Evaluating a Model of Software Visualisation for Software Comprehension

Michael J. Pacione

michael.pacione@cis.strath.ac.uk

June 2004

Department of Computer and Information Sciences
University of Strathclyde, Glasgow, UK
Abstract

This report discusses the process of evaluation in software comprehension, and applies it to a preliminary model of software visualisation for software comprehension. The techniques used to evaluate software visualisation and comprehension tools are discussed in the context of a number of examples from the literature. This discussion provides the basis for the evaluation of the proposed software visualisation model. We present a refined set of typical software comprehension tasks, based on our previous work [Pacione 2003a, Pacione 2003b], which we will use to evaluate the model. We then perform an initial evaluation of the proposed model to determine which aspects of it are most useful for software comprehension and hence most promising for further research. We also describe software comprehension strategies and discuss support for them in our model.
Keywords

Abstraction hierarchies, evaluation, facets, software comprehension, software visualisation, static and dynamic analysis
Acknowledgements

I would like to thank my supervisors, Dr Murray Wood and Dr Marc Roper, for many useful meetings, and for their helpful comments on drafts of this report.
Contents

Page i  Title page
Page ii  Abstract
Page iii  Keywords
Page iv  Acknowledgements
Page v  Contents
Page vii  List of tables

Page 1  1 Introduction
       1.1 The role of evaluation
       1.2 Our model of software visualisation for software comprehension
       1.3 Report structure

Page 3  2 Survey of evaluation techniques
       2.1 Globus and Uselton (1995)
       2.2 Murphy et al. (1996)
       2.3 Bellay and Gall (1997)
       2.4 Armstrong and Trudeau (1998)
       2.5 Storey et al. (1996)
       2.6 Sim and Storey (2000)
       2.7 Sim et al. (2000)
       2.8 Storey et al. (2000)
       2.9 Bassil and Keller (2001)
       2.10 Hatch et al. (2001)
       2.11 Knight (2001)
       2.12 Kollmann et al. (2002)
       2.13 Summary

Page 13 3 Evaluation based on representative tasks
         3.1 The basis for typical software comprehension tasks
         3.2 Typical software comprehension tasks
            3.2.1 General software comprehension tasks
            3.2.2 Specific reverse engineering tasks
         3.3 Task set analysis
         3.4 New task sets
            3.4.1 General software comprehension tasks
            3.4.2 Specific reverse engineering tasks
            3.4.3 Justification
         3.5 Summary

Page 20 4 Evaluation of the proposed model
         4.1 Model information required to support typical software comprehension tasks
         4.2 Summary
5 Validation of support for software comprehension strategies
  5.1 Software comprehension strategies
      5.1.1 The bottom-up model
      5.1.2 The top-down model
      5.1.3 The knowledge-based model
      5.1.4 The systematic and as-needed models
      5.1.5 The integrated model
  5.2 Object-oriented software comprehension
  5.3 Comprehension in software visualisation
  5.4 Support for software comprehension strategies in our proposed visualisation model

6 Summary, conclusions, and future work

References
List of tables

Page 19 Table 3.1  The correspondence between typical software comprehension activities and the revised task sets

Page 20 Table 4.1  Information required from each dimension of the proposed model to address the general software comprehension tasks

Page 21 Table 4.2  Information required from each dimension of the proposed model to address the specific reverse engineering tasks
1 Introduction

1.1 The role of evaluation

Evaluation is a crucial component of any project. In order to assess the results of the project we require evaluation criteria, by which we will determine what we have achieved. Without evaluation, it is impossible to state whether the project was a successful or not, or to draw any conclusions from the results. Before embarking on a project, the criteria by which it will be evaluated should be established. This provides a predetermined basis for assessing the project on completion.

Empirical evaluation is evaluation consisting of experimentation, rather than purely theoretical analysis [Basili 1996, Perry 2000]. Empirical studies are useful for testing hypothesis, as they provide experimental data that can be analysed. Empirical studies are often used in software engineering to perform more realistic evaluations than are possible with purely theoretical methods.

This report will discuss a number of methods for the evaluation of software visualisation and software comprehension tools. The techniques discussed are also more widely applicable in other areas of software engineering and beyond.

1.2 Our model of software visualisation for software comprehension

Previous work revealed that current software visualisation tools do not address the full range of software comprehension requirements [Pacione 2003a, Pacione 2003b]. Most current tools support only a very limited range of abstraction levels, lack multiple perspectives of the software, and do not integrate static and dynamic (runtime) information. These limitations have the result that current tools cannot support the majority of typical software comprehension activities. In order to address the deficiencies in current tools, we are proposing a new model for software visualisation that integrates multiple interrelated levels of abstraction, various perspectives of the software system, and a combination of static and dynamic information [Pacione 2004a]. A summary of this work is available [Pacione 2004b]; a brief overview is also given [Pacione 2004c]. The aim of this work is to improve the effectiveness of software visualisation techniques for large-scale, real-world software comprehension.

An example of the type of problem that we seek to address is the case of investigating the effect of making a change to one part of a software system on the rest of the system. In order to answer this question, information on the software at various levels of abstraction is required, in order to analyse the system at the appropriate levels of detail. Several different perspectives of the software are also needed, illustrating, for example, the system’s structure, behaviour, and data structures. Finally, a combination of static information from the program code and dynamic runtime information will enable a comprehensive and detailed analysis.

In this report, we will evaluate this model to determine which aspects are most useful for software comprehension, and hence most promising for further research, and
assess its support for software comprehension strategies. Future work will involve refining and evaluating this model to determine its usefulness in facilitating software visualisation for comprehension.

1.3 Report structure

This introductory section has described the role of evaluation and our proposed model of software visualisation for software comprehension. Section 2 presents a survey and discussion of evaluation techniques for software visualisation and software comprehension tools. Section 3 discusses evaluation using representative tasks, and describes and refines the evaluation task set used in our previous work [Pacione 2003a, Pacione 2003b]. Section 4 analyses the proposed visualisation model to determine which aspects are most useful in supporting software comprehension, and hence most promising for future research. Section 5 describes software comprehension strategies and discusses support for them in the proposed model. Section 6 summarises the report, draws conclusions, and presents future work.
2 A survey of evaluation techniques

Previous work presented a review and evaluation of fourteen visualisation tools [Pacione 2003a, Pacione 2003b]. Evaluation of visualisation tools is often superficial or neglected altogether. Where it is performed, the evaluation typically consists of reports on qualitative case studies performed by the authors on small or medium sized systems. A number of evaluation techniques for the evaluation of software visualisation and comprehension tools have been described in the literature. This section discusses that work.

2.1 Globus and Uselton (1995)

Globus and Uselton discuss the evaluation of scientific visualisation software [Globus 1995]. Although their analysis is concerned with modeling physical systems, such as the field of computational fluid dynamics, there are important points that are relevant to software visualisation. They observe that visualisation is becoming widely used, and that for a visualisation to be useful it is important to be able to evaluate it. However, they also note that there is a lack of evaluation in the visualisation community at the time of writing.

Firstly, they discuss the need for standardised test suites consisting of test data and a set of tests of particular visualisation techniques and functions. The concept of a set of standard tests for visualisation tools is comparable to the evaluation performed in our earlier work, where we used a set of representative tasks to evaluate tool performance [Pacione 2003a, Pacione 2003b]. In the case of software visualisation, the test data would be provided by a representative sample of systems to which the visualisation would be applied. The wider the range of the systems, the more generally applicable and useful the validated visualisation is likely to be.

