University of Porto
Faculty of Engineering

Patterns and Tools for Improving Framework Understanding: a Collaborative Approach

Nuno Honório Rodrigues Flores

December 2012

Scientific Supervision by

Doctor Ademar Aguiar, Assistant Professor
Department of Informatics Engineering

In partial fulfillment of requirements for the degree of
Doctor of Philosophy in Informatics Enginnering
by the Doctoral Program in Informatics Engineering

Contact Information:

Nuno Honório Rodrigues Flores
Faculdade de Engenharia da Universidade do Porto
Departamento de Engenharia Informática

Rua Dr. Roberto Frias, s/n
4200-465 Porto
Portugal

Tel.: +351 22 508 1400
Fax.: +351 22 508 1440
Email: nuno.flores@fe.up.pt
URL: http://www.fe.up.pt/∼nflores

This thesis was typeset on an Apple® MacBook® running Mac OS® X 10.6.4 using the free LaTeX typesetting system, originally developed by Leslie Lamport based on TeX created by Donald Knuth. The body text is set in Latin Modern, a Computer Modern derived font originally designed by Donald Knuth. Other fonts include Sans and Typewriter from the Computer Modern family, and Courier, a monospaced font originally designed by Howard Kettler at IBM and later redrawn by Adrian Frutiger. This colophon has exactly seventy five (75) words, excluding all numbers and symbols.

ISBN 978-972-752-138-8

9 789727 521388

. . . to my wife, Sara

. . . to my growing family

This page was intentionally left mostly blank.

Abstract

Understanding a piece of software is an important activity of software development, and one with a big social emphasis. An average software project requires several people to work together in collaboration. When trying to understand programs, to maintain or evolve them, developers turn first to the code and, when that fails, to their social network, i.e., the development team. Nevertheless, it is not easy to go for the team, mainly due to the lack of expertise awareness (who to ask), wasteful interruptions of the wrong people (unclear expertise localisation) and unavailability (either due to intrusion or time constraints).

Frameworks proved to be a powerful technique for large-scale reuse, but developers have to invest considerable effort to understand them. Their design is often very complex and hard to communicate, turning good quality documentation a crucial part. Producing such documentation can be costly as it needs to be easy to use, to cover different audiences, and to present different types of documents using different notations. But even if the documentation is produced with quality standards, the learners need to acquire knowledge from it and their cognitive needs must be attended. If the documentation doesn't help, or partially helps, how and where does the learner look for the knowledge needed to understand the framework and solve the task at hand?

In this dissertation, the author pursues solutions to tackle these issues concerning framework understanding, aiming to answer *how can framework understanding be improved and how can learners help each other without too much effort?* The solutions proposed in this dissertation combine two approaches: best practices and collaborative learning.

Observation and historical analysis show that understanding frameworks typically encompass a recurrent set of problems. If learners are aware of these obstacles and how to proceed to overcome them, they will surely accelerate their learning process. Therefore, a set of proven good solutions to those recurrent problems is presented in the form of patterns. These best practices aim at answering questions regarding where to start learning, at what level of abstraction should one go, how to cope with one's cognitive needs, and how to keep the knowledge produced. They provide a process of going through the documentation and, if it proves insufficient, alternative ways to gather information about the framework. To introduce collaboration into this learning experience, a collaborative environment was developed – the DRIVER platform.

The DRIVER platform is a collaborative learning environment where framework users can, in a non-intrusive way, store and share their learning knowledge while following the best practices of framework understanding (patterns). It provides a framework documentation repository, mounted on a wiki, where the learning paths (*how did one learn*) of the community of learners can be captured, shared, rated, and recommended. Combining these social activities, the DRIVER platform promotes collaborative learning, mitigating intrusiveness, unavailability of experts and loss of tacit knowledge.

Empirical evidence of the benefits of the proposed solutions is provided by the results of a controlled academic quasi-experiment. The experiment took groups of similar MSc students and measured their performance, effectiveness and framework knowledge intake, while developing a set of tasks using a new framework.

Shortly, the patterns and tools proposed in this dissertation help on improving framework understanding through best practices and collaborative learning, thus contributing to bridge the gap between the single learner and the learning community.

Resumo

O desenvolvimento de *software* tem-se tornado numa atividade com grande índole social, e onde a compreensão do mesmo é algo de elevada importância. Em média, um projeto de desenvolvimento de *software* requere que um conjunto de pessoas colabore entre si em equipa. Na tentativa de compreender como um determinado software funciona, os analistas começam por inspecionar o código-fonte e, em caso de insucesso, recorrem à sua rede social, ou seja, à equipa de desenvolvimento. No entanto, esta tarefa não é fácil, devido a fatores como: a incerteza sobre quem são os peritos (podendo levar a interrupção das pessoas erradas) e a falta de disponibilidade desses mesmos peritos.

As *frameworks* são uma poderosa técnica de reutilização de software em larga-escala, mas requerem um esforço considerável de aprendizagem. A sua arquitetura, devido à sua grande complexidade, é difícil de aprender. Desde logo, a documentação existente deve ter qualidade e ser adequada. Produzir esta documentação torna-se custoso, pois esta terá de ser de fácil leitura, cobrir diferentes tipos de leitores com diferentes tipos de documentos e formatos. Mas mesmo que esta documentação tenha a qualidade esperada, terá igualmente de satisfazer as necessidades cognitivas de aprendizagem dos leitores. De outra forma, onde irá o leitor obter o conhecimento necessário para compreender a *framework* e resolver a tarefa em mãos?

Nesta dissertação, o autor procura soluções que ajudem a atenuar estas questões, tentando responder à questão: *Como melhorar a aprendizagem de frameworks e como pode haver interajuda entre os intervenientes (pessoas) sem grande esforço?*. As soluções propostas nesta dissertação combinam duas abordagens: boas práticas e aprendizagem colaborativa.

Estudos sobre a aprendizagem de *frameworks* indicam que este processo é, tipicamente, afetado por um conjunto de problemas recorrentes. A perceção e conhecimento prévios destes obstáculos, e respetivas formas de os ultrapassar, seguramente acelerará o processo de aprendizagem. Por conseguinte, é apresentado um conjunto de soluções, garantidamente boas, para resolver estes problemas recorrentes, sob a forma de padrões. Estas boas práticas abordam questões relativas a: por onde começar, a que nível de abstração aceder, como abordar as necessidades cognitivas e como preservar o conhecimento. No geral, permitem guiar o leitor através da documentação e, caso seja insuficiente, propõe formas alternativas de obter informação sobre a *framework*. A colaboração neste processo de aprendizagem é introduzida através de uma plataforma denominada DRIVER.

A plataforma DRIVER é um ambiente de aprendizagem colaborativo, que permite aos utilizadores de *frameworks*, de uma forma pouco intrusiva, o registo e a partilha do conhecimento gerado durante a aprendizagem, tendo esta sido guiada pelas boas práticas. É composta de um repositório de documentos sobre a *framework*, apresentados numa *wiki* e onde os passos de aprendizagem (*learning paths*) de cada individuo são capturados, partilhados, avaliados e recomendados pela comunidade de utilizadores. A combinação destas atividades, comuns hoje em dia na *web* social, fomenta a colaboração na aprendizagem, atenuando fatores como a intrusão/interrupção, falta de disponibilidade e perda de conhecimento intrínseco.

A realização de uma experiência de validação num ambiente académico serviu para mostrar os benefícios que as soluções apresentadas trazem à aprendizagem de frameworks, analisando-se o desempenho, objetividade e conhecimento adquirido por grupos de estudantes.

Os padrões e ferramentas propostos nesta dissertação melhoram a compreensão de frameworks através da reutilização de boas práticas e através da aprendizagem colaborativa, aproximando o indivíduo, da comunidade.

Contents

List of Figures

List of Tables

Preface

I can only show you the door... You're the one who has to walk through it...

Morpheus *in "The Matrix"*

I was ten years of age when my parents presented me with the ZX Spectrum 48K, my first computer. Not so much for its 23x14cm, 550 grams, rubber keys and rainbow motif, but for how a small contraption would empower you to create things, it bewildered me. I became a wizard (i.e. *programmer*) and I could cast spells (i.e. *run programs*), summon creatures (i.e. *invoke subroutines*) and shape my own world (i.e. *develop software*). I had my own building blocks (i.e. *programming instructions*) just there in my spellbook (i.e. *BASIC manual*). I needed nothing more, just imagination, wits and that small contraption. That day changed my life forever and my fate was sealed: I was going to work on computers (i.e. *be a software engineer*).

I came a long way from those early days when I could do everything with just 48K of memory and a bunch of programming instructions. I soon realize that all things scale and grow. Fast-forwarding to years later and my definition of building blocks had *somewhat*[1] changed. I wasn't dealing with simple beginner programs, but with large-scale software development and high-complexity software components. Nevertheless, the thrill was still there. I was expressing the same excitement when I first looked at those BASIC commands that enabled me to play God, when looking at the empowering *frameworks* provided me, although now, on a different scale. I felt I had the perfect tool to enable *quick and easy* software development.

During my time in the industry, I soon realized that developing software had that boring part of having to tackle with the same common problems[2] over and over again. I found myself extracting and refactoring chunks of code to create my own personal library of reusable code, predicting I would probably need it later on. Frameworks did this for me, without any effort. Great! Henceforth, I came in contact with several frameworks,

[1] clearly, an understatement
[2] all applications have GUIs, CRUD data storages, and so on...

from generic to domain-specific, and every time, I had to spent a certain amount of time trying to use them properly. Reusing software became my scaffold for development.

Then *design patterns* stepped in. And I was equally dazzled. Not only we were reusing software, but software solutions and experience. The potential was enormous and so was my interest in the subject. Soon, I was developing my MSc research work in reverse engineering and recovery of design pattern instances in frameworks [Flo06]. I was poised to improve the learning curve of frameworks and help developers using them. By the end of my MSc, I was content with my results, but there was still something missing.

I needed to dig deeper, start at the beginning, and go to the core of the problem. As such, I delved into the realms of program comprehension. Understanding a framework starts at understanding programs. Programs are understood by humans, so one must be aware of the human cognitive processes behind it. Quickly, my research acquired an anthropological/sociological flavor to it, which it did not displease me. By this time, the *social*/Web 2.0 phenomenon was booming and, somehow, it struck a chord as it began contaminating my mind with lots of interesting concepts and ideas.

It wasn't until I came before James Surowiecky's book WISDOM OF CROWDS, that the path of my research became clear. This book was truly an inspiration for me and it cleared my mind of the doubts I had on how to proceed. It had strong assumptions of empirical knowledge, supported by documented case-studies, about how the whole is better than the sum of its parts and how *collective intelligence* can and will play an important role when making decisions and analyzing options. His claims were so clear to me that I knew I needed to pursue that line of thinking.

So the questions popped: How can we help potential framework users to learn a framework effectively? And, moreover, how can we make them help each other without too much effort? As the reader will realize throughout this dissertation, these were the starting points of my research. Software is developed in teams, in communities. Don't these groups have in them what it takes to improve learning? I believe so. They just need support and awareness of their ability, and to be nurtured to do so. Learning (in the sense of knowledge transfer) is not straightforward and depends on a bunch of things: People, goals, time, etc. We learn better when someone teaches us, so this work is based on a *"Let me show you how I did it"* philosophy (thus the preamble).

Throughout all the years it took me to undertake this endeavor, a noble group of people stood by me, helping, supporting and encouraging my work. To all of them, I am deeply grateful. At the top of this group, undoubtedly, is my colleague, friend and supervisor Ademar Aguiar. His stimulus, enthusiasm, patience and clever guidance were of vital importance. He stroke a perfect balance between giving me autonomy and pushing me forward when the need arose. Thank you Ademar.

I would like to endorse my thanks to both professors Eugénio Oliveira and Augusto de Sousa, of the Doctoral Program in Informatics Engineering (ProDEI) for all the support provided and to thank Paris Avgeriou and Isabel Ramos for their halfway steering. I must also extend my gratitude to the Department of Informatics Engineering (DEI) where I've been lecturing since 2005, most especially to the head of the department, Prof. Raul Vidal, for his undeniable support and charismatic influence throughout my entire academic career.

A special thanks to my PhD collegues Hugo Ferreira (your LaTeXtemplates were very useful) and Filipe Correia (always inquisitive) with whom I had the pleasure to debate, travel, brainstorm and laugh in numerous occasions. Thank you also to my research and work colleagues, Ana Paiva and João Pascoal Faria for their promptly availability and support. A very special thanks goes to Diana Soares, to whom I, most humbly, confer the title of "Javascript guru" and that proved vital to alleviate constraining problems while tackling with web technologies.

Thanks to both my close friends, Plácido, César, Helder, Ana, Rita and Sofia, for their companionship, and to my everlasting music buddies, Daniel Pereira, Gonçalo Cruz, Raquel Ferreira e João Conceição, a deep-felted thanks for all the concerts we did together and, hopefully, will keep doing. Arrefole forever!

Always there for me were my parents, Eugénio and Elvira and my sister Raquel whose reliability and love are eternal. I would also like to express my gratitude to my sisters-in-law, Ju and Inês, for their constant support.

My final thanks goes to my companion for life, Sara, to whom I dedicate this work.

[illegible]

Porto, Portugal
December 2011

Chapter 1

Introduction

The demand for software has been progressively increasing almost since it became present in our daily lives. Productivity leverage comes, not so much from writing code *faster*, but writing *less* code, while keeping high quality levels. Research showed that software reuse is the (only) realistic approach capable of bringing out the gains of productivity and quality the software industry needs [MMM95].

Frameworks are a powerful technique for large-scale reuse that helps developers to improve quality and to reduce costs and time-to-market. However, before being able to reuse a framework effectively, developers have to invest considerable effort on understanding it. Especially for first time users, frameworks can become difficult to learn, mainly because its design is often very complex and hard to communicate, due to its abstractness, incompleteness, superfluous flexibility, and obscurity.

Regardless, as software systems evolve in size and complexity, frameworks are becoming increasingly more important in many kinds of applications, in different technologies (object-orientation and recently aspect-orientation too), in new domains, and in different contexts: industry, academia, and single organisations.

Understanding a piece of software is an important activity of software development, with an increasing social emphasis. Advances in global software development are leading to teams continuously becoming more and more distributed. A software development project requires people to collaborate. Trends toward distributed development, extensible IDEs, and social software, influence makers of development tools to consider how to better assist the social aspects of development.

Can developers collaborate to help each other learning how to use a framework? How can we use this socialising trend in software development to improve framework learning and diminish its usage overhead? This dissertation tries to tackle these issues, first, through observation and establishment of recurrent good practices in understanding a framework in general, and second, supporting the collaboration of developers in specific learning needs.

This chapter gives a general introduction to the overall dissertation. It starts by contextualising the reader to the subject of frameworks and software reuse, raising issues regarding the understanding of frameworks and the problems that still affect framework users. It moves to state the research questions and goals that stood at the base of the presented work, overviewing how these were pursued and how the answers (to the research questions) were collected.

1.1 Software reuse and frameworks

The introduction of reuse in a software development process implies splitting the traditional software life cycle into two interrelated cycles: one focused on **developing reusable assets**, and another focused on **searching and reusing reusable assets already developed** [Agu03].

A framework is a reusable design together with an implementation. It consists of a collection of cooperating classes, both abstract and concrete, which embodies an abstract design for solutions to problems in an application domain [JF88][Deu89][FS97].

In the particular case of framework-based application development, the traditional life cycle can be organised in: a **framework development** life cycle devoted to building frameworks, corresponding to the abstraction phase of software reuse; and an application development life cycle (also known as **framework usage**) devoted to developing applications based on frameworks, corresponding to the selection, specialisation, and integration phases of software reuse.

Although the activities of framework development and application development are often separate and assigned to different teams, the knowledge to be shared between them is large, as the design of a framework for a domain requires considerable past experience in designing applications for that domain. In application development, frameworks act as generative artifacts as they are used as a foundation for several applications of the framework's domain. This contrasts with the traditional way of developing applications, where each application is developed from scratch.

The most distinctive difference between the traditional and the framework-based development of applications is the need to map the structure of the problem to be solved onto the structure of the framework, thereby forcing the application to reuse the design of the framework. The positive side of this is that we don't need to design the application from scratch. But, on the other hand, before starting application development, we need to understand the framework design, a task that sometimes can be very difficult and time-consuming, especially if the framework is large or complex, and is not appropriately accompanied with good documentation or training material.

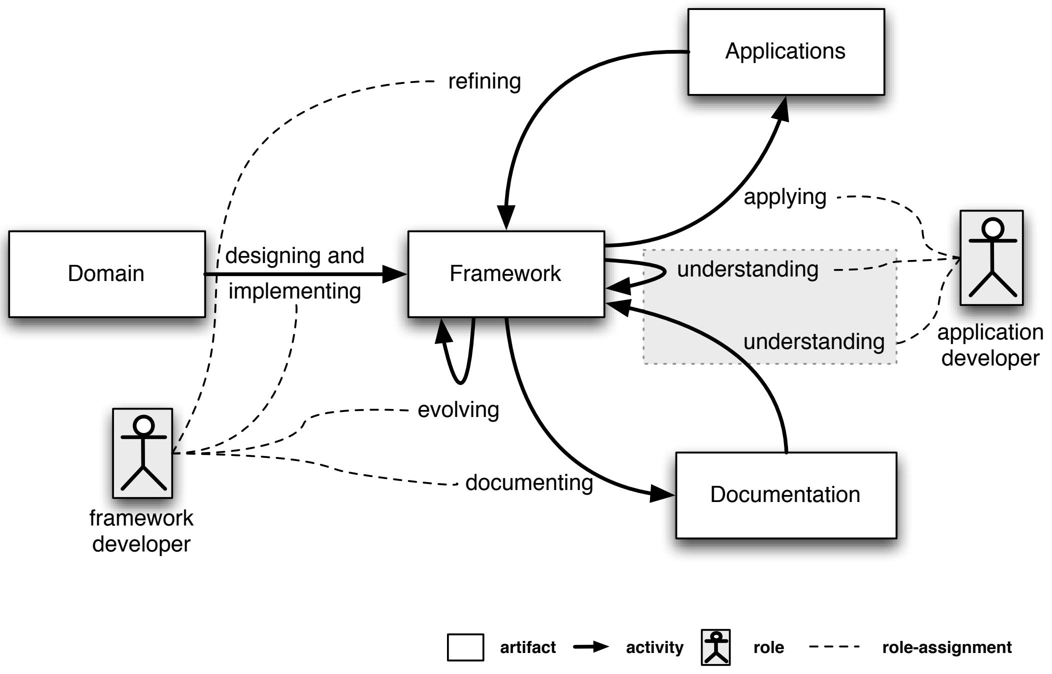


Figure 1.1: Activities, artifacts and roles of framework-based application development (extracted from [Agu03]).

Figure 1.1 shows a simplified view of framework-based application development that relates the artifacts, activities, and roles most relevant for the context of framework understanding. Because understanding frameworks is of major importance for application developers, in the figure, the activity is assigned exclusively to that role, but in fact, it can be also relevant for framework selectors, original framework developers (especially of large frameworks), framework maintainers, and developers of other frameworks, although not with the same degree of importance.

Although some of the problems here addressed could also be common to large or complex software systems, frameworks are specifically designed to be easy to reuse, thus adding special needs from the point of view of learning and understanding.

1.2 Understanding frameworks

A statement by Grady Booch says that [Boo94]:

> *"the most profoundly elegant framework will never be reused unless the cost of understanding it and then using its abstractions is lower than the programmer's perceived cost of writing them from scratch."*

Understandability comes as a key requirement for the effective reusability of frameworks. The easier it is to understand the framework, the easier it would be to reuse it. Briefly, a framework is hard to understand due to a set of design aspects, the same that convey it its power [But98]:

- the design is **very abstract**, to factor out commonality;
- the design is **incomplete**, requiring additional classes to create a working application;
- the design provides **more flexibility** than the strictly needed by the application at hand;
- and the design is **obscure**, in the sense that it hides existing dependencies and interactions between classes.

All these aspects inherently make framework design hard to communicate. Therefore, there should be an increasing concern in accompanying the framework with suitable documentation.

1.2.1 Documenting frameworks

Good quality documentation is crucial for the effective reuse of object-oriented frameworks [Agu03]. Without a clear, complete and precise documentation describing how to use the framework, how it is designed, and how it works, the framework will be particularly hard to understand and nearly impossible to use by software engineers not initially involved in its design.

But simple, complete, and easy-to-use documentation can be seen as the ACID[1] test for the quality of a framework. To be complete, the overall documentation of a framework usually combines a lot of information that must be produced, organised, and maintained consistent. The documentation must describe the application domain covered by the framework, its purpose, how-to-use it, how it works, and to provide details about its internal design, which globally may involve a large diversity of contents, and many different ways of presenting them [BKM00].

[1] standing for *atomicity, consistency, isolation, durability.*

The inherent complexity of framework documentation is mainly due to the following requirements:

- **Different audiences.** Framework actors range from developers, to users, to selectors, each with specific needs regarding the kind of information, abstraction level and level of focus.
- **Different types of documents.** Covering different levels of abstraction, presented using multiple views, as to fulfil the different audiences' requirements.
- **Different notations.** Combining free text, source code, and, formal specifications, it is convenient to provide a suitable mix of contents, throughly cross-referenced and navigable.
- **Easy-to-use.** Simple, minimalist, and declutered documentation that goes right to the point, is not easy to produce.

As a result of this complexity of requirements, the cost of producing good framework documentation, especially without well-defined methods and tools, can be very high, and even worst, it can be risky. For all these reasons, framework documentation is one of the most important activities in framework development, although its importance is not always recognised.

An approach to deal with this problem has been proposed by Aguiar [Agu03], who suggested a minimalist approach to framework documentation. It covered the overall documentation process, from the creation and integration of contents till the publishing and presentation, by focusing on minimising the obtrusiveness to the learner of training material, and offering the minimal amount of information that the learner needs to get the task done. This minimalist approach encompasses (i) a documentation model to organise the contents required to produce a minimalist framework manual along a virtual layered documentation space, (ii) a process which defines roles, techniques and activities involved in the production of minimalist framework manuals, and (iii) a specific set of tools, convenient to be adopted in a wide range of software development environments.

This approach deals with producing suitable documentation without too much effort. But there is still the process of acquiring the knowledge from the produced contents, that is, using that documentation to learn about the framework. The knowledge is there, but there is still no learning process that supports the effective retrieving of that knowledge. A process that suits the specific needs of the learner, and goes beyond just reading and navigating through the documentation.

1.2.2 When documentation is not enough

Frameworks are complex, so there needs to be documentation. The effort of producing information capable of covering all the afore-mentioned needs, produces a set of heterogenous, semi-structured, specialised, documentation artifacts. These documents need to be handled by the framework user, when trying to learn the framework.

Learning about a framework is a process that can be characterised according to a series of interdependent aspects:

- **Goal.** What is the role of the framework learner (user, developer, evolver, selector, etc.) and what does she want to do with the framework? Is she selecting, instantiating or evolving a framework?

- **Cognitive needs.** What is the cognitive profile [FS05] of the framework learner? Is she a *global* or a *sequencial* learner? Does she prefer a more *visual* or *verbal* information format? Would she go for a *top-down* or *bottom-up* approach? Furthermore, what does she need to know to be able to solve her problem? What are the cognitive steps she needs to take to satisfy her knowledge craving?

- **Abstraction level.** Depending on the learning goal, the learner may be required to navigate up or down the levels of abstraction present in the framework, trying to collect and assemble the knowledge needed to solve her problem.

- **Knowledge availability.** Does the documentation has all the information the learner needs? Is the documentation complete? Is it up-to-date? Where can the learner get more information, besides the documentation?

Documentation provides knowledge vessels, suitable to hold the knowledge about the framework, and to communicate it effectively. But they hardly induce the process behind learning about a framework, so the learner will have to find her own way through the documentation, while constructing her own mental model. Not so much a strenuous task, yet the problem relies in cases where the documentation is unsuitable or incomplete, making the process longer and harder.

Explicitly providing guidance to the process of learning about a framework could help mitigate the effort the learner spends. She would know where to start learning, what artifacts to look for (regarding her specific learning needs) and ways to construct her mental model in a more focused and direct way, without having to wander through the documentation trying to figure out where the relevant, useful knowledge might be. This process should not be a prescriptive, *script-like* formula that is recurrently applied, but a series of related patterns or best practices that have, provenly, lead to satisfying results and that can be adapted to the context at hand. Often, stating the (not so) obvious helps focusing the learner.

But even with a process behind learning a framework, the documentation and the framework itself may not be sufficient to provide solutions in a time-effective way. The effort investment becomes so great, that it may be simpler, and quicker, asking another framework learner for help. The communication is better, more focused and more effective than going through the documentation.

1.2.3 Grasping the community

Software development is a highly social activity. More and more, teams are distributed in space and time, and projects are inherently cooperative, providing the development of a shared understanding over the problem domain. Knowledge flows throughout the team, as the social-technical gap is lessened, with more tools and techniques emerging to support this collaborative software development.

People learn by reading source-code, documentation and asking questions to their peers. The quality of the software to be developed is determined not only by the sum of each developer's knowledge, but also by the social relationships of developers that impact the sharing of knowledge during the development process [NYY06].

Learning about a framework fits the same scenario and can, thus, be improved by resorting to the knowledge and aid of the team of developers. But, is that knowledge within our grasp?

Teams are composed of people and humans are slow and error-prone. Moreover, (natural) language is expressive but ambiguous, memory skips details, and teams rotate making it hard to keep track of everybody. The generated knowledge, if not harnessed at its best, is, eventually, lost. Observed obstacles in this knowledge safe-keeping are [NYY06][LVD03]:

- **Availability.** Not only it maybe difficult to know whom to ask, but most of the time the knowledgeable colleague is not accessible or available to help (task and time constraints).

- **Intrusion.** If helping means interrupting one's work, then it becomes intrusive and developers tend to give it a negative connotation. Interrupted developers lose track of parts of their mental model, resulting in laborious reconstruction or bugs and discouraging more frequent interruptions.

- **Tacit knowledge.** Developers spend vast amounts of time gathering precious, demonstrably useful information, but rarely record it for future developers. Keeping this information in their heads, renders it useless for helping other developers, unless it is shared.

- **Selfish ownership.** Most common in expert developers, there is an intrinsic sense of knowledge *ownership* that leads to a reluctance in sharing their expertise with others, fearing to loose their own status-keeping leverage.

With these issues in mind, the author believes that a collaborative learning environment will help mitigate these aspects and promote the collaboration between learners. The idea is to enhance the framework user's (a.k.a. application developer) activities with a strong contribution from the community of developers that share the same problems, issues and solutions, as seen in Figure 1.2. Rather than providing a *forum* or a *Q&A* service, this collaborative environment builds on learning knowledge, that is, the steps the learner took (while going through the documentation) that enabled her to build a solution to her problem.

