Empir Software Eng (2007) 12:243–274
DOI 10.1007/s10664-006-9027-z

Identifying and addressing problems in object-oriented framework reuse

Douglas Kirk · Marc Roper · Murray Wood

Published online: 29 November 2006

Editor: Jeff Offutt

Abstract This paper describes the results of a long-term empirical investigation into object-oriented framework reuse. The aim is to identify the major problems that occur during framework reuse and the impact of current documentation techniques on these problems. Four major reuse problems are identified: understanding the functionality of framework components; understanding the interactions between framework components; understanding the mapping from the problem domain to the framework implementation; understanding the architectural assumptions in the framework design. Two forms of documentation are identified as having the potential to address these problems, namely pattern languages and micro-architecture descriptions. An in-depth, qualitative analysis suggests that, although pattern languages do provide useful support in terms of introducing framework concepts, this can be bypassed by developers using their previous knowledge, occasionally to the detriment of the final solution. Micro-architecture documentation appears to provide support for simple interaction and functionality queries, but it is not able to address large scale interaction problems involving multiple classes within the framework. The paper concludes that, although a combination of pattern language and micro-architecture documentation is useful for framework reuse, the forms of these documentation types used in this study require further enhancement to become effective. The paper also serves as an example to encourage others to perform evaluation of framework understanding and documentation.

Keywords Object-oriented frameworks · Documentation · Empirical study · Software comprehension · Qualitative study · Framework reuse

D. Kirk · M. Roper · M. Wood (✉)
Department of Computer and Information Sciences, University of Strathclyde,
Livingstone Tower, 26 Richmond St., Glasgow G1 1XH, UK
e-mail: Murray.Wood@cis.strath.ac.uk

D. Kirk
e-mail: Douglas.Kirk@cis.strath.ac.uk

M. Roper
e-mail: Marc.Roper@cis.strath.ac.uk

1 Introduction

Object-oriented frameworks are promoted by both academics and industrialists (Buschmann et al. 1996; Fayad et al. 1999; Gamma et al. 1994) as having the potential to provide the benefits of large-scale software reuse. Whilst practical evidence does suggest that framework usage can increase reusability and decrease development effort (Moser and Nierstrasz 1996), experience has identified a number of issues that complicate framework application and limit potential benefits (Bosch et al. 1999). One of the major challenges is effective framework understanding (Butler et al. 2000). This paper presents the details of a long-term empirical investigation into two related aspects of framework understanding—identification of the key problems that arise during framework reuse and evaluation of documentation that aims to address these key problems.

Object-oriented frameworks enable large-scale reuse by capturing an abstract design and the core implementation for a domain from which a range of specific applications can then be derived. A framework defines the overall software architecture—the classes and objects, their responsibilities, their collaborations and the threads of control—for a group of related applications. A major obstacle to framework reuse is the potential amount of material that must be understood before a framework can be successfully instantiated. Many frameworks contain hundreds or thousands of classes and are often incomplete, containing abstract classes and using design patterns to create flexibility. In addition, developers often have to learn the language of the framework's domain which describes the abstractions and range of functionality available (Bosch et al. 1999).

In recent years application frameworks have emerged that address a wide range of domains, for example network communications (Schmidt 2005), graph modelling (White et al. 2005) drawing editors (Gamma and Eggenschwiler 2005), middleware, e.g. CORBA (OMG 2005a), and program development environments, e.g. Eclipse (Eclipse 2005). The widespread adoption of the object-oriented languages Java and C# has also promoted the use of frameworks that address common programming activities, e.g. JDBC (Sun Microsystems 2005a), Swing/AWT (Sun Microsystems 2005b) and ASP.NET (Microsoft 2005).

Over the past decade a large range of candidate techniques have been proposed for framework documentation including source code browsers (Robitaille et al. 2000), Javadoc (Sun Microsystems 2005c), UML diagrams (OMG 2005b), design patterns (Gamma et al. 1994), pattern languages (Johnson 1992; Lajoie and Keller 1994), and example based learning (Shull et al. 2000). The research reported here was motivated by the lack of investigation into the use of these techniques and, in particular, by the lack of investigation into the key challenges in reusing frameworks. That is, what aspects of framework reuse should documentation specifically address? As a result, researchers have no common understanding about where to best devote their research effort when attempting to address framework reuse problems. More critically, it is not clear what techniques should be used in practice to document frameworks.

The paper starts by reviewing related work on understanding and documenting object-oriented frameworks. The lack of evaluation highlighted here motivates the rest of the paper. Section 3 describes the initial empirical study which was designed to identify the key problems that can arise during framework reuse. A major contribution of the paper is the resulting identification of four problem categories: mapping, understanding architecture, understanding interactions and understanding functionality. Section 4 then revisits the literature and discusses the limited, existing evidence for the identified problem categories.

To try to address these problem categories, two adapted forms of framework documentation are proposed in Section 5, namely a pattern language and a set of micro-architectures. This section also describes the second key contribution of the paper, an in-depth, qualitative study that evaluates the strengths and weakness of the proposed documentation forms in addressing the identified four problem categories. The results suggest that pattern languages can help introduce key framework concepts, although this form of help may be bypassed when developers try to use their previous experience, sometimes to the detriment of the final solution. On the other hand, micro-architecture based documentation appears to provide support for simple interaction and functionality problems, but it is not able to address the larger scale interaction problems involving multiple classes across the framework. The paper concludes by arguing that, though a combination of pattern language and micro-architecture based documentation is potentially useful, both require further refinement and evaluation to effectively address the full range of framework reuse problems that may occur in practice.

2 Related Work

There are many different types of tools and documentation that are claimed to provide support for framework reuse. However, there is a paucity of evidence which evaluates their utility and justifies their selection as framework documentation. This section overviews the range of documentation techniques that has been advocated to support framework understanding, and it highlights any evidence that is available to support these techniques.

2.1 Source Code

Lajoie and Keller emphasise the importance of source code in framework comprehension (Lajoie and Keller 1994) claiming that source code is the ultimate reference for framework knowledge. A framework is defined by its source code. However, the whole point of framework documentation is to remove the developer's reliance on understanding the overwhelming detail typically associated with source code. Lajoie and Keller 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. Tools such as Together (Together 2005) and Eclipse (Eclipse 2005) can help developers to navigate through large amounts of source code and to quickly locate relevant information. Javadoc (Sun Microsystems 2005c) can also help with navigation. Such tools appear to help the developer to gain an understanding of the static structure of the framework. However, understanding the dynamic behaviour of a framework is more challenging, particularly given the separation of the static and dynamic perspectives in the object-oriented paradigm (Gamma et al. 1994).

2.2 Contracts

Helm et al. describe a structure called a contract which formally describes the communication protocol between classes (Helm et al. 1990). They claim their approach has specific relevance to frameworks commenting, "*Frameworks define solutions in terms of interaction between abstract classes*". Lajoie and Keller similarly believe that describing

key interactions is a requirement of framework documentation (Lajoie and Keller 1994). They decompose framework interactions into units they call micro-architectures, codifications of design knowledge in terms of the behaviour of object collaborations, which are described using a combination of design patterns (Gamma et al. 1994) and contracts. Lajoie and Keller claim that understanding the framework in terms of such groupings is the principal difference that distinguishes novice framework re-users from more experienced developers. By decomposing the framework into smaller subsections, micro-architectures may facilitate the comprehension of the entire framework as each section can be understood in isolation from the others. One potential weakness of the micro-architecture approaches is the lack of formal guidance to help identify the key interactions that they describe.

