Copyright 2002 IEEE.

Published in the International Conference on Program Comprehension (IWPC)

June 27-29, 2002 in Paris, France.

Personal use of this material is permitted. However permission to reprint / republish this material for advertising or promotional purposed or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE.

Contact
Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / PO Box 1331 / Piscataway, NJ 08855-1331, USA.
Telephone: + Intl. 908-562-3966.

Program Comprehension Experiences with GXL; comprehension for comprehension

Claire Knight and Malcolm Munro
Visualisation Research Group,
Research Institute in Software Evolution.
Department of Computer Science,
University of Durham,
Durham, DH1 3LE, UK.
{C.R.Knight, Malcolm.Munro}@durham.ac.uk

Keywords: Future Research, GXL, Tools, Experience Report

Abstract

Tools are vital to support the various activities that form the many tasks that are part of the program comprehension process. In order that these tools are used and useful it is necessary that they support the activities of the user. This support must complement the work methods and activities of the user and not hinder them. Whilst features of good tools have been identified tool builders do not always adhere them to. It is important to consider whether needs have changed, and if those desirable properties need augmenting or revising. From experience of maintaining and enhancing an existing program comprehension tool for the purposes of participating in a re-engineering activity many lessons on tool support have been learned.

1. Introduction

All too often the theories, techniques, and tools of a computer scientist are developed in isolation from reality. The true value and applicability of such advances can be diminished on their introduction to reality, if indeed that transition is ever made. Program comprehension occurs in maintenance and development activity. The approach to comprehension taken depends on the task at hand. Despite the importance of program comprehension to such a diverse and wide range of software activities there still remains much work to be done to improve the tools and refine the techniques that exist today. Software continues to increase in size and complexity. Whilst program comprehension theories may support this growth, the tools and techniques developed for helping maintainers have not kept up with the speed and size of change. In developing new and novel solutions to these problems there is a perception problem, whereby the differences from the old solutions are highlighted. Unfortunately these are often not the differences that potentially improve the tool, but the new features that are unfamiliar. Experience in carrying out program comprehension activities during the development of a re-engineering project has highlighted many such issues. There are many tools that are deficient in many respects when presented with real world software.

In order to talk about the support necessary in program comprehension tools it is first necessary to look at the range of theories that exist about how programmers carry out various tasks during the comprehension process. Once key features have been identified the support most needed can be incorporated into new tools. This paper sets out to do this. Various program comprehension strategies are introduced. The use of GXL and involvement in the SORTIE project are presented with reference to the tool being adapted and used. Details of the changes made are given to illustrate the support desired. These all feed into the final sections of the paper that discusses the sort of support that tools should provide, current tool deficiencies, and some of the ways in which these could be addressed.

2. Program Comprehension Overview; Strategies, Methods and Processes

Program comprehension is an important part of not only software maintenance, but also the entire software

engineering process. Program comprehension is carried out with the aim of understanding an existing piece of code. It is a gradual process of building up the necessary understanding by examining sections of the source code. Using the knowledge gained from the source code explanations and understanding can be built and refined. According to Biggerstaff et al. [1] this process of discovery and refinement is known as the *Concept Assignment Problem,* whilst several other program comprehension strategies have different terms or processes to describe the same activities (such as bottom up comprehension). An overview of many types of program comprehension can be found in Robson et al. [2] and Von Mayrhauser and Vans [3].

There is not the space to provide an exhaustive survey of program comprehension strategies. Instead, the salient ones are listed with brief information and references that provide much more information for the interested reader.

2.1 Top Down (Hypothesis Driven)

Brooks [4] proposed a top down theory of program comprehension that centred on beacons as knowledge structures. His theory is hypothesis driven and he theorises that programmers use increasingly specific hypotheses to derive the functionality of the code. The programmer then has to verify (or reject) these hypotheses through examination of the code and then refining those hypotheses as necessary. Soloway and Ehrlich [5] observed a top-down approach to comprehension by expert programmers when dealing with familiar code. A mental model is constructed by forming a hierarchy of goals and programming plans. Rules of discourse are then used to break down goals into lower levels and sub-goals.

2.2 Bottom Up

Bottom-up comprehension is based on the concept of building up understanding from the bottom. By reading source code and then mentally building these smaller pieces of information into higher-level abstractions. Pennington [6] suggests that programmers gather various sorts of information from program code and that these differing sorts of information have different mental representations. Following empirical studies of this method Pennington concluded that knowledge is initially built up at lower levels than functions. The results suggest that control flow information is acquired before detailed function information is added to the programmers' knowledge of the code.