Secondly, Globus and Uselton highlight the importance of the effect of error in visualisation. Errors should be minimised and it is important to characterise (recognise and, where possible, quantify) any error in order to provide an accurate visualisation. Although the potential for error is less in software visualisation than in the visualisation of continuous physical systems, it is important to recognise the possibility of inaccuracies in the visualisation (e.g. caused by an inappropriate or misleading abstraction technique).

Finally, Globus and Uselton discuss the evaluation of visualisations using human subjects. They argue that as the purpose of visualisation is to improve human insight into data, humans are best suited to evaluate the performance of a visualisation in achieving this. Although insight cannot be measured directly, task performance can be used an indicator of this. They state that is it easier to perform experiments that compare two visualisation systems than experiments intending to evaluate or characterise a single system. Given the experimental results, it may be possible to predict performance in related tasks, and to predict the effect of making changes to the visualisation system. As with all subject-based empirical evaluation, the choice of subjects is crucial and must be representative of the intended user base of the
visualisation. For example, in the case of software visualisation, industrial software engineers engaged in software maintenance with, say, two years experience of using visualisations might be an appropriate evaluation subject group.

2.2 Murphy et al. (1996)

Murphy et al. describe an evaluation of five static call graph extractors [Murphy 1996, Murphy 1998]. The tools analysed were cflow (a standard Unix tool), CIA [Chen 1990], Field [Reiss 1995], mkfunctmap [Hoagland 1995], and rigiparse [Müller 1988]. These tools were chosen as they are readily available, extract calls from C code in textual form, and all run on the same platform (SunOS on a Sun SPARC). The aim of the evaluation was to compare both quantitatively and qualitatively the call graphs produced by the tools. Three C systems were used for the case study: mapmaker (a molecular biology application) [Lincoln 1993], mosaic (a web browser) [NCSA 2003], and ggc (the GNU C compiler) [GNU 2004]; these were intended to represent a variety of application domains.

Call graphs were generated for each of the three applications by each tool. The results were then compared quantitatively (pairwise) to determine the number of calls detected by both tools, and by one tool but not the other. Of course, a higher number of calls detected does not make one tool better than another. The differences in calls detected are attributable to the analysis algorithms employed by the tools (which are often not elucidated). The results were also sampled and qualitatively analysed to assess the numbers of false positives (a call detected where one does not exist) and false negatives (a call not detected where one does exist) in the results. It was determined that all of the tools generate both false positives and false negatives. It appears that the study was conducted by the authors. The paper concludes with discussion on the design and use of call graph extractors.

The use of a number of different application types makes both the experimental procedure and results more generally applicable than if only a single system, or type of system, were considered. The use of objective evaluation criteria (number of calls detected, number of false positives, etc.) also increases the generality of the study.

2.3 Bellay and Gall (1997)

Bellay and Gall describe an experiment to evaluate the capabilities of four reverse engineering tools [Bellay 1997, Bellay 1998]. The tools analysed were Refine/C [Reasoning 1994], Imagix 4D [Imagix 2004], Rigi [Müller 1988], and Sniff+ [Wind River 2003]. The aim of the case study was to investigate the capabilities of the tools, and identify their advantages and disadvantages in terms of applicability to embedded software, usability, and extensibility. An industrial embedded train control system was used for the case study, containing approximately 150 KLOC (thousand lines of code).

The assessment criteria were expressed as a checklist, which was formulated based on the authors’ experience in applying the tools during the case study. The consequence of this is that the checklist is likely to be biased towards the system and tools analysed.
in the case study. The checklist was delineated into four categories: Analysis, Representation, Editing/Browsing, and General Capabilities. The Analysis category is concerned with the functionality and performance of the parser. The Representation section is concerned with the features of the representation used and how quickly it is generated; for textual reports, sorting is examined while for graphical reports the view type and editing facility is examined. The Editing/Browsing category is concerned with text editor integration and speed, and other user interface facilities, such as search and history functions. The General Capabilities section covers support for multiple platforms and users, extensibility, and storage, output, history, search, and help facilities.

The study finds that the different tools are best suited to different usage contexts. Some benefits and shortcomings of the extant reverse engineering tools are identified. The general result is that tool performance and capabilities depend on the case study and application domain as well as the purpose of the analysis. It appears that the evaluation was carried out by (one of) the authors. It is not clear exactly what was done in terms of analysing the software system in order to exercise the tools’ functionality.

2.4 Armstrong and Trudeau (1998)

Armstrong and Trudeau perform an evaluation to compare the functionalities of five architectural extraction tools [Armstrong 1998]. The tools analysed were Rigi, Dali [Kazman 1999], PBS [Finnigan 1997], CIA, and SNiFF+. The aim of the evaluation was to assess the extraction, classification, and visualization features of the tools. Two C systems were used for the case study: the CLIPS expert system tool [Riley 2003], and a small test program that was intended to be problematic for the tools to parse.

Similar to the study by Belay and Gall [Bellay 1997] described in Section 2.3, the evaluation criteria were in the form of a checklist. The criteria were devised based on the authors’ experiences of using the tools. As in the Bellay and Gall study, this makes it likely that the assessment criteria will be biased towards the particular tools and systems used for the case study, hence making both the results and the assessment criteria themselves difficult to generalise.

The Extraction checklist assessed the functionality of the tools’ parsers, such as the exclusion of library calls, the contents of C structs, and recursion. The Classification evaluation did not have an explicit checklist associated with it, but was based on using the tools to generate meaningful abstractions from the extracted data. The Visualization checklist was concerned with issues such as the types of nodes and edges available, and the navigation functionality.

Similar to the Bellay and Gall study, it was found that various features from the different tools are useful but no one tool integrates all of the features that would be desired. Again, it appears that the evaluation was carried out by (one of) the authors. It appears that general exploration of the software systems was performed in order to exercise the tools’ functionality, though there is little detail about this.
2.5 Storey et al. (1996)

Storey et al. [Storey 1996a] describe the preparation and execution of an empirical study to assess the usabilities of two interfaces to the Rigi reverse engineering tool [Tilley 1994, Wong 1995]. The aim of the study was to compare the interfaces to each other and to standard Unix command line tools (vi and grep). Three C game programs of similar complexity but varying size (300-1700 LOC) were used in the evaluation. Storey et al. evaluated the usability of the user interfaces by observing users completing a set of software maintenance tasks followed by a questionnaire and an interview. This technique is similar to that discussed in our previous work as it evaluates a tool by assessing its performance in a series of typical tasks [Pacione 2003a, Pacione 2003b]. However, Storey et al. used a group of twelve volunteers to evaluate the tool, while the evaluation described in our work was carried out by a single user. The small tasks involved in the Storey et al. study were intended to be typical of those performed by software maintainers working towards a larger goal; a trade-off was necessary between experiment time and task complexity. The tasks were divided into two groups of four tasks, ‘abstract’ and ‘concrete’, which were concerned with high- and low-level understanding, respectively. These task groupings are similar to the large- and small-scale tasks used in our previous work to typify general software comprehension and specific reverse engineering tasks, respectively, though the tasks used by Storey et al. were more straightforward (e.g. ‘Find an artefact that is not used’). The importance of experimental setup is stressed; a ‘dry run’ was conducted in advance, which helped to refine the experiment. The subjects were given training in each of the interfaces beforehand. In addition to the questionnaire and interview, the participants were observed performing the tasks. Appropriate statistical tests were applied to the results. In addition to useful results in terms of the relative usabilities of the tool interfaces, the paper also identifies a number of improvements to the experiment, namely the need for a larger user group, more tasks, longer time, and greater experimental control.