2.3 Examples

The capabilities of frameworks are often illustrated through a number of example applications. Johnson claims 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 (Johnson 1992). Schneider and Repenning claim that a specific form of example application, called a paradigmatic application, should be used to illustrate the important underlying mechanisms of a framework (Schneider and Repenning 1995). 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 and support their approach with a tool, called Objchart, which displays class diagram and sequence diagram information for an executing exemplar.

Cookbooks are compilations of examples that are structured around common reuse tasks within the framework (Krasner and Pope 1988). They differ from examples by not describing complete applications, instead featuring small fragments of code, and by providing an explanation of the purpose of the code and an indication of when it should be used.

Shull et al. appear to be one of the few groups that have investigated framework understanding to date (Shull et al. 2000). They present evidence that suggests that examples can be used as an effective framework learning strategy, especially for beginning to learn a framework. However, they also report that their study subjects occasionally had problems finding functionality of interest within the examples, and that example based documentation may prevent developers from going beyond the functionality that is presented therein.

2.4 Pattern Languages

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). Pattern languages present a decomposition of the design problem into a collection of sub-problems that can be tackled independently. Each sub-problem is described by a "pattern" that identifies the forces (constraints) that must be considered in solving the sub-problem and proposes a solution that resolves these forces. A pattern language takes the form of a general graph structure with relationships between patterns including refinement, composition and alternatives. Pattern languages consist of a mixture of structured narrative, diagrams and, in the software engineering domain, fragments of code or design.

In the case of frameworks, a pattern language aims to interweave a system of patterns into an explicit route-map through the framework architecture. Johnson was the first person to identify the potential of pattern languages as effective documentation for frameworks (Johnson 1992). Johnson identifies 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 pattern languages effectively address how to use the framework and that they can be extended to address all three of these issues. He demonstrates this with an example pattern language for HotDraw (Beck and Cunningham 2005), a Smalltalk framework for creating semantic drawing editors. Beck and Johnson also use a collection of patterns to explain the architecture of a framework and to justify its implementation (Beck and Johnson 1994). They argue that other documentation types focus too much on the *how* and not enough on the *why* of design. Beck and Johnson claim that novice users can use their pattern based documentation as a replacement for experience, helping them to understand and maintain the architectural relationships within the framework.

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. Meusel, Czarnecki and Köpf also extend Johnson's work by proposing a more algorithmic format for the patterns in a pattern language (Meusel et al. 1997). They propose a three-layered language that aims to address Johnson's three framework reuse categories (purpose, how to use, and detailed design). Finally, Froehlich et al. 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. These appear related to the idea of framework hotspots (Pree 1999) which are the areas of flexibility and customisation within a framework.

Some researchers have also attempted to encapsulate a pattern language within a tool. Both the FRED (Hakala et al. 1999) and SmartDoc (Ortigosa et al. 1999) tools present guidance on what modifications are possible based on the current state of the application, and they may even automate some of these. A potential concern is that such tools distance the re-user from the process of making a modification and may result in a poor understanding of the framework code, especially if the re-user has to make a change outside the scope of the tool.

2.5 A Framework Documentation 'Framework'

Butler et al. have proposed a 'framework' for framework documentation (Butler et al. 2000). This is derived from a theoretical analysis of a range of potential framework uses from simple instantiation through to, "... *mining a framework for design ideas to use in other frameworks*". For each use they identified a set of elementary software engineering tasks, each with its own documentation requirements (based on the specification of, and dependencies between, components). They then briefly assess the support provided by example applications, cookbooks, contracts, design patterns, framework overviews and reference manuals for each of these requirements. Their theoretical analysis suggests that no documentation technique explicitly addresses all of the requirements though cookbooks can provide implicit support for them all.

2.6 Summary

This review of related work shows that there has been a large range of documentation techniques proposed for frameworks over the last decade. The main observation is that there is a lack of agreement on the types of problems that occur during framework reuse and the documentation strategies to address them. To properly evaluate framework documentation types it was therefore important to firstly investigate the problems that actually arise during real framework reuse—this is the aim of the first study below. The second observation is that there has been very little empirical evaluation of framework documentation types, only that of Shull et al. The second study described in this paper aims to address this using the framework reuse problems that are identified from the first study.

3 Study 1: Identifying the Problems of Framework Reuse

The goal of this first study was to investigate actual framework reuse in order to identify and characterise the problems that framework documentation must address. This was done by recording and categorising the problems that developers face when using a framework.

3.1 Experimental Design

Evidence of framework reuse problems was collected from three different scenarios: an individual developer (the first author), a class of software architecture students and a group of project students. Each group performed a framework reuse task, and a variety of data collection techniques were used to capture the reuse problems they experienced. The tasks all used a single application framework, JHotDraw (Gamma and Eggenschwiler 2005).

The JHotDraw framework is a Java implementation of the Smalltalk framework originally developed by Beck and Cunningham (Beck and Cunningham 2005) and is a good example of a mature, well designed, and well documented object-oriented framework that is widely used as a case study in the research literature. JHotDraw comprises approximately 120 classes and has been designed for the creation of semantic drawing editors which support the creation of geometric and user defined shapes, editing these shapes, creating behavioural constraints in the editor, and animation of the drawing contents.

JHotDraw is supported by a range of documentation forms, including: the framework source code, Javadoc listings, a set of design pattlets (brief design pattern-like descriptions), a class diagram showing the high level design of JHotDraw, and a set of four example applications. This documentation was augmented in the two student studies with additional material created by the authors: a pattern language, a set of practical exercises which teach key parts of the framework, and the mentoring and teacher support that was offered during the software architecture class.

The pattern language was modified from an existing pattern language (Johnson 1992) that addressed the Smalltalk implementation of HotDraw. Johnson's pattern language was the first to be applied to software frameworks and mimics the work of Alexander (Alexander et al. 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 (see Fig. 1). The complete language is available in previous work by Kirk (Kirk 2005).

3.2 The Three Studies

The individual developer 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, connect them together using gravity relationships, and then animate the model to show planets moving through their respective orbits.

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 77 members of the class participated in the study. The students recreated the Orrery application from the individual developer 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. After this the students were challenged to produce a suitable modification to the Orrery as part of their class assessment. This proved to be an effective approach with the students providing a wide range of imaginative alterations, e.g. dynamic lighting caused by a sun, creating black holes, and adding rockets that flew between planets. 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 projects based on the use of JHotDraw (they also attended the software architecture class). The

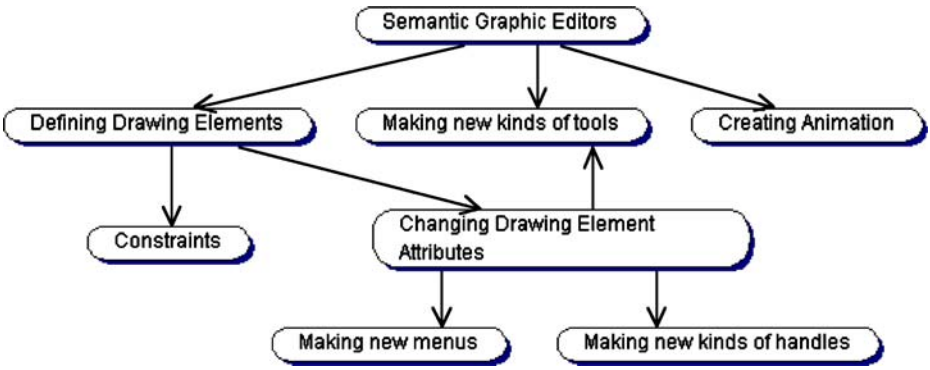

Fig. 1 Overview of the first pattern language. *Arrows* represent directed links between the patterns

students' experiences were collected in interviews that occurred at the end of their projects. Each project lasted approximately 6 months (part-time). The projects consisted of a golf hole designer, a UML class diagram editor, a model railway track editor and simulator, and a Scalextric™ racetrack editor and simulator.