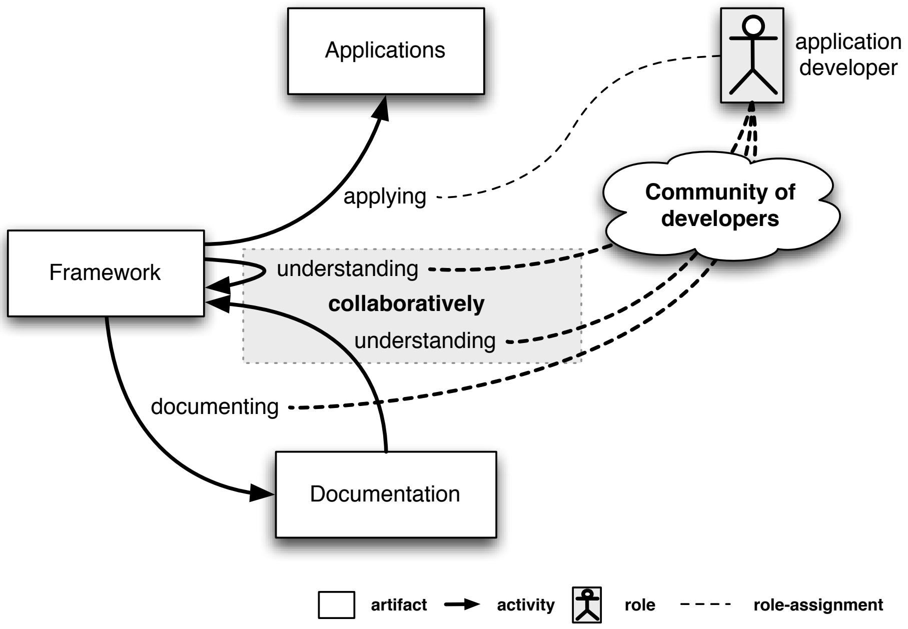


Figure 1.2: Collaboratively improving activities of framework-based application development, acknowledging the community of developers as a strongly contributive entity.

1.3 Research goals

Considering all that has been addressed, hitherto, this dissertation aims at providing contribution to the body of knowledge in software engineering, in concrete, framework understanding.

The main research goal is **to improve framework learning** by: (i) ***guiding learners on following best practices for framework understanding***, supporting its specific process and (ii) ***providing tools that assist framework understanding***, devising a collaborative environment that enables developers to share their learning strategies and allowing these to be captured and harnessed in a non-intrusive way.

These research goals are further detailed and explained in Chapter 5 (p. 69).

1.4 Research strategy

Software engineering is still maturing as a research area. Software development has specific characteristics that suggests its own research paradigm, combining aspects from other disciplines: it is a human creative phenomenon; software projects are costly and usually have long cycle times; it is difficult to control all relevant parameters; technology changes very frequently, so old knowledge becomes obsolete fast; it is difficult to replicate studies; and there are few common ground theories.

A categorization proposed at Dagstuhl workshop [THP92], groups research methods in four general categories, quoted from Zelkowitz and Wallace [ZW98]:

- **Scientific method.** *"Scientists develop a theory to explain a phenomenon; they propose a hypothesis and then test alternative variations of the hypothesis. As they do so, they collect data to verify or refute the claims of the hypothesis."*
- **Engineering method.** *"Engineers develop and test a solution to a hypothesis. Based upon the results of the test, they improve the solution until it requires no further improvement."*
- **Empirical method.** *"A statistical method is proposed as a means to validate a given hypothesis. Unlike the scientific method, there may not be a formal model or theory describing the hypothesis. Data is collected to verify the hypothesis."*
- **Analytical method.** *"A formal theory is developed, and results derived from that theory can be compared with empirical observations."*

These categories apply to science in general. Effective experimentation in software engineering requires more specific approaches. Software engineering research comprises computer science issues, human issues and organisational issues. It is thus often convenient to use combinations of research approaches both from computer science and social sciences. The taxonomy described by Zelkowitz and Wallace [ZW98] identifies twelve different types of experimental approaches for software engineering, grouped into three broad categories:

- **Observational methods.** *"An observational method collects relevant data as a project develops. There is relatively little control over the development process other* *than through using the new technology that is being studied"*. There are four types: project monitoring, case study, assertion, and field study.

- **Historical methods.** *"A historical method collects data from projects that have already been completed. The data already exist; it is only necessary to analyse what has already been collected"*. There are four methods: literature search, legacy data, lessons learned, and static analysis.

- **Controlled methods.** *"A controlled method provides multiple instances of an observation for statistical validity of the results. This method is the classical method of experimental design in other scientific disciplines"*. There are four types of controlled methods: replicated experiment, synthetic environment experiment, dynamic analysis, and simulation.

The best combination of methods to use in a concrete research approach is strongly dependent on the specific characteristics of the research study to perform, viz. its purpose, environment and resources. Hereafter, the research methods referred will use this terminology. Further description of each method can be found in [ZW98].

Based on the expected results and contributions of the work presented in this dissertation, the research strategy comprised a mix of *observational* and *historical* methods (case and field studies, literature search, and lessons learned) to substantiate the patterns contribution (Chapter 6, p. 81) and a *controlled* replicated experiment to complement it and provide evidence of the remaining contributions (Chapter 7, p. 109). A detailed rational of the methods is presented in Chapter 5 (p. 69).

1.5 Expected results

The expected outcomes of this thesis are the following contributions to the body of knowledge in software engineering:

1. **Elicitation of the best practices of framework understanding, documented in a pattern form.** Developers continually tackle with understanding problems, whenever they need to use a new framework or, again, recall how to use an already known, but dormant, framework. Through observation and experience, recurrently applied practices that lead to good results can be identified in the framework understanding process. Harnessing these best practices into an effective communication format (patterns), enables future framework learners to improve their learning process by accessing and reusing the experience and expertise of former framework learners. These patterns are not a recipe for success, but an important tool that

guides novice learners in the right direction. This contribution can be seen in detail in Chapter 6 (p. 81).

2. **Definition of a collaborative framework learning process.** Teams of developers collaborate. This collaboration generates knowledge, much of which is not captured and recorded for future use. New learners could benefit from the knowledge that previous learners generated when trying to understand a specific framework. In a collaborative, socially imbued setting, a non-intrusive, harnessing knowledge process, that enables the community to share and improve this knowledge, could nurture this learning collaboration and richly enhance the learning experience. Details on this contribution are present in Chapter 7 (p. 109).

3. **Development of tools that support the defined collaborative framework learning process.** Learners gain in collaborating. The issue lies, not so much in collaborating, but, in allowing collaboration to occur progressively without too much effort. If, collaterally, collaborating brings interruption, intrusion and entropy to development, then, it is abandoned. Therefore, there should be a set of tools that support the collaborative learning process so that it can occur seamlessly and naturally fitted into the development environment, without changing (to an extent) the developers' working habits. This contribution is further detailed in Chapter 7 (p. 109).

4. **Design of a repeatable experimental package to conduct further studies.** The apparent benefits of both patterns and collaborative learning process need to be validated and observed in the real world. Consequently, a (quasi-)experiment in a controlled academic experimental environment will be conducted, where the study of groups of undergraduate students that engaged in learning a new framework are expected to provide evidence that the proposed contributions helps learners and improve the learning process. This (quasi-)experiment is designed as an experimental package, to be performed in different locations, and by different researchers, in order to enhance the ability to integrate the results obtained and allow further meta-analysis on them. This contribution is detailed in Chapter 8 (p. 147).

1.6 How to read this dissertation

The remaining of this dissertation is logically organized into three parts, with the following overall structure:

Part 1: State of the art. The first part reviews the most important concepts and issues relevant to the thesis:

- Chapter 2, "Program comprehension" (p. 17), provides an extensive literature review on the field of program comprehension, its concepts, theories, models and supporting tools.
- Chapter 3, "Framework understanding" (p. 37), provides a state-of-the-art overview on framework understanding, a sub-topic of program comprehension, focusing the scope of research.
- Chapter 4, "Collaborative software development" (p. 53), focus on the fields of Groupware and Computer-Supported Collaborative Work, converging into the Collaborative Software Engineering research area, as an auxiliary field to the research work presented in this dissertation.

Part 2: Problem & solution. The second part states the problem researched and the proposed solution:

- Chapter 5, "Research problem and solution" (p. 69), lays both the fundamental and specific research questions in scope for this thesis, and overviews the proposed solution.
- Chapter 6, "Patterns for understanding frameworks " (p. 81), presents an unified set of patterns (best practices) for framework understanding.
- Chapter 7, "Collaborative learning with DRIVER" (p. 109), presents the implemented platform to support the collaboratively learning of frameworks.

Part 3: Validation & conclusions. The third part presents a (quasi-)experiment for the validation of the thesis and the conclusions of the dissertation:

- Chapter 8, "Academic quasi-experiment" (p. 147), addresses validation issues through a controlled experimental environment.
- Chapter 9, "Conclusions" (p. 181), drafts the main conclusions of this dissertation, and points to further work.

For a comprehensive understanding of this dissertation, all the parts should be read in the same order as they are presented. Those already familiar with program comprehension and frameworks, who only want to get a fast but detailed impression of the work, may skip the first part, and go directly to Chapter 5 (p. 69).

Some typographical conventions are used to improve the readability of this document. Pattern names always appear in SmallCase style. Relevant concepts are usually introduced in *italics*. Book titles and acronyms are type-faced in ALLCAPS. References and citations appear inside [square brackets] and in highlight color — when viewing this document in a computer, these will also act as *hyperlinks*.

Part I

State of the art

Chapter 2

Program comprehension

Program comprehension research can be characterised by both the theories that provide rich explanations about how programmers comprehend software as well as the tools that are used to assist in comprehension tasks.

Ever since the time of the first software engineering workshop [NAT68], challenges in understanding programs became ever present. As such, the field of program comprehension has evolved considerably as a research discipline. The main goal of the community is to build an understanding of these challenges, with the ultimate objective of developing more effective tools and methods that support them [Sto05].

This research has been rich and diversified, with various shifts in paradigms and research cultures during the last decades. A plethora of differences in program characteristics, programmer skills, and software tasks have led to many diverse theories, research methods and tools.

Consequently, there is, today, a wide variety of theories that provide rich explanations of how programmers understand programs and can provide advice on how program comprehension tools and methods may be improved.

This section presents an overview of existing comprehension theories, models and methods, as a way to set the wider boundaries of the research work presented in this dissertation. Figure 2.1 gives an overall depiction of the main topics.

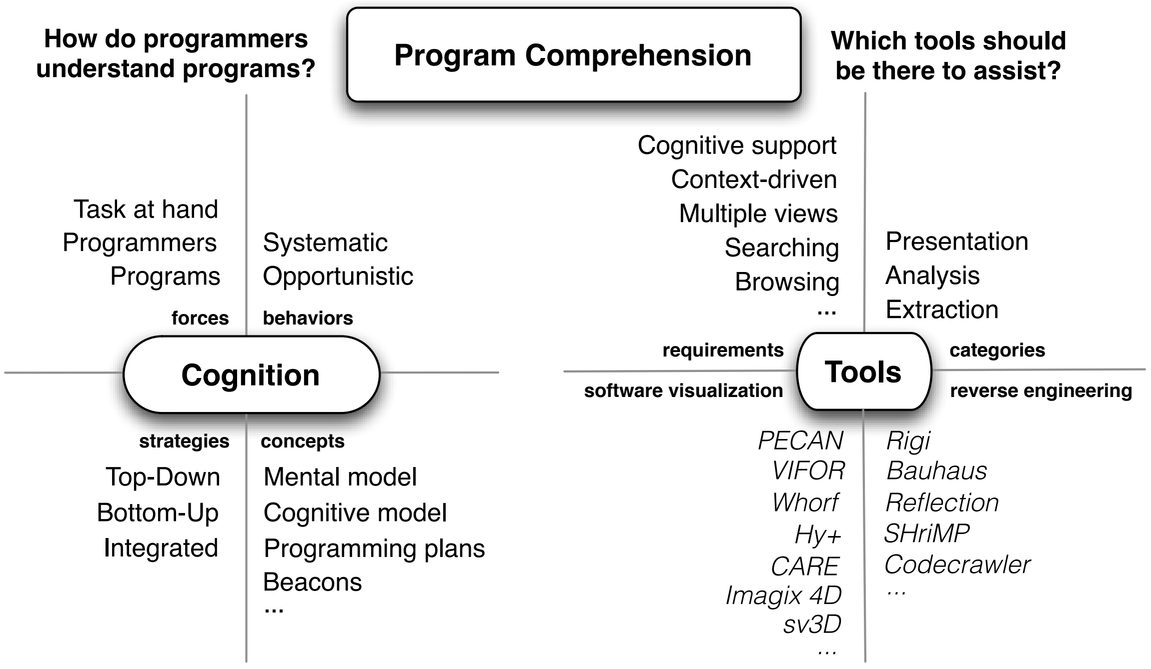


Figure 2.1: Overview of the program comprehension key research topics.

2.1 Cognitive theories and models

At its inception, experiments in the field were done without theoretical frameworks to guide the evaluations, and thus it was not possible to compare validation approaches [Dét01].

As the lack of theories was being recognised as problematic, methods and theories were borrowed from other areas of research, such as text comprehension, problem solving and education. These took the role of *building blocks* that led to the development of cognitive theories about how programmers understand programs and ways of building supporting tools. These theories brought rich explanations of behaviours that would lead to more efficient processes and methods as well as improved education procedures [Hoh96].

2.1.1 Concepts

A few common concepts incorporate the theories behind program comprehension and that are present along this chapter and referred throughout this dissertation. These concepts can be used as nomenclature when describing theories, therefore, for the sake of clarity, their name[1], definition and how they relate is briefly addressed, thus:

- A *mental model* describes a developer's mental representation of the program to

[1] in *italic*

be understood whereas a *cognitive model* describes the cognitive processes and temporary information structures in the programmer's head that are used to form the mental model.

- *Programming plans* are generic fragments of code that represent typical scenarios in programming. For example, a `sorting program` will contain a loop, which compares two numbers in each iteration, or `visiting a structure` will have a loop going through all its elements [SE84].
- *Beacons* are recognisable, familiar features in the code that act as clues to the presence of certain structures [Bro83].
- *Rules of programming discourse* capture the conventions of programming, such as coding standards and algorithm implementations [SE84].

Then there are strategies and behaviours. Strategies describe how a programmer interacts with the program in order to build her *mental model.* Behaviours describe how strategies are applied and how the programmer can shift between them.

2.1.2 Top-down comprehension strategy

Two main theories emerged that support a top-down comprehension strategy.

Brooks [Bro83] suggested that programmers understand a completed program in a top-down way, where the comprehension process relies on reconstructing knowledge about the application domain and mapping that to the source code. The process starts with a hypothesis about the general nature of the program, which is then refined hierarchically, by forming secondary hypothesis. These are then refined and evaluated in a depth-first manner, whose verification (or rejection) depends heavily on the absence or presence of *beacons.*

Soloway and Ehrlich [SE84] observed that top-down understanding is used when the code or type of code is familiar. They observed that expert programmers use *beacons*, *programming plans* and *rules of programming discourse* to decompose goals and plans into lower-level plans. They noted that *delocalised*[2] *(programming) plans* complicate program comprehension.

2.1.3 Bottom-up comprehension strategy

The bottom-up theory of program comprehension proposed by Shneiderman and Mayer [SM79] assumes that programmers first read code statements and then mentally chunk or

[2] Delocalised, meaning, scattered throughout the source code.

group these statements into higher-level abstractions. These abstractions (chunks) are aggregated further until a high-level understanding of the program is attained. They differentiate between syntactic and semantic knowledge of programs: syntactic knowledge is language dependent and concerns the statements and basic units in a program; semantic knowledge is language independent and is built in progressive layers until a *mental model* is formed, which describes the application domain.

Similarly, Pennington [Pen87] also observed programmers using a bottom-up strategy initially gathering statement and control-flow information. The *mental model* was formed in two phases: Firstly, these micro-structures (statements, control constructs and relationships) were chunked and cross-referenced by macro-structures (text-structure abstractions) to form a program model. Subsequently, a situation model was formed, also bottom-up, using application-domain knowledge to produce a hierarchy of data-flow and functional abstractions (the program goal hierarchy).

2.1.4 Systematic and opportunistic behaviours

Littman et al [LPLS86] observed that programmers use either: (1) a *systematic* approach, reading the code in detail and tracing through control and data-flow, or (2) they use an *as-needed* (opportunistic) approach, focusing only on the code related to the task at hand. Subjects using a systematic behaviour acquired both static knowledge (information about the structure of the program) and causal knowledge (interactions between components in the program when it is executed). This enabled them to form a *mental model* of the program. However, those using the opportunistic approach only acquired static knowledge resulting in a weaker *mental model* of how the program worked. More errors occurred since the programmers failed to recognize casual interactions between components in the program.

Soloway et al. [SPL+88] combined these two theories as macro-strategies aimed at understanding the software at a more global level. In the systematic macro-strategy, the programmer traces the flow of the whole program and performs simulations as all of the code and documentation is read. However, this strategy is less feasible for large programs. In the more commonly used opportunistic macro-strategy, the programmer looks at only what she thinks is relevant. However, more mistakes could occur since important interactions might be overlooked.

2.1.5 Integrated comprehension

Von Maryhauser and Vans [vMV95] combined the top-down and bottom-up strategies, and systematic/opportunistic behaviours into a single metamodel. In their experiments, they observed that some programmers frequently switched between all the approaches. They formulated an integrated metamodel where understanding is built concurrently at several levels of abstraction by freely switching between the several types of comprehension strategies and behaviours.

The model consists of four major components. The first three components describe the comprehension processes used to create mental representations at various levels of abstraction and the fourth component describes the knowledge-base needed to perform a comprehension process:

- The top-down (domain) model is usually invoked and developed using an opportunistic strategy, when the programming language or code is familiar. It incorporates domain knowledge as a starting point for formulating hypotheses.

- The program model may be invoked when the code and application is completely unfamiliar. The program model is a control-flow abstraction.

- The situation model describes data-flow and functional abstraction in the program. It may be developed after partial program model is formed using systematic or opportunistic strategies.

- The knowledge base consists of information needed to build these three *cognitive model*s. It represents the programmer's current knowledge and is used to store new and inferred knowledge.

2.1.6 Factors affecting comprehension strategies

The general opinion most researchers realise is that certain factors will influence the comprehension strategy adopted by a programmer [SFM97] [SD96]. These factors also explain the apparently wide variation in the comprehension strategies discussed in th previous sections. The variations are primarily due to:

- **Differences among programs**, such as paradigm, size, syntax, etc.

- **The task at hand**, whether it's designing, maintaining, evolving, etc.

- **Varied characteristics of programmers**, such as experience, skill, creativity, etc.

To evaluate how programmers understand programs, these factors must be considered and are further explored in sections 2.2.1 (p.22), 2.2.3 (p.24) and 2.2.4 (p.24).

With experience, programmers tend to adopt the most effective strategy for the given program and task. A change of strategy may be needed because of some anomaly of the program or the requested task. Program understanding tools should enhance or ease the programmer's preferred strategies, instead of imposing a fixed strategy which may not always be suitable.

2.2 Program, task and programmers

Both program and programmer influence a comprehension strategy choice by their inherent and varied characteristics. Additionally, this choice also depends on the task at hand. This section debates these issues giving an insight on the subject, available studies and trends for future research.

2.2.1 Program characteristics

Programs that are carefully designed and well documented will be easier to understand, change or reuse in the future. Nevertheless, experiments have shown that the choice of language has an effect on the comprehension processes [Pen87] [PBT97] [CW01]. For instance, given the nature and syntax of the programming language, COBOL programmers consistently fared better at answering question related to data-flow than FORTRAN programmers, while these would fare better at control-flow questions than their counterparts.

Also the paradigm of a programming language is a relevant factor. Object-oriented (OO) programs, in comparison with procedural programs, are often seen as a more natural fit to problems in real world because of "is-a" and "is-part-of" relationships in a class hierarchy and structure. Others, however, argue that objects do not always map easily to real world problems [Dét01]. In OO programs, abstractions are achieved through encapsulation and polymorphism. Message passing is used for communication between class methods and hence *programming plans* are dispersed (i.e., scattered) throughout classes.

2.2.2 Program trends

As new techniques and programming paradigms emerge and evolve, the comprehension process must shift to embrace these changes [Sto05]. New characteristics on both program and programming approaches seem to produce new trends for comprehension research, such as:

- **Distributed applications.** Along with web-based applications, both are becoming more prevalent with technologies such as .NET, J2EE, and web services (with already conducted studies of their impact in comprehension [GB04]). One programming challenge that appeared recently and increased rapidly is the combination of different paradigms in distributed applications, e.g., a client side script sends XML to a server application (which currently evolved to the AJAX [Gar05] technology).

- **Higher levels of abstraction.** Visual composition languages for business applications are also increasing. As the level of abstraction increases, comprehension challenges are shifting from code understanding to more abstract concepts. Model-driven architectures [OMG10] and adaptive-object modeling [YBJ01] are emerging concepts that dwell at a higher level of abstraction.

- **New programming paradigms.** The advent of Aspect-Oriented Programming (AOP) [KLM+97] and Feature-Oriented Software Development (FOSD) [AK09] caused some stir in the programming community. AOP introduced *aspects* as a construct to manage scattered concerns (*delocalized plans*) in a program and have proved to be effective for managing many programming concerns, such as logging [AG08] and security [SK10]. FOSD arose as paradigm for construction, customization and synthesis of large-scale software systems, where the concept of *feature* serves to describe the commonalities and variabilities of software systems, covering all the development steps. However, it is not clear how both *aspects* and *features* written by others will improve program understanding, especially in the long term. Despite some studies [ASFF11] [FKAL09], more empirical work is needed to validate the assumed benefits of these paradigms [AKT07].

- **Improved software engineering practices.** The more informed processes that are used for developing software today will hopefully lead to software that is easier to comprehend in the future. Component-based software systems are currently being designed using familiar design patterns [GHJV95] [BMR+96] and other conventions [OMG11] [OMG]. Development processes are more lightweight (*agile* [BBvB+]), self-aware (more metrics are collected [McA00]) and self-improving (retrospectives [DL06]). Future software may have traceability links to requirements and improved documentation such as formal program specifications [BLS05]. Also, future software may have autonomic properties [Fua07], where the software self-heals [HOB05] and adapts as its environment changes – thus in some cases reducing time spent on maintenance.

- **Diverse sources of information.** The program comprehension community, until quite recently, mostly focused on how static and dynamic analysis of source code, in conjunction with documentation, could facilitate program comprehension. Modern software integrated development environments, such as the Eclipse Java development environment [Ecl11], NetBeans [Net], or Visual Studio [Cord], also manage other kinds of information such as bug tracking, test cases, and version control. This information, combined with human activity information such as emails, instant messages and relevant social networking data, will be more readily available to support analysis in program comprehension. Domain information should also be more accessible due to model-driven development [AK03] and the semantic web [BLHL01].

2.2.3 Task characteristics

Studies related to the impact the task at hand has on program comprehension are focused, primarily, on interaction history (whether empirically [KM05] [DKCR05] or with analysis tools [RL07] [SG07]) and performing change or maintenance tasks [KMCA06] [MVS09]. The overall conclusions confirm the impact each type of task have on comprehension [YR11], and provide guidelines for requirements that assisting tools (further explored in section 2.3.2, p.29) might have to cover, in order to mitigate this impact:

- Development environments should provide visualization tools that aid in the task process. Without prescribing a workflow, the IDE should present the user with the information necessary to facilitate the task operation, e.g., a program trace, if debugging, or a structured-like visualization of the components or files when editing or navigating.
- Usage patterns (editing, navigating, searching) have been identified and should be supported (whether through documentation or tools) [ZGH07].
- Task shifting should be supported. Users tend to shift between related tasks which share similar contexts, as a strategy for a more speedier conclusion of tasks [KM05].

Future research tends to focus mainly on how users cope with tasks in emerging visualization technologies and how that impacts comprehension and effective task completion.

2.2.4 Programmer characteristics

There are many individual characteristics [PJ05] [Ben05] that will impact how a programmer tackles a comprehension task. These differences also impact the requirements for a supporting tool. There is a huge disparity in programmer ability and creativity, which cannot be measured simply by their experience. The entire environment behind the comprehension task affects the programmer, where the task complexity, available tools and time pressure are constraining factors. The way a programmer tackles with each of these issues, depends not only on experience, but also on personality, state of mind and motivation.

In her work, Vessey [Ves85] presents an exploratory study to investigate **expert** and **novice**'s debugging processes. She classified programmers as expert or novice based on their ability to chunk effectively. She notes that experts used breadth-first approaches and at the same time were able to adopt a system view of the problem area, whereas novices used breadth-first and depth-first approaches but were unable to think in system terms.

Détienne [Dét01] also notes that experts make more use of external devices as memory aids. Experts tend to reason about programs according to both functional and object-oriented relationships and consider the algorithm, whereas novices tend to focus on objects.

2.2.5 Programmer trends

As with everything else, programmers also adapt and evolve, trying to keep up with the paradigm shifts and the new trends in their development environment [Sto05]. Relevant trends are:

- **Program comprehension everywhere.** The need to use computers and software is present in our daily life. Programming, and hence program comprehension, is no longer a niche activity. Scientists and knowledge workers have to use and customise software to help them do science or other work. Scientists from diverse fields (medicine, astronomy, economics, etc.), are using and developing sophisticated software without a formal education in computer science. Consequently, there is a need for techniques to assist in non-expert and end-user program comprehension. Fortunately, there is much work on this area, especially at conferences such as Visual Languages [VLH11] and the PPIG [PPI11] group, where they investigate how comprehension can be improved through tool support for spreadsheet and other end user applications.

- **Sophisticated users.** Currently, advanced visual interfaces are not often used in development environments. A large concern by many tool designers is that these advanced visual interfaces require complex user interactions. However, tomorrow's programmers will be more familiar with game software, multitouch technologies and other media that displays information rapidly and requires sophisticated user controls. Consequently, the next generation of users will have more skill at interpreting information presented visually and at manipulating and learning how to use complex controls.

- **Globally distributed teams.** Advances in communication technologies have enabled globally distributed collaborations in software development. Distributed open source development is having an impact on industry. Some notable examples are Linux and Eclipse. Some research has been conducted studying collaborative processes in open-source projects [MFH02] [GRS04], but more research is needed to study how distributed collaborations impact comprehension.

2.3 Tools for program comprehension

Understanding a software program is often a difficult process because of missing, inconsistent, or even too much information. The source code often becomes the only knowledge source on how the system works. The field of program comprehension research has resulted in many diverse tools to assist in program comprehension [SS00]. When developing such tools, experts bring knowledge from other fields of research as Software Visualisation [Sof11] and Reverse Engineering [WCR11] as means to cover the researched requirements § 2.3.2 (p. 29). This section provides insight over the studies made to improve tool development to assist on program comprehension.

2.3.1 Tool requirements studies

Which features should an ideal tool have to efficiently support program comprehension? Needless to say that these tools will only play a supporting role to other software engineering tasks, such as design, development, maintenance, and (re)documentation.