2.3 Knowledge-Based

Letovsky [7] carried out some empirical studies of programmers understanding code and from this developed several theoretical strategies about hypothesis generation and verification. These were

- Questions
- Conjectures
- Inquiries

There are five types of question; Why, How, What, Whether, and Discrepancy. Conjectures were defined to be

"any plausible inference about the program"

From analysis of the empirical data, conjectures were split into two; content and certainty. Content conjectures are defined as why, how, what and word (where word is a subtype of what and based on meaningful program identifiers) whilst the certainty conjectures are defined as guesses or conclusions. An idealised inquiry is based around questions, conjectures and then searches of the code. Letovsky suggested that programmers are opportunistic and exploit either bottom-up or top-down comprehension strategies as needed.

2.4 As Needed/Goal Directed and Systematic

Based on the results of experiments carried out, Littman et al. [8] speculated that there were two types of strategy employed when comprehending existing source code; as-needed and systematic. The as-needed approach is based on the localised understanding of areas of the source code thought to impact and be impacted by a change. Those areas of the code that do not fall into the container of impact are not considered during the comprehension process. The level of knowledge achieved is based on a subjective judgement as to what is necessary by the maintainer. The systematic strategy suggests that the entire program code be understood before any changes are attempted.

In addition to the two strategies, Littman et al. [8] suggest that there are two forms of knowledge; static knowledge and causal knowledge. Static knowledge is knowledge gained from an analysis of the source code in it's textual non-running form. Causal knowledge covers the interactions between the various parts of the software, often when it is running. The authors also divide the mental model created by the programmer into weak mental models and strong mental models. Weak mental models contain only static program knowledge and are built by programmers using an as-needed

strategy. Strong mental models contain not only static program knowledge but also causal knowledge about the program. Programmers who use the systematic strategy for understanding build strong mental models, although the authors acknowledge that it is unrealistic for many real systems to even attempt to obtain complete systematic understanding.

2.5 Syntactic, Semantic and Plan Knowledge

According to Schneiderman and Mayer [9] program comprehension is the process of forming internal semantics about the program under consideration. This information would be represented in a range of abstraction levels, from an overview of the program's operation down to the function of a small piece of code. The authors then presented the knowledge required for this process as being split into semantic and syntactic. Semantic knowledge is domain and experienced based, such as general programming concepts whilst syntactic knowledge deals with the actual code statements required to achieve a given task.

Basic plans can be seen as program fragments of stereotypical code that achieves a simple, single, goal. Programs are therefore plans containing several plans (which may themselves contain other plans). Soloway and Ehrlich [5] suggest that expert programmers have knowledge not only of these plans but also of rules of programming discourse. These rules specify programming conventions and therefore set up expectations in the minds of programmers. The results obtained by these authors from experiments agree with their suggestions that plans are used by expert programmers during the comprehension process

2.6 Summary

Comprehension through a mixture of top-down and bottom-up strategies is now accepted. Studies carried out by Von Mayrhauser et al. [16] showed that programmers frequently switch between the levels of abstraction that they are working at and are primarily concerned with what the software does and how it is accomplished. Their studies show that cross-referencing of information from many sources is required and carried out by programmers when they are trying to understand program code.

The integrated model has four major components:

1. program model,
2. situation model,
3. top-down model (domain model) and
4. knowledge base.

The first three of these are the comprehension processes whilst the fourth is necessary in successfully building the previous three. The top-down model is usually invoked if the code is familiar where hypotheses are the driving force of cognition (Brooks [4], Letovsky [7]). If the code is new to the programmer then the program model is built up first (defined by Pennington [6] as the control flow). Once this basic program model exists then the situation model is developed. This again works from the bottom up and involves the mental creation of a dataflow abstraction.

A programmer builds an understanding of the system through the creation of a mental model of that system. This model is built from the scarcest information and then refined as more of the code is examined and placed in context. Wiedenbeck [10] writes that this initial orientation phase (the basic mental model) is important because it allows the basic goals and operations of the program to be structured, and provides a framework for a more detailed study of the program. Burd et al. [22] describe this process as gradually piecing together the software puzzle. As Davis [11] summarises, the information gathering process is significant in forming a mental representation. He also writes about the programmer trying to solve a puzzle because of the non-linear comprehension that has been shown to take place. Non-linear comprehension requires that related information can be freely navigated and is not restricted to a strict one directional fixed flow of information.