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 Honours student or recent graduate. In each case the participants were given 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.

3.3 Data Collection

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

3.3.1 Individual Developer Study

The study was the developer's first exposure to the framework and took approximately 80 h to complete. 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.

3.3.2 Software Architecture Students

A 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. 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).

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.

3.3.3 Project Students

The four project students were invited to take part in an interview at the end of their application development. The interviews were voluntary, they followed university ethics procedures, and they were carried out by a researcher who was completely independent from the project assessment process. The 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 interviews were tape-recorded and were later transcribed into a written account.

3.4 Threats to Validity

Threats to validity are factors that limit our ability to interpret or draw conclusions from the study's data (Perry et al. 2000). In this study there are two types of validity that must be considered—internal and external validity. Threats to internal validity are those issues that cause the conclusions drawn from the data to be questioned (Campbell and Stanley 1963) and therefore compromise the findings of the study. Threats to external validity are issues that call into question the applicability of the conclusions to other environments and may therefore limit the generalisation of the results to the wider population of object-oriented frameworks.

3.4.1 Internal Threats

- **Time between problems occurring and being recorded:** The coursework reports and the interviews occurred after the development had been completed, and memories had perhaps degraded. In contrast, techniques like the newsgroup and the logbook resulted in problems being captured much closer to their occurrence in the task.
- **Problem identification relies on the experience of the analyst:** Recognising a problem from a developer account and extracting information relevant to its comprehension, and eventual classification, is dependant upon the judgement of the researcher. There is a risk that the inexperience of the researcher 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. This was partly because of the amount of data that had to be processed (more than 800 pages of A4 text), but it was also due to 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.

3.4.2 External Threats

- **Results are limited to one framework:** The decision to use a single framework for this study was deliberate. This was due to practical considerations regarding the need for the

researchers to develop a depth of knowledge and experience with the framework, and also because any framework studied would require a range of documentation types. This necessarily limited the scope of the study and suggests that the results produced might only be applicable to other uses of JHotDraw. While this is recognised as a threat, it is argued that the general challenges in learning to use object-oriented frameworks are likely to be similar. To help ensure this, the study focused on the identification of course-grained reuse problems which, arguably, have more chance of being generally applicable to a range of frameworks.

- **JHotDraw may not be representative of industrial frameworks:** This threat is partially addressed by the fact that JHotDraw has been designed by experienced framework designers, having evolved from an earlier well respected framework (Beck and Cunningham 2005), and it continues to be actively developed (Kaiser 2005).
- **Participants may not perform in a way that is representative of industrial developers:** As it is difficult to gain access to real world developers for studies such as this the participants were all drawn from an academic background. It is recognised that use of students in empirical software engineering studies is controversial (Harrison 2000) with some authors arguing that, in certain contexts, the performance of experienced students is comparable to that of developers in industry (Höst et al. 2000). Whilst the use of students is recognised as a threat here, there is also an argument that the experienced participants used in this study are of a similar capability to the developers who may be expected to make use of object-oriented frameworks in an industrial context.

3.5 Data Analysis

Together the three studies collected a significant 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. The project student interviews totalled a further 33 pages of text. Instances of framework reuse problems were identified by analysing all of this text, trying to identify any description that could possibly relate to a framework reuse problem, and extracting the verbatim description. 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. This information was overwhelming in both its detail and volume, making it challenging to draw useful lessons. In order to be useful the data had to be reduced and concentrated to identify the general types of problems that occurred during reuse.

A manual data clustering process was used to reduce the specific problem descriptions into their generic types. Cluster analysis (Miles and Huberman 1994) is a technique in which data is positioned on a grid based on its relationship to other items. Clusters emerge through an iterative process of merging and splitting as common properties are discovered amongst the data. Ideally, after a period of time, the clusters should form a stable representation of the data. This manual clustering is differentiated from automatic statistical approaches to clustering as it attempts to address patterns within the semantics of the data rather than focusing on syntactic similarities.

The 209 problem descriptions were numbered and then drawn randomly (to prevent any ordering effect) and assigned to a position on a whiteboard. The clustering was

performed as a group activity involving the three authors. Each verbatim description was 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 description was actually a composition of a number of smaller problems and the problem would be spilt apart and assigned separately.

When clustering data an important decision is when to stop the process. As larger clusters are formed they necessarily become more general as they represent a wider range of problems. This increase in generality was a useful terminating condition for the analysis. The clustering was judged to be finished when the problem categories met two criteria: firstly, they were at a level where they did not appear to be specific to JHotDraw and could plausibly apply to other frameworks, and secondly, their granularity was such that they could be targeted by a small set of high level documentation types.

After processing about half of the problem descriptions the clusters were reduced down to a main set of five categories (functionality, searching, interaction, mapping and architecture), which was further reduced to four as the searching cluster was recognised as a mixture of problems from the interaction 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 problem descriptions were analysed to ensure consistency with the existing clusters.

3.6 Results

At the end of the cluster analysis from a total of 209 problem descriptions, the *mapping* category accounted for 38 problems, *interaction* 48 problems, *functionality* 60 problems and *architecture* 17 problems. 46 problems were viewed as 'non-problems' because they did not describe a problem associated with framework reuse, or they lacked sufficient detail to classify accurately.

3.6.1 Mapping

The mapping category is the problem of fitting an abstract, conceptual solution into a concrete implementation which makes use of the structures and constraints defined in the framework. Such problems are typically expressed as, "*What should I use to represent...?*" or, "*How do I achieve...?*" Mapping problems occur when a developer has difficulties expressing a solution using the features available within the framework. In such cases there is often a mismatch in perspective, or granularity, between the functionality that is offered by a framework and the functionality that a developer expects to be there.

An example of a mapping problem was demonstrated in the individual developer study when creating a tool to make rectangles, "*How do I create a rectangle? Why is there no Rectangle tool?*" 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. 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 used the prototype design pattern to specialise its behaviour for particular figure types.

3.6.2 Interactions

Understanding interactions focuses on problems concerning the communication between classes in the framework, "*What happens if...?*" or, "*Where should I place...?*" Such problems are significant because of hidden or subtle dependencies within the framework, often due to inheritance, run-time binding, or inversion of control (a characteristic of object-oriented frameworks, where the framework retains control of when and how user code is called (Gamma et al. 1994)). Interaction problems often manifest themselves through uncertainty about where to place modifications within the call graph of the framework. Usually these problems are dynamic in nature and so they are not spotted until late in development when the modified application is executed.

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 a student 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*." (from the 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.

3.6.3 Functionality

Understanding functionality describes problems in determining what specific parts of the framework actually do. Manifestations of this problem include, "*How does ... work?*", *"What does ... do?*" or, "*Where is ... defined/created/called?*" In part this is the familiar problem of understanding program source code. Functionality problems tend to result in developers overlooking existing functionality in the framework, or attempting to use it inappropriately.