There are mainly two ways of conducting studies to discover effective features to support program comprehension: (1) an empirical approach by observing programmers trying to understand programs [PSK07] and (2) an approach based on personal experience and intuition. Given the variability in comprehension settings, both approaches contribute to answering this complex question. As such, several studies already conducted by several authors revealed a number of tool requirements, as follows.

Biggerstaff

Biggerstaff [BMW93] notes that one of the main difficulties in understanding comes from mapping what is in the code to the software requirements – he terms this the concept "assignment problem". Although automated techniques can help locate programming concepts and features, it is challenging to automatically detect human oriented concepts. The user may need to indicate a starting point and then use slicing techniques to find a related code. It may also be possible for an intelligent agent (that has domain knowledge) to scan the code and search for candidate starting points. From his research prototypes, he found that queries, graphical views and hypertext were important tool features.

Maryhauser and Vans

Von Maryhauser and Vans [vMV93], from their research on the Integrated Metamodel, made an explicit recommendation for tool support for reverse engineering. They determined basic information needs according to cognitive tasks and suggested the following tool capabilities to meet those needs:

- Top-down model: online documents with keyword search across documents; pruning of call tree based on specific categories; smart differencing features; history of browsed locations; and entity fan-in.
- Situation model: provide a complete list of domain sources including non-code related sources; and visual representation of major domain functions.
- Program model: Pop-up declarations; online cross-reference reports and function count.

Singer and Lethbridge

Singer and Lethbridge [SLVA97] also observed the work practices of software engineers. They explored the activities of a single engineer, a group of engineers, and considered company-wide tool usage statistics. Their study led to the requirements for a tool that was implemented and successfully adopted by the company. Specifically they suggested tool features to support "just-in-time comprehension of source-code". They noted that engineers, after working on a specific part of the program, quickly forget details when they move to a new location. This forces them to rediscover information at a later time. They suggest that tools need the following features to support rediscovery:

- Search capabilities so that the user can search for code artifacts by name or by pattern matching.
- Capabilities to display all relevant attributes of the items retrieved as well as relationships among items.
- Features to keep track of searches and problem-solving sessions, to support the navigation of a persistent history.

Erdös and Sneed

Erdös and Sneed [ES98] designed a tool to support maintenance following many years of experience in the maintenance and reengineering industry. They proposed that the following seven questions needed to be answered, for a programmer to maintain a program that is only partially understood:

- Where is a particular subroutine/procedure invoked?
- What are the arguments and results of a function?
- How does control flow reach a particular location?
- Where is a particular variable set, used or queried?
- Where is a particular variable declared?
- Where is a particular data object accessed?
- What are the inputs and outputs of a module?

Murray and Lethbridge

Murray and Lethbridge [ML05] observed software professionals using a mixed approach, combining elements from specific methods used in software engineering empirical research and a sociological qualitative research called "ground theory". From this approach, they were able to develop the basis for a theory of the ways people think when explaining and comprehending software, which they called "cognitive patterns". These patterns can then be applied to further empirical observatory studies as a roadmap to capture programmer behavior.

Zayour

Zayour [ZL00] proposes a methodology for assessing cognitive requirements and adoption success for reverse engineering tools, from which he concludes five main rules of thumb:

- A clear and realistic definition of the problem space to be targeted is a must.
- Direct observation of the targeted user is required to form a realistic perception of users problems and tasks.
- Tool designers should document their perception of the user's problems and tasks.
- When determining the success of a tool, cognitive load is a more important indicator to measure than elapsed time (because it affects adoptability more).
- Design should be aimed at satisfying cognitive requirements and thus should be guided by cognitive principles.

Work by other authors included recall tests to evaluate the ability to answer questions regarding a piece of code programmers studied for a limited period of time [Pen87]. Subjective ratings [Shn77] have been used to measure different levels of comprehension. Additionally, other studies may ask subjects to label or group different code members based on the similarity of their functionalities [Ris86]. Soloway and Erlich [SE84] asked programmers to fill blank lines and complete unfinished programs on paper in an unfamiliar source code without providing specifications about the program's use or functionality. Similarly, Bertholf et al. [BS93] asked novice developers to complete incomplete literal programs on paper. Additional techniques to measure program comprehension involved completing incomplete call graphs, modifying existing code, report a bug, or separate source code from two different algorithms [Shn77]. Other studies were conducted and were already referred to in § 2.2.3 (p. 24).

2.3.2 Tool requirements

From the studies presented and derived from cognitive theories, Storey [Sto05] extracted and synthesized several tool requirements that still guide tool researchers today:

- **Browsing support.** The top-down process requires browsing from high-level abstractions or concepts to lower-level details, taking advantage of *beacons* in the code; bottom-up comprehension requires following control-flow and data-flow links, both novices and experts can benefit from tools that support breadth-first and depth-first browsing; and the Integrated Metamodel suggests that switching between top-down and bottom-up browsing should be supported. Flexible browsing support also will help to offset the challenges from *delocalised plans*.

- **Searching.** Tool support is needed when looking for code snippets by analogy and for iterative searching. Also inquiry episodes should be supported by allowing the programmer to query on the role of a variable, function, etc.

- **Multiple views.** Programming environments should provide different ways of visualising programs. One view could show the message call graph providing insight into the *programming plans*, while another view could show a representation of the classes and relationship between them as an object-centric or data-centric view of the program. These orthogonal views, if easily accessible, can facilitate comprehension, especially when combined.

- **Context-driven views.** The size of the program and other program metrics will influence which view is the preferred one to show a programmer browsing the code for the first time. For example, in an object-oriented program, it is usually preferable to show the inheritance hierarchy as the initial view. However, if the inheritance hierarchy is flat, it may be more appropriate to show a call graph as the default view.

- **Additional cognitive support.** Experts need external devices and scratchpads to support their cognitive tasks, whereas novices need pedagogical support to help them access information about the programming language and the corresponding domain.

These requirements serves as basis for assessment of features in existing and future tools [SCG05] [GCS05]. Existing IDEs such as Eclipse, NetBeans or Visual Studio cover many of these features, but researchers keep pushing the envelope to improve the existing tools even further and coming up with new approaches as new needs arise and paradigms shift.

2.3.3 Tool development

Programming comprehension tools can be roughly grouped into three categories, according to function [SD96]:

- **Extraction** tools include parsers and data gathering tools.
- **Analysis** tools do static and dynamic analysis to support activities such as clustering, concept assignment, feature identification, transformations, domain analysis, slicing and metrics calculation.
- **Presentation** tools include code editors, browsers, hypertext and visualisations. They are strongly linked to research in software visualisation.

Integrated software development (IDEs) and reverse engineering environments will usually have some features from each category. The set of features they support is usually determined by the purpose for the resulting tool or by the focus of the research. As such, these three categories can be fully incorporated into two major research areas: Software Visualisation (*presentation* tools) and Reverse Engineering (*extraction* and *analysis* tools), briefly presented next.

Software visualisation tools

Software visualisation tools and browsing tools provide information that is useful for program understanding. These tools use graphical and textual representations for the navigation, analysis and presentation of software information to increase understanding.

Mixed results have been reported through the literature on the role of text and graphics for program comprehension. While Green and Petre [GP92] observed that text was faster than graphics for experimental program comprehension tasks, Scanlan [Sca89] reported an improvement using graphical visualisations when comparing textual algorithms and structured flowcharts. Petre [PBT97] attributes the difficulty in understanding program visualisations to the fact that graphical representations have fewer navigational cues, namely secondary notations, when compared to program text: source code implies a serial inspection strategy. Moreover, she observed that experienced readers tend to use parallel textual and graphical information whenever available to assist their comprehension process: they use text as a main source to guide their understanding of graphical representation.

Several software visualisation tools show animations to teach widely used algorithms and data structures [M.H91] [SBdL92] [SFC94]. Another class of tools shows dynamic execution of programs for debugging, profiling and for understanding run-time behaviour [ISO87] [RCWP91]. Other software visualisation tools mainly focus on showing textual representations, some of which may be pretty printed to increase understanding [ea95] [HIBK97], use hypertext in an effort to improve navigability [PBT97] and annotations to communicate semantics [OH03]. Other approaches filter source code presentation, showing delocalised, related pieces of code in a bind, fluid manner [DSE06].

Many tools present relevant information in the form of a graph where nodes represent software objects and arcs show the relations between the objects. This method is used by PECAN [Pen87], Rigi [MK88], VIFOR [RDL90], Whorf [BGS92], CARE [LAD$^+$94], Hy+ [MS95], Imagix4D [Corb] and G^{SEE} [Fav01]. Other tools use additional pretty printing techniques or other diagrams to show structures or information about the software. For example, the GRASP tool uses a control structure diagram to display control constructs, control paths and the overall structure of programming units [SFM97]. More extensive surveys and analysis on these kind of tools can be found at [BK01] and [Pac05].

Reverse engineering tools

Reverse Engineering focuses on how to extract relevant knowledge from source code and present it in a way that facilitates comprehension. Several studies conducted in the past have proposed solutions on how to overcome caveats in the program comprehension process. As seen in § 2.3.1 (p. 26), Maryhauser and Vans [vMV93], Singer and Lethbridge [SLVA97], Zayour [ZL00], and others, have given their insight on how to address tool development for reverse engineering of useful information to assist on program understanding. Wong [Won00] also discusses reverse engineering tool features. He specifically mentions the benefits of using a “notebook” to support ongoing comprehension.

Usually, the reverse engineering tools and techniques associated to program comprehension are bundled into broader development environments where other types of tools also co-exist.

It is possible to examine each of these environments and to recover the motivation for the features they provide by tracing back to the cognitive theories. For example, the Rigi system [MK88] has support for multiple views, cross-referencing, and queries to support bottom-up comprehension. The Reflection tool [MNS95] has support for the top-down approach through hypothesis generation and verification. The Bauhaus tool [TKS01] has features to support clustering (identification of components) and concept analysis. The SHriMP tool [Sto03] provides navigation support for the Integrated Metamodel, i.e, frequent switching between strategies. And the Codecrawler tool [LD01] uses visualisation of metrics to support understanding of an unfamiliar system and to identify bottlenecks and other architectural features. A more extensive survey of tools that use dynamic analysis for program comprehension can be found at [CZvD+09].

All these tools combine reverse engineering tasks with software visualisation techniques to improve program comprehension on different levels of abstraction, gathering information recovered or simply mined together into user-friendly viewed chunks of valuable data for the programmer. Similarly to that stated in § 2.3.2 (p. 29), most of these techniques have been adopted and assimilated into popular IDEs, where the developers can configure their own programming environment and tailor it to their comprehension needs.

2.3.4 Tool trends

The forthcoming breakthroughs in tool technology seem promising as research and evaluation methods and theories become more relevant to end-users doing programming-like tasks. Therefore, directions in tool evolution appear to follow several guidelines presented next [Sto05].

- **Faster tooling and integration.** The use of frameworks as an underlying technology for software tools is leading to faster tool innovations, as less time needs to be spent reinventing the wheel. A prime example of how frameworks can improve tool development is the Eclipse platform [Ecl11]. Eclipse was specifically designed with the goal of creating reusable components, which would be shared across different tools. Given a suite of tools that all plug in to the same framework, together with a standard exchange format (such as GXL), researchers will be able to more easily try different combinations of tools to meet their research needs. This should result in increased collaborations and more relevant research results. Such integrations will lead to fewer disruptions for programmers and improve accessibility to repositories of information related to the software. These include code, documentation, analysis results, domain information and human activity information.

- **Recommenders and search.** Recommender systems are being proposed to guide navigation in software spaces. Examples of such systems include Mylar [KM05] and NavTracks [SES05]. Mylar, (now called MyLyn, and bundled with the Eclipse Platform) uses a degree of interest model to filter non-relevant files from the file explorer and other views in Eclipse. NavTracks provides recommendations of which files are related to the currently selected files. Deline et al. also discuss a system to improve navigation [DKCR05]. Wiemer et al. improve code recommendation through collaborative filtering [WKB09].The FEAT tool suggests using concern graphs (explicitly created by the programmer) to improve navigation efficiency and enhance comprehension [RM03]. Search technologies, such as Google and most specifically Codase [Cod], show much promise at improving search for relevant components, code snippets and related code. The Hipikat tool [CMSB05] recommends relevant software artifacts based on the developer's current project context and development history. The Prospector system recommends relevant code snippets [MC94]. It combines a search engine with the content assist in Eclipse to help programmers use complex APIs. All this work shows much promise and it is expected to improve navigation in large systems while reducing the barriers to reuse components from large libraries.

- **Adaptive Interfaces.** Software tools typically have many features, which may be overwhelming not only for novice users, but also for expert users. This information overload could be reduced through the use of adaptive interfaces. The idea is that the user interface can be tailored automatically, i.e., will self-adapt, to suit different kinds of users and tasks. Adaptive interfaces are now common in Windows applications such as Word. Eclipse has several novice views (such as Gild [Sto03] and Penumbra [Pen]) and Visual Studio has the Express configuration for new users. However, neither of these mainstream tools currently have the ability to adapt nor even to be easily manually adapted to the continuum of novice to expert users, in the sense of seamlessly accompanying the user experience progression.

- **Visualisations.** As already seen in § 2.3.3 (p. 30), these have been subject of much research over the past years. Many visualisations, and in particular graph-based visualisations, have been proposed to support comprehensions tasks. Other examples include Seesoft [BE96], Bloom [Rei01], Landscape views [Pen92], and sv3D [MFM03]. Graph visualisation is used in many advanced commercial tools such as Klocwork [Klo], Imagix4D [Ima] and Together [Tog11]. UML diagrams are also commonplace in mainstream development tools [Ent] [Ecl]. 3D visualisations are also being explored in tools such as CodeCity [WL07]. One challenge with visualising software is scale and knowing at what level of abstraction details should be shown, as well as selecting which view to show. Gaucho [OLDR11] allows the programmer to manipulate code as depictions of object-oriented constructs, instead of just plain text. More details about the user's task combined with metrics describing the program's characteristics (such as inheritance depth) will improve how visualisations are currently presented to the user. A recommender system could suggest relevant views as a starting point. Bull proposes the notion of model-driven visualisation [BS05]. He suggests creating a tool for tool designers and expert users that recommends useful views based on characteristics of the model and the data.

- **Collaborative support.** As software teams increase in size and become more distributed, collaborative tools to support distributed software development activities are more crucial. In research, there are several collaborative software engineering tools being developed such as Jazz and Augur [HCRP04] [FD04]. There are also some collaborative software engineering tools deployed in the industry, such as CollabNet [Col], Alfresco [Alf], or Jira [Jir], but they tend to have simple tool features to support communication and collaboration, such as version control, email and instant messaging. Current industrial tools lack more advanced collaborative features such as shared editors [Sub], and research falls short on providing empirical work to improve these tools. Another area for research that may prove useful is the use of large screen displays to support co-located comprehension. O'Reilly et al. [OBM05] propose a war room command console to share visualisations for team coordination. Guzzi et al. [GHL+11] propose collective code bookmarks, as a non-intrusive bookmarking tool that facilitates knowledge sharing. There are other research ideas in the CSCW field that could be applied to program comprehension, and which overview can be read in Chapter 4 (p. 53).

- **Domain and pedagogical support.** The need to support domain experts that lack formal computer science training will necessarily result in more domain-specific languages and tools. Non-experts will also need more cognitive scaffolding to help them learn new tools, languages and domains more rapidly. Pedagogical support, such as providing examples by analogy, will likely be an integral part of the future software tools. Technologies such as TXL [CDMS02] can play a role in helping a user see examples of how code constructs in one language would appear in a new language.

2.4 Summary

Program comprehension may come in many ways. Research shows us that understanding a program relies on a model or process (top-down, bottom-up, systematic, opportunistic, etc.). This process is chosen or tailored according to three aspects: the programmer, with its specific characteristics (experience, personal skills, etc.); the program, how it is presented and how it is structured (object-oriented, functional, aspect-oriented, modular, etc.); and the task at hand (reuse, maintenance, evolution, etc). The future trends predict a generalised, sophisticated and distributed comprehension of programs, relying on new media and the ongoing expansion of the information highways. Program comprehension is becoming global and collaborative.

Tools try to support the cognitive processes using several approaches. Whether through reverse engineering or software visualisation techniques, tools provide features that assist the developer in the understanding process that was chosen. Tools can be categorised in three major groups of features: extraction, analysis and presentation. One major concern that tools try to uphold is its flexibility to incorporate several comprehension strategies. This way, the developer can adapt the tool to her cognitive requirements. Tools are evolving to incorporate new paradigms and technologies and emphasising pedagogical concerns and a more effective knowledge transfer.

Chapter 3

Framework understanding

Program comprehension covers a wide range of sub-areas when it comes to comprehend programs. By *programs*, we mean software artifacts: constructs built upon source-code. A framework can be considered one of such artifacts and, due to its importance and growing adhesion by the software community, became a relevant research topic[1]. This topic deals with the issues behind using, implementing and evolving a framework, and the understanding and learning required to do so.

Object-oriented frameworks are a powerful form of reuse but they can be difficult to understand and reuse correctly. They are promoted as having the potential to provide the benefits of large-scale reuse [GHJV95] [BMR+96] [FSJ99]. While practical evidence does suggest that framework usage can increase reusability and decrease development effort [MN96], experience has identified a number of issues that hinder framework application and limit potential benefits [BMMB99]. One of the major challenges is effective framework understanding – a specialised kind of program comprehension.

Over the past twenty years, a large range of candidate documentation techniques has been proposed to support framework understanding, including design patterns [GHJV95], pattern languages [Joh92], cookbooks [KP88], hooks [FHLS97], exemplars [GM95] and minimalist documentation [Agu03].

Still, there was a lack of insight into problems that limit the comprehension and reuse of software frameworks. There was no true awareness of the impact these techniques

[1] As the, henceforth, referred literature will show.

had on framework understanding. As such, a few studies were conducted and its results identified some concerns and basis for future research. The next section will briefly address some of these studies, and, afterwards, present a brief review of some existing tools and approaches to aid in framework understanding and reuse. An overall depiction of the main issues behind framework understanding is shown in Figure 3.1.

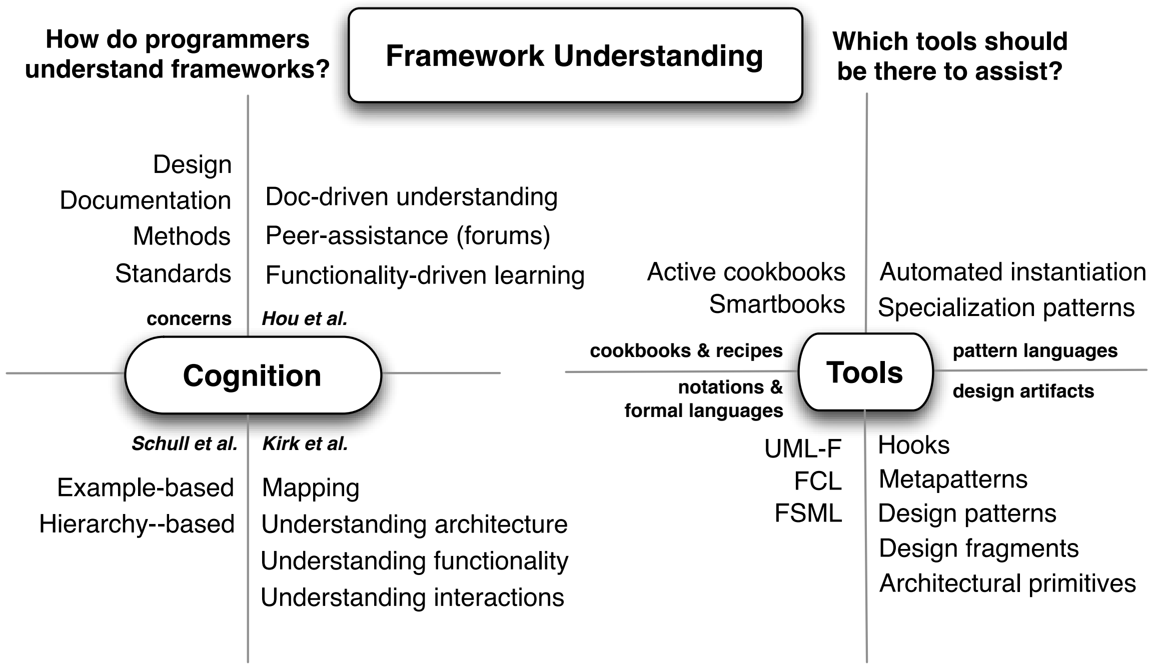


Figure 3.1: Overview of the framework understanding research topic.

3.1 Studies on framework understanding

There is a considerable amount of literature about frameworks, but it scarcely deals with the identification of reuse problems or evaluation of strategies to support and assist the framework *actor* (user, developer, evolver). There are tools that address topics under the realm of framework building, design recovery and documentation, and only a few deal specifically with framework instantiation. Overall, there is not a clear emphasis or study of the overall symptoms and problems behind ineffective framework reuse. Nevertheless, a few studies can be found that deal with these issues.

Johnson

In [Joh97], Jonhson identified three important areas for framework documentation to address: purpose, how to use, and design. He argued that the purpose of the framework and its constituent parts should be communicated so that developers may select the correct parts for a task. While knowledge of how those parts are expected to operate allows them to be employed correctly, a description of the underlying design provides developers with an understanding of how to adapt and extend the framework in a manner consistent with the existing structure.

Fayad and Schmidt

In [FSJ99], the authors claimed that different alternatives could improve framework understandability:

- Refining the framework's internal design.
- Using methods that can ensure a successful development and usage of frameworks.
- Adhering to standards for framework development, adaptation, and integration.
- Producing comprehensible framework documentation.

These guidelines are mainly preventive and don't focus on the issue of reusability, posing merely as general advices. Nevertheless, they can be relevant as rules of thumb for framework development and maintenance.

Butler, Keller and Milli

In [BKM00], the authors described a taxonomy of framework documentation primitives that appear to address reusability issues. They described six primitives, which emphasise the need for information about class interfaces and communication protocols between classes.

Schull et al.

In [SLB00], Schull et al. presented an evaluation of the role that examples play in framework reuse. Their study compared two approaches to framework reading and, eventually, its documentation: example-based approach and hierarchical-based approach. Their results suggested that examples are an effective learning strategy, especially for those beginning to learn a framework. They also identified potential problems with an example-based approach: finding the small pieces of required functionality in larger examples; inconsistent organisation and structure of examples; and lack of design choice rationale in example documentation. They also discussed the possibility that developers become too reliant on examples and do not understand the system at a sufficient level of detail, as to implement it effectively from scratch, if necessary.

Morisio et al.

In [MRS02], Morisio et al. conducted an empirical study in an industrial context on the production of software using a framework. The objective was to investigate quality and productivity issues and the effect of learning in framework-based object-oriented development. They observed higher quality and productivity levels in framework-based applications, due to *"a learning effect, or the improved skill of the programmer in performing a task, due to repetition of the task over time"*. Recognising that learning takes a great deal of time, they distinguished two types of learning: *operational*, that deals with speeding up repetitive operations and *conceptual*, that regards acquiring high-level knowledge. Framework user's tend to engage more on the ladder, when compared to non-framework software development. This consumes more time, but the end result is a more proficient developer.

Kirk et al.

In [KRW05], Kirk et al. conducted a research, through observation of both novice and experienced users, where they identified four fundamental problems of framework reuse:

- *Mapping* identifies the problem on translating an abstract, conceptual solution into a concrete implementation, which reuses the existing structures within the framework. Such problems were often expressed as “what should I use to represent...?” or “How do I express...?”
- *Understanding functionality* describes problems understanding what specific parts of the framework actually do. Manifestations of this problem included “How does ... work?”, “Where ... does happen?” or “Where is ... defined/created/called?”
- *Understanding interactions* focuses on problems concerning the communication between classes in the framework (“What happen if ...?” or “Where should I put ...?”). Such problems are significant because of hidden or subtle dependencies within the framework that may cause failures to occur elsewhere as the result of a wrongly positioned modification.
- *Understanding the framework architecture* is the problem of making modifications without giving appropriate consideration to the high-level architectural qualities of the framework. Such alterations might have no short-term effects but ultimately lead to the framework losing its flexibility.

From these problems, the authors experimented applying two known solutions they deemed the most suited to address these issues: pattern languages and micro-architectures.

Their results showed that the pattern language provided some support for mapping problems, particularly for those with no experience of the framework, by introducing key framework concepts and providing examples of framework use. However, it was clear that previous experience dominated the explicit use of the pattern language, as well as being an inhibitor to other forms of documentation as its immediacy often precludes consideration of alternative solutions.

Although the micro-architectures, used to help develop an understanding of the key interactions within the framework, seemed relatively ineffective, it was the authors' belief that documentation of this kind is necessary to address these problems in particular.

Studies by Hou

Several studies have been led by Daqing Hou, regarding framework usage and understanding.

In [HWH05], Hou, Wong and Hoover collected and analysed a sample of 300 newsgroup questions about the Java Swing framework, looking for key insights that might improve a framework's design, its tutorials, and programming practice. The main goal was, in the future, to guide the framework developers in addressing poorly design or badly documented features or problematic programmer practice. Issues regarding design, documentation and people were identified:

- *Design.* Tightly coupled variation points, delocalised concerns, confusing inherited features and excessive special cases were identified flaws in framework design.

- *Documentation.* Despite finding some documentation caveats, the authors claim that *"[In terms of comprehending software frameworks]...doc-driven understanding is more efficient than reverse engineering from source code. Source code should be the last resort for a programmer to consult; [...] [it] is just too expensive, and needlessly increases the cost of using a framework."*

- *People.* Forums[2], in general, proved to be a good communication media for debugging, learning about platform bugs and discussing design issues. Moreover, they took on an educational role, informing programmers that certain tasks required a serious effort at obtaining a deeper level of understanding of the framework, and not just quick answering with a solution.

While aiming at helping framework developers and technical writers to uncover and distill common problems, the study also unveiled issues regarding framework learning and

[2] or *fora*, plural of the latin word *forum*. In plain English the used form is preferred.

understanding: Novice learners rely on the existing documentation first, make a shallow study using available examples and, in distress, ask for help (forum).

In [Hou08], Hou investigates the effects framework design knowledge [Joh97] has on example-based framework learning [SLB00]. He studies the effectiveness of framework learning by augmenting example-based learning with up-front framework design instruction. Some[3] of his conclusions were:

- Previously instructed learners (on the framework design) exhibited stronger ability in correctly adapting example solutions.
- Initially, novice learners appeared to focus on learning functional aspects of the framework than non-functional aspects.
- A conservative reuse strategy (strictly cover the requirements) helps novice learners focus on gaining a comprehensive understanding of the example (used to learn the framework) rather than being distracted by "nice" features.

In sum, spending some time (up-front) learning about the design of the framework is beneficial to a more effective framework reuse, impacting on the application of examples, that should be functionality-driven.