3. GXL, SORTIE and GraphTool

Real world applications of program comprehension research (for whatever purpose) allows for a true test of theories and tools. In the case of many tools they are not necessarily used as part of the rollout of a commercial product because of the prototypical nature of the tool code. In a recent collaborative project (SORTIE) the goal was to apply theories and tools with some industrial C++ program code, with the ultimate aim of providing re-engineering suggestions. It was also a forum in which the use of GXL as a means of tool interoperability could be explored. This section provides some background information on GXL, SORTIE, and the tool used by the authors as part of this effort. The next section discusses the program comprehension, maintenance and evolution of the tool in more detail. For those unfamiliar with XML, the following terms are used in the rest of the paper:

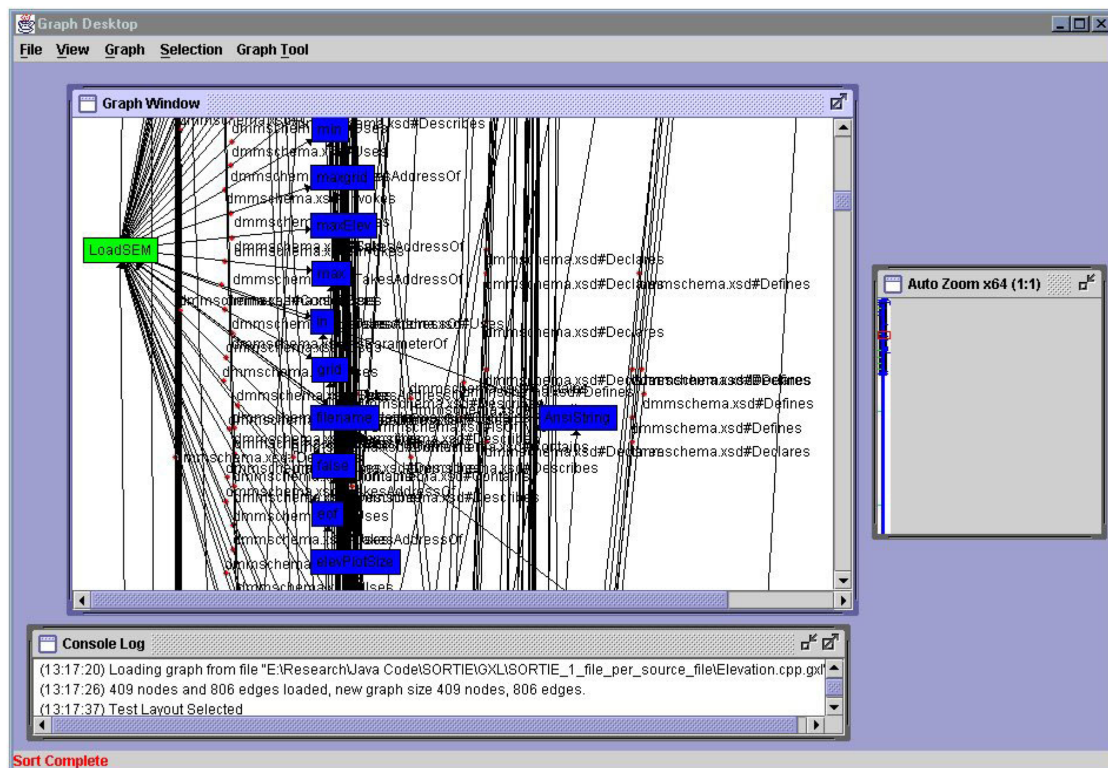

Figure 1 - *GraphTool* application screen with a SORTIE graph loaded

- **DTD** Document Type Definitions. A set of rules that define how XML data should be structured. DTDs describe the structure and syntax of an XML document. XML Schema are a more advanced way of achieving similar objectives except Schema are written in XML wheras DTDs have their own grammar
- **SAX** Simple API for XML. This is the *serial access* protocol for XML that is fast to execute. It is event driven whereby the parser invokes callbacks when tags are encountered in the source XML.
- **DOM** Document Object Model. This *random access* protocol converts an XML document into a collection of objects which can be visited at any time. The data structure can be manipulated as any other.

3.1 GXL

GXL (Graph eXchange Language) [13, 14, 20] is a standard exchange format for graph-based tools. It has been defined as a sub language of XML. GXL offers a general graph model that provides support for exchanging most types of graph. GXL deals with both instance graphs and any corresponding graph schema. The graph schema represents the graph structure, the definition of node and edge types and expected or supported attributes. This is possible because of the use of XML.

3.2 SORTIE

SORTIE [18] is a collaborative demonstration of reverse engineering tools. These tools are used in combination to solve a reverse engineering task on a legacy software system (SORTIE). The source code is from an established research tool for modelling forest succession, and actively used in British Columbia, Canada, and North-Eastern USA. There is approximately 28000 LOC of C++ with very little documentation. Evolution has occurred over time leaving the software with a brittle architecture, and hidden complexities, hence the need for re-engineering. There are several aims to this project, aside from ultimately helping to successful reverse engineer the software through advice and guidance. Many of these goals are tool oriented.

