Architecture of a Source Code Exploration Tool: A Software Engineering Case Study[1]

Timothy C. Lethbridge & Nicolas Anquetil

School of Information Technology and Engineering (SITE)
150 Louis Pasteur, University of Ottawa, Canada
tcl@site.uottawa.ca, anquetil@csi.uottawa.ca

Abstract

We discuss the design of a software system that helps software engineers (SE's) to perform the task we call *just in time comprehension* (JITC) of large bodies of source code. We discuss the requirements for such a system and how they were gathered by studying SE's at work. We then analyze our requirements with respect to other tools available to SE's for the JITC task. Next, we walk through system design and the object-oriented analysis process for our system, discussing key design issues. Some issues, such as dealing with multi-part names and conditional compilation are unexpectedly complex.

1 Introduction

The goal of our research is to develop tools to help software engineers (SE's) more effectively maintain software. By analyzing the work of SE's, we have come to the conclusion that they spend a considerable portion of their time exploring source code, using a process that we call just-in time program comprehension. As a result of our analysis, we have developed a set of requirements for tools that can support the SE's.

In section 2 we present our analysis and the requirements; we also explain why other tools do not meet the requirements. In section 3, we present our design for a system that meets the requirements. Since there is not space for exhaustive design details, we focus on key issues that we believe would be of interest to others who are building tools.

Our motivations for writing this paper are twofold:

- To contribute to the state of knowledge about software engineering tools by presenting requirements and how we have addressed problems. We believe there are not enough papers in the public domain that present issues in software architecture – most such writings can only be found in inaccessible software design documentation.
- To improve the state of research practice in this area by highlighting aspects of our research that we feel are important. Such areas include performing effective analysis of the SEs' task by studying real SE's, and focusing on tools for which there are demonstrable requirements.

2 Development of Requirements for the System

Requirements for the tool have been determined by the careful study of SE's in real working environments [10]. In this section we briefly discuss how the requirements were derived. We then present and explain the software engineering task we are addressing and key tool requirements. Finally we explain why other tools are not able to fulfill these requirements.

2.1 Studying Software Engineers to Determine Requirements

Many software engineering tools are developed by people who have 'good ideas', or by people

[1] This work is supported by NSERC and Mitel Corporation and sponsored by the Consortium for Software Engineering Research (CSER). The IBM contact for CSER is Patrick Finnigan.

responding to particular needs expressed by SE's (perhaps themselves).

We believe, however that a proper scientific and engineering approach should instead start with detailed observation of the work practices of SE's. Furthermore, as Von Mayrhauser and Vans point out [15], the SE's studied must be developing significant software systems in an industrial setting.

In our approach, we analyze such things as the detailed tasks the SE's perform (especially work-arounds they are forced to perform due to lack of adequate tools) and the constraints of their work environment. This approach has much in common with Vicente's cognitive work analysis methodology [14].

Development of tools is driven by this analysis; the tools are then validated by testing whether they actually improve the productivity of the working SE's.

We have used the following techniques to study SE's:

- General interviews about SEs' process, product and tools.
- Series of interviews held every few weeks for lengthy periods, where we discuss what SE's have done since the last interview.
- Observation sessions where we record what we observe the SE's doing.
- Automatic logging of the SEs' use of all tools available to them.
- Sessions where we walk through tools feature-by-feature.

Most of our work has taken place at Mitel Corporation, although we have also worked with people in several other companies. For more detail, see [4, 5, 12].

2.2 The Software Engineering Task that We Address: Just in Time Comprehension of Source Code

Almost all the SE's we have studied spend a considerable proportion of their total working time in the task of trying to understand source code prior to making changes. We call the approach they use *Just in Time Comprehension (JITC)* [11]; the reason for this label will be explained below. We choose to focus our research on this task since it seems to be particularly important, yet lacking in sufficient tool support.

The 'changes' mentioned in the last paragraph may be either fixes to defects or the addition of features: The type of change appears to be of little importance from the perspective of the approach the SE's use. In either case the SE has to explore the system with the goal of determining where modifications are to be made.

A second factor that seems to make relatively little difference to the way the task is performed is *class of user*: Two major classes of users perform this task: Novices and experts. Novices are not familiar with the system and must learn it at both the conceptual and detailed level; experts know the system well, and may have even written it, but are still not able to maintain a complete-enough mental model of the details. The main differences between novice and expert SE's are that novices are less focused: They will not have a clear idea at which items in the source code to start searching, and will spend more time studying things that are, in fact, not relevant to the problem. It appears that novices are less focused merely because they do not have enough knowledge about what to look at; they rarely set out to deliberately learn about aspects of the system that do not bear on the current problem. The vision of a novice trying to 'learn all about the system', therefore seems to be a mirage.

We observe that SE's repeatedly *search* for items of interest in the source code, and *navigate* the relationships among items they have found. SE's rarely seek to understand any part of the system in its entirety; they are content to understand just enough to make the change required, and to confirm to themselves that their proposed change is correct (impact analysis). After working on a particular area of the system, they will rapidly forget details when they move to some other part of the system; they will thus re-explore each part of the system when they next encounter it. This is why we call the general approach, just-in-time comprehension (JITC). Almost all the SE's we have studied confirm that JITC accurately describes their work paradigm – the only exceptions were those who did not, in fact, work with source code (e.g. requirements analysts).

2.3 List of Key Requirements for a Software Exploration Tool

We have developed a set of requirements for a tool that will support the just-in-time comprehension approach presented in the last section. Requirements of relevance to this paper are listed and explained in the paragraphs below. Actual requirements are in italics; explanations follow in plain text.