2.6 Sim and Storey (2000)

Sim and Storey describe a structured tool demonstration in which several reverse engineering tools were evaluated using a common software system and set of analysis tasks [Sim 2000a]. The aim was to provide a fair comparative demonstration of the capabilities of the tools. They argue that tool evaluations in the literature tend to be ad hoc, tools are rarely evaluated formally by users, and when they are evaluated it for only a short time by people unfamiliar with the tool. Potential users often assess tools on superficial observations, such as appearance or feature set, rather than more objective factors, such as ease of use or scalability. While useful, the results of case study evaluations are often difficult to generalise. The structured demonstration was intended to address some of these shortcomings in evaluation. The three main contributions were the establishment of an evaluation benchmark for reverse engineering tools, the combination of usability assessment with benchmarking, and the development of a package of materials to facilitate future tool evaluations.

The six tools evaluated in the study were Lemma [von Mayrhauser 1999], PBS, Rigi, TkSee [Singer 1997], Visual Age C++ [IBM 2004A], and Unix command-line utilities. Each tool was used by a team of expert users, who were monitored by
industrial observers in an attempt evaluate the usefulness of the tool in industrial software maintenance. The teams were presented with two reverse engineering tasks (‘Documentation’ and ‘Evaluate the structure of the application’) and three maintenance tasks (‘Modify the existing command panel’, ‘Add a new method for specifying arcs’, and ‘Bug fix: loading library objects’) to be performed on the \textit{xfig 3.2.1} utility [Xfig 2003], which consists of 50KLOC of ANSI C. The teams were then asked to present the results of their investigation. The documentation generated was less than expected, and there were some differences of opinion between the teams. A number of issues regarding the tools themselves were also uncovered.

The evaluation found that different tools are best suited to different tasks, and it would be useful to combine features from a number of tools. This is in agreement with our findings in our previous work [Pacione 2003a, Pacione 2003b]. Additionally, it is important to understand what the tool will be used for and select an appropriate tool accordingly. It is also important that the cost of introducing the tool can be justified. Certain users may be biased towards certain types of tool based on past experience. The organisers noted that a pilot evaluation would have been helpful in refining the experimental design, and is an important phase in any experimental evaluation. They also state that more inter-tool evaluations would have been interesting, more explicit instructions may also have been helpful, and more time (longer than one day) may have been desirable. The networking and teambuilding fostered by the collaborative demonstration was also noted. A future demonstration is planned based on parsing tools, which was an area that many of the teams had problems with.

2.7 Sim et al. (2000)

Sim et al. also present a number of observations regarding maintenance tools based on structured demonstration [Sim 2000b]. The aim was the same as that of the structured demonstration by Sim and Storey discussed above: to compare the capabilities of the tools. Three of the tools from the Sim and Storey evaluation discussed above (\textit{Rigi}, \textit{PBS}, and Unix utilities) were supplemented by two tools from the Workshop on Algebraic and Graph-Theoretic Approaches in Software Reengineering 2000 (\textit{GUPRO} [GUPRO 2004] and \textit{Bauhaus} [Koschke 2003]). Remarks on the tools and their application covered parsing, flexibility, quantity of extracted data, experience, and reasons for participation, while remarks on the demonstration scenario addressed the issues of tool selection, educational value, fairness, replication, and results scalability. A collaborative reengineering exercise is planned in which tools will be combined to address tasks.

2.8 Storey et al. (2000)

Storey et al. describe a subject-based evaluation of how program understanding tools affect users’ comprehension strategies [Storey 1997, Storey 2000]. Thirty subjects were observed carrying out a number of software comprehension tasks using the \textit{Rigi}, \textit{ShriMP} (Simple Hierarchical Multi-Perspective) [Storey 1996b], and \textit{SNIFF}+ tools. The goals of the study were to: examine the factors affecting the subjects’ choice of comprehension strategy; observe whether the tools enhanced the subjects’ preferred
comprehension strategy; devise a framework to characterise comprehension tools; and provide feedback for tool developers. A number of comprehension strategies have been proposed in the literature, such as: bottom-up [Shneiderman 1980, Pennington 1987], top-down [Brooks 1983, Soloway 1984], knowledge-based [Letovsky 1986], systematic [Littman 1986, Soloway 1988], as-needed [Littman 1986, Soloway 1988], and integrated [von Mayrhauser 1995].

Participants were assigned randomly to one of the three tools, and were asked to complete a number of comprehension tasks relating to an implementation of the Monopoly game [Brady 1974]. Each two-hour participant session consisted of orientation, training, practice tasks, formal tasks, a post-study questionnaire, and a post-study interview and debriefing. During the orientation phase, the outline of the experiment was explained to the subjects. The training phase was used to familiarise the subjects with the basic functionality of the tool they were to use. A number of practice tasks were used to allow the subjects to acquaint themselves with using the tool. The formal tasks were observed and videotaped, with the subjects encouraged to think aloud as they worked through the tasks. The questionnaire consisted of questions regarding the tools’ usabilities. Finally, the interview and debriefing was intended to stimulate further thoughts from the subject that may not have been expressed during the experiment.

Statistically significant results were obtained regarding the usabilities of the tools and the extents to which they supported each subject’s comprehension strategy. In general, it was found that the tools did enhance the subjects’ preferred comprehension strategies while carrying out the tasks, though there were instances where users were hindered by the tools. Future work is intended to study fewer, more experienced subjects with a broader task set over a greater time period. In this study, participants frequently browsed hierarchies of abstraction, which is further encouragement for our proposed model (see Section 1.2).

2.9 Bassil and Keller (2001)

Bassil and Keller describe a questionnaire-based evaluation of visualisation tools [Bassil 2001a, Bassil 2001b]. The questionnaire was available on the web and its location was publicised via mailing lists, newsgroups, and email. 107 responses were received, concerning more than 40 tools. This wide user base may make the results more generalisable. The aim of the study was to assess the functional, practical and cognitive aspects of visualisation tools that users desire, and how these compare to the functionality available in the various tools.

The questionnaire was designed around a list of properties of software visualisation tools, extracted from existing taxonomies. This should result in an objective questionnaire. The questionnaire consisted of two sections; the first for all software visualisation tool users, and the second for expert users. In the first section, participants were asked about their work context, the software systems they visualise, functional and practical aspects of software visualisation tools, and the tool they use. The functional aspects were assessed in terms of a list of 34 functional properties, such as source code browsing, graph visualisation, zooming, and program slicing. The second section asked technical questions about the software visualisation tool used by
the participants. Practical aspects were investigated in terms of a list of 13 aspects, such as tool cost, availability of technical support, ease of use, and portability. Most questions were closed, though there were fields to allow expanded answers for some questions.

Statistical analyses were performed on the survey results to reveal trends in the responses. Although the small number of participants compared to the number of tools makes results for individual tools insignificant, some statistically significant correlations were identified. The most interesting correlations in terms of our work were as follows. There was a positive correlation between software system size and desire to visualise it graphically, and a negative correlation between size and a desire to jump straight to the source code. There was also a higher correlation between source code visualisation and procedural systems than object-oriented systems. Lastly, there were high correlations between analysing object-oriented software and the desire for hierarchical representations and between OO software and the ability to navigate across hierarchies. These results reinforce the principles of our proposed visualisation model, which integrates multiple levels of abstraction (see Section 1.2).

Areas for improvement were identified as: finer grained choices, more open questions, and different surveys for different visualisation tool types. Future work is to include additional statistical analyses (e.g. factor and cluster analyses), more targeted surveys, and industrial integration.