The only way to improve on existing tools and theories is to examine their usage in real situations. This project provides just such an opportunity; existing tools can be evaluated, and new and improved ones

developed based on the empirical evidence of their usage. There is also much to be learnt about the compositional nature of tools, and their integration, information sharing, and different uses for given tasks. SORTIE also allows for better tool evaluations [19] to be developed that can better reflect the tasks that the tool is being used for. It also means that evaluations can be made using like-for-like results rather than generalised ones.

3.3 GraphTool

The tool that has been used for this work is "*GraphTool*". This is a graph layout tool that has been used internally within the Computer Science Department at the University of Durham for several years. It was (re)written in Java in 1999 from the original UNIX based C. Some necessary re-engineering was carried out because of the move from procedural C code to object oriented Java code. The implementation of the GUI was also changed from the widget style used on a lot of UNIX platforms to the Java Swing libraries. *GraphTool* is now approximately 25KLOC in size, an increase of 5KLOC, and consists of 86 files. Since that time GXL has become a way of sharing information that is pertinent to software analysis and support for this was deemed a necessary addition to the tool. A screen shot of the *GraphTool* application can be seen in Figure 1 after the GXL changes had been made. The source file used to generate the graph that is shown is one of the analysis files from SORTIE.

As a summary, *GraphTool* supports or includes the following features:

- Reading/writing of four file formats; .2dg, .gxl, .gin, .cll
- Two layout automatic layout algorithms
- User controlled dragging of nodes and edges
- The use of various colours for nodes and edges
- The use of names on nodes and edges
- Anonymous setting to obscure real names on graphs
- Generation of postscript output of a graph
- Generation of JPEG images of the graph
- Various analysis assistance such as grouping of nodes (not supported when saving to all file formats)
- 100% view of a graph with a zoom/context window to also show the complete graph

GraphTool does not support particularly sophisticated automatic layout algorithms at this moment in time. Whilst that is a possible general area of development at Durham, the further focus of this research is to incorporate the GXL code developed and graph data structures into various visualisation tools within a framework. *GraphTool* itself will then need some modifications to fit into that framework. The framework was presented by Knight and Munro [12]. Since that time the interfaces of this framework have evolved to accommodate technology advances, but there are still active research areas about the metaphors, connectivity, and true autonomous brokerage being addressed.

4. Task and Process

This section details the program comprehension activities and tools used in the comprehension of *GraphTool* tool. This has been done because the task carried out was to incorporate GXL support into *GraphTool*. This shows the process of comprehension on a real system, where that system itself was being used to analyse another real world system. This process will also help to illustrate how *GraphTool* can be of use for other analysis and also to identify shortcomings that future visualisation and program comprehension tool research should seek to address.

4.1 Extending *GraphTool* with GXL

Much of the work done in carrying out this task has involved the incorporation of GXL into the tool so that it can read and display GXL graphs. However, there is also a need to generate GXL files from the graph stored in memory regardless of original format. This means that the tool is then also capable of acting as a conversion agent between the file formats that it supports. The tool already supports three other file formats, all proprietary to Durham. One of these file formats was introduced when the software was converted to Java. It is quite verbose and includes various attributes for layout/colour as well as node and arc values and connections. Some of the attributes of this format that deals with graphics were included into the GXL generation from *GraphTool*. This is so that they can be used when reloading a saved graph into *GraphTool* or into other analysis tools. It essentially means that layout and colour attributes can be preserved if the tool reading them in chooses to make use of them.

Because *GraphTool* uses its own internal data structure to represent the nodes and edges, the integration of GXL into the tool required code to be added that converted from the XML to this data structure. Loading the information had to populate the

graph structure, and saving meant walking the data structure to output the necessary information. This influenced a design decision to use SAX parsing of the GXL file. GXL is based on XML and has a DTD specifying the valid structure of tags and data. Java tools exist for providing both SAX and DOM parsers of a file. The DOM approach creates a data structure of the tree of tags (and their values), whereas the SAX approach upon finding start and end tags operates a call-back feature providing identification of, values of and attributes of that tag. The cleaner approach from a parsing point of view is the DOM approach because a walk of the tree created using semantic knowledge allows for appropriate conversions and actions to be taken. However the creation of a secondary data structure in memory was considered inefficient, thus the decision to use SAX was made. This choice had interesting repercussions when adding semantic code to the call-back methods of the parsing code and when loading multiple graphs particularly there is a need to explicitly save the intermediate parse state.

