Effective Visualisation for Comprehending Object-Oriented Software: A Multifaceted, Three-Dimensional Abstraction Model for Software Visualisation

Michael J. Pacione

Michael.Pacione@cis.strath.ac.uk

March 2004

Technical Report EFoCS-52-2004
Department of Computer and Information Sciences
University of Strathclyde, Glasgow, UK
Abstract

This report presents further research in software visualisation that investigates and seeks to address the issues raised by previous work [Pacione 2003a, Pacione 2003b]. Software visualisation, abstraction, the integration of static and dynamic information, and effective ways of presenting, exploring, and querying visualisations are discussed. A multi-faceted, three-dimensional abstraction model for software visualisation is presented. It is intended that the model proposed will address the issues with current visualisation tools revealed in the previous work. The model proposed consists of multiple levels of abstraction arranged in separate facets representing various properties of software (e.g. structure, behaviour, data), and integrates static and dynamic information. Related work is discussed, and future work in software visualisation based on the model proposed is set out.
Keywords

Abstraction hierarchies, diagrams, facets, models, software comprehension, software visualisation, static and dynamic analysis, views
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 figures
Page viii List of tables

Page 1  1 Introduction
       1.1 Context, problem, and solution
       1.2 Software visualisation
       1.3 Report structure

Page 3  2 Abstraction
       2.1 The concept of abstraction
       2.2 The historical origins of abstraction
       2.3 The application of abstraction
       2.4 Abstraction in software engineering
       2.5 Abstraction in software visualisation

Page 6  3 Static and dynamic information
       3.1 Static software comprehension techniques
       3.2 Dynamic software comprehension techniques

Page 7  4 Effective presentation techniques for visualisation
       4.1 Diagrams for describing software
           4.1.1 Structured design diagrams
           4.1.2 Object-oriented diagrams
           4.1.3 Recent literature
       4.2 Views for software comprehension
           4.2.1 A single view illustrating a single facet
           4.2.2 Multiple independent views illustrating a single facet
           4.2.3 Multiple interdependent views illustrating a single facet
           4.2.4 A single view illustrating multiple facets
           4.2.5 Multiple independent views illustrating multiple facets
           4.2.6 Multiple interdependent views illustrating multiple facets

Page 13  5 Effective techniques for exploring and querying visualisations
        5.1 Exploration
        5.2 Querying
        5.3 Guided navigation
Page 14  6 A multifaceted, three-dimensional abstraction model for software visualisation
   6.1 Background
   6.2 Research hypothesis
   6.3 A meta-model
   6.4 A model for object-oriented systems
   6.5 Examples

Page 23  7 Related work
   7.1 Software visualisation
   7.2 Abstraction

Page 25  8 Summary, conclusions, and future work
   8.1 Summary
   8.2 Feasibility
   8.3 Key research challenges
   8.4 Future work

Page 27  References
List of figures

Page 8  Figure 4.1  An example UML sequence diagram for the Singleton design pattern.

Page 10 Figure 4.2  The six arrangements of views onto a software model. The rectangles around the views in parts c and f represent the coordination inherent in such interdependent arrangements.

Page 19 Figure 6.1  A multifaceted, three-dimensional abstraction model for software visualisation.

Page 20 Figure 6.2  An example of the structure abstraction hierarchy.

Page 21 Figure 6.3  An example of the behaviour abstraction hierarchy.

Page 22 Figure 6.4  An example of the data abstraction hierarchy.
List of tables

Page 9    Table 4.1    A selection of diagrams for describing software.

Page 15   Table 6.1    The multifaceted abstraction meta-model.

Page 16   Table 6.2    An instantiation of the multifaceted abstraction model for OO systems.
1 Introduction

1.1 Context, problem, and solution

Software visualisation is the process of modelling software systems for comprehension [Price 1993]. The comprehension of software systems both during and after development is a crucial component of the software process [von Mayrhauser 1995]. The complex interactions inherent in the object-oriented paradigm make visualisation a particularly appropriate comprehension technique, and the large volume of information typically generated during visualisation necessitates tool support.

A recent study by the author revealed that current visualisation tools address only specific software comprehension and reverse engineering issues [Pacione 2003a, Pacione 2003b]. Most are relatively tightly focussed, lack the capability to integrate statically and dynamically extracted information, and fail to address the difficulties in comprehension caused by inherent features of the object-oriented paradigm.

In order to address the disadvantages with current visualisation techniques, an approach is proposed that integrates abstraction, structural and behavioural perspectives, and statically and dynamically extracted information. The aim of this research is to improve the effectiveness of visualisation techniques for large-scale software understanding based on the use of abstraction, interrelated facets, and the integration of statically and dynamically extracted information.

1.2 Software visualisation

Software visualisation tools have a variety of applications in the reverse engineering and software comprehension processes, including software evolution, reengineering, refactoring, reuse, redocumentation, and legacy system migration. Recent work on software visualisation demonstrated that no current tool is adequate, when used individually, for all software comprehension and reverse engineering activities [Pacione 2003a, Pacione 2003b]. This work identified three principal issues for future research, namely: the integration of static and dynamic information and a range of abstraction levels; effective ways of presenting the results of the visualisation process; and effective ways of exploring and querying visualisations. This report considers these issues and proposes an extension of the abstraction scale proposed by Pacione [Pacione 2003a, Pacione 2003b]. As an evolution from the single linear scale, a multifaceted, three-dimensional model is proposed, with separate abstraction scales for ‘interesting’ facets of object-oriented systems (e.g. structure and behaviour). Each named level of each facet of the scale consists of a set of well-defined entities and relationships, along with appropriate diagram types to illustrate information at that level of abstraction. The third dimension of the model consists of a number of static and/or dynamic analyses of the system. The aim of this report is to begin work on the areas for future research identified in the previous work [Pacione 2003a, Pacione 2003b], and to lay the foundations for further work in the area. The original
motivation for this work was the apparent lack of use of software visualisation tools outside the research laboratory despite their potential applications.