In [HL11], Hou and Li revisit the newsgroups discussions about the Java Swing framework, this time, focusing on API-specific issues. Being APIs different[4] from frameworks, the authors focused mainly in API reuse issues and derived their results into future tool requirements:

- *Better communication.* Many novices were not able to ask the proper questions in natural language, whether using source-code excerpts to demonstrate their problem proved much efficient.
- *Better semantic search.* Word-based context-free search engines render too much irrelevant information, if the learner doesn't know how to perform the right query. Retrieval should be more context-oriented and include improved semantics.
- *Improved tracking of intermediate results.* When searching for web-based solutions, examples spawned several pages and learners had to spent time organising their knowledge and keeping track of their own reasoning.

[3] Deemed relevant for the scope of this dissertation.

[4] Simplistically, the main difference between a framework and an API, is that a framework controls the execution flow and provides extension points (hooks), whether the API can be just a set of utility classes.

- *IDE-integration.* Tools should be close to the developing code, serving as quick helpers and being aware of the task of the learners as to provide context-oriented knowledge and advice, whether automatically or on-demand.

All of the issues uncovered by these studies and referred in this section are summarised and depicted in the left side of Figure 3.1.

3.2 Tools and techniques for framework understanding

As for program comprehension tools § 2.3 (p. 26), the same line of thinking applies for framework understanding tools. Both subjects share the same problems and trends, yet some framework specific issues may be addressed when devising aids to framework learning and understanding.

The past and present research in this topic focuses on issues that range from uncovering design artifacts to representing processes and behaviours that might help using the framework. Mostly, the proposals converge to producing and enhancing existing documentation with adequate information that can be mined (design recovery) and represented using different formats (recipes, cookbooks), languages (patterns, beacons, idioms) and notations (textual, graphical, UML, formal languages, etc.). Next, a brief summary of these proposals is presented. The categorisation used emerged from its most relevant technique, yet several use mixed approaches, combining varied techniques to optimise their results.

3.2.1 Cookbooks and recipes

Humans are good at following step-by-step instructions if the results are guaranteed. In reusing and instantiating a framework, researchers introduced this prescriptive form (a metaphor borrowed from the cooking domain) through *cookbooks* and *recipes*, as an attempt to improve instantiation effectiveness.

Cookbooks

Confronting the challenge of communicating how to use the Model-View-Controller framework in Smalltalk-80, Krasner and Pope [KP88] built an 18-page cookbook that explained the purpose, structure, and implementation of the MVC framework. This cookbook was designed to be read from beginning to end by programmers and could also be used as a reference, but every recipe did not follow a consistent structure nor was it suitable for parsing by automatic tools.

Pree et al. provide a semi-automated tool to assist on framework instantiation, called active cookbooks [Pre95] [SSP95]. It enabled the user to enact recipe descriptions, providing an interactive interface that would guide her through the instantiation process. Although being good at providing step-by-step directions, the tool had little flexibility to cope with variations. Either the user had to follow recipes to the last detail, or not use them at all.

Smarter cookbooks

In [OC99], Ortigosa et al. proposed an improvement over the active cookbook, extending it with combination of the concept of user-tasks modelling and least commitment planning methods (*Smartbooks*). It extends traditional framework documentation with instantiation rules describing the necessary tasks to be executed in order to specialise the framework. Using these rules, a tool can be used to generate a sequence of tasks that guide the application developer through the framework specialisation process. SmartBooks provide a rule-based, feature-driven, and functionality-oriented system.

The FRamework EDitor / JavaFrames project [MHK+01] [IK02] [JK02] has developed a language for modelling design patterns and tools that act as smarter cookbooks, guiding programmers step-by-step to use a framework. As opposed to Smartbooks, this approach was pattern-based, architecture-driven, and more implementation-oriented. With the 2.0 release of JavaFrames, many of these tools work within the Eclipse IDE. Their language allows expression of structural constraints and the tool can check their conformance. Code can be generated that conforms to the patterns definition, optionally including default implementations of method bodies. Specific patterns can be related to general patterns: for example, a specific use of the Observer [GHJV95] pattern in a particular framework can be connected to its general definition.

Due to its prescriptive nature, cookbooks are prone to automation and reliable when coping with common, very specific, well-defined, well-constrained requirements. Nevertheless, they need to provide flexibility and variation through incompleteness, so that the developer can customise the framework to its own specific needs.

3.2.2 Design artifacts

A major concern in framework learning is the effective communication of design knowledge as a building block for understanding the internals of the framework. These communication vessels or artifacts can be initially present in the documentation, or tools can aid in the recovery of these elements.

Patterns

Ralph Johnson seems to be the first to suggest documenting frameworks using patterns [Joh92]. He noted that the typical user of framework documentation wants to use the framework to solve typical problems, but that cookbooks do not help the most advanced users [Joh97]. Patterns can be used both to describe a framework's design as well as how it is commonly used. He argued that the framework documentation should describe the purpose of the framework, how to use it, and its detailed design. After presenting some graduate students with his initial set of patterns for HotDraw [Bra95], he realised that a pattern isolated from examples is hard to comprehend.

Bruch et al. [BSM06] proposed the use of data mining techniques to extract reuse patterns from existing framework instantiations. Based on these patterns, suggestions about other relevant parts of the framework are presented to novice users in a context-dependent manner. They built FrUiT, an Eclipse plug-in that implements the approach and, yet at an early stage, already presents several benefits: relying on expert-written framework instantiations, there is no need to create special artifacts such as documentation or code snippets; using data mining, significant reuse rules are extracted, only concerning how to use the framework; and the tool makes automatic context search, relieving developers from explicitly searching for rules.

Hooks

Froehlich et al.'s hooks [FHLS97] focus on documenting the way a framework is used, not the design of the framework. They are similar in intent to cookbook recipes but are more structured in their natural language. The elements listed are: name, requirement, type, area, uses, participants, changes, constraints, and comments. The instructions for framework users (the changes section) read a bit like pseudo code but are natural languages and do not appear to be parsable by tools.

Metapatterns

Design patterns themselves can be decomposed into more high-level elements [Pre94]. Pree called these elements metapatterns and catalogued several of them with example usage. He proposed a simple process for developing frameworks where identified points of variability are implemented with an appropriate metapattern, enabling the framework user to provide an appropriate implementation.

From the declarative metaprogramming group from Vrije University, Tourwé and Mens [Tou02] [TM04] used Pree's metapatterns to document framework hotspots[5] and defined transformations for each framework and design patterns. Framework instances can be evolved (or created) by application of the transformations. The tool uses SOUL, a prolog-like logic language. The validation was done on the HotDraw framework by specifying the

[5] Areas of flexibility, where the framework can be configured, or code inserted, to develop the intended application.

metapatterns, patterns and transformations needed. The validation uncovered design flaws in HotDraw, despite its widespread use, along with some false positives. The declarative metaprogramming approach to modelling framework hotspots appears to have significant up-front investment before paying off, in order to provide its guarantees about a correct use of the framework. It may additionally assume a higher level of accuracy or correctness in frameworks than will commonly be found in practice. The authors commented that their approach specifically avoids design patterns in favour of metapatterns because there could be many design patterns. While this makes their technique generally applicable and composable, it will be difficult to add pattern-specific semantics and behaviour checking to their approach.

JFREEDOM [NA05] is a design recovery tool that discovers metapatterns in a framework or software system. It relies on Tourwe's formal definition of metapatterns and uses JQuery [Vol06], a logic inference-engine for Eclipse, to search the code for instances of these metapatterns. It then recommends possible GoF [GHJV95] design pattern instances based on its found metapatterns. Other design pattern recovery tools exist and a brief review of each one can be found in [NA05].

Design fragments

Fairbanks et al. [FGS06] presented a pattern language based on the notion of design fragment. A design fragment is a pattern that encodes a conventional solution to how a programmer interacts with a framework to accomplish a certain goal. It provides the programmer with a "smart flashlight" to help her understand the framework, illuminating only those parts of the framework she needs to tackle to accomplish the task at hand. They use XML to express these patterns, so that automation tools are a step away. They have analysed the 20 Java applets provided by Sun and came up with a catalogue of design fragments, which, evaluated against other 36 applets from the internet, proved that those design fragments were common and recurrent. Design fragments give programmers immediate benefit through tool-based conformance and long-term benefit through expression of design intent.

Architectural primitives

Zdun and Avgeriou [ZA05] proposed to remedy the problem of modelling architectural patterns through identifying and representing a number of architectural primitives that can act as the participants in the solution that patterns convey. According to the authors, these "primitives" are the fundamental modelling elements in representing a pattern and also they are the smallest units that make sense at the architectural level of abstraction (e.g., specialised components, connectors, ports, interfaces). Their approach relied on the assumption that architectural patterns contain a number of architectural primitives that are recurring participants in several other patterns. They chose UML as the preferred notation to represent the primitives and pretended to formalise the definitions using OCL.

All these design artifacts play their part in clarifying and communicating aspects of the framework. Their applicability depends on the framework user and his propensity to like one over the other. Contributing factors are the learning profile and needs, the task at hand and the available information. None alone can provide all the information necessary to understand the framework, although through combination and complementarity, they might prove more effective.

3.2.3 Pattern languages

In [BS00], Brugali et al. stated that:

> *"A set of patterns for a specific application domain, together with their structuring principles, becomes a high-level language, called a pattern language [AIS77]. It represents the essential design knowledge of a specific application domain, i.e., the experience gained by generations of designers in solving a class of similar problems."*

Moreover, they claimed that not only pattern languages generate frameworks, as these encompass design patterns that interrelate naturally in the application domain, but also frameworks support pattern languages because *"When a pattern language and a framework for a specific application domain are available, new applications do not have to be built from scratch, since the framework provides reusable implementations of each pattern of the pattern language."*

As already referred in the previous section, the patterns Jonhson [Joh92] suggested to document a framework are organised in a pattern language. Each pattern describes a recurrent problem in the domain covered by the framework, and then describes how to solve that problem. Its main goal is to teach how to use the framework, and then complement the task-oriented information with explanations about how the framework works, for those willing to know the details. This technique tries to strike a balance between prescriptive information (how-to-do) with descriptive information (how-it-works) as to reach a larger audience of different levels of experience.

Automated instantiation

According to Braga et al., pattern languages can be used to guide the construction [BM02a] and instantiation [BM03] of frameworks. The development of a pattern language for a specific domain relies on experience and reverse engineering of existing systems, mining recurrent problems and their solutions. Constructing a framework based on this pattern language has four main steps [BM02a] [BM02b]: hotspots identification, framework design, framework implementation, and framework validation. Instantiation is done through a tool or *wizard* that relies on the relationship between the pattern language and its associated framework, using mapping tables that link the patterns to the parts of the framework to instantiate.

Specialization patterns

In [HK06], Hautamäki et al. proposed a mining process of specialisation patterns of a framework, based on a goal-oriented approach. Instead of starting from a set of patterns that describe the usage of the framework, the authors attempted to find, specify and use the specific specialisation interface (as a set of patterns) that is directly linked to the assumed goals [And04] of the product developer. Providing a tool-supported setting, they experimented in an industrial environment, where the results were encouraging, but still required further studies to draw definite conclusions.

In order to be effective, patterns languages need to fine-tune its patterns to a large, heterogeneous audience where the experience and goals of the framework users are covered and its learning needs attended. This balance would always require dynamic adjustment, or otherwise, to be intentionally set by the user. Therefore, pattern languages need to be complemented with other techniques for a more effective learning experience.

3.2.4 Notations and formal languages

In order to cope with the specificities of frameworks, several authors have proposed new notations and formal languages to represent and communicate the mechanisms of frameworks and the constraints to which the users are bound, in order to understand and use the framework.

UML-F

A UML profile is a restricted set of UML markup along with new notations and semantics [FPR01]. Fontoura et al. presented the **UML-F** profile that provides UML stereotypes and tags for annotating UML diagrams to encode framework constraints. Methods and attributes in both framework and user code can be marked up with boxes (grey, white, half-and-half, and a diagonal slash) that indicate the method/attribute's participation in superclass-defined template patterns. A grey-box indicates newly defined or completely overridden superclass method. A white box indicates inherited and not redefined, a half-and-half indicates refined but call to `super()`, and a slashed box indicates an abstract superclass method. The `Fixed`, `Adapt-static`, and `Adapt-dyn` tags annotate the framework and constrain how users can subclass. `Template` and `Hook` tags annotate framework and user code to document template methods [Pre94]. Stereotypes for Pree's metapatterns (like unification and separation variants) are present, as are predefined tags for the GoF patterns. Recipes for framework use are present in a format very similar to that of design patterns but there is no explicit representation of the solution versus the framework. The recipe encodes a list of steps for programmers to perform.

FCL

The Framework Constraint Language (**FCL**) [HH01] applies the ideas from Richard Helms object oriented contracts [HHG90] to frameworks. Much like Riehle's role models [Rie00], FCLs specify the interface between the framework and the user code such that the specification describes all legal uses of the framework. The researchers raised the metaphor of FCL as framework-specific typing rules and validate their approach by applying it to Microsoft Foundation Classes, historically one of the most widely used frameworks. The language has a number of built-in predicates and logical operators and is designed to operate on the parse tree of the user's code.

FSMLs

The concept of Framework-Specific Modelling Languages (*FSMLs*) [AC06] was used to express models showing how framework-provided abstractions are used in framework-based application code. An FSML is a Domain-Specific Modelling Language [DSM] that is designed for a specific framework (its *base* framework). It supports automated round-trip engineering by mapping the abstract concepts of the framework into concrete completion code (*forward mapping*) and showing how to recognise an instance of a concept in the code (*reverse mapping*). They validated their approach using Eclipse Workbench API to demonstrate their agile round-trip engineering process, where the automated steps are executed on demand and not all together. In practice, the authors state that a single FSML will typically cover a small area of a framework's concern, thus multiple FSMLs will have to be provided to cover an entire framework, raising integration issues.

Again, a summary of the tools and techniques presented throughout this section can be found in the right side of Figure 3.1.

3.3 Trends

The framework understanding research topic still has room for expansion, and future work is needed to address existing open issues, besides those shared with the program comprehension field. Reuse problems must be better addressed by documentation or tool support if frameworks are to be widely adopted. There are still a few significant and stimulant trends:

- **Patterns and Pattern languages.** While developing pattern languages for framework documentation [AD11] and instantiation [BM03] [HK06], some issues have to be addressed such as identifying the expertise necessary to create effective pattern languages, how to identify the framework domain problems that should be the basis of patterns in the pattern language, how to best describe patterns, and what inter-pattern relationships should be included.

- **Widen context domain research.** There is a clear need to investigate the prevalence of framework understanding problems in industrial context frameworks [MRS02] [HK06]. Industry and academia have to join efforts to ascertain the impact framework learning problems have in large-scale software development environments, so that and adequate solution may be searched for.

- **Integrated environments.** With the advent of pluggable and extensible software development environments (like Eclipse), tools for assisting on framework understanding tend to be integrated into these self-sustainable platforms [MHK$^+$01] [NA05] [AC06], producing solutions that are multi-faceted and present different and varied approaches to accommodate different user needs. The combination and personalisation of these tools, offer flexibility to adjust the environments to the specific needs of particular users in particular tasks.

- **(un)Shared expertise.** A framework has specificities. The community that uses it (whether a small group or a large, distributed team) handles those specificities in a common contextualised fashion [Joh92] [GHJV95]. Sharing this knowledge and this expertise could prove useful, but the means and format by which it is communicated has to be effective. Not only the artifacts but the process need to convey and support the cognitive needs of the learners [FS05].

3.4 Summary

Using a framework is advantageous, but learning it might prove difficult. Framework understanding is program comprehension, nevertheless, it focuses more on framework specifics. Although the cognitive processes are the same, frameworks are built using several intermediate constructs (hooks, patterns, components) that, if looked for, may assist on a quicker and effective learning process. Usually, the knowledge for how to guide yourself through the framework lies in the documentation. But documentation needs to be updated and its contents need to be suitable for the intended audience. Existing tools try to produce better documentation, making an effort to capture those constructs and presenting them in a suitable format. But that might not be enough. Producing effective documentation takes time which is, often, scarce. Knowledge is lost, mainly because it is not properly captured and documented. But even if it is documented, how does the knowledge stand? Is it really effective? There is support for documentation production, but no support for the process of going through the documentation and learning from it. There is a “how to read the documentation”, but it is believed that a “how to learn from the documentation” will further improve framework understanding.

Chapter 4

Collaborative software development

Presently, software development relies prominently on processes that favor team work. Software projects usually involve a team or multiple teams that have to work together. These teams can be composed of a variety of domain-specific experts with different levels of experience and distinct socio-technical background. Communication is, therefore, paramount.

Back in the old days, traditional software development approaches grew from the fact that software development was, mostly, done by one-element teams (the "programmer") that resorted to a well-defined process in order to write code or automate a procedure. But, over time, software grew in size and complexity, and so did the team.

The switch from a process-centric activity to a more human-centric activity started in 1971, when Weinberg [Wei98] reengineered the software development process from a *"people empowering point of view"*. This theory gained momentum and points of view shifted. Process designers began to look at software development in different ways, and started to be concerned with the ways software developers work together. Research fields such as Groupware and Computer-Supported Collaborative Work (CSCW) rose to address collaboration supported by software. The Collaborative Software Engineering field deals with collaboration within software development. The next sections address these research areas in further detail.

4.1 Groupware

The term “groupware” dates back to 1978 when Peter and Trudy Johnson-Lenz defined it as [JLJL90]:

> *“intentional group processes plus software to support them.”*

This definition, however, was not widely accepted as it narrowed the scope of group work to a set of processes.

Another attempt to provide a definition came from Johansen [Joh88]:

> *“Groupware... a generic term for specialised computer aids that are designed for the use of collaborative work groups. Typically, these groups are small project-oriented teams that have important task and tight deadlines. Groupware can involve software, hardware, services, and/or group process support”.*

Again, this definition was non-consensual, as it would exclude categories of products that were not designed specifically for supporting work groups, like email or shared databases. Moreover, it focuses on small teams, which is restrictive.

To broaden the scope, Ellis et al. [EGR93] proposed to define groupware as:

> *“computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment.”*

Although less restrictive, this definition was considered too broad. Despite excluding multi-user systems (such as time-sharing systems where users don’t share the same goal), it would include shared database systems. Many argued that these systems cannot be considered groupware because they provide the illusion that every user has independent access, alas, they are not “group-aware.”

In general, as Grudin pointed out in [Gru94], groupware means different things to different people. According to Nunamaker et al. [NBM95], groupware is defined as

> *“any technology specifically used to make a group more productive.”*

Coleman stated [Col95],

> *“Groupware is an umbrella term for the technologies that support person-to-person collaboration; groupware can be anything from email to electronic meeting systems to workflow.”*

These definitions although quite broad captured almost all the products and projects that were identified as groupware.

All the above definitions share a common point: the notion of group work. Groupware is designed to support teams of people working together, focusing on software technology from human – computer to human – human interaction. Human interactions have three key elements: communication, collaboration and coordination. The goal of groupware is to assist groups in communicating, in collaborating and in coordinating their activities [EGR93], and it has been addressing these issues for years.

The fact that most early groupware tools failed to be widely adopted made clear the need for a better understanding of how groups of people work together [TN99]. A new research area emerged called: "Computer-Supported Collaborative Work".

4.2 Computer-supported collaborative work

Irene Greif (MIT) and Paul Cashman (DEC) coined the term CSCW, during a workshop in 1984 [Gre88]. Since then, this new field attracted a lot of interest. Amongst the various definitions, Wilson's seems to have captured the scope of CSCW [Wil91]:

> *"CSCW [is] a generic term, which combines the understanding of the way people work in groups with the enabling technologies of computer networking, and associated hardware, software, services and techniques."*

Greenberg [Gre91] added:

> *"CSCW is the scientific discipline that motivates and validates groupware design. It is the study and theory of how people work together, and how computer and related technologies affect group behaviour. CSCW collects researchers from a variety of specialisations – computer science, cognitive science, psychology, sociology, anthropology, ethnography, management, and information systems – each contributing a different perspective and methodology for acquiring knowledge of groups and for suggesting how the group's work could be supported."*

CSCW improved the understanding of groups and clarified that their relationships are not based only on communication, collaboration and co-ordination, as pointed out by Kling [Kli91]:

> *"In practice, many working relationships can be multivalent with and mix elements of co-operation, conflict, conviviality, competition, collaboration, commitment, caution, control, coercion, co-ordination and combat."*

In [Ack00], Ackerman described CSCW's main intellectual contribution as the effort to close the social-technical gap between what we know we must support socially and what we can support technically. He states that systems lack nuance, flexibility and ambiguity, clearly properties inherent to human activity. Therefore, the social aspects must be taken into account when designing systems for these to be increasingly effective.

CSCW researchers that design and build systems try to address core concepts in novel ways. These concepts have largely been derived through the analysis of systems designed by researchers in the CSCW community, or through studies of existing systems, where the most addressed are:

- **Awareness.** Individuals working together need to be able to gain some level of shared knowledge about each other's activities [DB92].
- **Articulation work.** Cooperating individuals must somehow be able to partition work into units, divide it amongst themselves and, after the work is performed, reintegrate it [SB92].
- **Appropriation (or tailorability).** How an individual or group adapts a technology to their own particular situation; the technology may appropriate in a manner completely unintended by the designers [Dou03].

However, the complexity of the domain makes it difficult to produce conclusive results. The success of CSCW systems depends heavily on the social context in which occurs, that it's quite hard to generalise. Consequently, newly designed CSCW systems, based on previous successes, may not be appropriate in other apparently similar contexts for a variety of reasons that are nearly impossible to identify *a priori* [Gru88].

In [RSVW94], Weber et al. contributed with a taxonomy that defines and describes criteria for identifying CSCW systems and serves as a basis for defining their requirements. The criteria are divided into three major groups:

- **Application.** From an application viewpoint, certain tasks are generically present in many scenarios, from general-purpose tasks such as brainstorming, note taking and shared agenda features, to more dedicated domains where there is the need for tailored tools. To the user, a CSCW system appears complete only when both specialised and generic tools are integrated.
- **Functional.** A CSCW system relates functional features with the social aspects of teamwork. Each functionality has an impact on the work behaviour and efficiency of the entire group using the system. Issues such as interaction, coordination, distribution, user-specific reactions, visualisation and data hiding must be taken into consideration. However, the psychological, social, and cultural processes active within groups of collaborators are the real keys to the acceptance and success of CSCW Systems.
- **Technical.** This criteria comprises hardware, software and network support. It divides the architecture of a CSCW system into four categories of classes or features:

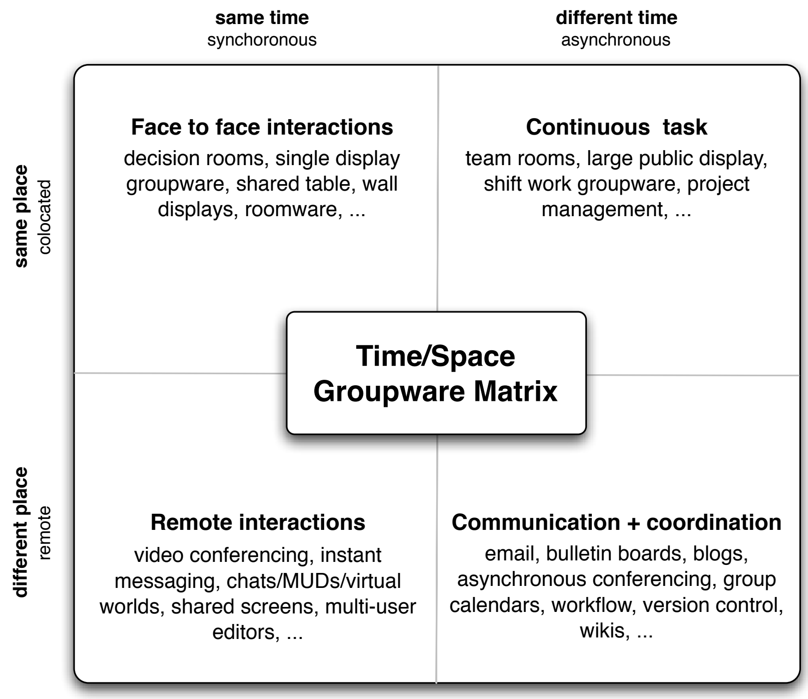

same time
synchoronous
different time
asynchronous
same place
colocated
different place
remote
Face to face interactions
decision rooms, single display groupware, shared table, wall displays, roomware, ...
Continuous task
team rooms, large public display, shift work groupware, project management, ...
Time/Space Groupware Matrix
Remote interactions
video conferencing, instant messaging, chats/MUDs/virtual worlds, shared screens, multi-user editors, ...
Communication + coordination
email, bulletin boards, blogs, asynchronous conferencing, group calendars, workflow, version control, wikis, ...

Figure 4.1: Groupware matrix (adopted from [Joh88] [ea95]).

(1) input, (2) output, (3) application, and (4) data. Each can be centralised or replicated.

For all of these groups, concerns such as flexibility, transparency, collaboration and sharing are addressed, and guidelines for supporting them are presented.

Another approach to conceptualising groupware and a CSCW system, states that its context can be considered along two dimensions: first, whether collaboration is co-located or geographically distributed, and second, whether individuals collaborate synchronously (same time) or asynchronously (not depending on others to be around at the same time). This approach can be seen in Figure 4.1 and was first introduced by Johansen [Joh88] in 1988, also appearing in [ea95].

The CSCW domains of application are quite a few, nevertheless, most of the research has converged to the design of collaborative tools that are build in order to assist software development.

4.3 Collaborative software engineering

Software engineering projects are inherently cooperative, requiring many software engineers to coordinate their efforts to produce a large software system. As such, this effort encompasses the development of a shared understanding surrounding multiple artifacts, each embodying its own model, over the entire development process. Figure 4.2 depicts that effective communication and awareness are crosscutting concerns across, not only the phases of software development but its models, process and infrastructure.

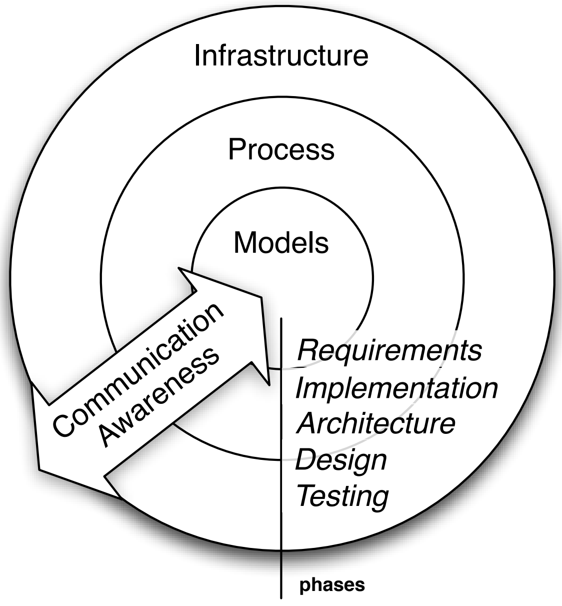


Figure 4.2: Collaborative software engineering aspects.

Collaboration techniques in software engineering have evolved to address our limitations: people are slow and error-prone, especially when working at high-levels of abstraction; our natural language is expressive but ambiguous; our memory skips the details of large projects and we can't keep track of what everyone is doing [End95].