A restriction imposed by the other file formats of *GraphTool* was that only one graph at a time could exist in memory. Disjoint nodes were considered to be just that, and not sub or related graphs. Since GXL supports multiple nested graphs some additions were made to the data structure to support this notion. This means that GXL loading (and saving) is information preserving within *GraphTool*. The other file formats support as much as they were originally intended to and have not been altered because of this change.

The GXL file containing all of the SORTIE analysis is approximately 34Mb in size. Partly because of the object based data structures used by *GraphTool* (i.e. not as efficient as low level C) and the temporary storage necessary when analysing a GXL file this analysis fails to finish loading. The ability to have an overall view provided by this file would have proved useful from an analysis perspective. No analysis carried out so far has provided a high level view in which to begin to hang any detailed analysis off. This provides justification for filtering of information. The loading/saving operations may have to be further changed to reduce the impact on memory, but the use of different views would enable just the higher-level nodes to be viewed in the first instance.

Since Java is an object-oriented language, there are features of the existing implementation that at first sight appear to be very clean. One example of this is the use of Node and Edge classes. This means, for example, each node knows what value it has, its name, some identification code and so forth. It also knows how to draw itself and therefore contains its graphical size and colouring attributes. There are also many other things on which a node can be asked to act such as saving itself, or having attributes added to it. In this way each node is very much a self-contained entity, and the graph just has to keep track of nodes and edges. On the other hand this creates some coupling issues.

Because of the GXL file format, loading is done via the XML parsing code, which creates (for a node) a new node and sets the necessary attributes accordingly. Saving is another matter, and has been implemented in a similar way to the other file formats. The graph saves the high level information it contains and then calls on the nodes and edges it knows about to save themselves. The higher-level routines are then not cluttered with node detail, but the level of dependence on a file format of many classes is high. To save a GXL file requires the use of (aside from menu options in user interface classes) over five different classes. Deciphering the flow of information and method calls is simple and logical although hand traces of calls are needed to find this out. There is a flow of information from the higher-level classes to those representing smaller conceptual entities. Discovering this flow of information, and which auxiliary methods at the same level of class structure are used, was a time consuming process.

It is interesting to note at this point the types of activities necessary to make these changes, and also which comprehension and software engineering tools have been used. The (re)development of *GraphTool* has involved, on the Windows/PC platform, the use of a text editor and a DOS box in conjunction with the standard Java development kit (jdk 13.1)as provided by Sun. It should be noted at this point that this is essentially the minimum requirements for such a project. There has been no use of integrated environments, graphical debugging tools, and certainly no use of any program comprehension tools. The first few of these relate more to the preferences of the first author, and the lack of good, affordable tools for Java development on the Windows platform that can be suitably tailored to her way of working. It is also a legacy of having developed in Java since its inception when this was the only way of creating applications. The more important issue in the context of this paper is that no program comprehension tools have been used. A lengthier discussion as to some of the reasons why this practice is common can be found in the latter sections of this paper. One of the main reasons are that tools are often language specific and there are relatively few for Java compared to other

languages. Other are the availability of the tools, and also the knowledge required to use them.

One tool that was used for syntactic debugging, and thus for a form of program comprehension was the Java compiler. This was used several times in place of a grep like tool. The use of the compiler was considered more than adequate because of the context that it can provide. String matching of method names (for example) is not always of use when the object that is being referred to is also important. Some changes made to the original *GraphTool* source code were to make it more object oriented and to hide variables and methods by changing access modifiers (public to private for example) and then providing access methods. Making changes to a few classes caused large impact changes on the rest of the code. In this instance the compiler was a great help. It is also worth noting here that many program comprehension tools (admittedly not all) require that the source be at least compilable. In this instance this would mean that the same method of assessing and repairing impact would have had to have been used. An example where integrated environments are improving is that they do support object hierarchy trees, and provide syntax highlighting that works (most of the time!) with incomplete source code.

The Java analysis coverage is currently quite good because of the amount of change that has taken place over the last few months. However there are still many enhancements that could be incorporated to create a better program comprehension aid. Some of these relate to the ability of the tool to group nodes. These groups (or nested graphs as far as GXL records such clusters) can be created but have to be done from the ground up at the moment. Once created they can only be removed or aggregated into higher-level clusters. The ability to move nodes around clusters and to remove some of the nodes from a cluster would greatly benefit exploratory analysis procedures. Other analysis support would be an improved layout, the use of schemas for filtering GXL graphs, and allowing annotations to add information to nodes and/or clusters. GXL would be able to preserve this information by saving it as an attribute.