The reader should note that there are many other requirements for the system whose discussion is beyond the scope of this paper. The following are examples:

- Requirements regarding interaction with configuration management environments and other external systems.
- Requirements regarding links to sources of information other than source code, such as documentation.
- Requirements about features that allow SE's to keep track of the history of their explorations (e.g. so they can push one problem on a stack while working on a subproblem).
- Detailed requirements about usability.

Functional requirements. The system shall:

F1 Provide search capabilities such that the user can search for, by exact name or by way of regular expression pattern-matching, any named item or group of named items that are semantically significant[2] *in the source code.*

The SE's we have studied do this with high frequency. In the case of a file whose name they know, they can of course use the operating system to retrieve it. However, for definitions (of routines, variables etc.) embedded in files, they use some form of search tool (see section 2.4).

F2 Provide capabilities to display all relevant attributes of the items retrieved in requirement F1, and all relationships among the items.

We have observed SE's spending considerable time looking for information about such things as the routine call hierarchy, file inclusion hierarchy, and use and definitions of variables etc. Sometimes they do this by visually scanning source code, other times they use tools discussed in section 2.4. Often they are not able to do it at all, are not willing to invest the time to do it, or obtain only partially accurate results.

Non-functional requirements. The system will:

NF1 Be able to automatically process a body of source code of very large size, i.e. consisting of at least several million lines of code.

As discussed in section 2.1, we are concerned with systems that is to be used by real industrial SE's. An engineer should be able to pick any software system and use the tool to explore it.

NF2 Respond to most queries without perceptible delay.

This is one of the hardest requirements to fulfill, but also one of the most important. In our observations, SE's waste substantial time waiting for tools to retrieve the results of source code queries. Such delays also interrupt their thought patterns.

NF3 Process source code in a variety of programming languages.

The SE's that we have studied use at least two languages – a tool is of much less use if it can only work with a single language. We also want to validate our tools in a wide variety of software engineering environments, and hence must be prepared for whatever languages are being used.

[2] We use the term *semantically significant* so as to exclude the necessity for the tool to be required to retrieve 'hits' on arbitrary sequences of characters in the source code text. For example, the character sequence 'e u' occurs near the beginning of this footnote, but we wouldn't expect an information retrieval system to index such sequences; it would only have to retrieve hits on words. In software the semantically significant names are filenames, routine names, variable names etc. Semantically significant associations include such things as routine calls, file inclusion.

NF4 Wherever possible, be able to interoperate with other software engineering tools.

We want to be able to connect our tools to those of other researchers, and to other tools that SE's are already using.

NF5 Permit the independent development of user interfaces (clients).

We want to perform separate and independent research into user interfaces for such tools. This paper addresses only the overall architecture and server aspects, not the user interfaces.

NF6 Be well integrated and incorporate all frequently-used facilities and advantages of tools that SE's already commonly use.

It is important for acceptance of a tool that it neither represent a step backwards, nor require work-arounds such as switching to alternative tools for frequent tasks. In a survey of 26 SE's [5], the most frequent complaint about tools (23%) was that they are not integrated and/or are incompatible with each other; the second most common complaint was missing features (15%). In section 2.4 we discuss some tools the SE's already use for the program comprehension task.

NF7 Present the user with complete information, in a manner that facilitates the JITC task.

Some information in software might be described as 'latent'. In other words, the software engineer might not see it unless it is pointed out. Examples of such information are the effects of conditional compilation and macros.

Acceptable limitations:

L1 The server component of the tool may be limited to run on only one particular platform.

This simplifies implementation decisions without unduly restricting SE's.

L2 The system is not required, at the present time, to handle object oriented source code.

We are restricting our focus to SE's working on large bodies of legacy code that happens to be written in non-object-oriented languages. Clearly, this decision must be subsequently lifted for the tool to become universally useful.

L3 The system is not required, at present, to deal with dynamic information, i.e. information about what occurs at run time.

This is the purview of debuggers, and dynamic analysis tools. Although it would be useful to integrate these, it is not currently a requirement. We have observed software engineers spending considerable time on dynamic analysis (tracing, stepping etc.), but they consume more time performing static code exploration.

2.4 Why Other Tools are Not Able to Meet these Requirements

There are several types of tools used by SE's to perform the code exploration task described in section 2.2. This section explains why, in general, they do not fulfill our requirements:

Grep: Our studies described in section 2.1 indicated that fully 25% of all command executions were of one of the members of the grep family (grep, egrep, fgrep, agrep and zgrep). Interviews show that it is the most widely used software engineering tool. Our observations as well as interviews show that grep is used for just-in time comprehension. If SE's have no other tools, it is the key enabler of JITC; in other situations it provides a fall-back position when other tools are missing functionality.

However, grep has several weaknesses with regard to the requirements we identified in the last section:

- It works with arbitrary strings in text, not semantic items (requirement F1) such as routines, variables etc.
- SE's must spend considerable time performing repeated greps to trace relationships (requirement F2); and grep does not help them organize the presentation of these relationships.

- Over a large body of source code grep can take a large amount of time (requirements NF1 and NF2).

Search and browsing facilities within editors: All editors have some capability to search within a file. However, as with grep they rarely work with semantic information. Advanced editors such as emacs (used by 68% of a total of 127 users of text-editing tools in our study) have some basic abilities to search for semantic items such as the starts of procedures, but these facilities are by no means complete.