1.3 Report structure

This section has provided a brief introduction to software visualisation and the open issues identified in previous work. Section 2 discusses the concept, origins, and application of abstraction, and its use in software engineering and software visualisation. Section 3 discusses the nature of and distinction between static and dynamic information. Section 4 is concerned with effective ways of presenting the results of the visualisation process, including the use of diagrams and views to describe software. Section 5 considers effective ways of exploring and querying visualisations. Section 6 presents the multifaceted, three-dimensional abstraction model for software visualisation proposed. Section 7 surveys related work and compares it with the ideas presented in this report. Section 8 summarises the report and outlines future work.
2 Abstraction

2.1 The concept of abstraction

Abstraction is the process of producing a simplified representation that emphasises the important information while suppressing details that are (currently) uninteresting, with the goal of reducing complexity and increasing comprehensibility [Berard 1993]. Lee and Fishwick define an abstraction as a “generalized, idealised model of a system” [Lee 1996]. Abstraction is employed in a wide variety of scientific fields, including statistics, simulation theory, management science, and software engineering. Two principal features of abstract models identified by Fishwick are that they are usually less complex and more comprehensible than the model from which they are derived [Fishwick 1988].

2.2 The historical origins of abstraction

Abstraction has provided the foundation that we use for performing mental tasks ever since human thought began [Kirsanov 1998]. The modern use of abstraction began in the early twentieth century in a variety of fields [Hooker 1996]. Hooker provides evidence for this with the examples of abstract art, atonal music, Einstein’s Theory of Relativity [Einstein 1920], and Keynesian economics [Keynes 1936]. In this modern context, abstraction refers to the view that separate aspects of human experience are independent of each other, and can hence be reasoned about in isolation.

2.3 The application of abstraction

Fishwick [Fishwick 1988] presents abstraction in the context of simulation using the dining philosophers (DP) problem [Dijkstra 1968]. The models used are a frequency distribution, finite state automaton, observed data, Petri net [Petri 1962, Peterson 1981], flow graph, and equations. These models are then presented as an abstraction network, consisting of the models and abstraction techniques that relate them. For example, a more abstract flow graph model of the DP system can be derived from the Petri net model using abstraction by representation. The model proposed in this report can be considered to be an example of an abstraction network. Fishwick describes a number of abstraction techniques, namely: abstraction by representation, abstraction by induction, abstraction by reduction, total systems morphism, and partial systems morphism.

In abstraction by representation, an abstract model represents a base model in another form. Such models are often purely structural and have no behaviour, except as defined by the more detailed base model. Abstraction by induction involves combining elements from the base model to form a smaller, more compact representation. Abstraction by reduction is achieved by deriving a representative summary of the base model. A total systems morphism (TSM) [Zeigler 1976] is a mapping between all of the elements in the base and abstract models. A TSM preserves both structure and behaviour. TSMs are well-suited for abstracting discrete
representations (e.g. graphs), but less so for continuous systems. A *partial systems morphism* (PSM) is a mapping between some subset of the elements in the base and abstract models. In contrast to a TSM, all structure and behaviour is not necessarily preserved in a PSM. Sensory (visual) and cerebral abstraction are also discussed; unlike the previous five techniques, these do not define any mappings. Sensory abstraction aims to produce a model that is convincing to an audience, but without the attendant complexity of a mapping technique, for example, particle systems simulating fire or explosions [Reeves 1983]. Cerebral abstraction relates to the way in which humans reason about models. Other methods of abstraction include geometric model abstraction, where complex geometric elements are approximated by simpler ones [Clark 1976, Feiner 1985].

It is important that abstractions are evaluated in order to determine their utility. Fishman and Kiviat [Fishman 1967] define three components of evaluation as verification (ensure the model is consistent and behaves as intended), validation (test the model against the real system to assess similarities and differences), and analysis (ensure the output data is correctly interpreted). Fishwick [Fishwick 1988] defines an abstraction method as being valid by dint of its definition (i.e. if the definition of the method is valid, then the method itself is valid). An abstract model is considered valid if it can be either validated empirically or produced from a valid base model using a valid abstraction technique. An example of an empirical validation of an abstraction model could be the percentage of human observers who found the model convincing. Fishwick argues that abstraction models should be formalised whenever possible.

### 2.4 Abstraction in software engineering

Abstraction is employed in software engineering to help manage the complexity of software systems. For example, a diagram may be used as an abstraction to illustrate the principal components of a system. A number of different types of abstraction are used in software engineering. *Functional or procedural abstraction* allows a package of program functionality to be considered as a ‘black box’ with a clearly defined interface and its implementation hidden [Alexandridis 1986, Liskov 1986]. *Iteration or action abstraction* is used to express repeated patterns of program behaviour, such as loop constructs [Zimmer 1985, Liskov 1986]. *Data abstraction* is based on the idea of ‘abstract data types’, which allow data to be stored and manipulated through a defined interface without concern for how the raw data is represented [Guttag 1977, Ledgard 1977, Shaw 1984]. *Process abstractions* are similar to data abstractions, but include a thread of control [Alexandridis 1986]. In the context of knowledge-based OO logic programming, Park defines *object abstraction* as the combination of knowledge abstraction (models of knowledge base representation and control), data abstraction, and connection abstraction (models of object hierarchy and communication) [Park 1991].