4.3.1 Goals

In his seminal paper, Whitehead [Whi07] outlines the goals that collaborative software engineering pursues:

- **Establish project scope.** Engineers must work with the users and stakeholders of a software project to describe what it should do at both a high level, and at the level of detailed requirements. How this collaboration takes place can have profound impact on a project, ranging from the up-front negotiation of the waterfall model, to the iterative style of evolutionary prototyping [LB03].

- **Architectural and design convergence.** System architects and designers must negotiate, create alliances, and engage domain experts to ensure convergence on a single system architecture and design [Gri99].

- **Dependency management and reduction.** This encompasses a wide range of collaborative activities, including typical management of subdividing work into tasks, ordering them, monitoring, assessing, and controlling the plan of activities [MC94]. An important mechanism for managing dependencies is to reduce them where possible, thereby reducing the need for collaboration. Defining per-developer workspaces helps reducing dependencies in development time.

- **Error handling.** Errors and ambiguities exist in all software artifacts, and many approaches have been developed to find and record them. Collaborative techniques such as inspections, reviews, beta testing and bug tracking assist on mitigating these problems and tracking the quality of the software.

- **Record organizational memory.** In long running projects, people may come and go. Collaboration is, in part, recording what people know, so that project participants can learn this knowledge now, and in the future [Ack00]. SCM[1] change logs are one form of organisational memory in software projects, as are project repositories of documentation. Process models also record organisational memory, describing best practices for how to develop software.

4.3.2 Characteristics

Additionally, the same author [Whi07] states that software engineering collaboration can be characterised according to two aspects:

- **structured vs unstructured.** Collaboration in software engineering can be unstructured, where occasional and sporadic informal conversations occur concerning a piece of software anywhere in the project's lifecycle. It can also be structured, where the focus goes to various formal and semi-formal artifacts (requirement specifications, architecture diagrams, UML diagrams, source-code, bug reports, etc).

- **artifact-based vs model-based.** Software engineering collaboration can thus be understood as artifact-based (as stated in the previous aspect), or model-based collaboration, where the focus of activity is on the production of new models, the creation of shared meaning around the models, and elimination of error and ambiguity within the models. Without the structure and semantics provided by the model, it would be more difficult to recognise differences in understanding among collaborators.

[1] Software Configuration Management.

4.3.3 Tools

There is already a myriad of tools that support collaboration in software engineering. The following researchers proposed a categorization of these tools, upon a detailed survey:

Booch and Brown

Booch and Brown [BB03] defined the concept of *Collaborative Development Environment (CDE)* as: *"a virtual space where the stakeholders of a project - even separate by time and space - can meet, share, brainstorm, discuss, reason about, negotiate, record, and generally labor together to carry out some task, most often to create some useful artifact and its supporting objects"*. Examples of such systems are GENESIS and OPHELIA [BNR+03], and CHIME [DK99]. The authors surveyed a series of collaborative sites (what they call *Web-based CDEs*) categorised according to they application domain, e.g., non-software domains, asset management, information services, infrastructure, community and software development. They organised the CDE features in three groups: *coordination*, *collaboration* and *community building* and proposed a conceptual model for software development-specific CDEs composed of three layers: project workspace, team tools and development resources.

Sarma

Sarma [Sar05] presented an extensive survey on collaborative tools for Software Engineering, where she categorised tools from the perspective of user effort, i.e., time expended in setting up the tools, monitoring the tools and interpreting the information from the tools. She proposed a classification framework in the form of a pyramid of five vertical layers and three horizontal strands. The five layers in the pyramid are: (1) *functional*, (2) *defined*, (3) *proactive*, (4) *passive*, and (5) *seamless*. Tools that are at a higher layer in the pyramid provide more sophisticated automated support, thereby reducing the user effort in collaborating. Each level, thus, represents an improvement in the way a user is supported in their day-to-day collaborative activities. The three strands in the pyramid are: *communication*, *artifact management*, and *task management*. These three dimensions are critical needs crosscutting all aspects of collaboration. This categorisation pyramid can be seen in Figure 4.3.

Whitehead

Whitehead [Whi07] categorised collaborative tools into four groups:

- **Model-based collaboration tools.** Software engineering involves the creation of multiple artifacts, covering all the phases of development and ranging from

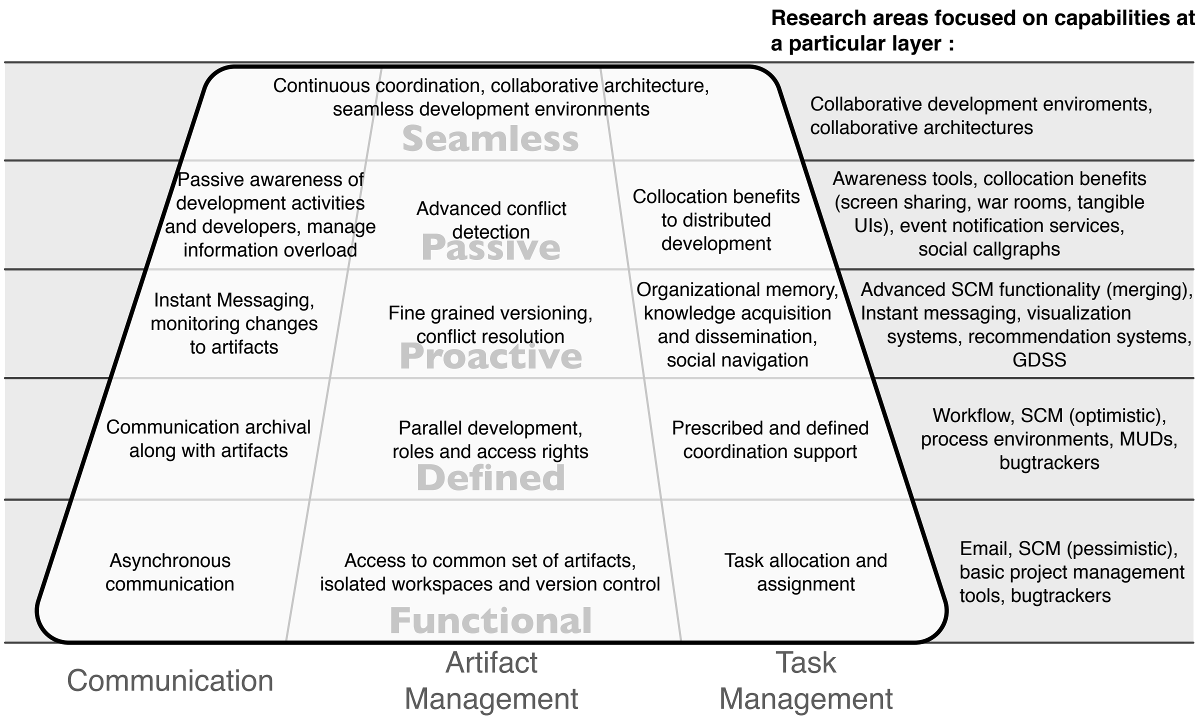

Research areas focused on capabilities at a particular layer :
Continuous coordination, collaborative architecture, seamless development environments
Seamless
Collaborative development enviroments, collaborative architectures
Passive awareness of development activities and developers, manage information overload
Advanced conflict detection
Passive
Collocation benefits to distributed development
Awareness tools, collocation benefits (screen sharing, war rooms, tangible UIs), event notification services, social callgraphs
Instant Messaging, monitoring changes to artifacts
Fine grained versioning, conflict resolution
Proactive
Organizational memory, knowledge acquisition and dissemination, social navigation
Advanced SCM functionality (merging), Instant messaging, visualization systems, recommendation systems, GDSS
Communication archival along with artifacts
Parallel development, roles and access rights
Defined
Prescribed and defined coordination support
Workflow, SCM (optimistic), process environments, MUDs, bugtrackers
Asynchronous communication
Access to common set of artifacts, isolated workspaces and version control
Functional
Task allocation and assignment
Email, SCM (pessimistic), basic project management tools, bugtrackers
Communication
Artifact Management
Task Management

Figure 4.3: User-effort collaborative tools categorisation pyramid (adapted from [Sar05]).

the end product to all the models, diagrams, specifications and source-code. Due to its specific semantics, creating these artifacts becomes a collaborative activity, supported by an already developed range of tools. These include collaborative requirements tools [BE98] [Cora], collaborative UML diagram creation [CLNC01] [Cre11], software configuration management systems and bug tracking systems [Bug] [Tra]. The focus on model-oriented collaboration embedded within a larger process is what distinguishes collaboration research in software engineering from broader collaboration research, which tends to address artifact-neutral coordination technologies and toolkits.

- **Process centred collaboration.** A software process model structures steps, roles and artifacts to create during software development. Process modelling and enactment systems have been created to help manage the entire lifecycle, supporting managers and developers in assignment of work, monitoring current progress, and improving processes [BT96] [LOJW98]. Typically, engineers manage the project as its goes, reacting when needed, thus reducing the initial coordination overhead. Overtime, experience dictates that coordination time still must decrease, through increasing predictability and defining points of collaboration. Process centred software development environments have facilities for writing software process models in a process modelling language, then executing these models in the context of the environment. Some examples of such systems are Arcadia [Kad92], Oz [BS94], Marvel [BSKH92], ConversationBuilder [KTC+92], and Endeavours [BT96]. In the commercial sphere, there are many examples of project management software, including Microsoft Project [Corc], Rational Method Composer [Core] and BaseCamp [Bas].

- **Collaboration awareness.** Software engineering is a human-driven and human-intensive activity where most medium- to large-scale projects involve multiple software developers that may or may not be co-located. In recent years, there has been much work in developing collaborative development environments [BB03] that provide support for coordination and communication during software development [HCRP04]. Seesoft [ESJ92], Palantir [SNvdH], Lighthouse [dSCdW+06] and Jazz [HCRP04] are but a few. A more extensive survey can be found at [Sar05]. A key issue in any collaborative tool is awareness, or *"knowing what is going on"* [End95]. More precisely, awareness is *"an understanding of the activities of others, which provides a context for [one's] own activity"* [DB92]. Awareness encompasses knowing who else is working on the project, what they are doing, which artifacts they are or were manipulating, and how their work may impact other work. In distributed collaborative work, maintaining awareness is considerably more difficult.

- **Collaboration infrastructure.** Various infrastructure technologies make it possible for engineers to work collaboratively. Tool integration, in the form of data integration (ensuring that tools can exchange data) and control integration (ensuring that tools are aware of activities of other tools and can take action based on that knowledge) make it possible for tools to coordinate their work. For example, nowadays, most IDEs know when a source-file is saved after editing and store it on a central repository (data integration) or SCM, then automatically call the proper compiler (control integration). Tools like Eclipse, Visual Studio, Marvel, DropBox [Dro11] and WebDAV [Dus03] already implement these behaviours, bringing a sustainable collaboration between engineers and theirs development tasks.

Teixeira

Teixeira [Tei09] surveyed existing collaboration tools according to three criteria: *Collaboration*, sub-divided into Awareness (team, context and resources), Communication (reaching peers) and Collective Knowledge (tagging, ranking, polls); *Integration*, application-oriented (plug-ins) or data-oriented (importing/exporting); and *Other characteristics*, e.g., licensing, supporting languages, etc. His analysis covered a wide range of tools from bug tracking, construction, design, engineering management and requirements elicitation. He then proposed a integrated web environment with a pre-selected set of features supporting collaboration between developers, as means to cover all the stages of software development.

4.4 Trends

As the research continues, both groupware and CSCW fields still face challenges. The current trends evolve mostly in the following directions:

- **Mobile technologies.** With the emergence of new mobile technologies and the increasing connectivity users enjoy, the importance of having light, easy-to-use and accessible groupware features is growing.
- **Web 2.0.** With the advent of concepts of the so-called second-generation web or "Web 2.0", collaboration and contextual-connectivity become even more present in our day-to-day activities. From blogs to wikis, social software is booming and its capabilities should be harnessed to improve group work.
- **Strong commercial interest.** Major commercial competitors such as Microsoft, Google, IBM, amongst others, are releasing solutions into the market at an increasing rate. This must come as an incentive to continue researching into these ever-increasing fields of interest.
- **Delocalisation of groups.** Teams and groups are becoming more and more delocalised. Work stops at one side of the planet and starts contiguously on the other side. Communication and synchronism become critical for a adequate and effective flow of work.

Specifically regarding collaboration in software engineering, research directions tend to address the following topics:

- **Web and desktop integration.** The migration of development tools to the web is increasing, now that the user interface is becoming more sophisticated (with technologies such as AJAX, Javascript and HTML 5.0) and the processing power of browsers is higher. UML and source code editing are no longer relegated only to desktop applications, whereas in the past, the web could not support such features. Despite this trend, there is a longstanding practice surrounding the use of integrated development environments (Visual Studio, Eclipse, Netbeans, etc.), which are not going to be displaced by completely web-based environments. Instead, future projects are likely to adopt a mixture of web-based and desktop tools, for which interfacing open standards between the desktop IDE's and the web-based services should be created. Although not an easy task, these open standards would allow a more seamless interaction with the complex information a software project creates.

- **Broader participation in the development process.** Contrary to current habits, software customers should be engaged during the entire development process, allowing a more actively assurance that their requirements are met. A participatory development model would allow customers a better tailoring of the software to their needs, balancing the source-code availability for refining by the customer. The trend towards providing support for distributed development teams in a wide range of development tools makes a broader engagement possible. Open source SCM tools like Subversion, as well as web-based requirements tools and problem tracking tools make it possible to coordinate globally distributed teams.

- **Capturing rational argumentation.** One of the strongest design criteria used in software engineering is design for change. This inherently involves making predictions about the future, having multiple engineers arguing over current facts and future potentials. Since only one vision of the system's model will prevail, the process of architecture and design is simultaneously cooperative and competitive. Providing collaborative tools to support engineers in the recording and visualisation of architecture and design argumentation structures would do a better job of capturing the nuances and tradeoffs involved in creating large systems. They would also better convey the assumptions that went into a particular decision, making it easier for succeeding engineers to know when they can safely change a system's design.

- **Using novel communication and presence technologies.** Software engineers tend to integrate new communication technologies into their development processes. Email, instant messaging and web-based applications are very commonly used in today's projects to coordinate work and be aware of whether other developers are currently active (present). Moreover, networked collaborative 3D game worlds are such an emerging technology that spawned "software immersion environments". Second Life [Sec] is an example of using such a 3D world to develop software, as their team uses its own platform to do so. There is a range of research issues inherent to the use of 3D virtual environments as a collaboration infrastructure, for example, how to synchronise physical and virtual worlds. It is still unclear if the benefits exceed the costs.

- **Improved assessment of collaboration technology.** Assessing the impact of the introduction of new technology into a project is difficult, and usually subjective. Without the uncovering of the pros and cons of specific collaboration tools (already introduced into the development process), forward progress in the field of software collaboration support tools is hard to measure. Developing improved methods for assessing the impact of collaboration tools would boost research in these areas by increasing confidence in positive results, and making it easier to convince teams to adopt new technologies.

4.5 Summary

Software development has always been about collaborating. Developing software has scarcely been an individual task. There has always been a team of developers for the software that really mattered. This team needed to work together and coordinate their efforts to develop the final product. Despite each developer having its own environment with some shared tools, the team would have to communicate physically and discuss most of the work outside the development environment, through meetings, casual conversations or other communication media.

Collaborative environments begin to support the social aspects of software development. With the advent of the Web 2.0 and the evolution of communication technologies, the software development teams are becoming more and more distributed. The development paradigm of the whole team in one physical space is fading with time. So far, collaborative development environments have been concerned with suitable presentation, seamless integration of tools and reliability of results. Only recently did the need for supporting the social aspects of software development became an issue. There was a need to bring the outside [the development environment] communication into the development environment, as a means to enable the needed collaboration between team members. Now, there is a real opportunity to harness knowledge that, otherwise, would be kept in people's heads and that would, usually, have a lifespan of only a few minutes during a conversation.

Learning can be collaborative, if you have the time. How many times does a developer asks for help from other team members? Well, the answer would be: it depends. But there is no doubt it would be more than once. When learning or understanding software, a developer often asks for help from his colleagues. Equally often, alas: his colleagues are not available; he forgets what he was told a few days back and asks again; he might address the wrong person to help him and so on. This becomes intrusive, especially if his colleagues don't have the time (which is common) to deal with his distress. He should be able to learn by himself, without bothering his colleagues too much. A collaborative software development environment should provide support for learning without too much intrusion and allow a satisfying degree of autonomy to learning developers.

Part II

Problem & Solution

Chapter 5

Research problem and solution

Program Comprehension deals with understanding programs and software artifacts. Framework Understanding focuses on a specific kind of software artifact: a framework. This understanding is often made resorting only to information on the artifact itself and accompanying documentation. More and more, software is developed collaboratively. Can this "collaboration" help in framework understanding?

In this chapter, several open research issues are raised focusing on framework understanding and the benefits collaboration can bring in improving the framework learning process. The underlying research questions and thesis statement are presented and explained, as well as the proposed solution approach. Finally, the research and validation strategies are debated as the baseline to pursue empirical studies and underline the need to design controlled experiments as repeatable packages for independent validation.

5.1 Open issues

From the state-of-the-art review presented in the previous chapters, a number of open research issues arise. An insight of the most relevant ones follows, intended to focus the scope of the work presented in this dissertation:

- **Frameworks are often hard to understand and use.** The difficulty in understanding frameworks is a serious inhibitor of effective framework reuse. This is mainly due to framework design being, frequently, very complex, and thus hard to communicate: (1) very abstract, to factor out commonality; (2) incomplete, requiring additional classes to create a working application; (3) more flexible than needed by the application at hand; (4) obscure, in the sense that it usually hides existing dependencies and interactions between classes [But98]. The learning curve becomes steep, requiring a considerable amount of effort to understand and learn how to use a framework (see Section 1.2, p. 4).

- **Good framework documentation is hard to produce and is often outdated.** Good documentation significantly improves the process of learning and understanding new frameworks. This documentation should be easy to use, support different audiences and provide multiple views through different types of documents and notations. The difficulty of producing contents for these requirements may hinder its applicability and demotes its importance within the development process. Most commonly during maintenance or evolution phases, documentation is used to assist on these tasks but its update is often discarded or neglected (see Section 1.2.1, p. 4).

- **Programmers (both experts and novices) recurrently tackle with understanding problems.** Every time a software developer needs to reuse a piece of code, whether it's a snippet, class, library or framework, she goes over the entire cognitive process of analysing, understanding and capturing the relevant information she needs. Depending on the purpose of the task at hand (learning, teaching, communicating, using), the format (quality, clarity, structure, abstraction level, etc.) of the code (seen on Section 2.2.1, p. 22), and the experience (expert or novice) of the programmer (seen on Section 2.2.4, p. 24), the understanding process may go through various approaches (top-down, bottom-up, etc., seen on Section 2.1, p. 18), not always leading to the desired outcome in a straight forward manner. Choosing the adequate understanding process should not be difficult, and changing from one to another should be feasible without too much overhead.

- **Different tools provide sparse results with variable quality.** By itself, each of these tools (seen on Section 2.3, p. 26) has its own problems and limitations, thus producing quality-questionable results. For instance, many of the problems design recovery (reverse engineering) tools have, tend to converge to selection of results (elimination of false positives) and semantic overlapping (same result can have several meanings) [Flo06]. With such discrepancy amongst results, it becomes difficult to ascertain tool efficiency and compare results regarding precision and recall.

- **The process of understanding a framework is not properly dealt with.** The palette of tools available (seen on Section 3.2, p. 43) to the framework learner scarcely deals with specific aspects of framework understanding. Without questioning its local and highly focused solutions, each tool aids in a specific aspect, whether capturing high-level design artifacts, browsing the code for hot-spots, or helping on producing sustainable output formats. Alas, the framework user has to navigate through a plethora of tools trying to figure out where the relevant information might be.

- **Collective knowledge of the development team is often not harnessed at its best.** Software development is a highly social process. It has been perceived that, when trying to understand a piece of code, developers turn first to the code itself and, when that fails, to their social network, that is, the team or community of developers [LVD03]. This behaviour, not only happens during code understanding, but also throughout the whole understanding process. Nevertheless, it is not easy to go for the team [NYY06]. Firstly, it is not clear who to address for clarification, for there is a lack of awareness of what other members of the team are doing or how do they relate to the work done. Secondly, the fields of expertise are not clear or stated, leading to wasteful interruptions of the wrong people. Thirdly and most often, the team or the experts are not available for consulting or rebuke their fellow colleagues due to interruption. Interrupted developers lose track of parts of their mental model, resulting in laborious reconstruction or bugs and discouraging more frequent interruptions (See Section 1.2.3, p. 7).

- **Intrinsic developers' knowledge is not captured and shared as effectively as it could be.** Developers go to great lengths to create and maintain rich mental models of design and code that are rarely permanently recorded. Very often, developers, without referencing written material, can talk in detail about their product's architecture, how the architecture is implemented, who owns what parts, the history of the code, to-dos, wish-lists, and meta-information about the code. For the most part this knowledge is never written down, except in transient forms such as sketches on a whiteboard. The bottom-line problem here is that "Lots of [useful] information is kept in peoples' heads" [LVD03]. Without capturing, storing and sharing this information, it eventually decays and becomes useless.

5.2 Research questions

From the aforementioned open research issues, a few research questions revolve around a major question that is considered central to the presented research work: *How to improve framework understanding?*. Those questions are listed next.

- What are the actual goals of the framework learner? Where does she start? What does she look for?
- Are there typical and repeated behaviours, that learners apply when trying to learn how to use a framework?
- How can fellow learners help each other without too much effort?
- How can collaboration help in improving framework understanding?
- How can tools assist and support the learning process?
- What is missing from existing development environments to assist on framework understanding?

5.3 Research focus

The research work presented in this dissertation covers subjects from all of the fields and topics described in the earlier chapters (Chapters 2, 3 and 4). For clarity on the domain areas and focusing of the research, Figure 5.1 depicts the main target where the results of the work are expected to bring the most contribution.

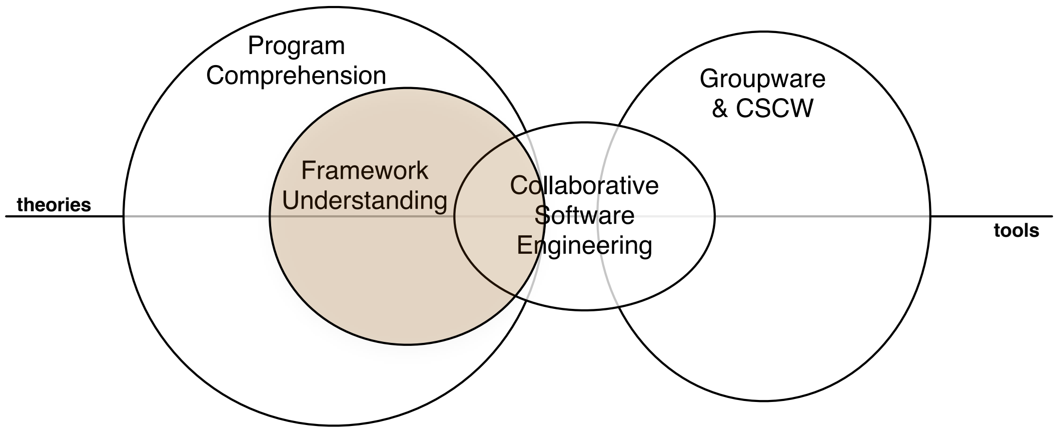


Figure 5.1: Research domains where the presented work focuses. The coloured, filled circle represents the area the work most contributes to.

5.4 Thesis statement

Based on the research challenges presented (sections 5.1 and 5.2) and the state-of-the-art review (Chapters 2, 3 and 4), the author states that:

> *"Providing a collaborative environment that supports the best practices of framework understanding will allow framework learners, both experts and novices, to produce and share relevant knowledge, thus improving their learning process and their effectiveness on framework reuse."*

This statement uses terms whose meaning may not be consensual, and therefore lead to questions that deserve further discussion:

- **What is meant by "*best practices*"?**

  To put is shortly: proven good recurrent solutions to recurrent problems, also known as, *patterns*. These can be identified by studying and observing the process of understanding a framework. A proper way to communicate best practices to others is using patterns [AIS77]. These best practices are presented and further detailed in Chapter 6 (p. 81).

- **How does one "*support the best practices*"?**

  Typically, framework learners and users aren't explicitly aware of these best practices, specially novices. "Support" means promoting its use, without much effort. The collaborative environment enables this in two ways: (i) Presenting them in a proper format (*patterns*), easy to understand and apply[1] and (ii) providing a way to collaboratively allow learners to guide each other through successfully taken paths to a solution (Chapter 7, p. 109).

- **Who are the *novice* and the *expert* framework learners?**

  Any developer who wishes to use (select, instantiate, evolve, etc.) the framework and has little or no knowledge of the framework is considered a novice learner. Through framework usage, gradually, this novice learner will become more proficient with the framework and will attain the expert status. When, exactly, this status is achieved is fuzzy. It is always a relative measurement, where someone is an expert when compared to someone else. Therefore, and for the purpose of this dissertation, what differentiates a novice from an expert is the usage gap of the framework, that is, the time difference each one had, dealing[2] with the framework. For validation purposes,

[1] applying here doesn't mean automatically or prescriptively. The way patterns are applied depends on the learner's interpretation of the pattern's descriptive guidelines.
[2] assumes weekly iterations where valuable deliverables are implemented using the framework.

it is assumed a minimum gap of 3 months (see section 8.1.1, p. 148, for further details).

- **What is meant by "*produce and share relevant knowledge*" ?**

  Instead of "relevant knowledge", one should read "relevant learning knowledge". The purpose is not to produce knowledge on *how to use the framework*[3], but on *how we learn how to use the framework*. What steps did the learner take to build her mental model of the solution to her specific problem? Which artifacts did the learner interact with and how did she get there? This (intrinsic) knowledge is, usually, not captured and lost. The collaborative environment provides a tool to enable this capturing and sharing of this knowledge in a non-intrusive way, that is, without preempting the learner's working context.

- **What is meant by "*learning process*" and how does one measures "*improvement*"?**

  This covers the whole cognitive process of searching for a satisfying answer to a problem or question the learner has about the framework. As seen in the literature review (Chapters 2 and 3), this process can go through several strategies (section 2.1, p. 18) conditioned by several aspects (cognition characteristics and needs, seen on sections 2.2.1, p. 22 and 2.2.4, p. 24). The metrics for improvement can be both objective (time, cognitive load, i.e., amount of acquired knowledge) or subjective (personal satisfaction, tool usage).

- **How is *effectiveness* measured?**

  Effectively (re-)using a framework can be measured by verifying if the purpose (by which the framework was used) was satisfied. The intention is to prove that, not only the reuse tasks led to satisfying deliverables in terms of requirements, but also in a shorter time, when compared to undertaking the same tasks without the proposed collaborative environment.

The original thesis statement can be decomposed in the following hypothesis, as a more objective means to validate the author's assumptions:

- **H1:** Providing novice framework learners with the best practices of framework understanding, not only reduces the time they need to reuse the framework, but increases their knowledge of the framework.

