Understanding Object-Oriented Frameworks

Submitted to the Department of
Computer and Information Sciences,
University of Strathclyde,
Glasgow.
For the degree of doctor of philosophy.

By Douglas Samuel Kirk

August 2005

Abstract

Frameworks are an attractive form of reuse due to the reductions in cost and time they can provide to software projects. Despite their benefits the size and complexity of most frameworks makes understanding how to use them difficult, lessening their appeal. In addition documentation to support framework reuse often lacks experimental validation and there is a poor understanding of what artefacts must be documented to increase the effectiveness of documentation techniques.

This thesis describes an empirical investigation into framework documentation. Its aim is to identify the major problems of reuse and the impact of current documentation techniques on these problems. A qualitative approach is employed and four major reuse problems are identified as barriers to reuse: understanding the functionality of components; understanding the interactions between components; the mapping from the problem domain to the framework implementation and understanding the architectural assumptions in the framework design.

The effectiveness of current forms of documentation is evaluated using these problem categories and, as a result, the extension of two existing forms of documentation are suggested, namely a pattern language and a set of micro architectures. An in-depth, qualitative analysis of both techniques evaluates the strengths and weaknesses of their support for framework understanding, whilst confirming the significance of the four problem categories. The analysis shows that the pattern language developed in this thesis has some capability to support mapping type problems although it was often overridden by developers' previous knowledge to the detriment of the solution. The micro architecture notation provides support for simple interaction and functionality queries but was not able to address large scale interaction problems within the framework.

The thesis concludes that the combination of a pattern language and micro architecture documentation can provide useful support for framework reuse but both require modification to become more effective. The thesis also concludes that the evaluation of framework documentation is an essential activity for the advancement of framework comprehension. It serves as an example to encourage other researchers to perform more evaluation of framework documentation in the future.

Acknowledgements

This has been the hardest thing I have ever done. That seems a rather pathetic thing to say given that this is only a PhD thesis but it is true. It wasn't hard in a day to day sense – in fact I was surprised how straightforward and enjoyable the work proved to be. Instead it was hard because it seemed endless. I am delighted that I have now finally managed to complete this work and although it is not perfect I am content with what I have achieved.

The thesis would never have been completed if it wasn't for the love, help and support offered by many people over the course of the past few years. Primarily I have to thank my two supervisors Murray and Marc. They have always believed in me and in my capability to do this task. From day one they have always taken my thoughts and ideas seriously (even when not deserved) and offered constructive advice and criticism which has taken me to this point. Most importantly I believe they have helped me to change my attitude to learning from merely doing enough to get a pass, to trying my very best to learn all I can. I hope I haven't caused you too much stress during this time. Thank you both.

I would like to thank EPSRC for funding this research and also the participants who took part in the experimental studies. Without them this work would not have been possible. Thank you very much. I also want to thank my wife, Fi. She has always believed in me and made many sacrifices so that I would have the time to work on my thesis. Her love and moral support kept me going and are a large contribution to the existence of this thesis. She has also given me a beautiful son called Ben. Darling I love you. Thank you so much for being in my life. I would also like to thank my Mum and Dad for encouraging my interest in computing and providing much love and support. Finally I would like to thank the staff of the Department of Computer and Information Sciences, University of Strathclyde. I am very grateful for the wonderful, supporting environment which they have provided. I would also like to specifically thank my fellow EFoCSers Neil, Michael and Matt for all their advice, encouragement and friendship during this time. I would like to add a special thank you to Dr Ian Ferguson for help with financial support during the final months of this thesis and also to Dr Al Dunsmore for showing me the ropes when I was still a 'green' first year.

Doug Kirk

23 / 5 / 2005

Publications

Technical Reports

Kirk, Douglas. 2001. **Patterns For HotDraw**. Technical Report (EFoCS-38-2001). University of Strathclyde, UK.

Kirk, Douglas. 2001. **Identifying The Problems of Large Scale Reuse: A Personal Case Study**. Technical Report (EFoCS-41-2001). University of Strathclyde, UK.

Kirk, Douglas. 2001. **Understanding Object Oriented Frameworks: An Exploratory Case Study**. Technical Report (EFoCS-42-2001). University of Strathclyde, UK.

Kirk, Douglas. 2001. **Framework Reuse: Process, Problems and Documentation**. Technical Report (EFoCS-43-2001). University of Strathclyde, UK.

Kirk, Douglas. 2002. **JHotDraw Pattern Language**. Technical Report (EFoCS-47-2002). University of Strathclyde, UK.

Kirk, Douglas. 2002. **Evaluation of a Pattern Language for JHotDraw**. Technical Report (EFoCS-48-2002). University of Strathclyde, UK.

Kirk, Douglas, Marc Roper and Murray Wood. 2002. **On the Creation of Pattern Languages for Framework Reuse**. Technical Report (EFoCS-49-2002). University of Strathclyde, UK.

Publications

Kirk, Douglas, Marc Roper and Murray Wood. 2002. Defining the Problems of Framework Reuse. In the Proceedings of the 2002 Computer Software and Applications Conference held in Oxford, UK, August, 2002, 623-626 and 282-283. IEEE Computer Society. .

Kirk, Douglas, Marc Roper and Murray Wood. 2005. Identifying and Addressing Problems in Framework Reuse. In the proceedings of the 2005 International Workshop on Program Comprehension held in St. Louis, Missouri, USA. May, 2005 77-86. IEEE Computer Society.

Contents

1 Understanding object-oriented frameworks

1.1. Introduction

Object oriented frameworks are large scale software applications that are designed for reuse. Recent studies (Fayad, Schmidt and Johnson 1999) (Moser and Nierstrasz 1996) suggest that frameworks can achieve significant levels of reuse and cause large reductions in development effort and time to market on software projects. Such benefits place object oriented frameworks in a prime position to replace bespoke application development as the mainstay of modern software development. Despite their utility, frameworks are complicated structures to learn and the effort and time spent gaining an understanding of how to use a framework often outweighs its potential benefit (Gamma et al 1994) (Johnson and Foote 1988). This thesis investigates why frameworks are so difficult to reuse and also how documentation techniques can be improved to help shorten this learning curve.

1.2. The importance of software reuse

Software reuse has been a goal of the software industry for the last forty years (McIlroy 1968). It is widely accepted within the community that reuse is an important step in the maturity of software development as an engineering discipline (Pressman 1994) (Meyer 1997), (Somerville 2001). Yet despite this opinion software reuse has had little impact upon how we build software today, with most development projects still building a significant proportion of their code from scratch.

1.2.1 Motivations for reuse

Software reuse provides several benefits to developers: it reduces cost, time and effort and it can also improve the quality of the software that is created (Pressman 1994). Reuse improves software quality because it is often the more experienced developers or domain experts who are asked to write reusable code (Meyer 1997). Their expertise helps to ensure that the correct design and implementation is chosen for each reusable component. Quality is also preserved because the components have a lifespan out-with any one project. Errors in a component are detected and removed as that component is reused across projects helping to improve its quality over time.

Reuse reduces the cost of application development even though reusable code may take longer to write than ordinary software. It can achieve this because reusable code only has to be written once and the cost of its creation can be amortised across many different projects.

Reusing software also helps to reduce the time taken to bring an application to market. When reusing code there is less software to write as existing parts can be assembled together to create the required functionality. This allows applications to be created in less time than developing solutions from scratch.

1.2.2 Limited uptake of reuse

The advantages of reuse are clear: it allows you to build software, faster, at less cost and with better quality than traditional software development. Yet software reuse is still not a common activity. Why isn’t more software being reused? There are many possible reasons for the lack of adoption, including: the unavailability of suitable code libraries, lack of confidence in the quality of the reusable code and differences in opinion about the context of reuse.

A simple practical barrier that can prevent software reuse is the lack of reusable components. If suitable libraries do not exist then the developer has no option but to create the required material. Perversely in many cases the opposite can also be a problem. The availability of too much reusable code can make it difficult to select the relevant class from a range of alternatives. Even when a class can be found other factors can limit its reusability: It might be prohibitively priced or have been written in a different language from the host system or for a different hardware platform in each case rendering the software useless to a potential re-user.

Software reuse also requires a certain amount of trust to be effective. Developers reusing a software library have to trust that it has been constructed correctly and that it features a suitable set of functionality to properly meet their application’s needs. This trust is required because library developers often do not supply source code to third parties making it difficult for re-users to make alterations or corrections to the code. Developers also have to accept that reusing other people’s abstractions may introduce some inefficiency into their design. Reusable code has to cater to a wide audience and hence tends to support a broader and more general set of features than a specific solution might implement. For some this compromise is hard to accept and they would prefer to create their own solution rather than reuse code. Commentators in the literature have dubbed this inability to trust other peoples code as the ‘not invented here’ syndrome (e.g. Meyer 1997).

A less obvious inhibitor to code reuse is the difficulty of reusing abstractions across different reuse contexts. This occurs because even in a highly modular paradigm, like object orientation, reusable code is not completely isolated from the remaining code within the program. Instead separation is achieved by programming to interfaces (Meyer 1997). These divide the system into its constituent parts but at the same time create dependencies between the interfaces. Such dependencies can become a problem during reuse because they define an expectation about how a component should be reused. When a component is inserted into a new context those assumptions, expressed in its interface, may no longer hold true and the component might not be able to operate correctly in its new environment.

To illustrate this point, consider a Person component. In a banking system this class might consist of a person's name, address and account number hidden behind a suitable interface, in a medical system the same component might have to contain information about height, weight, prescribed medication, etc. and have a correspondingly different interface. The concept of a Person is used very differently by these two scenarios and there is no one definition that is suitable for both. This is a problem for re-users as it limits the applicability of classes to domains similar to their original target. It can also be difficult to infer from a component alone what its intended domain may be. This problem also extends to the non functional properties of a component. For example the runtime performance of code that is suitable in one context may not be appropriate even for a similar context because its performance requirements are different. These problems reduce the opportunities for code reuse in object-oriented systems.

This is the reality of code reuse for many software developers today. Finding appropriate code to reuse, having confidence in its quality, assessing its performance and being aware of its limitations are all significant issues that hamper their ability to reuse code. Developers cannot simply pick up a piece of software and understand how to reuse it and so they don't. In many situations reuse has become something analogous to creating software documentation. It is something that everyone wants and can see the benefits of but few people are actually prepared to do! Object-oriented frameworks were created to address some of these problems while preserving the benefits of software reuse.

1.3. The reuse of object-oriented frameworks

An object-oriented framework is a special type of software system which has been created to be highly flexible in order to support a wide range of specific applications. It provides an incomplete implementation of an application, sometimes called a code skeleton, which developers reuse to gain a head start when developing their own applications. Frameworks can also provide class libraries to provide code that anticipates common application requirements. This further reduces the amount of effort required by application developers to fill out the partial application. There is compelling evidence which suggests that the use of such frameworks can make a significant improvement to the amount of reuse that occurs and to the corresponding befits that are achieved by the development process (Moser and Nierstrasz 1996).

1.3.1 Framework skeleton

The code skeleton of a framework defines the range of applications that a framework can support. It describes the architectural core of an application which includes gaps, or areas of flexibility, that can be fleshed out later to create a complete application. A typical framework skeleton is constructed from a collection of interfaces and abstract classes, which together specify the structural and behavioural relationships that the framework supports. Frameworks also frequently employ design patterns (Gamma et al. 1994) within the skeleton to create its flexible areas. Framework developers talk about the framework having a domain. This acts as a boundary allowing developers to decide whether a particular type of application can be created from a given framework skeleton.

1.3.2 Framework class libraries

A framework usually supports its code skeleton by supplying a number of class libraries which contain ready made abstractions to flesh out the gaps within the skeleton. These abstractions often provide common behaviour with respect to the framework domain and can be reused directly within applications or serve as a starting point for users to define their own abstractions via sub-classing.

1.3.3 Types of framework

Frameworks can be divided into a number of different types. Some frameworks are known as black box frameworks while others are described as white box (Roberts and Johnson 1996). The colour refers to the amount of control the application developer has over framework customisation. With a white box framework the developer has complete access to the framework source code. They can understand the implementation details of the framework and can modify and extend parts of the framework in their customisations. In a black box framework modifications are much more restricted, typically source code is not available and developers must reuse the components that are provided with the framework to configure it for different circumstances. Some critics have argued that white box and black box frameworks lie at either end of a continuum and seldom occur in practice. They argue that most framework customisation occurs somewhere in between these two extremes. The term grey box has been used to describe such frameworks (Johnson and Foote 1988).

Another distinction that can be drawn amongst object-oriented frameworks regards the role of the framework within a system. Some frameworks control all aspects of an application. For example a call centre framework will enable the creation of a number of call centre applications. Such frameworks are commonly known as application frameworks. Other frameworks known as utility frameworks are more constrained, specialising in only one aspect of an application. For example graphical user interfaces such as Swing/AWT (Sun Microsystems 2005b) or .Net Windows Forms (Microsoft 2005c) focus on the user interface of an application and ignore the rest of the system. Another form of framework is known as an infrastructural framework. Such systems define services which other applications can make use of during execution. They include systems such as CORBA (OMG 2005a), which define an architecture for distributed and platform independent code sharing.

The different types of framework all have their place within software reuse. Each may be expected to differ in terms of the problems they cause re-users and the requirements they have for documentation. This thesis does not have the scope to address all types of frameworks equally. Instead it focuses only upon a white box application framework, the insights gathered in this context being representative of the fundamental problems that apply to all types of framework. The results of this thesis can be used as a platform for future investigations to consider the unique needs of each alternative framework type.

1.3.4 The growth in popularity of frameworks

Frameworks first became popular with the Smalltalk object-oriented language during the eighties (Johnson and Foote 1988). For example, a small framework called Model View Controller (MVC) (Krasner and Pope 1988) was often used to create user interfaces for Smalltalk applications. MVC splits the elements of a user interface into three logical components: a model, a view and a controller. The model captures the state of the system, the view displays a representation of the model and the controller allows a user to modify the model's state. It also defines the relationships between the components such that a model never knows what views are dependent upon it. A model can have multiple views and each view creates a set of controllers that can alter the model. The popularity of the MVC framework has led to it transcending its framework roots to become an architectural pattern (Buchsman et al. 1996) that is widely advocated for the construction of graphical user interfaces across object-oriented languages. The origin of frameworks within the graphical user interface community (e.g. (Krasner and Pope 1988), (MacApp 1984), (Wienand, Gamma and Marty 1988)) initially caused some developers to believe that frameworks were only useful for developing user interfaces but gradually their more general applicability was realised and their use has spread to a much wider range of application domains. A large number of third party application frameworks new exist. These address a wide range of domains, including network communications (Schmidt 2005), graph modelling (White, et al. 2005), drawing editors (Gamma and Eggenschwiler 2005), (Vlissides 1990), call centre applications (Graham Technology 2005) and network management (Cisco Systems 2005).

Recently more mainstream object oriented languages such as Java and C# have also promoted software frameworks. Both languages come with extensive combinations of class libraries and frameworks that address common programming activities. For example database access (ADO.Net (Microsoft 2005a), JDBC (Sun Microsystems 2005d)), user interface design (.Net Windows Forms, Swing/AWT) and the creation of web services (ASP.Net (Microsoft 2005b), Java Server Pages (Sun Microsystems 2005c)). Today frameworks have become commonplace and they are frequently used in the construction of modern software applications.

1.3.5 Discovery costs

The learning curve associated with reusable code can be considered as a cost of learning to use that product. Such costs are sometimes labelled discovery costs (Mancl, Opdyke and Fraser 2002). Frameworks can be argued to help such discovery costs because they embody assumptions about a domain which represents the wisdom of the developers who created the framework. This can help other developers to jumpstart their problem solving within a domain by using the design clues embedded within the framework code. A relatively recent panel at the OOPLSA conference (Fraser 1997) were asked to what extent they felt that the technologies of patterns and frameworks helped to mitigate discovery costs. A range of interesting opinions were expressed by the panellists. In general the majority believed that both technologies had the potential to help with discovery costs for a domain but only after their own initial learning curve had been traversed

In particular frameworks were noted as having considerable learning curves and of being helpful only for more experienced framework re-users (in other words those who have already scaled the learning curve). Concerns were also expressed that discovery costs would only be improved by a framework if it correctly predicted the variability that was required by a particular problem (the framework design has to be flexible enough to allow a solution to be created). Despite these concerns there was considerable enthusiasm for the idea that frameworks codify domain knowledge and help to transfer design experience onto subsequent users. This was argued as most keenly felt during the design stages of a project where a framework can help provide a shared language for team members to communicate about a design.

1.3.6 The difficulty in reusing frameworks

Despite their growing popularity object-oriented frameworks are difficult to reuse (Bosch et al. 1999). This limits the community who are prepared to learn how to use them and it can result in frameworks being reused inappropriately by confused application developers.

An obvious obstacle to reuse is the amount of material that must be understood before a framework can be successfully instantiated. Many frameworks are large, often containing hundreds or thousands of classes, and they are often incomplete, containing abstract classes and using design patterns to create flexibility. This places a considerable burden on potential re-users as they have to absorb a lot of information about the behaviour of the framework and its scope for modification before it can be reused. In addition, whether they are developing with a framework or not, developers have to learn about the application’s domain (Bosch et al. 1999). This provides the language used to describe the abstractions and range of functionality they can expect the system to provide. Becoming familiar with this vocabulary places an additional challenge for framework developers making reuse even harder to achieve.

Developers must also understand and accept the design rationale used to create parts of the framework (Beck and Johnson 1994). This can be difficult because often the need for flexibility results in unintuitive relationships within the framework. Another problem can arise when developers attempt to reuse combinations of frameworks together. Often application frameworks expect to be reused in isolation and dictate through their class skeleton the main flow of control of the application. This is known as the inversion of control principle (Fayad and Schmidt 1997). Inverted control causes problems when several such frameworks are used together or in conjunction with an application because their main control loops compete against each other preventing the frameworks from operating correctly. This situation requires careful mediation by the application developer in order to resolve the conflict (Mattsson, Bosch and Fayad 1999).

1.4. Describing frameworks

Understanding object-oriented frameworks is a difficult problem. The solution may lie in improving the quality of support provided by software documentation. It has the potential to describe both the implementation of the software and to explain how it should be used. Many forms of documentation have been proposed in the literature including source code browsers (Robitaille, Schauer and Keller 2000), JavaDoc (Sun Micro-Systems 2005), UML diagrams (OMG 2005b), design patterns (Gamma et al. 1994), pattern languages (Johnson 1992), (Lajoie and Keller 1994) and example based learning (Shull, Lanubile and Basili 2000). The number and diversity of approaches suggested in the literature is encouraging as it suggests that there is much to describe about object frameworks. The abundance of techniques is also a problem as it can be difficult to select an appropriate subset of techniques with which to document a framework. In part this is because available documentation lacks a critical appraisal of its utility and in part because there is a lack of understanding about what information framework re-users actually require. This thesis addresses both of these issues.

There is currently little culture of evaluation amongst the proponents of framework documentation. New techniques are proposed in the literature with little evidence to support how well they have performed during reuse. The lack of evaluation has created a situation where numerous techniques exist but nobody knows which, if any, are useful. Such ignorance is stifling progress in framework documentation as researchers have no common understanding about where to best devote their research effort when attempting to address framework reuse problems. It also prevents techniques being adopted by framework developers as there is no incentive for them to subscribe to a particular form of documentation without some prior evidence of its utility.

Researchers also lack an understanding of the range problems that occur during framework reuse. Existing techniques have been proposed from opinion, or the experience of a few developers. Documentation is also often taken from other areas and reapplied within the context of framework reuse (Butler and Dénommée 1997). Such approaches do not consider the entire gamut of problems that occur during reuse. Instead each technique addresses a small subset of problems while ignoring the others. Existing approaches have little regard for how such a fragmented understanding will fit together. A more complete understanding of reuse problems would allow an investment in features that users actually need. It could also allow more effective combinations of documentation to be proposed, as techniques could be selected to minimise the redundant overlap of material.

1.5. Product line architectures

Software product lines are an alternative form of large scale reuse (SEI 2005) (Bosch 2001). A product line is a set of related applications which share a common set of design artefacts. This includes a range of material from domain models, test cases and documentation through to software architectures, design patterns and reusable code. The assumption behind product lines is that a company is often called upon to develop many similar versions of a software product. By explicitly designing their product to be a reusable asset they can make the transition between versions of the product easier to achieve and maximise the reuse of assets between applications.

Product lines differ from frameworks in a number of significant ways. Primarily they address a narrower application scope than a framework. While a framework addresses a domain of applications a product line might only address variations to one type of application in that domain. However, this more precise focus can help to promote greater reuse of assets and make more accurate predictions about cost and time scale of development. Product lines are also more holistic in the scope of their reuse. A framework provides reusable source code but a product line provides more. It typically includes design assets such as architectural models and test cases as well as reusable code. Finally the deployment of product lines differs from frameworks. Frameworks are typically developed by one team of developers and reused by another. There is a clear separation between those responsible for the framework code and those responsible for the application. In a product line architecture it is more likely that one group of people will be responsible for the creation and deployment of the product architecture. Separate teams might exist for individual application products but they will be contained in the one organisation and communication between those groups is likely to be better than in a framework situation. This makes it possible for each application to be adapted to its context and for common changes to filter back into the product line architecture.

Ultimately both frameworks and product lines provide support for large scale software reuse. Product lines provide greater infrastructural support but are more restrictive than object oriented frameworks in the applications they can produce. Frameworks are also more independent and distributable that product lines which can enable then to reach a wider audience of potential re-users.

1.6. Thesis outline

This thesis investigates and identifies the key problems of framework reuse. It does this by recording users as they worked with an application framework and constructing a profile of what information was required in order to reuse the framework code. The thesis evaluates many popular forms of framework documentation to determine how well they fare against the identified reuse requirements. It also investigates improvements to some existing forms of documentation and develops new techniques in order to provide more comprehensive support for framework reuse problems. The proposed documentation is evaluated to determine its utility.

1.6.1 Contribution to knowledge

This thesis contributes to knowledge in the following ways:

- **Identifies key problems of framework reuse**: It identifies a set of problems that occur during framework reuse. This provides a basis for evaluating documentation and also helps to identify combinations of documentation that might offer suitable support for framework reuse.
- **Evaluates framework documentation:** It evaluates many common forms of framework documentation. This identifies the relative merits of common approaches and provides insights into user reaction and opinion about using different types of documentation.
- **Improves existing forms of documentation**: It investigates alterations to existing forms of documentation that specifically address the identified problems. The modified approaches are described and evaluated to identify their strengths and weaknesses.
- **Provides guidance for future evaluations**: It provides guidance for the empirical evaluation of framework documentation. The thesis describes a number of qualitative approaches to documentation evaluation. This will help researchers performing similar work in the future.

1.6.2 Thesis Assumptions

This thesis makes some global assumptions about how frameworks are used and how the difficulty in understanding them should be addressed. This section enumerates those assumptions.

For the purposes of this thesis a framework is assumed to be a stable core suitable for a wide range of applications. This represents an ideal framework, one where the core of the design is fixed and end developers customise details to create a final design. Such designs are arguably more suited to documentation because the central core will be highly similar across implementations. The notion of stability is not true for every framework. Some are created and used in a more pragmatic fashion where approximate suitability for a problem leads to the framework being deployed but being heavily modified during the implementation. The findings of this work may not be as relevant to such situations.

This thesis also makes the assumption that improvements to comprehension are most likely to be discovered by leveraging documentation to teach users about a framework. Other approaches to reducing the difficulty of learning frameworks are possible, for example constraining the size and scope to make them easier to understand or utilising code generators to remove the need to work with source code. While such techniques deserve proper investigation this work contends that documentation is an obvious approach to the problem which is relatively easy to implement and likely to make substantial improvements to comprehension. This work therefore focuses on documentation but some of its findings, for example the problems that developers experience during frameworks reuse, will likely have general significance to all comprehension approaches.

1.6.3 Thesis structure

The remainder of this thesis is structured as follows:

- Chapter 2 presents a review of the literature, highlighting the different forms of documentation that have been proposed for framework reuse. It also summarises the limited attempts at evaluation that have been performed in this field.
- Chapter 3 presents an observational study that identifies the key categories of problems that arise reusing an object-oriented framework. The study captures the problems experienced by a number of framework re-users as they attempt to reuse an application framework. The study also collects anecdotal evidence about the capabilities of existing forms of documentation and the developers’ perception of them.

- Chapter 4 presents the creation of two new forms of framework documentation. This chapter extends an existing technique and explores the development of a novel form of documentation to provide better support for the documentation requirements identified earlier in the thesis.
- Chapter 5 presents a detailed qualitative evaluation of the proposed documentation. A protocol analysis is performed on a number of framework re-users equipped with the new documentation. This analysis yields a detailed account of their thoughts and actions during the course of framework development. Analysing this material provides insights into how the documentation has been used and whether it provided the support expected.
- The thesis concludes by discussing the implications of this work for future forms of framework documentation. It also provides practical advice for developers wishing to perform similar types of qualitative study and closes with a discussion about what research questions remain open and require further research in the future.

2 Framework documentation techniques

2.1. Introduction

There are many different types of documentation which claim to provide support for framework reuse. However, there is a scarcity of evidence to justify their selection and to evaluate their utility as framework documentation. This review critiques the available approaches, categorises them by type and identifies, where available, evidence to support their utility.

The review is organised into six categories of framework documentation (source code, micro architectures, macro architectures, hotspots, examples, and prescriptive techniques). The categories describe the major features that documentation might illustrate. In some cases these are general qualities that apply to any software system (e.g. source code); in other cases they apply specifically to features found within object-oriented frameworks (e.g. hotspots). The individual techniques discussed in this review are assigned to one of these categories. The categorisation is not intended to be exhaustive as it only characterises features found in existing forms of framework documentation. Future forms of documentation may exploit different aspects of the software and deserve new categories to describe them, nor are the categorisations strictly orthogonal. Documentation is pigeon-holed according to its dominant characteristic. It is quite possible that some documentation types apply across multiple categories but this review considers only the primary contributions of each form of documentation.

This review shows that there are a great number of promising techniques available to address framework reuse. In the majority of these cases, it also shows that there has been little justification provided for their creation. The review also highlights the lack of available evidence to identify useful forms of framework documentation. It concludes that the uncertainty about the capabilities of documentation is preventing future research from improving upon its quality and effectiveness for supporting framework reuse.

2.2. Source code

A framework is defined by its source code. This makes understanding the code an important part of understanding the framework. Source code is written primarily to be understood by a machine which can cause it to become verbose and pedantic for humans to understand. Frameworks also tend to be large pieces of software, featuring hundreds or thousands of classes which are contained in a number of files. This can make it difficult for developers to find areas of the source code that are relevant to their task and to remember details about its behaviour. Lajoie and Keller emphasise the importance of source code in framework comprehension (Lajoie and Keller 1994). They claim that source code is the ultimate reference for framework knowledge, *“If one wants to thoroughly understand a class and/or method, code inspection still remains the most precise and sure way”*. They also call for source code to be more tightly integrated with other forms of documentation to encourage traversal from the documentation into source code and back again.

Other researchers have suggested the use of tools to help make working with source code more manageable. Robitaille, Schauer, and Keller describe a tool called SPOOL which helps users to navigate through large collections of code (Robitaille, Schauer and Keller 2000). The SPOOL tool (**Figure 1**) was designed to assist with the navigation of large amounts of source code. Robitaille et al. argue that the volume of information within source code and its distribution across multiple files is a critical barrier to program comprehension. SPOOL is an enhanced source code browser that allows developers to query structural relationships within the code. Users can ask which classes call a method, inherit from a class or are composed together. Responses to queries are displayed both graphically via a UML-like class diagram and also as a list of classes. Queries can also be composed together to create more complicated questions (e.g. find all classes that inherit from class A <u>and</u> implement interface B). The authors claim that chaining queries together in this manner allows the user to understand the system at a higher level of abstraction than by merely browsing the code. Products similar to SPOOL are beginning to become popular in other areas of software development. Tools such as Together (IBM 2004), SNiFF (WindRiver 2005) and Eclipse (Eclipse 2005) are improving the ability of developers to navigate through large amounts of source code and to quickly locate relevant information within code.

As a mechanism to assist framework reuse source code browsers such as SPOOL appear to offer useful but incomplete support for framework problems. One particular weakness of such tools is that they are unable to offer any information beyond that already present within the code. It cannot, for example, offer advice about how components of a framework ought to be used. The strength of source code browsers lies in the ease with which a user can move between files and can produce views of related classes. This allows developers to maintain their train of thought while reading code across multiple files and also allows them to gather an accurate understanding of what material is available within the framework’s class libraries.

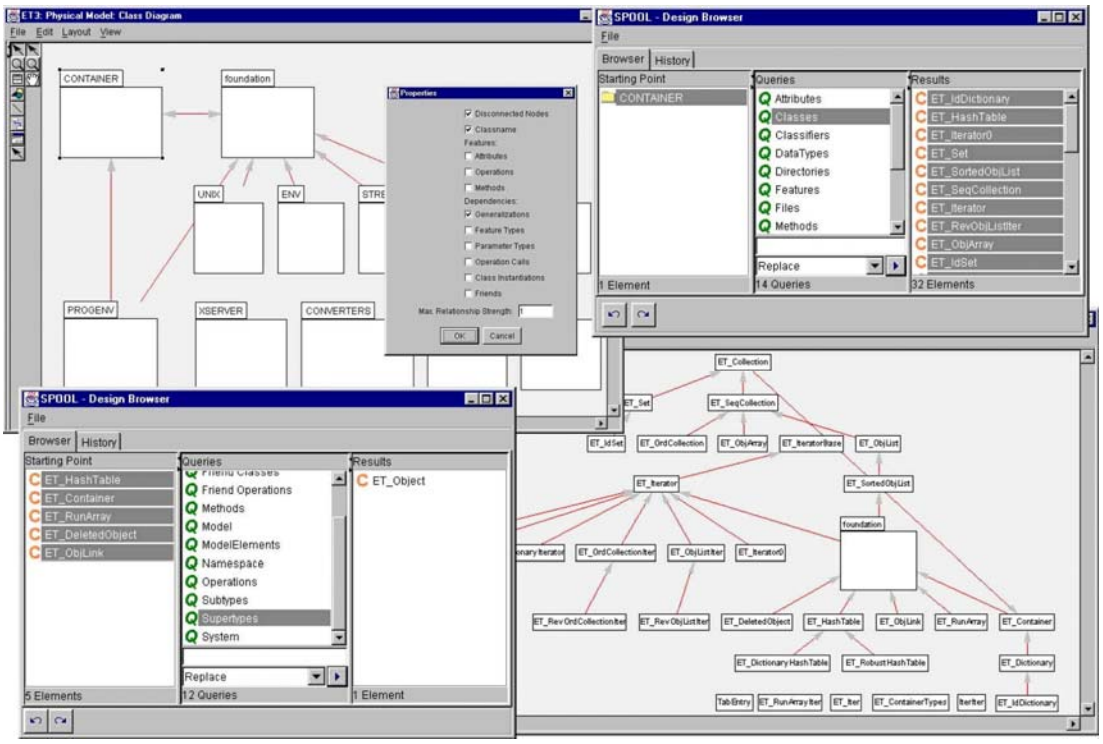


Figure 1: SPOOL Source code browser (Robitaille, Schauer and Keller 2000)

An alternative view of source code is provided by Sun’s JavaDoc tool. It emphasizes structural details, augmented with explanatory comments, to help users identify and understand parts of the source code. (Sun Microsystems 2005a). JavaDoc generates information for Java programs formatted as a set of HTML web pages. By default the tool will automatically extract class signatures and inheritance information from the source code. The tool also extracts specially formatted comments within the source code to provide explanations about the role of each class and the functionality offered by its methods. JavaDoc's advantage over some other forms of documentation is the relatively low amount of effort that is required to create the documentation and to keep it up to date. It also takes advantage of hyperlinks to relate classes together (for example one can view information about super classes and jump to their definition with a single click) this can make navigation through the class structures of a framework very easy to perform. The disadvantage of JavaDoc is that its main contribution arguably comes from the additional comments created by a developer. If these are missing or of poor quality then the resulting impact of the documentation is equally poor. Creating JavaDoc to a consistently high standard also increases its cost and makes it more difficult to produce. Nevertheless the benefits of JavaDoc, despite its reputation for variable quality, are such that it has become a common source of documentation and similar tools are available for several other mainstream languages (e.g. C++, C#, Perl).

Tool support provides assistance for the navigation of source code and limited support for its comprehension. Both the SPOOL and JavaDoc tools can help the user to navigate around the collection of classes that comprise a framework. They can also help the developer to gain an understanding of the static structure of the framework and the interfaces supported by its classes. Understanding the dynamic behaviour of a framework is not so well supported. JavaDoc provides some opportunity for comments to be used to explain the framework's functionality. However, there is no control over the content of the comments and these can vary considerably in their quality, sometimes providing good insight into a class or method, other times stating little more than the obvious. It seems likely that where such tools are available they will have a positive effective upon a developer's ability to understand and navigate through framework code.

2.3. Micro architectures

Object oriented frameworks feature a lot of internal communication between the classes that comprise the framework. Such communication is seen by many researchers to be a significant factor in the comprehension of large software systems (Booch 1994), (Lajoie and Keller 1994), (Gamma et al. 1994). This has resulted in a number of documentation techniques which attempt specifically to describe the interactions of software. This thesis considers such techniques collectively as micro architectures, i.e. small parts of a larger system.

Helm et al, describe a technique that encapsulates inter class communication (Helm, Holland and Gangopadhyay 1990). Their technique creates a structure called a contract which describes the communication protocol between classes (**Figure 2**). Contracts are described in a formal notation and comprise four sections: type obligations which describe the interfaces of collaborating participants (for example in **Figure 2** Subject must support calls to SetValue and Notify); causal obligations which describe the sequence of invocations between participating classes (e.g. in the figure Update -> Draw implies that calls to update will result in a subsequent call to draw); invariants which must be upheld by the contract and finally an instantiation section which describes the preconditions that must be true before the contract is valid. Each contract is composed of a series of interactions which complete some function within an application (whilst omitting any interactions which are superfluous to that functionality). This relates the behaviour of the source code to the behaviour of the application domain making it easier to identify which parts of the source code are responsible for a particular functionality.

```
Contract SubjectView
        Subject supports[
                value: Value
                SetValue(val:Value) → delta value{value = val}; Notify()
                GetValue(): Value → return value
                Notify() → (|| v: v ∈ Views : v → Update())
                AttachView(v:View) → {v ∈ Views}
                DetachView(v:View) → {v ∉ Views}
        ]
        Views : Set(View) where each View supports [
                Update() → Draw()
                Draw() → Subject → GetValue() {View reflects Subject.value}
                SetSubject(s:Subject) → {Subject = s}
        ]
        Invariant
                Subject.SetValue(val) → ( ∀ v : v ∈ Views : v reflects Subject.value)
        Instantiation
                (|| v : v ∈ Views : (Subject → AttachView(v) || v → SetSubject(Subject)))
End Contract
```

Figure 2: An interaction contract (Helm, Holland and Gangopadhyay 1990)

Helm et al. argue that contracts are an orthogonal structure to classes in the description of object-oriented systems. They name their approach *“interaction oriented design”* and suggest that collaborations should be defined first, before being factored into class definitions. Helm et al claim their approach is relevant to all object-oriented applications, but they also realise it has specific value to object-oriented frameworks commenting *“Frameworks define solutions in terms of interaction between abstract classes”*. They also refer to their experience of the Interviews (Linton, Vlissides and Calder 1989) and Unidraw (Vlissides 1990) frameworks, where they describe specific framework problems that they claim would have been prevented if the system had been documented in contract form (e.g. understanding the relationship between a scene and its contents in the Interviews framework). Contracts raise the profile of interactions within an object-oriented system. This makes interactions easier to identify and understand while having the additional benefit of relating the behaviour of the source code to the high level behaviour of the application. On the other hand, the formal notation of the contracts makes them somewhat awkward to understand and the ability to identify which interactions to document appears to be critical to the success of this technique.

Design patterns (Gamma, et al 1994), are primarily intended to support forward engineering, but they can also be used to document existing software. Patterns describe good solutions to commonly occurring problems in object-oriented design. For example the Observer pattern describes a mechanism to allow dependencies between classes without the classes involved explicitly knowing about each other, making it easier to alter classes while maintaining the relationship between them. Design patterns do not represent an actual implementation; instead they describe an generic solution, abstracted from several different examples. Thus patterns focus on communicating the design principles behind the source code rather than providing an implementation to copy blindly. As documentation this can help a user to understand how a section of the system is organised and provide an idealised notion of the interaction that occurs within a section of code.

Patterns have a close relationship with software frameworks. Apparently design patterns were originally discovered through Gamma et al.'s experience of developing object-oriented frameworks. They noticed that the requirement to create flexibility within frameworks often resulted in repeated arrangements of source code. These recurring sections of code eventually became what are now known as design patterns. This is significant for framework documentation because it implies that frameworks tend to have a large number of design patterns embedded within them. Therefore teaching users about design patterns ought to have a beneficial effect upon their ability to understand the structures that exist within framework code. This may be limited by the abstract nature of pattern descriptions which could make them difficult to recognise within the concrete implementations of a framework.