### 2.5 Abstraction in software visualisation

Abstraction is crucial in software visualisation to allow the large quantities of information involved to be comprehended usefully. A recent case study of dynamic visualisation tools found that the various tasks typically involved in software
comprehension and reverse engineering efforts are best addressed at different levels of abstraction [Pacione 2003a, Pacione 2003b]. The work also showed that most extant dynamic visualisation tools operate at only one or two such levels (as measured on the five-level abstraction scale proposed in that work). Consequently, it is currently necessary to utilise several tools in combination in order to address satisfactorily the full range of activities. The multifaceted abstraction model proposed in this report seeks to address this issue by considering the full range of abstraction levels. Future work will include an investigation of the abstraction techniques applicable to software visualisation.
3 Static and dynamic information

3.1 Static software comprehension techniques

Software comprehension techniques can be classified as either static or dynamic. Static techniques analyse a system by examining the source or object code of the system under investigation. Static techniques can help in understanding the relationships between classes in a system, and in identifying the system architecture [Müller 1993]. They can also give some idea of the potential behaviour of the system. Software systems written in procedural languages are well suited to analysis with static techniques, as the program code maps relatively clearly to the program’s execution, though there are still significant limitations when issues such as dynamic memory allocation and pointers are considered. In contrast, aspects of the object-oriented paradigm, such as polymorphism, overloading, and dynamic binding, make it more difficult to gain a complete understanding of an object-oriented software system using static techniques alone. Gamma et al. state, “An object-oriented program’s run-time structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program’s run-time structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa. […] With such disparity between a program’s run-time and compile-time structures, it’s clear that code won’t reveal everything about how a system will work.” [Gamma 1995, pp. 22-23]

3.2 Dynamic software comprehension techniques

Dynamic software comprehension techniques analyse a software system by extracting information from the system as it is executing. Dynamic techniques can help to illustrate the interactions between objects in a target system, the flow of control between the system’s components, and the system structure. Dynamic software comprehension techniques address many of the shortcomings of static techniques in the comprehension of object-oriented software systems. A potential disadvantage of dynamic techniques is that they can consider only a subset of the software system’s possible behaviour. While static techniques can analyse the entire system, dynamic techniques analyse only the behaviour evident in the execution trace. It is the responsibility of the analyst to ensure that a suitably representative trace (or combination of traces) is selected for analysis.

The case study by Pacione found that some tasks are more readily answered using static or dynamic information [Pacione 2003a, Pacione 2003b]. Again, a combination of tools would be necessary to address all of the questions posed. The three-dimensional model proposed in this report seeks to address this issue by considering both static and dynamic information. The combination of static and dynamic information will allow the broad coverage of static techniques to be combined with the detail of dynamic techniques.
4 Effective presentation techniques for visualisation

4.1 Diagrams for describing software

The goal of software visualisation is to present information about the software system under investigation to the analyst in a format that is useful in helping them to achieve their software comprehension tasks. A variety of diagram types for describing software systems have been proposed in the literature and implemented in CASE (computer-aided software engineering) and visualisation tools. A selection of these are listed in Table 4.1, and discussed below.

4.1.1 Structured design diagrams

Before object-oriented techniques became popular in the early 1990s, a number of diagrams for supporting the traditional structured design process had been proposed. These included Petri nets [Petri 1962], Nassi-Shneiderman diagrams [Nassi 1973], entity relationship diagrams [Chen 1977], control flow diagrams [Hatley 1987], data flow diagrams [Pressman 2000, Sec. 12.4.1], data structure diagrams [Pressman 2000, Sec. 13.4.7], and statecharts [Harel 1990].

4.1.2 Object-oriented diagrams

The advent of the object-oriented paradigm produced a new set of diagrams. These included Booch diagrams [Booch 1994] and message sequence charts (MSCs) [ITU-T 1996]. A popular set of OO diagrams is that defined by the Unified Modeling Language (UML) [Rumbaugh 1999, OMG 2003]. UML version 1.5 defines a set of nine diagrams for describing various aspects of the analysis, design, and implementation of software, which are popular during the forward engineering process. These diagrams consist of boxes representing entities (e.g. classes, objects, components), connected by arcs representing relationships (e.g. inheritance, communication, dependency). An example UML sequence diagram for the Singleton design pattern [Gamma 1995, pp. 127-134] is shown in Figure 4.1. Selonen et al. [Selonen 2001] discuss transformations between UML diagram types. Burd et al. [Burd 2002] describe an experiment demonstrating that animation aids understanding of UML sequence diagrams.

UML models are essentially graph-based, and basic graphs (with one type of node and one type of edge), such as call graphs, can also be used to represent software (e.g. in the Program Explorer tool [Lange 1995a]). MSCs were a precursor to UML sequence diagrams, while UML statechart diagrams are derived from Harel’s statecharts. Further description and discussion of these diagram types is given in [Pacione 2003a].

The UML diagrams described above are implemented in many popular CASE tools. A number of additional diagrams that are not part of the UML standard, such as robustness analysis diagrams and business process diagrams, as well as diagrams intended specifically for modelling real-time systems, such as system context
diagrams, system architecture diagrams, and event sheet diagrams, and XML (XML structure diagrams), are also available in some tools.