2.10 Hatch et al. (2001)

Hatch et al. describe the strengths and weaknesses of four strategies for software visualisation evaluation [Hatch 2001]. Guidelines and frameworks can be useful during the initial formulation of a visualisation, but can also be used during evaluation if appropriate. Care must be taken to avoid 'self-measurement' when a visualisation is constructed according to a set of guidelines then evaluated against those same guidelines. Feature-based evaluation frameworks are useful for assessing the features of a visualisation against a set of questions. Care must be taken to select appropriate questions and question types. Also, current frameworks often omit potential negative features of the visualisation. Scenarios and walkthroughs allow a visualisation to be evaluated according to specific tasks, though it is easy to show the visualisation in its best light, while hiding undesirable features. However, the evaluation is influenced by the user and their biases. User and empirical studies can be a valuable source of evidence for evaluation. However, these entail overheads such as selecting and training subjects and analysing the results, and are also subject to user bias. It may be possible to perform statistical analyses on the results, though care must be taken when attempting to generalise any findings.

2.11 Knight (2001)

Knight discusses briefly some considerations to be taken into account when deciding whether or not a visualisation is effective [Knight 2001]. This is expressed in the form of an equation: effectiveness = suitability for task(s) + suitability of representation, metaphor, and mapping based on the underlying data. It is important
to take into account influences from the domain for which the visualisation was
designed, and also the dataset that it was intended to visualise.

2.12 Kollmann et al. (2002)

Kollmann et al. evaluate four static UML-based reverse engineering tools [Kollmann
2002a]. The tools evaluated were Together [Borland 2004], Rational Rose [IBM
2004b], Idea [Kollmann 2002b], and Fujaba [Wikman 1998]. The aim of the case
study was to compare the class diagram generation facilities of the tools. The Java-
based Mathaino legacy user interface migration tool was the subject of the case study
[Kapoor 2001].

The tools were assessed quantitatively by examining various properties of the class
diagrams they produced from the program code, such as the number of classes, types
of associations, multiplicities, and role names. The results were compared by
performing model operations using the BMO Toolkit [Koskinen 2001]. While basic
diagram generation results were broadly similar across the tool set, the research tools
were able to handle more advanced diagram concepts than the industrial tools, such as
multiplicities, inverse associations, and container resolution. It appears that the
investigation was carried out by the authors, though there is little information on the
experimental procedure. The use of the BMO Toolkit to compare the quantitative
results should result in an objective comparison of the tools’ capabilities, provided the
chosen measures provide an accurate reflection of the tools’ capabilities.

2.13 Summary

Globus and Uselton mention the use of a standardised set of test criteria, and also
advocate the use of human subjects in visualisation evaluation [Globus 1995]. We
plan to employ such approaches in the evaluation of our proposed model.

Murphy et al. illustrate the relevance of both quantitative and qualitative analysis of
empirical results [Murphy 1996, Murphy 1998]. Their study considers a broad base of
application types, which makes both their experimental procedure and results more
generally applicable than if only a single system, or type of system, were considered.
They also make use of objective evaluation criteria (number of calls detected, number
of false positives, etc.), which also increases the generality of the study.

Bellay and Gall use a checklist-based approach to investigate tool functionality
[Bellay 1997, Bellay 1998]. The checklist used is likely to be specific to the case
study of the particular embedded system and tools employed, and hence it (and the
experimental results) is not necessary immediately generalisable. It appears that the
investigation was carried out by one or two users, though the precise experimental
procedure employed is not explained.

Armstrong and Trudeau also adopt a checklist-based approach, with its attendant
drawbacks [Armstrong 1998]. Again, it appears that the evaluation was carried out by
one or two users, though there is little detail on the experimental procedure employed.
The four studies by Storey et al [Storey 1996a], Sim and Storey [Sim 2000a], Sim et al. [Sim 2000b], and Storey et al. [Storey 1997, Storey 2000] discussed above all analyse performance in typical comprehension tasks as a basis for tool evaluation, as in our previous work [Pacione 2003a, Pacione 2003b]. In addition, they advocate the use of multiple subjects.

Bassil and Keller choose an objective base for their questionnaire study, based on existing taxonomies [Bassil 2001a, Bassil 2001b]. They also use a broad base of participants.

Hatch et al. [Hatch 2001] summarise four techniques for software visualisation evaluation: guidelines and frameworks; feature-based evaluation framework; scenarios and walkthroughs; and user and empirical studies.

Knight briefly discusses some criteria to be considered when assessing whether or not a visualisation is effective [Knight 2001].

Kollmann et al. use an automated technique to compare the quantitative results of their study [Kollmann 2002a]. This should result in an accurate and objective analysis, provided the measures used are appropriate for the intended goals of the study.

The survey by Bassil and Keller reveals that as the size of software grows, analysts are more likely to employ graphical visualisations, and also that analysts are less likely to go straight to the source code [Bassil 2001a, Bassil 2001b]. This survey also shows that analysts are less likely to examine source code visualisations for object-oriented software, which implies that visualisations at a higher level of abstraction are most useful in the context of OO systems. The studies by Storey et al. [Storey 1997, Storey 2000] and Bassil and Keller both describe users navigating hierarchies of abstraction, which further reinforces the evidence from our prior work [Pacione 2003a, Pacione 2003b] that advocates a visualisation approach that integrates multiple levels of abstraction (see Section 1.2).

This section has discussed studies demonstrating a number of evaluation strategies:

- **Evaluation type**
  - Standard tests
  - Checklist
  - Specific tasks
  - Interviews
  - Observation
  - Questionnaire
  - Scenario/walkthrough

- **Participants**
  - Small-scale – usually authors
  - Medium-scale – around 10-30 participants, usually students
  - Large-scale – >100 participants, feasible only for questionnaires

- **Analysis**
  - Qualitative
  - Quantitative
In our previous work, we evaluated tools by assessing performance in a series of tasks intended to represent typical software comprehension tasks [Pacione 2003a, Pacione 2003b]. This was a small-scale study, and both qualitative and quantitative analyses were applied to the results. We plan to use the representative tasks approach to evaluate the visualisation model we have proposed, as we believe this approach provides the most realistic and objective assessment of the usefulness of the model for enhancing software visualisation for comprehension. There are some similarities between our approach and aspects of the studies by Storey et al [Storey 1996a], Sim and Storey [Sim 2000a], Sim et al. [Sim 2000b], and Storey et al. [Storey 1997, Storey 2000] discussed above. However, our approach clearly identifies the comprehension bases that lead to each task, hence providing justification that the tasks do in fact effectively assess the model’s usefulness for software comprehension. The following section considers the tasks we employed in our previous work, examines the basis for such a set of representative tasks, and proposes a new set of tasks to be used for the evaluation of the model proposed.
3 Evaluation based on representative tasks

Previous work evaluated the performance of visualisation tools by assessing their performance in typical software comprehension tasks [Pacione 2003a, Pacione 2003b]. This section: describes what the basis for such a task set should be in terms of what information is useful for the comprehension of OO systems; reviews the set of tasks used in that evaluation and evaluates their appropriateness and usefulness in evaluating a model for software comprehension; and presents a revised evaluation task set, with accompanying justification. It is intended that this revised task set will be used to evaluate the multifaceted, three-dimensional model proposed (see Section 1.2).