[3] Although the collaborative environment in itself allows for this to be recorded and shared, if the learner so wishes.

- **H2:** Providing novice framework learners with collaborative tools to guide them through the process of framework understanding, not only improves the time they take to reuse the framework, but increases their knowledge of the framework.

- **H3:** Providing expert framework learners with best practices, supported by collaborative tools to guide them through the process of framework understanding, increases their knowledge intake of the framework, without penalising their time effectiveness.

5.5 Research goals

This dissertation aims at contributing to the body of knowledge in software engineering. Concretely, it strives to improve framework learning for both novice and expert developers, by enabling them to share their expertise and allowing the intrinsic learning knowledge to be captured and harnessed in a non-intrusive way. This will be achieved in two ways:

1. **By guiding learners on following best practices for framework understanding**. Learning how to use a framework is a recurrent task in a software developer's life. Providing them with a proven set of best practices (patterns) will shorten their learning curve and improve their learning outcome.

2. **By laying grounds for the development of tools to support framework understanding through a collaborative environment**. Devising a collaborative environment suitable for framework learning, that supports the learning process. Also, providing a subset of tools that aid in the capture, storage, sharing, ranking, presentation, and recommendation of acquired learning knowledge.

5.6 Proposed approach

For pursuit of the defined research goals, a solution approach was devised. As such, and according to the author,

> *The (framework) learner is usually engaged in a (learning) **process** composed of a series of activities. This process has best practices (**patterns**) that can be followed to improve its outcome. These practices could be actively applied and improved having **tools** to support them.*

This statement provides the common grounds for the solution proposed by this dissertation, which are depicted in Figure 5.2 and further detailed next.

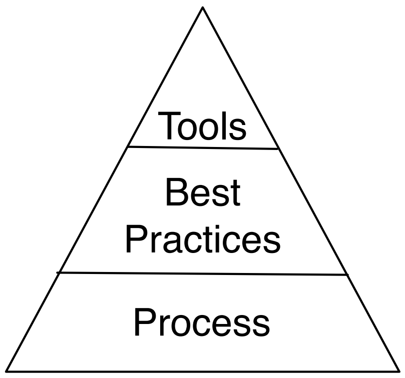


Figure 5.2: Common grounds for the proposed approach.

5.6.1 Learning process

In a broad sense, while learning about a framework, a developer's activities may fall into three interaction categories:

- **Code**. That, ideally, should suffice to answer all our questions. Too bad it takes too long to do that. What we need to know is not explicitly in front of us. Furthermore, frameworks make it particularly difficult to find what we need to know. As an example, recovering design knowledge implicitly present in the code is a recurring practice to help clarify the framework's structure and purpose. The issues dwell on what kind of design artifacts, to what kind of audience, and how to store and present them, so that they become useful.

- **Documentation**. When the developer wants to learn how to use a framework (or any reusable software artifact, for that matter), she goes for the documentation, if it exists. But, is there always documentation? And is that documentation clear, well-suited and complete? Does it have all the answers? There are known ways of producing good documentation for frameworks [But98][BKM00][Agu03]. The issue is nurturing the developers to easily produce and access that documentation, even during the learning process.

- **Social network**. When all else fails, the developer loses her self-sufficiency as a learner and resorts to her contacts, meaning, strong candidates to bear knowledge that might help her. Call it team, peers, social network, buddies or any other term, there is knowledge that one can't find anywhere else but on people's minds. It is called *intrinsic or tacit knowledge.* Getting this knowledge should not be intrusive, in the sense that it should not disrupt the normal working activities and habits. There should be ways of harnessing this knowledge without such disruption.

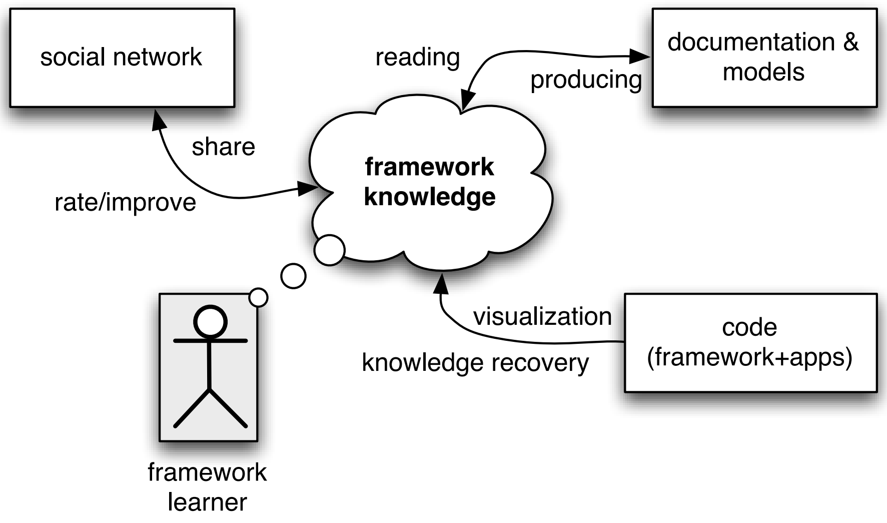


Figure 5.3: Framework learning activities and actors.

In short (see Figure 5.3), a framework learner looks at the code, reads the documentation, visualises information and asks her colleagues for help, as going through a learning process of understanding how to use the framework.

5.6.2 Improving the learning process - patterns and tools

As a support for the activities taken during the learning process, the author proposes to help the learner in two ways: (1) Providing a "guide" or "map" of the best way to undertake those activities and (2) allowing the learner to tap into the knowledge of the learning community through the use of appropriate tools. Both strategies are integrated into a collaborative, shared data-driven environment where the learner can perform her learning activities (Figure 5.3), supported by these enhancements.

Patterns

To provide the so-called "guide" or "map", there is, associated with the learning process, a series of good practices on how to deal with each stage of the learning process that the author captured into ***patterns***. These patterns are further detailed in Chapter 6 (p. 81).

(Collaborative) Tools

Depending on several factors (learner's experience, existing artifacts, learning goal, etc.) the learning process to undertake may resort to different practices and paths. What works for some, might not work for others, and may even vary between frameworks. Novices and experts will take different paths.

Yet, in a truly collaborative environment, where, at first, there is no distinction between who is expert and who is novice, sharing experiences and advising the global community proves useful [Sur04]. The importance given to an advice or counsel is measured by its actual applicability. You become experienced and expert by giving valid and helpful feedback into the community.

By supporting this sharing of knowledge, the learners may benefit from their collective intelligence, thus improving their own learning processes. Therefore, the supporting tools should be prepared to capture this learning knowledge (detailed in section 7.3, p. 117), share it and assist other learners in their tasks.

In practice, the author intends to develop a (small) set of ***tools*** integrated in an existing collaborative environment, that will support the capture and sharing of the learning process and will enable the rating and recommendation of directions to take when trying to reach similar learning goals. This overall supporting process is depicted on Figure 5.4.

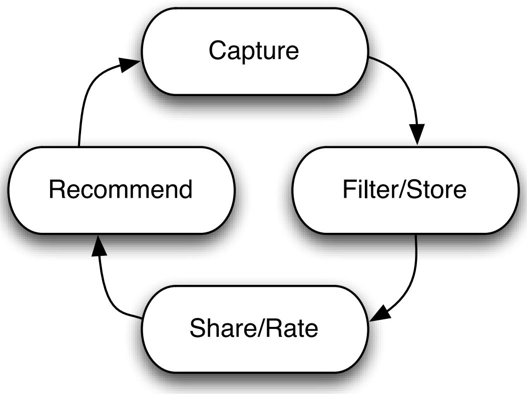


Figure 5.4: Supporting steps to improve the learning process.

The purpose will be to capture the learning steps (based on best practices) taken by the learner. Whether she looks at the code first, goes for documentation, explores certain artifacts, and recovers others, until she reaches a satisfying conclusion.

This path taken is then recorded, stored and shared. Sharing means that other learners may reuse it or get assistance through it to guide them on their own learning path. If the shared knowledge really helped them, then they should rate it accordingly. As the collected knowledge keeps improving (through sharing, usage and rating), the best learning strategies will be recommended to recurrent learners and, hopefully, improving their learning process. This process and how it is supported by the tools is further detailed in Chapter 7 (p. 109).

5.7 Research strategy

In order to pursue a scientific validation of the aforementioned thesis, it is necessary to adequately define the experimental protocols which assess these claims in a rigorous and sound way.

Understanding the way software engineers build and maintain complex and evolving software systems, requires researchers to focus beyond the tools and methodologies; they need to delve into their social surroundings and cognitive processes, which encompass individuals, teams, and organisations. In this sense, research in software engineering is regarded as inherently coupled with human activity, where the value of generated knowledge is directly linked with the methods by which it was obtained.

Because the application of reductionism to assess the practice of software engineering, particularly in field research, is very complex (if not unsuitable), the author claims that the presented research is to be aligned with a pragmatistic view of truth, valuing acquired practical knowledge. Consequently, the author chose to use whatever methods seemed more appropriate to prove — or at least improve our knowledge about — the questions here raised.

As such, in the author's understanding, the most suitable way of validating the thesis would be to rely on empirical studies and controlled (quasi-)experiments to provide evidence that sustain the validity of the hypothesis here stated. Discussion on guidelines for performing and reporting empirical studies have been recently approached by the works of Shull *et al.* [SSS07] and Kitchenham *et al.* [KAKB$^+$08]. The typical tasks and deliverables of a common experimental software engineering process can be found in [GA07].

Formally, a systematic scientific approach based on this assumption, requires the use of mixed methods. As such, *observational* and *historical* methods were used to gather knowledge that supported the set of patterns contribution (Chapter 6, p. 81) and a controlled replicated experiment (Chapter 8, p. 147) was conducted to complement it and to provide evidence to support the remaining contributions (Chapter 7, p. 109).

Due to the effort required, and the operational difficulties of conducting such experiments in the field of software engineering, it was decided to conduct an experiment in an academic controlled setting. This experiment studies intermediate-experienced developers in understanding a framework, collecting time and knowledge acquisition metrics and comparing results from different set-up learning environments. The detailed experiment protocol and results can be seen in Chapter 8 (p. 147).

The independent experimental validation of claims is not as common in Software Engineering as in other, more matured, sciences. Hence, the author stresses the need to build reusable experimental packages that support the validation of each claim by independent groups. Therefore, the (quasi-)experiment was designed as an *experimental package*, to be performed in different locations, and lead by different researchers, in order to enhance the ability to integrate the results obtained and allow further meta-analysis on them.

5.8 Summary

Despite advances in framework understanding, there are still issues and challenges in this domain of research, viz. (i) framework documentation is often incomplete, (ii) learners keep tackling with understanding a framework without supporting guidance and (iii) collaboration in learning is not properly supported. In what concerns the current research, two main goals were identified: (i) guiding learners on following best practices for framework understanding, providing them with a set of instructive patterns and (ii) devising a collaborative learning process and its respective supporting tools, all integrated in a collaborative environment. Aligned to a pragmatist view of truth, valuing acquired practical knowledge, the author proposes to validate the proposed goals through the usage of mixed methods, amongst which are (i) *observational* and *historical* methods for the contribution regarding the set of patterns and (ii) controlled (quasi-)experiments, performed in academic contexts, for the remaining contributions.

Chapter 6

Patterns for understanding frameworks

This chapter focuses on the first objective of this dissertation, namely **to improve framework learning by guiding learners on following the best practices of framework understanding.** This is achieved by providing learners with a set of patterns that communicates these best practices. Its goal is to help users become aware of the problems that they will typically face when starting to learn and understand frameworks. These patterns are targeted for framework learners, especially novices. The patterns were mined from existing literature, lessons learned, and expertise on using frameworks, based on previous studies and literature reviews.

6.1 Why patterns?

The concepts of pattern and pattern language were introduced in the software community by the influence of Christopher Alexander's work, an architect who wrote extensively on patterns found in the architecture of houses, buildings and communities [AIS77][Ale79][Lea94].

Patterns help to abstract the design process and to reduce the complexity of software because they specify abstractions at a higher level than single classes and objects. This higher-level is usually referred to as the *pattern level*. They represent useful mental building blocks for dealing with specific problems of software development.

The opening of the third volume of the book on PATTERN LANGUAGES OF PROGRAM DESIGN [MRB97] starts with the following sentence: *"What's new here is that there's nothing new here"*. This single assertion characterises the epistemological nature of patterns, in what concerns its methodology and goals; patterns result from the observation, analysis and formalisation of empirical knowledge in search for stronger invariants, allowing rational choices and uncovering newer abstractions. A pattern should not report on surface properties but rather *capture hidden structure* at a *suitably general level*. A comprehensive discussion on the epistemology of patterns and pattern languages can be found in a recent work by Kohls and Panke [KP09], where the authors state that

> *The argument that there is "nothing new" in a pattern must be rejected; otherwise there would be nothing new to physics either, since physical objects and the laws of physics have been around before[...].*

Coplien states in his book SOFTWARE PATTERNS [Cop96] that

> *[...]the most important patterns capture important structures, practices, and techniques that are key competencies in a given field, but which are not yet widely known.*

Although applied to software frameworks, the patterns presented in this chapter don't address software design *per-se* but the process of understanding software, prior to the respective design activities. They deal with the cognitive-oriented procedures that build a mental model of the framework thus allowing its *usage*, that is, going into the specific details of the framework and effectively use it. As such, the patterns' instructive nature provide a suitable format to communicate the empirical knowledge they bear.

6.2 Pattern form

The patterns community have been experimenting with several structures of the pattern description. There is the original structure that has been defined by Alexander *et al.* in their book A PATTERN LANGUAGE: TOWNS, BUILDINGS, CONSTRUCTION [AIS77], and is commonly known as the *Alexanderian Pattern Form* (APF). Then, there is the seminal work of Gamma *et al.* [GHJV95] where a different format was used specifically tailored for the area of software design, commonly known as the *Gang of Four* format (GOF). Both have benefits and liabilities: the APF is implicitly structured, and results in a fluid, narrative-like text, persuading the reader to identify herself with the pattern; the GOF form poses a more methodological partitioning with several explicit subsections.

Analysing both approaches, the presented patterns adopt the form: *Name-Context-Problem-Solution-Consequences* [AIS77], with a few additions. A brief description of each section follows:

1. **Name.** The name of the pattern should be adequate to transmit the metaphor behind the solution.

2. **Context.** An introductory paragraph that describes the scenario in which the problem recurrently occurs.

3. **Problem.** This section starts by describing the empirical background of the pattern, and the range of different ways the problem can be manifested. It ends with an *emphasised* statement of the essence of the problem.

4. **Forces.** This section describes the set of forces, that is, aspects that should be weighted and balanced in order to achieve a good solution.

5. **Solution.** This section starts with an *emphasised* headline, which describes the concrete actions necessary to solve the stated problem. It then elaborates on the solution, describing the steps it takes to implement it and, if applicable, admissible variations.

6. **Consequences.** Applying a pattern generates a resulting context, where the resolution of the forces now poses benefits and liabilities.

7. **Rationale.** This section provides an explanation of the basis and fundamental reasons behind the solution.

8. **See also.** A pattern is a pattern because there is empirical evidence for its validity. This section gives reading directions to further cases where the pattern can be observed.

6.3 Patterns overview

Before going into the details of each pattern, here is a brief overview of the pattern set with each pattern's intent, and a map (Figure 6.1) showing their relationships.

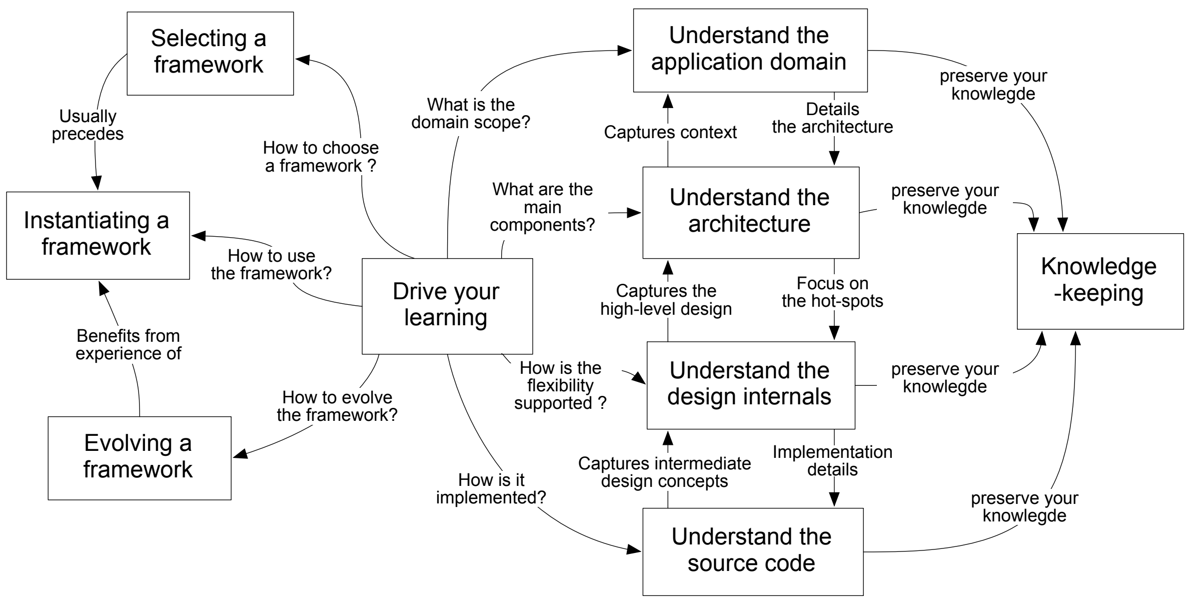

Selecting a framework
Usually precedes
Instantiating a framework
How to choose a framework ?
How to use the framework?
Benefits from experience of
Evolving a framework
How to evolve the framework?
Drive your learning
What is the domain scope?
Understand the application domain
Captures context
Details the architecture
preserve your knowlegde
What are the main components?
Understand the architecture
preserve your knowlegde
Knowledge -keeping
Captures the high-level design
Focus on the hot-spots
How is the flexibility supported ?
Understand the design internals
preserve your knowlegde
Captures intermediate design concepts
Implementation details
How is it implemented?
Understand the source code
preserve your knowlegde

Figure 6.1: Framework understanding patterns and their relationships.

- Selecting a framework. This pattern allows deciding whether or not to select a framework, after evaluating its appropriateness for an intended application domain.

- Instantiating a framework. This pattern shows how to learn about instantiating a framework in order to implement an application.

- Evolving a framework. This pattern shows which steps should be taken to learn how to evolve a framework.

- Drive your learning. This pattern shows how to plan your learning process throughout the task of understanding a framework.

- Knowledge-keeping. This pattern shows how to preserve the acquired knowledge about a framework.

- Understand the application domain. This pattern guides the learner in how to know what is the application domain covered by the framework.

- Understand the architecture. This pattern shows the learner how to find architectural knowledge about the framework.

- UNDERSTAND THE DESIGN INTERNALS. This pattern tells the learner how to look for knowledge about the design internals of the framework.

- UNDERSTAND THE SOURCE CODE. This pattern helps on identifying, in the source code, where are the important parts that enable the developer to implement the application.

When referring to a related pattern within this set, its name will appear in SMALLCAPS, otherwise it will appear in *SMALLCAPS ITALICISED* font if it's an "outside" pattern, together with the proper reference. The reader should take notice that these patterns are intended to have an autonomous existence outside the contents of this dissertation as part of an independent patterns catalogue. As such, despite references to other sections of this document being present in the pattern contents, it should be assumed that these are not present in the aforementioned catalogue, thus the referred sections being, somewhat, re-explained and re-written in the pattern.

6.4 Problems addressed

The problems addressed by the patterns are basically raised by the following questions:

- What do I need to understand about the framework to accomplish my task? What kind of knowledge do I need? More concrete or abstract? At code level, design level, documentation level?

- How can I acquire the knowledge I need? Which learning strategy should I adopt? Which one is better for my specific needs?

- Which kind(s) of tools can I use to gather, organise, explore and preserve the knowledge I value most?

According to [But98], framework reuse can be divided into categories according to the re-user's interests, whether a framework selector, an application developer, a framework maintainer, or a developer of other frameworks. These categories range from selecting, instantiating, flexing, composing, evolving and mining a framework. For the scope of the patterns presented in this chapter, only the most commonly used will be addressed: *selecting*, *instantiating* and *evolving*.

6.5 Related patterns

The presented set of patterns has a close relationship with another pattern set, namely PATTERNS FOR EFFECTIVELY DOCUMENTING FRAMEWORKS [AD11]. These patterns aim at helping developers becoming aware of the typical problems they face when documenting object-oriented frameworks. They describe a path commonly followed when documenting a framework, although its strict following from start to end is not mandatory in order to achieve effective results.

In fact, many frameworks are not documented as extensively as suggested by the patterns, due to different kinds of usage (whether its a white-box or black-box framework [FSJ99]) and different balancing of tradeoffs between cost, quality, detail, and complexity. One of the goals of the patterns is precisely to expose such tradeoffs, and to provide practical guidelines on how to balance them to find the best combination of documents to the specific context at hand.

According to the nature of the problems addressed, the patterns are organised in ***process*** patterns related with the process of cost-effectively documenting frameworks (*how to do it? which activities, roles and tools are needed?*) and ***artifact*** patterns (*which kind of documents to produce? what should they include? how to relate them?*). Artifact patterns address problems related with the documentation itself, here seen as an autonomous and tangible product, independent of the process used to create it. They provide guidance on choosing the kind of documents to produce, how to relate them, and what to include. The patterns presented in this chapter relates closely to the artifact patterns, referring them often throughout its description, and whose general overview can be seen in Figure 6.2.

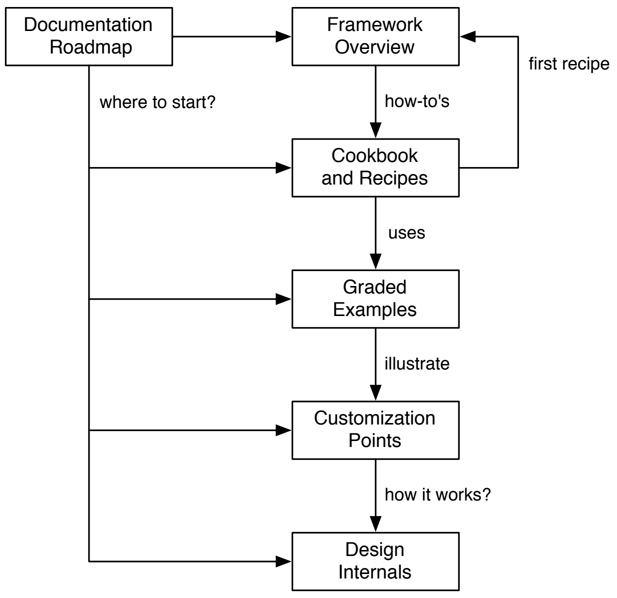


Figure 6.2: Patterns for effectively documenting frameworks (Adapted from [AD11]).

Together with Roberts, Johnson provides a pattern language that deals with "EVOLVING FRAMEWORKS" [RJ97] and how to develop a framework. This pattern language elaborates on the common path a framework takes to become viable, i.e., suitable for developing applications. This pattern language is related to the Evolving a Framework pattern § 6.8 (p. 91) as it deeply explores the issues around framework evolution.

6.6 Pattern Selecting a Framework

You are someone (manager, project leader, developer) who is responsible for finding a solution for an application development project in a certain domain. You are about to select a framework that can help you to solve your problem.

Problem

Framework selection consists of deciding whether or not to reuse a framework, while evaluating its appropriateness for an intended application in a specific domain.

What do you need to learn about a framework in order to select it effectively?

Forces

- **Effort**. You don't want to spend too much time learning what you need to know to effectively decide if a framework is selectable.
- **Certainty/Sureness**. You need to be sure that the framework you're about to select covers, not only your application domain, but also all of your specific needs.
- **Documentation**. The existing documentation may not give the necessary insight into the applicability of the framework.
- **Complexity**. The more complex a framework is, the harder it is to understand.

Solution

Start by quickly understanding the framework under consideration. Look for a short description of the framework's purpose, the domain covered, and an explanation of its most important features, preferably illustrated with examples.

In order to ascertain if a specific framework covers your domain requirements, you need to Understand the Application Domain in a clear way, i.e., the domain covered by the framework, and the range of solutions for which the framework was designed and is applicable.

However, knowing the purpose of the framework is not always enough to ensure that this framework may meet all the problems. It can be important to go deeper to Understand the Architecture, Understand the Design Internals, or Understand the Source Code, until being sure of the framework's appropriateness for the problem at hand.

To be more effective, you may want to Drive Your Learning according to your experience and specific requirements.

Consequences

- **Cost-effectiveness**. You quickly gain insight into the scope of the framework and its coverage of your specific needs. Going into detail gives you more accurate hints on how the framework is built and addresses your problems.
- **Narrow knowledge**. Yes, it solves your specific problems, but that doesn't give you a whole grasp over what other specific problems it might address. Further investigation might be needed when new contextual-related problems arise.

Rationale

When using frameworks, one of the key decisions that need to be made is whether or not the framework fits the application. Since frameworks can be complex, gaining a deep understanding of the framework (in order to make that decision) often requires the time-consuming process of, actually, using the framework. Capturing information about the applicable domain of the framework proves easier to decide [FHLS00]. Limitations and design trade-offs about the framework can help to show for what the framework can and can't be used. There will always be a degree of uncertainty, but that can be mitigated by existing documentation. Moreover, the potential user will often perform experiments to increase his understanding of the framework and to evaluate its appropriateness to the new application requirements.

See also

In [TW07], Andrew Turner and Chao Wang had to evaluate a set of existing AJAX frameworks to select the most suited for their requirements. Their process relied on ascertaining that all the frameworks could cover their specific domain and high-level requirements. They had to dig deeper into the framework internals and even develop some prototypes to test if the framework could address and solve their specific issues.

In [APP04], Ahamed et al. proposed and applied a criteria for ascertaining the suitability of a framework to a specific project. It relied on a set of areas to inspect, starting with the intended domain and evolving into detailed issues like the presence of design patterns and lower-level concerns such as error handling and degree of coupling.

They then applied their criteria to characterise an existing framework for a transaction processing system implementation called jPOS ISO 8583, to see if it was suitable for selection.

6.7 Pattern Instantiating a Framework

You have been given, or previously selected, a framework to build a solution for a specific problem. You are now about to instantiate the framework in order to implement the intended functionalities and build your application.

Problem

Framework instantiation usually consists on deducing, designing and implementing application-specific extensions to the framework. Despite knowing which extensions the framework requires, it is hard to understand where to "plug" those extensions in the framework.

What do you need to learn about a framework in order to instantiate it quickly?