![UML Sequence Diagram for Singleton Design Pattern](image)

**Figure 4.1** An example UML sequence diagram for the Singleton design pattern.

### 4.1.3 Recent literature


As stated in Section 2.5, previous work has revealed that the diagrams used in the extant dynamic visualisation tools address only a single abstraction level, or a small range [Pacione 2003a, Pacione 2003b]. Arranging diagrams in an interrelated hierarchy encompassing the entire range of abstraction levels would increase their
utility and aid comprehension, as all levels of abstraction could be conveniently addressed.

Table 4.1 A selection of diagrams for describing software.

<table>
<thead>
<tr>
<th>Structured design diagrams</th>
<th>UML extension diagrams</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic graph</td>
<td>Robustness analysis diagram</td>
</tr>
<tr>
<td>Petri net</td>
<td>Business process diagram</td>
</tr>
<tr>
<td>Nassi-Shneiderman diagram</td>
<td>Real time modelling</td>
</tr>
<tr>
<td>Entity relationship diagram</td>
<td>System context diagram</td>
</tr>
<tr>
<td>Control flow diagram</td>
<td>System architecture diagram</td>
</tr>
<tr>
<td>Data flow diagram</td>
<td>Event sheet diagram</td>
</tr>
<tr>
<td>Data structure diagram</td>
<td>XML modelling</td>
</tr>
<tr>
<td>Statechart</td>
<td></td>
</tr>
<tr>
<td>Pre-UML OO diagrams</td>
<td>XML structure diagram</td>
</tr>
<tr>
<td>Booch diagram</td>
<td></td>
</tr>
<tr>
<td>Message sequence chart</td>
<td>Recent SE literature</td>
</tr>
<tr>
<td>UML diagrams</td>
<td>Execution pattern [De Pauw 1998]</td>
</tr>
<tr>
<td>Class diagram</td>
<td>Reflexion model [Murphy 2001]</td>
</tr>
<tr>
<td>Object diagram</td>
<td>Story board diagram [Fischer 2000]</td>
</tr>
<tr>
<td>Sequence diagram</td>
<td>SoftArch diagrams [Grundy 2000]</td>
</tr>
<tr>
<td>Collaboration diagram</td>
<td>Virtual reality [Knight 2000]</td>
</tr>
<tr>
<td>Component diagram</td>
<td>[Martin 2002]</td>
</tr>
<tr>
<td>Deployment diagram</td>
<td>[Riva 2002]</td>
</tr>
<tr>
<td>Activity diagram</td>
<td>Visualization in contexts [Yin 2002]</td>
</tr>
<tr>
<td>Statechart diagram</td>
<td>Polymetric views [Bertuli 2003]</td>
</tr>
<tr>
<td>Use case diagram</td>
<td>DRT [Chan 2003]</td>
</tr>
</tbody>
</table>

4.2 Views for software comprehension

Diagrams, such as those described in the previous section, are used to illustrate models of software. Different views of a software model are possible - these views are implemented using diagrams. We propose that there are six possible arrangements of views onto a software model, illustrated in Figure 4.2, namely: (a) a single view illustrating a single facet; (b) multiple independent views illustrating a single facet; (c) multiple interdependent views illustrating a single facet; (d) a single view illustrating multiple facets; (e) multiple independent views illustrating multiple facets; and (f) multiple interdependent views illustrating multiple facets. This categorisation distinguishes view arrangements by the number of views (one or multiple), the number of facets (one or multiple), and their relationship (independent or

1 A facet in this context is taken to mean a (interesting) property of a software system, such as its structure or behaviour [Jahnke 2002].
2 The views are independent in the sense that there is no coordination between them. Two models of the same system may be implicitly dependent on each other unless they refer to disjoint parts of the system.
interdependent) The remainder of this section describes these arrangements in more
detail. Further discussion of the tools described in this section is available previous
work by Pacione [Pacione 2003a].

4.2.1 A single view illustrating a single facet

This arrangement illustrates a single facet of the software system in one view. A
single facet may not illustrate all of the information necessary for comprehension, but
for a specific task it may be sufficient. A single view provides the analyst with only
one perspective of the facet under investigation, which may restrict exploration. The
reference implementation of the Dali workbench described in [Kazman 1999] and
jRMTool [Murphy 2001] implement this arrangement. In the case of these tools, either
structural or behavioural facets can be visualised.

4.2.2 Multiple independent views illustrating a single facet

This arrangement uses multiple views to illustrate a single facet of the software
system. Multiple views give the analyst a number of perspectives of the facet, and
may improve the navigability of the model (cf. Baldonado et al.’s ‘Rule of Diversity’ and ‘Rule of Complementarity’ [Baldonado 2000]). However, the lack of relationships between the views can cause the analyst cognitive difficulties in reconciling the multiple views and transferring information between them (cf. Baldonado et al.’s ‘Rule of Parsimony’ [Baldonado 2000]). An example of this arrangement would be the use of a number of single view, single facet tools in combination to visualise a single facet from multiple views. For example, the Dali and jRMTool tools mentioned above could be used in combination to provide two independent views of structural or behavioural information.

4.2.3 Multiple interdependent views illustrating a single facet

This arrangement also illustrates a single facet of the software system using multiple views. In this case, the interdependency between views alleviates many of the cognitive difficulties inherent in the previous arrangement (cf. Baldonado et al.’s ‘Rule of Self-Evidence’ and ‘Rule of Consistency’ [Baldonado 2000]). Such interdependent arrangements are typically implemented using a Model-View-Controller architecture [Krasner 1988] to maintain synchronisation between the views and with the model. Scene [Koskimies 1996], architecture-oriented visualization [Sefika 1996], ISVis [Jerding 1997], Sced [Koskimies 1998], Ovation [De Pauw 1998], AVID [Walker 1998], Gaudi [Richner 1999], Jinsight [De Pauw 2002], and Collaboration Browser [Richner 2002] implement this arrangement. The facet in these tools illustrates behavioural information.