An example from the individual developer logbook 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()*." Initially this seemed straightforward as the standard drawing class supports a centre method which returns the point at the centre of the drawing. When the developer used this method it returned (0,0), despite the fact that the drawing appeared clearly visible in the user interface. Eventually, after studying the code in detail, the developer realised that what was actually visible in the interface was the drawing view, and that the length and breadth of the actual drawing were both still zero.

3.6.4 Architecture

Understanding the framework architecture is the problem of making modifications which are consistent with 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. Architectural problems are caused by a lack of understanding of the high level design of the framework. They formed the smallest cluster of problems identified in this study, perhaps because they are less common and, arguably, because architectural

mismatches tend to manifest themselves in the longer term beyond the timescale of this investigation.

A common architectural mistake made by the students during the coursework exercise was to provide figures with a reference to the drawing or the drawing view, "*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 design pattern to link the drawing (and via this the drawing view) to the figures. The use of 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, or the drawing view, 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).

4 The Significance of the Framework Reuse Problems

With hindsight, the four problem categories that have been identified may appear obvious. However, it is *not* clear that previous research proposing framework documentation techniques has recognised the need to address them. Existing documentation approaches only partially address these issues providing limited support for framework re-users and requiring a combination of techniques to be deployed. However, despite the lack of investigation into framework reuse problems, there is some discussion in the literature that provides limited confirmation of the identified problem categories.

The strongest evidence for the mapping problem can be found in the work of Johnson (Johnson 1992). He argues that documentation for framework reuse should address *how to use* the framework. Mapping is closely related to this quality. Johnson argues that explaining how to use a framework requires the communication of the purpose of the framework and its individual parts. The mapping problem is also related to the problem Bosch et al. identify as learning the language of the framework's domain (Bosch et al. 1999). Further support for the mapping problem is provided in an analysis of framework programming environment support by Fairbanks (Fairbanks 2004). He describes a number of questions that developers 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 constrained by the existing structure of the framework.

Johnson's work also provides some evidence for the interaction and functionality problems. He argues that framework 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. Butler et al.'s taxonomy (Butler et al. 2000) of framework documentation primitives also appears relevant to the problems of interaction and

functionality. 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. Fairbanks also suggests that programmers ask, "*Have I done all I need to do?*" when making a modification and, "*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 to the architectural problems identified in this study. The authors cite several reasons for this 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.

As Table 1 shows, the relative frequency of each of the problem types suggests that functionality problems are experienced most during reuse, closely followed by interaction and then mapping, with only a relatively small number of architecture problems occurring. However, 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 in 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.

5 Study 2: Addressing the Problems of Framework Reuse

The goal of the second study was to investigate in detail the support provided for framework reuse by two particular forms of framework documentation: pattern languages and micro-architecture descriptions. Specifically, pattern languages were hypothesised as having the potential to provide support for mapping and architectural problems, whilst micro-architecture documentation was hypothesised as having the potential to address

Table 1 Relative frequency of the four problem types in the first study

Functionality	37%
Interaction	30%
Mapping	23%
Architecture	10%

functionality and interaction problems. This study was also used to further investigate the framework reuse problem categories that were identified as a result of the first study.

5.1 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. Of all the candidate documentation types available, pattern languages stood out as having the potential to address mapping related issues. Pattern languages aim to describe how to solve the important design problems that occur when using a framework. Each pattern description addresses a key problem in the framework domain, describing the context of the problem, identifying the constraints that must be considered in solving the problem and providing a solution (or solutions) that resolves those constraints. Each pattern also includes a number of links to alternative or related pattern descriptions. Furthermore, Beck and Johnson have previously argued that pattern languages can help explain the framework architecture (Beck and Johnson 1994).

The only alternative that was considered realistic for addressing mapping problems in our study was a set of example applications integrated with a narrative that explained the context and purpose of the example in the style of Schneider and Repenning (Schneider and Repenning 1995) or Gangopadhyay and Mitra (Gangopadhyay and Mitra 1995). However, it was decided to take advantage of the existing pattern language and to strengthen it based on the experience of its use in the first study, including the integration of more examples, rather than attempt to develop and describe a full set of example applications.

The pattern language created for the first study was derived from Johnson's pattern language for a Smalltalk implementation of HotDraw (Johnson 1992). The use of a pre-existing structure allowed the pattern language to be created quickly for JHotDraw, but differences in the functionality of the original Smalltalk version and the Java version meant that the new pattern language did not fully describe the unique aspects of the JHotDraw system. Experience from the first study suggested that the first pattern language was a useful introduction to the framework, but it 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 pattern description, and in the number of relationships between patterns in the language. Improvements were also suggested from the pattern language literature. These included the addition of source code examples (Lajoie 1993) and UML class hierarchies (Meusel et al. 1997).

The pattern language was improved in three different dimensions:

- **Increased number of patterns:** The number of patterns in the second language was increased from 8 to 18 (see Fig. 2). 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, e.g. creating handles and locators.

- **Detail was added to each pattern:** The detail of each pattern was also improved (see Fig. 3). 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. 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 used as a cut and paste solution. The patterns were augmented with the addition of six class hierarchies (Figures, Tools, Locators, Handles, Painters and Connectors—see the grey boxes in Fig. 2). 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:** In contrast to the revised pattern language, the original language contained only eight links between its patterns, and only one pattern was reachable from multiple areas of the language (compare Figs. 1 and 2). As the language grew it became more desirable to create different paths through the network of patterns. Multiple paths allow developers to find new ideas that are related to or contrast with the current pattern.

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

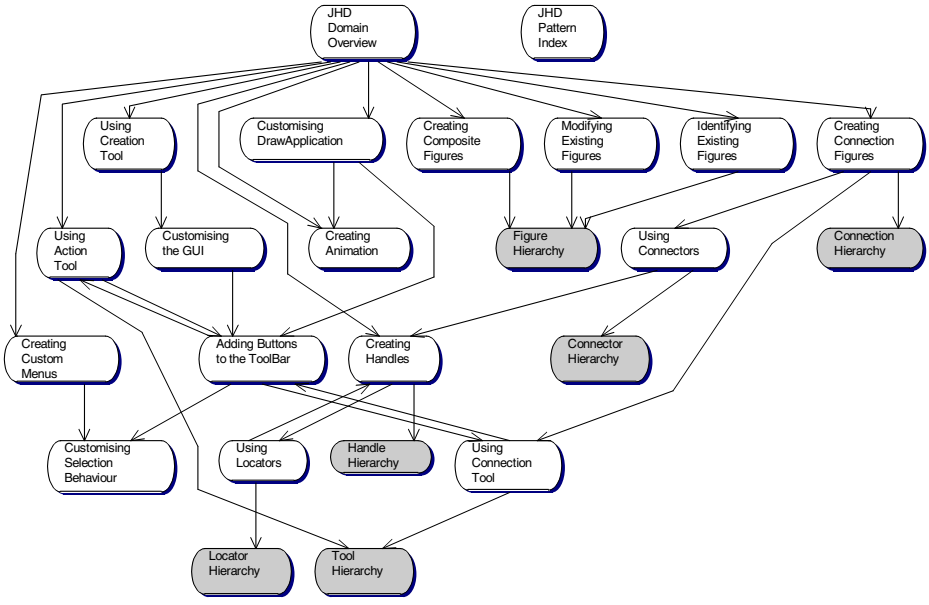

Fig. 2 Overview of the revised pattern language. The *grey boxes* are UML diagrams. *Arrows* represent directed links between the patterns

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.