4.2 *GraphTool* Experiences in SORTIE

Having carried out the task of changing *GraphTool* to support GXL, the process of analysing the SORTIE system could begin. This required the loading of the GXL files that had been generated from the C++ source code. A decision was made to use the GXL output of the parser developed as part of other research work [17]. This consisted of information about the methods and classes contained in the source code. By sharing the data in this way the idea of tool interoperability is both supported and demonstrated.

Because *GraphTool* is a generic graph application then there is no particular layout or analysis incorporated into the tool for use when looking at software and systems. This meant that the GXL graphs loaded with all nodes defaulted to a position of x=0, y=0. Automatic layout provided a start, but to get some of the better layouts (on the smaller graphs) then hand-layout was used. For the larger graphs it was decided that it was easier to leave the layout as suggested by the automatic algorithms, even if there were lots of crossed lines and so forth. This decision was made for initial ease of use, but was quickly found to cause analysis problems, primarily because of the number of nodes and arcs in the graphs.

There are a lot of **type** nodes in the GXL used for this analysis. Whilst the importing of the information makes this distinction from named nodes through setting of the node colour, it may be that the removal of some of these (or a reduction in the number of edges) may make for much clearer graphs. As can be seen in Figure 1, many of the edges are labelled as starting with dmmschema#. This means that if stylesheet support was included in *GraphTool* that it would be possible to remove some of these edges through representing the information in another way. GXL, depending on its generation, can be very information rich, and the use of stylesheets can provide a way of filtering that information into manageable views. Different transformations (via the stylesheets) provide different views of the same information. These can be used in combination to support the various program comprehension activities of any user.

Because of the size of the SORTIE system (and therefore directly the graphs) and the non-specific nature of *GraphTool* the analysis is quite time consuming. However, these were both anticipated. It perfectly illustrates the need for using a variety of views, visualisations, aggregations, and analyses in order to achieve understanding. Different visual metaphors, dimensions, and layouts can be combined with different schema style views in order to provide more complete understanding through management of the underlying mass of data.

Having spent time trying to clean up some graphs using *GraphTool* it became obvious that not only were different visualisations required to complement the

graph views, but also analysis aids at the graph level. When loading a GXL graph, *GraphTool* uses two colours for the node. All nodes are given their type as a name to start with. If any name is then provided as an attribute this replaces the type. At this point, the colour of the node is then set to green. The remaining nodes are left with the default node colour (specified by user preferences in *GraphTool*, although on saving a GXL file with graphical attributes this colour is saved for future use). This was helpful when looking at the graphs, because it actually showed (unintentionally) where collapsing nodes and joining edges, or other visualisations encoding the information, would be useful. A lot of the clutter in the graphs stemmed from nodes (and hence edges) that essentially provided some contains/declares information. Another drawback is that the types of the nodes and edges were all prefixed with `dmmSchema.xsd#` which as a reference into a specific part of that XML Schema file that was not available, was of little use. This may provide some layout assumptions when using tools that are able to incorporate stylesheets. It highlights the need for specialist processing for some graphs which may only be available in software visualisation specific tools (rather than a general graph oriented tool), or that should be provided in general tools in an abstract manner. It is possible to select nodes by hand and delete them, but it would have been preferable to have been able to search for all (for example) nodes of `dmmSchema.xsd#sourcePart` and then to collapse the nod to merge the edges, not remove them as the delete operation currently does. It also highlights the situations where other visualisations may automatically provide some of that analysis through aggregation of information to generate a particular glyph with colours, sizing, position, and orientation based on connectivity and attributes of that information item.

Because many of the graphs, and certainly those of any realistic size, were very cluttered when viewed this hindered any real analysis judgements about the SORTIE system. The only removal of clutter that was made easy was node deletion that also deleted edges. This then removed links that did and should have existed; thus making the understanding even harder!

4.3 Summary

An interesting outcome of trying to analyse SORTIE with *GraphTool* is that because the GXL file came from elsewhere (and no source code has been seen) no assumptions can be made. Any analysis relies solely on the GXL provided and any tools that can utilise it. This has proved an interesting exercise from a program comprehension perspective because of the way in which it highlights what tool support is necessary. Much of GXL provided is detailed enough to give enough information for recommendations, but the tool support does not make this particularly easy.

The previous two sections show the process of program comprehension from two perspectives. The first shows the processes and issues when making changes to a system. Thus the information needs are targeted towards specific aims; in this case identifying where to make the change and what sort of changes needed to be made. The latter illustrates the use of the tool that has just been enhanced being used for some general program comprehension activities where overall system understanding was the first goal. Obviously both have the benefit of the person carrying out the analysis knowing the tool being used to do it, but they both show that the current program comprehension tools have some inadequacies.

