main

reason for choosing Prolog, is that it provides the user with powerful scripting opportunities for making complex operations on a large amount of data with only a few small scripts. This gave us a user interface language that was not only well documented and used, but included in the database system itself.

Due to the fact that it is not entirely clear what the individual user wants to use out framework for, the task of designing a user interface which would suit everyone became a very difficult, if not impossible one. Instead, our solution is based on a minimal GUI which interacts with the database through a series of plugins. These are stand-alone applications with functionality for accessing the database. For the same reasons as the user interface is open due to the open nature of the framework, the output from the plugins are not specified. We haven't provided any facilities for generating output (except for the output generated in our sample plugins), as the user is expected to know more about what the framework should be used for. The problem is that it is difficult to find proper uses for the framework, or any design extraction tool for that matter, since it requires good knowledge about the system which is to be analysed in order to know what needs to be done. A consequence of this is that we, since we are no experts on the mobile platform, decided to make the tool as flexible and open as possible, so that people who knows the system could tailor our system to fit his/her specific problem.

One thing that we soon found when the basic framework was in place, was that it could be applied to more situations than initially expected. While asked if we could solve seemingly unrelated problems in the code, our framework proved to be useful in reducing the time it would take to develop a separate program for that task drastically, due to the ease of adding functionality to any part of the system.

In closing, our prototype should be developed further in order to produce a complete tool for information extraction. Based on conversations with SwAG, such a tool is wanted, as it solves some of their more time-consuming problems. However, the prototype is by no means perfect in terms of optimal coding and choice of algorithms, but since we have focused mostly on extendibility, any future work on the framework and the plugins for the parser back-end or the client system can be done separately as we are confident in our interface designs. The thought of integrating parts of our system as eclipse plugins has also surfaced, but due to time constraints, this path has not been explored.

Appendix A

Bibliography

[1] Andrew W. Appel, *Modern Compiler Implementation in Java, Second Edition*, Cambridge University Press, 2002

[2] Ivan Bratko, *PROLOG Programming for Artificial Intelligence, Third Edition*, Addison Wesley, 2001

[3] Rick Kazman, S. Jerome Carrière, *Playing Detective: Reconstructing Software Architecture from Available Evidence*, Technical Report, Software Engineering Institute, Carnegie Mellon University, 1997

[4] Rick Kazman, S. Jerome Carrière, *View Extraction and View Fusion in Architectural Understanding*, Proceedings of the Fifth International Conference on Software Reuse, 1998

[5] Rick Kazman, Liam O'Brien, Chris Verhoef, *Architecture Reconstruction Guidelines, Third Edition*, Technical Report, Software Engineering Institute, Carnegie Mellon University, 2003

[6] Gail C. Murphy, David Notkin, *Lightweight Source Model Extraction*, Department of Computer Science & Engineering, University of Washington, 1995

[7] Liam O'Brien, Christoph Stroemer, Chris Verhoef, *Software Architecture Reconstruction: Practice Needs and Current Approaches*, Technical Report, Software Engineering Institute, Carnegie Mellon University, 2002

[8] Scott R. Tilley, *Discovering DISCOVER*, Technical Report, Software Engineering Institute, Carnegie Mellon University, 1997

[9] *Gnu C Preprocessor*, `http://gcc.gnu.org/onlinedocs/cpp/The-preprocessing-language.html`

[10] *Java 5 Features*, `http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html`

[11] *JavaCC*, `https://javacc.dev.java.net`

[12] *Lattix LDM*, `http://www.lattix.com`

[13] *Rigi* `http://www.rigi.cs.uvic.ca`

[14] *SHriMP*, `http://www.thechiselgroup.org/shrimp`

[15] *SWI Prolog*, `http://www.swi-prolog.org`