Browsing facilities in integrated development environments: Many compilers now come with limited tools for browsing, but as with editors these do not normally allow browsing of the full spectrum of semantic items. Smalltalk browsers have for years been an exception to this, however such browsers typically do no not meet requirements such as speed (NF2), interoperability (NF4), and multiple languages (NF3). IBM's VisualAge tools are to some extent dealing with the latter problem.

Special-purpose static analysis tools: We observed SE's using a variety of tools that allow them to extract such information as definitions of variables and the routine call hierarchy. The biggest problems with these tools were that they were not integrated (requirement NF6) and were slow (NF2)

Commercial browsing tools: There are several commercial tools whose specific purpose is to meet requirements similar to ours. A particularly good example is Sniff+ from Take5 Corporation [13]. Sniff+ fulfills the functional requirements, and key non-functional requirements such as size [NF1], speed [NF2], multiple languages [NF3], its commercial nature means that it is hard to extend and integrate with other tools.

Program understanding tools: University researchers have produced several tools specially designed for program understanding. Examples are Rigi [6] and the Software Bookshelf [3]. Rigi meets many of the requirements, but is not as fast [NF2] nor as easy to integrate other tools [NF6] as we would like. As we will see later it differs from what we would like in some of the details of items and relationships. The Software Bookshelf differs from our requirements in a key way: Before somebody can use a 'bookshelf' that describes a body of code, some SE must organize it in advance. It thus does conform fully with the 'automatically' aspect of requirement NF1.

3 Issues in System Design

In this section we examine several issues in system design for the software exploration tool.

3.1 The Need for a Precompiled Database

Figure 1 shows an architecture for the simplest possible software exploration system. Such a system processes the source code on demand, whenever an application program makes a query. This is the way tools such as grep function.

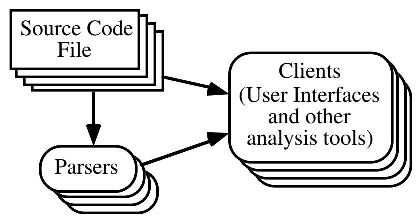

Figure 1: Simple data flow diagram of a system that lacks a database.

In an architecture like figure 1, it would be possible to create new parsers (requirement NF3) and user interfaces (requirement NF5). Also, some user interfaces might bypass parsers for certain operations (e.g. simply displaying code).

Clearly however, such a system can not be fast enough to simultaneously meet requirements F1, F2, NF1 and NF2. This is because some operations have to process much or all of the source code files before returning an answer. Hence there must be a precompilation process that generates a database of information about the software. Such an architecture is shown in figure 2. Discussion about the schema of the database–and how we can make the database fast enough–is deferred to section 4.

In figure 2, we have shown two API's, one each for writing and reading the database. Discussion of these is deferred to sections 5.1 and 5.2

respectively. The latter contains basic calls that provide all the facilities applications need to satisfy requirements F1 and F2.

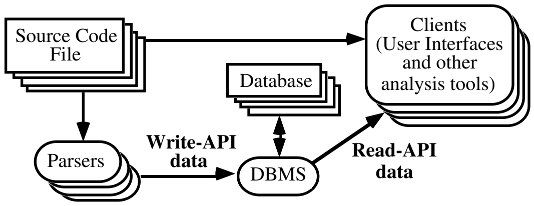

Figure 2: Modified version of figure 1, with the addition of a database to represent information about the source code.

3.2 The Need for an Interchange Format

Since third-party parsers are to be used, as shown in figures 1 and 2, we should impose a constraint that these parsers must make calls to a common write-API. However since we also want to allow the system to interoperate with other tools that independently produce or consume data about source code (requirement NF4), then we need to develop an interchange format. Changes to the architecture to accommodate this are shown in figure 3.

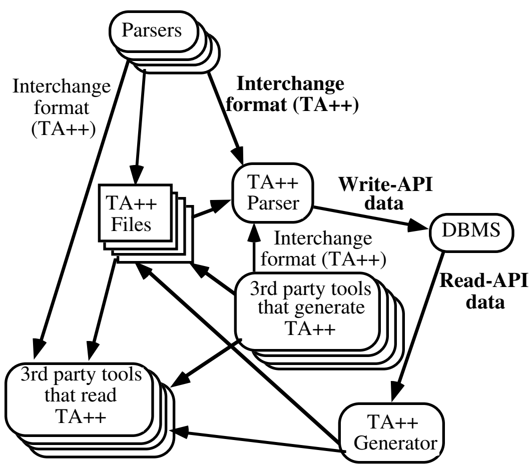

Figure 3: Expansion of part of figure 2 to show the use of an interchange format.

We call our interchange format TA++ because it uses a generic tuple-attribute syntax called TA developed by Holt [2]. We are working with his group on various tools that will interoperate. TA++ is described in more detail in section 5.3.

TA++ is generated by all programming language parsers. It is directly piped into a special TA++ parser which builds the database (although storage of TA++ in files or transmission over network connections is also anticipated). Our system also has a TA++ generator which permits a database to be converted back to TA++. This is useful since TA++ is far more space-consuming than the database thus we don't want to unnecessarily keep TA++ files.

Data in TA++ format, as well as data passed through the DBMS write-API, are both merely lists of facts about the software as extracted by parsers.

3.3 The Need for a Query Language

Although the read-API provides basic query facilities, an API doesn't easily lend itself to composite queries, e.g. "Tell me the variables common to all files that contain calls to both routine x and routine y." For this, a query language is needed, as illustrated in figure 4. Discussion of this language is in section 5.4.