5. Lessons Learnt

Program comprehension is very much a gradual process where the maintainer gathers information through studying various aspects of the code at different times, and possibly by returning to previously examined pieces of code. This process is true regardless of the strategy employed to examine the various pieces of code that constitute the system. The process of linking together pieces of evidence and any relations between them (such as validating alibis) is a common process in detective work; essentially what is happening when trying to understand an existing piece of code. The main things that have been learnt are:

- There is need for flexible tool support
- Cognitive issues are important, but such guidelines need to take into account the tool support aims
- The importance of a variety of viewpoints, including supporting different levels of detail
- Cross referencing is valuable
- Live/interactive linking between related pieces of information can greatly enhance usability
- Java analysis tools are necessary

Generally the process of investigation is deductive because the programmers are not trying to create any new “axioms”. The main concern is to try and make sense of the information (or evidence) that they already have. This process of detective work points to the need to provide flexible tools that allow for the evidence

gathering and hypothesis refinement to be achieved in several ways, thus supporting many of the different strategies said to be employed during program comprehension. To be able to freely move between related pieces of information allows this knowledge discovery and clarification to proceed in a non-linear fashion. This should then enable the maintainer to work more easily simply by using the tool to follow their train of thought, or to put it another way, their line of inquiry.

Storey et al. [15] identified a hierarchy of cognitive issues that are important when considering what facilities a program comprehension tool should include. They identify the fact that software exploration tools can be likened to hypermedia document browsers. Because of this a hierarchy of hypermedia cognitive issues has been adapted to form program comprehension guidelines. Also identified is the lack of support in existing systems for the integrated and top-down models of comprehension and the inability to switch between different mental model information. Navigation and orientation cues were also identified as an area for future research. The work done by Chan and Munro [21] identifies the need to provide different viewpoints for maintainers. This allows them to choose the most appropriate view for the current task, and also to be able to switch between views to gain a higher or lower level understanding of some piece of information.

Von Mayrhauser et al. [16] suggest that cross-referencing of related areas of code would make identification of areas where changes need to be made easier. These cross-reference links should be, where possible, hypertext and also link to algorithm and/or domain information. They also identify the need to provide orientation cues in the documentation and propose the use of some form of browser history with on-line sticky notes to make this effective. They also think that documentation of the system (which could be included in any tool that was used to aid comprehension) should have a high-level *road map of the system structure*.

There is a need for tools that support Java. Some of the analysis issues that are problematic for C++ do not exist for Java because of the design of the language. Regardless of any possible difficulties it is important to realise that Java code is being used in industry as well as in academic systems, and this therefore creates comprehension and maintenance problems that need to be solved. A situation where it is common to find Java code in use industry that already needs maintenance is where it is based on legacy C code (for example). In cases such as these where the need was to upgrade the language used (however real or perceived that need) the applications have often been badly translated from their original implementations. Not only do non object oriented features get replicated in object oriented code, but any fragile parts of the system (due to prior bad maintenance) also get moved to the new system. This creates a system in instant need of understanding and repair.

In the tool demonstration documented in [19] several interesting results emerged. The first was that by seeing what their tools were able to do in a real setting provided insight for the developers of those tools. Another was of flexibility. If tools are too rigidly defined in their ways of working, or more precisely, in how they allow user(s) to work with them then they will receive little in the way of widespread acceptance. It was also highlighted how important it is to evaluate tools within given contexts. This is vital if a tool that is highly specialised is not to be penalised because of its inability in all of the other situations it is presented with. A final important point is that the analysis and comparison, and indeed the authors of this paper believe, use of tools should focus on different combinations of interoperable tools. In this way, a tool can be developed to provide good and thorough support for a particular sub-part of several tasks. It can then be reused in different situations as necessary, but it then does not fall into the problem of *jack of all trades, master of none* where in trying to satisfy all demands on its usage it manages to do none well.

6. Conclusions and Discussion

This paper has highlighted the use of program comprehension for both the analysis of a system to provide recommendations and also to make changes to the very system that was to be used to analyse the first. The activities show how many aspects of the current program comprehension tools are insufficient. This is not advocating that every research project or group should make it their business to create saleable and robust tools. In a research environment where finances do not relate to implementations of ideas and strategies, this does not make sense. However, incorporating useful and good features into prototypes will help to demonstrate to industry that program comprehension in the real world with real systems is a viable and sensible option.