4.2.4 A single view illustrating multiple facets

This arrangement presents multiple facets of the software system in a single view. Multiple facets present more information to the analyst, which may help with comprehension of the software. However, compressing all the information into a single view can lead to information overload and reduced comprehensibility (cf. Baldonado et al.’s ‘Rule of Decomposition’ [Baldonado 2000]). The implementation of story board diagrams described by Jahnke et al. [Jahnke 2002] is an example of this arrangement. The tool described illustrates structural, behavioural, and data facets.

4.2.5 Multiple independent views illustrating multiple facets

This arrangement uses multiple views to illustrate multiple facets of the software system, with no interaction between the views. While this arrangement combines the benefits of multiple facets and multiple views, the lack of relationships between the views can cause difficulties in comprehension as described in Section 4.2.2 above. An example of this arrangement would be the use of a number of single view, single facet tools in combination to visualise multiple facets from multiple views. For example, the Dali and jRMTool tools mentioned in Section 4.2.1 could be used in combination to provide two independent views of structural and behavioural information.
4.2.6 Multiple interdependent views illustrating multiple facets

This arrangement also uses multiple views to illustrate multiple facets, with the addition of interrelationships between the views. This arrangement has the same advantages as the previous one, but the interdependency between views aids comprehension as described in Section 4.2.3 above. This arrangement is employed in Kruchten’s 4+1 View Model [Kruchten 1995] and in work by Hofmeister et al. [Hofmeister 1999] to illustrate structural, behavioural, and (in Kruchten’s work) data facets. The Program Explorer [Lange 1995] and Shimba [Systä 2001] visualisation tools and the Together CASE tool [Borland 2004] implement this arrangement. These tools illustrate both structural and behavioural facets.

It appears from the foregoing discussion that an arrangement of multiple interdependent views illustrating multiple facets of a software system is the most desirable arrangement of views for software comprehension. Multiple views give a variety of different perspectives on various facets of the software, while the interdependency between the views aids cognition. Such an arrangement would allow software to be described conveniently using a set of diagrams, such as those discussed in Section 4.1, illustrating relevant information at appropriate levels of abstraction. The use of multiple views in visualisation is discussed in more detail by Baldonado et al. [Baldonado 2000].
5 Effective techniques for exploring and querying visualisations

A crucial factor in the usefulness of a visualisation system is the ease with which the analyst can interact with the visualisation to obtain the information they require. We classify the two principal types of navigation technique observed in the extant visualisation tools as exploration and querying.

5.1 Exploration

A system employing the exploration technique presents the visualisation to the analyst and allows them to explore it freely. Although giving the analyst complete freedom to explore the visualisation, the large volume of information typically generated by dynamic analysis can make it difficult to find the cogent information required for the analyst’s tasks. The complexities inherent in the object-oriented paradigm compound this issue. Tools such as ISVis and Together utilise the exploration technique.

5.2 Querying

A system employing the querying technique allows the analyst to specify queries to be applied to the visualisation and then view the results. Queries can be specified in a textual or visual query language, such as SQL [ANSI 1998] or MURAL [Reiss 2002] respectively, or using a GUI. This approach can help the analyst to focus the visualisation on the information pertaining to their specific tasks. However, the analyst must know enough about the system to be able to form useful queries. Tools such as Gaudi, Collaboration Browser, and BLOOM [Reiss 2001] utilise the querying technique. Gaudi uses a textual query language, Collaboration Browser uses a GUI, and BLOOM uses a visual query language.

5.3 Guided navigation

There is a third possibility that has not been observed in the extant visualisation tools that we term guided navigation. A system employing guided navigation would assist the analyst in achieving their goals by suggesting likely lines of enquiry. A wizard-based approach may be suitable for this technique.

In practice, some systems employ a combination of the exploration and querying techniques. Such an arrangement combines the flexibility and scope of exploration with the focussing power of the querying technique. Guided navigation, possibly using wizards, is an interesting and complementary alternative.
6 A multifaceted, three-dimensional abstraction model for software visualisation

6.1 Background

Recent work to assess the capabilities of dynamic visualisation tools found that no single tool examined was capable of satisfying slightly more than half of the typical software comprehension and reverse engineering tasks set [Pacione 2003a, Pacione 2003b]. However, if all five of the tools in the study were used in combination, it should be possible to address 13 out of the 15 tasks.3 However, such an arrangement of multiple independent views would cause the analyst cognitive difficulties in reconciling the multiple views and transferring information between them, as described in Section 4.2.2 above. It is clear from these results that a tool combining the desirable properties of the individual tools in the previous study would perform well in these representative software maintenance tasks. It would therefore be reasonable to expect that such a tool would be useful in real world software maintenance.

The distinguishing properties of the tools in the previous study were the extraction, analysis, and presentation techniques of the tools, and the abstraction levels of the visualisations they produced. The extraction technique determined whether the tool used statically or dynamically extracted information about the software system. Analysis techniques applied to the extracted data included abstraction, pattern recognition, selective instrumentation, and suspension/resumption of tracing. Presentation techniques were graphs, MSC-based, UML, or textual. Abstraction level was measured on an ordinal scale from 1 (microscopic) to 5 (macroscopic). Extraction, analysis, and presentation techniques, and abstraction levels are discussed further in the previous work [Pacione 2003a, Pacione 2003b].

6.2 Research hypothesis

We propose that a model that supports visualisation of software through a range of abstraction levels that incorporate structural and behavioural views and integrates statically and dynamically extracted information will provide effective support for comprehension across a wide range of maintenance tasks.