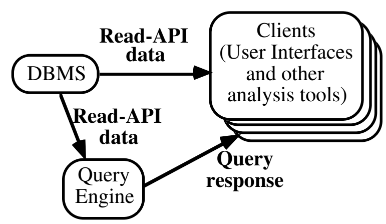

Figure 4: Expansion of part of figure 2 to show a query engine.

3.4 The Need for Auxiliary Analysis Tools

The architecture discussed so far presupposes that all data needed by clients comes from two sources:

- The precompiled database of facts about the source code.
- Unprocessed source code itself.

However, although the database contains comprehensive information about source code objects and their relationships, there are certain data whose storage in the database would not be appropriate, but which is still needed to satisfy requirements F1, F2 and NF6. These include, but are not limited to, the following:

- Details of the complete syntax tree (or just the control flow) *within* a file. We concluded in design issue A that a precompiled database is necessary to store system-wide data; however intra-file information can be rapidly enough obtained at query time, while still satisfying requirement NF2.
- Data to permit full-text keyword or regular expression searches of such aspects of a system as comments. We believe that these are best stored in an information retrieval system that is optimized for that purpose. Also we believe that traditional grep can be effectively incorporated within our architecture when it is to be used within a single file or routine.
- Conceptual knowledge about the system (with pointers into the source code). See [9] for a discussion of this.

Figure 5 shows how auxiliary analysis tools are incorporated into our architecture. Figure 6 shows the composite data flow diagram, combining figures 2 through 5.

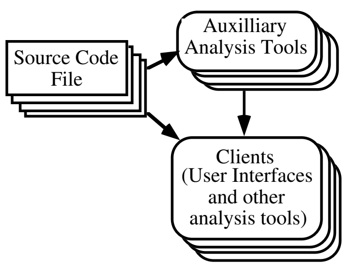

Figure 5: Expansion of part of figure 2 to show the incorporation of auxiliary analysis tools that have their own precompilation facilities, or are fast enough not to need precompilation.

4 Database Design Issues

In this section we examine key issues that arose during the design of the database. Most involve design of the schema, although some problems forced us to think about parsing strategies.

We will use the OMT modeling notation and methodology of Rumbaugh et al. [8], although decisions about the actual implementation paradigm will be deferred to section 4.10

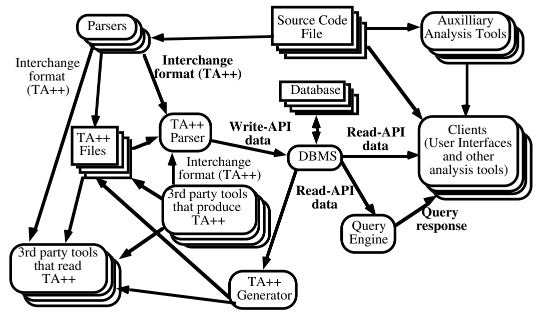

Figure 6: Final data flow diagram for the system. Composite of figures 2 through 5.

4.1 Initial Lists of Classes and Associations.

To fulfill requirement F1, we must ascertain all classes of named items in the source code (restricting ourselves to non-object-oriented languages as permitted by limitation L2). In section 3.4, we considered how to deal with words in comments and syntax-tree issues that have no impact across the boundaries of files. We are therefore left, upon initial analysis, with the following objects of interest:

a) Source code files
b) Routines
c) Variables
d) Types
e) Constants
f) Fields

The following associations are then of interest:

a) Locations of definitions
b) Locations where defined objects are used
c) File inclusion (really a special case of b)
d) Routine calls (another special case of b)

4.2 Recursive Hierarchy of Source Units

Figure 7 presents a simplistic object model relating source files and routines. We can improve this by recognizing that routines are just smaller units of source. We thus derive the recursive diagram shown in figure 8. See [7] for a discussion of this design pattern[3].

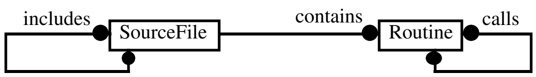

Figure 7: Very simple object model for representing information about source code. This is the starting point for subsequent discussions. Attributes are not considered at this stage of analysis.

Note that a potential extension to figure 8 would be additional subclasses of SourceWithinFile, such as blocks.

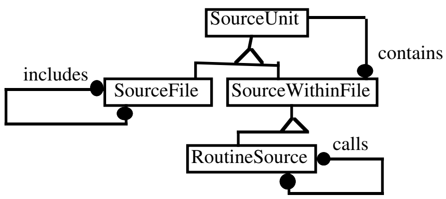

Figure 8: Modification of figure 7 using a recursive framework. This recognizes that not only files can contain routines, but so can routines thmnselves in some languages. Other kinds of items can be defined at any level of source unit.

4.3 References vs. Definitions

In figure 8, the call hierarchy is represented by pointers from RoutineSource to RoutineSource. Upon careful thought however, it should be clear that this relationship should not be reflexive.

Consider routine R1, in which a call is made to R2. We know R1 exists, but we don't know for sure that R2 exists – there may be an error, or the code may be incomplete (yet to fulfill requirement NF7, we still need to record the information). All we know about R2 is that *there exists a reference* to something with that name. This principle extends to file inclusion, and reference to variables and types (which may be externally defined) etc.

In the database, we therefore create a fundamental distinction between objects that we know exist (SourceUnit and Definition) versus references we know are made (ReferenceExistence). Figure 9, illustrates how this division is implemented as an inheritance hierarchy in the database.