Lajoie and Keller similarly believe that interactions are important aspects of framework documentation (Lajoie 1993), (Lajoie and Keller 1994). They decompose framework interactions into units they call micro-architectures (to be contrasted with the more generic use of the term to classify all of the techniques in this section) which are described using a combination of design patterns (Gamma et al 1994) and contracts (Helm, Holland and Gangopadhyay 1990). Lajoie and Keller claim that understanding the framework in terms of such groupings is the principal difference which separates novice framework re-users from more experienced developers. They further argue that design patterns describe micro architectures in an abstract manner to *"ensure wide applicability".* Because of this design patterns *"are difficult to understand in isolation"* and contracts must be used as an intermediate representation to help re-users understand how a design pattern operates within the implementation provided by a framework. Lajoie and Keller also observe that there is not always a design pattern for every micro-architecture in a framework. In such cases they suggest that contracts be used on their own to help users understand the behaviour of the micro architecture. Unfortunately, Lajoie and Keller do not provide many examples of their approach and it is not clear from their description how the set of micro architectures used to describe a framework ought to be identified.

Lange and Nakamura describe a tool called Program Explorer which helps to identify occurrences of design patterns within framework code (Lange and Nakamura 1995). They argue that detecting design patterns is critical to framework understanding because it allows a lot of detail to be ignored without compromising the significant behaviour of the framework. Their tool combines a mixture of static and dynamic information to assist in the detection of patterns. Example applications are used to identify functionality of interest within the framework. The example application is then executed within the tool and the functionality of interest is exercised in the application. The resulting dynamic trace contains information about which classes of the framework are involved with the given functionality.

Lange and Nakamura point out that the dynamic trace for even a simple piece of functionality can be very large and difficult to understand. They provide an example to identify the code behind a slider mechanism in the Interviews framework. The resulting trace *"creates more than 3000 objects"* and contains *"at least twenty thousand events"* making it overwhelmingly large. To combat the scale of information Lange and Nakamura suggest filtering the information using knowledge of design patterns. In their approach the developer anticipates the existence of a design pattern within the system and uses knowledge about the structure and naming conventions of the pattern to filter the dynamic information, searching for relevant interactions.

They illustrate how their tool can be used to identify the Observer pattern in the slider example. Lange and Nakamura admit that their approach is ultimately one of trial and error, as design patterns are used as a hypothesis to identify areas of functionality within the example application. However, exploration of an example in this manner does allow a developer to gain familiarity with small areas of the system without becoming overwhelmed by extraneous detail. Perhaps a larger problem with their approach is the manual effort involved on the part of the user. They have to be familiar with a range of design patterns before they can use the tool, they have to be able to predict which patterns are likely to underpin functionality within the framework and they also have to overcome differences in naming methods and classes when searching for the patterns within the trace information. This high level of knowledge and involvement required beforehand excludes a significant number of developers from being able to use the tool to understand a framework.

Micro architecture approaches to framework documentation may help to address the large scale of object-oriented frameworks. By decomposing the framework into smaller subsections they facilitate the comprehension of the entire framework as each section can be understood in isolation from the others. They also help to relate the functionality of the framework to the structures which define that functionality within the source code, helping the developer identify the capabilities a framework offers and where to locate modifications within the source code. One weakness of micro architecture approaches is the lack of guidance to help identify meaningful subsections of a framework to describe.

2.4. Macro-architectures

Recently, considerable attention has been focused on describing software systems in terms of their architectural structure. This demand for a higher-level view of the system is appealing for framework comprehension. It presents the foundation of a mental model which subsequent investigation can fill out. Bass, Clements and Kazman provide the following definition of software architecture *“The software architecture of a program or computing system, is the structure or structures of the system which comprise software components, the externally visible properties of those components and the relationships among them”* (Bass, Clements and Kazman 1998).

Beck and Johnson use a collection of design patterns to explain a framework's architecture and justify its implementation (Beck and Johnson 1994). They argue that existing documentation focuses too much on the ‘how’ and not enough on the ‘why’ of design and they propose a modified form of design pattern (emphasising the motivation for a pattern) to address this. They define an architecture as *“the way the parts work together to make the whole”* and claim that frameworks are themselves a form of architectural documentation – the architecture being expressed as source code. The patterns are also adapted to describe implementation details for a framework making them less abstract and easier to understand. Beck and Johnson point out that the description provided by design patterns is only a rationalisation of the design and does not represent the actual development process. They argue that this kind of knowledge is similar to the mental model of a framework held by experienced developers. They claim that novice users can use their documentation as a replacement for experience helping them to understand and maintain the architectural relationships within the framework. Design patterns cannot address all aspects of a framework and this raises questions about the completeness of their approach. It may be possible that important aspects of the framework architecture go unreported by this technique simply because they are not related to a design pattern. Nevertheless this form of documentation does help to identify the location of patterns within a framework and describes the motivations behind their selection. Such information is likely to be helpful to developers seeking to uphold the design principles within a framework. This approach to documentation is further illustrated in the work of Odenthal and Quibledly-Cirkel (Odenthal and Quibledly-Cirkel 1997) and again by Beck this time with Gamma in (Beck and Gamma 1999).

Richner and Ducasse describe an approach to architectural documentation which uses a mixture of static and dynamic information to provide insight into the interactions that occur across a framework (Richner and Ducasse 1999). They argue that a predefined set of architectural views is too restricting and that a developer ought to be able to query the system to dynamically generate views as required. Their proposed approach uses a logic language to define a set of static and a set of dynamic facts about a program based on its syntax (i.e. what inheritance looks like, what a method invocation looks like). An execution trace of the program is produced which captures details of method invocations in a log file. The code and the execution trace of the system are then parsed and a database of facts created (which class is related to which, what methods a class has invoked, etc). The tool then allows queries to be written which extract information from the database and display it as a series of graphs (e.g. **Figure 3**).

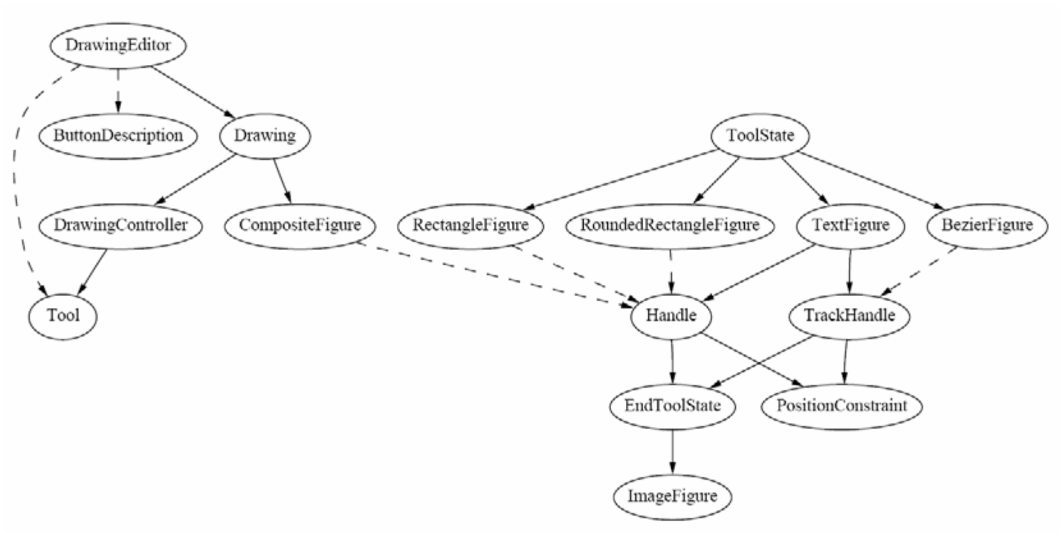
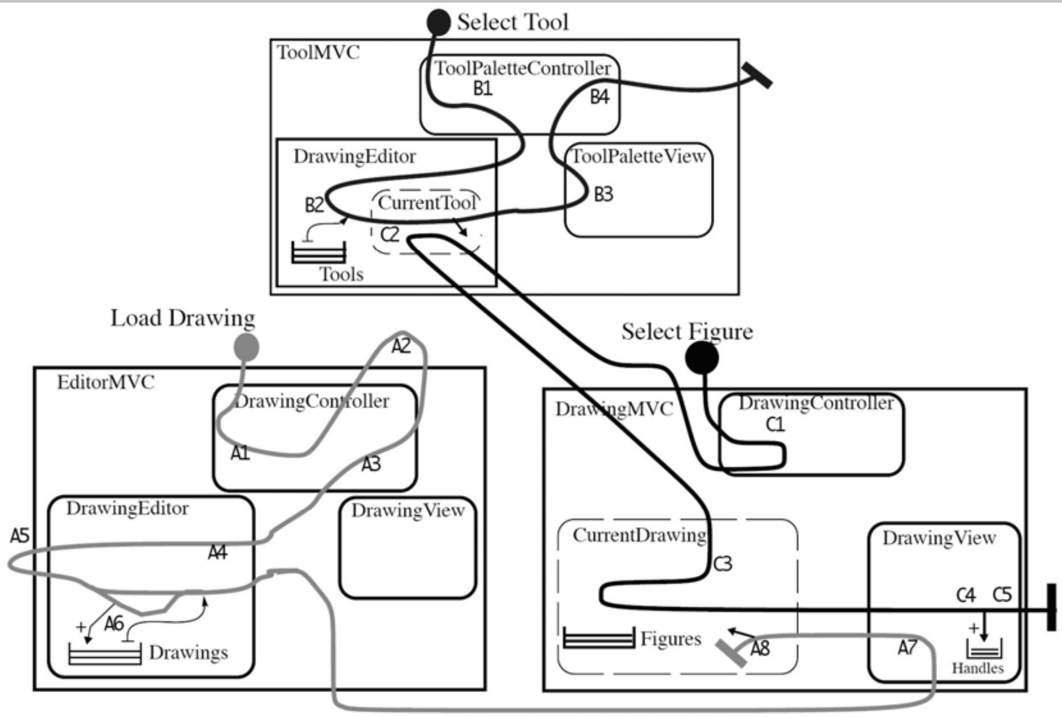


Figure 3: Graph of creation invocations from the HotDraw framework (Richner and Ducasse 1999)

Interestingly their technique can be used to represent information at different levels of granularity by abstracting information about low level events into more general categorisations (e.g. specific subclasses are abstracted to the roots of their hierarchies and only the interactions between these high level entities are shown). This allows a user to move between different views of the system depending on the queries they wish to answer. Richner and Ducasse suggest starting with interactions between components and working down to interaction between objects. This 'top down' approach is key to their strategy and they believe that developers are led from the coarse grained information to ask further queries which recursively descend through different layers of abstraction until the details are resolved. This approach would appear to place a lot of responsibility on the shoulders of the re-user. They must interpret each diagram and from that decide what query they want to make next. It is not clear whether developers are actually able to formulate successful queries in practice. There are also concerns about the dynamic information used to generate parts of their views. If the execution trace used is unrepresentative of normal operation then the information within the diagram may be incomplete or misleading in its description.

Buhr suggests another form of documentation to describe interactions at an architectural level (Buhr 1996). He proposes a notation called a use case map (UCM), which illustrates graphically how a use case flows through the components of a system (**Figure 4**). This differs from a conventional sequence diagram by focusing on the gross level communication between parts of a system rather than individual method invocations. Use case map diagrams contain hierarchies of boxes, which are used to represent the components of the system. These are most often classes but Buhr also describes a larger unit, which he calls a team, that conceptually relates clusters of classes together (perhaps reminiscent of micro architectures). The interaction between parts of the system caused by the use case is illustrated as a line that zigzags through the components in the map indicating interacting components. The line can split apart into multiple lines or merge together to indicate concurrency within the system. The final element of significance on a use case map is a short textual description, which provides insight into the actions that occur at various positions along the line. These are annotated on the diagram with a number, which is referenced in the textual description. Buhr shows that several different traces can be juxtaposed together onto a single diagram to illustrate how different use cases exercise the system.



Path *Load Drawing* Responsibilities. *C1*. inform current tool of mouse press; *C2*. react to mouse press. *C3*. find figure at cursor position; *C4*. create handles for the selected figure; *C5*. display handles.. *A1*. display menu; *A2*. choose menu item (load); *A3*. decode menu item; *A4*. display load drawing dialogue box; *A5*. choose drawing to edit; *A6*. create new drawing or select existing drawing and add to the use case path; *A7* update the drawing view; *A8*. install new current drawing.

Path *Select Tool* Responsibilities. *B1*. handle mouse press in tool palette; *B2*. find tool at cursor and put in CurrentTool slot; *B3*. receive update; *B4* highlight the currently selected tool.

Path *Select Figure* Responsibilities. *C1*. inform current tool of mouse press; *C2*. react to mouse press. *C3*. find figure at cursor position; *C4*. create handles for the selected figure; *C5*. display handles.

Figure 4: A use case map for HotDraw (Buhr 1996)

Buhr claims that use case maps allow re-users to stand back from the mass of implementation details to see the larger picture behind the application. He argues that this is beneficial because this coarse grained view is less subject to change than an understanding based on detailed interactions and that it helps identify important interactions which are usually lost within the detail. He points out that UCMs are not capable of explaining the complete behaviour of a system as they do not support enough knowledge of fine grained interactions. Use case maps appear to be accessible and easy to interpret. The major weakness of this approach is that it relies upon a set of use cases to illustrate completely the range of functionality on offer. Such information is difficult to provide for a framework as it can only describe use cases via instantiated applications which in turn can not completely illustrate the range of functionality supported by a framework.

Macro architectures encompass a variety of different types of documentation. From the pattern like descriptions of Beck and Johnson to the dynamically created graphs of Richner et al. there is considerable variation in the approaches that have been suggested. Other techniques such as use case maps suggest an overlap between architectural descriptions and software visualisations. In general the approaches all share a desire to communicate an overview or overarching description of the system being documented. Such approaches appear particularly relevant during early phases of framework reuse where there is a particular need to gain an initial familiarity with the structures of a framework.

2.5. Hotspots

Frameworks are abstract applications. They require developers to fill in the blanks during reuse to create concrete applications. This has led some researchers to consider documentation which seeks to describe the points of flexibility and customisation that exist within a framework. Pree coined the term ‘hotspots’ to describe these abstract areas of a framework *“A framework defines a high-level language with which applications within a domain are created through specialisation. Specialisation takes place at points of predefined refinement that we call hotspots”* (Pree 1999). Pree goes on to describe a number of patterns which he claims illustrate the possible modifications that can be made to a framework via a hotspot. He calls these patterns ‘meta-patterns’ which are variations of the Template pattern described in (Gamma et al. 1994). The meta-patterns describe two categories of modification, “*unification”*, which is essentially modification via inheritance, and “*separation”*, which is modification via composition. Pree suggests that hotspots are difficult to create within a framework and that they emerge over time as a framework matures.

Pree’s hotspot viewpoint is not shared by all framework developers. Codenie, De Hodt, Steyaert and Vercammen disagree with the scope of information required to make a modification to a framework (Codenie et al. 1997). They argue that it is not sufficient simply to know where to make a modification, there is also a need to understand the impact of a change upon the existing system. They illustrate their argument with an example of an inheritance modification. They suggest a modification to a collection class to notify another class whenever an item is added into the collection. The modification is made by overriding an AddItem method of the collection but this creates a problem with another method of the class, AddItems, which adds multiple elements to the collection at one time (and for efficiency reasons does not call AddItem when adding to the collection). If the developer does not modify both methods (which Codenie et al. argue may easily be overlooked) the notification will not occur correctly. The above problem could have been identified easily through inspection of the source code but Codenie et al. argue that *“…in practice inspecting* *the code to reuse a class is undesirable; this kind of analysis should be feasible at the design level"*. Instead they propose a notation called a reuse contract which they claim helps to describe the intra-hierarchical dependencies between subclasses in a framework.

A reuse contract describes the dependencies between methods in a class and between subclasses in a hierarchy (Steyaert et al 1996). They are textual descriptions similar to class interfaces that are associated with a framework class. Contracts are divided into two sections which list the abstract and concrete methods that are defined within the associated class (**Figure 5**). The concrete methods also list the methods that they call within the class (no inter class behaviour is described). Reuse contracts do not describe all of the methods belonging to a class including only those methods which are considered part of the design rather than the implementation (the categorisation of each method is at the discretion of the documentation writer). Steyaert et al. recommend that implementation methods be removed from the class interface to prevent the possibility of naming conflicts with future subclasses (i.e. by creating inner classes to model implementation detail). Contracts can also be extended using three different operators (and their reverse operators), concretisation, extension and refinement (reverse operators: abstraction, cancellation and coarsening). These operators are used to describe explicitly how subclasses modify their parent classes within the framework. Steyaert et al. argue that such information can help to describe the intentions of each subclass allowing developers to detect conflicts when a subclass breaks an existing contract. Steyaert et al.'s work suggests that framework modifications may at times be more wide ranging in their scope than previously considered by other descriptions. Reuse contracts appear to provide a detailed understanding of framework hierarchies which may help to preserve the existing architecture and avoid unexpected interactions between framework code and new modifications.

```
Reuse Contract DragableView
is an extension of View
        Abstract
                Draw
        Concrete
                Update   {Draw}
                Drag     {Draw}
End Reuse Contract
```

Figure 5: A reuse contract (Steyaert et al 1996)

Fontoura, Pree and Rumpe describe another approach to hotspot documentation using an extension of UML called UML-F (Fontoura, Pree and Rumpe 2000). This documentation modifies standard UML by the inclusion of a set of tags which indicate variation points (hotspots) within a design. Fontoura et al describe three types of variation points which can apply to class diagrams: variable methods, extensible classes and extensible interfaces. Variable methods are denoted with a {variable} tag which is inserted next to the method name. This indicates that the method is open for modification via sub-classing or composition. Extensible classes, denoted with an {extensible} tag, indicate that subclasses can add new methods to an interface within the framework and extensible interfaces, marked with an {incomplete} tag, indicate hierarchies where new implementations can be added to increase the options available to re-users. All of these variation points can be further quantified by the addition of a {static} or {dynamic} tag which indicates whether a modification is required to apply at compile time or runtime. Finally OCL (object constraint language) comments can be added to a diagram to define instantiation restrictions which supply additional constraints onto the type of modification that can occur (i.e. to specify that an attribute of a class will not be modified by any new method added via an {extensible} tag). An example of their notation can be seen in **Figure 6**.

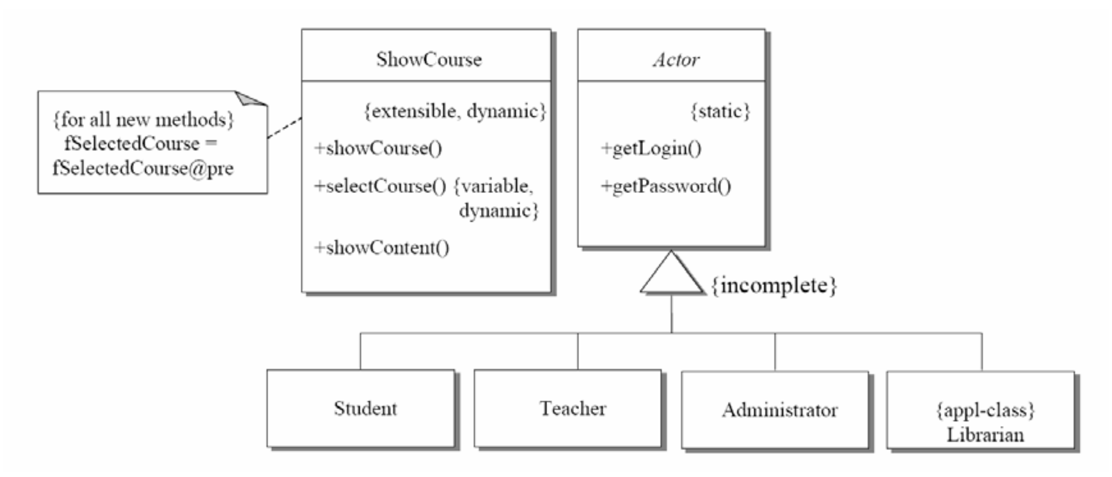


Figure 6: A UML-F class diagram (Fontoura, Pree and Rumpe 2000)

Fontoura et al see a critical role of their work as reducing the amount of code that re-users have to be exposed to during a modification. *"It is quite cumbersome that framework users today often need to browse the framework code, which generally have complex and large class hierarchies, to try to identify the variation points"*. Despite this it could be argued that the suggested tags present information that is already obvious to a re-user, as most hierarchies in a framework are meant to be extended, methods to be overridden and interfaces widened. The benefit of not having to search through lots of code for this information is merely a by-product of using UML. Perhaps the strongest contribution of this work is the inclusion of OCL constraints upon potential modifications. These appear to offer some support to preserve the architectural relationships of the framework.

Hotspots are a phenomenon of framework development. They identify areas of the framework that are intended to change and dictate the types of change that are possible. Researchers appear to disagree on how these areas of a framework ought to be described. Pree et al. seem to favour a more laissez-faire approach, being content to identify potential hotspots, while Steyaert et al. wish to formalise the contract between the framework and the application code. Further work is required to identify exactly what role hotspots play during reuse and what form of support will be most effective.

2.6. Examples

Examples are an effective and widely used learning strategy in many areas of education. It is therefore unsurprising to find them recommended as a form of documentation to explain framework reuse. Examples illustrate a framework's capabilities. They can show the scope of modifications that are possible with a framework and can illustrate best practice in the manner in which modifications are implemented. Johnson considers examples to be an important part of framework documentation (Johnson 1992). He cites many frameworks that use examples to assist re-users, (i.e. MVC, MacApp and UniDraw) and claims that examples play a *"key role"* in framework documentation. Johnson believes that *"Studying examples is a time honoured way of learning a framework"* as he argues that they illustrate the flow of control, the capabilities and the design of object-oriented frameworks. He attributes the effectiveness of examples to their description of concrete structures as opposed to the less tangible abstract behaviour offered by a framework alone.

Schneider and Repenning believe that well designed example applications can help address the lack of explicit design performed by framework re-users (Schneider and Repenning 1995). They also claim that current example applications are encouraging framework re-users to implement cosmetic features of the framework before more important functionality. This, they argue, damages the design of the application as its core functionality has to be retrofitted into a design already complete with cosmetic and often inappropriate details. Schneider and Repenning also argue that a stronger focus is required upon the design process of framework reuse and they propose a risk-based approach to address this problem.

In their approach modifications with the greatest risk to success are performed first to reduce the cost of failure when a modification cannot be achieved. To support this idea they claim that a specific form of example application, called a paradigmatic application, should be used to illustrate the important underlying mechanisms of a framework. Paradigmatic applications are concrete examples of framework customisation that differ from traditional examples in their focus on what Schneider and Repenning refer to as *"abstract reusable mechanisms"* (useful combinations of primitive framework functionality). Each paradigmatic application defines one such reusable area within a concrete example and a set of alternative descriptions (or shallow analogies) that sketch out how that example could be implemented in a number of different applications. For example they describe one abstract reusable mechanism as *"propagating agents through a discrete space constrained by conductors"* and illustrate this mechanism in an example application of electricity flowing through a circuit. They then provide analogies for the flow of traffic, water and money in different application contexts.

Schneider and Repenning argue that their approach marks a clear distinction between the abstract framework code and the concrete application code, making it easier for re-users to adapt the example to their own application requirements. Finally they suggest that the shallow analogies should be presented in a form of textual overview that accompanies the example code. A limitation of this approach is the apparent difficulty in identifying good abstract reusable mechanisms to document with examples. Schneider and Repenning do not offer any explicit guidance but they do suggest that experience of several applications within the framework domain is important in order to detect the common modifications that occur within a framework.

Gangopadhyay and Mitra (Gangopadhyay and Mitra 1995) also advocate an example driven approach to framework learning. They describe a special type of example called an exemplar which they claim can help teach users about the architecture of a framework. An exemplar is an example application which instantiates at least one concrete class for every abstract class within the framework. Gangopadhyay and Mitra describe how exemplars can be used to support a top down learning process for framework comprehension. They argue that the exemplar should be understood as a complete entity before a user narrows in on the area they wish to modify. This they claim allows a user to better understand the responsibilities and relationships within the framework. Once an area of the exemplar has been identified for modification the user then has to search for an alternative class within the hierarchies of the framework. If an appropriate class can be found it can be reused directly otherwise a new class has to be created to fit the requirement.

They support their approach with a tool, called Objchart, which displays class diagram and sequence diagram information for an executing exemplar. It allows a user to click on a method of interest in the class diagram to view a sequence diagram illustrating its behaviour. It also allows users to view class hierarchies of available framework components. Gangopadhyay and Mitra argue that their approach is more flexible than prescriptive techniques (addressed later) claiming, *"Our use of an exemplar is akin to adapting a template. However, we believe that our approach to understanding through active exploration gives the re-user a fundamental understanding of the relevant dependencies, which is not achievable through predefined or prescriptive steps"*.

While it is true that their approach describes a logical and methodical approach to framework modification, its assumption that modifications will neatly fit into the existing modularity of the framework seems somewhat limiting. It seems likely that framework re-users (especially in white box frameworks) will require greater flexibility to go outside the existing constraints of the system when making their modifications. It is also uncertain how their approach will scale to accommodate larger frameworks (the example they use is a small framework containing only six key classes). Understanding the exemplar in a large framework would not be a straightforward task and many modifications would be required to transform it into the required application.

Michail and Notkin present a tool which compares example applications to determine the similarities in how they have exploited the underlying software framework (Michail and Notkin 1998). Their tool, CodeWeb, calculates 'reuse boundaries' between example applications and the framework code. A reuse boundary (**Figure 7**) is a class diagram which is comprised of classes from the framework which are directly used by the example application (either via inheritance or via composition). Michail and Notkin claim that this approach focuses a re-user's attention onto the *"important aspects of a library that are applicable to most applications independent of their purpose"*. The tool can present boundaries formed from the intersection of multiple example applications and can show details of how each example application has made use of the framework classes. In the figure the larger view shows the reuse boundary for two example applications, while the two small panes underneath show the original applications. This approach presents a useful way in which to display example applications, as it enables users to easily compare examples to see how different types of framework modification have been made. This technique is heavily dependant on the diversity of examples that can be compared. Examples with a lot in common presumably teach the re-user less about how the framework can be used. Finding a good set of examples which exhibit such diversity would appear to be difficult.

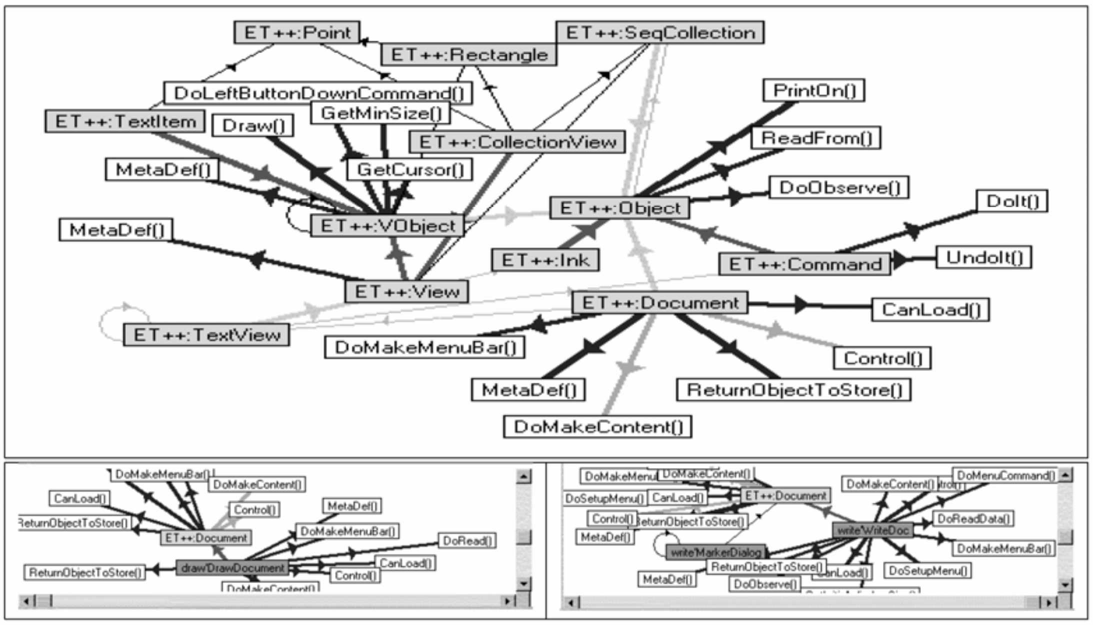


Figure 7: Reuse boundary from ET++ (Michail and Notkin 1998)

Frameworks often illustrate their capabilities through a number of example applications that are provided with the framework. Some researchers have argued that careful selection of which examples to include could increase their utility. Dénommée argues that collections of examples need to be constructed carefully to maximise their pedagogical benefits (Dénommée 1998). He believes that examples should focus on the introduction of one framework concept at a time and should be graded so that easy, generally applicable concepts are introduced before more complicated modifications. He observes that current examples are often used to illustrate too many features of an application framework at one time. This reduces their effectiveness because the examples can become too complicated to understand. Dénommée claims that by grading the examples the re-user will be better able to identify the new functionality and then imitate it. Dénommée's work echoes the argument of Sparks, Benner and Faris who also claim that frameworks should be documented with a series of examples to illustrate the frameworks capabilities (Sparks, Benner and Faris 1996). Dénommée has not produced any examples of such documentation for discussion and given the apparent effort required to create a set of examples, there are concerns about the feasibility of this approach. Graded examples may also have difficultly scaling to address the wide range of functionality that is possible with many frameworks.

Shull, Lanubile and Basili present an evaluation of the role that examples play in framework reuse (Shull, Lanubile and Basili 2000). Their study compared two approaches to framework documentation: an example based and a hierarchical approach. The example based documentation consisted of a set of example applications and a suggested reading order which emphasised examples that were considered to be of particular importance, while the hierarchical technique focused on describing the role of the abstract classes in the framework, guiding developers through increasingly concrete levels of the framework hierarchies until all details were described. The comparison between the two techniques took place in an academic environment using groups of students as participants. Their conclusions are presented as a set of hypothesis for further investigation rather than as concrete findings.

Shull et al. present evidence to suggest that examples are an effective framework learning strategy. They claim that this is especially so for people beginning to learn a framework, citing the fact that all the participants in the study eventually moved over to an example based approach including those who had originally been taught the hierarchical technique. Their findings suggest that examples are not a perfect form of documentation. They report that subjects occasionally had problems finding functionality of interest within the examples (especially when the functionality was a small part of the example) and that the subjects were confused by inconsistencies between approaches taken by different example applications. They also conclude that example based documentation may prevent developers from going beyond the presented functionality suggesting that examples do not provide enough details of the framework’s construction.

Shull et al. also report a temptation, from the subjects, to take more elaborate functionality from an example than was required, compromising the architectural integrity of their design. Shull et al. draw comparisons between this finding and the work of Schneider and Repenning discussed earlier (Schneider and Repenning 1995), suggesting that examples can have the potential to harm framework reuse by encouraging trivial modifications to be done before the major functionality of the application is addressed.

Shull’s report appears to present a duality about the role of examples as documentation. On one hand they appear accessible and easy to use, they lead developers to functionality and they can be used as a predefined starting point to be customised into the desired application. On the other hand, their benefits must be balanced against the evidence that they do not teach details of the frameworks design, that they can overcomplicate an application and that they can lead re-users astray.

Examples appear to be widely accepted in the framework literature as a documentation technique. They help to resolve the abstract information within a framework by providing a concrete illustration of a framework’s capabilities but in doing so run the risk of only presenting a narrow subset of possible modifications to the user. Examples are not a particularly expensive documentation to create and often frameworks come supplied with a number of examples to illustrate their capabilities. This review has suggested that to get the most out of example applications they ought to be carefully constructed to introduce one concept at a time and be ordered in terms of their difficulty. This may increase the cost of examples as a larger number are required and more care must be taken in their design. Shull et al.’s evaluation suggests that examples do have a role to play in reuse but also identifies further concerns about an example based approach. They report that examples do not teach users about the wider architecture of the system and that they can encourage developers to create more elaborate applications than they actually require.

2.7. Prescriptive documentation

Prescriptive documentation differs from other approaches in that it focuses on the activities that the re-user should perform to customise a framework rather than details about its structure. This allows prescriptive documentation to capture the actions of more experienced framework users and convey them to novice users, hopefully helping them to create effective modifications to a framework.

Cookbooks are a prescriptive technique and are one of the earliest forms of framework documentation. Their origins can be traced back to the Model View Controller (MVC) description provided by Krasner and Pope (Krasner and Pope 1988). This illustrated how the MVC framework ought to be used when creating Smalltalk applications. Cookbooks are effectively compilations of examples that illustrate common reuse tasks within the framework. They differ from examples by not describing complete applications, instead featuring small fragments of code, and by providing a textual explanation of the purpose of the code and when it should be used. When a cookbook has support for a modification it can be a very effective documentation technique but when a modification is not described developers may be left with little support to guide them.

Pattern languages are a documentation technique for solving design problems that was originally proposed by the architect Christopher Alexander in the context of civil architecture and design (Alexander et al. 1977). The motivation for this work stemmed from the opinion that modern alternatives to architectural design were creating cold, uncaring and ineffective structures that were generally unappreciated or even harmful to society. Alexander notes that successful designs have a form of objective truth about them, in general people agree strongly when a structure works well regardless of its style or cosmetic properties. Furthermore Alexander and his co-workers noted that it was possible to derive an abstraction of the key relationships that caused a design to work well, these relationships often being derived from empirical observation. Alexander’s pattern language consists of approximately 250 patterns divided into three layers – Towns, Buildings and Construction – partially ordered by the scale of problem they address.

As an example the patterns in the Building section describe patterns such as the appropriate number of stories in a building, appropriate orientation for a building and creating useful rooms for different purposes. The language does not provide a definitive, static instruction set for construction rather it is general and flexible to allow tailoring for specific needs in particular domains. Another characteristic of a well-constructed Alexandrian pattern language is that of ‘generativity’. *“Each pattern is a rule which describes what you have to do to generate the entity which it defines”* (Alexander 1979). Alexander’s work has been a major influence on the ‘pattern’ community and has ultimately led to the development of pattern languages for a large range of software engineering design processes, including user-interface design (Borchers 2001), relational database development (Brown and Whitenack 1996), as well as the documentation of object-oriented frameworks.

In the case of frameworks, a pattern language aims to interweave a system of ‘patterns’ into an explicit route-map through the framework architecture. The properties of generality and flexibility are also highly relevant, since object-oriented frameworks are likely to be tailored to applications that are not anticipated by the original framework designer. Finally, the generative characteristic of pattern languages is attractive in that it implies the production of a solution.

Johnson was the first person to identify the potential of pattern languages as effective documentation for frameworks (Johnson 1992). Johnson identified three fundamental problems that, in his opinion, limited the potential of frameworks for large-scale reuse: identifying the purpose of the framework, understanding how to use its parts and understanding its design. He claims that, although pattern languages effectively address how to use the framework, they can also be extended to address all three of these issues. Johnson has produced an example pattern language for HotDraw (a Smalltalk framework for creating semantic drawing editors). This language is considerably smaller than Alexander's consisting of ten heavily inter-related patterns (an example of which can be seen in **Figure 8**). Johnson interprets Alexander's work as describing common cases of construction. Similarly Johnson's patterns describe stereotypical modifications rather than more esoteric adaptations of the HotDraw framework. Johnson's language is ordered in the sequence that decisions are to be made when developing using the framework and the patterns themselves contain only the essential information required to instantiate that part of the framework.

Pattern 4: Complex Figures

Some figures have a visual presentation with internal structure. For example, they may have attributes that are displayed by other figures. It should be possible to compose them from simpler figures.

Complicated figures like PERTEvent can be thought of as being composed of simpler figures. For example, a PERTEvent is a RectangleFigure with several TextFigures for the title, the duration, and the ending date. Complex figures like PERTEvent are subclasses of CompositeFigure.

A CompositeFigure is a figure with other figures as components, and it displays itself by displaying its components. It has a bounding box that is independent of the bounding box of its components, and it will display its components only if they are inside of its bounding box. The selection tool and text tool will operation on its components when the left shift key is pressed. Custom tools can operate directly on the components, if you want.

In general, a figure should be a subclass of CompositeFigure whenever one of its attributes will be edited directly by a tool. The most common example is that an attribute is a string, and must be edited with the text tool. Instead of storing the text attribute in an instance variable, store it in a TextFigure. Do this by first ensuring that the attribute is read and written only by a pair of accessing methods. Instead of a string-valued instance variable, make a TextFigure-valued instance variable, and make the string's accessing methods read and write it from the TextFigure. This can be generalized for any kind of attribute that is represented by another figure. The attribute should be stored in the component figure, changes to the attribute result in changes to the figure, and changes to the figure result in changes to the attribute. If changes to one component might effect others then constraints should be used. (See **Constraints (5)**). The **initialize** method of the complex figure must create the figure representing the attribute and add it to the complex figure. It may also need to create constraints. PERTEvent is a good example.