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

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 overridden. 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]

Fig. 3 Excerpt from the revised pattern language

the inclusion of class hierarchy information. A complete version of this refined pattern languge can be found in previous work by Kirk (Kirk 2005).

5.2 The Micro-Architecture Documentation

Many of the existing documentation techniques could be used to address the problems of understanding interaction and functionality. However, experience from the first study suggested that techniques such as Javadoc, design patterns and UML only partially address these problems. In particular, there appears to be a trade-off between the depth and breadth of coverage that such documentation can provide. Javadoc, for example, can address all of a system but, depending on the nature of embedded comments, tends to provide a cursory description of its functionality, and it 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 they don't typically provide this coverage over the entire framework. UML provides breadth and depth of coverage at a higher level of abstraction than source code, but it can still be overwhelming in its detail, and therefore make it difficult for users to focus in on key areas of interest such as particular framework interactions.

A potential solution to this problem was derived from the idea of micro-architectures (Lajoie and Keller 1994), which document key aspects of design knowledge in terms of the behaviour of object collaborations. The idea was to decompose the framework into a number of smaller subsystems, based on the key interactions in the system, and to document each of them in isolation. The challenge then was how to identify and describe these important subsystems.

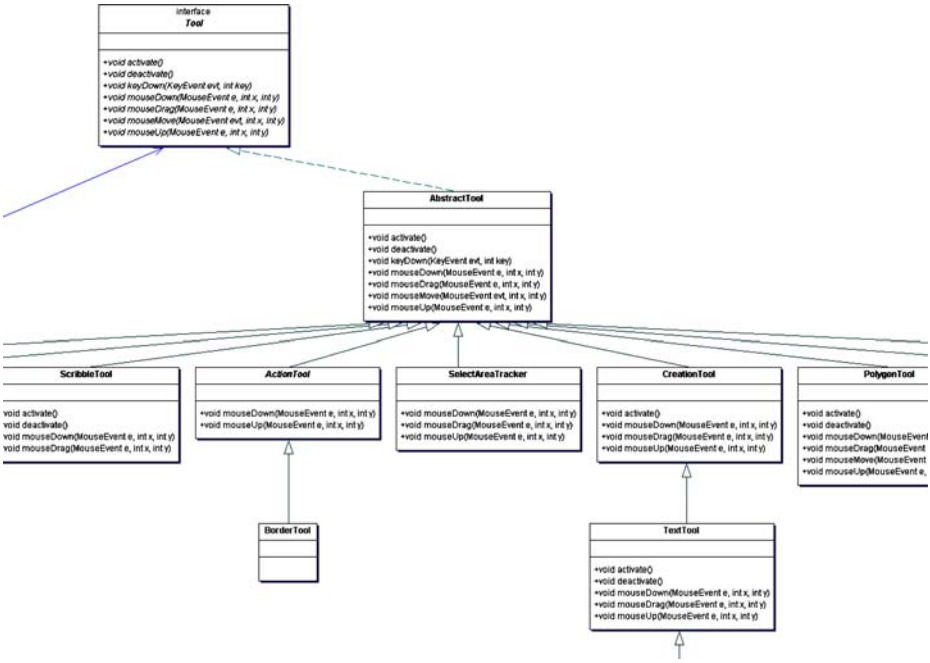

Fig. 4 A partial section of the tool micro-architecture hierarchy view

Initial candidates to document micro-architectures were classical design patterns (Gamma et al. 1994) or the use of a notation similar to Lajoie and Keller's. However, experience clearly demonstrated that many of the key interactions in JHotDraw did not correspond to design patterns as defined in Gamma et al. The published work by Lajoie and Keller provides only a limited description of their documentation style and the mechanisms for identifying micro-architectures. It was therefore decided to develop a micro-architecture based documentation comprising three views: an interface view, a call graph view, and a hierarchy view. The interface view displayed the methods available in each interface within the framework (see for example the Tool interface at the root of Fig. 4). The call graph view (Fig. 5) shows the call sequences within the framework that result in a call to a method from a given interface. The hierarchy view (part of which can be seen in Fig. 4) shows which classes implement which methods for each interface.

These separate views are joined together by an index which lists all of the available interfaces in the framework. The technique assumes the availability of the framework source code to provide descriptions of the functionality presented by callers or implementations of an interface. Html links between the views allow navigation from one source of information to another. The index is linked 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 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 parts were represented on the graph using nodes of different colours: green for input (PaletteListener, DrawingEditor and UserInput in Fig. 5), blue for interface methods (activate() in Fig. 5), and red for intermediate methods (the seven other methods in the middle of Fig. 5). Arrows indicate a method invocation and each intermediate 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 description for the named interface and continue to follow the call from there. The complete set of micro-

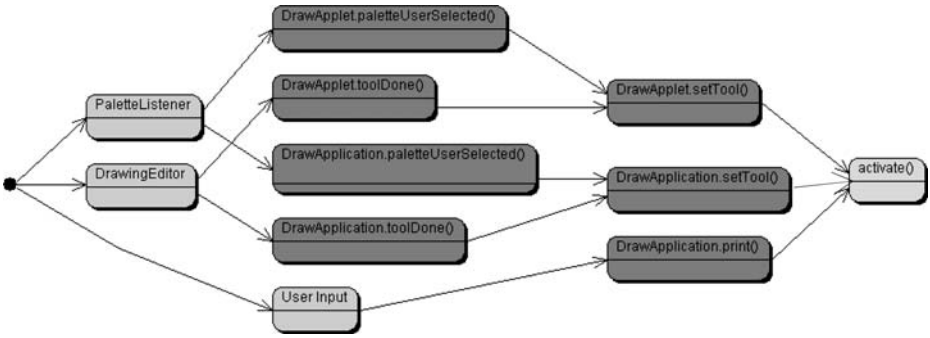

Fig. 5 The tool.activate() call graph view. *Arrows* indicate method invocation

architecture diagrams consists of 19 interfaces, 19 hierarchies and 171 call graphs and is available in previous work by Kirk (Kirk 2005).

5.3 Experimental Design

To investigate these new forms of documentation an exploratory, qualitative study was designed and performed. 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) has 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.

5.3.1 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 community, three final year undergraduate students who were using JHotDraw in their final year projects (these were different subjects from the first study discussed in this paper), and to the teaching staff of Strathclyde's software architecture class (the second and third authors) which teaches framework development using JHotDraw. 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 population from which subjects were selected represented a cross section of experience with the framework. The undergraduate students all represented experienced developers with the framework as they had each spent the past 6 months using it to achieve sizable individual projects. The postgraduates represented a wider spectrum of experience, as some of them had no experience of the framework at all while others had previously used JHotDraw as undergraduates in the software architecture class. Finally, the software architecture class lecturers had a good familiarity with the framework domain and its design, but they did not have a lot of experience developing solutions with it.

From a postgraduate population of around 30 students three agreed to participate in the study. From the three undergraduates approached, two agreed to take part, and both of the staff involved in the software architecture class also agreed to participate. As before, the study followed university ethics procedures and was carried out by a researcher who was independent from all assessment procedures associated with the undergraduate students. The seven individuals who agreed to take part are from here on referred to as participants A through G.

5.3.2 Reuse Task

The task that was chosen for this study was to recreate a model of a Blocks World in JHotDraw (Fig. 6). 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:

- 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. Blocks could be moved only if they did not have another block on top of them.