3.1 The basis for typical software comprehension tasks

A set of typical software comprehension tasks should seek to encapsulate the principal activities typically performed during real world software comprehension. Software comprehension activities can be divided up into those performed during general software comprehension, where the intention is to gain an overall understanding of (a subset of) a system, and those performed during a specific reverse engineering effort, where the intention is to carry out a specific task (e.g. fix a bug). Some activities may involve examining the structure of the software system, its behaviour, or both. Analysis at various levels of abstraction is often required. Depending on the activity, statically or dynamically extracted information, or a combination of both, may be desirable.

A number of typical software comprehension tasks are suggested in the literature. Storey at al. used two sets of tasks in their study of interfaces to the Rigi tool [Storey 1996a]. The ‘abstract’ tasks, which were high-level comprehension activities that involved understanding the overall structure or design of the software, were:

1. Show familiarity with the game [that the system simulates]
2. Summarise what subsystem $x$ does
3. Describe the purpose of artefact $x$
4. On a scale of 1-5, how well was the program designed?

The ‘concrete’ tasks, which were low-level comprehension activities that involved understanding only part of the software, were:

1. Find all artefacts on which artefact $x$ directly or indirectly depends
2. Find all artefacts that directly or indirectly depend on artefact $x$
3. Find an artefact that is not used
4. Find an artefact that is heavily used

Sim and Storey used two sets of tasks in their structured tool demonstration (Section 2.6, above) [Sim 2000a]. The tasks were intended to be representative of those encountered by a software developer in their everyday work. The reverse engineering tasks were:

1. Provide a textual and/or graphical summary of how the [system’s] source code is organised
2. Was [the system] well designed initially?
3. Do you think the original design is still intact?
4. How difficult will [the system] be to maintain and modify?
5. Are there some modules that are unnecessarily complex?
6. Are there any GOTOs? If so, how many? What changes would need to be made to remove them?

The maintenance tasks were:
1. Modify the existing command panel
2. Add a new method for specifying arcs
3. Bug fix: loading library objects

Storey et al. used a set of tasks in their evaluation of the comprehension strategies supported by the Rigi, SHriMP, and SNiFF+ tools that were intended to be typical of what a maintenance programmer would be asked to do [Storey 1997, Storey 2000]. These were:
1. Look at the real Monopoly game until you understand the general concept and rules of the game. Have you played Monopoly before?
2. Spend a while browsing the program using the provided software maintenance tool and try to gain a high level understanding of the structure of the program.
3. In the computer game, how many players can play at any one time?
4. Does the program support a ‘computer’ mode where the computer will play against one opponent?
5. There should be a limited total number of hotels and houses; how is this limit implemented and where is it used? If this functionality is not currently implemented, would it be difficult to add? What changes would this enhancement require?
6. Where and what needs to be changed in the code to implement a new rule which states that a player in jail (and not just visiting) cannot collect rent from anyone landing on his/her properties?
7. Overall, what was your impression of the structure of the program? Do you think it was well written?

In their description of the Shimba reverse engineering tool, Systä et al. suggest three sets of tasks supported by the tool [Systä 2001]. The ‘overall understanding’ tasks were:
1. What are the static software artefacts and how are they related?
2. How are the software artefacts used at run-time?
3. What is the high-level structure of a subject system?
4. How do the high-level components interact with each other?
5. Does the run-time behaviour contain regular behavioural patterns that are repeated? If so, what are the patterns and under which circumstances do they occur?
6. How heavily has each component of a subject system been used at run-time and which components have not been used at all?

The ‘goal-driven reverse engineering tasks’ were:
1. How does a certain component behave and how is it related to the rest of the system?
2. When was an exception thrown or when did an error occur? What happened before that and in which order?
3. How is the component that causes exceptional behaviour constructed?
The ‘object/method behaviour’ tasks were:
1. What is the dynamic control flow and the overall behaviour of an object or a method?
2. How can a certain state of an object be reached (i.e. which execution paths lead from the initial state to this state) and how does the execution continue (i.e. which execution paths lead from this state to the final state)?
3. To which messages has an object responded at a certain state during its lifetime?
4. Which methods of the object have been called during execution?

Kirk et al. conducted a questionnaire survey of students reusing a framework [Kirk 2001]. The questions asked how difficult the students found understanding the following aspects of the framework:
1. Understanding individual classes and their methods
2. Using abstract classes and interfaces
3. Mapping your solution to framework code
4. Understanding the structure of inheritance hierarchies and object compositions
5. Understanding design patterns
6. Understanding the dynamic structure of the framework
7. Choosing from alternative framework solution strategies
8. Understanding the [framework’s] problem domain

The study found that the key issues were:
3. Mapping your solution to framework code
6. Understanding the dynamic structure of the framework
7. Choosing from alternative framework solution strategies

The principal comprehension activities elicited from the literature tasks are as follows.
A1. Investigating the functionality of (a part of) the system
A2. Adding to or changing the system’s functionality
A3. Investigating the internal structure of an artefact
A4. Investigating dependencies between artefacts
A5. Investigating runtime interactions in the system
A6. Investigating how much an artefact is used
A7. Investigating patterns in the system’s execution
A8. Assessing the quality of the system’s design
A9. Understanding the domain of the system

A set of typical software comprehension tasks should address all of these activities.

3.2 Typical software comprehension tasks

A definitive set of typical software comprehension tasks does not appear to exist in the literature. Therefore, in our previous work, we compiled two sets of tasks that were intended to be representative of those performed in a typical software comprehension effort [Pacione 2003a, Pacione 2003b]. The tasks were divided into those typical of general software comprehension tasks, usually carried out when attempting to understand a large part of the system, and those typical of specific reverse engineering tasks, usually carried out on smaller parts of the system to perform a specific purpose.
3.2.1 General software comprehension tasks

The following questions are intended to be typical of those asked during the course of a general software comprehension effort. Questions G1-G6 were inspired by the six ‘overall understanding’ questions of Systä et al. [Systä 2001, p.378]. Questions G7 and G8 address issues that are particularly relevant to framework reuse, while G9 is an important software comprehension issue.

G1. What is the static structure of the software system?
G2. What interactions occur between objects at runtime?
G3. What is the high-level structure/architecture of the software system?
G4. How do the high-level components of the software system interact?
G5. What patterns of repeated behaviour occur at runtime?
G6. What is the load on each component of the software system at runtime?
G7. What design patterns are present in the software system's implementation?
G8. Where in the software system are the hotspots where additional functionality can be added?
G9. What impact will a change made to the software system have on the rest of the software system?

3.2.2 Specific reverse engineering tasks

The following questions are intended to be typical of those asked during the course of a specific reverse engineering effort. Questions S1, S2, and S6 were inspired by the ‘goal-driven reverse engineering’ and ‘object/method behaviour’ questions of Systä et al. [Systä 2001, p.378]. Questions S3, S4, and S5 address issues typically encountered in framework comprehension and are typical maintenance activities [Kirk 2001].

S1. What are the collaborations between the objects involved in an interaction?
S2. What is the control structure in an interaction?
S3. How can a problem solution be mapped onto the functionality provided by the software system?
S4. Where is the functionality required to implement a solution located in the software system?
S5. What alternative functionalities are available in the software system to implement a solution?
S6. How does the state of an object change during an interaction?

3.3 Task set analysis

The classification of these tasks into general software comprehension tasks and specific reverse engineering tasks delineates the tasks into those that are most conveniently solved using higher and lower levels of abstraction, respectively, which constitutes the first dimension of the model proposed. The tasks can also be classified by whether they are concerned with the system’s structure, behaviour, or both – the second dimension of our model.
• Structural
  o G1 What is the static structure of the software system?
  o G3 What is the high-level structure/architecture of the software system?