Complex figures should be a subclass of CompositeFigure, and figures that display one of its aspects should be a component of it

To enforce constraints between the components of a complex figure, see **Constraints (5)**.

Figure 8: An illustration of Johnson's patterns (Johnson 1992)

Of special importance in Johnson's framework is the introductory pattern. This represents a clear starting point in the pattern language providing a high-level overview of the framework, its vocabulary and its capabilities. This first pattern acts as a starting point for searches via links to the key sub-problems that must be solved when using the framework.

There are a number of other key comparisons between Johnson’s and Alexander’s pattern languages: Johnson's patterns describe specific details about a framework whilst Alexander's patterns focus on generic qualities true of a range of buildings. Johnson also has more emphasis on guidance, activities that the framework re-user must do, whereas Alexander is concerned with describing the key relationships between entities in his domain that resolve a problem. Alexander’s patterns are derived from observation and empirical evidence – they are solutions to problems that have been shown to work. Johnson’s patterns lack this empirical evidence but are based on significant experience and deep understanding. Alexander’s descriptions of the problems that the pattern seeks to address are described in greater depth than in Johnson's patterns. Johnson’s language is understandably smaller, has fewer links and also has dead ends (patterns with no links). His patterns appear to be coarser than Alexander's, where Alexander has a general pattern that is subdivided into lots of specific sub patterns; Johnson tends to provide a single more general pattern that handles the variation internally. Johnson has an explicit starting point and path through the language, whereas in Alexander’s patterns the user jumps around as needed. Both use examples to illustrate points but perhaps for different reasons, Johnson to illustrate potential solutions, Alexander to illustrate potential problems.

Lajoie and Keller extend Johnson’s work based on the observation that developers require more detailed knowledge of the framework design (Lajoie and Keller 1994). They have proposed a multi-document refinement that integrates a micro architecture documentation (described earlier) with a pattern language. Furthermore they propose linkage from the framework source code back to the pattern language to support understanding both in a top-down and bottom up manner. **Figure 9** contains an illustration of their approach. Lajoie and Keller have not yet provided a complete example of their proposed pattern language and there is a lack of guidance on how such a language should be constructed.

Motif:DRAW New Graphical Shape

Situation

There are a variety of graphical shapes that can be incorporated in a graphic editor. Here we describe those shapes and how one goes about integrating them in an application.

Situation Discussion

Each graphical shape is a subclass of Shape. There are already subclasses of shape for the simple objects (LineShape, BoxShape, OvalShape, PolyShape, ImageShape, DynShape, PictureShape, TextShape), and these may in turn be subclassed to create more complicated shapes (RegionShape, BezierShape, ArcShape, RcBoxShape, Connection, DynShape2). The minimum required to define a subclass of Shape includes the definition of the methods Draw, Outline and GetImage.

Each application developed from DRAW will have a class *draw* whose constructor is responsible for setting up a palette of shapes. When adding a new shape, you are required to update the palette correspondingly. To do so, an image item (bitmap), to be displayed as the selection button, must be created and included in the new class's implementation, for example,

```
Static short BoxImage()={
#include ‘‘images/BoxShape.im’’
};
```

The new shape class must then be added to the list of shapes in the palette, i.e. added to a collection called *prototypes*. This is performed in the constructor of the class *draw*.

```
prototypes->Add(new BoxShape);
```

Most shapes will additionally define stretching functionality, input/output capabilities, selection handles, etc.

References

- Graphical shapes may depend on each other – see **design pattern** *Observer*.
- Stretching capabilities – see **motif**: *Graphical Shapes Stretching Capabilities*.
- Graphical shapes may be connected – see **design pattern**: *Connection* and/or **contract**: *Connection*.
- Complicated graphical shapes – see **motif**: *Subclassing Simple Shapes*

END – Motif:DRAW Graphical Shape

Figure 9: A Lajoie style pattern (Lajoie 1993)

Meusel, Czarnecki and Köpf also extend Johnson's work by proposing a more algorithmic format for the patterns in a pattern language (Meusel, Czarnecki and Köpf 1997). They propose a three-layered language that aims to address Johnson's three framework reuse categories (purpose, how to use and detailed design). The first layer defines the purpose of the framework and provides special patterns called catalogue patterns. These are intended to describe a framework's capabilities in order that a developer can determine the applicability of the framework. The second layer describes how to use the parts of the framework and consists of application patterns. These are similar to Johnson's but have a strict format, as opposed to Johnson's free-form narrative, with sections for problem description, context and solution. Again the solution is more prescriptive than Johnson's and is expressed as a numbered list of actions and as a flowchart. The individual steps inform the developer of the actions that they must perform (e.g. subclass this class, override this method). In some cases a step will reference other patterns that provide a required service or reference an example that completes the generic steps in the pattern with specific information. The third layer addresses design documentation. This is less specific and may include descriptions of design patterns or architectural overviews. Supplementary to this is a glossary that describes framework vocabulary and a set of tutorials based on examples that show the range of framework capabilities. The more prescriptive nature of the patterns makes them easier for re-users to implement but it also reduces the scope of problem that each pattern can address. This makes the language behave more like a cookbook, requiring more patterns to be produced to provide enough coverage of possible modifications.

Name: Select Existing Tools
Requirement:
  The application needs a particular tool or set of tools which is already provided as a part of HotDraw.
Type: Enable a Feature, Multi-Option
Area: Tools
Uses: Incorporate Tools
Participants:
  ExistingTools set of (toolclass, toolName,description),
  ChosenTools sequence of (toolClass, toolName)
Changes:

```
repeat as needed
    choose t from ExistingTools
    ChosenTools add (t.toolClass, t.toolName)
Incorporate Tools[ChosenTools]
```

Constraints:
  SelectionTool is required for movement of figures using the mouse
  Set(BringToFrontTool, SendToBackTool)

Figure 10: Illustration of a hook

Finally, Froehlich, Hoover, Liu and Sorenson describe a documentation called hooks that closely resembles a pattern language structure (Froehlich et al. 1997). Hooks focus on how to use the framework and omit any details about the framework design. Individual hooks describe the sequence of steps that must be performed to customise a part of the framework (**Figure 10**). Hooks are comprised of a set number of fields which describe its purpose or what actions are required to implement the hook in an application. The most important parts of a hook are the participants, changes and constraints fields. The participants field identifies which framework classes are involved in the hook. Changes describe a list of actions that the reader must perform to implement the hook, and constraints outline important invariants of the framework that must be maintained by all modifications. Hooks differ from the patterns proposed by other researchers in terms of scale. Typically there are eight to ten hooks for each of Johnson's patterns, each describing separate features that a developer may customise. This level of description ties hooks closely to the code they describe. Hooks appear related to the idea of framework hotspots (Pree 1994) as they describe predefined areas that are intended to change whilst the rest of the framework remains fixed.

Some researchers have also been inspired to encapsulate a pattern language within a tool. Both the FRED (Hakala et al. 1998) and SmartDoc (Ortigosa, Campo and Salomon 1998) tools present guidance on what modifications are possible based on the current state of the application and partially automate some modifications. Although such tools are appealing, there are concerns that they distance the re-user from the process of selecting a modification and result in a poor understanding of the framework code. This is arguably not important if the tools are adequate for all framework modifications but this seems a rather demanding requirement. It may be the case that such tools, rather than teaching new developers how to use a framework, might be more effective in supporting experienced re-users by automating the generation of tedious sections of framework code.

Prescriptive approaches to framework documentation can deliver important information to re-users of a framework. They provide support by telling the user what to do in a particular circumstance. There is little discussion in the literature to suggest how the range of circumstances which require support can be identified. From this review it can be seen that there appears to be general agreement on the advantages of pattern language-based documentation, but there is also great variety in their application. Further there is a disturbing lack of evaluation of the actual strengths and weaknesses of different pattern formats, and no guidance on how to create the pattern languages described. It also appears to be generally accepted by the framework community, for example Lajoie (Lajoie 1993) and Fontoura et al. (Fontoura, Pree and Rumpe 2000), that prescriptive documentation must be augmented with additional support to provide insight into the structural and behavioural aspects of the framework, which are often overlooked by such documentation.

2.8. Conclusions

This review has identified and compared many different approaches to framework documentation. It has categorised the documentation into six major themes: source code, micro architectures, macro architectures, examples, hotspots and prescriptive techniques, which describe the major contribution of each document type. A surprising variety and number of documentation approaches exist but in each case they appear to have been suggested with little evidence to justify their selection. This is frustrating because it prevents researchers from drawing conclusions about how complete the coverage of relevant types of documentation has been and it prevents a systematic narrowing of scope onto a few likely candidate techniques. Documentation techniques also lack thorough evaluation of their utility. This review can identify only one significant evaluation of framework documentation (Shull, Lanubile and Basili 2000). Many more are needed. Research cannot hope to make reliable suggestions about which techniques to use, nor can it comment on details of format and content to make future documentation more effective without an understanding of the capabilities and usefulness of existing documentation. This thesis attempts to address these limitations, firstly by identifying the problems that occur during framework reuse and secondly by evaluating documentation techniques against those problems to identify effective documentation support for frameworks.

3 Identifying framework reuse problems

3.1. Introduction

There is currently little understanding of the problems that developers experience during framework reuse due to a lack of evaluation. Without this information documentation cannot be properly designed to address the needs of re-users. Instead current techniques are based on guesswork and opinion which limits their ability to provide effective support. This lack of insight into framework reuse problems makes it difficult to advocate a technique or combination of techniques with which to describe object-oriented frameworks. The lack of knowledge also frustrates research as there is no indication of which problems are being over addressed or which are being ignored by existing documentation.

This chapter investigates framework reuse in order to characterise the problems that framework documentation must address. It does this through observation of several framework reuse efforts which record the problems that developers face when modifying a framework. It generalises these problems to identify the types of information that documentation must support. This study will also collect insight into the assistance provided by existing forms of documentation by soliciting developer opinion about their utility. Identifying framework problems in this manner and understanding the capabilities of current documentation allows existing support for reuse to be compared and improved. It also suggests how combinations of documentation can be used to address reuse problems and may assist the development of new forms of documentation that are better informed by reuse problems.

3.2. Experimental design

This study has collected evidence of reuse problems from three different scenarios: an individual developer, a class of software architecture students and a group of project students. Each group performed a framework modification task and a variety of data collection techniques were used to capture the reuse problems they experienced. The tasks all used a single framework, JHotDraw, as a basis for the modifications.

3.2.1 JHotDraw Framework

The JHotDraw application framework (Gamma and Eggenschwiler 1998) is implemented in Java and is a good example of a mature, well designed and well documented object-oriented framework. It is a remake of an earlier Smalltalk implementation (Beck and Cunningham 2005). A variety of other implementations of HotDraw exist and they are used both industrially (RoleModel Software 1996) and as a test bed for documentation research (Johnson 1992), (Richner and Ducasse 1999) and (Meusel, Czarnecki and Köpf 1997).

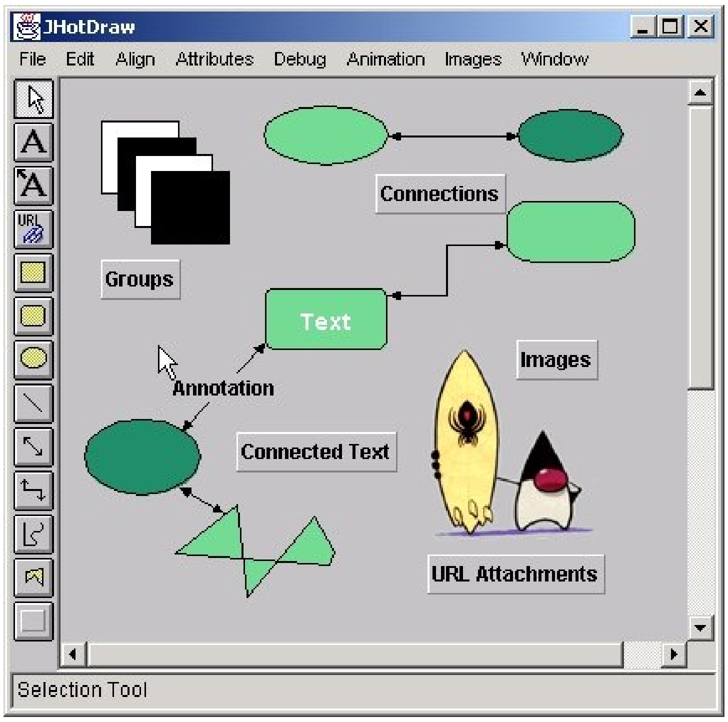


Figure 11: JavaDraw. A JHotDraw application (Gamma and Eggenschwiler 1998)

JHotDraw has been designed for the creation of semantic drawing editors. It provides support for a range of applications from simple paint programs (**Figure 11**) to more complex applications that have rules about how their elements can be used and altered (for example a UML diagramming tool or a Petri Net tool). The framework provides support for the creation of geometric and user defined shapes, editing those shapes, creating behavioural constraints in the editor and animation. JHotDraw is comprised of approximately 120 classes and features object-oriented concepts such as abstract classes, interfaces and design patterns. It has proved to be a rich environment for application development with sufficient depth and complexity for developers to experience a range of reuse problems during its modification. JHotDraw was also selected for this study because of its documentation support. It comes with implementations of many different documentation techniques, including:

- The framework source code.
- JavaDoc listings.
- A set of design pattlets (brief design pattern descriptions).
- A class diagram, showing the high level design of JHotDraw.
- A set of four example applications.

This documentation was used to support each of the reuse tasks (described below) and was augmented in the two student studies with additional material created by the author. The additional material includes:

- A pattern language.
- A set of practical exercises, which teach key parts of the framework.
- The ad-hoc mentoring and support that was offered by the teachers of the software architecture class.

The pattern language was modified from an existing pattern language (Johnson 1992) that addressed a Smalltalk implementation of HotDraw. The practical exercises were created from the application developed during the individual developer study. The application was divided into separate tasks each introducing one new aspect of the framework at a time.

3.3. The pattern language

A pattern language describes how to solve design problems within a domain. It presents a decomposition of the design of the framework into a collection of design problems which can be understood independently from the whole. Each sub-problem is described by a “pattern” that identifies the requirements that must be considered in solving the sub-problem and proposes a solution that resolves them. A language is formed by creating relationships between patterns (perhaps indicating problems which occur together, or identifying contrasting solutions). A fundamental property of patterns in a pattern language is that they do not attempt to provide complete solutions; each outlines a generic process or solution that can be tailored to the unique needs of individual situations. This provides scope for the designer to adapt the design to meet the needs of a particular environment.

Several different implementations of patterns have been suggested to tackle the problems of framework reuse (Chapter 2). These languages combine knowledge of the anticipated modifications to the framework with guidance about how such tasks should be carried out but vary in the format and scale of information provided. The lack of agreement about which structures to include in a pattern language is frustrating and the lack of evaluation of existing approaches prevents a direct comparison between techniques. There is also a lack of guidance for the creation of pattern languages, which makes it hard to reproduce the structures described in the literature for other frameworks.

Johnson describes a pattern language for the older Smalltalk version of HotDraw (Johnson 1992). This pattern language was the first to be applied to software frameworks and closely mimics the work of Alexander (Alexander 1977). Initial unfamiliarity with the structure of JHotDraw prompted the decision to create a literal translation of Johnson's existing patterns for JHotDraw. This was only partially achievable; some patterns were removed as they did not seem relevant and all patterns required considerable effort to translate. Also, there was no way of identifying any important new concepts in JHotDraw due to a lack of experience with the framework. The language generated was relatively small with eight patterns compared to Johnson's original ten (**Figure 13**). The complete language is available in Appendix B.

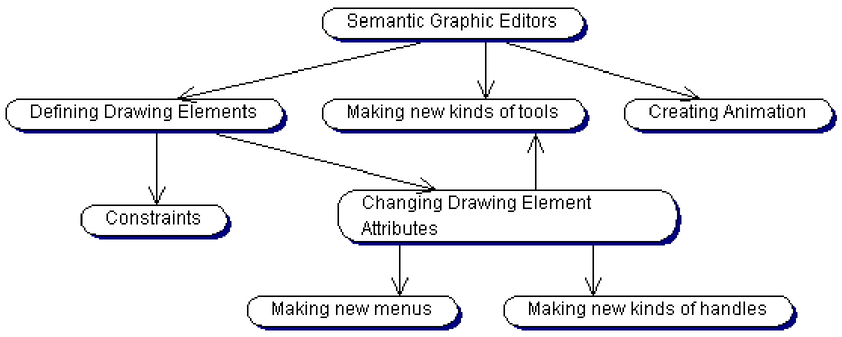

Semantic Graphic Editors
Defining Drawing Elements
Making new kinds of tools
Creating Animation
Constraints
Changing Drawing Element Attributes
Making new menus
Making new kinds of handles

Figure 12: Overview of pattern language

Defining Drawing Elements

There are an infinite variety of primitive figures that can be included in a drawing. Thus, there needs to be a way to make new figures for each application. Each kind of drawing element is a subclass of Figure. In HotDraw there are four strategies to figure creation. The relative strengths and weaknesses of each are explained below.

1. **Use existing classes:** HotDraw is supplied with default implementations for many common figures, such as EllipseFigure, RectangleFigure, and LineFigure. These may be reused ‘as is’ in new applications, this saves developers the time and effort required to create such elements themselves but reduces the knowledge the developer has about how that figure is implemented (important in efficiency and robustness arguments).
2. **Subclass existing classes:** Often the default implementations of Figure are ‘almost but not quite’ what is required by an application. In such circumstances it makes sense to customise the existing figure to fit the applications needs. This saves time compared with starting from scratch and provides insight into the organisation of that figure but sub classing will increase the number of classes in the system and carries the responsibility of ensuring that the original intent of that inheritance hierarchy is maintained.
3. **Composition:** An alternative to sub classing, composition can be used in situations where the new functionality required can be created by arranging several pre existing figures together. e.g. A text label could be defined as a rectangle with a text object inside. To facilitate this kind of creation HotDraw provides a subclass of Figure called CompositeFigure that essentially acts as a collection of figures. Composition frees the developer from class details, can reuse existing tools and allows dynamic configuration (new arrangements of figures can be constructed while the program runs). Against it, composition doesn’t provide as much freedom as sub classing (it can only compose what is available) and it can be difficult to debug due to its dynamic nature.
4. **Custom figure:** The ultimate amount of creative freedom comes from sub classing Figure or AbstractFigure. The choice of which class to choose depends on which is closest to the needs of the application. If the required Figure is very unusual then it may be required to subclass from Figure, but this should be avoided were possible due to the large number of abstract methods that require a definition. Instead it’s much more usual to subclass AbstractFigure which only requires four methods to be defined, basicMoveBy, basicDisplayBox, displayBox, and handles. This approach provides almost total control over how a figure is implemented however it comes at the cost of requiring an understanding of how the interface of Figure is used by the HotDraw framework.

Each drawing element in a HotDraw application is a subclass of Figure. Figures can be used directly or customised by sub classing or composition. In extreme cases a custom figure may be created by sub classing from the Figure class directly.

- **To let the user change the attributes of a figure, see** Changing drawing element attributes.
- **To enforce constraints between different figures, see** Constraints.

Figure 13: A pattern from the JHotDraw pattern language

3.3.1 The three studies

The individual developer (the author) created an Orrery application with the framework and recorded his experiences in a logbook. An Orrery is a mechanical model of the solar system, which illustrates a group of planets and their orbits. This was realised in JHotDraw as a two dimensional representation, with planets modelled as circles and gravity relationships represented as lines connecting orbiting planets together. The application allowed users to create and position planets and then connect them together using gravity relationships to form a model solar system. This model could then be animated showing the planets moving through their respective orbits. Each planet also had a mass, which, when the planet was animated, altered the speed of its orbit. The application was not intended to be an accurate model, for example the gravity relationships were not governed by the mass of the planet nor did adjacent planets affect the orbits of their neighbours. Rather it was designed to be fun to use and to demonstrate many of the features supported by the framework. The study was the developer's first exposure to the framework and took approximately 80 hours to complete.

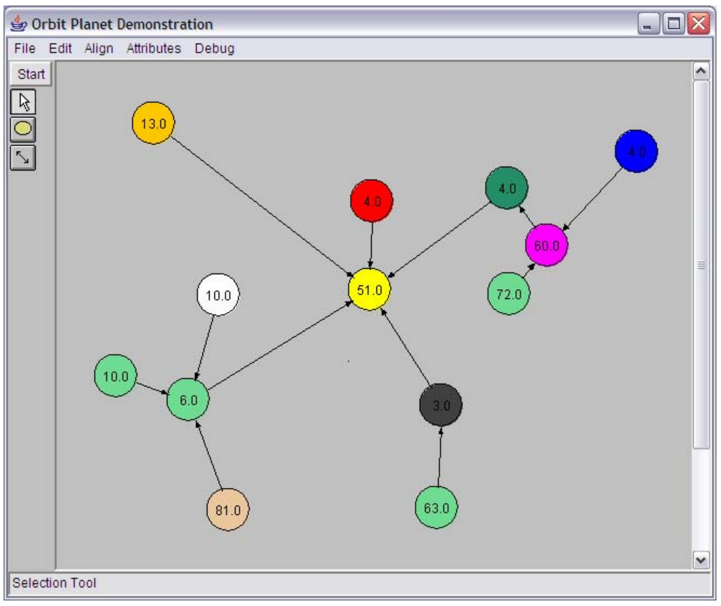


Figure 14: The Orrery application

The class-based study used students from the Honours level Software Architecture class at the University of Strathclyde in Glasgow. The class teaches design patterns, software architecture and object-oriented frameworks. All seventy-seven members of the class participated in the study. The students recreated the Orrery application from the individual study through a series of practical exercises. The original application was divided into five tasks: to create a default editing application, to create a representation for a planet, to create a tool to add planets in the editor, to create a representation for gravity constraints and a tool to add them between planets and finally to animate orbiting planets (Appendix A). After this the students were challenged to produce a suitable modification to the Orrery as part of their class assessment. Students were responsible for the design and implementation of their own modification deciding what functionality to implement and what constituted an acceptable implementation. This proved to be an effective approach with the students providing a wide range of imaginative and effective alterations. The ideas they suggested included dynamic lighting caused by a sun, creating black holes and adding rockets that flew between planets. Students reported their problems in a class newsgroup and also in an assessed coursework report. The students were given a 2 week period to create their modification to the Orrery (including the practical exercises this gave them a 7 week exposure to the framework) and data was collected both during development (via a newsgroup) and at the end of the assessment (from the students coursework reports).

The project students consisted of four final year Honours students who chose to perform a framework modification as part of their final year project. The four students each chose a project from a list of suggestions (Appendix A). The suggestions described the basic idea of each of the applications but the students had the freedom to decide what functionality was appropriate to their application and how that functionality should be implemented. The students' experiences were collected in voluntary interviews that occurred at the end of their projects (all four students agreed to take part). Each project had a duration of approximately 6 months.

The first project student created a golf hole designer. This application provided functionality to create a number of different types of surface (fairway, green, rough, etc) and position them to create a golf hole. The user could then add detail to the hole by placing streams, paths, buildings and trees onto the design. The application placed constraints on what could be considered a valid golf hole. For example, each hole had to have only one green and one tee area, flags could not be positioned outside of the green and buildings were not allowed on the fairway. The editor also calculated the par for each hole that was created, measuring the distance from the tee to the flag.

The second student created a UML class diagram editor. The editor contained tools for creating classes and interfaces, adding methods to classes and creating relationships, such as inheritance or composition between classes. The application could generate a code skeleton from a diagram and it also contained a collection of design pattern templates that could be added to the diagram with a single click.

The remaining two students created track editor applications. One created a model railway editor the other a Scalextric™ racetrack editor. Both applications allowed the user to create a track from a set of smaller segments. These were added individually to the diagram and could then be connected together to form a track. The applications also allowed users to create vehicles, which could be animated to make them travel around the track and prevent users from joining incompatible pieces of track together. The railway application featured a constraint that prevented trains colliding, while cars in the Scalextric™ application would fall off the track if they took a corner at high speed. **Figure 15** illustrates some of the applications produced by the project students.

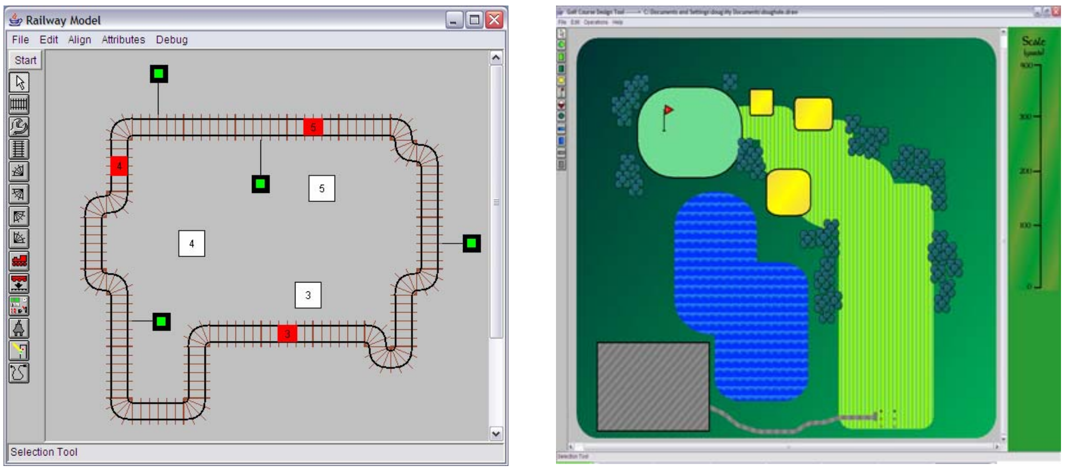


Figure 15: Some of the applications created by project students

The choice of participants and scenarios across all three studies was largely opportunistic; they were the people that were available and able to participate in the study. They all had similar levels of software engineering experience, that of a final year or recent graduate, and were representative of the kind of people who might be asked to modify frameworks in industry. In each case the participants were given a lot of control over the design and implementation of their applications. They decided what functionality to include, how it should operate and when it could be considered complete. This helped motivate the participants and further promoted the diversity of content and hence problems that were experienced during development.

3.4. Data collection

Using a variety of data capture techniques helped ensure that an accurate view of reuse problems was presented from the studies. This section describes the approaches used and describes the amount and nature of data collected by each technique.

3.4.1 Individual developer study

Observations about the task were recorded in a notebook during development. The developer would pause every few minutes while working and write down what he had been thinking and doing. This record of his thoughts and actions showed how problems had occurred, what solutions had been proposed, which solutions had been attempted and whether or not they had been successful. It also recorded the developer's knowledge and understanding of the framework and how this had developed during the task. At the end of the task the developer's observations totalled twenty-eight A4 pages of text. (The logbook is available in Appendix A)

The logbook was not a perfect method of data capture. Occasionally the developer would forget to record important details about his actions. In particular the developer tended to focus on what he was doing rather than why, making it hard to determine his intentions in some areas of the logbook. Also the developer often forgot to attribute information to a particular form of documentation making it difficult to identify the support offered by documentation in this task.

```
How do you find centre of drawing? <Looking in drawing JavaDoc – Wrong!>
Pt AbstractFigure.center() [drawing inherits from this]
[Opens StandardDrawing SC]
[Opens EllipseFigure SC]

How do you add shape @ position on the drawing?               3:44
No obvious add functionality, may need to set figure locator
<Locator suggests that it works in conjunction with handles>
Just have to use brute force method (place it directly)
Figure.basicDisplayBox(Pt, Pt) <-(still hasn't worked!)       3:57
Turns out that AbstractFigure.center returns (0,0) for the Drawing. 4:11
And displaybox() also returns (0,0,0,0) – why? And how can I find the centre?
< -> DisplayBox is the union of all figures>
```

Figure 16: Extract from developer logbook

The excerpt from the developer's logbook (**Figure 16**) describes his attempts to find a way to position a figure in the centre of the application. The developer found a method called centre in the application's Drawing class that appeared to provide the midpoint of the Drawing. When called the centre method positioned the figure at the top left hand corner of the Drawing rather than in the middle. By debugging the code, the developer found that the centre method returned a value of (0, 0), which would correspond with the top left hand corner. On closer inspection the problem was found to lie in the definition of the Drawing's size. The developer had assumed that a drawing had a fixed width and height (and therefore a midpoint). The source code revealed that the drawing did not have a fixed size at all. Instead it calculated its size as the smallest bounding box that would surround its current contents (for an empty drawing this resulted in a zero sized boundary and hence a zero value for the centre). Realising that the centre method was not going to provide a solution the developer had to abandon that approach.

3.4.2 Software architecture students

The newsgroup was set up as a forum for the class lecturers to communicate details about the class and to monitor the problems student developers were having with the framework. It also served as a self-help service, where students could appeal to their classmates for help on particular topics. This kind of community atmosphere was encouraged in order to prevent students feeling overwhelmed with the complexity of the framework. Although not originally intended as a source for data in the study it rapidly became clear that the newsgroup would be a valuable source as many postings described succinctly the problems that were occurring. Often postings would also include information about solutions already attempted or the amount of time spent on a problem, which helped to provide a context to understand the problem discussed. One negative aspect of a newsgroup is that it may hide the scale of particular problems (i.e. if one person posts a problem and it gets answered then anyone else with the same problem, or who would have had that problem in the future, will now have a solution). Another concern is that encouraging a sense of community amongst students may increase the chances of them talking about problems together outside of the newsgroup. This probably did occur but was limited by the heterogeneous nature of the students' modifications.

```
Subject: Border on drag
Date: Mon, xx Nov 2000 xx:xx:xx +0000
From: xxxx xxxx
Organization: Department of Computer Science, University of Strathclyde
Newsgroups: strath.cs.ugrad.sw-architecture

How would you draw the border on selection and remove it on deselection.
I can sub class selection tool, and draw the border round an ellipse,
but how to combine the two?
```

Figure 17: Posting from software architecture newsgroup

In **Figure 17**, a posting from the Software Architecture newsgroup, a student describes a problem about how to make an alteration to the framework. In this case the developer wants to modify the selection behaviour of the application to make it draw a rectangle around figures that have been selected. Parts of the solution are apparent, the developer is thinking of changing the selection tool, and already knows how to make the border appear around a figure but the developer doesn’t know how to combine this knowledge together to create the desired modification (The postings from the newsgroup are available in Appendix A).

The coursework reports were also used as a form of data collection in this study. The class required students to describe the modification they had attempted, the problems they faced and what use they had made of the available documentation. The reports were a rich source of information as the students described many problems and experiences that they had encountered during the task. Finding the descriptions within each report was difficult because students scattered them throughout the text, even though each report had a designated section to describe reuse problems. This meant that the remainder of the report also had to be searched to check for any other problems that may have been included. (The coursework reports are not included in the Appendices for ethical reasons)

> *“Some of HotDraw's menus supplied by DrawApplication aren't really appropriate for the simulation, but it wasn't clear how to disable them. Overriding the methods to add entries is mentioned in the documentation but not how to remove entries. Overriding them with empty methods causes exceptions to be thrown in the frameworks classes and a crash, so the menus were left in place”.*

Figure 18: Extract from a student's coursework report

Figure 18 is an account from the coursework reports in which a student describes a problem caused by turning off part of the existing framework. In this example the student wished to remove some of the menus that come pre-supplied in JHotDraw and which were no longer relevant to the application. The location of the menus in JHotDraw was found and the modification made to remove them. Later, when the developer executed the application to check the modification, it unexpectedly crashed, the error apparently being caused by the removal of the menu code.

3.4.3 Project students

The four project students were invited to take part in an interview at the end of their application development. The interview questions were informed by experience gathered from the previous individual and software architecture studies and covered topics such as how the students had used the framework, what problems they had encountered and how they felt documentation had supported them during reuse (the question template is available in Appendix A). An interview is a flexible technique which allows the interviewer to follow up interesting responses with additional questions. This is a surprisingly difficult skill to master and the interviewer occasionally found himself asking leading questions whenever the interview strayed from the anticipated path. The interviews were tape-recorded and were later transcribed into a written account (Appendix A). This was a time consuming and frustrating process because the sound quality of the tapes was not always clear. Fortunately the relatively short amount of data to transcribe prevented the task from becoming overwhelming.

DK: Can you think of anything specific in your project? [Student asks for clarification] Having two options and having to choose one basically?

Student: The thing that comes to mind was the flag, creating the flag… that you had the choice of just like a rectangle figure then just redrawing what it looks like in the display box or going about it with a composite figure that consists of like a triangle figure and then just a line. I think the second way is how I would… you know… composite figure but that sort of opens itself as well just because it doesn't specify the layout so it was difficult just sort of laying out the figures so that it would draw it properly on the screen when you clicked the mouse. There were a few occasions that I thought I had done it and you would try it and place it somewhere then maybe drag it someplace else and it would just go up to top corner. I think I actually ended up looking at… was it Pert, the Pert application had an example of that and composite figure specified the layout for the figures inside that. I think I maybe used a good bit of that.

Figure 19: Extract from a project student interview

In **Figure 19** a project student describes the problem of having to choose between two different ways to create a modification. The student wants to add a flag to the golf hole design application but can see two possible approaches to model the flag. In one approach the student could create a new figure, perhaps by sub classing an existing figure in the framework, and alter its appearance so that it looks like a flag. An alternative would be to use the existing composite figure class that would allow a combination of a line figure and a triangle figure to be positioned to create the flag. The problem for the student is to decide which approach to use. One gets the feeling that he would rather use the composite figure (he talks about it more) but has had problems getting the composite to work properly in the past and is tempted by the easier, although less elegant solution of creating a new figure.

Together the three studies collected a large amount of data about framework reuse. 28 pages of text were recorded in the individual developer's logbook. Approximately 770 pages of text were collected from the software architecture coursework reports and the class newsgroup contained 216 postings, which covered approximately 83 pages of A4 paper. The project student interviews totalled a further 33 pages of text. This data had to be searched for problems. In total 209 problems were collected from the data. 59 of these problems were derived from the individual study, 35 from the project students and 115 problems from the architecture class.

3.5. Threats to validity

This section considers the factors which may impinge upon the results of this study. Internal threats are those which compromise the findings for this study, external threats compromise the ability of findings from this study to generalise to the wider framework population. In each case the threat to validity is described and the steps taken to limit this effect are explained.

3.5.1 Internal threats

- **Data capture relies on the developer to recognise problems:** Identifying and describing problems relies on the developer detecting that a problem has occurred and capturing it in suitable detail so that it can be understood and analysed. This places a great deal of responsibility on the developer to identify and describe problems correctly. There is a risk that the developer may overlook problems or describe them poorly. These factors must be weighed against the difficulty of detecting problems through other mechanisms which would be more invasive and might disrupt the developer during the task.
- **Time between problems occurring and being recorded:** There is a risk that the time between experiencing a problem and reporting it will lead to inaccuracies in the description of the problem. In the worst case this might lead to problems being forgotten altogether. For this reason a variety of approaches were used to capture data. These varied in terms of their intrusiveness and also in terms of the time delay in capturing problems. Some techniques like the coursework reports and the interviews were relatively passive approaches to data collection but they occurred after the development had been completed and memories had degraded. In contrast techniques like the newsgroup and the logbook required more involvement from the developer but resulted in problems being captured much closer to their occurrence in the task. This is not to say that the former techniques were less effective, indeed the contrast in recording problems as they happen and from the perspective of the complete system, undoubtedly helped to capture different insights about the framework problems and also about the use of documentation.
- **Problem identification relies on the experience of the analyst:** Recognising a problem from a developer account and extracting information germane to its comprehension and eventual classification is dependant upon the judgement of the analyst. There is a risk that the inexperience of the analyst in both qualitative analysis

and framework reuse could lead to problems being overlooked or incorrectly attributed during the analysis.

- **Individual problems were hard to identify:** Problems were described in natural language and were often recorded amongst other insights about the framework. It was quite difficult to identify and separate problems from the collection of data. In part because of the amount of data that had to be processed (more than 800 pages of A4 text) but also because of the difficulty in identifying where one problem stopped and another began. A liberal approach to problem identification was taken where anything remotely suspicious was included as a problem with the knowledge that any false positives could be removed later on.