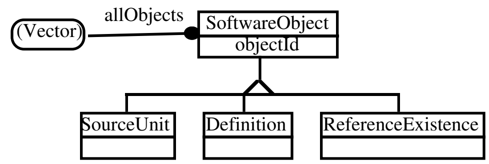

Figure 9: The top of the inheritance hierarchy of software objects. The vector and the objectId are discussed later.

Figure 10 gives examples of how ReferenceExistence and other classes are related. As suggested in section 3.4, if a SourceUnit makes more than one reference with the same to a given class of object (e.g. several calls to the same routine) then only a single link is made to the ReferenceExistence with that name. To find out the total number of references in a file therefore requires a run-time scan of that file.

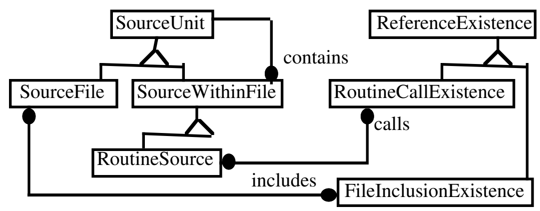

Figure 10: Modification of figure 8 showing separate references.

4.4 Dynamic References

One of the more difficult problems we have encountered is how to deal with situations where pointers to routines can be manipulated. In such cases, it becomes impossible to statically create a complete call hierarchy. In limitation L1, we accepted not to perform dynamic analysis; however in some programs, routines are almost all called in this manner.

We have not adequately solved this problem. Our proposed partial solution is to flag:

- routines whose name is assigned to a variable;
- variables that contain routine names

...so that explorers at least are aware of potential complexity. This solution will not work, how-

[3] In the more popular design pattern book by Gamma et al[1], this is really an inverse *Composite*, because the root of the hierarchy is recursion-limiting case, instead of leaves.

ever, with languages where routines can be called from strings that are created at run-time.

4.5 The Many-to-One Relationship Between Source Objects and Names.

So far, none of the OMT diagrams above have included the names of objects. A naive approach would be to add 'name' as an attribute of SoftwareObject (or perhaps of two or more less abstract classes). However there are three problems with this:

Firstly, many objects can have the same name (e.g. a variable and a routine, or two variable in different scopes). Secondly, given a name we want to be able to quickly find everything with that name. Thirdly, some definitions can be unnamed (e.g. a structure used within a more complex type).

Figure 11, is thus an appropriate model for names.

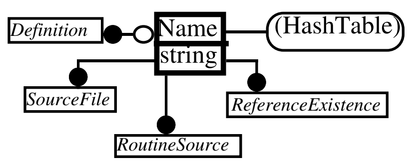

Figure 11: Separating names from objects. The hash table is explained in section

4.6 Multi-Part Names

Some references may not be to particular defined items (e.g. simple variable and field names), but rather to defined items *qualified* by field names.

For example, imagine file f1 contains reference a.b and file f2 contains references a.b.c, and b.d. The components following the dots are field names.

Figure 12 shows an object model that permits the SE to search for particular qualified references. E.g. the SE could find out that 'a.b.c' occurs in file f2. Unfortunately, there is likely to be a combinatorial explosion of possible qualified references.

An alternative approach, illustrated in figure 13, is to store the fact that 'a' is followed by 'b', and 'b' by 'c' *somewhere*, but not to say in exactly which files these qualifications occur. The SE can still find out in which files the objects themselves occur.

This second approach prevents a potential exponential increase in memory requirements. We believe also that it meets requirements NF1, NF2 and NF7 because:

- It provides sufficient information for the system to narrow the set of files for a fully qualified search, such that the search can be quickly done by scanning files at run-time for the particular pattern.
- Fully qualified searches will be relatively rarely requested, compared to searches for variables and fields.

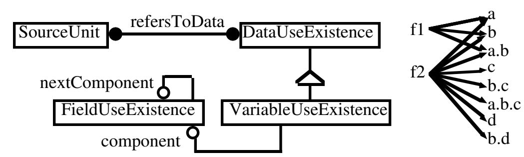

Figure 12: One approach to storing multi-part names that permits searches for particular combinations of components. The class diagram is at the left, and an example instance diagram is at the right. This example requires 15 objects and 16 links.

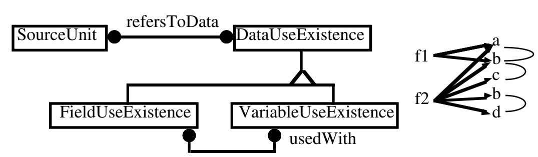

Figure 13: A less space-consuming way of storing multi-part names that sacrifices some search capability. This same example as figure 12 needs only 7 objects and 10 links. Note that b appears twice because the first is a FieldUseExistence and the second is a VariableUseExistence.

4.7 Conditional Compilation

Due to the presence of conditional compilation, it is naive to think of a file containing a simple series of definitions. In fact, when conditional compilation is used, there can be a combinatorial explosion of sets of distinct definitions logically contained within the file.

Although only one of the possible sets of definitions will be present each time code is compiled, it is not acceptable (due to requirement

NF7) to pick just one set to present to the SE exploring code. He or she needs to know about which sets are possible and what is the effect of each set.

We have found this to be a particularly hard problem to solve, because the design of parsers is made significantly more complex (a parser for an ordinary grammar for a language will fail because conditional compilation statements can break up the syntax at arbitrary points). The following is the approach we favor:

- Pre-parse each file looking for conditional compilation directives, and the boundaries of top-level definitions that they break up.
- Reconstruct each file, building as many alternative top-level definitions as needed to account for all cases of conditional compilation. Each top-level definition variant is then given a modified name that reflects the conditions required for it to be compiled.
- A composite file, containing all top-level definition variants is then compiled in the normal way.