6.3 A meta-model

In order to combine the benefits of these alternative approaches, this report proposes a multifaceted, three-dimensional abstraction model for software visualisation. Similar to the abstraction scale proposed by Pacione [Pacione 2003a, Pacione 2003b], the first dimension of the model consists of a number of abstraction levels from microscopic

3 The remaining two tasks involved automatic framework hotspot and design pattern detection. Though they are amenable to visualisation, these are non-trivial tasks that require a high level of analyst interaction.
to macroscopic. This arrangement allows the analyst to explore the software system at the level(s) of abstraction appropriate to the comprehension task they are undertaking. The second dimension of the model consists of a number of facets [Jahnke 2002], each representing some property of the system. The use of interrelated facets allows the analyst to examine a property of the software system individually or in combination, allowing them to focus the visualisation on the information appropriate to their query.

Each abstraction level of each facet is a view and consists of a name, a description, a set of entities, a set of relationships between those entities, and a set of diagrams that illustrate software at that abstraction level of that facet. This arrangement will provide the analyst with a clearer view of the software under analysis. We propose the meta-model shown in Table 6.1. The principal challenges associated with this model are the way in which information extracted from the software system will be represented, how view hierarchies will be generated from this information, and the definition of inter- and intra-hierarchy relationships between views. It will also be important to identify which views are useful for a variety of comprehension tasks.

Table 6.1. The multifaceted abstraction meta-model.

<table>
<thead>
<tr>
<th>Abstraction Level</th>
<th>Facet A</th>
<th>Facet ...</th>
<th>Facet Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name_{An}</td>
<td>Name_{n}</td>
<td>Name_{zn}</td>
</tr>
<tr>
<td>n</td>
<td>Description_{An}</td>
<td>Description_{n}</td>
<td>Description_{zn}</td>
</tr>
<tr>
<td>(macroscopic)</td>
<td>EntityTypes_{An}</td>
<td>EntityTypes_{n}</td>
<td>EntityTypes_{zn}</td>
</tr>
<tr>
<td></td>
<td>RelationshipTypes_{An}</td>
<td>RelationshipTypes_{n}</td>
<td>RelationshipTypes_{zn}</td>
</tr>
<tr>
<td></td>
<td>Diagrams_{An}</td>
<td>Diagrams_{n}</td>
<td>Diagrams_{zn}</td>
</tr>
<tr>
<td>n-1</td>
<td>Name_{An-1}</td>
<td>Name_{n-1}</td>
<td>Name_{zn-1}</td>
</tr>
<tr>
<td></td>
<td>Description_{An-1}</td>
<td>Description_{n-1}</td>
<td>Description_{zn-1}</td>
</tr>
<tr>
<td></td>
<td>EntityTypes_{An-1}</td>
<td>EntityTypes_{n-1}</td>
<td>EntityTypes_{zn-1}</td>
</tr>
<tr>
<td></td>
<td>RelationshipTypes_{An-1}</td>
<td>RelationshipTypes_{n-1}</td>
<td>RelationshipTypes_{zn-1}</td>
</tr>
<tr>
<td></td>
<td>Diagrams_{An-1}</td>
<td>Diagrams_{n-1}</td>
<td>Diagrams_{zn-1}</td>
</tr>
<tr>
<td>...</td>
<td>Name_{A...}</td>
<td>Name_{...}</td>
<td>Name_{Z...}</td>
</tr>
<tr>
<td></td>
<td>Description_{A...}</td>
<td>Description_{...}</td>
<td>Description_{Z...}</td>
</tr>
<tr>
<td></td>
<td>EntityTypes_{A...}</td>
<td>EntityTypes_{...}</td>
<td>EntityTypes_{Z...}</td>
</tr>
<tr>
<td></td>
<td>RelationshipTypes_{A...}</td>
<td>RelationshipTypes_{...}</td>
<td>RelationshipTypes_{Z...}</td>
</tr>
<tr>
<td></td>
<td>Diagrams_{A...}</td>
<td>Diagrams_{...}</td>
<td>Diagrams_{Z...}</td>
</tr>
<tr>
<td>2</td>
<td>Name_{A2}</td>
<td>Name_{2}</td>
<td>Name_{Z2}</td>
</tr>
<tr>
<td></td>
<td>Description_{A2}</td>
<td>Description_{2}</td>
<td>Description_{Z2}</td>
</tr>
<tr>
<td></td>
<td>EntityTypes_{A2}</td>
<td>EntityTypes_{2}</td>
<td>EntityTypes_{Z2}</td>
</tr>
<tr>
<td></td>
<td>RelationshipTypes_{A2}</td>
<td>RelationshipTypes_{2}</td>
<td>RelationshipTypes_{Z2}</td>
</tr>
<tr>
<td></td>
<td>Diagrams_{A2}</td>
<td>Diagrams_{2}</td>
<td>Diagrams_{Z2}</td>
</tr>
<tr>
<td>1</td>
<td>Name_{A1}</td>
<td>Name_{1}</td>
<td>Name_{Z1}</td>
</tr>
<tr>
<td>(microscopic)</td>
<td>Description_{A1}</td>
<td>Description_{1}</td>
<td>Description_{Z1}</td>
</tr>
<tr>
<td></td>
<td>EntityTypes_{A1}</td>
<td>EntityTypes_{1}</td>
<td>EntityTypes_{Z1}</td>
</tr>
<tr>
<td></td>
<td>RelationshipTypes_{A1}</td>
<td>RelationshipTypes_{1}</td>
<td>RelationshipTypes_{Z1}</td>
</tr>
<tr>
<td></td>
<td>Diagrams_{A1}</td>
<td>Diagrams_{1}</td>
<td>Diagrams_{Z1}</td>
</tr>
</tbody>
</table>
6.4 A model for object-oriented systems