3.5.2 External threats

- **Results are limited to one framework:** The different reuse tasks all occur within the same framework. This prevents the study from distinguishing which problems are caused generally by frameworks and which are caused specifically by JHotDraw. Studying multiple frameworks was impractical because the documentation and set of tasks would have had to be adapted to investigate a different framework and this would have been impossible in the given timescale.
- **JHotDraw may not be representative of industrial frameworks:** There is a risk that JHotDraw might be unrepresentative of the frameworks commonly found in industry. However, JHotDraw has been designed by experienced framework designers, having evolved from an earlier well respected framework (Johnson 1992), and it continues to be actively developed (Kaiser 2005). It is also well respected by the framework community as it is used frequently as a testbed to develop new forms of documentation. Its popularity in this community suggests that it is representative of best practice.
- **Participants may not represent industrial developers:** The participants were all drawn from an academic background. This might limit the generalisation of the results because they might differ in their approach and motivation from industrial framework re-users. It is difficult to gain access to real world developers and academic participants should provide a reasonable approximation to the real thing. This is especially so in this case as only final year students were solicited.

3.6. Analysis

The study collected information about a large number of framework reuse problems. This information is overwhelming in both its detail and volume, making it difficult to draw useful lessons from which to improve framework documentation. In order to be useful the data had to be reduced and concentrated to identify the general types of problems that occurred during reuse.

3.6.1 Cluster analysis

A data clustering process was used to reduce the problems into their generic types. Cluster analysis (Miles and Huberman 1994) is a technique in which related data is positioned on a grid. Items which have a lot in common are positioned close together and those which don't are placed far apart. As data is positioned on the grid clusters begin to emerge where similar items have been placed together. Such clusters can then be analysed by looking at the properties that the data have in common and using those properties to create a name and definition for that cluster. The process is iterative and often large clusters will form and then be split apart as finer subdivisions are spotted. Also, sometimes initially unrelated clusters can merge together as they are found to share a common property. Eventually after a period of time the clusters should begin to settle down to form a stable representation of the data. The clustering was performed manually and can be differentiated from automatic statistical approaches to clustering as it addressed patterns within the semantics of the data rather than merely identifying syntactic similarities.

The 209 problem descriptions collected from the reuse scenarios were used as data for the cluster analysis. The problems were collated into a numbered list before being cut out and placed into a cardboard box. The problems were then drawn randomly from the box (to prevent any ordering effect) and assigned to a position on a whiteboard. The clustering was performed as a group activity with the author and his two PhD supervisors taking part. Each problem that was pulled from the box would be read aloud and then its specific qualities debated until it could be assigned to a position on the board. In some cases the debate revealed that a problem was actually a composition of a number of smaller problems: when this occurred the problem would be spilt apart and assigned separately. There were also cases where supposed problems were found not to be an actual problem or where problems were stated so vaguely as to be unassignable. In these cases a separate cluster was created to hold the 'spoilt' problems.

Following this process clusters began to appear and as they were identified names were assigned and written beside them on the whiteboard. After processing about half of the problems fourteen categories had been identified (Partial understanding, Unexpected behaviour, Impact/Choice, Where to put?, How to?, Integrity, Where to go?, Why not here?, Search for, Delocalisation, Why here, Mismatch, Composite figure and the Spoilt column). Some of the clusters seemed quite stable and well supported but others were less well defined and had only a handful of entries. **Figure 20** shows a photograph of the white board in this state. At this point the clustering was paused and a review of the clusters was performed. The evidence for each cluster was reconsidered and some clusters were merged and reorganised to create a more substantial clustering. Initially the clusters were reduced down to a set of five (Functionality, Searching, Interactions, Mapping and Architecture), after a while this was further reduced down to four as the Searching cluster was recognised as a combination of problems from both the Interactions and Functionality clusters and was redistributed accordingly. The remaining four clusters appeared to be stable and well supported by the evidence but to test the validity of the reorganisation the remaining problems were analysed to check that they did not contradict the existing clusters. The remaining problems were found to be compatible, resulting in a final classification of four general types of reuse problem: Mapping, Interactions, Functionality and Architecture.

Figure 20: Initial clustering of reuse problems*

At the end of the cluster analysis from a total of 209 problems, the mapping category accounted for 38 problems, interaction 48 problems, functionality 60 problems and architecture 17 problems. A further 46 problem descriptions were not included in the cluster analysis because they either did not describe a problem with framework reuse or lacked sufficient detail to classify accurately.

* A more detailed version of this image with annotated problem numbers is available in Appendix A.

3.7. Problem categories

The four types of problem category identified by the cluster analysis represent a number of more specific problems. The properties shared by those problems in turn act as a definition for each category. The following paragraphs provide informal descriptions based on those properties for each of the four categories.

3.7.1 Mapping

Mapping (38 problems) identifies the problem of translating an abstract, conceptual solution into a concrete implementation which reuses the existing structures within the framework. Such problems are often expressed as *"What should I use to represent…?" or "How do I express…?"*. Mapping problems occur when a developer has problems expressing a solution using the abstractions available within the framework. In such cases there is often a mismatch in granularity, or perspective, between the functionality that is offered by a framework and the functionality that a developer expects to be there. When this mismatch occurs the developer finds it difficult to change their perspective and create a design that aligns appropriately with the existing structure of the framework. Mapping problems appear to occur early in the reuse cycle as logically one must decide what to do before attempting to do it. This increases their importance because bad decisions taken during mapping can cause further difficulties during the modification.

Mapping problems typically feature a goal, something that the developer is trying to achieve with the framework. Often problems occur because the solution a developer wants to take to achieve a goal does not match the expectations expressed in the framework's design. An example of this was demonstrated in the individual developer study when creating a tool. The developer wanted a tool to make rectangles. *"How do I create a rectangle? (Why is there no Rectangle tool?) – Subclass creation Tool?"* (Individual developer report). The developer initially assumed that a rectangle tool would exist within the framework but attempts to find such a tool failed and the developer began to consider creating a tool from scratch. Fortunately before doing so the developer referenced an example application and found that many different types of figures, including rectangles, were being created by a Creation Tool. Closer examination of the Creation Tool revealed that it implements the prototype design pattern which enables it to create any type of figure if given a suitable prototype to copy from. Having discovered this, the developer then had no problem implementing a tool to create rectangles. The mapping problem in this example occurred with the initial assumption to create a specific tool for a figure. This seems reasonable but it did not take into consideration the existing architecture of the framework and had the developer not been fortunate enough to stumble across a contradicting example then he may very well have implemented an unnecessary class to achieve the modification.

Another example of the mapping problem can be seen when the individual developer attempted to add a figure to the drawing. The developer wanted to add the figure at a particular location but could not find a suitable way to achieve this. *“How do you add shape at a position on the drawing? No obvious add function, may need to set figure locator”*. (Individual developer report). The Drawing had an add method but it only took a figure; no parameters were available to set its location. The developer had previously read about Locators in the framework, classes used to set the position of a figure, and began to consider a solution using them. However, they seemed overly complex and more suited to placing figures at relative positions rather than absolute values. Eventually the developer found the displayBox method in Figure which sets the figures size. Somewhat unintuitively this method also controls the figure’s position on the drawing as the parameters it takes are coordinates on the Drawing. This problem was caused because the expectations of the developer, that a drawing would allow a figure to be positioned, were incorrect.

A final example of a mapping problem comes from the software architecture coursework. A student wanted to remove Connectors from the end of a ConnectionFigure (Connectors are small white circles that are drawn where the figure and the connection meet). *“Initially, to hide connectors, they were set to the background colour. However, they were still visible if they crossed text or figures. It may be possible to use HotDraw to set them as being not visible but the same effect can be achieved by overriding the draw method with an empty method”* (Coursework reports). Initially the student had the idea of changing the colour of the connector to match the figure so that it could not be seen. This did not work because in some cases the connector is drawn outwith the coloured boundary of the shape. Eventually a solution was achieved by overriding the Connector class and replacing its draw method with an empty method. The student experienced these problems because they assumed that changing the colour of the connector would be equivalent to removing it.

Mapping problems occur when a developer is faced with a problem and has difficulty translating a problem domain concept to a framework solution concept. Failure to make the correct mapping at best results in wasted time and in some cases can result in a poor solution being adopted (as almost happened in the example about the rectangle tool).

3.7.2 Interactions

Understanding interactions (48 problems) focuses on problems concerning the communication between classes in the framework *“What happens if … ?”*, or *“Where should I put … ?”*. Such problems are significant because of hidden or subtle dependencies within the framework that may cause failures to occur elsewhere as the result of a wrongly positioned modification. Interactions problems occur whenever the interactions between classes cause confusion. They often manifest themselves through uncertainty about where to place modifications within the call graph of the framework (a problem that is exacerbated by the inversion of control characteristic of object oriented frameworks). Such problems are significant because framework classes are largely interdependent on one another. Illustrations of such problems might include: inserting code in the wrong area (for example making calls to an object before it has been initialised) or removing unwanted code which other parts of the framework might be dependant upon.

Interaction problems tend to occur because the communication between parts of the framework is poorly understood by a developer. This can result in modifications unintentionally altering the framework, or in extreme cases causing the framework to crash. An example of this occurred when the individual developer attempted to remove some unnecessary menus from JHotDraw. *“Override methods to turn off menus first tried setting createMenu()/createTools() to null behaviour -> crashed HD! – Seems like I want to override createXXXX methods. Looking at D/A s/c for example! - Might not be possible seems that you can only add extra not take away – not very flexible.”* (Individual developer report). The modification involved the removal of standard user interface code that created two menus on the user interface. When the modification was made the application crashed because other parts of the framework expected the menu options to exist. It is arguable that in this case the problem was actually caused by a weakness in the original design of JHotDraw as it seems unnecessary for the code to have a hardwired dependency upon menus in the user interface. The developer decided to abandon the removal operation in case there was any reason why those menus had to exist within the framework. Interestingly, the same problem was reported by a software architecture student during the study (**Figure 18**) with similar results.

Another interaction problem occurred when a developer attempted to add a border around a figure. *“I’m having a bit of difficulty with this practical … for drawing the box, I’m over-riding SelectionTool, and in particular mouseDown() so that when the figure is clicked the box is drawn. This bit works, however when trying to drag the figure, if I do something similar the rectangle flickers like mad.”* (Newsgroup). The code to draw the rectangle was placed in a part of the selection tool which was called every time the mouse was moved. This resulted in a large number of redraw commands being sent to the framework causing the figure to flicker whenever the mouse moved. A better understanding of the relationship between mouse events, the tool and redraw code could have prevented this problem from occurring.

A similar interaction problem was captured in the coursework reports. A student had attempted a modification to the Orrery application to create a rocket and allow the rocket to fly along a connection between planets. *"There is one more small problem with when I try to connect two planets using the route connection, I get a null pointer exception, although it still functions properly in that the connection does not occur. I did work out what the problem was but I couldn't think of a way to fix it. The problem occurs because I have set up the route connection figure and the rocket figure so that a rocket can only be connected to one planet at a time. When the connection is removed I set the rocket figure Boolean canConnect to true so that it may be used in another connection. However, the method this occurs in is called whenever two figures have tried to be connected, but are unsuccessful and so will result in the Boolean trying to be set but there is no reference to a rocket figure class."* (Coursework report). When the student created a connection between the planets the framework generated a null pointer exception. The problem was caused by the student's custom connection figure which had overridden one of the connection methods to test if a rocket could move along the connection. The method that contained this code was also called as part of the initialisation of a connection, which because it happened before a rocket was added to the connection, resulted in the null pointer exception. The student's inability to see how the method was used by the existing framework prevented them from detecting the problem until after the solution had been built, by which time it was too late.

Interaction problems are difficult to detect because it is hard to tell how the classes of the framework interact and whether a modification will have any ramifications beyond that which was intended. Often these problems are dynamic in nature and so they are not spotted until late in development when the modified application is executed. This can be quite depressing when a time consuming or difficult modification is found not to work due to an unexpected interaction problem.

3.7.3 Functionality

Understanding functionality (60 problems) describes problems understanding what specific parts of the framework actually do. Manifestations of this problem include *"How does … work?"*, *"Where does … happen?"*, or *"Where is … defined/created/called?"*. In part this is the familiar problem of understanding functionality described by source code but it is also affected by other factors such as a person's perspective, ability and domain knowledge. These affect a developer's ability to perceive and understand the functionality described within the framework. Functionality problems tend to result in developers overlooking existing functionality in the framework or attempting to use it inappropriately.

Functionality problems typically occur because of mistaken assumptions about the framework's behaviour. An example from the individual developer logbook (which has already been seen in **Figure 16**) concerned finding the centre of a drawing. *"Create circle in centre of screen – How do you find centre of drawing? Look in drawing > wrong! AbstractFigure.centre()."* (Individual developer report). Initially this seemed straight forward as the standard drawing class supports a centre method which purports to return the point at the centre of the drawing. When the developer used this method it returned (0,0) despite the fact that the drawing was clearly visible in the user interface. Eventually, after studying the code in detail, the developer realised that the results were correct. This was unintuitive because the developer could see the drawing in the user interface and so assumed it must have a fixed size. What was actually visible in the interface was the Drawing's view which was given an initial size to make it visible in the user interface. This problem was caused by the developer's assumption that a drawing had a fixed size and hence a centre point. Only by looking at the relevant parts of the implementation could he eventually determine that this was not the case.

Another problem which was quite common amongst students during the coursework was a difficulty in locating the methods to control a figure's colour. For example *"I have been looking everywhere to try to change the default colour of the ellipse from green to red. Do you know if this is possible and if so what class should I look at to do this?"* (Newsgroup). In part this problem was caused by the relevant information being contained in an abstract class near the top of the inheritance hierarchy (hiding it from view) but it was also caused by the naming of the relevant methods. To change the colour of a figure developers had to use a figure's setAttribute method and pass it the attribute name (FillColor or FrameColor) and a colour value. A more obvious approach would have been to have a set colour method on the figure. Developers often could not find the set attribute functionality and so resorted to elaborate solutions such as overriding the draw method to change the figure's colour.

Sometimes the inability to find relevant functionality resulted in solutions being chosen that were less than optimal. In one student's coursework report a requirement existed to check what figures exist at a particular point on the drawing. The student had correctly identified the find figure method of drawing as a potential solution but this only returns the topmost figure and sometimes the developer wanted a figure underneath this. The solution that was implemented was to go through each figure on the drawing and check whether it contained the point in question. This solution involved a linear search through the contents of the drawing despite the fact that most figures would be nowhere near the point in question. A more efficient solution was possible using the existing members of the drawing class (using a combination of find figure inside and find figure without) but the student did not recognise this and had to spend time and effort implementing the alternative solution.

Functionality problems are pervasive during framework reuse as there is much to comprehend. One strategy to deal with this volume of information is to make assumptions about what functionality is available. This can work well but, inevitably will sometimes be incorrect.

3.7.4 Architecture

Understanding the framework architecture (17 problems) is the problem of making modifications without giving appropriate consideration to the high-level architectural qualities of the framework. Such alterations might have no short-term negative effects but ultimately lead to the framework losing its flexibility. Architecture problems are the smallest cluster of problems identified in this study. They occur in situations where developers make, or plan to make, modifications to the framework without giving appropriate consideration to the architectural qualities they might be affecting. The discovery of relatively few architectural problems may be because such modifications do not immediately become problems. Rather they lie dormant until some future modification arises which cannot be made without rewriting the current solution extensively. Such long term effects would have been difficult to detect by the participants in this study given the short timescales involved.

Architectural problems occur when the developer does not respect the original design of the framework in a proposed modification. This can lead to both immediate problems and more long term problems as the flexibility of the framework is reduced. One such problem was the attempt to combine Swing user interface components with the existing JHotDraw user interface. *“Build GUI for tool using Java swing stuff [taken wrong approach here]”* (Individual developer report). This created a problem because JHotDraw is implemented upon the AWT framework, which although it underpins Swing is not directly compatible with it. The resulting conflict caused the user interface to be displayed incorrectly and the offending code had to be removed.

Another architectural problem occurred when the individual developer decided to edit the selection tool to prevent a figure from changing size. *"How do I prevent resize > think handles. Look in Ellipse Figure > redefine handles. Subclass ellipse? Seems a bit extreme. What about changing selection tool?"* (Individual developer report). The modification was successful but after it was complete the developer noticed a flaw. The changes to the tool prevented the figure in question from changing size but it also affected all other figures on the Drawing. The global affect of the modification had not been properly considered beforehand and it had to be removed in favour of a more local alteration to the figure.

A common architectural mistake made by the students during the coursework exercise was to supply figures with a reference to the drawing or the drawing view. *"Figures having access to Drawing/DrawingView – Figures do not by default have any access to either the Drawing or the DrawingView in which they are contained. This prevents them from accessing information such as the size of the drawing. However, it is possible to overcome this problem by passing the view into the constructor of a figure, which can then store and access this as required."* (Coursework reports). This violated the existing architecture of the framework which used the observer pattern to link the drawing to the figures. The use of the observer allowed a drawing to remain independent of its figures maximising the potential for reuse but making it difficult for a figure to determine which other figures were on the same drawing. Students often wanted this information and so would directly couple the figure and the drawing together despite the architectural problems this would cause (in the above example the framework was unable to initialise because the figure referenced a null drawing view, the drawing view being created later in the initialisation code!).

Architectural problems represent the smallest of the categories identified in this study. In some cases they are caused by a lack of foresight, such as the example of the tool which modified more than the target figure. In other circumstances they represent a lack of awareness of the high level design of the framework (as in the case of the drawing and the figure being merged together).

3.7.5 The significance of reuse problems

The above categorisation, with hindsight, might appear quite obvious but it is nonetheless useful because it describes the scope of problems that effective documentation must address. It is *not* obvious that previous research, proposing framework documentation techniques, has recognised the need to address this range of problem categories. Existing approaches often only partially address these issues providing limited support for framework re-users and requiring a combination of techniques to be deployed.

The relative frequency for each of the problem types suggests that functionality and interaction problems are the most common types experienced during reuse. These are followed by a smaller number of mapping and architecture problems. This says little about the relative importance of each problem category. In fact it could be argued that problems like mapping and architecture, although less frequent, are actually more important than the other categories. While functionality and interaction tend to focus on problems in classes or methods, architectural and mapping problems deal with much wider issues to do with the choice of a solution within the framework. Therefore these problems cause more disruption to the reuse process when they cannot be addressed by documentation. Mapping and architecture may share even more in common. It can be argued that documentation that assists developers to map solutions onto the framework ought to do it in an architecturally compatible manner. This would prevent architectural issues arising later during development, effectively addressing the two problems with one form of documentation. Mapping also stands out as a problem that may occur early during framework reuse as it arises when the developer is deciding upon how to implement a given requirement within the framework. It is followed by a range of functionality, interaction and then architectural problems as the solution is developed further. This makes mapping especially important because if it is performed well it may help to reduce subsequent problems from the other categories. This suggests that it is worthwhile investing heavily in documentation that can address mapping issues as this could have a corresponding benefit for all the other problem categories.

3.7.6 Related work

Despite the lack of empirical investigation into framework reuse problems there is a limited discussion in the literature about the type of problems that can occur during reuse and some evaluation of problems found in the wider field of software maintenance. There appears to be a similarity with the observations made here and those found in the literature which provides some confirmation of the identified problem categories.

Support for the mapping problem can be found in work from the program comprehension community. Ruven Brooks (Brooks 1977) describes a model of program comprehension in which a programmer hypothesises about the structure of the software using a mixture of domain knowledge and programming experience. The programmer subsequently tries to confirm the conjecture with a detailed study of the program source. Searching the code either supports or invalidates the hypothesis. This model of anticipation followed by investigation mirrors closely the process that was discovered during framework reuse by this study. In particular it is closely related to the problems of mapping and functionality which can be seen respectively as analogous to the process of hypothesising about the expected structure of the framework and the difficulty of comprehending the source code when trying to confirm a hypothesis.

Further evidence for the mapping problem can be found in the work of Johnson (Johnson 1992). He argues from his experience working with and developing object frameworks that documentation should explain how to use a framework. Johnson claims that this requires the communication of the purpose of the framework and its individual parts. This description is similar to the information developers reverse engineer from a framework as they perform a mapping task. Each successive mapping teaches more about the purpose of that region of the framework and helps to explain how it can and should be used.

Butler, Keller and Milli describe a taxonomy of framework documentation primitives that bears some resemblance to the problems identified here (Butler, Keller and Milli 2000). They describe six primitives which are derived from their knowledge of framework reuse. The first of these, *signature of the participants (SP)* is defined as, an enumeration of the interfaces within a framework; This is a trivial although significant part of the mapping problem. It relates to the idea of seeing the framework as a collection of smaller entities and is a first step towards understanding how each element differs from its neighbours and when they should be used.

Further support for the mapping problem is provided in a survey of framework programming environment support by Fairbanks (Fairbanks 2004). He suggests a number of questions (derived from a logical argument) that developers can ask of their environments when trying to understand a framework. One of these questions is *How do I accomplish this?* Fairbanks observes that this question is harder to answer in a framework because the space of possible solutions is restricted by the existing structure of the framework. This question appears to be closely related to the mapping problem. The programmer needs to know what structures exist within the framework and has to be able to map their notions of a solution onto those structures. Failure to make the link between the existing structures and the desired solution causes mapping problems.

Johnson provides limited support for the interaction and functionality problems presented in this study. He argues that frameworks re-users should delay exposure to the detailed design of the framework for as long as possible, preferring instead to use the framework in a black box manner. However, he concedes that eventually developers will need access to implementation details and that this demand should be met by documentation. Johnson's notion of detailed design seems similar to the requirements of interaction and functionality problems described in this work, although Johnson's claim that these details should be delayed as long as possible does not appear to be borne out by this research.

Butler, Keller and Milli's taxonomy of framework documentation primitives also appear relevant to the problems of interaction and functionality. They describe five primitives which appear to be relevant: *Behavioural specification of the participants (BSP)*: a description of the behaviours of the framework classes; *Computational specification of the participants (CSP)*: a domain independent and static behavioural specification of the participants; *Structural dependencies between the participants (SDP)*: a description of the structural relations between the framework classes (both static and dynamic); *Behavioural dependencies between the participants (BDP)*: a description of inter-object behaviour, which is often a consequence of the structural dependencies; and *Computational dependencies between the participants (CDP)*: a domain-independent BDP. The behavioural and computational specification of participants appears to be a call to understand the detailed workings of parts of the framework, while the structural, behavioural and computational dependencies appear to be closely related to the problem of understanding the interactions in a framework.

Rosson and Carrol describe a detailed evaluation of software maintainers. They observed four programmers performing two small scale reuse tasks in a graphical user interface framework (the tasks were to create a colour mixing application and to add a hierarchical view to an existing library acquisitions application). Their main observation was that programmers, when reusing a class, where much more interested in seeing an example of that part in operation rather than reading its implementation detail. This finding echoes the result of a similar study performed by Lange and Moher (Lange and Moher 1989). Although not definitive such studies seem to indicate that developers are reluctant to study source code in detail. This work provides some support for the functionality and interaction problems presented in this work as it suggests that developers are not willing or able to digest large amounts of implementation detail from the source code.

Fairbanks (Fairbanks 2004) also describes a couple of questions which appear relevant to an understanding of the detailed design of a framework. He suggests that programmers ask *have I done all I need to do?* when making a modification and ask *what is going on here?* when browsing existing parts of the framework. These questions appear closely related to the problems of interaction and functionality.

Van Grup and Bosch (van Grup and Bosch 2001) describe the problem of design erosion which appears to be similar architectural problems identified in this study. Design erosion relates to the gradual worsening of the quality of a design over time. Van Grup and Bosch's observations are derived from several studies of software architectures in industrial situations and are supported by anecdotal evidence in the architecture literature. The authors cite several reasons for design erosion but one that is particularly relevant to this work is the notion of architectural drift. They claim that drift occurs in situations where code is maintained by developers who do not fully understand the design and make sub optimal decisions during maintenance. Over time this erodes the architectural assumptions behind the design and can make the code more difficult to change in the future. This is a similar situation to the one facing re-users when tasked with making a modification to the framework. Often they must make their modifications without a full understanding of the surrounding system and this can lead to architectural entropy.

Krueger, in his survey of software reuse (Krueger 1992), describes a model for reuse that is consistent with the four problems identified here. In his opinion reuse can be described as a three-phase activity: selection, specialisation and integration. Selection identifies the relevant artefacts for reuse; this is similar to the problems of mapping and understanding functionality. Specialisation involves the customisation of the relevant artefacts to meet the needs of the new situation; this is represented in our investigation by the problems of understanding the behaviour of the part and understanding the architectural roles that surround it. The final phase, integration, describes the process of placing an artefact within the flow of control of a larger system. This relates to the problem identified as understanding interaction.

3.8. Questionnaire

To confirm the results of this study a questionnaire was sent to the students of the software architecture class, independently of the cluster analysis, asking their opinions on the problems they had experienced and the documentation they had used. The questionnaire was delivered to students via email and was sent a couple of days after the submission date for the class coursework assessment. It contained two questions about reuse problems and documentation use, each with a number of parts. Participation from the students was voluntary. The construction of the questionnaire had been informed by experience of reuse problems collected from the individual developer study. The questionnaire explained the purpose of the questions and how to respond. In total sixteen members of the class responded to the questionnaire, from a class size of seventy seven (approximately a quarter of the class), which is a reasonable response rate for this type of collection technique (Edwards 1972). The complete questionnaire is available in Appendix A.

Question 1	How difficult did you find understanding the following aspects of the JHotDraw framework?
1.1	Understanding individual classes and their methods.
1.2	Using abstract classes and interfaces.
1.3	Mapping your solution to framework code.
1.4	Understanding the structure of inheritance hierarchies and object compositions.
1.5	Understanding design patterns.
1.6	Understanding the dynamic structure of the framework.
1.7	Choosing from alternative framework solution strategies.
1.8	Understanding the HotDraw problem domain.

Table 1: Rating the difficulty of framework problems

Question 2	How effective have you found the following in solving your problems during the practicals?
2.1	Browsing JavaDoc files.
2.2	Using JHotDraw Pattern Language.
2.3	Design pattern knowledge.
2.4	Asking lecturer.
2.5	Asking newsgroup.
2.6	Asking other students.
2.7	Studying existing examples.
2.8	Previous practical solutions.

Table 2: Rating the available documentation

The first question (**Table 1**) described a collection of potential problems that students may have experienced during framework reuse and asked students to rate them according to their perceived difficulty. Participants were asked to rate each problem on a five-point scale ranging from difficult to easy. The second question (**Table 2**) explored documentation utility, collecting information on how helpful the participants considered each technique to be. Responses were also on a five-point scale and varied from not helpful to very helpful.

Table 3 shows the distribution of responses to each part of question 1. It can be seen from the table that question 1.6, understanding the dynamic structure of the framework, was rated the hardest of the activities. One possible reason for this is the omission of any documentation on the dynamic aspects of the framework's behaviour. Question 1.3, mapping the solution into the framework was rated as the second hardest, suggesting that students had difficulty in identifying relevant parts of the existing framework to modify. Question 1.7, choosing from alternative solution strategies, was considered difficult indicating that, where students could identify functionality, problems were still present in trying to select the best approach from those available. Question 1.5, understanding design patterns, was also considered hard by the majority of students. This may in part be due to the students' lack of familiarity with design patterns as they had only recently been taught them and this was their first experience of seeing instantiations in a real example.

Response\ Question	Very easy	Easy	Moderate	Hard	Very hard
1.1 Classes and methods.	1	8	5	1	1
1.2 Abstract classes and interfaces.		8	6	2	
1.3 Mapping solutions	1	1	2	11	1
1.4 Inheritance and compositions.		3	3	8	2
1.5 Design patterns.	1	1	5	9	2
1.6 Framework dynamics		1	2	8	5
1.7 Alternative solutions.		1	5	8	2
1.8 Problem domain.	2	4	5	4	

Table 3: Problems understanding framework aspects

Question 1.4, understanding the structure of inheritance hierarchies and object compositions, was rated as a moderately difficult activity. This is perhaps because the way in which inheritance and composition distribute knowledge across the system is common to the majority of object-oriented programs, therefore students had prior experience dealing with this kind of problem. Question 1.8[†], understanding the framework domain, shows a very even distribution between the hard, moderate and easy categories. It is possible that this spread can be attributed to variations in developer experiences of graphics-related domains.

The majority of students on the course had little experience of understanding abstract classes and interfaces (question 1.2). Initially the concept seemed to cause some confusion but after a brief learning curve the students appeared to grasp the idea. It was encouraging to see abstract classes and interfaces being introduced and understood in such a short period of time because they are such fundamental features of framework programming. Finally question 1.1, understanding classes and their methods, was rated the easiest activity in the table. This was unsurprising, as the students were all relatively experienced Java programmers.

Response\ Question	**No Use**	**Barely Useful**	**Moderately Useful**	**Useful**	**Very Useful**
2.1 JavaDoc		3	2	4	7
2.2 Pattern language.		1	7	4	4
2.3 Design patterns		5	10	1	
2.4 Asking lecturer		2	7	4	1
2.5 Asking newsgroup.			8	4	2
2.6 Asking students.		1	5	7	3
2.7 Examples.			4	6	6
2.8 Practical solutions.				7	9

Table 4: Usefulness of supporting materials and techniques

[†] One student omitted Question 1.8.

Table 4 describes how effective the students found each of the techniques intended to help framework understanding. The two most significant were questions 2.7 (Examples) and 2.8[‡] (Practical solutions). The fact that the coursework was a natural extension of the previous practical solutions is likely to have exaggerated their usefulness. The practical solutions can also be considered as a form of example, suggesting that the students found examples to be a very positive aid to understanding. JavaDoc was rated as the third most useful of the techniques. This demonstrates that, although there may be a need for additional support in framework learning, the basic forms of documentation are still useful in a framework environment.

The three questions 2.4, 2.5[§], 2.6 can be grouped together and considered as 'asking a more experienced individual for help' or mentoring. The results of these three questions are all similar with the majority of students rating them as a useful source of information. Presumably, one of the key advantages in asking another person about a problem is that a dialogue can then take place, answers can be clarified, additional questions asked etc. The flexibility of this approach is its main advantage over other forms of learning.

The JHotDraw pattern language was rated as moderately useful by the students. This was disappointing because it is a technique specifically intended to address framework problems. The pattern language was only presented to the students halfway through the practical exercises (due to a delay in conversion). This may have reduced its benefit because by the time it was available students had already some familiarity with the concepts within the framework. Perhaps if it had been available earlier the students would have gained more benefit and rated it higher.

The design pattern knowledge was the final form of documentation rated by the survey. The students generally considered this to be the least useful of the techniques presented (but still quite useful). It appears that the students' lack of experience with design patterns may have reduced its ability to act effectively as framework documentation.

The results of the questionnaire appear to support the problem categories identified by the clustering. The questionnaire was independent of the cluster analysis but still addressed the same population, and had taken place prior to clustering. The problems it discovered appear

[‡] Due to a typing error Question 2.8 appeared as Question 2.9 in the original questionnaire.

[§] Two students omitted Questions 2.4 & 2.5.

to confirm the existence of the four problem categories. Mapping problems are addressed by questions 1.3 and 1.8, interaction problems by questions 1.4 and 1.6, functionality problems by 1.1 and 1.2 and architecture by question 1.7 (where the search for alternative solutions can be seen as an attempt to find the most architecturally compatible solution). The results of the questionnaire suggest that developers found interactions and mapping to be particularly hard tasks during reuse, while architecture problems were of medium difficulty and functionality problems were considered the easiest of the problems encountered.

The questionnaire also provided a preliminary impression of the documentation used during this study. It suggested that users found examples and practical exercises the most useful of the documentation provided, followed by JavaDoc, mentoring and the pattern language. Least useful was the design pattern information, although the fact that developers rated design patterns difficult to understand (question 1.5) may suggest that the student's unfamiliarity with patterns contributed to its poor performance. None of the available documentation was considered to be completely useless during reuse implying that developers were grateful for the support offered by all documentation and that some information was better than none.

Source code and UML were overlooked in the design of the questionnaire so it is not clear how those forms of documentation compare with the other techniques. The questionnaire also only presents a very rough measure of documentation utility as it does not provide any insight into the reasons why developers rated documentation as they did. In the next section a more detailed review of the available documentation will be presented.

3.9. Documentation review

Having identified four major problem categories for framework documentation it seems appropriate to consider what support existing forms of documentation provide for them. An exhaustive evaluation of all possible framework documentation would not have been practical in this study, so this review only considers the techniques which are available for the JHotDraw framework. These include JavaDoc, a pattern language, design patterns, UML, practical exercises, examples, mentoring and the framework source code. In each case evidence to support the claims made about documentation is provided from data collected during the study.

3.9.1 JavaDoc

JavaDoc is an appealing form of documentation because of its relative simplicity and familiarity. It appears most relevant for addressing questions of functionality about a framework. In particular its ability to provide textual overviews of each class' behaviour can be a considerable advantage as it prevents the developer from having to spend time understanding the framework source code. *"Without these [JavaDoc] files it would have been near impossible to determine what functions a particular class provided" (Coursework reports).* However, such textual overviews often vary in their quality and sometimes do not provide sufficient support for developers to avoid having to consult the code. *"I tried to use the documentation provided within the package HotDraw. Although there is quite a large amount of this, i.e. there is a JavaDoc page for every class in the framework, I found that each individual page did not contain enough information to be used as the sole tool for learning the framework…I thought that the documentation should have contained a more in-depth description of what each method did" (Coursework reports).* JavaDoc also appears to help navigation across a framework's source code as it features hypertext links which allow rapid traversal between classes and up and down inheritance hierarchies. While some developers found this useful, *"Firstly there were the JavaDoc files. These provided a form of 'roadmap' that was used to determine a path through the framework hierarchy to see the relationships of particular classes and functions" (Coursework reports)*. Others did not find this useful, complaining that the rapid changes made them become disoriented and forget what they were doing. *"Generally, determining what methods were available in any one class and which capabilities HotDraw provided was awkward, due to the need to check all super classes through several layers of inheritance, as well as the interfaces implemented by the super classes" (Coursework reports).*

3.9.2 Pattern languages

Pattern languages have been targeted specifically as a form of framework documentation. This suggests that they might be more supportive of framework reuse problems than other more generic techniques. Indeed pattern languages appear to offer some level of support for all of the four problem categories. They support mapping problems by identifying potential classes suitable for a given circumstance, *"I already knew that there was an abstract class called Action Tool that would handle performing an action on a figure because I had read about it in [the pattern language]" (Coursework reports)*. They support interaction and functionality problems by showing how classes can be used via illustrative examples. *"For the animation there was no other way of understanding how to use this feature as there is no examples within HotDraw and I think that it would have been very difficult if [the author] hadn't written the [pattern language] or it wasn't covered within the practicals" (Coursework* *reports)*. They also promote architectural consistency by suggesting valid solutions and offering advice about which solutions might be most appropriate. *“[The pattern language] gave me the idea for using menus to undecorate figures. In the paper [the author] states that menus are preferable for selection oriented events” (Coursework reports)*. However, pattern languages are not complete in their support. Developers criticised the pattern language for under-supporting some areas of the framework domain. *“[The pattern language] was a good aid, but I thought the solution descriptions were too brief (and wholly incorrect on occasion)” (Coursework reports).* There are also questions about which format of pattern would be most effective for communicating information to developers.

3.9.3 Design patterns

Design patterns are commonly used within object-oriented frameworks. This suggests that a designer with a good understanding of common design patterns could be at a distinct advantage when learning to reuse a framework. In particular patterns might be expected to help developers gain an understanding of the interactions that occur between elements of the system and also to help them to appreciate the architectural roles that must be enforced when making modifications to the framework. They can also assist during mapping, because each pattern provides a rationale to explain the reasons for adopting the given solution and they also support functionality because they explain how the parts of the pattern operate. The evidence from this study suggests that design patterns were not very helpful as a form of documentation. The comments provided by developers about design patterns focused more on how they had implemented new design patterns in their solutions rather than maintaining existing ones. *“I used the template method design pattern to factor out the duplicated code in my ellipse figure classes…” (Coursework reports)*. They also tended to make general comments about the benefits of design patterns without commenting on their specific relevance to frameworks. *“The extensive use of design patterns enables programmers with experience of such patterns to quickly obtain a good understanding of the framework” (Coursework reports)*.

Evidence from the questionnaire sheds some light on these responses because it suggests that developers found design patterns difficult to understand and use (most respondents rated patterns moderately to barely useful). This claim is strengthened by comments made during the study where students admitted that rather than patterns helping to explain the framework it was actually the other way round with the concrete implementation of the framework helping to explain the purpose of the pattern *“We touched on it [the decorator pattern] in the class before we saw it in HotDraw. So perhaps [I] didn’t understand it that much in the class but once I saw the HotDraw example it was a bit easier to appreciate it”*

(Coursework reports). Perhaps, because students in this study had only recently been introduced to design patterns, their unfamiliarity limited the potential to use them as documentation.

3.9.4 UML

The UML overview provided with JHotDraw is little more than a description of its high level interfaces and the important relationships between them (**Figure 21**). As such it is not fully representative of the power and expressiveness available from UML documentation.