4.8 Macros

Macros are similar to conditional compilation in the sense that they can, at their worst, result in arbitrary syntax (for example, in cases where punctuation is included in the macro). Of course it is very bad programming practice to create trick macros that break syntax; however tools like our system are all the more needed when code is very obscure.

The following are approaches to this problem:

- Parse code that has already been preprocessed. The drawback to this is that information is lost – and thus requirement NF7 is not met.
- Preprocess only those top-level definitions where syntax is broken by macros. Treat all other macros as constants or function calls. This is the approach we favor.

4.9 Access to Global Starting Points in the Database.

To fulfill the speed requirement (NF2), there must be a way to rapidly access objects by name and unique object-id (unique handles which can be manipulated by other programs). These requirements explain the vector and hash table objects in figures 9 and 11.

Figures 14 and 15, complete the system's object model and show a couple of extra vectors that optimize routine and file enumeration.

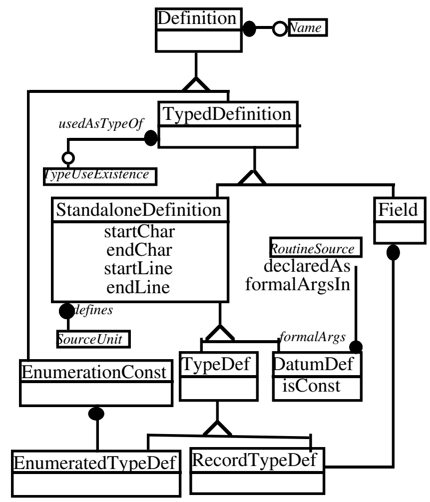

Figure 14: Complete Definition hierarchy.

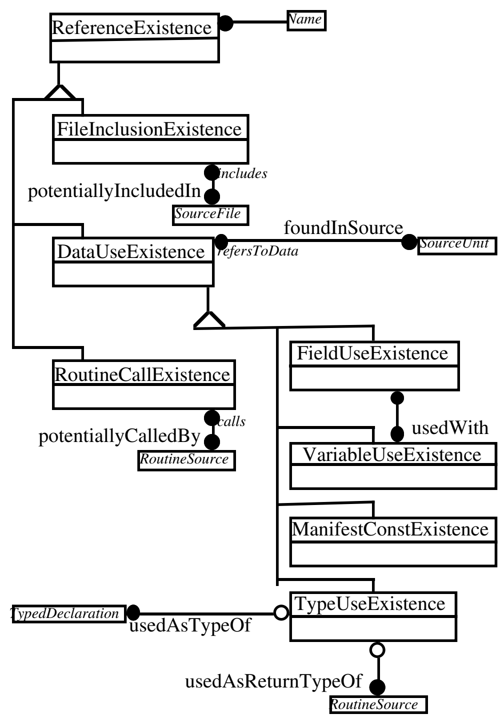

Figure 15: Complete ReferenceExistence hierarchy..

4.10 Database Logical Format

Although we used object-oriented modeling for our database schema, the issue of logical format (paradigm) for the database implementation remains. The key decision here is whether to use a relational or object oriented approach. We chose the latter for the following reasons:

- Relational databases are useful when one is uncertain about the types of queries that will be made. One can create interesting joins to mine new data out of the database. In our case, however, we are confident of the relatively limited number of simple queries that will be needed.
- Object oriented databases are useful when the data naturally involves inheritance. We observe this in our model. Inheritance is hard to model in a relational database.
- Object oriented databases are more practical when typical queries will hop from object to object many times. In a relational database, such queries would require many joins; however, in an object-oriented database one can simply follow pointers.

4.11 Database Physical Format

Having decided to use an object oriented database, the next decision is which product to use. We noted the following factors when making the decision:

- The database will be written once, by the parser, and read concurrently by many people.
- There is, therefore, no requirement for locking to allow for concurrent updates.
- The database will be very large (on the order of 100-200 MB) for typical systems.
- As requirement NF2 states, access must be fast.

These factors indicate that we do not need or want the overhead of a full database product. Instead we adopted a simpler solution: We use the Unix mmap facility, which allows a file to be mapped into virtual memory. When a database is being built, C++ objects are allocated from this region of virtual memory, and are thus written to the file. When somebody then wants to access the database, they merely remap the database into virtual memory and all the C++ objects are available again.

To make this scheme work, we designed a general-purpose facility (an improvement over the standard gnu memory-mapping package) that wraps around mmap and provides replacements for malloc, free etc. and allows the memory map file to dynamically grow and shrink as objects are created and destroyed.

This solution makes programming very simple. The programmer can completely ignore the issue of persistence, except to make calls to map and unmap the database. This is a tremendous improvement over the normal situation where much programming effort goes into designing methods to input and output each object.

The biggest difficulty with this solution was overcoming limitations of C++ compilers. The GNU compiler, and others, add an extra hidden field to each object that points to the virtual table (which allows dynamic binding). The virtual table appears at a different place every time changes are made to the system and it is recompiled. This renders database files with old memory-mapped objects invalid (their virtual table pointers point to the wrong place).

We overcame this problem by encapsulating all the database access code in a Unix shared object. Now all the other parts of the system can be recompiled and linked, but as long as the shared object is untouched, databases remain valid.

5 Language and Interface Design Issues

In this section we highlight important features of the various languages and interfaces that we developed as part of this work. Due to space, we are not able to give complete syntaxes and semantics, but we plan to include additional details as attachments in the CD-ROM version of this paper (distributed separately from the printed CASCON proceedings).