• Behavioural
  o G2 What interactions occur between objects at runtime?
  o G4 How do the high-level components of the software system interact?
  o G5 What patterns of repeated behaviour occur at runtime?
  o S1 What are the collaborations between the objects involved in an interaction?
  o S2 What is the control structure in an interaction?
  o S3 How can a problem solution be mapped onto the functionality provided by the software system?
  o S5 What alternative functionalities are available in the software system to implement a solution?
  o S6 How does the state of an object change during an interaction?

• Both
  o G6 What is the load on each component of the software system at runtime?
  o G7 What design patterns are present in the software system's implementation?
  o G8 Where in the software system are the hotspots where additional functionality can be added?
  o G9 What impact will a change made to the software system have on the rest of the software system?
  o S4 Where is the functionality required to implement a solution located in the software system?

All of the tasks (except L5 ‘What patterns of repeated behaviour occur at runtime?’ and L6 ‘What is the load on each component of the software system at runtime?’) can be analysed using either statically or dynamically extracted information; the third dimension of our model integrates statically and dynamically extracted information.

3.4 New task sets

None of the tools in our previous work were able to answer either of question G7 ‘What design patterns are present in the software system's implementation?’ or G8 ‘Where in the software system are the hotspots where additional functionality can be added?’. These tasks are most applicable to frameworks and may not have been anticipated by the tool developers. Work by Keller et al. [Keller 1999] and others on identifying design patterns, and by Schauer et al. [Schauer 1999] and others on identifying hotspots, stress the role of the human analyst and reveal that detecting design patterns and hotspots is a non-trivial task that can benefit from tool support. It is for these reasons that we exclude these tasks from our revised task set.

Tasks G3 ‘What is the high-level structure/architecture of the software system?’ and G4 ‘How do the high-level components of the software system interact?’ are more abstract versions of G1 ‘What is the static structure of the software system?’ and G2 ‘What interactions occur between objects at runtime?’, respectively. However, this
similarity is desirable to allow higher levels of abstraction to be evaluated. To clarify this distinction, we add the word “class” to G1.

We remove the word “static” from G1, as we do not mean to imply that we are concerned solely with the structure as defined by statically extracted information. For the same reason, we remove the phrase “at runtime” from G2. Information on both a systems structure and behaviour can be extracted both statically and dynamically.

We add a new task G8 ‘What are the data structures that are used in the software system?’, which is typical of comprehension of data-intensive software, such as databases or transaction processing systems.

We believe that these task sets address all of the issues relating to the questions from previous studies identified in Section 3.1, above, and that they constitute typical software comprehension tasks that can be used to realistically evaluate the usefulness and effectiveness of software comprehension models and tools for real-world software comprehension.

3.4.1 General software comprehension tasks

G1. What is the class structure of the software system?
G2. What interactions occur between objects?
G3. What is the high-level structure/architecture of the software system?
G4. How do the high-level components of the software system interact?
G5. What patterns of repeated behaviour occur at runtime?
G6. What is the load on each component of the software system at runtime?
G7. What impact will a change made to the software system have on the rest of the software system?
G8. What are the data structures that are used in the software system?

3.4.2 Specific reverse engineering tasks

S1. What are the collaborations between the objects involved in an interaction?
S2. What is the control structure in an interaction?
S3. How can a problem solution be mapped onto the functionality provided by the software system?
S4. Where is the functionality required to implement a solution located in the software system?
S5. What alternative functionalities are available in the software system to implement a solution?
S6. How does the state of an object change during an interaction?

3.4.3 Justification

The above task set is intended to exercise all of the features of our proposed model. It has a selection of tasks requiring structural, behavioural, data, and combined information, various levels of abstraction, and statically and dynamically extracted
information. The tasks are intended to be representative of typical software comprehension tasks, and are based on software comprehension activities, as described in Section 3.1, above. Therefore, an evaluation of our proposed model using this task set should provide an accurate assessment of its utility and effectiveness in supporting software visualisation for program comprehension.

Table 3.1 illustrates the principal correspondences between the typical software comprehension activities identified in Section 3.1 and the revised evaluation tasks from Section 3.4.1 and 3.4.2. This table illustrates that the revised evaluation tasks address all of the typical of software comprehension activities, without redundancy. The number of tasks that address each activity varies as not all activities are at the same level of granularity. These tasks are proposed as a complete set of typical comprehension tasks, representative of the full range of comprehension activities, and encompassing all those found in the related literature.

Table 3.1. The correspondence between typical software comprehension activities and the revised task sets

<table>
<thead>
<tr>
<th>Activity</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>G1, G2, S1</td>
</tr>
<tr>
<td>A2</td>
<td>G7, S3, S4, S5</td>
</tr>
<tr>
<td>A3</td>
<td>G1, G8</td>
</tr>
<tr>
<td>A4</td>
<td>G1, G3</td>
</tr>
<tr>
<td>A5</td>
<td>G2, G4, S1, S2, S6</td>
</tr>
<tr>
<td>A6</td>
<td>G2, G6</td>
</tr>
<tr>
<td>A7</td>
<td>G5</td>
</tr>
<tr>
<td>A8</td>
<td>G3, G4, G7</td>
</tr>
<tr>
<td>A9</td>
<td>G3, G4</td>
</tr>
</tbody>
</table>

3.5 Summary

This section has discussed the basis for a set of typical OO software comprehension tasks. It also reviewed the set of tasks used in our earlier work and evaluated their appropriateness and usefulness in evaluating a model for software comprehension. On the basis of this analysis, a new evaluation task set, with accompanying justification, was presented. In the following section, we will use this task set to evaluate our proposed software visualisation model.
4 Evaluation of the proposed model

Previous work proposed a multi-faceted, three-dimensional model for software comprehension (see Section 1.2), which was designed to address the comprehension shortcomings in current software visualisation tools identified in prior work [Pacione 2003a, Pacione 2003b]. In this section, we will apply the evaluation technique described in Section 3 to this model theoretically, in order to determine which aspect(s) of the model are most useful in improving the effectiveness of software visualisation for comprehension and hence most promising for future research. We will also analyse the refined model to assess its support for software comprehension strategies.

4.1 Model information required to address typical software comprehension tasks

The model was evaluated theoretically by comparing the information required by each task against the information provided by each aspect of the model. For example, in order to answer question G6 ‘What is the load on each component of the software system at runtime?’, we require both structural and behavioural information concerning classes, components, and distribution (levels 2-4), and only dynamically extracted information would be useful. As another example, to answer question S6 ‘How does the state of an object change during an interaction?’, we require behavioural information at the intra-object level (level 1), and both statically and dynamically extracted information would be useful.

Tables 4.1 and 4.2 illustrate the information required from each dimension of the proposed model to address each of the typical software comprehension tasks from Sections 3.4.1 and 3.4.2 respectively.