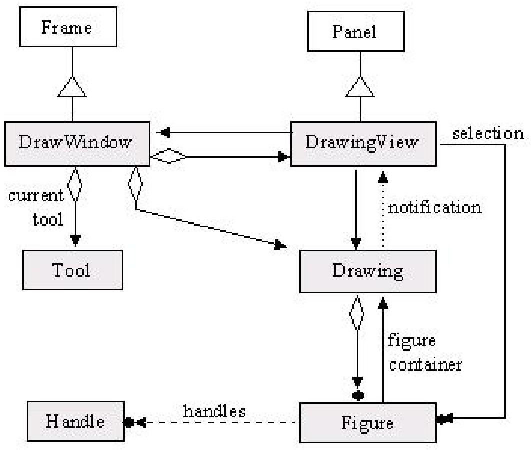


Figure 21: UML Overview of JHotDraw

Nevertheless it was provided with the framework and gives an understanding of the key abstractions in the framework and thier significant relationships. As such the overview might be expected to contribute towards mapping and interaction type problems. Few developers passed comment upon its utility. One of the few mentions it does receives indicates that it was at least partially helpful for identifying the roles that exist within the framework. *"From analysing a UML diagram representing the Drawing class and the figures that are stored in Drawing, I found that Drawing is an interface class. I found that by overriding a method in the Standard Drawing View called createDrawing I was able to achieve my goal" (Coursework reports)*.

3.9.5 Practical exercises

The set of practical exercises used during the software architecture class were roundly praised by many of the developers who used them. They cited the benefits of practicals as the ability to introduce the main concepts within the framework. *"The previous practical exercises gave a solid introduction to the framework, revealing that no matter how baffling a* *requirement may be, there will usually be a straightforward way to go about implementing it" (Coursework reports)* and also to provide support for how to use many common parts of the framework. *"I reused the code from practical 5 to remove all functionality from the handles and changed the colour of the figure in much the same way as I had changed the colour of previous figures" (Coursework reports)*. This suggests that practicals could help to support functionality and interaction problems within the framework. Mapping and architecture problems may have also been supported, after the fact, by the inclusion of sample solutions which attempt to show best practice within the framework. It should also be noted that the practicals may have only performed so well as documentation because they were designed to teach the specific skills required of students for the final assessed task. This close relationship between documentation and task is unlikely to be reflected in other situations, which may reduce its effectiveness.

3.9.6 Examples

Examples were another form of documentation that developers responded positively about. Once again they helped to introduce framework concepts to the developer, *"By reviewing the demo applications that came with HotDraw package … I was able to see where certain classes and methods could be used" (Coursework reports)*, and providing practical guidance on how to implement solutions using those concepts, *"In order to work out how to create and align the individual figures I paid particular attention to the PertFigure class within the Pert Application example as it used the composite figure class" (Coursework reports)*. This suggests that examples may provide some support for mapping, interaction and functionality problems. Examples were criticised for encouraging a piecemeal form of development which could have negative consequences for the architecture of a system. *"The lack of 'how-to' documentation proved frustrating – the code of the example applications was critical in gathering an understanding of how to use JHotDraw. Just referring to the code however, resulted in me searching for a solution to a single aspect of the problem at a time, resulting in a 'Frankenstein' solution" (Coursework reports)*. Another problem with examples is the difficulty of providing adequate coverage for a range of different tasks. Whereas a pattern language might describe a general concept and use an example to illustrate its purpose, examples are by themselves much more specific about what they can teach about the framework. This suggests that example based documentation may struggle to provide ample coverage for the range of modifications possible within a framework.

3.9.7 Mentoring

Mentoring, the process of using more experienced developers to coach or guide novices through a framework modification, can offer effective support for all types of problem category. Its greatest strength is possibly its dynamic nature where the mentor can react to the developer to compensate for areas they find hard or to skim over areas they find easy. Another chief advantage of this approach is that novices can gain early criticism about their intended design. This can help to ensure the architectural consistency of the framework and it also saves the developer a lot of time and effort. In this study mentoring was used as part of the software architecture practicals: developers were encouraged to talk through their solutions with members of academic staff who had some familiarly with the JHotDraw framework. Students also received a form of mentoring through participation in the class newsgroup. Despite its utility, mentoring is too expensive to be applicable to mainstream framework reuse as it requires experienced developers to be diverted from software development and instead train novices (although its potential use in conjunction with pair programming (Beck 2000) may increase its feasibility). It also suffers from the problem of how to train a set of mentors in the first place! They have to somehow learn the framework to begin with, so the problem of how to describe the framework with other forms of documentation remains.

3.9.8 Source code

The framework source code is the last line in framework documentation. Developers must use this to gain an understanding of the system when all other forms of documentation have been exhausted. As such it can play a role in either functionality or interaction problems, but source code is difficult to understand and developers can have problems sifting through the volume of material to find items of interest. *"The most difficult aspect concerning the understanding of the framework was attempting to find where instances were created, where methods were implemented and where methods were actually called. The number of levels in the inheritance hierarchy make it difficult to see what methods can be called by a particular class. It was therefore necessary to have many classes open in an editor at once, often more than 15, and the Java API in another window in order to fully follow the associations and inheritance present in the framework" (Coursework reports)*. Its biggest strength is its accuracy, the code defines the behaviour of the framework and developers often responded positively to this during the evaluation, *"…the most useful source of information when developing a solution … was the framework source files. These provided invaluable implementation details and increased my understanding of HotDraw a great deal more than the JavaDoc files could have done alone" (Coursework reports)*. Despite this, source code will continue to be a last resort for framework comprehension. Ultimately it is designed to express concepts in a machine, not human, readable form and as long as this is the case there will always be a need for more human friendly sources of information.

All forms of documentation presented during this study were considered to be useful by developers but the presence of reuse problems despite the documentation confirms that the existing approaches on their own are not enough. Also, specific forms of documentation appear to favour subsets of reuse problems (e.g. pattern languages for mapping type problems, source code for functionality). This suggests that it is unlikely that a single documentation will be found that can support all reuse problems; instead combinations of documentation will have to be sought to address different aspects of reuse.

3.9.9 Summary of Documentation techniques

Problem/Documentation	Mapping	Interactions	Functionality	Architecture
JavaDoc	✗	✗	✓	✗
Pattern Language	✓	✓	✓	✓
Design patterns	✓	✓	✓	✓
UML	✓	✓	✗	✗
Practical Exercises	✓	✓	✓	✓
Examples	✓	✓	✓	✗
Mentoring	✓	✓	✓	✓
Source code	✗	✓	✓	✗

Table 5: Summary of documentation evaluation

Table 5 presents a summary of the results of this evaluation. In the table ticks represent situations where documentation has at least some potential to support a problem category, while a cross means that the technique offers no (or very marginal) support. Pattern languages stand out as a form of documentation that could better address mapping problems in the future. Their ability to introduce new concepts from the framework domain and to show how they can be implemented is directly relevant to the mapping task. Other approaches which were relevant to mapping such as design patterns, practical exercises and examples can be discounted as they only provided partial coverage of the framework, while mentoring, although effective, is infeasible because of its high cost.

The pattern language used in this study, being a translation from a different implementation, did not always provide enough detail specific to JHotDraw. This was only realised after the study had begun and the author had developed a better familiarity of the framework. With this insight a better pattern language could be created providing more patterns to address the range of capabilities in the framework and to provide greater accuracy in each pattern with respect to the JHotDraw implementation.

Many techniques appear appropriate for functionality and interaction problems (for example JavaDoc, design patterns and UML). However the existing approaches appear to suffer from a trade-off between coverage and detail. Approaches that cover the entire framework, e.g. JavaDoc, do so at the expense of detail, while approaches that provide detail (e.g. design patterns) cannot describe the behaviour of the entire framework. To become more effective a balance has to be found between describing the implementation of the framework in enough detail and providing thorough coverage for all areas of the system.

Architectural problems could be addressed via a range of different techniques. Documentation such as design patterns and pattern languages can provide support for architectural constraints as they provide a rationale for the design helping to enforce the system’s architecture. Despite this, it can be argued that architectural problems are best addressed during the mapping phase of reuse. If documentation can address mapping properly then it may minimise the subsequent architectural issues that can arise. Pattern languages appear to be an important mechanism to provide such support as they can address mapping issues while at the same time informing about architectural relationships that ought to be upheld.

3.10. Conclusions

This chapter has described an investigation into framework reuse problems. Mapping, functionality, interactions and architecture have all been identified as problems that affect framework reuse. The performance of common forms of documentation has also been evaluated to assess which approaches should be the focus of future work. Three separate studies have been performed, each looking at framework reuse problems in different ways. The first study, a personal investigation, recorded the observations of the author as he attempted to create an application with a framework. His experiences were recorded in a logbook. The second study captured the experiences of students enrolled in a software architecture course. They were asked to create a framework modification as part of their assessment for the class. Their experiences were recorded in coursework reports, a class newsgroup and via a questionnaire. The final study recorded the reuse process of four students using a framework as part of their final year projects. This development occurred over a period of six months and provided an in-depth exposure to a framework. Their experiences were captured via semi-structured interviews.

All three studies represent a wealth of reuse experience each recording a number of framework reuse problems during their modifications. The problems from each study were collected and combined together in a cluster analysis process to identify the common characteristics of reuse problems. The resulting categorisation suggests that four problems dominate framework reuse: mapping solutions onto the implementation of the framework, understanding the interactions that occur between framework classes, understanding the functionality of parts of the framework and understanding the architectural constraints that exist within the framework. In the future documentation should seek to address these problems to assist developers during reuse.

This chapter has also summarised the performance of existing forms of framework documentation. This was done by asking developers to record their opinions on the documentation that was available to them during reuse. This work suggests that, although all forms of documentation are appreciated by developers, some are more applicable to reuse problems than others. In particular pattern languages stand out as a useful mechanism to address the mapping problem as they can communicate details about the expected use of key parts of a framework. Functionality and interaction problems, obvious targets for documentation to address, are not well supported by existing techniques. Design patterns and source code provide partial support but new techniques need to be considered to address the large amount of, often subtle, implementation detail present in software frameworks. Architectural issues appear to be related to mapping problems, as they indicate a failure to consider architectural qualities during the initial design of a modification. This study therefore contends that architectural issues ought to be addressed by documentation during the mapping phase of framework reuse.

4 Documentation for framework reuse

4.1. Introduction

This chapter describes the development of a pattern language and a micro architecture based documentation to address the problems of framework reuse. Previous chapters have summarised the state of the art in framework documentation and have provided some evaluation of existing techniques. During this evaluation it was noted that there are few forms of documentation that are capable of addressing the problem of mapping solutions into the language of a framework. Notable exceptions are pattern languages and sets of examples. The limited evaluation that has been performed on both forms of documentation suggests that although examples are accessible and convenient they do not have the clarity or the coverage of a pattern language. Indeed pattern languages are often a good place to exhibit examples because the surrounding pattern can provide context for the example and draw out the lessons to be learned. In addition because a pattern language describes how parts should be used it may also provide support for the problem of maintaining the architectural consistency of the framework.

The problems of understanding the functionality and interactions that occur within software are by contrast frequently addressed by existing documentation. Techniques such as UML, JavaDoc and design patterns are some of the more common approaches employed to explain implementation details about a framework. Evaluation of these techniques suggests that although many of them are useful, none of them are entirely effective at explaining how a system works. In particular many of the existing techniques have difficulty scaling to address the large size of software frameworks. Existing approaches either provide the information all at one time, in which case it quickly becomes overwhelming or else they intentionally omit material, leaving gaps in the coverage provided. In this chapter an alternative approach will be considered where implementation details will be explained by a set of micro architecture descriptions. Micro architectures are a way to decompose the structure of a framework to illustrate how the framework implements domain functionality e.g. (Lajoie and Keller 1994). They allow a framework to be understood in small meaningful pieces but are created to allow the pieces to fit together to create a cohesive explanation of the whole.

4.2. The pattern language

Mapping problems require documentation to present information about how the framework is expected to be used and to identify possible solutions that will preserve the architectural quality of the existing framework. Pattern languages seem a likely candidate for this role as they provide an opportunity to introduce aspects of the framework, explain the problem they address and show how they can be used.

The pattern language created for the earlier work was derived from an existing language for a different implementation of HotDraw (Johnson 1992). The use of a pre-existing structure allowed the pattern language to be created quickly for JHotDraw but it did not allow the language to fully describe the unique attributes and features of the Java implementation. The initial evaluation suggested that the first pattern language was generally useful but lacked detailed guidance and coverage of some important topics. For example, it did not describe how and when to use composite figures or locators within the framework and provided only a cursory overview of important topics such as creating a tool or handles. The language was also very lightly interconnected which made it difficult to navigate through when searching for particular topics. This indicated a need for improvement in three directions: in the completeness of coverage, in the technical depth of description and in the number of relationships between patterns in the language.

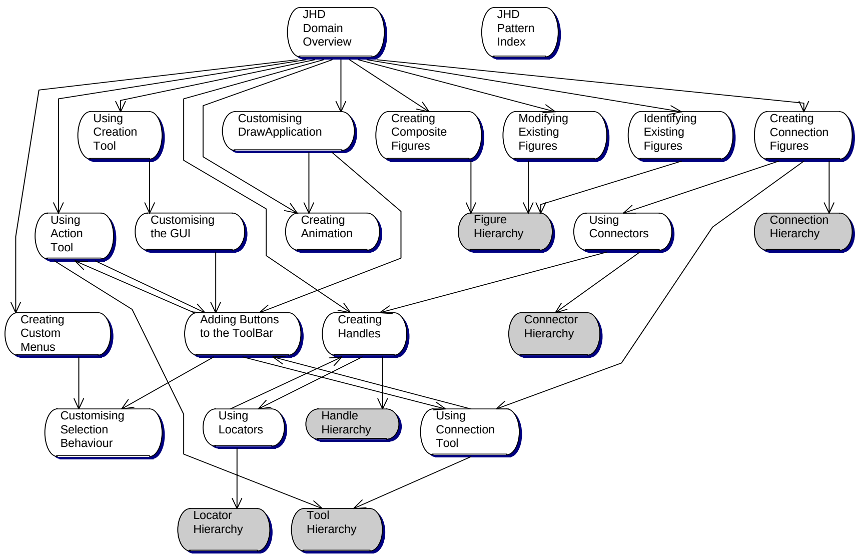


Figure 22: Overview of second pattern language

(The dark grey boxes are UML diagrams)

Improvements were also suggested from the pattern language literature. These included the addition of source code examples (Lajoie 1993) and UML class hierarchies (Meusel, Czarnecki and Köpf 1997). The content of individual patterns retained the textual narrative of Johnson's patterns rather than the more algorithmic descriptions of Meusel et al. (Meusel, Czarnecki and Köpf 1997) and Froelich et al. (Froelich et al. 1997) as these are focused upon isolated modifications and lacked the motivation and architectural awareness that was present in Johnson descriptions.

The pattern language was improved in four different dimensions:

- **Increased number of patterns:** The number of patterns in the second language was increased from 8 to 18 (**Figure 22**). Some of the patterns were created by dividing and expanding existing patterns into separate topics (for example 'Defining Drawing Elements' was divided into 'Identifying Existing Figures', 'Modifying Existing Figures' and 'Creating Composite Figures'). New patterns were also created to address issues which were not present in the Smalltalk implementation but were relevant to JHotDraw. For example, concepts such as creating handles and locators within the framework were identified through experience and patterns were created to support these topics.
- **Detail added to each pattern:** The detail of each pattern was also improved (**Figure 22**). Every pattern was designed to introduce a concept from the framework domain. Paragraphs at the start of the pattern would describe the role the part played in the framework and describe how it may be used. Images were often used to help illustrate these descriptions (for example when talking about handles, images of the different types of handle available are used to assist the textual description). A more significant difference was that a large number of patterns featured source code examples. These reinforced the description in the pattern by showing a concrete example of the part in use. The fragments of code were often incomplete showing only the minimum amount of code to illustrate a topic. The intention was that the code should be read and understood and the relevant information extracted from it, not to cut and paste the example into a developer's solution. The patterns were augmented with the addition of six class hierarchies (Figures, Tools, Locators, Handles, Painters and Connectors). These described important hierarchies that developers often had to select elements from. The class hierarchies were created as separate patterns (with no textual description) and were linked to any pattern where a choice from the hierarchy was relevant.
- **The network of patterns was enriched:** The original pattern language only contained eight links between its patterns and only one pattern was reachable from multiple areas of the language. With such a small language this was not a problem but as the language grew it became more desirable to create many different paths through the patterns.

Multiple paths allow developers to find new ideas that are related to or contrast with the current pattern. This type of structure can help them to discover new parts of the framework by relating similar patterns together. A pattern language of 18 patterns is perhaps still small enough that the density of relationships is not yet very critical but it seems reasonable to assume that as a language grows the relationships between the patterns will become an important navigational aid to the re-user.

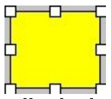
- **An overview pattern was created:** Johnson's pattern language placed a great deal of importance on the initial pattern. This was intended to guide developers to other patterns of relevance in the language. The first implementation of the JHotDraw pattern language had a simple version of this that led the user to one of three possible start points in the language. With the increase in size of the new language it was possible to make this initial pattern much larger, linking it to the majority of other patterns directly. The overview pattern was also accompanied by an index pattern, which provided links to all the patterns, listed in alphabetical order for easy reference (so that if a developer already knew a pattern of interest he or she could jump directly to it).

The new pattern language was created with a far deeper understanding of the framework's structure than the previous language. The major elements of the framework are all represented by a pattern and the patterns themselves have been strengthened with the inclusion of class hierarchy information. The patterns also provide detailed code examples to illustrate how parts of the framework can be used.

These improvements were intended to make the pattern language better able to support users when faced with mapping problems during reuse. The wider number of topics addressed by the pattern language ensures that the major features of the framework are represented within the documentation and increases the chance of finding a relevant pattern. The increased technical depth of each pattern provides a better description of how to use parts of the frameworks and also helps to discriminate between potential solutions. The larger number of links between patterns and the improved overview pattern improve navigation by connecting related information together. Complete versions of both the first and second iteration of the pattern language can be found in Appendix A.

JHotDraw Pattern Language

Creating Handles



A selected figure displaying resize handles

Direct manipulation of figures on a drawing is achieved through the use of Handles. The AbstractHandle class implements the Handle interface and provides default behaviour for all handles in the framework.

- For futher information about the Handle hierarchy see The Handle Hierarchy

JHotDraw predefines several types of handle; they include ChangeConnectionHandle, ElbowHandle, LocatorHandle and PolygonHandle. It should be noted that because Handles tend to be specific to the figure they were created for, opportunities of reuse across different types of figure are rare. Therefore developers should expect to have to write their own handles either by sub-classing AbstractHandle or one of the above classes.

To add a handle to a figure the figures **`handles()`** method must be overriden. This method is called by other parts of the framework to draw the selected figures handles on the DrawingView.

Resize handles are often required for figures in JHotDraw applications therefore the framework provides a utility class (BoxHandleKit) which simplifies adding resize handles to a figure.

- How to add handles to a figure (example from GroupFigure).

```
public Vector handles() {
 Vector handles = new Vector();
 handles.addElement(new GroupHandle(this, RelativeLocator.northWest()));
 handles.addElement(new GroupHandle(this, RelativeLocator.northEast()));
 handles.addElement(new GroupHandle(this, RelativeLocator.southWest()));
 handles.addElement(new GroupHandle(this, RelativeLocator.southEast()));
 return handles;
}
```

- How to add handles to a figure (using BoxHandleKit).

```
public Vector handles() {
  Vector handles = new Vector();
  BoxHandleKit.addHandles(this, handles);
  return handles;
}
```

When creating custom handles the dynamic behaviour of a handle has to be understood. Handles define three important methods. **`invokeStart()`**, **`invokeStep()`** and **`invokeEnd()`**. These methods are called when the mouse is respectively clicked, dragged and released on top of a handle. Every interaction with a handle will therefore follow a sequence where **`invokeStart`** will be called, **`invokeStep`** may be called (if the mouse is dragged) and **`invokeEnd`** will be called when the interaction ends. This granularity across the interaction allows the developer to control how the handle responds to the user input.

The appearance of a handle can also be altered. This might be appropriate to indicate the action the handle will perform or the current state the handle is in. To change a handles appearance override its **`draw(Graphics)`** method.

To create handles at a position on a figure the **`locate()`** method should be redefined. This method returns a point around which the Handle will be centred.

- For more information on positioning Handles see [Using Locators]

Figure 23: Excerpt from second pattern language

4.3. The problems of interaction and functionality

The existence of interaction and functionality problems during framework reuse suggests that re-users have difficulty understanding the existing structural and behavioural details of object-oriented frameworks. This is unsurprising as frameworks are typically large structures that contain a lot of interaction between their constituent parts. They also tend to use potentially unintuitive code structures such as design patterns, abstract classes and polymorphism which can make them even more difficult for re-users to understand.

4.3.1 Existing Documentation techniques

Many documentation techniques exist which claim to address the problems of understanding interaction and functionality. Techniques such as JavaDoc, design patterns and UML provide some support for these problems. In the previous chapter an evaluation of framework documentation suggested that existing techniques are only partially useful for functionality and interaction problems. In particular there again appears to be a trade-off between the depth and breadth of coverage that documentation can provide. JavaDoc for example can address all of a system but tends to provide a very cursory description of its functionality and provides no insight into its interactions. Design patterns on the other hand, can provide a detailed description of the interactions and functionality that occur in a part of the system but cannot provide this coverage over the entire framework.

A potential solution to this problem may be to decompose the framework into a number of smaller subsystems and document each of them in isolation. Each section could then be described in some detail and combinations of them could be used to address the entire framework. By introducing individual parts of the system one at a time a re-user has the opportunity to understand the framework as a collection of separate yet interacting mechanisms. This should reduce the amount of information to be understood at any one time while still allowing that material to make sense in the context of the larger framework.

4.3.2 Decomposing a framework

To some extent the object oriented paradigm already helps re-users to decompose a framework by partitioning its functionality into a collection of classes. Each class describes an abstraction that is relevant either within the domain of the framework or from a more general programming context. Classes are very useful for the comprehension of a software system. Re-users can learn, by studying the source code, what operations a class supports and what behaviour occurs whenever an operation is invoked. On the other hand, frameworks are more than just a collection of classes they also define how those classes interoperate. This makes comprehension harder because re-users must understand not only the class but the sequence of interactions that surrounds it. This implies that the decomposition of a framework must occur at a coarser granularity than the class.

The idea that functionality may exist in larger groupings resonates with the experience of the author. It was found that, after a period of time developing with JHotDraw, it was understood in terms of clusters of interacting classes. These clusters were repeatedly used by other parts of the framework and appeared to define behaviour that was significant to the framework domain. For example, a particular combination of method calls would occur between the drawing, its view and a few other subsidiary classes whenever a redraw of the screen occurred. Other combinations were identified for connections between figures, using handles and creating animation. Eventually the author began to depend upon these clusters to identify where and how to make modifications to the framework code.

Further evidence for the division of large object oriented applications into sub systems can be found within the software engineering literature. Design patterns, for example, share much in common with groups of interacting framework classes. In fact many patterns were identified by studying interactions within software frameworks. Gamma et al. describe patterns as *"micro architectures that contribute to the overall system architecture"* implying that an understanding of the design patterns within a system is important to the comprehension of the entire system (Gamma et al. 1993).

The only distinction between design patterns and the class structures found within frameworks are that design patterns contain a more general description of behaviour. *"An important distinction between frameworks and design patterns is that frameworks are implemented in a programming language. Our patterns are ways of using a programming language. In this sense frameworks are more concrete than design patterns"* (Gamma et al. 1993). Patterns are primarily intended to educate a reader about the abstract qualities of a design. They act like a template from which many different implementations can be created. In contrast framework clusters are specific entities which are expressed as source code and do not have an existence outwith the context of the framework. In some ways they can be thought of as instantiations of design patterns but without the restriction of having to be applicable in other contexts.

The distinction between design patterns and framework clusters is subtle but significant. It is this difference which makes design patterns unable to scale as documentation to describe an entire framework. Their descriptions of functionality and interaction can only approximate the behaviour of an actual system and there are many code structures within frameworks which do not map onto any recognised design patterns. Admittedly some of these structures might be from, as yet, unidentified design patterns but others seem more likely to exist only within the domain or implementation of the framework and therefore are not addressed by design patterns.

Lajoie and Keller also recognise the importance of understanding a framework through clusters of interacting classes. They argue that the term *“micro architecture”* be used to describe the specific structures found within object oriented frameworks and propose a combination of design patterns and contracts to document them. *“As frameworks codify design knowledge of a particular domain, micro architectures codify design knowledge in terms of the behavior of object collaborations”* (Lajoie and Keller 1994).

Their use of design patterns differs from Gamma’s as they use the documentation to record details from the specific implementation rather than an idealised exemplar. They also suggest that knowledge of micro architectures distinguishes experienced framework developers from novices. *“These structures, micro architectures, are of course known by the framework designers, but unfortunately by very few others. … the informed framework designer has a comprehensible, coarse-grain picture of the framework, whereas the novice framework user is overwhelmed with the many, seemingly unrelated framework classes”* (Lajoie and Keller 1994).

Use case maps are another form of documentation which suggests that clusters of interacting objects are significant. A use case map visually shows the path of a use case as it interacts with parts of a system. The elements of the system are shows as boxes but maps often include groups of objects which interact strongly together and are called teams. *“In use case maps, teams are used to group operationally related components, without committing to whether or not the teams themselves will have explicit existence in the implementation”* (Buhr 1996). Teams form significant parts of use case maps and appear to relate strongly to the idea of micro architectures within frameworks.

Other evidence for the existence of micro architectures can be found in the work of object oriented methodologists such as Booch and Meyer. Booch argues for the concepts of ‘mechanisms’ within object oriented design claiming that they arise from the object structure of an application. *“The object structure is important because it illustrates how different objects collaborate with one another through the patterns of interactions we call mechanisms.”* (Booch 1994, p21) He goes on to provide an example of such a mechanism from a graphical user interface. *“Consider the drawing mechanism commonly used in graphical user interfaces. Several objects must collaborate to present an image to a user: a window, a view, the model being viewed, and some client that knows when (but not how) to display the model. The client first tells the window to draw itself. Since it may encompass several sub-views, the window next tells each of its sub-views to draw itself. Each sub-view in turn tells its model to draw itself, ultimately resulting in an image shown to the user.”* (Booch 1994, p166) This description seems very similar to the types of interaction one might expect to find within a framework.

Meyer also describes a similar unit of composition called a cluster. *“A cluster is a group of related classes, or recursively, of related clusters.”* (Meyer 1997, p923). Meyer claims that, because of their scale, clusters play an important role in comprehension. *“The cluster is also the natural unit of single developer mastery: each cluster should be managed by one person, and one person should be able to understand all of it – whereas in a large development no one can understand all of a system or even a major subsystem.”* (Meyer 1997, p923).

The evidence from the software engineering literature suggests that clusters, from here on referred to as micro architectures, are composed of a small number of classes (or bits of classes) interacting to achieve some purpose within the framework. They appear to present a natural way to decompose a large software system for comprehension but the descriptions are sufficiently vague to make it difficult to know how to best document a micro architecture and also to be able to identify a set of them within a framework.

4.3.3 Documenting micro architectures

Design patterns are an attractive form of documentation because they combine information about the functionality and interaction of a micro architecture into one location. Design patterns typically represent structural and behaviour information via a combination of a textual narrative and UML diagrams. They also tend to include sections on the motivation for the pattern, examples and potential variations. Although design patterns were originally intended as a generic description of a micro architecture Lajoie’s work shows that patterns can be used to effectively describe details from a specific implementation. This also includes the identification of patterns which may not be expected to exist in multiple contexts (e.g.

Lajoie suggests the initialisation code of a framework might be a candidate pattern). From this it seems reasonable to conclude that design patterns should be considered to document the functionality and interactions within a framework.

An example of how these ideas might be realised for a framework can be seen in **Figure 24**. This describes a part of the JHotDraw framework which deals with the placement of connections upon a figure. Often in graphical applications situations will arise where a connection must be made to a figure. In such circumstances there are a number of decisions that can be made to determine the nature of the connection, its location and how it reacts to changes in the figure or its position. For example connections might be placed in a fixed position on a figure or be free to move around its edge, a figure might react differently to particular types of connection or may have regions each of which behaves in a different way when connected to. For each of these conditions the solution in JHotDraw is to encapsulate the alternative behaviours within a Connector class. A connector covers a rectangular area of a figure and whenever a connection is made to that area the figure delegates to the connector to determine the actual point of contact for the connection. The micro architecture documentation describes the motivation for this solution and shows the static and dynamic behaviour that occurs between a figure, a connector and a connection figure whenever a connection is made. The implementation section also describes potential areas of variation supported by this solution, including having multiple connectors for a figure or having different types of connector for fixed locations or movable ones. Where relevant definitions already exist within the framework their names are provided for ease of reference.

Despite their apparent suitability, in practice design patterns struggle to capture the dynamic information present in micro architectures. Polymorphism results in many alternative behaviours which are difficult to capture in documentation without recourse to separate dynamic traces for each combination (which would make the documentation hard to produce and difficult to understand). In the example shown this problem was minimised because the use of connectors is governed by a single protocol defined in the updateConnection method of the LineConnection class. The Connector class contains two hook methods findStart and findEnd which subclasses can override to alter how the connector calculates the endpoints of a connection. The use of a single protocol allows one trace to be presented to the reader while still being relevant to all potential implementations. In general however this pattern of behaviour does not repeat across the framework. In some cases an interface is implemented very differently by its subclasses and they share very little behaviour in common. In such situations multiple views have to be created to show the different behaviours that can occur.

Pattern: Connection Points

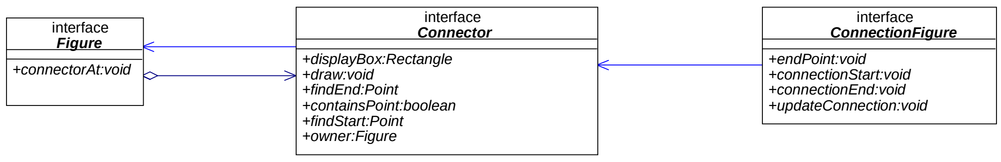
Problem

When connections are made between elements in a drawing the point at which the connection attaches to the element must be described. Describing this position requires an intimate knowledge about the geometry of the element, which the connection cannot have. A simple solution to this problem is to define a standard connection point for all elements (for example the centre point) but this proves insufficient for many applications. In addition the connection might require the connection point to change during the course of execution (for example when the element is moved) or perhaps the type of connection should determine its location. **Connection Points** allow precise positions on an element to be specified and to allow those positions to be changed during use.

Solution

Delegate the connection point to a connector object. This indirection allows the connection point to vary by changing which connector is used and allows precise location of the connection by creating connectors that understand the geometry of particular drawing elements. This is an implementation of the Strategy pattern.

Structure



Participants

A Figure represents an element on the drawing.

A Connector describes a connection point on a figure.

A ConnectionFigure models a relationship between figures.

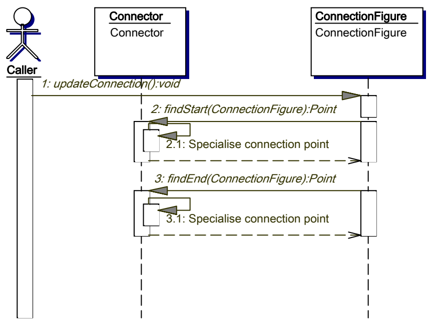
Collaborations

Initialisation:

Get the connector for the source figure of the connection. (Connectors are obtained by querying a Figure's ConnectorAt(int,int) method). Get the connector for the target figure of the connection. Complete the initialisation of ConnectionFigure by passing the source connector into the connectorStart(Connector) method and the target connector into the connectorEnd(Connector) method. This behaviour can be triggered by a mouse interaction involving a handle or a tool. The ConnectionHandle, ChangeConnectionHandle and the ConnectionTool all instigate this behaviour.

Connector protocol:

Whenever an updateConnection() request is made to a ConnectionFigure it must negotiate with its connectors to determine new start and end points. This behaviour is triggered when either one of the connected figures or the ConnectionFigure receives a move request.


Caller
Connector
Connector
ConnectionFigure
ConnectionFigure
1: updateConnection():void
2: findStart(ConnectionFigure):Point
2.1: Specialise connection point
3: findEnd(ConnectionFigure):Point
3.1: Specialise connection point

Implementation

Points of variation:

By default all existing figures in the framework contain a single type of connector. It would be possible to modify the behaviour of connectorAt(int,int) with the use of containsPoint(int,int) to return different connectors for different areas of a figure.

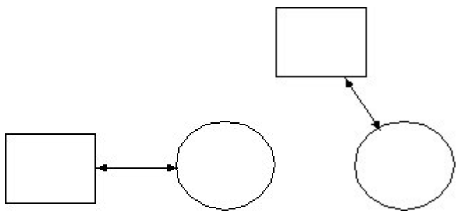
Another, more obvious, modification is to customise the algorithm used by a connector to calculate the connection point. This can be done by overriding the behaviour in the findStart() and findEnd() methods of connector or creating an alternative mechanism to replace them (See locate() in LocatorConnector). The connector hierarchy contains specialisations of connector for many common types of connection. ChopBoxConnector and its descendants define connection points around the perimeter of specific shapes. LocatorConnector defines a connector which can be repositioned on a figure during execution and ShortestDistanceConnector attempts to minimise the length of ConnectionFigure used to make a connection.

Default settings:

By default all descendants of AbstractFigure use the ChopBoxConnector. Exceptions are EllipseFigure, PolyLineFigure and PolygonFigure, which use the ChopEllipseConnector, PolylineConnector and ChopPolygonConnector respectively. Another exception is RoundRectangleFigure, which uses the ShortestDistanceConnector.

Example

To illustrate the role that connectors play in a drawing consider this example from JavaDraw.



The connection between the rectangle and the ellipse touches both elements on their perimeter. If either shape moves the connection between them is updated and the connectors are used to calculate the new end points. This example demonstrates the use of the ChopBoxConnector on the rectangle and the ChopEllipseConnector on the ellipse.

Figure 24: A candidate micro architecture notation

Another problem that arises from dynamic behaviour is capturing the frameworks reaction to changes in state. Often such changes result in alterations to the sequence of interactions within the framework. This is most obvious during initialisation of the framework where a different sequence of behaviour might be carried out from that which occurs during normal execution. This variation not only requires a separate diagram to be produced but it may also affect the static view of the architecture because methods might only be used at start up or in a particular state and these will have to be included in the class diagram.

Attempting to provide coverage of design patterns across an entire framework also presents problems. Deciding how and where to subdivide the system into a series of micro architectures is a difficult decision. This is more challenging that it might at first seem because there is little support from the framework source code to signify where conceptual boundaries might be drawn. Take the connection points example mentioned earlier. Arguably part of its attraction is the cohesive description of a small piece of the larger framework. In helps people to understand that connections between figures can be modelled in a number of ways and identifies the relevant parts of the framework that are involved. It also doesn't include any irrelevant material. There is no discussion of how connections are redrawn, what constitutes a figure or even the other roles that a connection figure plays within the framework. This is significant because it is this reduction which lends the example a simplicity which suits its illustrative purpose.

Deciding what information one ought to leave out of a micro architecture appears to require significant experience of both the framework implementation and its domain. Only with this knowledge can a developer make decisions about where to draw boundaries between concepts to create micro architectures. A subjective, opinionated, approach is liable to leave gaps in the documentation damaging its ability to support reuse. It is also liable to make the technique far more expensive and difficult to produce further reducing its appeal.

4.3.4 A micro architecture notation

If dividing the system by functionality is difficult to reliably achieve then perhaps another way of decomposing the system would be more effective? One of the most obvious qualities of object oriented frameworks is that some classes within the framework appear to be more important than others. More specifically some classes within the framework model domain abstractions while others only provide implementation detail. It could be argued that this makes the domain abstractions more important to understand as they represent parts of the framework's design. For example the Figure interface is important to JHotDraw and is widely used throughout the framework. Understanding something about Figures is vital for any use of JHotDraw. In contrast understanding any specific implementation of the Figure interface reveals far less about the behaviour of the entire system. This observation that some classes are more important than others appears to be supported by Meyer who argues that there are three types of classes in a system. Analysis classes which model abstractions within the domain, design classes, which model architectural structures within the application (i.e. design patterns often fall into this category) and implementation classes which provide detail to the application (Meyer 1997). Perhaps this information could be exploited to decompose a framework for documentation?

An alternative to a functional decomposition could focus on the key domain abstractions present within the framework, drawing them out from the background noise of subclasses and utility classes and making them easier to understand. This approach, although not strictly creating micro architectures, simplifies the decomposition of the framework as domain abstractions are comparatively easy to spot. They tend to be named in the language of the domain so anyone with some familiarity with domain vocabulary should be able to identify candidate classes. They also tend to be abstract classes or interfaces and can often be found at the top of inheritance hierarchies within the framework. The down side to this approach is that it merges information about different aspects of functionality showing how they intersect an abstraction within the framework. This trades the ease of following micro architecture specific information across a framework for a better understanding of each individual class' role within the system. Although not explicitly describing micro architectures this decomposition can still help develop an understanding of them as the interactions surrounding important abstractions contain subsets of micro architecture information. Understanding the system in terms of its key abstractions should therefore assist re-users to learn framework micro architectures as they provide insight into some of the most important framework behaviour. Decomposing the system into key abstractions also has the advantage that the documentation provides coverage for the breadth of functionality and interactions supported by the framework.