Empirical evidence of two very small contributions to program comprehension in the large have highlighted that they are limitations to the existing tools, and shown that certainly for visualisations, other representations

and analyses that could be of use. Important questions to ask at this point are "what is it that we want?", "what do we want to see, and when?" and "what importance is the what, as well as the how, to our understanding?". This last question is very important based on the work described in this paper. Often the questions being asked of the code were more than just simple issues relating to compositions of classes.

Improving interoperability between tools means that independently specialised tools can be composed together to form sophisticated and powerful analysis suites for program comprehension, re-engineering, and maintenance. GXL has provided one way of doing this, and if the work in this project is continued successfully then it provides a means of allowing interoperability within tools at one site or collection.

Acknowledgements

This work is financed by an EPSRC ROPA grant: VVSRE; **V**isualising **S**oftware in **V**irtual **R**eality **E**nvironments and by Malcolm Munro through VRG.

References

[1] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, *Program Understanding and the Concept Assignment Problem*, Communications of the ACM, Vol. 37, No. 5, pp72-82, May 1994.

[2] D. J. Robson, K. H. Bennett., B. J. Cornelius, and M. Munro, *Approaches to Program Comprehension*, Journal of Systems and Software 14, pp79-84, 1991.

[3] A. Von Mayrhauser and A. M. Vans, *Program Comprehension During Software Maintenance and Evolution*, IEEE Computer, pp44-55, August 1995.

[4] R. Brooks, *Toward a Theory of Comprehension of Computer Programs*, International Journal of Man-Machine Studies, Vol. 18, No. 6, pp542-554, 1983.

[5] E. Soloway and K. Ehrlich, *Empirical Studies of Programming Knowledge*, IEEE Transactions on Software Engineering, Vol. SE10, No. 5, pp 595-609, September 1984.

[6] N. Pennington, *Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs*, Cognitive Psychology, Vol. 19, No. 3, pp295-341, July 1987.

[7] S. Letovsky, *Cognitive Processes in Program Comprehension*, Journal of Systems and Software, Vol. 7, pp325-339, 1987.

[8] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, *Mental Models and Software Maintenance*, Empirical Studies of Programmers, Ed. E. Soloway and S. Lyengar, pp80-98, 1986.

[9] B. Shneiderman and R. Mayer. *Syntactic/Semantic Interactions in Programmer Behaviour: A Model and Experimental Results*, International Journal of Computer and Information Sciences, Vol. 8, No. 3, pp219-238, 1979.

[10] S. Wiedenbeck, *The Initial Stage of Program Comprehension*, International Journal of Man-Machine Studies, Vol. 35, pp517-540, 1991.

[11] S. Davis, *A Guessing Measure of Program Comprehension*, International Journal of Human-Computer Studies, Vol. 42, pp245-263, 1995.

[12] C. Knight and M. Munro, *Mediating Diverse Visualisations for Comprehension*, Proceedings of the IEEE 9th International Workshop on Program Comprehension, May 2001.

[13] R. C. Holt, A. Winter, and A. Schürr, *GXL: Towards a Standard Exchange Format*, Proceedings of the 7th IEEE Working Conference on Reverse Engineering (WCRE 2000), pp162-171, 2000.

[14] A. Winter, *Exchanging Graphs with GXL*, Proceedings of Graph Drawing, 9th International Symposium (GD 2001), Springer Verlag, P. Mutzel (Editor), 2001.

[15] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller, *Cognitive Design Elements to Support the Construction of a Mental Model During Software Visualization*, Proceedings of the 5th IEEE International Workshop on Program Comprehension, pp17-28, May 28-30, 1997.

[16] A. Von Mayrhauser, A. M. Vans, and A. E. Howe, *Program Understanding Behaviour during Enhancement of Large-scale Software*, Journal of Software Maintenance: Research and Practice, Vol. 9, pp299-327, 1997.

[17] Sergey Marchenko, Tim Lethbridge (KBRE group); http://www.site.uottawa.ca/~tcl/kbre/

[18] SORTIE web site; http://www.csr.uvic.ca/chisel/collab/

[19] S. E. Sim, M.-A. Storey, and A. Winter, *A Strutured Demonstration of Five Program Comprehension Tools: Lessons Leant*, Proceedings of the 7th IEEE Working Conference on Reverse Engineering (WCRE 2000), pp184-193, 2000.

[20] GXL web site; http://www.gupro.de/GXL/

[21] P. S. Chan and M. Munro, *PUI: A Tool to Support Program Understanding*, Proceedings of the IEEE 5th International Workshop on Program Comprehension, pp192-198, May 28-30, 1997.

[22] E. L. Burd, P. S. Chan, I. M. M. Duncan, M. Munro, and P. Young, *Improving Visual Representations of Code*, University of Durham, Computer Science Technical Report 10/96, 1996.