Table 6.2 shows an instantiation of the meta-model in Table 6.1 for object-oriented systems. We have chosen five levels of abstraction for this example; the program code can be considered to be at level 0 as it is the least abstract representation of the software. We have selected structure, behaviour, and data as the three facets, as these are the principal elements of typical OO systems. Classes, packages, and files provide structural abstractions; procedures, functions, and – in OO systems – methods and interfaces provide behavioural abstractions; and abstract data types provide data abstractions. (Jahnke et al. [Jahnke 2002] also use these three facets.) Each named view consists of a description, a set of entities and relationships, and example diagram types that can be used to illustrate information from the facet at the specific level of abstraction. It is intended that the analyst will be able to move conveniently between these views during the course of their investigation in order to examine the information relevant to their task. The views selected are intended to represent the information that an analyst would find useful during software comprehension.

Diagrams (with the exception of storyboard diagrams) that can illustrate information in more than one facet appear at the same level of abstraction in each facet of the model, though this is not a requirement. Each facet need not have the same number of abstraction levels. There are no diagrams at level 5 or 1 of the structure facet. This is because the system structure is not relevant at a business level (only behaviour and the data it operates on are specified), and the internal structure of classes is not relevant or visible outside the class. There are no diagrams at level 4 of the behaviour facet. This is because the behaviour distribution is dictated by, and therefore encapsulated in, the structure distribution. There are no diagrams at level 3 of the data facet. Abstract data types are typically described using textual descriptions, algorithm pseudocode, and specific pictorial representations. Robustness analysis diagrams bridge between levels 5 and 2: they relate business entities to classes. System context diagrams bridge between levels 5 and 3: they relate business entities to components.

Table 6.2. An instantiation of the multifaceted abstraction model for OO systems.

<table>
<thead>
<tr>
<th>Abstraction level</th>
<th>Structure</th>
<th>Behaviour</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 (macroscopic)</td>
<td>Business structure</td>
<td>Business behaviour</td>
<td>Business data</td>
</tr>
<tr>
<td></td>
<td>The structure defined by the high-level business goals of the system</td>
<td>The behaviour defined by the high-level business goals of the system</td>
<td>The data defined by the high-level business goals of the system</td>
</tr>
<tr>
<td></td>
<td>{}</td>
<td>{BusinessEntity}</td>
<td>{BusinessEntity}</td>
</tr>
<tr>
<td></td>
<td>{}</td>
<td>{BusinessRule}</td>
<td>{DataDependency}</td>
</tr>
</tbody>
</table>

This is because SBDs illustrate data and behaviour at a low level, but are contained within a package. The SBD implementation considered in this report is that described in [Jahnke 2002].
| 4 | System structure deployment | System behaviour distribution | Data distribution |
|  | {Component, Machine} | {} | {DataObject, Machine} |
|  | {Dependency, Containment} | {} | {Dependency, Containment} |
|  | {Deployment diagram} | {} | {Deployment diagram} |

| 3 | System architecture | Component interaction | Abstract data types |
|  | The structural relationships between the system’s high-level components | The behavioural relationships between the system’s high-level components | The abstract data types used to encapsulate the system’s data |
|  | {Component} | {Component} | {} |
|  | {Dependency} | {Usage} | {} |
|  | {Component diagram, system context diagram, system architecture diagram, reflexion model, story board diagram} | {System context diagram, system architecture diagram, reflexion model, [Martin 2002]} | {} |

| 2 | Inter-class structure | Inter-object interaction | Physical implementation |
|  | The structural relationships between the system’s classes | The behavioural relationships between the system’s objects | The classes used in the physical implementation of the system’s data structures |
|  | {Class} | {Object} | {Class} |
|  | {Inheritance, Implementation, Aggregation, Composition} | {Invocation} | {Inheritance, Implementation, Aggregation, Composition} |
The third dimension of the abstraction model consists of static and/or dynamic analyses of the software. As discussed in Section 3, static analyses have broad coverage but less detail, while dynamic analyses are more focussed and more detailed. In this three-dimensional model, the width of static analysis can be combined with the depth of dynamic analysis, without their attendant disadvantages. This is achieved by the combination of analyses. Combining a single static analysis with several dynamic analyses results in a visualisation that is both detailed and broad in its coverage. A combination of multiple dynamic analyses (without static analysis) could also be used to achieve this to an extent. The principal challenge with respect to this aspect of the model is the way in which statically and dynamically extracted information is combined and presented.

The multifaceted, three-dimensional abstraction model is illustrated in Figure 6.1. This model is an example of the ‘multiple interdependent views illustrating multiple facets’ arrangement described in Section 4.2.6 above. The model will be refined as the research progresses. The interrelationships between facets and analyses are not shown explicitly; these will be defined as part of the formalisation process described in the future work. A metamodel such as the UML metamodel [OMG 2003] or the Dagstuhl Middle Metamodel (DMM) [Lethbridge 2004] may be useful in defining the model and its interrelationships.
6.5 Examples