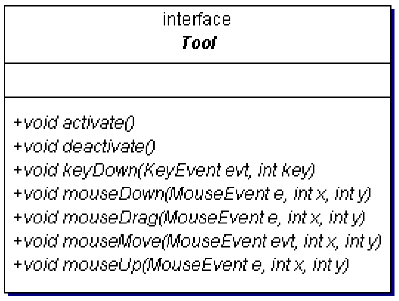

interface
Tool
+void activate()
+void deactivate()
+void keyDown(KeyEvent evt, int key)
+void mouseDown(MouseEvent e, int x, int y)
+void mouseDrag(MouseEvent e, int x, int y)
+void mouseMove(MouseEvent evt, int x, int y)
+void mouseUp(MouseEvent e, int x, int y)

Figure 25: A micro architecture interface view

The notation used by this study to describe micro architectures comprises three views; an interface view, a call graph view and a hierarchy view. The interface view (**Figure 25**) displays the methods available for a particular interface within the framework. The call graph view (**Figure 26**) shows the call sequences within the framework that result in a call to a method from a given interface. The hierarchy view (**Figure 27**) shows which classes within the framework implement the interface and which methods they implement or inherit from other classes.

These separate views are joined together by an index (**Figure 28**) which lists all of the available interfaces in the framework. The technique also makes use of the framework source code to provide descriptions of the functionality presented by callers or implementations of an interface. Links between the views allow navigation from one source of information to another. The index links to the interface descriptions and each method on an interface then links to its own call graph. The class hierarchy for each interface can be accessed separately from the main index. Accessing the framework source code is performed manually (the documentation provides the class and method names to look for).

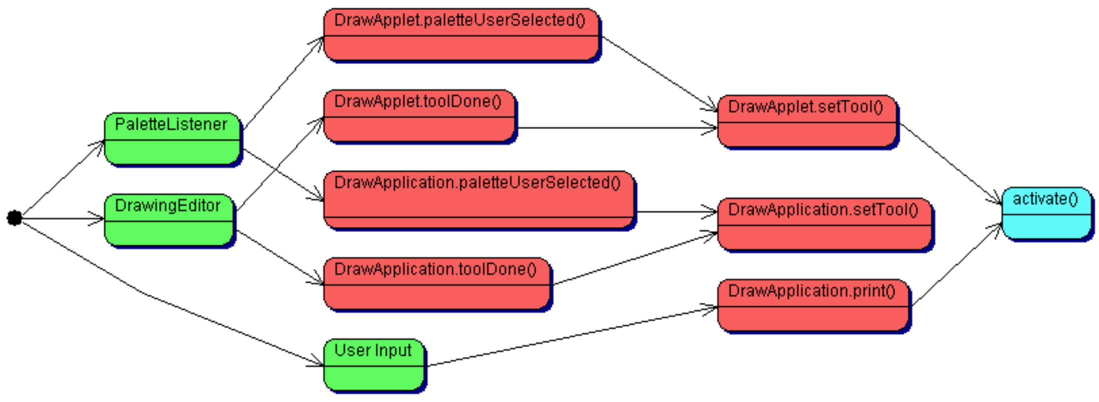

DrawApplet.paletteUserSelected()
DrawApplet.toolDone()
PaletteListener
DrawingEditor
DrawApplication.paletteUserSelected()
DrawApplication.toolDone()
User Input
DrawApplet.setTool()
DrawApplication.setTool()
DrawApplication.print()
activate()

Figure 26: The Tool.activate call graph view

The interface and hierarchy views use standard UML notation to convey their information. The call graph view is different and uses a customised activity diagram to represent the call sequence leading up to the framework interface. Each graph is made up of three parts, a set of inputs that begin the call sequence, a set of intermediate methods and a call to an interface method. These three types are represented on the graph using nodes of different colours, green for input, red for intermediate methods and blue for interface methods, arrows indicate a method invocation and each node describes the concrete type and method that is invoked by the call. The call graph inputs are constrained to describe user input events, calls from a Java library or calls from another framework interface. This helps to modularise the call graphs and prevents repetition of commonly occurring interactions. Users wishing to follow calls further back in the call sequence can look up the micro architecture for the named interface and continue to follow the call from there. (The complete set of micro architecture diagrams is available in Appendix B, 19 interfaces, 19 hierarchies and 171 call graphs)

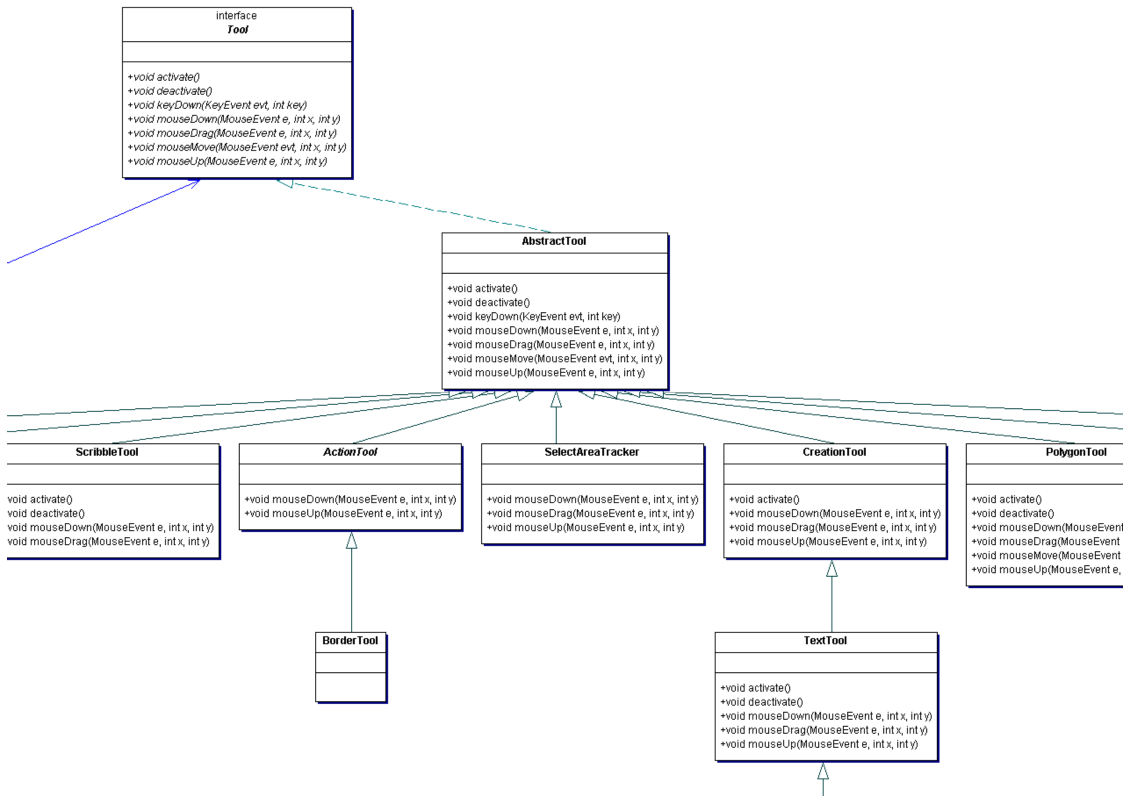

interface
Tool
+void activate()
+void deactivate()
+void keyDown(KeyEvent evt, int key)
+void mouseDown(MouseEvent e, int x, int y)
+void mouseDrag(MouseEvent e, int x, int y)
+void mouseMove(MouseEvent evt, int x, int y)
+void mouseUp(MouseEvent e, int x, int y)
AbstractTool
+void activate()
+void deactivate()
+void keyDown(KeyEvent evt, int key)
+void mouseDown(MouseEvent e, int x, int y)
+void mouseDrag(MouseEvent e, int x, int y)
+void mouseMove(MouseEvent evt, int x, int y)
+void mouseUp(MouseEvent e, int x, int y)
ScribbleTool
void activate()
void deactivate()
void mouseDown(MouseEvent e, int x, int y)
void mouseDrag(MouseEvent e, int x, int y)
ActionTool
+void mouseDown(MouseEvent e, int x, int y)
+void mouseUp(MouseEvent e, int x, int y)
SelectAreaTracker
+void mouseDown(MouseEvent e, int x, int y)
+void mouseDrag(MouseEvent e, int x, int y)
+void mouseUp(MouseEvent e, int x, int y)
CreationTool
+void activate()
+void mouseDown(MouseEvent e, int x, int y)
+void mouseDrag(MouseEvent e, int x, int y)
+void mouseUp(MouseEvent e, int x, int y)
PolygonTool
+void activate()
+void deactivate()
+void mouseDown(MouseEvent
+void mouseDrag(MouseEvent
+void mouseMove(MouseEvent
+void mouseUp(MouseEvent e,
BorderTool
TextTool
+void activate()
+void deactivate()
+void mouseDown(MouseEvent e, int x, int y)
+void mouseDrag(MouseEvent e, int x, int y)
+void mouseUp(MouseEvent e, int x, int y)

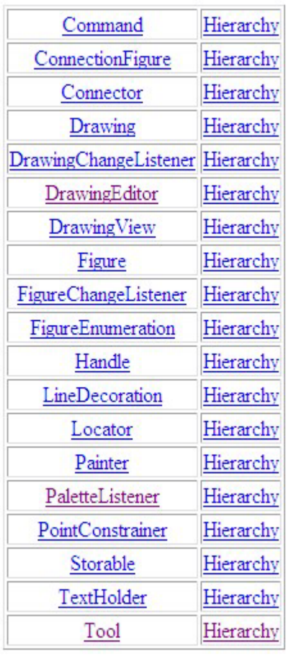
Figure 27: A section of the Tool micro architecture hierarchy view

4.3.5 An example of the micro architecture documentation

The following example illustrates how the resulting documentation could be used to investigate the Tool interface in JHotDraw. This illustration assumes that a framework re-user has read about Tool in the pattern language or has seen it in the list of micro architectures and wants to know more about how it is used and what it does.

The user selects Tool from the list of framework interfaces (**Figure 28**). This displays the interface for a Tool and allows the re-user to select individual methods from the interface to understand how they are used by the framework (**Figure 25**). The user decides to look at the activate method because it appears relevant to the initialisation of the Tool. Clicking on the activate method displays its call graph (**Figure 26**). From this graph it can be seen that two framework interfaces and one user input make use of the activate method. Focusing on the interfaces it can be seen that activate is invoked either from a call to PaletteListener.paletteUserSelected or by DrawingEditor.toolDone. The user can also see that these two interfaces are implemented by two classes DrawApplet and DrawApplication. Of these DrawApplication is the most interesting (because the user is creating an application). From both calling interfaces (PaletteListener and DrawingEditor) it can be seen that the code eventually calls DrawApplication.setTool and that this method is responsible for calling activate. This information presents the user with two choices: They may wish to understand more about the PaletteListener and DrawingEditor interfaces and the conditions which will result in them producing a call to activate or they may wish to drill down into the implementation of DrawApplication.setTool to understand more about the functionality that occurs during the initialisation of a tool.

If the other interfaces are consulted (by looking at the corresponding micro architectures) the user will discover that activate is called as the result of three conditions; either a tool button being selected on the user interface, a drawing being saved or loaded into the editor or from the action of another tool. If on the other hand they decide to read about the functionality of setTool they will discover that it first deactivates the current tool before activating the new tool and updating the status bar (a text field at the bottom of the application) to reflect the new tool name. Having understood something about the behaviour leading up to a call to activate the re-user may still be interested in understanding what behaviour activate actually performs.



Command	Hierarchy
ConnectionFigure	Hierarchy
Connector	Hierarchy
Drawing	Hierarchy
DrawingChangeListener	Hierarchy
DrawingEditor	Hierarchy
DrawingView	Hierarchy
Figure	Hierarchy
FigureChangeListener	Hierarchy
FigureEnumeration	Hierarchy
Handle	Hierarchy
LineDecoration	Hierarchy
Locator	Hierarchy
Painter	Hierarchy
PaletteListener	Hierarchy
PointConstrainer	Hierarchy
Storable	Hierarchy
TextHolder	Hierarchy
Tool	Hierarchy

Figure 28: The micro architecture index

Tool, being an interface, has no implementation for the activate method but the existing Tool implementations within the framework give some idea of the type of behaviour that activate is expected to perform. To support the re-user in searching through these options the micro architecture documentation provides a hierarchy view (**Figure 27**) which displays the interface and all existing implementations within the framework. From this view the implementations of Tool which define an activate method can be identified (six out of the thirteen implementations define an activate method the remainder reuse an implementation through inheritance).

All the implementations inherit from a common parent AbstractTool which seems to provide a default implementation of a Tool. The re-user may decide to look at this implementation first because it is widely used by other implementations but may also decide to look at some of the other implementations to see how they differ from the default behaviour. The implementation of activate in AbstractTool clears the current selection within the drawing view. In the other implementations activate is often defined to reinitialise the state of the tool back to some default value (for example the scribble tool sets the figure it creates to null). This information allows the user to understand how to initialise a tool within the framework. It identifies how existing parts of the framework use the activate method during initialisation and explains how existing implementations define activate to clear the selection and reset the tool to a default state ready for input.

The micro architecture documentation can thus support the developers comprehension of the activate method within the Tool interface. The other methods of the tool interface can be understood in a similar way allowing the developer to reuse the concept of a Tool in their own applications and to take advantage of existing code through inheritance when implementing their own Tool class.

4.4. Conclusions

This chapter has described the novel extension of two existing forms of framework documentation designed to address the four problem categories of reuse. The second iteration of the pattern language is expected to improve upon the performance of the first. Specifically it is hoped that the increase in size and detail of the language will help it to better address mapping problems during reuse. It is also anticipated that the language’s implicit guidance about how to make modifications to the framework should help to encourage architectural conformity and ensure that modifications do not damage the existing non-functional qualities of the framework. The micro architecture descriptions are intended to provide support for the interaction and functionality problems that occur during framework reuse. Interaction problems are supported by showing the interaction context of framework interfaces. Functionality problems are addressed by identifying classes that call framework interfaces and classes which implement interface methods. This provides a starting point from which to explore the framework source code. The proposed documentation, a combination of the pattern language and micro architecture techniques, is intended to provide effective support for a broad range of framework reuse issues. These claims need to be tested and the following chapter will present a thorough empirical evaluation of both forms of documentation to determine their utility.

5 Evaluating framework documentation

5.1. Introduction

The combination of pattern language and micro architecture documentation described in the previous chapter offers support for the four categories of framework reuse problem. The pattern language with its structural decomposition of the framework as a set of patterns should help developers to identify what elements exist for modification or reuse. In addition its use of examples and hierarchies should enable developers to see how solutions can be implemented and to assess what options are available for a particular role. These features should make the pattern language well suited to supporting the mapping and architectural problem categories. The micro architectures, on the other hand, focus on the key interfaces of the framework and provide a mechanism to trace sequences of interactions back through the existing framework code. This should help developers to identify what functionality is available and also to develop an understanding of the interactions that exist between the classes of the framework. Together this combination suggests support for all of the identified problem categories, which, if it were true, would have significant implications for the documentation of object-oriented frameworks. This study evaluates these claims through a user trial of the pattern language and micro-architecture documentation. Developers were supplied with the documentation and a suitable reuse task and were asked to produce a solution relying only on the documentation and the framework source code for guidance. In order to develop an understanding of how documentation was used during the task the process was observed and documentation accesses and developer actions recorded for later analysis.

The primary motivation for this investigation is to understand and collect evidence of how the pattern language and micro-architecture documentation perform when addressing framework reuse problems. This enables weaknesses to be identified and may suggest modifications to make the documentation more effective. There are other reasons why this investigation is merited. One is the comparative lack of similar studies within the framework literature which currently appears to champion the creation and ad-vocation of documentation over its evaluation. This failure to evaluate has limited our ability to distinguish between effective and ineffective forms of documentation and also limits our ability to identify what characteristics of documentation have effects on its usefulness. This study will demonstrate that evaluations are both practical and useful to perform, hopefully encouraging other researchers to follow a similar approach in the future. It also provides an opportunity to validate and possibly refine the framework reuse problem categories identified earlier in this thesis.

5.2. Experimental design

The goal of this study, to make detailed observations about a process, suggests that a qualitative approach to data analysis will be useful. Qualitative analysis categorises patterns of behaviour within a task and attributes meaning and effects onto these. This allows the cause of events to be traced to their origins, helping to identify how documentation (alongside other possible sources of information) have been used to affect the solution. This kind of detailed insight into what happened during the process would not be available in a quantitative analysis, which abstracts the process under investigation into a number of discrete quantities and draws inferences from the magnitude of those variables.

The user of the documentation is as important to this evaluation as the documentation itself. Of course the documentation's structure and content is an essential factor in deciding what information is available during reuse but it is the reader of the documentation that chooses what pages to access and to a certain extent they also control what knowledge to deduce from the documentation. Therefore the re-user's perception of the problem, their thoughts on a solution and their selection of what documentation to read are critically important to the evaluation of that documentation.

5.2.1 Data capture

The data collected by this study consists of the documentation accessed during the task and the developer's plans and actions in response to that documentation. Documentation accesses are directly observable and therefore easy to capture. However, gathering information about the developer's thought process is more difficult. This study used a talk aloud protocol to obtain this insight. Participants were required to describe their thoughts out loud as they worked on the task. The data produced was in two forms, audio for the talk aloud protocol and video to capture the documentation that the user had on screen during reuse. Fortunately, because the documentation was available online, a natural way to capture the data was to use screen and audio capture software (Netu2 2005). This could reside as a background task on the developer's machine and with the addition of an external microphone would provide an accurate recording of all the spoken and visual activity that occurred during the trial.

5.2.2 Plan of analysis

There is no single approach to qualitative analysis recommended by the literature. Instead each study has to make its own decisions about how to process its data to identify relevant patterns and relationships between elements. Despite the lack of process there is common agreement on what the key activities to perform are. These include transcribing the data into a textual format, clustering data into categories, using visualisation techniques to draw out patterns between categories, and paying particular attention to the differences between participants rather than the similarities (Miles and Huberman 1994), (Dey 1993) and (Judd et al 1991).

Data was transcribed for each participant into a textual narrative that describes their reuse attempt. This makes analysis easier and helps to preserve the anonymity of the participant. The narratives were then read to identify what solutions had been proposed for each section of the task. These were then considered for their completeness and the quality of solution that was achieved. The analysis then considered how those solutions arose by identifying critical documentation accesses and categorising them with respect to the type of problem addressed and whether the documentation provided useful information. This information was explored further for significant patterns with which to characterise the documentation use and allow the study to make specific claims about the utility of both forms of documentation.

5.2.3 Experimental subjects

The study solicited volunteers from the Computer and Information Sciences department at the University of Strathclyde in Glasgow. Requests were made to the postgraduate students, three final year undergraduate students (who were using JHotDraw in their final year projects) and to the teaching staff of Strathclyde’s Software Architecture class (Roper and Wood 2004), which teaches framework development using JHotDraw.

Individuals were invited to participate in the experiment via email. The call outlined the details of the study, but did not describe specifics about the task to prevent any preconditioning about possible solutions. It went on to describe what work would be expected from the participants and the methods that would be used to monitor them. It also explained their ethical rights and how to sign up for participation. Participation in the study was voluntary and participants were assured of their anonymity and of their right to stop the study at any time (The call for participation is available in Appendix C).

The intention in targeting these different sources for volunteers was to obtain a range of experience of the JHotDraw framework. The undergraduate students all represented experienced developers with the framework as they had each spent the past six months using it to achieve sizable individual projects. The postgraduates at Strathclyde represented a wider spectrum of experience, as some of them had no experience of the framework at all while others had been students at Strathclyde and had used JHotDraw in the Software Architecture class. Finally the Software Architecture lecturers had a good familiarity of the framework domain and its design but did not have a lot of experience developing solutions with it. By targeting this mixture of people, it was hoped that the study could attract some that had a lot of experience with the framework, some which had moderate experience and some that had never used it at all. It was felt that observing people with a range of different experience levels of the framework would be of value, as they are likely to use documentation differently.

From a postgraduate population of around thirty students only three agreed to participate in the study. From the undergraduates approached two agreed to take part and both of the staff involved in the Software Architecture class also agreed to participate. This produced a total of seven participants for the evaluation. With hindsight the poor participation from the postgraduates might be explained because the duration and nature of the task was a maximum of three hours performing software development activities. This might have been off-putting for a lot of potential participants and may also have been considered too much effort for no reward. The seven individuals that did agree to take part (from here on referred to as participants A through G) covered the range of experience that was sought and produced such a large amount of data that it would probably have been infeasible to work with larger numbers.

5.2.4 Reuse task

The task that was chosen for this study was to recreate a model of a Blocks World in JHotDraw (**Figure 29**). Taken from the artificial intelligence community (e.g. Slaney and Thiébaux 1994) a Blocks World is a simple abstraction of the geometric problem of positioning blocks on a ground. The task comprised of four subtasks listed below. To keep the application simple the functionality of removing a block was not considered (the complete task description can be found in Appendix C).

- A representation had to be selected for the ground and the ground had to be created in the application at start up.

- A representation also had to be developed for a block, which had to be a given size and coloured red.
- A mechanism for adding blocks to the world had to be created.
- The blocks had to be constrained to only exist on the ground or on top of another block (i.e. they could be stacked into towers). Blocks could be moved only if they did not have another block on top of them.

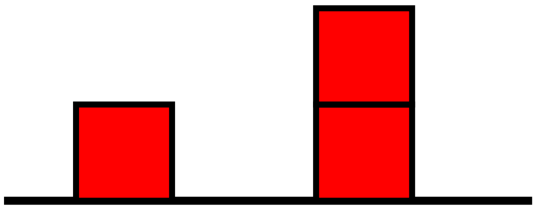


Figure 29: An example Blocks World

The selection of a Blocks World application satisfied two goals. Firstly it was important to the study that the task fits well within the domain of the framework as this ensures that the framework will support the modification. Secondly, it was important that the task was clear and simple enough for participants to understand and for them to be able to produce solutions within the time constraints of the experiment. This had to be balanced against the desire to exercise as much of the framework as possible and to as realistic a level of detail as possible to better mimic real application development. The participants were not required to code their solutions but instead asked to articulate them verbally (coding was considered to be too time consuming, and there was a concern that less important implementation issues would become dominant). One participant (A) did decide to code a solution and took significantly longer than three hours to complete the task.

In addition to the above task participants would also be asked to take part in two interviews (one before and one after the task) to capture their background within this domain and their reactions to the documentation. They would also be given a period prior to the study (no longer than 30 minutes) where they could familiarise themselves with the experimental documentation. Altogether the participants would be given a maximum of three hours to perform the task and the related activities.

5.3. Threats to validity

This section considers the major threats to the results of this study. Internal threats are those which compromise the findings for this study while external threats compromise the ability of findings from this study to generalise to the wider framework population. In each case the threat to validity is described and the steps taken to limit this effect are explained.

5.3.1 Internal threats

- **Unfamiliar documentation:** There is a risk that developers will shy away from the new forms of documentation because they are less familiar and unproven compared to other more conventional forms. To limit this developers were provided with a short tutorial on the micro-architecture notation and a period at the start of the study to familiarise themselves with the new documentation (The tutorial is available in appendix C). Also all other forms of documentation, except the source code, were removed to encourage the use of the new techniques.
- **Selective coverage of the framework:** The same task was used for each participant under observation. This enabled a fair comparison of the relative performance of each developer but it meant that only specific parts of the framework were being exercised by the task. This might have affected the performance of documentation if it were particularly suited, or not, to those areas. To address this, the task was created to cover a wide range of framework behaviour and to be as realistic as possible to properly exercise the available documentation.
- **Talk aloud intrusion:** The use of a talk aloud protocol can have negative effects on the participant in a study. It can affect the individual's concentration and might make them feel embarrassed or awkward during the task. To prevent this the studies were performed in private with only the researcher and the participant present. The researcher could then act as a focus for the participant's speech and they could also prompt for thoughts if the developer fell silent.

5.3.2 External threats

- **Selection effect:** Participants in the study have volunteered which might somehow distinguish them from other potential framework re-users who would not volunteer. There is little that can be done to control this, as it would be impractical and also unethical to perform a similar study without the user's consent.
- **Using a single framework:** All of the reuse activity occurs using one framework so it is not possible to say which problems are particular to the framework and which are applicable to frameworks in general. It would have been impractical to perform this evaluation with more than one framework. The amount of time required to create comparable documentation for them would have been prohibitive and the

contribution provided by each participant would have had to be unrealistically increased to provide enough time for multiple reuse tasks.

- **Choice of participants:** The participants were all drawn from an academic background. This might limit the generalisation of the results because they might differ in their approach and motivation from industrial framework re-users. It is difficult to gain access to real world developers and academic subjects should provide a reasonable approximation to the real thing. This is especially so in this case as only final year students, post graduate students or members of staff were solicited. Also, participants were frequently reminded that they could leave the task at any point hopefully ensuring that only those interested and suitably motivated continued to take part.
- **Choice of task:** The task chosen for the reuse task has been shaped in part by the need to cover a wide range of features in the framework and to be achievable within a strict time constraint. This inevitably means that it is smaller and somewhat artificial in its requirements when compared to genuine reuse tasks. Care has been taken to ensure that the tasks set mimic those found in real life and are not designed to fit exactly on top of the existing features of the framework forcing the developers to make modifications to the framework to achieve the task.

5.4. Data

In total twenty-one hours of video and audio data were collected from the seven studies. The participants varied in the amount of time they spent on the task: some completed the task before the three hours were up, others asked to continue past the time limit. The times for each participant are available in Appendix C. The amount of data was potentially overwhelming because almost anything within it could become important evidence in the study. This section describes how the volume of material was managed and distilled, through several stages, to produce the evidence that characterises the documentation’s use.

5.4.1 Transcription

The first step of analysis was to transcribe the data into a textual account for each participant. This was important because text is much easier to analyse than video. Skipping back and forth to compare ideas is a matter of turning a page rather than rewinding a tape. It is also much faster to read a textual account of what has happened than to listen to it occurring on tape in real time; this is especially important when sections have to be revisited many times to make comparisons or to retrieve information. Another practical difference between the two media is that text can be printed allowing the researcher to work away from the machine increasing the amount of time available for analysis.

Time	Documentation Accessed	Talk aloud comments	Non verbal observations
31	PL Overview PL Identifying existing figures PL Figure hierarchy	First thing that I'm thinking about is representing the ground. I'm guessing it will be some sort of figure. I'm going to look in the pattern language. It's annoying me it's too big (laugh). **R: What are you searching for?**	Scanning Figure hierarchy Scrolling around
32		I'm thinking about either using a line figure or just using some sort of rectangle. Not entirely sure… don't know if that is a reasonable use of line figure or if they are meant to exist as part of more complex figures.	
33		Line figure exists by itself… Only thing that bothers me is that line figure extends poly line figure, doesn't seem like a proper use of it because its not poly!	Maybe this is why sub is worried about line figure?
34	PL Identify existing figures	I reckon that a line figure would be okay. Aesthetically it would look, hmmm concerned about thin line try to alter its attributes to make it a big thick line.	
35	PL Overview PL Modifying existing figures	I want to modify a line Now I'm frustrated I've been to figure hierarchy… I feel that I've hit a dead end now. My reaction previously would have been to look at the source code, I'll have a look in the micro architectures but I don't feel that its going to tell me what I want to know.	Mod figure links back to figure hierarchy in PL.

Table 6: Excerpt from a transcript

The transcription had to capture an accurate account of both the talk aloud protocol and the documentation used during the task. To achieve this each transcription was performed in a grid with columns for different types of information to be recorded (**Table 6**). Information was captured on time, documentation accessed, talk aloud comments, and non-verbal observations. The time column recorded the time, in minutes, from the start of a task that an event had occurred. This provided a way to reference data when used as evidence later in the report. The documentation accessed column recorded what documents, and if applicable what pages, were accessed during the task. This column also allowed the sequence of accesses and duration of each access to be considered. The talk aloud protocol column captured the thoughts and reactions of the developer during the task. It was important that this information be recorded verbatim as later analysis would focus on the meaning of each sentence and even small errors in transcription could have a large effect on the semantics of that sentence. Finally, the observations column was used to capture any interesting non-verbal actions that had occurred during the task. For example, if a user gestured using the mouse to an item in the documentation then this would be recorded here.

The microphone used to record the participants talk aloud monologues did not perform very well capturing faint and at times distorted audio. This made some areas of the tapes difficult to understand and required more effort to transcribe. The microphone had offered reasonable quality during practise tests, but during the task its quality was poor because the participants often spoke quietly or turned to address their comments to the researcher who was sitting opposite the microphone (behind and to one side of the participant). In addition traffic noise from the street below and the sound of people passing in the corridor occasionally drowned out the speaker's voice making short sections of the tape unintelligible. The poor quality of the tape made transcription difficult; sections of the tape had to be replayed multiple times until an accurate account could be produced and some words and phrases on the tape could not be deciphered at all.

Transcription was surprisingly difficult to perform. It was difficult to transcribe in real time and as a result the tape had to be stopped frequently in order to allow the previous few moments of audio to be written down before listening to the next segment and pausing again. The frequent stops and starts added significantly to the transcription time and it soon became apparent that, with each tape lasting approximately three hours, it would take too long to transcribe everything. Instead the decision was taken to only focus on the sections of each tape that contained the participants working on the task. This meant that their pre- and post-task interviews and documentation orientation data were not used in this study which reduced the amount of data to be analysed by around a third to a still considerable but critical fourteen hours of data.

It was difficult to decide how to transcribe some portions of the tapes. Spoken text conveys a lot of information through the tone of the speaker's voice and the timing with which it is delivered. This information is non verbal and difficult to capture using punctuation alone. To capture this in the transcription any passages of tape that displayed a strong emotion were tagged with a brief description either beside the affected comment, or in the non-verbal observations column, allowing at least some sense of the speaker's emotions to be preserved. Timing between words and phrases in the monologue was represented by padding the written text with white space and ellipsis to give the reader a sense of the length of each pause. Together these additions along with the other non-verbal observations helped to flavour the transcription with those characteristics that did not transfer well into text.

In total the transcription produced almost two hundred pages of text and took a period of around five months to complete. The complete set of transcripts can be found in Appendix C. The amount of effort and time that was required to transcribe the data was staggering and something that the study had seriously underestimated. On the other hand, transcription had enabled the researcher to become familiar with the information on the tapes, and start to recognise patterns of behaviour across different participants, which was not only encouraging but also of significant benefit in later stages of analysis.

5.4.2 Blocks World solutions

In order to assess the performance of each developer their solutions to the task requirements were extracted from the transcriptions. The solutions were often difficult to identify as they were expressed in the talk aloud protocol and spread throughout the transcription. In order to make the assessment easier the transcripts were searched and relevant material separated into a summary table (**Table 7** shows a representative example) that described the details of each particular solution. By identifying these solutions the performance of each participant could be recorded and some general comparisons and observations made about them. (The numbering in the solution column refers to the order in which tasks were carried out)

The transcripts were searched for solutions using the requirements of the task. The Blocks World problem consisted of four main requirements: creating the ground, creating a block, creating a mechanism to add blocks and enforcing the constraints between the ground and the blocks; each of these could be further decomposed into a number of sub categories that had to be addressed. Identifying what solutions had been provided for each of these problems enabled the performance of each developer to be understood and compared to the others performing the task. To facilitate these comparisons the details of each solution were augmented with information about the forms of documentation that were most influential to its development (captured in the Critical Insight column), the amount of interference caused by the observer who sometimes prompted developers when they were stuck (captured in the researcher interference column – R.I.) and the duration of time spent working on that task (Duration). This information was collected for each of the sub-tasks performed by a developer and was compiled together into the summary table.

Requirement	*Solution*	Critical Insight	R.I.	Duration
Represent the Ground	1 Using Line figure	Prev Knowledge + Pattern Language	0	4min
Set size **Set position** **Set colour**	2 Thickness: Display box (wouldn't really work) 5 Position: using basic display box	Prev Knowledge + Source code Prev Knowledge	0 0	11min 1min
How to make ground appear	3 Don't want a tool 4 Create drawing and drawing add	Prev Knowledge Pattern Language + Micro architecture	0 1	2min 29min
Adjust position on resize	6 Involves the interaction between DV and F (cant get this) 7 Feels should be a listener (stop – potential. solution suggested)	? Prev Knowledge	? 0	69min 4min
Prevent size colour pos.	<Not done>			
Represent the block	9 Using Rectangle figure (or subclass)	Prev Knowledge + Pattern Language	0	1min
Set size **Set colour**	10 Colour: using set attribute 11 Size: using constructor (Researcher inference crucial)	Prev Knowledge + Source code Prev Knowledge + Source code	0 1	2min 12min
Add blocks	8 Use creation tool	Prev Knowledge	0	1min
Prevent size and colour	12 Turning off handles 13 Using Null handles 14 Creation tool, turn off drag and change mouse down	Prev Knowledge + Pattern Language Pattern Language + Source code Micro architecture + Source code	1 0 1	10min 10min 13min
Add in valid position	15 Use CT mouse down to place blocks 16 Use display box to make the transition	Prev Knowledge Prev Knowledge	0 0	20min 3mins
Constrain valid position	17 Check done within CT – sub wants to use isEmpty and mouse up (only thinking about block / ground) 18 Getting ref between Tool and Status bar (incomplete)	Prev Knowledge + Pattern Language	0 3	15min 38min
Only move top block	Not done – developer ran out of time			

Table 7: A developer's solution table

Compiling the summary table was straightforward. It was generally easy to identify areas in the text where developers were working on a particular requirement and to locate the solution they had proposed. In a few cases it was not so straightforward. Sometimes the developer had not realised that there was a problem to be solved and had therefore never addressed it; these were recorded as ‘Not Done’ in the table. In other cases the problems addressed were of a finer granularity than those proposed and so evidence had to be collected from several areas of the transcript to create a composite understanding of the proposed solution, and on other occasions solutions were sometimes altered by a developer later in the task, as they gained a better understanding of the problem or realised that a previous solution was not good enough. This required the data in the table to be continually updated to reflect the latest solution proposed. Despite these difficulties tables were produced for each participant and the conclusions drawn from them are presented in the analysis section below. The complete set of tables can be found in Appendix C.

5.4.3 Documentation accesses vs. reuse problems

By focusing on documentation accesses within the transcripts it was possible to understand what problems participants faced and whether the documentation helped them to answer those problems. An overview was prepared of each participant’s documentation accesses. This presented a summary, in the form of a matrix, of what accesses had occurred and whether they had successfully resolved a given problem. The matrix recorded documentation types along one axis, and problem categories along the other (**Table 8**). This allowed it to relate the types of problems experienced by a participant to the types of accesses made trying to resolve them and to consider how successful such attempts were (helpful accesses were recorded in the light grey column and un helpful accesses in the dark grey column). Accesses were recorded as a letter representing the participant and the time when the access occurred.

The study was also interested in how the documentation had been used and what the developers thought about it. The text surrounding each documentation access was studied in detail and comparisons were made across accesses to the same documentation. This captured a variety of information about the documentation: some comments described how developers felt about the documentation, while others showed the kind of information that it did or did not provide. Together these insights provided a detailed description of the support provided by documentation, where it had failed to support reuse and how the developers felt about using the different types of documentation.

The identification of documentation accesses was straightforward as they were separated during transcription. However, while reading through the text it became apparent that developers' existing knowledge was also playing a significant role in shaping the solution. This knowledge was a mixture of past experience of the framework and more general experiences, about algorithms, design patterns and language idioms. This information had originally been overlooked but having been recognised it was felt important to include it in the analysis. This required transcripts to be reprocessed to identify incidents of previous experience, in effect treating it as if it were another form of documentation.

	Pattern Language		**Micro Architecture**		**Source code**		**Previous knowledge**		**Other**	
Mapping	E101 E102 E103 E114 E136	E22 E115 E135 E137 E140	E20				E17 E42 E44 E45 E53 E100 E105 E112 E129 E132 E146 E153	E21 E55 E141		
Interaction	E114		E24 E27 E47		E117		E140			
Function			E25 E45 E55 E106 E107 E132 E145 E148	E22 E115 E119 E125 E148	E21 E34* E34 E38 E48 E152	E26 E28 E35 E37		E43 E132		
Architecture								E30		
Other										

Table 8: A problem vs. documentation matrix

Identifying documentation accesses was quite straightforward but relating them to problems proved to be more difficult. Each access occurs for a reason; in order to understand this reason, one must look into a developers thoughts prior to the access to try to identify what information was required to proceed with their solution. Similarly, to establish what information has been retrieved from the documentation one must look at the thoughts and actions that occur after the access has occurred. This allows a chain of cause and effect from emergence of a problem to its proposed solution to be built. However, it was not always easy to understand the sequence of events that led up to, and that occurred after, a particular access.