Table 4.1 Information required from each dimension of the proposed model to address the general software comprehension tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>Abstraction levels</th>
<th>Facets</th>
<th>Static/dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>2</td>
<td>Structure</td>
<td>Both</td>
</tr>
<tr>
<td>G2</td>
<td>2</td>
<td>Behaviour</td>
<td>Both</td>
</tr>
<tr>
<td>G3</td>
<td>3-4</td>
<td>Structure</td>
<td>Both</td>
</tr>
<tr>
<td>G4</td>
<td>3-5</td>
<td>Behaviour</td>
<td>Both</td>
</tr>
<tr>
<td>G5</td>
<td>1-5</td>
<td>Behaviour</td>
<td>Dynamic</td>
</tr>
<tr>
<td>G6</td>
<td>2-4</td>
<td>Structure, Behaviour</td>
<td>Dynamic</td>
</tr>
<tr>
<td>G7</td>
<td>1-5</td>
<td>Structure, Behaviour, Data</td>
<td>Both</td>
</tr>
<tr>
<td>G8</td>
<td>1-5</td>
<td>Structure, Data</td>
<td>Both</td>
</tr>
</tbody>
</table>
Table 4.2 Information required from each dimension of the proposed model to address the specific reverse engineering tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>Abstraction levels</th>
<th>Facets</th>
<th>Static/dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>2</td>
<td>Behaviour</td>
<td>Both</td>
</tr>
<tr>
<td>S2</td>
<td>1-2</td>
<td>Behaviour</td>
<td>Both</td>
</tr>
<tr>
<td>S3</td>
<td>1-3</td>
<td>Behaviour</td>
<td>Both</td>
</tr>
<tr>
<td>S4</td>
<td>2-3</td>
<td>Structure, Behaviour</td>
<td>Both</td>
</tr>
<tr>
<td>S5</td>
<td>2-3</td>
<td>Behaviour</td>
<td>Both</td>
</tr>
<tr>
<td>S6</td>
<td>1</td>
<td>Behaviour</td>
<td>Both</td>
</tr>
</tbody>
</table>

Firstly, Tables 4.1 and 4.2 show that a variety of abstraction levels are required to address the typical software comprehension tasks.

Secondly, it is clear from these tables that the Data facet is rarely used, and when it is used it is in conjunction with the Structure facet. We believe that this is because in the object-oriented paradigm, a system’s implemented data structures are encapsulated in the system structure. Hence, the information that may have made a data facet appropriate in a procedural system is available in the structure facet for object-oriented systems. Higher-level data structures may also be present, but not apparent in the structure facet due to physical implementation details: a logical data structure may be implemented as a number of smaller physical data structures. For example, a hash map may be implemented as an array of Vectors in Java. The higher levels (3-5) of the data facet would illustrate these higher-level data structures if the required information on such structures were available.

Thirdly, these tables show that dynamically extracted information is useful for all tasks, and a combination of statically and dynamically extracted information is useful for addressing most of the tasks. Only the two general software comprehension questions regarding runtime behaviour do not require statically extracted information.

4.2 Summary

It would appear that the most useful features of our proposed model are the range of abstraction levels and the structural and behavioural facts. In our future work we will investigate the abstraction scales for the structural and behavioural facets. The first step of this will be to refine and specify these features of the model.
Validation of support for software comprehension strategies

It is important to validate any model intended to facilitate software comprehension to ensure that it does in fact support commonly used software comprehension strategies. This section describes a number of software comprehension strategies that have been proposed in the literature and observed in practice, and examines how our proposed model supports these strategies.

5.1 Software comprehension strategies

A number of software comprehension strategies have been proposed in the literature to describe how analysts understand software. An overview of these is given in work by Storey et al. [Storey 1997, Storey 2000]; we summarise them below.

5.1.1 The bottom-up model

Shneiderman proposed a bottom-up model for software comprehension [Shneiderman 1980]. This involves the analyst reading the source code and ‘chunking’ low-level details into higher-level abstractions. These abstractions are then grouped further, the process being repeated until a high-level understanding of the software is obtained.

Pennington also observed a bottom-up strategy being used [Pennington 1987]. Statement and control flow information was used to build micro-structures, which were then chunked by macro-structures to build a ‘program model’. Domain knowledge was then used to build a ‘situation model’, consisting of a hierarchy of functional abstractions.

5.1.2 The top-down model

In contrast to Shneiderman, Brooks proposed a top-down model for software comprehension, in which knowledge about the application domain is mapped to the source code [Brooks 1983]. This strategy begins with an initial hypothesis about the program, which is then refined into a hierarchy of secondary hypotheses, based on the presence or absence of ‘beacons’ in the code.

Soloway and Ehrlich observed the use of a top-down strategy when the program or application domain is familiar to the analyst [Soloway 1984].

5.1.3 The knowledge-based model

Letovsky suggested that programmers are capable of employing both top-down and bottom-up comprehension processes [Letovsky 1986]. His theory consists of a knowledge base containing the programmer’s domain and programming knowledge, a mental model representing the programmer’s current understanding of the program,
and an assimilation process that describes how the mental model is developed using the knowledge base and program information. The assimilation process consists of ‘inquiry episodes’, in which the programmer asks a question, proposes an answer, then examines the program code and documentation to confirm or reject the answer.

5.1.4 The systematic and as-needed models

Littman et al. observed that programmers employ either a systematic comprehension strategy, in which they read code in detail and trace control and data flow, or an as-needed strategy, in which they concentrate on only the code related to their current task [Littman 1986].

Soloway et al. combined these two theories into ‘macro strategies’, aimed at a more abstract level of understanding [Soloway 1988]. The systematic macro strategy involves tracing the whole program’s control flow and performing simulations, while reading all of the program code and documentation. This strategy becomes less feasible as the size of the program increases. The as-needed macro strategy is more commonly used, in which the programmer examines only those parts of the program that they think are relevant. More mistakes may occur with this strategy as relevant interactions may be mistakenly overlooked.

5.1.5 The integrated model

Von Mayrhauser and Vans combined the bottom-up, top-down, and knowledge-based models into a unified metamodel [von Mayrhauser 1995]. They proposed that software comprehension occurs at various levels of abstraction by alternating between these three strategies. Their metamodel combines the program and situation models of Pennington with the top-down model of Soloway. Experiments showed some programmers switching frequently between all three models during the comprehension process.

5.2 Object-oriented software comprehension

The models above were initially developed in the context of procedural systems. As our proposed model is concerned with object-oriented software comprehension, it is important to investigate whether or not these models are also applicable to object-oriented systems.

Corritore and Wiedenbeck conducted an empirical study of procedural and object-oriented programmers performing maintenance tasks, and attempted to determine the comprehension strategies used by each group [Corritore 2000]. The study examined both the scope and the direction of the programmers’ comprehension activities. The scope of comprehension refers to the breadth of the programmer’s understanding of the software. As described in Section 5.1.4, Littman et al. proposed the systematic and as-needed models of software comprehension. Programmers employing the systematic model would be expected to have a broader scope of comprehension than those employing the as-needed model. The direction of comprehension refers to
whether the comprehension approach used was bottom-up, top-down, or a combination of the two. As described in Sections 5.1.1 and 5.1.2, programmers employing a bottom-up strategy would be expected to access more low-level information, while those employing a top-down strategy would be expected to make greater use of more abstract information. The programmers’ comprehension scope was measured in terms of the number of files they opened. Comprehension direction was measured in terms of the abstraction level of the files they opened; documentation files were classified as the highest level, followed by header files, then source code files.

Two groups were used in the study, one consisting of OO programmers, the other procedural programmers. Two versions of an ~800 LOC airline records database system were prepared in C and C++, with accompanying documentation. The programmers completed three tasks, each 7-10 days apart. The first task was to study the system and perform a simple practice maintenance task. The following two tasks involved performing maintenance activities for which data was collected.

The results indicate that both groups employed a wide comprehension scope throughout the tasks, though it narrowed slightly as the tasks progressed. The procedural group had a slightly broader scope than the OO group. This is in contrast to the result of a (procedural) comprehension study by Koenemann and Robertson, who reported a very restricted scope of comprehension [Koenemann 1991]. The procedural group employed a bottom-up comprehension strategy throughout the study, while the OO group initially used a top-down strategy before moving to bottom-up.