Forces

- **Documentation.** Tutorial documentation can help you to walkthrough the initial contact with the framework and to acquire knowledge about the framework's entry points.
- **Effort.** You don't want to spend too much time learning what you need to know to instantiate the framework.
- **Learner's experience.** If you are already acquainted with the framework, you try to find similar areas of flexibility where to customise the framework. A novice learner will look for representative examples that might give her a hint of where to start poking the code for those flexibility areas.
- **Complexity.** Complexity may not mean "difficult to use", but surely means "difficult to learn". Issues like indirection, abstraction and obscurity give the framework its power but also hinder its ability to be learnt and understood.

Solution

Find the areas of the framework that can be adapted for reuse by looking at the existing documentation and instantiation examples to clarify how to use those areas.

Look at the documentation and find the *Customisation Points* [AD06b] where framework instantiation is supported. In addition, look also into some *Graded Examples* [AD06a] that explain how to use the framework to implement more common functionalities. The customisation of a framework is usually possible through sub-classing of framework abstract classes and/or composition of concrete classes. Understanding how these classes relate and interoperate is crucial to be able to use them properly.

If you're dealing with a white-box framework, it is important to further Understand the Architecture and to Understand the Design Internals. Only then can you start to Understand the Source Code and effectively start reusing the framework.

To be more effective, you may want to Drive Your Learning according to your experience and specific requirements.

Consequences

- **Framework know-how.** You gain knowledge on how to instantiate the framework, progressively increasing your expertise and being able to incrementally build your application.

- **Blind trust.** Using a framework means trusting in code you have never seen. So if the framework is poorly built or has features that it publicizes but are not implemented or don't work well, your solution may suffer with it. It's not uncommon to see frameworks whose internal code is not available for debugging or modification, therefore you can't correct or improve the framework's internal code.

Rationale

Framework instantiation into domain-specific application takes place at points of predefined refinement called hotspots [Pre94]. Thus knowing where they are, when and how to use these points leads to an effective framework instantiation. Moreover, one of the best ways to start learning a framework is by example [FSJ99], specially for novices. Most frameworks come with a set of examples that you can study, and those that don't come with examples are pretty hard to learn. Examples are concrete, thus easier to understand than the framework as a whole. Frameworks are easier to learn if they have good documentation.

See also

In [FHLS97], Froelich et al. § 3.2.2 (p. 45) resort to a Hooks-model to describe the framework customisation points and use it to instantiate the SEAF (Size Engineering Application Framework). Their approach is similar to this as it relies on documentation describing the customisation points (hooks) and uses it to know where to instantiate the framework.

In [FPR01], Fontoura et al. § 3.2.4 (p. 48) presents the UML-F profile that provides UML stereotypes and tags for annotating UML diagrams to encode framework constraints. Amongst other tags, Template and Hook tags annotate framework and user code to document template methods. Stereotypes for Pree's metapatterns § 3.2.2 (p. 45) are present (like unification and separation variants), as are predefined tags for the GoF patterns. Recipes for framework use are present in a format very similar to that of design patterns but there is no explicit representation of the solution versus the framework. The recipe encodes a list of steps for programmer to perform.

6.8 Pattern Evolving a Framework

You are a software engineer who is responsible for the maintenance and evolution of a framework. Your task may be to evolve the framework to support new requirements, to refactor its design, or to correct errors, while preserving its backward compatibility.

Problem

To evolve a framework means understanding where the evolution will take place within the framework and to which extent do you need to go in learning about it. You need to know what elements to evolve and its impact on the framework as a whole.

What do you need to learn to evolve a framework?

Forces

- **Documentation.** The documentation is almost always descriptive, which is not good for framework evolvers, because original framework designers can't predict how the framework might be extended in the future through additional flexibility on existing hotspots, or in additional hotspots.
- **Maintenance expertise.** It is expected that the framework maintainers are both domain experts and software design experts.
- **Evolution task.** Your task may be adding new functionalities or improving existing ones, correcting errors or refactoring the design. Different information needs arise according to the task at hand.
- **Tools.** There might be the need to recover lost design information that is important to the evolution task. Existing reverse engineering tools may prove useful.

Solution

Look at the architecture of the framework and understand how it is built and how it meets its purpose. Gain further insight of its components by looking at the design internals and areas of flexibility and treat each variability issue separately.

Have a good Understanding of The Architecture and its rationale, in order to avoid the architectural drift problem [CHSV97], commonly consequential of poor framework evolution. Understanding The Design Internals and Understanding The Application Domain helps at keeping the evolution process in perspective. Look at the *Customisation Points* [AD06b] that support the flexibility offered by the framework and plan you evolution tasks.

To be more effective, you may want to Drive Your Learning according to your experience and specific requirements.

Consequences

- **Evolution expertise**. You gain enough insight to adequately address your evolution tasks. Be alert to issues regarding delta analysis, architectural drifts, version proliferation and over-featuring [CHSV97].

- **Ignorant surgery**. Evolving parts of the framework means understanding its interaction with its other parts. Sometimes, focusing too much on the problem at hand may cause what is called "ignorant surgery" [RCM04]. Inadequate investigation prior to performing a change task limits the understanding of the existing design of a system. The evolver performs a change in a single location in the code that is better understood, but which may lead to unforeseen effects throughout the framework as its dependencies aren't properly identified and taken into account.

Rationale

The need to evolve a framework usually arises during any of the following situations: (1) new domain concepts need to be incorporated into the framework, (2) reducing the complexity of the framework through re-design and (3) initial design issues that were neglected need to be addressed [CHSV97]. The evolution process usually involves the execution of two tasks: restructure (refactoring) and extension. In order to restructure it properly, the developer must be aware of all the repercussions and dependencies of the components or customisation areas she intends to extend or alter. Another concern is application compatibility. Backward compatibility testing should be performed, so that the framework can remain compatible with earlier developed applications. A faulty evolution process may change the way the framework is supposed to be used, closing otherwise opened customisation points. By understanding how the framework is supposed to be used will enable the developer to maintain its interface coherent, without too much effort.

See also

As referred in section 6.5 (p. 85), Roberts and Johnson [RJ97] present a pattern language for evolving frameworks where they show that there is need for the understanding of different levels of detail concerning the framework components.

In [CFL06], Cortés et al. present a tool to support framework evolution tasks, namely refactoring and extension. They propose to automate certain kinds of refactoring tasks and applying extension rules based on Pree's meta-patterns § 3.2.2 (p. 45), which implement variation points as a combination of template and hook methods.

6.9 Pattern Drive Your Learning

You are about to learn a framework in order to reuse it. You have your understanding goals, but no process of learning to guide you through.

Problem

Upon defining your learning goal, you need to start learning. Knowing what to learn is as important as reaching those goals through an effective learning process. Adopting a learning strategy is, therefore, essential. But what strategy is more suitable?

How do you define the most effective process for your learning needs?

Forces

- **Top-down vs. bottom-up.** A top-down approach will start at a higher-level progressing downwards, giving a good overview with little effort but poor details. A bottom-up approach starts at a low-level progressing upwards, giving good detail with little effort, but hindering awareness of the global impact of changes.
- **Learner's experience.** Your experience with the framework can affect you learning strategy, when choosing where to start and how to proceed.
- **Learning style.** You may be a more "global", "reflective" learner or you may possess a more "sequential", "active" learning behaviour [FS05].

- **Documentation.** Depending on the existing documentation artifacts, the learner will have to adapt his learning strategy to better fill in her knowledge gaps. For example, if high-level artifacts are well documented but the lower-level ones lack the necessary detail, the learner will have to dig out those details by herself.

Solution

Start learning the framework at an abstraction level (*entry point*) you feel comfortable with. Progressively, converge towards the understanding level that gives you increasing knowledge intake, navigating up or down the abstraction levels whenever needed.

An entry point is selected according to your experience level and learning style. A more experienced developer tends to adopt a more top-down approach (start at the top, understand the high level concepts), whereas a novice developer will go for a more bottom-up approach (start at the bottom, understand the low level concepts)[SMV06][KRW05][SLB00]. Remember you can start at any abstraction level.

Regarding style, a "global", "reflective" [FS88] learner will start at a higher level of abstraction (domain or architecture) and will "top-down" gradually into the framework, because she needs the big picture first. A "sequential", more "active" learner will start at a lower level (usually poking at the source code), try things out and "bottom-up" into the framework, gathering bits and pieces to form her mental model. Then, change directions, that is, swap strategies, as needed. This is beneficial to reduce cognitive overload and focus on the goal.

Look at the *Documentation Roadmap* [AD05a] and choose the documentation artifacts that may better assist you on your understanding tasks, namely *Framework Overview*, *Graded Examples*, *Customisation Points*, *Design Internals* and *Cookbook & Recipes* [AD05a][AD06a][AD06b].

Consequences

- **Methodical approach.** A methodical investigation proves more effective than a chaotic one [SLB00][Hou08]. By defining a course of action the chances of reaching an answer faster, increase.

- **Personalised cognitive process.** Navigate freely along the abstraction levels until you feel satisfied with the things you've learned. Your mental model will progressively increase throughout task execution.

Rationale

As seen on section 2.1 (p. 18), many researchers have studied how programmers understand programs through observation and experimentation [Sto05][SFM97]. This research has resulted in the development of several cognitive theories to describe the comprehension process. These range from a bottom-up § 2.1.3 (p. 19) or top-down § 2.1.2 (p. 19) strategies with opportunistic or systematic § 2.1.4 (p. 20) behaviours converging into an integrated model that combines all of these § 2.1.5 (p. 20). This integrated model would serve a wider range of learners, as it would give the learner the option of choosing the most effective learning strategy. All of these cognitive models use existing knowledge together with the code and documentation to create a mental representation of the program.

See also

In [SLB00], Schull et al. § 3.1 (p. 39) perform a study about reading techniques while learning about a framework and divide them into two categories: hierarchy (of framework design components)-based and example-based. While experienced learners mostly use the former, the latter gains the preference of the most novice learners. Nevertheless, one important conclusion of the study is that the learning process should not be strict and allow the learner to freely choose the way she feels more comfortable with, thus potentially achieving the better results faster.

In [RCM04], an exploratory study was performed on how developers investigate source-code in order to perform a change task. One of the major results of that study was that a methodical investigation of the code of a system was more effective than an opportunistic approach. Nevertheless, this theory does not imply that a purely systematic approach to program investigation is the most effective. Successful subjects also exhibited some opportunistic behaviour.

6.10 Pattern Knowledge-Keeping

You want to keep what you have learned while understanding the framework. You want to be able to use that knowledge in the future so that you don't have to do it all over again. Also, you want it to be fit for other framework users.

Problem

Learning how to use a framework means finding, browsing, using and building understanding knowledge. Reusing the knowledge in future learning tasks is as useful as reusing design and code. Developers go to great lengths to create and maintain rich mental models of code that are rarely permanently recorded [LVD03]. Preserving this knowledge for later use is, therefore, of utter importance.

How to adequately preserve the acquired learning knowledge?

Forces

- **Existing Documentation.** Adopting existing documentation artifacts as templates to harbour new knowledge depends on its availability, easiness of use and quality of its contents.

- **Intrinsic knowledge.** Much relevant information is kept in the minds of experts that have used the framework. This knowledge decays with time and never becomes useful to others but the expert himself. Sharing this knowledge is important, but might be expensive to experts as it causes interruption and can be time-consuming.

- **Tools.** Documentation generation tools, using recovery and extraction techniques, might be used to generate several specific kinds of views and formats over the information about the framework.

- **Motivation.** Producing documentation can be tiresome and boring. The long-term cost-benefit is often overlooked, thus affecting the motivation to spend time and resources producing documentation.

- **Maintenance.** Documentation quickly becomes outdated. Therefore, it should be easy to maintain, or else it will rapidly loose its value.

Solution

Use documentation methodologies and tools to produce documentation artifacts and store them in an open, shared, collaborative environment where the information can be accessed and evolved over time.

Choose the documentation artifacts that most adequately register the knowledge you've acquired, namely *Framework Overview*, *Graded Examples*, *Customization Points*, *Design Internals* and *Cookbook & Recipes* [AD05a][AD06a][AD06b].

Consequences

- **Shared knowledge base.** The learning knowledge is shared through the community of learners, from experts to novices, all being able to use and improve it according to their needs.

- **Collaborative effort.** By opening the knowledge to the community, its quality improves from the constant revision and maintenance by a heterogeneous group of learners.

Rationale

Good quality documentation is crucial for the effective reuse of object-oriented frameworks. Without a clear, complete and precise documentation describing how to use the framework, how it is designed, and how it works, the framework will be particularly hard to understand and nearly impossible to use by software engineers not initially involved in its design.

Documenting a framework is not trivial § 1.2.1 (p. 4). Producing framework documentation needs to address several issues ranging from contents consistency to contents organisation. Using framework documentation also poses a problem where issues like understandability, searchability, and effectiveness need to be adequately addressed [Agu03].

Adopting known documentation artifacts [AD05a][AD06a][AD06b], specific to our learning task, to store our understanding knowledge helps to lessen the burden of recording our findings. If that knowledge is then shared with a community of other fellow users, that burden can be even less as the other contributors also share the responsibility of keeping the information up-to-date.

The "community" factor also contributes to the refining and quality increase of the documentation as factors like diversity, independence, decentralisation and aggregation [Sur04] will mitigate quality issues like accommodating different audiences, having different views over the information or even the lack of standards.

See also

In [Agu03], a minimalist approach to framework documentation is proposed. It presents an extensible documentation infrastructure based on the WikiWikiWeb concept and XML technology. It provides several document templates and a simple cooperative web-based environment to produce and use minimalist framework documentation. The proposed approach covers the overall documentation process, from the creation and integration of contents till the publishing and presentation. It encompasses a documentation model, a process and a set of supporting tools.

In [WBWW90], Wirfs-Brock et al. introduce CRC cards as a simple design alternative that can be quickly applied to any object-oriented project. The short learning curve makes this responsibility driven design approach a natural choice for small projects. CRC cards are also highly effective as a front end to other design methods. The same authors created CRC card models of high-level framework abstractions (as an overview) seconded by illustrative sequence diagrams of the collaborations and relationships between the framework components.

6.11 Pattern Understand the Application Domain

You have a framework you want to use, but you don't know its general purpose or if it covers your application domain.

Problem

You need to be sure that the framework answers your functional and domain requirements. Not only the general purpose of the framework must be clear but also its reach and the assurance that it covers, if not all, the required problem domain areas and constraints of the application to develop.

> *How do you learn what is the purpose of the framework and the domain scope it covers?*

Forces

- **Learner's domain knowledge.** The easiness of finding where the domain concepts are present, and which areas relate to those domains, strongly depend on the learner's knowledge about the application domain. Metaphor and technical jargon may be useful to track down and identify hints on component names that might relate to domain concepts.
- **Expert domain knowledge availability.** If an expert on the application domain is available for consult, it should speed up domain knowledge acquisition and promote a domain-driven analysis of the framework.
- **Documentation.** The documentation should give ideas on how the domain is mapped onto the framework. It could contain a brief description of the framework and its main purpose and concepts.

Solution

Identify the general purpose of the framework and its application domain by browsing the existing documentation and capture the main domain concepts, how they relate, and how the framework addresses them.

A *Framework Overview* [AD05a] is a good way to do so and *Graded Examples* [AD06a] provide detail on how the main features can be implemented.

Find the framework top components (abstractions) and their metaphor (names and designations) and UNDERSTAND THE ARCHITECTURE of how they are related to cover the domain concepts.

Preserve all the information gathered, adopting a KNOWLEDGE-KEEPING strategy.

Consequences

- **Broadness.** Viewing the framework at this level enables the learner to know the general purpose of the framework and its overall domain applicability.
- **Shallowness.** Without going into more detail it is sometimes difficult, if not impossible, to ascertain if a certain functionality or technology is covered by the framework. As such, one needs to dig deeper and try to UNDERSTAND THE DESIGN INTERNALS in order to understand how some pieces fit in together, because the system requirements need detailed specifications of certain functionalities.

Rationale

When you know nothing about a framework, usually you try to see what the framework is for. You look for the title, a paragraph, maybe the name of the components. These elements are usually on the documentation that accompanies the framework, whether is a specific document, website or other kind. When trying to find out its purpose, you look for keywords or something that will shed some light about the domain concepts of the framework. Is it about graphics? Is it about networks? Is it general-purpose? What are the concepts it encompasses and how? Only after you've acquired this information you start looking for other details.

See also

In [APP04], the process of determining a framework's suitability to a problem domain starts with the domain analysis activity. This activity has several non-contiguous steps to reach a domain model, where existing documentation (when this documentation is not available for the framework itself, they resort to examining documentation belonging to existing applications developed using that framework) is reviewed and domain experts are consulted. Also existing standards for the domain are studied. The result of the activity is a domain analysis model containing the requirements of the domain, the domain concepts and the relationships between concepts.

6.12 Pattern Understand the Architecture

You are using a framework and you want to know if its architecture is compatible with your application needs. You want to understand how the framework elements are structured and how they relate.

Problem

Using or evolving a framework impacts the framework as a whole. The awareness of the full implications of any change to the framework requires a sound notion of the framework's architecture and how its elements, which map the domain concepts, relate with one another. You need to understand its architecture.

How do you learn about the framework's architecture, its components and internal relationships?

Forces

- **Framework maturity.** A mature framework is likely to be better structured, being easier to identify its main architectural elements.
- **Documentation**. If there exists documentation that explains the overall architecture, it can be a great understanding aid.
- **Tools**. These can complement the lack of overview documentation, by reverse-engineering the architectural information.

Solution

Look into the documentation or any existing reverse-engineered design information and search for instances of architectural patterns [BMR+96]. Usually present in a more mature framework, these can indicate its main architectural style.

Browse through the *Design Internals* [AD06b] to identify the main architectural concepts and its relationships. If you need more detail, look for architectural primitives § 3.2.2 (p. 46). These can give an incremental view of the overall architecture by identifying interfacing ports between framework components and later, by aggregation, lead to defining a known architectural pattern or structure.

Preserve all the information gathered, adopting a Knowledge-Keeping strategy.

Consequences

- **High-level awareness.** There is an awareness of all the framework internal components and how they relate. You can piece together all the framework's parts to see if it fits your application needs.

- **Shallowness.** Despite being comprehensive, there is no grasp of how the components that relate to each other, interoperate, or how they function internally. You need to further Understand the Design Internals, to be able to know more about their behaviour.

Rationale

A framework is an architectural abstraction. An architectural abstraction identifies and names a composition of elements with a certain structure and functionality. This facilitates communication about designs. A framework provides a set of abstractions that are useful when discussing and describing a domain [JN99]. When a white-box framework is used, it is necessary to understand the concepts and architectural style of the framework in order to develop applications that conform to the framework. Many errors can be avoided and the application can be constructed more efficiently if the framework user understands its strategies and styles. In a mature framework, during its design, a suitable architectural style was adopted and usually these are known domain-specific architectural patterns.

See also

In [SG96], Shaw and Garlan, first introduce the notion of software architectural styles as a family of systems in terms of a pattern or structural organisation. More specifically, it determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined.

In [BMR+96], Buschmann et al. presents a pattern catalogue of architectural styles based on the work of Shaw and Garlan and introduces a software design classification system consisting of architectural patterns, design patterns and idioms, covering different perspectives and different abstraction levels.

In [ZA05], Zdun and Avgeriou § 3.2.2 (p. 46) propose to remedy the problem of modeling architectural patterns through identifying and representing a number of "architectural primitives" that can act as the participants in the solution that patterns convey. According to the authors, these "primitives" are the fundamental modelling elements in representing a pattern and also they are the smallest units that make sense at the architectural level of abstraction (e.g., specialised components, connectors, ports, interfaces). Their approach relies on the assumption that architectural patterns contain a number of architectural primitives that are recurring participants in several other patterns.

6.13 Pattern Understand the Design Internals

You want to use the framework to solve a specific problem. You need to know how the framework can be used to implement a specific solution to that problem.

Problem

To effectively use a framework, its flexibility and reuse points reside mostly at an intermediate design level. Due to its complexity, it is not clear where those points are and how they are used to implement the solution. Therefore, understanding the design internals is essential to find those flexibility points.

How do you understand the design internals of a framework?

Forces

- **Design complexity.** A framework design is, by nature, complex. Most of its complexity can be found at a level of design where, usually, design patterns are used.
- **Inheritance vs. composition.** Design variations for the same problem may prove to be a hindrance because they obfuscate the identification of existing solutions as they seem dissimilar.
- **Documentation.** Depending on the existing documentation, one may find bits and pieces of information about how design solutions were implemented to solve specific domain problems.
- **Tools.** Reverse engineering and software visualisation tools that aid in identifying known design structures and patterns may save time and give a different view over the whole or part of the framework's design.

Solution

Go through the Design Internals [AD06b] documentation, or browse the existing classes, and identify the concept classes and their interactions.

Look for instances of known design patterns. Design patterns [GHJV95] are often used as building blocks for frameworks, because they introduce the flexibility it needs. The more mature a framework is, the more design patterns will it encompass. Design patterns aggregate these "hot spots" or *Customisation Points* [AD06b]: areas of flexibility we can "hook" [FHLS97] into and take full advantage of the framework's reusability.

Preserve all the information gathered, adopting a Knowledge-Keeping strategy.

Consequences

- **Framework internal mechanisms.** You gain knowledge about how the framework provides the flexibility for adapting its semi-implementation to develop an application. You acquired most of the information to adapt the framework to your needs.
- **Still at a design level.** Understanding is kept at a design level. Adapting means implementing, and implementing means coding. You need therefore to Understand the Source Code.

Rationale

Generally, template methods are used to implement the frozen spots of a framework, and hook methods are used to implement the hot spots. The frozen spots are aspects that are invariant along several applications in a domain, possibly representing abstract behaviour, generic flow of control, or common object relationships. The hot spots of a framework are aspects of a domain that vary among applications and thus must be kept flexible and customisable.

The difficulty of good framework design resides exactly on the identification of the appropriate hot spots that provide the best level of flexibility required by framework users. More hot spots means more flexibility, but results in a framework more difficult to design and use, so somewhere in between resides a balanced design.

Frameworks are designed and implemented to fully exploit the use of dynamically bound methods. Template and hook methods [Pre94][GHJV95] are two kinds of methods extensively used in the implementation of frameworks, conferring its flexibility and adaptability.

In [Pre94], several ways of composing template and hook classes are identified, and presented under the form of a set of patterns, globally called meta-patterns. Meta-patterns § 3.2.2 (p. 45) categorise and describe the essential constructs of a framework, on a meta-level. Design patterns provide proven solutions to recurrent design problems and are extremely useful to design object-oriented frameworks. The motivation for using meta-patterns is to provide a means to categorise and describe design patterns on a meta-level, and to support framework construction. Therefore, design patterns become the building blocks of frameworks.

Design patterns can be used as inspiration when looking for flexible hot-spots within a framework. A framework that contains design patterns can be understood in terms of these; therefore when adapting a framework, users can perceive the specific adaptation steps (sub-classing or configuring framework classes) as adaptations of small wholes – the involved design patterns – instead of making new atoms (classes). Users see their adaptations in a perspective larger than that of a single class [JN99].

See also

In [BSM06], Bruch et al. § 3.2.2 (p. 44) propose the use of data mining techniques to extract reuse patterns from existing framework instantiations. Based on these patterns, suggestions about other relevant parts of the framework are presented to novice users in a context-dependent manner.

In [FGS06], Fairbanks et al. present a pattern language based on the notion of design fragment § 3.2.2 (p. 46). A design fragment is a pattern that encodes a conventional solution to how a programmer interacts with a framework to accomplish a certain goal. It provides the programmer with a "smart flashlight" to help her understand the framework, illuminating only those parts of the framework she needs to understand for the task at hand. Design fragments give programmers an immediate benefit through tool-based conformance and long-term benefit through expression of design intent.

6.14 Pattern Understand the Source Code

You are to code your solution using the framework.

Problem

To actually use a framework you have to code. Therefore, understanding the framework's source code is mandatory. But a framework is not a common piece of code: it has no clear entry point and there isn't a "main" method from where to start understanding the flow of control. Its hotspots are scattered across the code and the way to use them may not be straightforward.

What to look for to understand the source-code and plug-in your solution?

Forces

- **Hollywood principle.** Your code will have to be inserted at a specific location that the framework will eventually call and execute. It might not be straightforward how, where and when, that calling will take place.

- **Language familiarity.** If you are not familiarised with the programming language in which the framework is built, you're going to take more time to understand the code.
- **Task-orientation.** To be cost-effective, learners tend to focus on the task at hand, and to find the quickest way to solve their immediate problem.
- **Code Annotations.** Code annotations and inline documentation can give helpful insight on a certain code fragment was implemented or served a purpose.
- **Documentation.** Usually, the framework comes with examples on how to quick start or how to quickly address initial problems. These can be extremely helpful as they show how to begin and force you to try to understand how the system works.

Solution

Browse the documentation for examples on how to address the task at hand. Identify where in the code you will have to add your own, suitable for your specific task.

Usually, the framework comes with a *Cookbook & Recipes* [AD05a] on how to solve common problems the framework addresses. These show you how to begin coding and enable you to understand how the overall system works.

If no documentation is present, try to look for beacons and idioms[1] § 2.1.1 (p. 18) that might hint to the execution starting point(s) of the framework (Control classes and "main" methods, where the flow of control may start) and track down the flow of control. Idioms are coding patterns that are used to solve recurrent problems (You use a loop to iterate over an array, etc.). Beacons are fragments of code that may resemble algorithm techniques or coding strategies to known problems. Classifying and chunking code into these concepts might prove useful to increase code granularity search.

Identify your insertion or extension points as you go along and preserve all the information gathered, by adopting a Knowledge-Keeping strategy.

Consequences

- **Missing the whole picture.** At such a low-level, the learner has local expertise but might overlook more global side effects of code insertion or modification. A broader notion of what is happening might be necessary, so you might need to Understand the Design Internals to gain further awareness.

[1] Also called programming plans.

Rationale

Prior to performing a software modification task, developers must inevitably investigate the code of the target system, in order to find and understand the code related to the change. With frameworks, this theory applies. If we assume that the way a developer investigates a program influences the success of the modification task, then ensuring that developers effectively investigate the code of the system can yield important benefits. This leads a decrease in the cost of performing software changes and an increase in the quality of the change.

In general, developers should: (1) follow a general plan when investigating a program, (2) perform focused searches in the context of this plan, and (3) keep some form of record of their findings.

Documentation here is crucial. Not only should there be some sort of guide for browsing the code, but also examples of how to address the most common problems. Going through these examples would be a valuable assisted first “dive” into the framework code and would help emerge the control-flow mechanism of the framework and the way it is supposed to be used.

See also

In [SMV06], Sillito et al. performed a study where they observed developers trying to understand a system in order to perform a change task. They harvested and identified 44 different kinds of questions developers ask during that process and divided them into four categories. These categories were based on how deep into the source code (graph) the developer had to go to acquire the information needed for answering a given question: (1) finding the initial focus points, (2) building on those points, (3) understanding the sub-graph of dependencies over those points, (4) and dependencies over such sub-graphs.

In [RCM04], Robillard et al. conducted another study where they observed successful and unsuccessful developers while performing a software evolution task. They came up with a theory of program investigation effectiveness in the form of a series of observations and associated hypotheses. Overall, they found that successful developers exhibited a highly methodical approach to program investigation, where they identified the high-level structures and planned the changes to be made, without forcefully spending more time than a more opportunistic approach.