Examples of the structure, behaviour, and data abstraction hierarchies are given in Figures 6.2 – 6.4 respectively. It would be possible to synthesize a fourth facet that combines the three existing facets and represents the integration of structure, behaviour and data in a single hierarchy. However, as discussed in Section 4.2.4 above, this can lead to information overload and reduced comprehensibility. It would instead be more useful to allow the analyst to define their own views by combining information from the three existing facets.
Figure 6.2. An example of the structure abstraction hierarchy.
```java
int myMethod(int a, int b) {
    System.out.println("Start");
    if (a==b)
        return 1;
    else {
        System.out.println("2");
        return 2;
    }
}
```

Figure 6.3. An example of the behaviour abstraction hierarchy.
class ResearchStudent extends Person
{
    String studNum;
    ResearchStudent(Person p, String s)
    {
        super(p);
        studNum = s;
    }
    setStuNo(String s) {...}
    setTopic(String t) {...}
    setTitle(String t) {...}
    submitThesis() {...}
}

Figure 6.4. An example of the data abstraction hierarchy.
7 Related work

7.1 Software visualisation

Prior research in software visualisation appears to focus on specific sub-problems of comprehension. For example, Pacione’s case study found that the reflexion model technique [Murphy 2001] was useful for answering large-scale questions, but could not address small-scale tasks [Pacione 2003a, Pacione 2003b]. Similarly, the Jinsight tool [De Pauw 2002] was more successful in answering questions relating to the dynamic behaviour of the system than its static structure. The Shimba tool [Systä 2001] combines statically and dynamically extracted information, integrates structural and behavioural information, and employs a limited range of abstraction levels. A case study found the ability to use static information to focus the dynamic analysis particularly useful. Slicing the static representation using dynamic information was found to be useful in determining the cause of behaviour. Raising the level of abstraction of the dynamic visualisation using static abstractions was useful in understanding communication between high-level components. These tools are described in more detail in previous work [Pacione 2003a, Pacione 2003b].

The idea of interrelated hierarchies for separate facets of software is discussed briefly by Jahnke et al. [Jahnke 2002]. In contrast to the approach described in this report, they propose the use of story board diagrams [Fischer 2000] to represent the integration of structure, behaviour, and data at a single level of abstraction. The multifaceted abstraction model described here maintains three distinct, interdependent hierarchies of abstraction. The multiple interdependent view approaches of Kruchten [Kruchten 1995] and Hofmeister et al. [Hofmeister 1999] illustrate various aspects of software at a single level of abstraction.

7.2 Abstraction

Lee and Fishwick [Lee 1996] propose a taxonomy of structural and behavioural abstraction for continuous real-world systems. It is interesting to note the parallels between the model they describe and the model proposed in this report. In their model, structural abstraction is subdivided into data and model abstraction. Model abstraction is further subdivided into homogeneous and heterogeneous abstractions. Homogeneous abstractions use a single model type at each level of abstraction in a multi-layered model. Heterogeneous abstractions allow a variety of model types to exist in a single structure. Behavioural abstraction can be either static or dynamic. Static behavioural abstraction considers only the steady state output of the system. Dynamic behavioural abstraction uses time-dependent input and output information.

Lee and Fishwick’s abstraction taxonomy has many similarities with the multifaceted, three-dimensional abstraction model presented in this paper. Both model structural and behavioural information. Lee and Fishwick present data abstraction as a subdivision of structural abstraction, rather than as a separate hierarchy. This is comparable to the situation when modelling information extracted from an OO system, as (the implementation of) the system data will be encapsulated by the system.
structure. The subdivision of behavioural abstraction into static and dynamic abstractions can be likened to our use of statically and dynamically extracted information.
8 Summary, conclusions, and future work

8.1 Summary

This report has described software visualisation and the open issues identified in previous work [Pacione 2003a, Pacione 2003b]. Abstraction, static and dynamic information, and effective ways of presenting, exploring, and querying visualisations were discussed. A multifaceted, three-dimensional abstraction model for software visualisation was proposed, which is intended to address the issues identified. This model consists of interdependent abstraction hierarchies for the structure, behaviour, and data abstractions of a software system, and integrates statically and dynamically extracted information. Related work was also discussed and compared with our approach. A summary of this work is given in [Pacione 2004].

8.2 Feasibility

There already exist techniques for statically and dynamically extracting information from software systems. Most current visualisation tools address either structural or behavioural information, though few represent both or attempt to combine this information. Some visualisation tools also present information at more than one level of abstraction. These facts demonstrate that the approach proposed here is a feasible solution.

8.3 Key research challenges

There are a number of key research challenges associated with this solution. One such challenge is the way in which the visualisation information will be stored as a model, and how this will be used to generate view hierarchies. Another challenge is the definition of the inter- and intra-hierarchy relationships between the views. Abstraction techniques applicable to software visualisation will also be investigated. Identifying which views are appropriate and useful for which comprehension tasks is a further challenge. The way in which statically and dynamically extracted information is combined and presented will also require investigation.

8.4 Future work

Future work is to include the development of a formalism to describe the multifaceted, three-dimensional abstraction model proposed, and to allow transitions between levels and facets, and the combination of information from different views. A meta-model such as the UML metamodel [OMG 2003] or the Dagstuhl Middle Metamodel (DMM) [Lethbridge 2004] may be suitable for this purpose. The selection of facets and abstraction levels must be validated. The usefulness of the multifaceted, three-dimensional model in software comprehension will be evaluated. An empirical approach based on the questions used in previous work [Pacione 2003a, Pacione 2003b] could provide useful results. Multiple subjects and studies may be used if
practicable. The creation of new diagrams for describing software for comprehension may be explored. Effective methods for exploring and querying visualisations may also be investigated. A further possibility is the construction of a software visualisation tool that implements the multifaceted, three-dimensional model proposed (or aspects of it), and the evaluation of its use in software comprehension.
References


28