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. 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 mimic real application development. The first participant in the study, Participant A, was given the choice to code or design the solution and nominated to develop code. This significantly increased the time required to produce a solution and also raised a concern that less important implementation issues were dominating the task. As a result, other participants were not given the option of coding a solution but instead were required to articulate them verbally.

Participants also took part in two interviews (one before and one after the task) to capture their background knowledge of this domain and their reactions to the documentation. They were also given a period prior to the study (no longer than 30 min) where they could familiarise themselves with the experimental documentation using a short tutorial. Altogether the participants (except A) were given approximately 3 h to perform the task and the related activities.

5.3.3 Data Capture

The data collected consisted of the documentation accessed during the task and the participant's plans and actions in response to that documentation. Documentation accesses were directly observable and therefore easy to capture. However, gathering information about a participant's thought process was more difficult. A talk aloud protocol was used 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. As 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 participant'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 study.

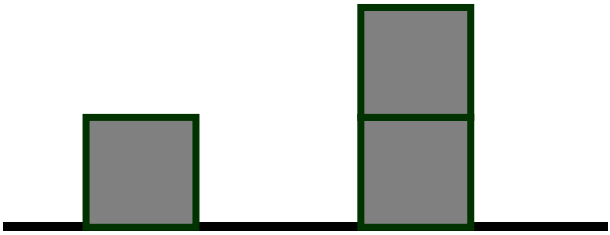

Fig. 6 An example blocks world

5.3.4 Plan of Analysis

Although there is no single approach to qualitative analysis recommended by the literature, there is common agreement on the key activities to perform. 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 (Dey 1993; Judd et al. 1991; Miles and Huberman 1994).

Data was transcribed for each participant into a textual narrative that described their reuse attempt. This made analysis easier and helped preserve the anonymity of the participant. The narratives were then read to identify what solutions had been proposed for each section of the task. 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 had provided useful information. This information was then explored further for any significant patterns which could characterise the use of the documentation.

5.4 Threats to Validity

As in the first study, the two main sources of validity threats were internal and external.

5.4.1 Internal Threats

- **Unfamiliar documentation:** There was a risk that participants would shy away from the new forms of documentation because of unfamiliarity. To limit this, participants were provided with a period at the start of the study to familiarise themselves with the new documentation. 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 participant but it meant that only specific parts of the framework were being exercised by the task. This might have affected the usefulness 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 may have altered the participants' behaviour and performance.
- **Lack of coding:** The fact that participants did not actually code their solutions (except for Participant A) may have impacted the range of problems encountered and the participants' performance during the study.

5.4.2 External Threats

In addition to the same generalisation threats that arose in the first study (those of using a single framework, the framework not being representative, and the participants not representing industrial developers) there were two further potential threats.

- **Selection effect:** Participants in the study volunteered which might 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.
- **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.

5.5 Data Analysis

In total 21 h 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 3 h were up, others asked to continue past the time limit. The amount of data was potentially overwhelming. The first step of analysis was to transcribe the data into a textual account for each participant. 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 (see Table 2). 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 later used as evidence.

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

Table 2 Excerpt from a study 2 participant transcript

Time	Documentation accessed	Talk aloud comments	Non-verbal observations
31	PL overview	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	Scanning Figure hierarchy
	PL identifying existing figures	It's annoying me it's too big (laugh)	Scrolling around
	PL Figure hierarchy	**R: What are you searching for?**	
...		...	...
35	PL overview	I want to modify a line	Modify figure [pattern] links back to figure hierarchy in PL
	PL modifying existing figures	Now I'm frustrated I've been to the 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	

thoughts and reactions of the participant 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. The comment in bold prefixed by "R:" is an instance where the researcher seeks clarification of participant behaviour. Finally, the non-verbal observations column was used to capture any interesting 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.

In total the transcription produced almost 200 pages of text and took a period of around 5 months to complete. The complete set of transcripts can be found in previous work by Kirk (Kirk 2005).

By focusing on documentation accesses within the transcripts it was possible to understand what problems participants faced and whether the documentation helped them to address these. An overview in the form of a matrix (see Table 3) was prepared of each participant's documentation accesses. This related the types of problems experienced by a participant to the types of documentation accessed in trying to resolve them. Columns headed by pluses represent documentation accesses that were considered helpful and minuses are used to represent accesses that did not help. Accesses were recorded as a letter representing the participant and the time when the access occurred—this allowed them to be tracked back to the original data.

While reading through the text it became apparent that participants' 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. The use of this information had originally been overlooked but, once recognised, it was important to include it in the analysis. This required the transcripts to be reread to identify incidents where previous experience appeared to be used, in effect treating it as another form of documentation.

Table 3 Participant E's problem vs. documentation matrix

	Pattern language		Micro-architecture		Source code		Previous knowledge		Other	
	+	−	+	−	+	−	+	−	+	−
Mapping	E101 E102 E103 ...	E22 E115 E135 ...	E20				E17 E42 E44 ...	E21 E55 E141		
Interaction	E114		E24 E27 E47		E117		E140			
Function			E25 E45 E55 ...	E22 E115 E119 ...	E21 E34 E34* ...	E26 E28 E35 ...		E43 E132		
Architecture								E30		
Other										

(The asterisk at E34 indicates that two problems were discovered at time 34.)

5.6 Problem versus Documentation Matrices

Table 4 presents a summary of the problem versus documentation matrices. The cells contain the number of accesses recorded for all participants during the task and are separated by problem and documentation type. From the table it appears that the pattern language has been accessed frequently for mapping type problems, although 20 out of the 50 accesses did not provide useful support for the problem at hand. Also noteworthy is the comparatively high number of accesses of micro-architecture and source code documentation during functionality problems, suggesting that participants perceived some benefit from these techniques for these problems.

Both the mapping and functionality problems appear to dominate this study. The high number of mapping problems is surprising because these were expected to be relatively few in number but with 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. However, it more likely reflects the fact that participants 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 participants are worrying about the future consequences of their actions rather than experiencing the effects of bad architectural decisions.

5.7 Detailed Observations

The goal of this study was to investigate the support provided for framework reuse by pattern languages and micro-architecture descriptions. One of the unexpected findings was how participants' previous knowledge impacted on the results. The key findings were that: tacit problem solving knowledge and previous experience can result in developers bypassing the pattern language, occasionally to the detriment of the solution; the pattern language was generally useful for introducing framework concepts, providing examples of common code and choosing from inheritance hierarchies; the micro-architecture documentation was useful for small scale interaction issues but was less useful for larger scale interaction problems; and, there was further confirmation of the four categories of framework reuse problem: mapping, interaction, functionality and architecture.

Table 4 Summary of the problem vs. documentation matrices

	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

5.7.1 Tacit Problem Solving Knowledge and Previous Experience Impacts on the Solution

During mapping, participants often pre-empted documentation access by suggesting an initial plan or solution seemingly off the top of their head. This reaction appears, in part, 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*" (Participant E, 112). The supposition that a figure is required presumably comes from the participant's previous experience of the framework. In many cases this immediate selection of a solution correctly identified a viable approach to the problem.

This was even more obvious when a participant had previous experience of solving similar problems. In those cases participants 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, participants attempted to apply a solution which was not suitable for the current context. In these circumstances the participant was slow to realise the poor fit, if such a realisation ever happened, persisting with the solution despite its awkwardness.