All of these languages and interfaces have been designed to allow various parts of the system to be independently developed, and to allow others to add facilities to the system.

5.1 The Database Read-API

The read-API is one of the two external interfaces to the DBMS, as illustrated in figures 2, 3, 4 and 6. It is designed to allow programs to ask simple questions about the objects in the database schema (figure x).

The following are interesting design issues:

Datatypes manipulated by the API: The DBMS is written in C++, however for flexibility the Read-API is designed to be linked to both C++ and C programs. C++ programs communicate through the API using specific C++ classes, whereas these become opaque references when a C program uses the API. The only thing a C program can do is pass one of the opaque references back through the API.

Among the C++ classes that form arguments to Read-API functions are:

- Each of the concrete classes described in the database schema.
- A series of special Iterator classes. Iterators are objects that return one item at a time from a collection. The read-API returns an iterator to allow a client to traverse the elements of an association. We had to create a sophisticated general-purpose hierarchy of iterator classes to provide such facilities as nested iteration (one iterator calls another internally) and sequential iteration (a special iterator that calls a sequence of iterators to logically concatenate two lists). Iterators are important in this system so that clients can display partial results of a query as soon as they are available. They are also important for relational operators in the query language (section 5.4).

Categories of functions in the Read-API: The Read-API can be subdivided as follows:

- Four functions that provide all that is necessary to query the associations. The four functions are:

```
cdbGetObjectListThatReferToObject()
cdbGetObjectListReferedByObject()
cdbGetObjectListThatDefineObject()
cdbGetObjectListDefinedByObject()
```

The first two traverse the refers/referred-by associations in either direction, while the second traverse the defined-in/defined-by associations in either direction[4].

They all return an iterator and take as arguments 1) an object that is the starting point for links to be traversed, and 2) a class that constrains the links to be traversed. For example, I could ask for all the 'refers-to' links from a RoutineSource to class ReferenceExistence in general, in which case I would get a list containing routines, variables, types etc. On the other hand I could ask for the 'refers-to' links from a RoutineSource to a more specific class such as RoutineCallExistence, in which case I would be asking about one level of the call hierarchy, and would get a subset of the results of the former query.

The above design allows for a small number of functions to efficiently handle a large number of types of query.

- About 20 functions to request simple attributes of each class of object.
- Four functions perform global searches (efficiently using the hash table, or, for regular expressions, less efficiently using the vectors) over all objects or all objects of a class. They return an iterator.
- Several miscellaneous functions to connect to a database and obtain global statistics (e.g. the number of instances of each class).

5.2 The Database Write-API

The Write-API is somewhat simpler than the Read-API. Although it was designed to be used by a variety of programs, it is only used by the TA++ parser in our current architecture (Figure 6). It contains the following types of functions:

- One function to create each concrete class. Arguments are the attributes of the class. Return value is the new object.
- Generic functions to add links of refers/referred-by and defines/defined-by associations. A single call to one of these functions creates both directions of the association.

---

[4] Although there are various different association names, they can be divided into these two categories.

5.3 TA++

As discussed in section 3.2, we designed an interchange format to allow various different program comprehension tools to work together (e.g. sharing parsers and databases).

The TA language [2] is a generic syntax for representing the nodes and arcs. Here are some key facts about it:

- Each TA file has two sections.
- The first 'fact tuple' section defines the legal nodes and instances of binary relations (arcs) between the nodes. Each line of this section is a 3-tuple containing the relation, and the two nodes being related.
- There are certain special relations built-in to TA. For example there are relations to define the 'scheme' of allowable node types. Also there is the relation '$instance', which relate a new node to its type.
- The second 'fact attribute' section of a TA file contains a separate syntax to specify attributes of nodes.

For our system, we adopted TA with no changes whatsoever. This means that any program capable of processing TA in general can read our interchange format. However TA++ is restricted in the sense that it require a specific 'scheme'. The scheme defines the set of valid node types (the concrete classes in our database) with their allowable attributes. Our system therefore cannot read generic TA because it requires that the TA++ file be limited to our specific scheme.

The following is an example of TA++:

```
FACT TUPLE:
$INSTANCE flistaud.pas SourceFile
$INSTANCE audit_str_const_1 DatumDef
defines flistaud.pas audit_str_const_1

FACT ATTRIBUTE:
audit_str_const_1 {
  startChar = 1883
  endChar = 1907
  isConst = 1 }
```

Two objects are first defined. This is followed by a tuple that declares that the datum audit_str_const_1 is defined in file flistaud.pas. The final section specifies where in the file the definition can be found, and that the definition is in fact a constant (as opposed to a variable).

It should be clear that blocks of TA++ code map fairly directly to calls to the Write-API. The key difficulty is one of ordering:

- Ensuring that all the information about attributes of objects has been obtained before the object is created.
- Ensuring an object is defined before instances of relations are made that need it.

5.4 The Query Language

As discussed in section 3.3 we designed a text-oriented query language for our system. This permits:

- Complex nested queries
- Communication using simple pipes, sockets etc.

The query language processor simply calls the Read-API to do most of the work. The following is a description of the queries:

- Responses to most queries are lists of objects, one per line. Each object is represented by a unique object-id (an integer, see figure 9), so that objects with the same name can be distinguished. The object-id is followed by several key attributes: such as a code representing its class and its name. These attributes are redundant since the 'info' query provides this information; however it is nevertheless useful for clients since they almost always want this information.

- The following are basic query formats that return object lists. If *result-class* is omitted in any of the following then SoftwareObject is assumed. Parentheses may be used to group when necessary.