In part, the difficultly in building up a chain of events was caused by the problems of identifying reuse problems and relating them to documentation accesses. Developers sometimes described problems poorly as their attention was focused more on the description of solutions. They also worked frequently on multiple tasks at one time and this caused problems to become overlapped in the transcription making it harder to match accesses to specific problems. Problems sometimes also mutated or were decomposed into different problems as a task progressed, again making it harder to keep track of what problems were current and which had been abandoned. Relating documentation successfully to problems required iteratively going through the transcript and building up a detailed understanding of what had happened in the areas surrounding each access.

Each developer experienced his or her own unique set of problems while performing the reuse task. To ease the comparison between developers, each problem was categorised into one of the four problem categories identified previously in this thesis (mapping, interaction, functionality and architecture). These categories helped to abstract away details of each specific problem and to merge them into groups that shared particular information requirements.

While analysing the transcription for reuse problems and related accesses, occasionally other passages of text would be found that, although not directly relevant, did have something interesting to contribute to the study. This might have been an offhand remark made by the developer about documentation or the difficulty of the task. Or it may be a comment made by the researcher who occasionally overstepped the mark when assisting developers who had become stuck. Whenever any potential interesting material such as this was found, it was recorded and, if possible, clustered into groups of similar items. These clusters were then used in the detailed analysis to provide a more holistic view of the major events in the task.

5.5. Analysis

This section describes how the data contained in the developer solutions, the problem versus documentation matrices and the detailed observations was analysed to identify information about the usefulness of the pattern language and micro architecture documentations.

5.5.1 Developer solutions

Table 9 presents a summary of the solutions produced by each developer during the task. The solutions are grouped by experience level of the participant into three bands, high, medium or low depending on their previous exposure to the framework. Each cell in the table provides a brief description of the part of the framework used by the solution. Those cells that are shaded grey indicate that the task was either not completed or would not have worked if implemented. The table allows a comparison between the solutions produced and also provides an overview of each developer’s performance during the task.

The high degree of uniformity between the solutions produced by different developers is interesting. It is difficult to say what has caused this similarity to occur. In part the limited scope of the requirements, the features available in the framework and the documentation provided might all have played a role.

No developer managed to complete all of the tasks in the study. The majority of them did complete the core activities of creating a ground, a block and methods to add these to the application. One task that was not addressed adequately by anyone was constraining blocks to appear only in valid positions on the ground. This requirement, although difficult, was most likely unsolved because it relied upon other solutions and hence occurred later in the task. Developers, rather than being unable to solve it, simply ran out of time while working on it. Another task that was not solved during the study was the implicit requirement to ensure that parts of the framework, other than those intended to, could not modify the size, colour or position of elements in the application. This task, while quite simple to achieve, was overlooked either because developers were not aware that other parts of the framework might affect their application or because time pressure meant that such trivial details were not considered. It is also possible that not implementing a solution in code may have reduced the detail of proposed solutions.

	High Experience		Medium Experience			Low Experience	
Requirement	**C**	**D**	**A**	**B**	**E**	**F**	**G**
Represent the Ground	Rectangle Figure	Rectangle Figure	Rectangle Figure	Line Figure	Rectangle Figure	Line Figure	Bottom of View
Set size	Constructor	Not Done	Display Box	Display Box	Told Not To!	Not Done	Not Done
Set position	Told Not To!	Not Done	Display Box	Display Box	Told Not To!	Not Done	Get Size
Set colour	Not Done	Not Done	Set Attribute	Not Done	Told Not To!	Not Done	Not Done
How to make ground appear	Create Drawing	Init. Drawing	Create Drawing	Create Drawing	Create Drawing	Constructor!	Not Done
Adjust position on resize	Drawing	Not Done	Not Done	Partial Solution	Draw Application!	Not Done	Not Done
Prevent size colour pos.	Not Done	Not Done	Not Done	Not Done	Not Done	Not Done	Not Done
Represent the block	Rectangle Figure	Rectangle Figure	Rectangle Figure	Rectangle Figure	Rectangle Figure	Rectangle Figure	Rectangle Figure
Set size	Display Box	Display Box	Display Box	Constructor	Constructor	Not Done	Not Done
Set colour	Set Attribute	Set Fill Colour	Set Attribute	Set Attribute	Set Attribute	Not Done	Set Fill Colour
Add blocks	Creation Tool	Creation Tool	Creation Tool	Creation Tool	Creation Tool	Creation Tool	Creation Tool
Prevent size and colour	Partial Solution	Creation Tool Handles	Creation Tool Handles	Creation Tool Handles	Creation Tool Handles	Creation Tool Handle - Partial Solution	Not Done
Constrain valid position	Partial Solution	Partial Solution	Partial Solution	Partial Solution	Partial Solution	Not Done	Not Done
Only move top block	On Top	On Top	On Top	Not Done	Drawing	Not Done	Not Done

Table 9: An overview of developer solutions

Interestingly, **Table 9** suggests that developer A has performed the best in terms of the number of tasks completed. This developer was the only individual in the study to code their solution (the others produced verbal accounts). It is possible that the act of coding has in some way helped the developer to achieve this extra performance although he also asked to work on longer than the three hours to achieve this solution.

The allocation of participants to different bands of experience was performed by considering the previous exposure the participants had of the framework. The two undergraduate students who took part were both recently involved with large scale modifications to the framework. This work lasted for several months and required detailed knowledge of the framework. Both students were therefore considered highly experienced developers in this study. In contrast two of the post graduate students had no previous knowledge of the framework before the study. They were therefore considered to be low experience subjects in the study. The remaining three participants represented a medium level of experience because they had used the framework in the past to complete the practical exercises for the software architecture class (or in the case of the two class lecturers they had supervised labs where these exercises were taught and had presented example answers to students).

Considering the effects of experience upon reuse suggests that the two low experience developers (F and G) perhaps unsurprisingly have performed less well than the other participants, while the medium (A, B, E) and highly experienced (C, D) individuals appear much more similarly matched. This may support the argument that early phases of framework comprehension are critically important but also suggests that documentation is still not providing enough support for this area.

Table 9 also illustrates the impact the researcher had on the developers' performance. The researcher was present during the task primarily to observe the participants performance but was also there to answer questions about the process and to help developers if they became hopelessly stuck. The researcher was not as impartial an observer as he should have been and occasionally directed developers to work on a particular task or to avoid one that seemed trivial (hence the incidents of 'Told Not To' in the table). This assistance would not have been a problem had the behaviour been provided consistently across participants but this was not the case. There is a danger that this interference in the task may have altered some developers' performances. To understand the affect this might have had more analysis will be performed of researcher interactions later in this chapter.

5.5.2 Problem versus documentation matrices

Table 10 presents a summary of the problem versus documentation matrices. The cells contain the number of accesses recorded for all developers during the task and are separated by problem and documentation type. Pluses represent documentation accesses that were helpful and minuses are used to represent accesses that did not help. From the table it appears that the pattern language has been accessed frequently for mapping type problems. Of those accesses, a significant number of them did not provide any useful support for the problem at hand suggesting that the pattern language was not entirely successful in supporting the mapping problem. The use of the pattern language for mapping is insignificant in comparison to the use of previous knowledge. This dominates mapping problems and unlike the pattern language it appears to have been used successfully in the majority of cases. One might suggest that this is a self-fulfilling phenomenon because most developers will put forth rational plans of action based on previous knowledge, which have a correspondingly high chance of success, independent of whether or not those solutions are the best available. Also noteworthy is the comparatively high number of accesses of micro architecture and source code documentation during functionality problems suggesting that developers perceived some benefit from these techniques for functionality problems.

	Pattern Languages		**Micro Architecture**		**Source code**		**Previous knowledge**	
	+	**-**	**+**	**-**	**+**	**-**	**+**	**-**
Mapping	30	20	3	1	6	0	84	15
Interaction	5	5	12	11	5	3	13	14
Functionality	3	9	35	24	48	18	15	7
Architecture	0	0	0	0	0	0	1	3
Total	38	34	50	36	59	21	113	39

Table 10: Summary of the Problem vs. Documentation Matrices

Both the mapping and functionality problems appear to dominate this reuse task. The high number of mapping problems is surprising because these were expected to be relatively few in number but have wide reaching consequences. The large number of functionality issues was expected and underlines the significance of this problem during reuse. The comparatively small number of interaction issues that arose may suggest that these are less common than the other categories but may reflect the fact that developers were not asked to code their solutions and were therefore not exposed to many situations where interactions matter. Finally, the small number of architectural problems encountered is unsurprising because the study was not of sufficient duration or complexity to warrant the kinds of changes to requirements that cause architectural issues to arise. Those issues that did occur are all incidents where developers are worrying about the future consequences of their actions rather than experiencing the affects of bad architectural decisions.

Table 11 and **Table 12** present the total number of problems types and documentation accesses from the documentation matrices stratified into bands of experience. Average numbers are used to adjust for the extra participant of medium experience. The totals for highly experienced developers suggest that they relied on their previous knowledge rather than the pattern language while performing tasks with the framework. This is unsurprising, but it is interesting to note that the situation is reversed with inexperienced developers: those with little previous knowledge to rely on used the pattern language, as expected, to compensate for their missing knowledge.

Experience	**Mapping**		**Interaction**		**Functionality**		**Architecture**	
	+	-	+	-	+	-	+	-
High	14.5	3.5	0.5	2	6.5	1.5	0	0
Medium	22	6	10.3	6.7	24	13	0.3	0.3
Low	14	5.5	1.5	4.5	8	8	0	1

Table 11: Average no of problems experienced stratified by experience.

Experience	**Pattern Language**		**Micro architecture**		**Source code**		**Previous Knowledge**	
	+	-	+	-	+	-	+	-
High	2	0.5	2.5	2	5	0.5	12	4
Medium	6.3	7	13	8	12.3	3.3	25	7.6
Low	7.5	6	5	5	9	5.5	7	4

Table 12: Average no of documentation accesses stratified by experience.

Another interesting result, apparent from the tables, is that developers with medium amounts of experience seem to have encountered more problems and also have had more documentation accesses than either of the two other groups. This may be because developers with medium levels of experience are not affected by factors which cause the other two groups to bypass reuse problems. Low experienced developers get tied up with fundamental questions about what the framework offers, which may prevent them from tackling as many problems as the other developers. On the other hand, highly experienced developers are presumably more likely to select solutions which are well suited to the existing architecture and hence encounter less resistance when making their modifications. Medium level re-users may fall in between and therefore may be competent enough to tackle a large number of problems, but inexperienced enough not to select the best course of action, making their solutions difficult to implement.

It is also interesting to note that highly experienced developers did not report any architectural problems during the study. One might speculate that this is due to their familiarity with the framework, which makes them more confident about producing solutions which complement the existing structure. Both of the other categories of developer did report some level of architectural concern (albeit a relatively small amount). This study did not encounter a great deal of evidence for architectural problems although as mentioned earlier this is perhaps unsurprising. Architecture issues are likely to be encountered more frequently some time after the original solution has been crafted (when changes to circumstances reveal an inflexibility in the chosen implementation). Since these studies took place over a three hour timescale they did not generate this kind of circumstance. There is an indication that some developers were aware of architectural issues as they expressed concerns regarding the quality of their chosen solution. They would express doubts about the quality of their solution and look for some evidence to support their choice. *“No I really worry about hard coding it because there is so much stuff here to do with listeners, I really worry about that and I feel that I'd be missing something out and I wouldn't be happy that I would pick up on the events properly.”* (Subject B, 216). Although this can be seen as something of an architectural question it would appear to be more the preserve of the mapping problem, where such questions should affect the choice of solution made.

5.6. Detailed observations

This section presents detailed observations from the participants’ transcripts about the use of documentation during the task (this includes the use of previous knowledge and the source code). In each case a number of observations are made and evidence is presented from the transcripts to support those arguments.

5.6.1 Previous knowledge

The evidence from this study shows that developers favoured a combination of previous experience and tacit problem solving knowledge rather than the pattern language when developing their solutions. The pattern language did offer some support for problem solving but it was more supplementary in nature rather than decisive in shaping a particular solution.

Developers, upon reading a requirement of the task, often pre-empted documentation references by suggesting an initial plan or solution seemingly off the top of their head. This immediate reaction appears to have derived from previous exposure to the framework as in the following quote *"So I assume that when the application pops up, … and you have got the ground, for the sake of argument at the bottom of the screen. So that would obviously be a figure because it's on the diagram"* (Subject E, 112). The supposition that a figure is required presumably comes from the developer's previous experience of the framework. In many cases this immediate selection of a solution correctly identified a viable approach to the problem but it is nonetheless troubling because it can exclude other less common solutions from consideration. For example, the above quote describes using a figure to represent the ground, however an alternative solution (supported by the framework) is to implicitly represent the ground as part of the drawing (the drawing has support to add background and foreground images). The particular merits of this approach may be debatable, and perhaps the previous solution would be preferred in most cases, but the fact that no evaluation has occurred (or perhaps that it has be pre-empted so early on that the developer has not even verbalised it) is a problem. It suggests that the individual requirements of the task are not being evaluated fully against the existing capabilities of the framework. This has the potential to produce solutions which are less well adapted to the capabilities of the framework than they might otherwise be. This may have been caused by the artificial nature of the task where developers were under tight time constraints and were simply looking for a feasible solution.

This problem of immediate selection was even more acute when a developer had previous experience of solving similar problems. In those cases developers often recalled their previous solutions and attempted to fit them to the current problem irrespective of their suitability. In some cases this worked perfectly well, i.e. when the problem was the same and only parameters had changed, for example when changing the colour of a figure. At other times developers attempted to apply a solution which was not suitable for the current context. In these circumstances the developer was slow to realise the poor fit (if such a realisation ever happened), persisting with the solution despite its awkwardness. The following example illustrates the kind of problem this tended to cause: A developer attempting to constrain the block's size during its creation tried to override the creation tool, but found that he was unable to control the size in this manner (the created figure was a private member of the creation tool). This led the developer to cut and paste the implementation of creation tool into a new class parallel to the existing creation tool in the tool hierarchy and proceed with the modification from there. When asked why this approach had been taken the developer replied, *"I'm not that happy about it. I have seen it used before though I wasn't that convinced about it was a…I'm surprised and I would expect there is a better way of doing it. Multiple students have told me I need to do that… Its convincing… yeah I suppose it is because I've heard it from two or three sources so… and having had a look here I can't see…"* (Subject A, 228). This indicates that, although the solution was ugly, its use in previous circumstances (and the apparent lack of any viable alternative) suggested that the approach was required now. In fact several alternatives did exist but none involved setting the size in the subclass.

Using previous solutions in this manner was not restricted to developers with previous experience of the framework. Novice users also referred to previous solutions to address their problems, only their knowledge did not come from the framework but from previous programming tasks in different domains. One example comes from a developer who was trying to make a button to create blocks in his application *"I'm trying to find that location. When you actually press the button what actually happens…Hmmm… have listeners… hmmm. Ah that's what I'm after. I want to know what happens when you press the button…Ah I see what they have done. So the button that is actually in there at the moment hasn't actually got an action listener on it."* (Subject G, 130). The developer became stuck because he anticipated the typical Java approach of creating a button, which had an action listener defined elsewhere in the code. The participant couldn't find any such action listener in the example and this caused considerable confusion until he deduced that this was handled behind the scenes by the framework.

5.6.2 Pattern language

Although many of the problems were addressed by developers using their previous knowledge and experience there was one circumstance where this could not happen. Novice users with little or no previous exposure to the framework did not have the knowledge to produce solutions that were informed by the existing structure of the framework. The pattern language seems to have played an important role in overcoming this problem as its patterns help decompose and explain the major concepts and roles that exist within the framework. This has helped developers to utilise those concepts to create solutions or to replace initial ideas about a solution with ones that are consistent with the existing structure. For example one participant, who was trying to create a mechanism for adding blocks to the Blocks World, began with a solution based on overriding mouse behaviour *"So … with the left click you would add … and the right click would move it. That would be my solution as simple as possible …I'd probably use the mouse interaction on top of the canvas."* (Subject F, 32). This opinion changed upon reading about the concept of a tool in the pattern language *"Then again I'm seeing that we have got an adding buttons to the toolbar here. If there is already a set procedure for adding a button and making that … for adding figures to the canvas. I'd be as well to use that… [Reading the pattern]… Yeah so okay. I have moved away from the idea of left and right clicking of the mouse. This is giving me [an idea] how to create a tool button"* (Subject F, 35). Despite its benefit to novices, experienced framework developers did not appear to gain much benefit from the pattern language in this way, suggesting that once learned the basic concepts of the framework are not forgotten but instead are internalised into a developer's perception of the framework.

The pattern language also helped developers (both experienced and novice alike) by providing examples and inheritance hierarchies within its patterns. Examples helped to illustrate how the concepts introduced by the pattern language could be implemented in code while hierarchies were used to identify specific classes to fill a role in the framework. For example *"The pattern language I know from using it before is a good example for creation tool. So basically just put that code into the draw application obviously with a new creation tool that would create a new blocks figure and we have also got help with adding the button as well."* (Subject C, 33) and *"I'm going to create a figure. I found the pattern language… the hierarchy of the figures quite good…. Two options I'd have a rectangle figure or a group figure."* (Subject C, 22).

In both cases one can observe a qualitative difference between novice and experienced users. For novices (e.g. participant F) both the examples and hierarchies appear to provide information about how a part should be used and to identify existing classes to reuse within the framework. For experienced developers (e.g. participant C) the information provided was different; they already knew how to perform the common tasks covered by examples (e.g. creating a tool) and they also already knew at least some of the possible options in each hierarchy. Instead, for them, the example becomes a piece of boilerplate code that can be easily modified for their needs (saving time more than anything) and hierarchies become defensive tools to make sure that no suitable classes have escaped attention, rather than to find new candidates.

The original expectations for the use of the pattern language appear to have been optimistic. It was supposed to act as a guide for developers leading them towards good solutions in the framework. Instead this seems only to be true in cases where the developer is completely unfamiliar with the framework and then only to the extent that it allows concepts from the framework to be integrated with the developers own plans. Developers with more experience appear not to require this stepping-stone, instead using previous knowledge of the framework and reference to previous solutions when constructing new solutions to framework problems (of course this knowledge may have been developed through previous exposure to the pattern language). For all developers the pattern language did offer useful support in terms of both examples and inheritance hierarchies, which enable low-level details such as how parts work and what parts are available to be addressed.

5.6.3 Micro architectures

The data collected in this study suggests that understanding the interactions in a framework remains a hard problem for documentation to address. Developers answered many trivial forms of interaction problem effectively but occasionally a larger question involving a series of interactions and dependencies would arise that developers found much more difficult to answer. Documentation support for these problems was found to be lacking and in particular the micro architecture call graph documentation proved to be ineffective at both identifying interactions and describing their significance to developers.

The most common form of interaction problem that occurred during the study was the need to obtain a reference to another class. This problem was typically addressed from some combination of source code, class interfaces (from the micro architectures) or previous experience with the framework. Using this material, developers found it quite easy to string together a sequence of references that would result in access to the correct class or interface, as the following example illustrates *"[looking in source code] There is initialise drawing there. Which looks helpful but that's calling… create drawing…. Back to init drawing! (Sigh)",* (Subject D, 36). In part this finding was due to the developer simply adding to the interactions already in the framework without affecting its existing behaviour. A more significant problem presented itself whenever a developer was asked to modify the sequence of interactions within the framework. This required a more detailed understanding of the relationship between classes than in the previous case and appeared to be difficult for developers to achieve.

One such problem is illustrated in the following example. The developer wants to detect changes to the drawing's size in order to reformat the contents of the drawing with respect to the space available in the window. This problem is complicated because JHotDraw utilises the MVC design pattern, which de-couples the appearance of the drawing (its view), from the state of the drawing (its model). In order to reformat after a window resize, the developer had to understand the sequence of events that would flow from the window through the model and view. The developer was familiar with the MVC pattern and could identify the key roles of drawing and drawing view within the framework but despite this he was unable to create an accurate account of the behaviour between the two classes upon a resize. This process continued for over an hour, *“I'm clutching at straws I'm just looking at anything that… seems to go down… I just want to tie this figure up! …to something I can’t see a place to tie a figure, to register it.”* (Subject B, 220), and the final answer produced was less than convincing, *"Drawing view… when it does a check damage it gets the listeners but I will tell you why I'm not happy with it, I'm not happy with it… explicitly registered the… the relationship between the drawing and the drawing view I suspect is established elsewhere, right, and that a drawing has automatically a listener for a drawing view. I'm not comfortable with that at all…, does that take you far enough?"* (Subject B, 228). The important point to highlight from this example is the amount of time that was wasted searching for a solution because of the lack of understanding of the interactions across the framework.

The micro architecture call graph notation was the primary support for interactions offered in this study. It was supposed to illustrate how each of the major interfaces in the framework was called by other parts of the framework. The intention was the developers could use this call graph to understand how the existing code in the framework made use of that interface. This expectation proved to be somewhat optimistic. Developers seldom used the call graphs at all and when they did they were not interested in finding out about the behaviour of the surrounding code.

The failure of the call graph documentation to support interaction problems can perhaps be attributed to a number of specific weaknesses. Developers appeared to find it difficult to know which interface to start from as there was no guidance in the documentation that related behaviour to interactions. The interactions were also fragmented into little pieces by the need to have separate graphs for each method of an interface and to limit the length of call sequences to include only the calls made between interfaces. This meant that the interactions that were being shown were largely devoid of meaningful domain semantics (i.e. application functionality), making it difficult for developers to appreciate their significance. The notation used exacerbated this problem, as it provided nothing other than the relationship between method calls. The behaviour of the calls was not included and was supposed to be looked up in the source code if required. Developers did not appreciate this separation, *"Yeah, if you could then, yeah, if you were able to click on a method in the coloured blocks diagram and then jump straight to the source code that would be helpful…"* (Subject E, 58). Finally, the developers seemed less interested in the behaviour leading up to a method call than in the behaviour after the call (although this might be more to do with gaining an understanding of functionality than interactions): *"You see there is something up about the…about this [Developer is looking at Drawing.add call graph] I'm wanting to see… and I've found this a number of times looking at this. This is telling you what calls that and I want to see what add is doing."* (Subject A, 355). This does not mean that the call graphs were never useful. There were occasions where a developer successfully mined information from them. For example *"So the only methods I need to worry about are standard drawing view selection handles and decorator figure handles."* (Subject E, 47). But these occasions were few and they do not live up to the expectations placed on the documentation at the outset of the study.

Interactions appear to have been addressed poorly by the documentation. The proposed call graph documentation appears to be too fragmented and simplistic to offer developers the support they require for addressing these problems. It can also be argued that developers were not sufficiently familiar with the technique (revealed in informal talks with participants after the task) and that this reduced its effectiveness. Perhaps better education about the technique and more opportunity to practice before the study would have improved its performance.

5.6.4 Source code

Functionality support was primarily provided by the source code. Developers were expected to use the other documentation, particularly micro architectures, to identify classes of interest and then use the source code to understand how those classes operate. To a certain degree this was found to be the case. Developers did identify classes in other documentation and often turned to the code to gain further insight but that was not the whole story: developers also complained about having to use source code and frequently requested access to other documentation, which would supplement this information.

Use of the source code varied across situations, sometimes the reader would be searching for a particular class or method at other times the reading would be less directed and more opportunistic, capturing pieces of knowledge by accident rather than intent. There was a lot of use of the source code by all of the developers in the study and many accesses appeared to reveal information that was helpful to the developer *"Ah right okay, so that's where…yeah its got set methods … This has definitely been helpful because it can initialise line figure."* (Subject F, 102). However, not all developers were happy to use the code and several complaints were recorded. *"Then again, it should be noted that the idea of looking up source code to see how an application works is the least appealing option."* (Subject F, 203), *" So I don't have an example of how to use this and I don't have any JavaDoc. I'm just going to have to resort to the source code which is a bit [frustrating]"* (Subject E, 116). This suggests that, although accurate, it was an effort to identify and understand material of interest.

Given the apparent unpopularity of source code it appears relevant to question why examples were so popular in the pattern language? One potential argument is that they provide an instant solution to a problem that can simply be cut and pasted into their application. They also remove a lot of superfluous detail and provide a concrete illustration of what structures of the framework can be used. It was also interesting to note that the source code was also used as an implicit example for modifications. Developers would look to the existing code to find a way of doing something and then generalise it to another case (often claiming to cut and paste the solution from one place to another). *"So I think we have identified creation tool but we have still to come up with how we are going to add these blocks. Well I think I would probably get a yeah a … so what you have here is an example of the selection tool."* (Subject F, 138). It is hard to tell from this study whether such behaviour is caused by a desire to maintain architectural consistency or simply as a mechanism to achieve a solution as quickly and simply as possible.

The unpopularity of source code is further supported by the many requests made by developers for access to JavaDoc documentation. For example, *"So back to the code. I'd rather use JavaDoc if it were here."*, (Subject A, 141) and *"I'm going to look in the source code but I'd really like to look at the JavaDoc",* (Subject B, 258). JavaDoc can be considered as a more abstract representation of the behaviour of the code although this comes at the cost of lower precision but with high navigability. Developers also appear to have avoided source code by the use of the micro architecture class interfaces, which were often consulted to identify methods to use. On some occasions the identified methods would be checked against the source code definitions but on many others assumptions about the likely behaviour (or perhaps past knowledge of the behaviour) appear to have been used.

Functionality was described mainly through the source code of the framework. Developers demonstrated that they could understand the behaviour of parts of the code in order to make decisions about their intended solutions. Developers also illustrated the potential for code to operate as an example, both from its popularity within the pattern language documentation and also by reusing snippets of code in new modifications but there appears to be a significant effort involved in locating and understanding relevant code, which has resulted in developers turning to alternative sources where possible.

5.7. Researcher interference and reliability

The researcher had a great deal of experience with JHotDraw and its operation and so was well placed to offer critiques of the participants' solutions. At times the researcher found it difficult to provide this critique because the participant's solution differed from the anticipated answer. This occasionally led to solutions being accepted when in reality they would not work or would require additional steps to complete. It was also difficult to avoid participating in solutions and on occasion the bounds were overstepped and a dialogue developed. This actually helped encourage the developer to describe what they were doing but ran the risk of too much support being provided and prompting them towards a solution.

Qualitative analysis relies critically on the researcher's judgement when categorising data. This has the potential to spoil the categorisation because this judgement might be flawed or biased with a particular mindset. To protect against this the researcher's judgement must itself be analysed to determine how accurate it appears to be. To perform this measurement an inter rater reliability test is used which compares the categorisation of data performed by the researcher against other researchers to detect if there are significant differences of opinion.

In this case the inter-rater test was carried out by using the researcher's two supervisors as alternate researchers (Rater one and Rater two). They were both given a short section of a developer's transcript (six pages) and were asked to produce a problem versus documentation matrix for it. The matrix was chosen for the test because it involves the most significant amount of categorisation in the analysis. Researchers have to identify problems, relate them to accesses and then decide whether an access was helpful or not for the re-user.

The results of the reliability test are shown in **Table 13**. Initially, this table appears to show a significant difference between the categorisation made by the three raters. The common problems are shown in normal type while those where a disagreement occurred are surrounded by a box. In order to ascertain whether this was true the raters met and went through each entry in the table, identifying the problem that was observed and why it has been allocated to a particular category on the table. This provided a richer insight into the categorisation below and revealed that a number of factors were exaggerating the apparent difference between raters.

	Pattern Language		**Micro architecture**		**Source code**		**Previous knowledge**	
Mapping		22	19 $\boxed{18}$ $\boxed{19}$	20			17 $\boxed{18}$ $\boxed{16}$	21
Interactions			24 27					
Function	$\boxed{22}$	$\boxed{23}$	25 $\boxed{24}$ $\boxed{25}$ $\boxed{27}$ $\boxed{33}$	23 $\boxed{20}$ $\boxed{22}$ $\boxed{23}$ $\boxed{24}$ $\boxed{20}$ $\boxed{22}$ $\boxed{24}$ $\boxed{27}$	21 34 34 38 $\boxed{21}$ $\boxed{33}$ $\boxed{34}$ $\boxed{20}$ $\boxed{34}$ $\boxed{24}$ $\boxed{28}$	26 28 35 37 $\boxed{21}$ $\boxed{26}$ $\boxed{28}$ $\boxed{35}$ $\boxed{38}$ $\boxed{33}$ $\boxed{26}$ $\boxed{36}$	$\boxed{21}$ $\boxed{33}$ $\boxed{39}$ $\boxed{33}$ $\boxed{30}$ $\boxed{39}$	$\boxed{25}$ $\boxed{30}$
Architecture							$\boxed{31}$ $\boxed{41}$	30
Other								

Table 13: Composite view of inter rater problems

Discussing the problems revealed that some of the differences were caused by raters disagreeing on the time to record a problem. This caused one rater to record a time of, for example, 16 minutes when the others recorded 17 minutes despite the fact that they were describing the same problem. To resolve this a single time was agreed by all raters to represent each problem. Raters occasionally missed problems that one or both of the others had identified. In each case the other raters were asked if they agreed about the problem and its position in the table and, if so, it was added to the problems found. Sometimes the raters disagreed on the position in which problems were allocated on the table. This resulted in five of the original researcher’s problems being re-categorised from mapping or interaction problems into the functional category.

	Pattern Language		Micro Architecture		Source code		Previous knowledge	
Mapping		~~22~~	19	~~20~~			**17**	21
Interactions			~~24~~ ~~27~~					
Function			**25** 27	23 20 22 24	**21** **34** 34 38	**26** **28** **35** **37**	33 39	30
Architecture							41	~~30~~
Other								

Table 14: Adjusted problem documentation matrix

Having made these adjustments the final categorisation is shown in **Table 14**. The problems that were agreed by all raters and did not move position are marked in bold. Those that were agreed and have moved are shown in normal type in the new position and are struck through in their original position. Finally, those problems that were added by the researcher or one of the raters have been underlined.

	Same	Moved	Researcher	Other Raters
Problems	17 21 25 26 28 34 35 37	20 22 24 27 30	21 23 34 38	19 33 39 41
Total	8	5	4	4

Table 15: Detail of inter rater differences

Table 15 shows the problems that each rater had in common, those that were moved and those that were overlooked by the researcher or by the other raters. From this table it can be shown that the raters agreed upon the majority of problems discovered. Further to that, five problems were moved from the original categorisation but only in terms of the problem categorisation, the documentation type and documentation support were always agreed upon.

From this it is possible to conclude that the work of the original researcher, at least in terms of documentation type, problem type and support offered by documentation, is in broad agreement with that of his peers, indicating that the transcripts are not affected significantly by developer bias. The rating has shown that a single developer will miss some of the relevant problem accesses and also suggests that there may be a tendency for the original researcher to categorise functional problems mistakenly as either interaction or mapping tasks. This may reduce the significance of the number of problems found in the study but this is relatively unimportant because seven subjects were never a representative sample of the population and therefore the size of each problem category must in any case be treated with caution.

5.8. Results

The evaluation of the pattern language and micro architecture documentation has helped to develop an understanding of what support documentation offers a reuser. The study has also provided a detailed look at the existing problem categories for framework reuse and has provided valuable experience in the use of qualitative analysis for evaluating documentation. This section concludes the analysis by summarising what was learned about the documentation, the framework reuse problems and the process that was used.

5.8.1 Pattern language

The evidence presented suggests that the pattern language was effective at introducing concepts to developers, particularly novices. Its patterns described many of the important areas of the framework, explaining what parts exist and the roles they are expected to play. Examples in the pattern language also helped to introduce concepts by illustrating how parts could be used to solve common problems in the framework.

On the other hand, pattern languages struggle to compete with a developer's previous knowledge during mapping problems. There is no doubt that such knowledge is an important aspect of reuse. Developers will learn from past experiences in the framework and there is nothing wrong with them using this knowledge to advance their current reuse task. Such experience is not always a benefit; sometimes it can override other sources of information and create problems during reuse. For instance, when the experience comes from outside the domain of the framework, there is a danger that it will be at odds with the existing architecture of the system, complicating rather than assisting the reuse task. Problems can also occur with framework specific knowledge. In this circumstance a solution can be selected because it was useful in the past. Just because a solution was good for a problem in the past is no guarantee that it will be a good solution in the present but because of the close relationship between a developer's problem solving ability and their previous knowledge it is difficult for other documentation to inject opinions and force a wider viewpoint.

The pattern language in this study contained examples and hierarchies both of which were found to be useful. Examples help to introduce concepts and show implementation detail while hierarchies present a selection of interchangeable classes and can provide suggestions of functionality. The question is not whether examples and hierarchies are useful but rather whether they should be integrated into a pattern language? The answer to this question is not clear; while both forms of documentation integrate well with the pattern language it is possible to conceive of both working as standalone documentation. There also was a suggestion during the study that adding these to the pattern language detracts from the patterns themselves; with developers appearing to be drawn to these elements and sometimes overlooking relevant material in the pattern text.

5.8.2 Micro architecture

The micro architecture documentation provided minimal support for interaction problems. Developers seldom used the call graph notation, which was supposed to provide this support, and there is evidence that they found the syntax and purpose of the graphs confusing. They also complained that the call graphs described the wrong type of information, claiming to prefer information about the interactions that occurred within the implementation of an interface rather than the sequence of calls that led up to it.

The interface descriptions and class hierarchies were both found to be useful for the identification of functionality during reuse. However, the support provided was rather trivial as it merely presented class and method names and relied on the developer to speculate about their functionality.

The disjoint nature of the micro architectures was also unpopular with developers. Moving between views was awkward as the reader had to backtrack to the micro architecture index before each switch. It was also difficult to move between the micro architectures and source code, as this required the developer to find the relevant source code amongst the various files of the framework and open it in a separate editor. There is a suggestion that some of these problems may have been caused by lack of familiarly with the micro architecture documentation. While it seems unlikely that this could explain all of the weaknesses identified in this evaluation, it may have reduced its effectiveness in some situations. Future studies must do more to properly communicate how to use a documentation technique before evaluation.

5.8.3 Reuse problems

The evaluation has provided an opportunity to gain further insight into the problems that occur during reuse. The study did not discover any new types of problem but did provide a tentative view of the relative frequency and significance of each problem during the task.

Mapping problems occur throughout the task and are not restricted to low experience subjects. High and medium experience subjects are also affected although the problem for them might be more one of selecting a solution rather than identification. Mapping problems still appear to be the most significant of the problems discovered. They dictate the actions of the developer over large periods of the reuse task and by their nature cause other problems to occur or to be avoided depending on how well the solution has been mapped onto the existing parts of the framework.

Interaction problems were found to occur less frequently than originally assumed. In part this may be because of the experimental situation, which did not go into the detail of coding a solution, but it also might reflect that this type of knowledge is required less frequently from documentation. One caveat to this argument is that when a significant interaction problem occurs, such as trying to trace aspects of MVC in the framework, a large amount of time and effort can be spent attempting to identify and understand the interactions. This suggests that, although interactions may not be frequent, they can be important to the reuse task.

Functionality problems were very frequent in the task. Despite their frequency they do not appear to trouble developers overly during reuse. The reason for this is that each functionality problem is relatively small and therefore quite contained. In such circumstances a developer’s failure to understand a part of the framework does not affect their overall solution. Functionality problems are also well supported by available documentation. This study has shown that developers do not like using source code, but the fact is they can use it and often do get useful information from it. There are also other alternatives, such as JavaDoc, or developers can also use class or method names to guess at the underlying functionality.

Architecture problems were not captured well by this study. They remain a target for future investigations, which will have to take place over a longer timescale to identify the size and significance of architectural issues. One observation that can be made about the architectural concerns identified in this study is that they all represented worries about the future impact of solutions. Arguably, this can be seen as an extension of the mapping problem where one is trying to find a solution which not only fits onto the existing architecture but that is also intended by that architecture.

5.8.4 Study lessons

The evaluation of framework documentation is a relatively rare occurrence. In part this is because of the difficulty in gaining access to relevant information and then being able to analyse how that data relates to documentation performance. This study has attempted to use a qualitative approach to overcome some of the problems associated with such evaluations. While this has provided many useful insights into documentation use it has also been a learning process in the use of such evaluations.

The use of a talk aloud protocol and video recording software worked well. It helped to provide insight into the thought processes of developers during reuse and it allowed the developers’ thoughts to be recorded immediately after they had occurred. This is a tremendous advantage over the first study in this thesis because there is no opportunity for events to be forgotten or reorganised to hide mistakes and the taping means that the data can be captured verbatim for later analysis.

One of the most significant challenges for this type of investigation is how to deal with the volume of data it produces. The talk aloud protocols created an almost overwhelming amount of material to transcribe and analyse. If possible, future studies should try to minimise the amount of information they capture, or alternatively increase the amount of person hours available to process it. Also, if recording to tape, high quality microphones should be used and several should be positioned around the environment to make certain that the audio captured is as strong as possible. Similarly, the acoustics of the environment should be considered and, if possible, a quiet location chosen which will not be affected by other sources of noise.