5.7.2 The Pattern Language Was Effective for Introducing Framework Concepts

In the cases where participants did not use previous knowledge, there is evidence that the pattern language played an important role informing them about the availability and applicability of parts of the framework to their problem. 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.*" (Participant 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* [pattern] *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*" (Participant F, 35).

5.7.3 The Pattern Language Was Useful for Providing Examples of Common Code and for Choosing Components from the Inheritance Hierarchies

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. In some cases (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 for reuse within the framework. Whereas in other cases (e.g., Participant C), the information provided was different; that participant already knew how to perform the common tasks covered by examples (e.g., creating a tool) and also already knew at least some of the possible options in each hierarchy. Instead, for participant C, the example became a piece of boilerplate code that could be easily modified for the task, providing

details that are easily forgotten. Similarly, hierarchies became defensive tools to make sure that no suitable classes had escaped attention, rather than to find new candidates.

5.7.4 The Micro-Architecture Documentation Was Useful for Small Scale Interaction Issues

The data collected suggests that understanding the interactions in a framework remains a hard problem to address using documentation. Participants answered many trivial forms of interaction problem effectively using the micro-architecture documentation, but occasionally a larger question involving a series of interactions and dependencies would arise and participants found these much more difficult to address.

The disjoint nature of the micro-architectures was unpopular with participants. Moving between micro-architectures required backtracking to the index before each switch. The interactions were fragmented into small pieces by the need to have separate graphs for each method on an interface and to limit the length of call sequences to only include the calls made between interfaces. This also meant that the interactions that were being shown were largely devoid of meaningful domain semantics, making it difficult for participants to appreciate their significance. It was also difficult to move between the micro-architectures and source code, as this required the participant to manually find the relevant source code amongst the various files of the framework and open it in a separate editor. Participants 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...*" (Participant E, 58). There was a suggestion that some of these problems may have been caused by a lack of familiarly with the micro-architecture documentation.

The micro architecture call graph notation was intended to illustrate how each of the major interfaces in the framework was called by other parts of the framework. The idea was that the participants could use this call graph to backtrack through the framework, identifying larger units of interaction than individual method calls, to gain an appreciation of the frequently occurring interactions and their purpose. This expectation proved to be optimistic. Participants seldom used the call graphs at all and when they did they certainly did not relate several of them together into a sequence nor did they give any indication during the study that they were aware of larger units of interaction operating within the framework.

5.7.5 There Was Confirmatory Evidence for the Four Categories of Framework Reuse Problems

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

Mapping problems still appear to be the most significant of the problems discovered. They dictated the actions of the participant over large periods of the reuse task, and, by their nature, they caused other problems to occur, or to be avoided, depending on how well the solution was mapped onto the existing parts of the framework.

Interaction problems were found to occur less frequently than originally assumed. In part this may have been because of the experimental setup, which did not go into the detail of coding a solution. However, when a significant interaction problem occurred, such as trying

to trace aspects of MVC in the framework, a large amount of time and effort was be spent attempting to identify and understand the interactions.

Functionality problems were very frequent in the task. Despite their frequency, they did not appear to trouble participants excessively. The reason for this is that each functionality problem is relatively small and therefore quite contained. In such circumstances a participant's failure to understand a part of the framework did not affect their overall solution. Functionality problems are also well supported by available documentation. This study suggested that participants do not like using source code, but the fact is they can use it and often do get useful information from it.

Architecture problems were infrequent in this study. One observation that can be made about the architectural concerns that did arise 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 works within the existing architecture, but one that is consistent with its essential qualities.

6 Conclusions

This paper makes several contributions to the problem of comprehending and documenting 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 categories can also help to drive the development of new forms of documentation, informing both their content and presentation.

This study has investigated the concept of a pattern language and modified its format in an attempt to make it better suited to supporting mapping and architectural reuse problems. The findings suggest that pattern languages can be an effective way to introduce framework concepts to developers. In many cases, when a developer is unsure how to proceed, the language can act as an effective guide. It supports mapping by describing the context of key framework reuse problems, and their solutions, and shows examples for developers to understand and copy. However, in this research, pattern languages seemed less successful at making participants in the study consider other alternatives when they had previous experience or knowledge of the problem that had to be solved.

The study also investigated micro-architecture descriptions as an aid to framework comprehension and proposed a documentation style 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 key chunks of interaction. In this study the micro-architecture documentation was not as useful as expected. In particular, the call graph view, which was supposed to help participants understand and trace interactions through the framework, was hardly used during the evaluation. Participants made better use of the interface descriptions and class hierarchies, which were both useful for suggesting functionality and assisting navigation to relevant parts of the source code, although this support could arguably be 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 are required to produce effective micro-architecture based documentation in the future. Identification of candidate micro-architectures within a framework remains a challenge.

This research shows 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 research, including the use of different types of framework, more realistic development environments, and longer periods of evaluation.

A question that naturally arises from this research is how should object-oriented frameworks be documented. The answer is somewhat qualified, but two approaches can be identified as promising candidates. The combination of a pattern language and a set of micro-architecture documentation as explored here is one approach. Pattern languages have the potential to address mapping and architectural problems, and micro-architecture oriented documentation has the potential to address both interaction and functionality problems, particularly if closely integrated with the framework source code. However, this research has revealed deficiencies within both of these forms of documentation as used in this study. As such, it is difficult to recommend this approach without further research to improve their effectiveness.

An alternative approach is to investigate a combination of examples and practical exercises. There is some evidence from this research and in the literature that such an approach can be effective. In the first study there was some evidence in the coursework reports that examples and practicals were found to be a useful form of documentation. This view is backed up by the work of Dénommée (Dénommée 1998), Sparks et al. (Sparks et al. 1996) and Shull et al. (Shull et al. 2000), who have all commented positively on example driven techniques. Despite their utility, examples are not a panacea. There is evidence from this study and the wider literature to suggest that they can be incomplete in their coverage (Shull et al. 2000) and potentially damaging to the architecture of a framework (Schneider and Repenning 1995).

Finally, it is recommended that those responsible for the documentation of object-oriented frameworks explicitly consider the four problem categories identified as a result of this research and, where relevant, specifically address them with appropriate documentation types. Further research is required to validate these problem categories using frameworks across a range of domains. Further research is also necessary to refine and evaluate documentation techniques that better address the types of problem that arise in real framework reuse.

Acknowledgments This work was made possible by funding from UK EPSRC research grant GR/N07509. We are also grateful to the students from the University of Strathclyde software architecture class for their enthusiasm and contributions and to the postgraduate students who volunteered to participate in the second study. We also thank the reviewers for their useful comments and suggestions which helped to improve the content and clarity of the paper.