  `<result-class> <name>`
  Returns objects of that class with that name.

  `<result-class>` **`reg`** `<pattern>`
  Returns objects that match the pattern. Grep patterns are used. If pattern is empty, then all objects of the class are returned.

  `<result-class> <association> <query2>`
  Returns all objects that are in the given relation to the objects retrieved as a result of query2.

  `(<query>) <operator> <query>`

Returns objects by composing queries. Operator **and** means find the intersection of the two queries; operator **or** means find the union; operator **–** means set difference

- There are some special flow-control directives that allow clients to obtain partial answers to lengthy queries and then decide if they want more. These take advantage of the iterators built into the Read-API (see section 5.1).

  `first <n> query`
  Requests the system to return only the first n responses.

  `next <n>`
  Requests more responses, if any, to the last query.

- The only other important queries are as follows. These cannot be composed.

  `info <object-id>`
  Returns the attributes in a format similar to TA++

  `classinfo <class-id>`
  Provides statistics about the class in general (e.g. number of instances).

6 Conclusions and Future work

SE's spend a considerable portion of their time performing just-in-time program comprehension. To assist them, they need a fast, open and integrated tool that allows them to explore the relationships among all items of interest in a body of source code of arbitrary size.

In this paper we discussed the design of such a system. We highlighted the need for a scientific approach to work analysis so as to effectively define the requirements. We then discussed design issues such as the following:

- Information about software needs to be stored in a database for fast access, and also exchanged with a variety of tools using a low-level fact-language as well as a high-level query language.
- A database schema needs to handle the recursive nature of program source code, the distinction between references and definitions, as well as the various complexities in the way that program objects are named.
- Parsers that extract data from source code for use in the database have to intelligently deal with conditional compilation and macros. Parsing becomes more complex than when simply designing a compiler because it is important to show the SE all alternative parses before macro expansion.
- An object oriented database that lacks the overhead of most commercial DBMS products is needed to deliver sufficient responsiveness.

The design process described in this paper has opened several areas for potential future research. Of particular interest are effective syntaxes for interchange and query languages, and how to deal with conditional compilation in a program understanding tool. A separate issue, not discussed in this paper, is designing an effective user interface.

Acknowledgments

We would like to thank the SEs who have participated in our studies, in particular those at Mitel with whom we have worked for many months. We would like to thank the members of our team who have developed and implemented the ideas discussed in this paper: Abderrahmane Lakas, Sue Rao and Javed Chowdhury. Janice Singer has been instrumental in the studies of SE's. We also acknowledge fruitful discussions with others in the Consortium for Software Engineering Research, especially Ric Holt now of the University of Waterloo.

About the Authors

Timothy C. Lethbridge is an Assistant Professor in the newly-formed School of Information Technology and Engineering (SITE) at the University of Ottawa. He teaches software engineering, object oriented analysis and design, and human-computer interaction. He heads the Knowledge-Based Reverse Engineering group, which is one of the projects sponsored by the Consortium for Software engineering Research. Prior to becoming university researcher, Dr. Lethbridge worked as an industrial software developer in both the public and private sectors.

Nicolas Anquetil recently completed his Ph.D. at the L'Université de Montréal and is now working as a research associate and part time professor in SITE at the University of Ottawa.

The authors can be reached by email at tcl@site.uottawa.ca and anquetil@csi.uottawa.ca respectively. The URL for the project is http://www.csi.uottawa.ca/~tcl/kbre

References

[1] Gamma, E., Helm, R., Johnson, R and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995

[2] Holt, R., An Introduction To TA: The Tuple-Attribute Language, Draft, to be published. www.turing.toronto.edu/~holt/papers/ta.html

[3] Holt, R., Software Bookshelf: Overview And Construction, www.turing.toronto.edu/~holt/papers/bsbuild.html

[4] Lethbridge, T. and Singer, J, Strategies for Studying Maintenance", *Workshop on Empirical Studies of Software Maintenance*, Monterey, November 1996.

[5] Lethbridge, T. and Singer J., Understanding Software Maintenance Tools: Some Empirical Research, Submitted to: *Workshop on Empirical Studies of Software Maintenance (WESS 97)*, Bari Italy, October, 1997.

[6] Muller, H., Mehmet, O., Tilley, S., and Uhl, J., A Reverse Engineering Approach to Subsystem Identification, *Software Maintenance and Practice*, Vol 5, 181-204, 1993.

[7] Pree, W., *Design Patterns for Object-Oriented Software Development*, Addison-Wesley 1995

[8] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., *Object-Oriented Modeling and Design*, Prentice Hall, 1991

[9] Sayyad-Shirabad, J., Lethbridge, T. and Lyon, S, A Little Knowledge Can Go a Long Way Towards Program Understanding, *International Workshop on Program Understanding*, Dearborn, MI., 1997.

[10] Singer, J and Lethbridge, T, To be determined, Submitted to *CASCON 1997*

[11] Singer, J., and Lethbridge, T. (in preparation). Just-in-Time Comprehension: A New Model of Program Understanding.

[12] Singer, J. and Lethbridge, T, Methods for Studying Maintenance Activities, *Workshop on Empirical Studies of Software Maintenance*, Monterey, November 1996.

[13] Take5 Corporation home page, http://www.takefive.com/index.htm

[14] Vicente, K and Pejtersen, A. *Cognitive Work Analysis*, in press

[15] von Mayrhauser, A and & Vans, A., Program Comprehension During Software Maintenance and Evolution, *Computer*, pp 44-55, Aug. 1995