A major problem with qualitative analysis is that it relies on the specific circumstances of an investigation to dictate what information to capture and what processes to use during analysis. To some extent this is unavoidable but it can be mitigated by careful preparation. A pilot study, performed on a small amount of data before the real investigation began, would allow a researcher time to identify and to practise suitable analysis techniques.

Future studies should also seek to address different timescales and environments of reuse. This study was limited by its use of academic volunteers and the relatively artificial task they were asked to perform. Ideally future studies should seek industrial settings to perform evaluations. Industrial users are likely to have different motivations to complete a task and real world tasks will produce more authentic requirements which would be of benefit to future evaluation. There is also a need to apply similar studies to the many forms of unvalidated documentation that exist. This may be quite difficult because many of the techniques are not described in sufficient detail to allow others to create effective replications but it is at least hoped that the developers of future documentations might consider the importance of evaluation and provide evidence to support their claims.

5.9. Conclusions

This chapter has described the evaluation of a pattern language and micro architecture based documentation that aimed to address the four problems of framework reuse. The study involved seven academic participants instantiating the JHotDraw framework to create a simple Blocks World application. The evaluation was performed using qualitative techniques and employed a talk aloud protocol and video capture to record developer thoughts and actions during the study. The data was transcribed into textual narratives describing each developer's reuse process and the problems they encountered. This was then analysed by creating different views of the data to identify what problems were experienced, what documentation accesses occurred and what solutions were produced.

The results show that the pattern language provides some support for mapping problems by introducing concepts, providing examples and class hierarchies. Sometimes, this support is overwhelmed by a developer's natural instinct to trust their previous experience which, although often useful, can sometimes trap developers into poor solutions. The micro architecture documentation was less useful and its support for interaction problems was largely ignored. It did provide some support for functionality but even this was quite limited, consisting of identifying classes and methods only by their names. The study also revealed that source code, although effective, was unpopular with developers who often requested JavaDoc and were reluctant to use the code. It has also confirmed the existence of the four problem categories of framework reuse and has provided insight into the frequency of their occurrence and the impact they have had on developers' reuse processes.

The information provided by this study, while identifying weaknesses in both documentations has not ruled out the use of either as framework documentation. In the future improvements can be made which might help the pattern language to complete against and influence the decisions made by a developer's previous knowledge. The micro architectures could also be improved by providing a more unified notation which possibly describes semantic sub sections of the framework and by better educating developers about its use.

6 Conclusions

This thesis makes several contributions to the comprehension and documentation of object-oriented frameworks. It has identified a set of problem categories that affect framework users during reuse. Namely: mapping, interaction, functionality and architectural problems. The identification of these categories enables the comparison of the level of support offered by competing documentation techniques. An understanding of these problem categories can also help to drive the development of new forms of documentation, informing their content and presentation to increase the support offered for the problem categories. Finally, they can assist with the identification of useful combinations of documentation to provide coverage for different types of problem while minimising the amount of redundancy or overlap in the material provided.

This study has investigated the concept of a pattern language and modified the format in an attempt to make them better suited to supporting mapping and architectural reuse problems. It suggests that pattern languages can be an effective way to introduce new concepts to developers. In many cases, when a developer is unsure how to proceed, the language can act as a prompt and suggest ideas to them. It supports mapping solutions by describing relevant parts of the framework and shows examples for developers to copy. However, pattern languages are far less successful when a developer has already made up their mind about what to do. Sometimes framework developers appear to become fixated with a solution even if it is not the best option to take. In such situations developers ignore the pattern language and proceed with their own solution regardless of its effects on the rest of the system. The pattern language was expected to act like a devil's advocate in such situations, challenging the developer with alternative solutions and allowing them to consider the relative merits of each approach. This did not happen and as a consequence, developers would sometimes produce modifications which were difficult to make, were incompletely specified or which damaged the integrity of the existing framework.

The thesis has also investigated micro architectures as an aid to framework comprehension and proposed a documentation to describe them. Micro architectures help to address the scale of a framework, making the implementation detail easier to digest by dividing the code into independent sections of functionality. The micro architecture documentation has not been as useful as expected. In particular, the call graph view, which was supposed to help developers understand and trace interactions through the framework, was hardly used during the evaluation. In part this may have been caused by lack of familiarity with this type of documentation (clearly it did help in one or two isolated cases) but more importantly, it appears to have required too much effort to be useful. Developers made better use of the other views offered by the micro architecture documentation. The interface descriptions and class hierarchies were both useful for suggesting functionality and assisting navigation to relevant parts of the source code, although this could arguably be better provided by existing approaches such as JavaDoc. There is a suggestion that some of the difficulties may have been caused by a lack of familiarity with the technique but it seems likely that further alterations will be required to produce effective micro architecture based documentation in the future.

An unexpected finding of this work was the extent to which previous knowledge influences the decisions taken during framework reuse. Previous knowledge shapes the way developers think about and perceive reuse problems. It can derive from any past learning experience, not just computer science or programming knowledge but other forms of problem solving or logical thought. This can have a significant affect upon how a developer attempts to map a solution onto the existing structures of the framework. Sometimes, when the expectations of the re-user align with the existing architecture, there is no problem and previous knowledge actually helps to create the solution. Sometimes, this doesn’t happen and there seems to be a disparity between the solutions proposed by the developer and the material offered by a framework. In these situations previous knowledge can blinker a developer and prevent the consideration of alternative solutions. It is possible that the extent of this problem has been exacerbated by the experimental set-up used for the evaluation. The short timescale may have placed too much pressure on developers to produce a solution, resulting in a tendency to go with their first instinct or to rely more heavily on past experience. This remains an open question for future studies to address.

Another contribution made by this thesis is the identification and description of two alternative approaches to documentation evaluation. A lightweight strategy was adopted in the first experimental study. It used a combination of user reports and questionnaire information to construct a profile of framework reuse and the capabilities of different documentation. This approach provides good insight into a developer’s thoughts about documentation and also allows a wide range of techniques and large numbers of developers to be considered at the same time. On the other hand, it can only provide a second-hand insight into the reuse process and there is a possibility that this might reduce the accuracy of its findings. A more heavyweight process was employed for the second evaluation, using an in-depth, observation study to assess the utility of the two new forms of framework documentation. This approach provides a much more accurate insight into documentation performance but is far more expensive to perform and can only address a small number of documentation techniques and developers at a time.

Both techniques have demonstrated their usefulness during evaluation but it may be possible to make alterations from the implementation presented in this thesis to improve future studies.

6.1. Lessons learned

Several valuable lessons have been learned which should benefit future research. The evaluation of documentation has revealed that it doesn’t always operate as expected. It is important to detect such occasions and to try to identify the reasons for the unexpected behaviour. To illustrate the benefits of evaluation consider the micro architecture documentation proposed by this thesis. It initially seemed to be an effective technique. It presents important parts of the framework to a re-user and provides an understanding of the context in which the part should be used. However, when evaluated the documentation was not used as expected. Developers did not develop an understanding of the surrounding interactions; instead they used the documentation to provide limited insight into functionality. The disparity between intended and actual use was apparently caused by information being distributed across multiple views which required considerable effort to navigate. This was something that developers did not seem prepared to do which severely reduced the effectiveness of the documentation. This result, although potentially disappointing, is actually quite helpful because it provides a direct suggestion about how such a technique could be improved (namely by integrating separate views into one cohesive document). By performing such an evaluation, and possibly iterating over several versions, the final documentation produced should be more likely to provide useful support for framework reuse.

This thesis has also identified a number of problems which can arise during empirical work. Experimental factors such as time pressure and lack of familiarly with documentation are difficult but important problems to overcome. Time pressure can lead to participants producing poor quality solutions in a rush to meet the task deadline. A lack of familiarly with documentation can cause developers to favour existing techniques or use new techniques inappropriately. The solution to these problems is straightforward. Better training ought to be provided so that developers know how to use documentation prior to an experimental study. Where possible studies should take place over longer time scales (e.g. days or weeks) or require less work in the available time (e.g. provide focused questions for the developer to answer). Other improvements that could be made to the evaluation process include: reducing the amount of data captured during observations, allocating more person hours to process the data, and the use of a pilot study to refine the analysis procedures beforehand. Reducing the data captured by a study is unusual for a qualitative experiment, the traditional argument being that it is better to collect too much data rather than too little and then sort out what is relevant during analysis. This can result in large amounts of data and a difficult subsequent analysis. It may be possible to design a study which only captures key information; for example explicit references to documentation, which would reduce the amount of analysis required. Alternatively, if a large amount of data must be captured, it would seem sensible to budget for a large period of time for analysis, or provide extra personal to reduce the burden. In either case, it is necessary to include a pre-study to allow researchers to practice their observation skills and refine suitable analysis techniques.

A question that naturally arises from this thesis is how should object oriented frameworks be documented? The answer is somewhat qualified, because much more remains to be done in the evaluation of framework documentation, but two approaches can be identified as promising candidates. The combination of a pattern language and a set of micro architecture documentation is one approach. Pattern languages can address mapping and architectural issues and micro architectures appear relevant for an understanding of the implementation details of a software framework. In both cases this thesis has revealed deficiencies within the existing documentation that limit their effectiveness. As such it is difficult to recommend this approach without further research to improve their usefulness.

An alternative approach, which may be of benefit in the meantime, is to use a combination of examples and practical exercises to teach developers how to use a framework. There is some evidence in this study and in the literature that such an approach can be useful. Students in the software architecture class were exposed to this environment for a number of weeks before being asked to create their modifications to the framework. While other documentation was available, it is clear from their responses in the questionnaire (Chapter 2) and in the coursework reports, that they found examples and practicals to be a useful form of documentation during this time. This perception is backed up by the work of Dénommée (Dénommée 1998), Sparks et al. (Sparks, Benner and Faris 1996) and Schull et al. (Schull, Lanubile and Basili 2000) who have all commented positively on example driven techniques. Arguably, this approach also has the benefit of being a relatively simple form of documentation to create. Examples are easy for experienced developers to create and while some critics argue that the choice of example is important (e.g. Dénommée 1998) this work used a fairly arbitrary collection of examples and practical exercises to achieve at least an initial level of framework comprehension. Despite their utility examples are not a panacea, as there is evidence in this thesis and also in the wider literature to suggest that they can be incomplete in their coverage (Schull, Lanubile and Basili 2000) and damaging to the architecture of a framework (Schneider and Repenning 1995). In the longer term, it is hoped that pattern languages and micro architectures will be able to provide a documentation technique which overcomes these limitations while still retaining the ease of creation and use of an example driven approach.

6.2. Future work

This study has suggested that a systematic empirical evaluation of documentation can be an effective strategy to identify the problems of framework reuse and to enhance the support provided by framework documentation. There are a number of ways in which this study could be improved upon by future evaluations, including: the use of different frameworks, more realistic environments and longer periods of evaluation. The use of different frameworks (especially different types of framework) may help to further define the problem categories identified in this study. It may identify problems which did not occur within JHotDraw or provide a more accurate understanding of the relative importance and frequency of problem categories across frameworks. Setting evaluations in different environments will also improve the accuracy of the evaluation. This thesis drew its findings from academic subjects, working in an experimental situation. An alternative setting, using industrial participants and using actual development situations could result in more accurate findings. It would also be useful to assess documentation use over a longer period of time. This would enable developers to overcome any learning effects from new documentation, expose the documentation to a wider range of problem situations, and enable more exploration of the differences between experienced and novice framework users. However, longer evaluations would be harder to perform because more data would have to be collected and it would be difficult to account for external influences affecting a developer during observation periods.

6.2.1 Improvements to investigated documentation

The form of pattern language and micro architectures used in this study were found to have weaknesses in addressing framework reuse problems. The study has found no evidence to suggest that either form of documentation should be abandoned altogether but the results of the user evaluation suggest areas of future work which may improve the effectiveness of both forms of documentation.

The pattern language did not do enough to attract the user away from their pre-conceived ideas of a solution to those more suited to the framework. Future pattern languages should concentrate on providing such support. This may require a greater emphasis on the types of problems a developer may encounter during a modification rather than a more general description of framework functionality (a point advocated in other discussions about pattern languages e.g. (Meszaros and Doble 1998)). It could also require the text of each pattern to contain a section on motivation to argue for the adoption of that solution. Future studies ought to compare different types of pattern against each other to provide a better understanding of what types are possible and which are best at communicating mapping support to developers. Another topic which requires significant research is how to identify the range of patterns that ought to be included in a language. This is important because it defines the range of support offered by the documentation. At present it is not clear how to reliably and systematically identify a set of patterns which will provide this support without extensive domain knowledge on behalf of the pattern writer. Such guidance is critical in order to reduce the effort and cost involved in pattern writing and key to writing effective pattern languages.

The micro architecture documentation requires greater modification to become useful. Its major weakness was the number of different views it contained and the difficulty in navigating between them. This could be improved in a number of ways; perhaps the most obvious being to bring the separate pieces of information together into one larger description. A potential example of this type of documentation was shown earlier in the thesis (**Figure 24**, Chapter 4). That particular form of description was disregarded because of the difficulty in identifying and describing parts of the framework as cohesive subsystems. More work has to be done in this area to investigate whether such micro architectures can be reliably decomposed from a larger framework and if so whether a notation similar to that shown in **Figure 24** is adequate to document them.

A final question of interest is the relationship between pattern languages and micro architectures. In this thesis they have been presented as two separate forms of documentation but there is a need to consider the relationship between them. This is because mapping and architectural concerns eventually have to give way to implementation detail as a solution is created. This suggests that the relationship between the high level design type documentation and the low level implementation detail is important and warrants further investigation. Another motivation for this work is that both the pattern language and micro architecture are closely related to the concept of a pattern. This similarity leads one to suggest that perhaps, although serving different purposes, the two forms of documentation are actually very similar. This could lead to the creation of a unified pattern language with design oriented patterns at the top and implementation oriented patterns at the bottom. Such an idea is appealing because the interplay between the two forms of pattern might help with the process of identification. In other words, the relationship between the two levels of documentation might actually help with the identification of both types of pattern.

6.3. Other documentation techniques

This thesis has focused primarily on the evaluation of a pattern language and micro architecture based documentation but a number of other forms of documentation could benefit from further exploration particularly; sets of examples, UML and forms of tool support.

JHotDraw comes supplied with four example applications and these have been thoroughly mined and explored by participants searching for clues about how to use parts of the framework. Examples can be successful, and they are often advocated in the literature as the means to document frameworks. This study would not disagree with their utility but from a practical perspective they have limitations. Examples are relatively cheap to produce; the creator of the framework can churn out a few simple applications that show a lot of functionality quite easily. The challenge is to properly package those examples so that subsequent developers can easily appreciate what each example is trying to get across. In addition the concrete nature of examples suggests that there would have to be a great number of them to cover the breadth of framework functionality; these factors increase the cost of creation. As an alternative, this study has proposed that examples should play an important role as an integrated part of a pattern language. Here the benefit of examples, as concrete descriptions of behaviour, can be felt while the pattern helps to generalize their lesson to a wider audience.

An omission in the coverage of documentation investigated in this thesis has been a serious appraisal of UML. It was only looked at briefly using the high level model of JHotDraw that is provided with the framework. This provides little more that a starting point for further understanding and there is little evidence that developers found it critical in their understanding. But UML can be much more than a pithy overview. It can describe the classes, the methods and the static and dynamic information that is present within a system. UML has also been adapted and modified specifically for frameworks; UML-f (Fortuora, Pree and Rumpe 2000) attempts to show where parts of the framework can be modified and what alternatives exist to plug into the gaps. It was considered beyond the scope of this work to create a complete coverage of a framework with UML. However, the views provided by current reverse engineering tools e.g. Together (Borland 2005) do provide some idea of its utility. It would appear to offer the type of support that may be useful for functionality or interaction type problems but there are issues with the scale of diagrams and the dynamic nature of frameworks. UML diagrams that describe a framework can easily cover several pages making them unwieldy to manipulate and absorb. Framework understanding also requires a significant degree of dynamic information which is difficult to provide using UML. Polymorphism is employed extensively by frameworks to create flexibility but it creates problems for UML. For example, instead of one sequence diagram to describe the interactions of a part of the framework there might be several diagrams, one for every potential polymorphic substitution. These diagrams might in turn feature other examples of polymorphism each of which could be multiplied out creating a large number of diagrams to comprehend. Both of these problems, scale and dynamics, are serious inhibitors to the use of UML as framework documentation. Nevertheless, the prominent position of UML as a mechanism to describe object-oriented designs warrants further exploration in the future.

Tool support is another area that is worthy of future attention. Obvious advantages of tools are that they can present a number of different views of a framework and they can also handle a framework’s scale by providing support for searching and indexing. The downside of tool development is the cost and difficulty of producing the tools. There is a need to investigate the utility of current tools and to identify where tools can best contribute to framework comprehension. This research has provided some insight into the support that tools might offer framework developers which may serve as a starting point for further investigation and development.

Tools could offer useful support for mapping problems by leading a developer through the choices they have to make when instantiating an application. This could help novice users who may not be familiar enough with the framework, to ask relevant questions, in the correct order to produce a complete application (Tools such as FRED (Hakala et al. 1998) provide some support for this functionality). Such support is not as useful to an experienced developer who may find such guidance inflexible or redundant. Instead they may benefit more from tools which can automate repetitive or boring tasks within the frameworks (such as generating boilerplate code for common modifications) allowing them to concentrate more on the unique aspects of their modification.

Other tool support could use static and dynamic analysis to provide users with information about the call structure of their application and the relative frequency with which particular calls occur. Such information could be an important starting point for the construction of a mental model of a frameworks interaction. Existing tools can provide such information but it is often provided in overwhelming amounts. The challenge is to find ways to filter or otherwise reduce the volume of information presented so that it can be more easily absorbed by framework developers.

The potential of tool support for architecture and functionality problems is less obvious. To a certain extent architecture can be addressed by supporting the mapping problem but perhaps tools could be used to mine architectural information from the framework code. At a basic level this might provide insight into invariants which must be preserved across the framework but more ambitiously might help to provide insight into the original developer's intentions about a solution helping to guide the choice of modification. Functionality may receive some support from software visualizations tools which could reverse engineer views of the source code to present information in a more digestible manner, although it is not clear at present what nature such visualisations might take.

6.4. Conclusions

Object-oriented frameworks are a popular form of large scale reuse. They combine the benefits of reusable class libraries with software architectures to create a skeleton application that can be customised to suit a wide variety of circumstances. This utility does not come cheaply. Frameworks are large and feature significant interaction amongst their components. This makes them difficult to understand and to modify during reuse.

This thesis has argued that better documentation is the key to tackling this problem. The more information that can be effectively communicated to re-users about the construction and operation of a framework the better. This thesis has also argued that there are key requirements that documentation must meet in order to provide useful information. In particular it must help the developer to map solutions onto the framework code, to understand the functionality and interactions that exist within the framework implementation and to describe the overall architecture of the framework so that developers can plan appropriate modifications.

Currently little evaluation is performed of framework documentation to ensure that it meets developer needs. Ideally this thesis will convince others of the importance of careful evaluation of framework documentation. The evaluation of a pattern language and micro architecture based documentation shows how expectations are not always borne out in practice and also how evaluation can usefully feed back into development by identifying areas of weakness in the original documentation. It is vitally important for framework reuse that the documentation community begins to properly validate its work. Only through this approach can we produce a wider understanding of what documentation works and in which circumstances it ought to be used.

References

Alexander, Christopher. 1979. The Timeless Way of Building. New York: Oxford University Press.

Alexander, Christopher, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdhal-King and Shlomo Angel. 1977. A Pattern Language - Towns, Buildings, Construction. New York: Oxford University Press.

Apple. 2005. MacApp. http://developer.apple.com/tools/macapp/. (Accessed on 8 February 2005).

Bass, Len, Paul Clements and Rick Kazman. 1998. Software Architecture in Practice. Reading, MA: Addison Wesley.

Beck, Kent. 2000. Extreme Programming Explained: Embrace Change. Reading, MA: Addison Wesley.

Beck, Kent and Ward Cunningham. 2005. HotDraw. http://c2.com/cgi/wiki?HotDraw. (Accessed on 9 Februrary 2005).

Beck, Kent and Erich Gamma. 1999. JUnit A Cook's Tour. JavaReport 4(5). Available online at http://junit.sourceforge.net/doc/cookbook/cookbook.htm.

Beck, Kent and Ralph Johnson. 1994. Patterns Generate Architectures. In Proceedings of the 1994 European Conference on Object Oriented Programming held in Bologna, Italy, June, 1994, edited by M. Tokoro and R. Pareschi, 139-149. Springer.

Booch, Grady. 1994. Object Oriented Analysis and Design with Applications 2nd ed. Redwood City CA: Benjamin/Cummings Publishing Company.

Borchers, Jan. 2001. A Pattern Approach to Interaction Design. John Wiley and Sons.

Borland. 2005. Together. http://www.borland.com/together/. (Accessed on 10 February 2005).

Bosch, Jan, Peter Molin, Michael Mattsson, PerOlaf Bengtsson. 1999. Framework Problems and Experiences. In Building Application Frameworks: Object-Oriented Foundations of Framework Design, ed. M. E. Fayad, D. C. Schmidt and R. E. Johnson, 55 - 82. John Wiley and Sons.

Bosch, Jan. 2001. Design and Use of Software Architectures: adopting and evolving a product line approach. Harlow, UK: Addison Wesley/Pearson.

Brooks, Ruven. 1978. Using a Behavioural Theory of Program Comprehension in Software Engineering. In the Proceedings of the 3rd International Conference on Software

Engineering held in Atlanta, Georgia, USA. May 1978. 196-201. Piscataway, NJ: IEEE Press.

Brown, Kyle and Bruce Whitenack. 1996. Crossing Chasms: A Pattern Language for Object-RDBMS Integration. In Pattern Languages of Program Design Vol. 2, ed. J. Vlissides, J. O. Coplien and N. Kerth, 227-238. Reading, MA: Addison Wesley.

Buhr, Raymond J.A., 1996. Use Case Maps for Attributing Behaviour to System Architecture. In Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems held in Honolulu, Hawaii, USA, April, 1996. 3-11. Washington, DC: IEEE Computer Society.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. 1996. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley and Sons.

Butler, Greg and Pierre Dénommée. 1997. Documenting Frameworks to Assist Application Developers. In Eighth Workshop on Institutionalising Software Reuse. Available online at: http://www.umcs.maine.edu/~ftp/wisr/wisr8/papers/butler/butler.html (Accessed on 19 August 2005).

Butler, Greg, Rudolf K. Keller and Hafedh Milli. 2000. A framework for framework documentation. ACM Computing Surveys 32(1).

Cisco Systems. 2005. Cisco Element Manager System. http://www.cisco.com/en/US/products/sw/netmgtsw/ps829/. (Accessed on 8 February 2005).

Codenie, Wim, Koen De Hondt, Patrick Steyaert and Arlette Vercammen. 1997. From Custom Applications to Domain-Specific Frameworks. Communications of the ACM 40(10): 70-77.

Dénommée, Pierre. 1998. A Case Study in Documenting and Developing Frameworks. Master's Thesis, Concordia University, Canada.

Dey, Ian. 1993. Qualitative Data Analysis: A User-Friendly Guide for Social Scientists. London, UK: Routledge.

Eclipse. 2005. Eclipse. http://www.eclipse.org/. (Accessed on 11 February 2005).

Edwards Brian, 1972. Statistics for Business Students. Collins.

Fairbanks George. 2004. Software Engineering Environment Support for Frameworks: A position paper. In Workshop on Directions in Software Engineering Environments. Available online at: http://hdcp.org/Publications/WoDiSEE_ICSE04_Fairbanks.pdf (Accessed on 19 August 2005).

Fayad, Mohamed, Douglas C. Schmidt and Ralph E. Johnson. 1999. Building Application Frameworks: object oriented foundations of framework design. New York , NY, John Wiley and Sons.

Fayad, Mohamed, Douglas C. Schmidt. 1997. Object Oriented Application Frameworks. Communications of the ACM, 40(10):32-38.

Fontoura, Marcus, Wolfgang Pree and Bernhard Rumpe. 2000. UML-F: A Modelling Language for Object Oriented Frameworks. In Proceedings of 2000 European Conference on Object Oriented Programming held in Sophia Antipolis and Cannes, France, June, 2000, edited by E. Bertino, 63-82. Springer.

Fraser, Steven, Kent, Beck, Grady Booch, Jim Coplien, Ralph Johnson and Bill Opdyke. 1997. Beyond the Hype: Do patterns and frameworks reduce discovery costs? In Proceedings of 1997 conference on Object Oriented Programs, Languages and Applications held in Atlanta Georgia, USA, October 1997, 342 - 344, edited by A. Michael Berman, New York NY: ACM Press.

Froehlich, Garry, James Hoover, Ling Liu and Paul Sorenson. 1997. Hooking into Object Oriented Application Frameworks. In Proceedings of 1997 International Conference on Software Engineering held in Boston, Massachusetts, USA, May, 1997, 491-501. New York: ACM Press.

Gamma and Eggenschwiler 1998. JHotDraw. http://members.pingnet.ch/**gamma**/JHD-5.1.zip. (Accessed on 9 Februrary 2005).

Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides. 1994. Design Patterns: Elements of Reusable Object Oriented Software. Reading, MA: Addison Wesley.

Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In Proceedings of 1993 European Conference on Object Oriented Programming held in Kaiserslautern, Germany, July, 1993, edited by O. Nierstrasz. 406 - 431. London, UK: Springer-Verlag.

Gangopadhyay, Dipayan, Subrata Mitra. 1995. Understanding Frameworks by Exploration of Exemplars. In Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering held in Toronto, Ontario, Canada, July, 1995, 90-100. Washington, DC: IEEE Computer Society.

Graham Technology. 2005. What is GT-X? http://www.gtnet.com/site/Home/ProductsPage.htm. (Accessed on 8 February 2005).

Hakala, Markku, Juha Hautamäki, Jyrki Tuomi, Antti Viljamaa, Jukka Viljamaa, Kai Koskimies and Jukka Paakki. 1999. Managing Object Oriented Frameworks with Specialization Templates. In Proceedings of the Workshop on Object Oriented Technology held in Brussels, Belgium, July,1998, edited by S. Demeyer and J. Bosch, 199-209. London, UK: Springer-Verlag.

Helm, Richard, Ian M. Holland, Dipayan Gangopadhyay. 1990. Contracts: specifying behavioral compositions in object-oriented systems. In Proceedings of the 1990 European Conference on Object Oriented Programming held in Ottawa, Canada, October, 1990, edited by N. Meyrowitz, 169-180. New York: ACM Press.

Johnson, Ralph. 1992. Documenting Frameworks using Patterns. In Proceedings of the 1992 conference on Object Oriented Systems, Languages and Applications held in Vancouver, British Columbia, Canada, October, 1992, 63-76. New York: ACM Press.

Johnson, Ralph and Brian Foote. 1988. Designing Reusable Classes. Journal of Object Oriented Programming, 1(2):22-35.

Judd, Charles M., Elliot R. Smith, Louise H. Kidder. 1991. Research methods in social relations 6th ed. Fort Worth TX: Holt Rinehart and Winston.

Kaiser, Wolfram. 2005. JHotDraw as Open-Source Project. http://www.jhotdraw.org/. (Accessed on 8 February 2005).

Krasner, Glen E., and Stephen T. Pope. 1988. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. Journal of Object Oriented Programming 1 (3):26-49.

Krueger, Charles W., 1992. Software Reuse. ACM Computing Surveys 24(2): 131-183.

Lajoie, Richard. 1993. Using Reusing and Describing Object-Oriented Frameworks. Master's Thesis, McGill University, Canada.

Lajoie, Richard and Rudolph K. Keller. 1994. Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert. In the Proceedings of the Colloquium on Object Orientation in Databases and Software Engineering held in Montreal, Canada, May, 1994, edited by V. S. Algar and R. Missaoui, 295-312. River Edge, NJ: World Scientific.

Lange Beth. M. and Thomas G. Moher. 1989. Some Strategies of Reuse in an Object Oriented Programming Environment. In Proceedings of the conference on Human factors in computing systems held in Austin, TX, USA. 30 April-4 May 1989. Edited by K Bice and C Lewis. 69-73. New York, NY: ACM Press.

Lange, Danny B., and Yuchi Nakamura. 1995. Interactive visualization of design patterns can help in framework understanding. In Proceedings of the 1995 conference on Object Oriented Systems, Languages and Applications held in Austin, Texas, USA, October, 1995, 342-357. New York: ACM Press.

Mancl Dennis, William F. Opdyke and Steven D. Fraser. Tackling the Discovery Costs of Evolving Software Systems. In the Proceedings of the 2002 conference on Object Orientated Systems, Languages and Applications held in Seattle Washington, USA. November 2002. Page 83. New York, NY: ACM Publishing

Mattsson, Michael, Jan Bosch and Mohamed Fayad. 1999. Framework Integration, Problems, Causes, Solutions. Communications of the ACM, 42(10):80-87.

McIlroy, M. Doug. 1968. Mass produced software components. In the Proceedings of the NATO Software Engineering Conference held in Garmisch, Germany, October, 1968, edited by P. Naur and B. Randell, 138-155. NATO Science Committee.

Meszaros, Gerard and Jim Doble. 1998. A Pattern Language for Pattern Writing. In Pattern Languages of Program Design 3, ed. R. C. Martin, D. Riehle, F. Buschmann, 529-574. Reading, MA: Addison Wesley.

Meusel, Matthias, Krzysztof Czarnecki and Wolfgang Köpf. 1997. A Model for Structuring User Documentation of Object-Oriented Frameworks Using Patterns and Hypertext. In Proceedings of the 1997 European Conference on Object Oriented Programming held in Jyvaskyla, Finland, June, 1997, edited by M. Aksit and S. Matsuoka, 496-510. Springer-Verlag.

Meyer, Bertrand. 1997. Object Oriented Software Construction 2nd ed. Upper Saddle River, NJ: Prentice Hall.

Microsoft. 2005a. Overview of ADO.NET. http://msdn.microsoft.com/ library /default.asp?url=/ library/en-us/cpguide/html/cpconoverviewofadonet.asp. (Accessed on 8 Februrary 2005).

Microsoft. 2005b. Microsoft ASP.NET. http://www.asp.net/. (Accessed on 8 Februrary 2005).

Microsoft. 2005c. Microsoft .NET Framework: Windows Forms. http:// www. Windowsforms .net/. (Accessed on 8 February 2005).

Michail, Amir and David Notkin. 1998. Illustrating Object-Oriented Library Reuse by Example: A Tool-Based Approach. In Proceedings of the thirteenth conference on Automated Software Engineering held in Honolulu, Hawaii, October, 1998, 200-203. Washington, DC: IEEE Computer Society.

Miles, Matthew B. and A. Michael Huberman. 1994. Qualitative data analysis : an expanded sourcebook 2nd ed. Thousand Oaks, CA: Sage Publications

Moser, Simon and Oscar Nierstrasz. 1996. The Effect of Object Oriented Frameworks on Productivity. IEEE Computer 29(9): 45-51.

Netu2. 2005. MediaCam High Speed Screen Recording. http://www.netu2.co.uk/home.htm. (Accessed on 8 February 2005).

Odenthal, Georg and Klaus Quibeldly-Cirkel. 1997. Using Patterns for Design and Documentation. In the Proceedings of the 1997 European Conference on Object Oriented Programming held in Jyvaskyla, Finland, June, 1997, edited by M. Aksit and S. Matsuoka, 511-529. Springer-Verlag.

OMG. 2005a. Corba. http://www.corba.org/. (Accessed on 8 February 2005).

OMG. 2005b. The Unified Modelling Language. http://www.uml.org/ (Accessed 14 April 2005).

Ortigosa, Alvaro, Marcelo Campo and Roberto M. Salomon. 1999. Enhancing Framework Usability through Smart Documentation. In Proceedings of the 3rd Argentine Symposium on Object Orientation held in Buenos Aires, Argentina, September 1999, 103-117. SADIO.

Pree, Wolfgang. 1999. Hot Spot Driven Development. In Building Application Frameworks: Object-Oriented Foundations of Framework Design, ed. M. E. Fayad, D. C. Schmidt and R. E. Johnson, 379-394. John Wiley and Sons.

Pressman, Roger, S. 1994. Software Engineering: A practitioner's approach. 3rd ed. London, UK: McGraw Hill.

Richner, Tamar and Stéphane Ducasse. 1999. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. In Proceedings of the 1999 International Conference on Software Maintenance held in Oxford, England, UK, September 1999, 13-22. IEEE Computer Society.

Roberts, Don and Ralph E. Johnson. 1996. Evolving Frameworks: A Pattern-Language for Developing Object-Oriented Frameworks. In Proceedings of the 1996 Pattern Language of Programming Conference held in Illinois, Chicago, USA, September 1996, 471-486, Addison Wesley.

Robitaille, Sébastien, Rienhard Schauer and Rudolf K. Keller. 2000. Bridging Program Comprehension Tools by Design Navigation. In Proceedings of the 2000 International Conference on Software Maintenance held in San Jose, CA, USA, October 2000, 22-32. Washington, DC: IEEE Computer Society.

RoleModel Software. 1996. Drawletts. http:// www.rolemodelsoftware.com/ drawlets/ index.php. (Accessed on 9/ Februrary 2005).

Roper, Marc and Murray Wood. 2004. 52.440 Software Architecture and Design. https:// www.cis.strath.ac.uk/ teaching/ug/classes/52.440/. (Accessed on 11 February 2005).

Rosson Mary B. and John M. Carroll. 1996, The Reuse of Uses in Smalltalk Programming. Transactions on Computer-Human Interaction 3(3) 219-253.

Schmidt, Doug C., 2005. The Adaptive Communication Environment Framework. http:// www.cs.wustl.edu/ ~schmidt/ACE.html. (Accessed on 8 Februrary 2005).

Schneider, Kurt and Alexander Repenning. 1995. Deceived by Ease of Use: Using Paradigmatic Applications to Build Visual Design Environments. In Proceedings of the Symposium on Designing Interactive Systems: Processes, Practices, Methods and Techniques held in Ann Arbor, MI, USA, August, 1995, edited by G. M. Olsen and S. Schon, 177-188. New York: ACM Press.

SEI. 2005. Software Project Lines. http://www.sei.cmu.edu/productlines/index.html (Accessed on 19 July 2005).

Shull, Forrest, Filippo Lanubile and Victor R. Basili. 2000. Investigating Reading Techniques for Object-Oriented Framework Learning. IEEE Transactions on Software Engineering 26(11): 1101-1118.

Slaney, John and Sylvie Thiébaux. 1994. Adventures in Blocks World. Technical Report (TR-ARP-7-94). Research School of Information Sciences and Engineering and Centre for Information Science Research. Australian National University.

Somerville, Ian. 2001. Software Engineering. 6th ed. Harlow, UK: Addison Wesley/Pearson.

Sparks, Steve, Kevin Benner and Christopher A. Faris. 1996. Managing Object-Oriented Framework Reuse. IEEE Computer 29(9): 52-60.

Steyaert, Patrick, Carine Lucas, Kim Mens and Theo D'Hondt. 1996. Reuse Contracts: Managing the Evolution of Reusable Assets. In the Proceedings of the 1996 conference on Object Oriented Systems, Languages and Applications held in San Jose, CA, USA, October 1996, 268-285. New York: ACM Press.

Sun Microsystems. 2005a Core Java: JavaDoc Tool. http://java.sun.com/j2se/javadoc/. (Accessed on 7 February 2005).

Sun Microsystems. 2005b. Desktop Java: Java Foundation Classes (JFC/ Swing). http://java.sun.com/products/jdbc/. (Accessed on 8 February 2005).

Sun Microsystems. 2005c. J2EE: Java Active Server Pages. http:// java.sun.com/ products/ jsp/. (Accessed on 8 February 2005).

Sun Microsystems. 2005d. J2EE: JDBC Technology. http://java.sun.com/products/jdbc/. (Accessed on 8 Februrary 2005).

Van Grup, Jilles and Jan Bosch. 2001. Design Erosion: Problems and Causes. Journal of Systems and Software 61(2): 105–119.

Vlissides, John. 1990. UniDraw: A Framework for Building Domain-Specific Graphical Editors. ACM Transactions on Information Systems 8(3): 237-268.

White, Scott, Joshua O'Madadhain, Danyel Fisher and Yan-Biao Boey. 2005. JUNG: Java Universal Network/Graph Framework. http://jung.sourceforge.net/. (Accessed on 8 February 2005).

Wienand, Andre, Erich Gamma and Rudolf Marty. 1988. ET++ an object oriented application framework in C++. In the Proceedings of the 1988 conference on Object Oriented Systems, Languages and Applications held in San Diego, CA, USA, September, 1988, edited by N. K. Meyrowitz , 46-57. New York: ACM Press.

WindRiver. 2005. SNiFF+. http:// www.windriver.com/ products/ development_tools/ ide/ sniff_plus/. (Accessed on 11 February 2005).