References

Alexander C, Ishikawa S, Silverstein M, Jacobson M, Fiksdhal-King I, Angel S (1977) A pattern language—towns, buildings, construction. Oxford University Press, New York

Beck K, Cunningham W (2005) HotDraw. http://www.c2.com/cgi/wiki?HotDraw (Accessed February 2005)

Beck K, Johnson RE (1994) Patterns generate architectures. In Proceedings of the European Conference on Object-Oriented Programming, Bologna, Italy, pp139–149

Bosch J, Molin P, Mattsson M, Bengtsson P (1999) Framework problems and experiences. In: Fayad ME, Schmidt DC, Johnson RE (eds) Building application frameworks: object-oriented foundations of framework design. Wiley, pp 55–82

Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996) Pattern-oriented software architecture, vol 1: a system of patterns, Wiley.
Butler G, Keller RK, Mili H (2000) A framework for framework documentation. ACM Comput Surv 32(1)
Campbell D, Stanley J (1963) Experimental and quasi-experimental designs for research. Houghton Mifflin Company, Boston
Dénommée P (1998) A case study in documenting and developing frameworks. Master's Thesis, Concordia University, Canada
Dey I (1993) Qualitative data analysis: a user-friendly guide for social scientists. Routledge, London
Eclipse (2005) Eclipse. http://www.eclipse.org (Accessed February 2005)
Fairbanks G (2004) Software engineering environment support for frameworks: a position paper. In Workshop on Directions in Software Engineering Environments. Available online at: http://www.hdcp.org/Publications/WoDiSEE_ICSE04_Fairbanks.pdf (Accessed August 2005)
Fayad M, Schmidt DC, Johnson RE (1999) Building application frameworks: object-oriented foundations of framework design. Wiley, New York
Froehlich G, Hoover J, Liu L, Sorenson P (1997) Hooking into object-oriented application frameworks. In Proceedings of the International Conference on Software Engineering, Boston, USA, pp 491–501
Gamma E, Eggenschwiler T (2005) JHotDraw. http://www.members.pingnet.ch/gamma/JHD-5.1.zip (Accessed February 2005)
Gamma E, Helm R, Johnson RE, Vlissides J (1994) Design patterns: elements of reusable object-oriented software. Addison Wesley, Reading, MA
Gangopadhyay D, Mitra S (1995) Understanding frameworks by exploration of exemplars. In Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering, Toronto, Canada, pp 90–100
Hakala M, Hautamäki J, Tuomi J, Viljamaa A, Viljamaa J, Koskimies K, and Paakki J (1999) Managing object-oriented frameworks with specialization templates. In Proceedings of the Workshop on Object-Oriented Technology, Brussels, Belgium, pp 199–209
Harrison W (2000) N=1: an alternative for software engineering research? Beg, borrow, or steal: using multidisciplinary approaches in empirical software engineering research, workshop, 5 June, 2000 at 22nd International Conference on Software Engineering
Helm R, Holland IM, Gangopadhyay D (1990) Contracts: specifying behavioral compositions in object-oriented systems. In Proceedings of the 1990 European conference on object-oriented programming, Ottawa, Canada, pp 169–180
Höst M, Regnell B, Wohlin C (2000) Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. Empir Softw Eng J 5(3):201–214
Johnson RE (1992) Documenting frameworks using patterns. In Proceedings of the conference on object-oriented systems, languages and applications, Vancouver, Canada, October, pp 63–76
Judd CM, Smith ER, Kidder LH (1991) Research methods in social relations 6th edn. Holt Rinehart and Winston, Fort Worth, TX
Kaiser W (2005) JHotDraw as an Open-Source Project. http://www.jhotdraw.org (Accessed February 2005)
Kirk D (2005) Understanding object-oriented frameworks. PhD Thesis, University of Strathclyde, Glasgow, UK. http://www.cis.strath.ac.uk/research/efocs/abstracts.html#dk_thesis (Accessed January 2006)
Krasner GE, Pope ST (1988) A description of the model–view–controller user interface paradigm in the Smalltalk-80 system. J Object-Oriented Program 1(3):26–49
Lajoie R (1993) Using reusing and describing object-oriented frameworks. Master's Thesis, McGill University, Canada
Lajoie R, Keller RK (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, Montreal, Canada, pp 295–312
Meusel M, Czarnecki K, Köpf W (1997) A model for structuring user documentation of object-oriented frameworks using patterns and hypertext. In Proceedings of the European Conference on Object-Oriented, Jyvaskyla, Finland, pp 496–510
Microsoft (2005) Microsoft ASP.NET. http://www.asp.net (Accessed on 8 February 2005)
Miles MB, Huberman MA (1994) Qualitative data analysis: an expanded sourcebook 2nd edn. Sage Publications, Thousand Oaks, CA
Moser S, Nierstrasz O (1996) The effect of object-oriented frameworks on productivity. IEEE Comput 29 (9):45–51
Netu2 (2005).MediaCam high speed screen recording. http://www.netu2.co.uk/home.htm (Accessed February 2005)
OMG (2005a) Corba. http://www.corba.org (Accessed February 2005)
OMG (2005b) The unified modelling language. http://www.uml.org (Accessed April 2005)

Ortigosa A, Campo M, Salomon RM (1999) Enhancing framework usability through smart documentation. In Proceedings of the 3rd Argentine Symposium on Object-Orientation, Buenos Aires, Argentina, pp 103–117

Perry DE, Porter AA, Votta LG (2000) Empirical studies of programmers: a road map, international conference on software engineering, Proceedings of the Conference on The Future of Software Engineering, pp 345–355

Pree W (1999) Hot spot driven development. In: Fayad ME, Schmidt DC, Johnson RE (eds) Building application frameworks: object-oriented foundations of framework design. Wiley, pp 379–394

Robitaille S, Schauer R, Keller RK (2000) Bridging program comprehension tools by design navigation. In Proceedings of the International Conference on Software Maintenance San Jose, USA. Washington, DC: IEEE Computer Society, pp 22–32

Schmidt DC (2005) The adaptive communication environment framework. http://www.cs.wustl.edu/~schmidt/ACE.html (Accessed February 2005)

Schneider K, Repenning A (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, Ann Arbor, MI, USA, pp 177–188

Shull F, Lanubile F, Basili VR (2000) Investigating reading techniques for object-oriented framework learning. IEEE Trans Softw Eng 26(11):1101–1118

Slaney J, Thiébaux S (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

Sparks S, Benner K, Faris CA (1996) Managing object-oriented framework reuse. IEEE Compu 29(9):52–60

Sun Microsystems (2005a) J2EE: JDBC Technology. http://www.java.sun.com/products/jdbc (Accessed February 2005)

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

Sun Microsystems (2005c) Core Java: Javadoc Tool. http://www.java.sun.com/j2se/javadoc (Accessed February 2005)

Together (2005) Borland Together Designer. http://www.borland.com/us/products/together/index.html. (Accessed January 2006)

Van Grup J, Bosch J (2001) Design erosion: problems and causes. J Syst Softw 61(2):105–119

White S, O'Madadhain J, Fisher D, Boey Y (2005) JUNG: Java Universal Network/Graph Framework. http://www.jung.sourceforge.net (Accessed February 2005)

Douglas Kirk received an honours degree and a Ph.D. in Computer Science from the University of Strathclyde in Glasgow, UK. He is currently a research assistant at the Department of Computer and Information Sciences at the University of Strathclyde, Glasgow, UK. His research interests are in object-oriented design, the comprehension and documentation of software frameworks, mobile code and empirical studies.

Marc Roper received an honours degree in Computer Science from the University of Reading and his Ph.D. degree from the CNAA. He is currently a reader in the Department of Computer and Information Sciences at the University of Strathclyde, Glasgow, UK. His research interests cover a range of software engineering topics, and currently focus on the development of tools and techniques to support software engineers in analysing and understanding large software systems.

Murray Wood received an honours degree and a Ph.D. degree in Computer Science from the University of Strathclyde in Glasgow, UK. He is currently a senior lecturer in the Department of Computer and Information Sciences at the University of Strathclyde, Glasgow, UK. His research interests are in the design, understanding and verification of large-scale software systems, and in empirical software engineering.