This study provides empirical evidence for the comprehension strategies used during OO software comprehension. However, there are a number of influencing factors that must be considered before these results can be generalised. The authors recognise the limitations of the granularity of the study, and mention the need for more detailed studies involving videotaping and ‘thinking aloud’. They also acknowledge that the number and abstraction level of the files opened are surrogates for the measures of scope and direction respectively. It must be considered whether these are indeed appropriate measures for assessing software comprehension scope and direction. They also point out that the order that the files were viewed was not considered. This would appear to be important when attempting to determine the direction of comprehension. It could be argued that the use of a single type of system (database) will affect the results. The programmers were familiar with the domain of the system, which may not always be the case. All of the OO programmers were familiar with procedural programming, though the reverse was not the case. It must also be considered whether the C and C++ versions of the same system represent realistic comprehension targets. For example, is the OO version really a realistic OO system, or simply an ‘objectisation’ of the C program? Similarly, an attempt was made to give the same level of documentation to each paradigm group. This raises the question of whether the documentation provided was realistic. For example, OO documentation would typically consist of class diagrams, sequence diagrams, and use cases, which were not present here. The authors acknowledge that a trade-off was required between experimental control and generalisability. They call for laboratory experiments of large systems over time, and industrial studies over longer time periods.
Burkhardt et al. observed that expert OO programmers developed domain models in terms of objects and interactions during comprehension [Burkhardt 1997]. Burkhardt et al. also suggest that OO experts employ more top-down analysis than OO novices in comprehension [Burkhardt 1998]. In comparing OO and procedural programmers in comprehension, Corritore and Wiedenbeck report that OO programmers initially developed a stronger understanding of program structure (e.g. inter-object relationships) and a poorer understanding of operations and control flow than procedural programmers, suggesting that OO programmers employ a more top-down approach initially [Corritore 1999]. Henry and Humphrey observed that modifications to OO programs were more localised than procedural modifications, possibly indicating that an as-needed comprehension strategy is appropriate [Henry 1993].

5.3 Comprehension in software visualisation

Petre et al. make a number of pertinent observations regarding the cognitive aspects of software visualisation [Petre 1997]. They remark that most software visualisations are not constructed with cognitive representations of software in mind. We intend to address this point in the following subsection by illustrating how our proposed model supports software comprehension strategies. Petre et al. also highlight the importance of determining the tasks that programmers perform, then constructing visualisations to support these tasks. We have achieved this by means of the task set developed in Section 3 and the analysis and initial refinement of our model in Section 4. We will evaluate our model using typical software comprehension tasks to ensure that we have succeeded in our goal of producing a visualisation model that supports software comprehension. Petre et al. discuss the difficulty of mapping the multitude of software information onto the limited number of dimensions available in a visualisation, and emphasise the importance of making appropriate use of scarce resources in this regard. Each dimension of our multi-dimensional model represents a principal aspect of software that will be useful for comprehension, namely: multiple interrelated levels of abstraction, different perspectives of the software, and the integration of statically and dynamically extracted information. Petre et al. also point out that one visualisation is unlikely to be suitable for all users, and that individual skill and variation must be considered. Our model is intended to be flexible: various diagram types can be used to represent the information in the model, and information from various aspects of the model can be combined.

5.4 Support for software comprehension strategies in our proposed visualisation model

Software comprehension tools should be designed to support software comprehension strategies. Storey et al. conducted an experiment to evaluate such support, which was discussed in Section 2.8 [Storey 1997, Storey 2000]. Our proposed model is intended to provide support for software comprehension strategies, which were summarised in Section 5.1.

The bottom-up strategy consists of chunking lower-levels artefacts to build higher-level abstractions (see Section 5.1.1) [Shneiderman 1980, Pennington 1987]. This requires the analyst to be able to start with low-levels details about a software system,
and gradually move to higher-level details. The multiple interrelated levels of abstraction in our model facilitate this. Our model should therefore provide support for the bottom-up comprehension strategy.

The top-down strategy involves mapping application domain knowledge to program source code (see Section 5.1.2) [Brooks 1983, Soloway 1984]. This requires the analyst to start with an abstract representation of the software, and move through successively less abstract representations, searching for beacons, in order to validate his initial hypothesis about the software. Again, the multiple interrelated levels of abstraction in our model facilitate this process. Our model could therefore be expected to support the top-down comprehension strategy.

The knowledge-based strategy consists of enquiry episodes, where the analyst searches through the software in an attempt to find an answer to their current question in order to build up a their mental model of the software (see Section 5.1.3) [Letovsky 1986]. This strategy requires the ability to move freely through the software system in order to assimilate information about it. Our model is intended to be easy to navigate and manipulate, so should therefore provide support for the knowledge-based comprehension strategy.

The systematic approach involves tracing control and data flow and reading documentation for the entire program, while the as-needed strategy involves focussing only on the code that is perceived as being relevant to the task at hand (see Section 5.1.4) [Littman 1986, Soloway 1988]. The systematic approach requires that the analyst can conveniently access information about the entire system at an appropriate level of abstraction. The multiple interrelated levels of abstraction are intended to provide this facility, so our proposed model should provide support for the systematic comprehension strategy. The as-needed approach relies on the analyst being able to access information relevant to the current task. Again, the multiple interrelated levels of abstraction should facilitate this, though locating the desired information in a visualisation is an inherently tricky issue. Nevertheless, it is expected that our proposed model will provide support for the as-needed comprehension strategy.

The integrated approach combines the bottom-up, top-down, and knowledge-based approaches into a single metamodel (see Section 5.1.5) [von Mayrhauser 1995]. This approach relies on the analyst being able to switch freely between these strategies, and hence move freely through the software system and access information conveniently at various levels of abstraction. As our proposed model should provide support for the three strategies contained in the integrated model, as described above, it would be hoped that it would also provide support for the integrated comprehension strategy. Our proposed model is not aimed at supporting a single comprehension strategy, which should allow the analyst to switch between comprehension strategies without hindrance. The model is also intended to be easy to navigate. The multiple interrelated levels of abstraction should provide the information required for this approach. We therefore believe that our proposed model should provide support for the integrated comprehension strategy.
5.5 Summary

This section has given an overview of several software comprehension strategies. We also discussed the applicability of these accepted strategies to OO comprehension. Relevant cognitive aspects of software visualisation were discussed. Finally, we assessed support for these comprehension strategies in our proposed model. It appears that our model provides support for all of the established software comprehension strategies identified above, though a more rigorous evaluation will be required to confirm this.
6 Summary, conclusions, and future work

This report has described the role of evaluation, and introduced our proposed model of software visualisation for software comprehension. Evaluation techniques for software visualisation and software comprehension tools were discussed. Evaluation using representative tasks was discussed in more detail, and a refined set of evaluation tasks was produced. This task set was used to evaluate our proposed visualisation model to determine the aspects most useful for comprehension. We then described software comprehension strategies and discussed how these are supported in our proposed model.

The evaluation in Section 4 appears to indicate that the most useful features of our proposed model are the range of abstraction levels and the structural and behavioural facets. Our future work will concentrate on these two aspects. The integration of statically and dynamically generated information would also provide an interesting research possibility.

In our future work we will investigate the abstraction scales for the structural and behavioural facets. This will involve the refinement and specification of these features of the model. We will then evaluate the final model using the evaluation task set established in Section 3. We intend that the result of this process will be a software visualisation model that supports large-scale, real-world software comprehension.
References