6.15 Summary

In this chapter, a set of patterns for understanding frameworks was presented. These enclose the solutions to the main problems that occur when learning about a framework. The format by which the patterns are presented instruct the learner on how to cope with the main difficulties and how to proceed according to her specific needs and constraints. Every pattern builds on top of empirical knowledge captured through observation, experience and existing literature as a means to communicate the invariant aspects of the solutions in this domain of study. These patterns have already been presented and evaluated by the patterns community and their feedback incorporated [FA08].

Chapter 7

Collaborative learning with DRIVER

This chapter focuses on the second objective of this dissertation, which is **to improve framework learning by laying grounds for the development of collaborative tools to support framework understanding.** This is achieved by providing learners with a collaborative environment that supports the learning process, named DRIVER. This setting includes a (sub-)set of tools that enables capture, storage, sharing, rating and recommendation of learning knowledge, namely *learning paths*. This toolset is built upon a *wiki* that provides documentation artifacts about the framework and which configuration allows knowledge acquisition in several ways. The extensible nature of the wiki presents the tools as a set of *plug-ins*, enabling extensibility and further additions, laying grounds for future improvements of the collaborative environment. This chapter starts by presenting the DRIVER platform, the theoretical grounds behind its conception, progressing to an overview of its main features and concluding with a comparison with other related tools.

7.1 What is DRIVER?

DRIVER is a platform that enables framework users to effectively learn how to use a framework in a collaborative, user-friendly, knowledge-intensive environment. It promotes social learning within a community of framework users, with different levels of experience, motivated with finding answers to their problems and sharing them for the benefit of all. Its architecture relies on the notion of Collective Knowledge System § 7.2.3 (p. 115), supporting knowledge quality evolution through social interaction. Its features include:

- **Collective knowledge management.** The learning knowledge is captured and maintained by the community in a non-intrusive way. Learners can search and rate available knowledge and get recommendations on the best course of action (see section 7.4.2, p. 125).
- **Best practices support.** The patterns presented in Chapter 6 are present and can be consulted for guidance.
- **Collaborative documentation.** The framework documentation artifacts are available for editing and updating by the community of learners (see section 7.4.1, p. 123).
- **Social Classification.** Tagging and folksonomies are at the basis of the learning knowledge classification (see section 7.3.4, p. 121).
- **Extensibility.** The platform is open for extension to accommodate new features that might appear in the future.

These features cover a set of requirements § 7.3.5 (p. 122) that derived from further research on *collective intelligence* and *collaborative learning*. The following sections describe, in some extent[1], how this research progressed until reaching the devised solution.

7.2 Improving knowledge collaboratively

Software development is a knowledge-intensive activity [P.N99]. Developers become learners (thus, knowledge acquisitors) when they need to locate potentially relevant source code and understand how to modify it to solve the task at hand.

Software development is also a social activity [NK02]. The activity is carried out by a group of developers, forming a community and engaging in collective creative knowledge work [NOY00]. It is a social activity mediated through artifacts, which are, primarily, source code and documents. Although sharing knowledge and information within a community of developers being indispensable, the primary means for developers to obtain knowledge is not through communicating with their peers, but through artifacts. Developers invest great effort recovering implicit knowledge by exploring code and documents. If this fails,

[1] Although suitable for Chapter 4, the author felt that these sections would be more relevant here, to keep the reader focused

they turn to their social network [LVD03]. But how can the community provide what the artifacts couldn't? And if it can, how does it translates to useful, relevant knowledge to solve the task at hand? Can we trust others, just because we have no other choice?

This section tries to answer these questions by progressively focusing the reader into the issues of collaboratively (in a community) acquiring the knowledge the developer needs and ultimately reaching the proposed collaborative solution.

7.2.1 Going for the crowd

In his book, WISDOM OF THE CROWDS: WHY THE MANY ARE SMARTER THAN THE FEW AND HOW COLLECTIVE WISDOM SHAPES BUSINESS, ECONOMIES, SOCIETIES AND NATIONS [Sur04], James Surowiecky presents an extensive analysis on how knowledge and reasoning in a group of people provide better results, on average, than an informed, expert individual. He hypothesises that people should act collectively to make decisions and to solve problems in matters of general interest (to the community). He states that, despite unawareness of it, *"we are collectively smart"* and intellectual superior to the isolated individual. When dealing with groups of people, there are concerns, not only regarding size and uniformity, but cognition, coordination and cooperation of the individuals. Nevertheless, *"groups do not need to be dominated by exceptionally intelligent people in order to be smart"*, performing better at deciding between possible solutions than coming up with them.

He then presents the four pillars that sustain the *wisdom of the crowd*: **diversity**, **independence**, **decentralisation** and **aggregation**. These are further detailed next.

Diversity

The best collective decisions come from disagreement. In a diverse group, each person should have some private information, even it's just an eccentric interpretation of the known facts, to add perspective that would otherwise be absent. Diversity proves easier for individuals to say what they really think, consequently generating lots of *losers* (alternatives). Large collectives have the inherent ability to recognise these *losers* quickly and *kill them off*.

Independence

It may come as a paradox, but each member of the group should act as independently as possible. They should be free from the influence of others, as people's opinions should not be determined by the opinions of those around them. This generates new, unfamiliar, ungeneralised data and keeps mistakes uncorrelated. Nevertheless, there are hindrances to this independence that arise from our own *fabric* as persons in a group:

- **Social Proof.** We are social beings, who most of the times think that *"if everybody is doing it, there must be a good reason."* In group behaviour, when things are uncertain, the best thing to do is just to follow along. This is also called *herding*: sticking with the crowd and failing small, rather than trying to innovate and run the risk of failing big. As reputation goes, it is better to fail conventionally then to succeed unconventionally.
- **Information Cascade.** This phenomenon happens while making decisions based on bad judgment (one thinks is right) from who came before. This spawns a sequence of uninformed choices, so that collectively the group ends up making a bad decision. This is not always bad, if all other members are good judgers and spot the occurrence.
- **Imitation.** Most of the time, as a rational response to our cognitive limits, we *piggyback* on the wisdom of others and, most of the time, it works. But it shouldn't be a *slavish* imitation, where blind mimicry hurts the group. It should be an *intelligent* imitation that, if used well, is an effective and powerful tool to spread good ideas fast. Having a wide array of options and information, and the willingness to put their own judgment ahead of the group's, are requisites for this kind of imitation. This can break negative cascades by consciously identifying bad choices.

Independence can be enforced and promoted by making sure, as much as possible, that decisions are made simultaneously (or very close) rather that sequentially, making people pay much less attention to what everyone else is saying. By *keeping the ties loose*, making groups ranging across hierarchies and exposing individuals to as many diverse sources of information as possible, independence can be maintained.

Decentralization

People are able to specialise and draw on local knowledge. Decentralisation fosters (and is fed by) specialisation, increasing the scope and diversity of the opinions and information in the system. The closer to a problem, the more likely a good solution spawns, although there is no guarantee all information reaches everyone. Also it allows for *tacit* knowledge input. This is a very valuable knowledge, yet it is knowledge a person knows because they've been there, but they can really explain or communicate. Individual knowledge remains resolutely specific and local, but becomes globally and collectively useful.

Aggregation

Somehow, there must be a mechanism to *compute, aggregate and broadcast* the private judgments into a collective decision. These mechanisms need to be available to all the members, even unreliably assuring that the information reaches its destination (the member might ignore that knowledge). If this is some kind of *whiteboard* or global, sharable, communication infra-structure, that is not important as long as it serves its purpose.

Resorting to a *wise crowd*, or community, to solve problems can be advantageous. But how do we ask the community for help and effectively capture its answers, i.e., knowledge? Even if the community is only composed of experts, can we effectively tap into their collective knowledge? How do we acquire that knowledge?

7.2.2 Grasping the collective knowledge

Effectively capturing expertise from several heterogenous sources in a social environment is the goal of the Collaborative Knowledge Acquisition field of study, a spin-off of the Knowledge Acquisition domain. A succinct description is presented next.

Knowledge acquisition

The Knowledge Acquisition (*KA*) field deals with the process of extracting, structuring, and organising knowledge from human experts so that the problem-solving expertise can be captured and transformed into a computer-readable form. This captured knowledge forms the basis for the reasoning process of an expert system and has three main concerns: (i) involvement of appropriate human experts, (ii) proper knowledge elicitation techniques and (iii) a structured acquisition approach [Wat86][Lio92b]. The term comes from the field of Expert Systems as the task of gathering the required knowledge from human experts, turning it into a computable form and fuelling the expert system. KA is a complex task with several identified issues that capturing techniques should address [Lio92b] [MD85]:

- **Most (but not all) knowledge is in the heads of experts.** Capturing and sharing this knowledge increases its already high value, although it should be shared in such a way to allow non-experts to understand it.
- **Experts have vast amounts of knowledge.** It is therefore important to focus on the essential knowledge.
- **Each expert doesn't know everything.** Knowledge should be gathered and collated from different experts, and these should be allowed to interact.

- **Experts have a lot of tacit knowledge.** An expert knows more than he/she can account for. Besides being hard (or nearly impossible) to describe, tacit knowledge is also hard to capture.
- **Experts are very busy and valuable people.** Capturing techniques should take experts off the job for short periods of time, ideally, never, if they were seamlessly integrated into their working environment.
- **Knowledge has a "shelf life".** Knowledge evolves. Experts find new knowledge. Therefore knowledge should be maintained and validated throughout time.

As such, KA is a difficult and time-consuming process that frequently creates a bottleneck for building expert systems. It is possible, applying the right tools and methodologies, to improve and mitigate this bottleneck.

In [Cor89], Cordingley provides a survey of knowledge acquisition methods and procedures, with suggestions about in which circumstances different methods are useful. These methods range from informal techniques such as *user observation* through common social science methods (interviews, questionnaires, and discourse analysis) to more formal techniques used in KA for expert systems. The reason for so many techniques lies in the fact that there are many different types of knowledge possessed by experts, and different techniques are required to access the different types of knowledge. This is referred to as the *Differential Access Hypothesis* [And04], and has been shown experimentally to have supporting evidence. Most recently, new developments in methodologies [SAA+00], the emergence of ontologies, improved software tools, and the expansion of knowledge management [Dav98] beyond that of expert systems have brought new insights into KA.

Collaborative knowledge acquisition: abandoning the useless

Knowledge acquisition in a social environment shares the same issues as seen earlier. Additionally, the developer has to rely on distributed knowledge resources (artifacts and people) where not everyone is an expert. This becomes even worse if the community scope goes beyond the team of developers and extends to the web, where other developers may have the answer for a specific problem regarding a well-known shared software artifact, API or framework.

The quality of the retrieved knowledge is evaluated by the behaviour of the community towards that knowledge. *If it is useful, it is used, if not, it is abandoned.* One way of capturing this behaviour is to give the community ways of expressing their intent, whether through rating or commenting. Otherwise, there are ways of implicitly capturing the community behaviour, like *page hits*[2] or *social bookmarking*. This is known as ***Collaborative***

[2] The number of web users that visit that page.

Knowledge Acquisition [Lio92a], as it gathers information from several heterogeneous sources, such is the morphology of the Internet.

Systems that enable this kind of knowledge acquisition are denominated *Collective Knowledge Systems*, as described in the next section. The collaborative environment and toolset proposed by this dissertation can be characterised as such a system.

7.2.3 Collective knowledge systems

In [Gru07], Tom Gruber states that:

> *The web, as a community, is not yet a "collective" intelligence, rather a "collected" intelligence. This comes from the fact that there is no new level of understanding. User-generated content is being shared, gathered and collected in domain-specific sites. We can find what things are more popular or what are the current fads. However, while popularity is one measure of quality, it is not a measure of veracity. Mass authoring is not the same thing as mass authority.*

This classification of **collected vs. collective** intelligence of the web renders a definition of what a Collective Knowledge System can be and its key properties are summarised below:

- **User-generated content.** The bulk of the information is provided by humans participating in a social process. A traditional database or expert system, in contrast, gets the bulk of its information from a systematic data gathering or knowledge modelling process.

- **Human-machine synergy.** The combination of human and machine provides a capacity to provide useful information that could not be obtained otherwise. These systems provide more domain coverage, diversity of perspective, and sheer volume of information than what it could be achieved by searching official literature or talking to experts.

- **Increasing returns with scale.** As more people contribute, the system becomes more useful. The system of rewards that attracts contributors and the computation over their contributions is stable as the volume increases. In contrast, a text corpus and simple keyword search engine does not get more useful when the volume of content overwhelms the value of keywords to discriminate among documents. Similarly, if the reward system encourages fraud or fails to *bubble up* the best quality content, the system will get less useful as it grows.

- **Emergent knowledge.** The system enables computation and inference over the collected information, leading to answers, discoveries, or other results that are not found in the human contributions. *This fourth property is what differentiates a collective from a collected knowledge system.*

Conveying these key properties, a Collective Knowledge System can be composed of the following elements, depicted in Figure 7.1:

- **Community of motivated people with problems and solutions.** These contributors share their expertise and knowledge on the specific domain.
- **Larger population of intelligent people with similar problems.** Who actively search for personalised solutions to their problems.
- **Computer mediated social communication.** Whether through tagging, blogging or commenting, the social process is augmented and nurtured.
- **Semi-structured information repository.** Acting like a storage facility for a more long-term memory and where the solutions are collected and shared.
- **Socially clustered data knowledge-base.** Where the solutions are catalogued and clustered according to the social interaction and multidimensional analysis.
- **Faceted search engine.** So that the solutions seekers can look for personalised solutions, through contextual browsing.
- **Recommendation engine.** To keep the users in perspective and assisting in obtaining more rapid and effective answers to their specific issues.

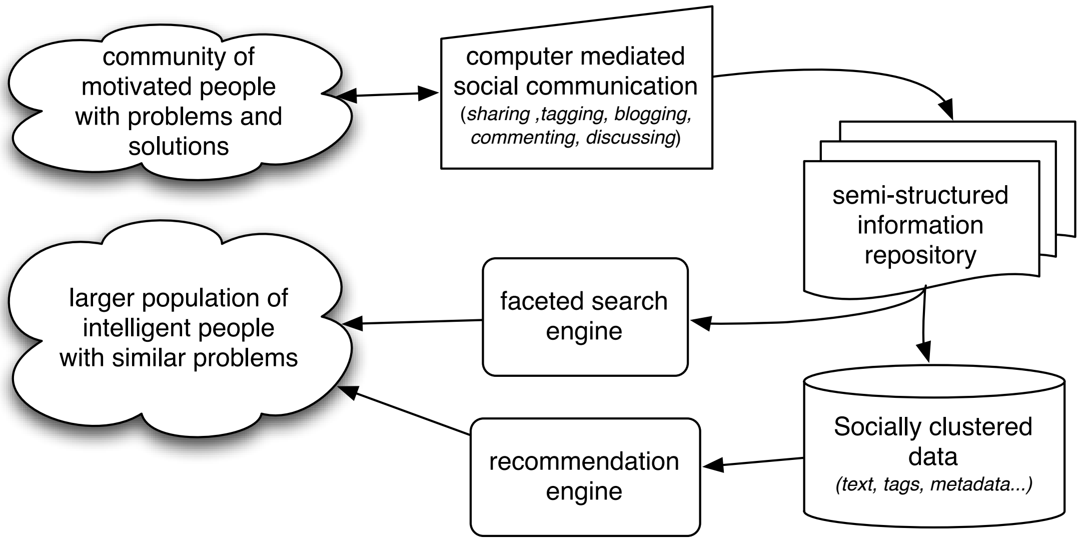

community of motivated people with problems and solutions
computer mediated social communication
(sharing ,tagging, blogging, commenting, discussing)
semi-structured information repository
larger population of intelligent people with similar problems
faceted search engine
Socially clustered data
(text, tags, metadata...)
recommendation engine

Figure 7.1: Composing elements of a Collective Knowledge System (adapted from [Gru07]).

It might be relevant to say that not all of these elements need to be present for a system to be considered of Collective Knowledge. At least, the knowledge quality evolution through social interaction and its access need to be enforced.

7.3 Collaborative framework learning

It is the author's belief that a framework learner may benefit from such a system, as a means to improve the effectiveness of the learning process. This is achieved by providing the learner with a constantly improving source of knowledge built by a community of learners with the same needs and expectations.

This section focus on the framework learning context, presenting the proposed supporting process, its concepts, phases and tool requirements.

7.3.1 Concepts

For the sake of clarity, the meaning of *learner*, *artifact* and *knowledge-base* is explained to prevent misinterpretations when reading the remainder of this section.

- **Learner.** Any framework user, developer or evolver that needs to acquire knowledge about the framework.
- **Artifact.** For the context at hand, this means any documentation artifact available to the learner and present in the proposed collaborative environment. Is is assumed that there is a framework documentation artifacts repository (*FDAR*) present for consultation by the learner.
- **Knowledge-base.** This regards the storing facility of the collaboratively generated learning knowledge. Despite referencing the FDAR, it is a different data-source.

7.3.2 "Pave the cowpath" revisited

The idea behind the proposed collaborative approach evolved from what is commonly known as *"pave the cowpath"*.

The expression has its origin in a poem written by american Sam Walter Foss (1858-1911) called *"The Calf-Path"*[Fos]. The poem tells the story of a strained calf, lost in the woods, who, when returning home, *"made a trail all bent askew, a crooked trail, as all calves do"*. That trail kept being followed by beasts and humans until today, currently being the main streets of a metropolis. The moral is that *"[..] men are prone to go it blind, along the calf-paths of the mind, and work away from sun to sun, to do what other* *men have done."* This poem serves as criticism to the lazy, narrow-minded men that mindlessly follow pre-defined paths without questioning its effectiveness or usefulness. Another similar allegory is the *Cage of Monkeys*.[3] This poem is popularly attributed to the streets of Boston, given their peculiar layout.

Of course, this connotation has issues. Its common knowledge that cattle are actually pretty good at finding the path of least resistance, which is, often the best route for a road. But let's transpose this concept to the context at hand.

"Smart cows, collective herd"

Cows walk with their heads down, are beasts of habit and usually move in herds. But a *solitary* cow is, usually, smarter than the bunch as it can't rely on the group to reach her goal, whether reaching a pasture or returning home. In fact, they have a good sense of direction and can quickly retrace their steps back to the herd or to the point of origin. The herd factor is simply a matter of blind trust. *Smarter cows* keep their head up and introduce independence to the herd, making it more wise, and question the effectiveness of the trail they take.

Transposing to the collaborative framework learning context, all *cows* (framework learners) are smart. Therefore, the *cowpath* becomes **the steps the learners took to reach a solution**. The problem is that there is no *stepping on the grass*, that is, those steps aren't being recorded. Most probably, the next learner that undertakes the same steps will not be aware of a *pathway* forming. By *paving* that pathway, it becomes easier for future learners to quickly reach the same solution. This pathway is called **learning path**.

Providing framework learners (*smart cows*) with learning paths (*paved cowpaths*), improves their learning experience by focusing of the relevant knowledge (*steps*) other learners (*collective herd*) already have. This allows for a quicker, more effective knowledge transmission, in the sense that it provides the learner with directions on which artifacts to look at and in what order.

[3] Imagine a cage full of monkeys where a ladder is the only way to reach a banana bunch. Every time a monkey tries to climb the ladder, the keeper showers all the monkeys with a hose of ice-cold water. This happens until all monkeys stop trying to climb the ladder to try to reach the bananas. Then a monkey is replaced by a new one. The new monkey, naturally, tries to climb the ladder but all other monkeys stop him by, savagely, beating him up. Every time the new monkey attempts to climb the ladder, the beating ensues. He eventually gives up. Then another monkey is replaced and the pattern repeats itself. Eventually, all hosed monkeys are replaced by new ones (that never knew of the hosing) yet the beating pattern continues, without apparent logical reason but *"instated traditional behaviour"*, thus the expression *"monkey see, monkey do."*

Pavement decays

If a road is not used and maintained, its pavement breaches, erodes and decays, making it harder to use. It might be because there is a better road than this one. This is also true with learning paths. The quality of the learning paths is maintained by the community of learners. The most useful and effective learning paths are prone to evaluation and rated accordingly. This rating indexes the learning paths, so that the most used and approved by the community of learners are presented first. It follows the *If it is useful, it is used, if not, it is abandoned* rule seen previously § 7.2.2 (p. 114).

7.3.3 The learning knowledge cycle

Putting it simply, the author believes that providing a learner with the steps others (learners) took to solve their problems, can improve the learning experience and produce better and quicker outcomes. The motto is: *Show me how you learnt it.* This section details the four-step learning knowledge cycle (Figure 7.2) the proposed collaborative approach defines as a means to support the previously stated. The goal is to non-intrusively **capture** the learning steps a framework user takes, **store** it in a **shareable** knowledge-base, where other users can access it. This knowledge relies on the community's potential to maintain its relevance and quality, by **rating** it and allowing the system to **recommend** possible next steps that aid on the learning task. The four steps are detailed next.

Capture

This is the first step of the learning cycle. Here the learner begins her learning quest to find knowledge that might solve her problem. The trail of steps is captured as she browses through the artifacts, trying to find the relevant knowledge that might help her. This step ends when she is satisfied with her findings.

Filter/Store

On the second step, the learner looks at her captured learning path and *clears the weeds*, that is, improves it. This is done by trimming off those steps that, despite taken, didn't lead to the required knowledge. Seldom a novice learner takes a straight route to the knowledge she needs, unless in cases where she is already strongly familiar with the artifacts and needs little or no assistance to reach her answer (she would, by then, be considered an expert). This step allows for the improvement of the captured learning path, as to prevent other learners from *running in circles* or hitting *dead-ends*. Afterwards, the *pruned* and *grafted* learning path is stored in a knowledge-base.

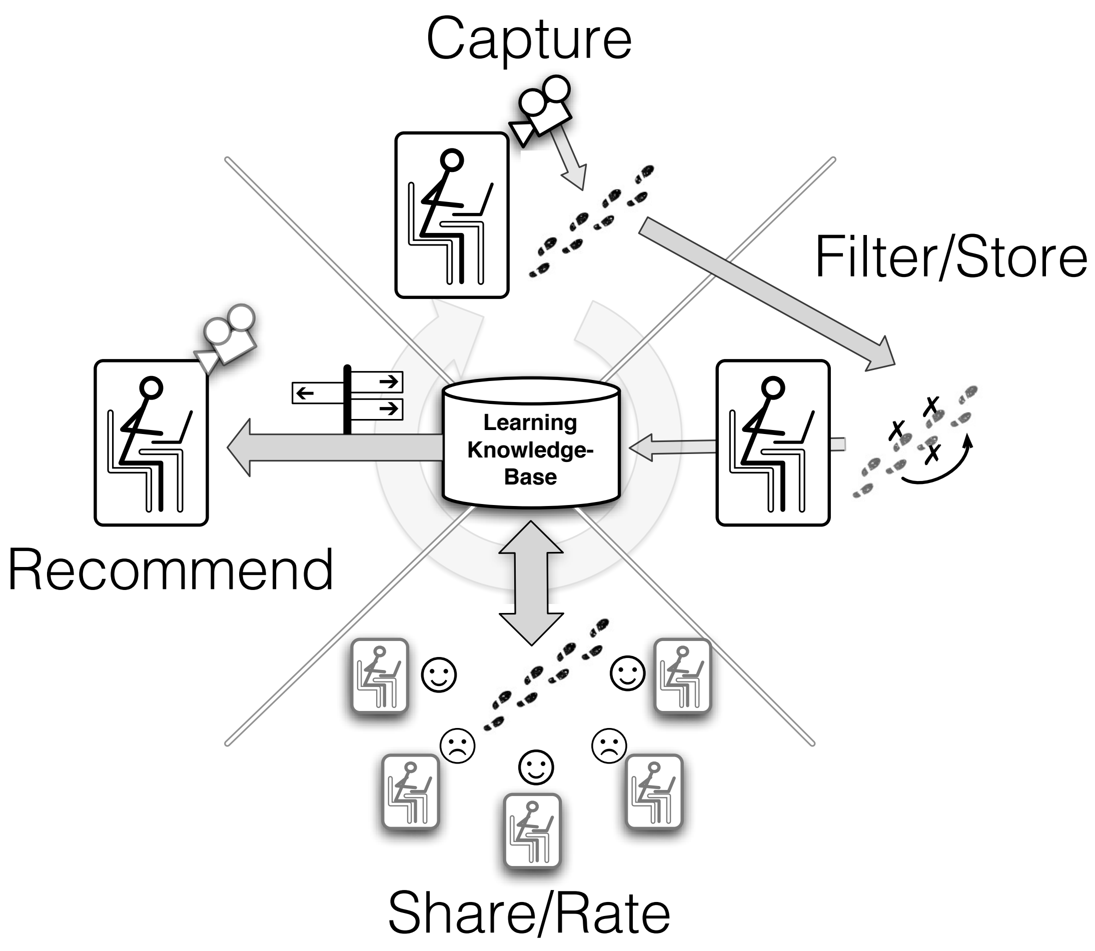


Figure 7.2: The proposed four-step learning knowledge cycle.

Share/Rate

The third step regards the sharing and rating of the learning paths stored in the knowledge-base. The learners access the knowledge-base, searching for learning paths that might help them. They evaluate its usefulness (taking the steps, a.k.a., *walking through* or just inspecting the visited artifacts) and rate them according to its effectiveness. There are no standard quality metrics here, the learner simply gives her opinion on how satisfying a specific learning path was for the current context.

Recommend

This step enables the recommendation of possible next steps (on a learning path that is being currently captured), based on previous learning paths other learners have took. As such, this step occurs during the first one (Capture). Of course, this recommendation has an heuristic that relies on the amount of learning paths already captured. The more learning paths get captured (and rated), the better results the recommendation step provides. Usually, this is intrinsically sensed by the community, so this step is a motivating feature that spurs the participation of the community.

7.3.4 Learning knowledge categorisation

Knowledge is useless if you can't get to it. In order to be able to access the information, we have to give it meaning. As with the notion of *Definition* [Lon10], humans need some form of classifying information so that they can, semantically, store it and access it easily. The web communities have, consciously or not, developed a light form of providing this categorisation, through what is called *tagging.*

Tagging

Tagging is the labelling of an entity (usually a web page or something with a URI[4]) with words or phrases so one can remember them later and group them with related finds. This was a shift from a common, rigid and hierarchical form of categorisation (e.g. folders) into a more flexible, grouping and descriptive-like form. It provides more means to obtain information as it enriches the identification of objects. On a folder-based categorisation, one needs to remember the exact name of the folder, whether, in a tag-based categorisation, one only needs to remember an aspect of the object in question (that hopefully was tagged that way). Of course neither approach is perfect, therefore they can be combined, complementing each other[5].

This tagging phenomenon has quickly spread across the web, and led to the notion of *folksonomy.*

Folksonomy

The term *Folksonomy* is credited to Thomas Vander Wal, for combining the words *"folk"* and *"taxonomy"* to create this neologism from what he calls *"bottom-up social classification"* [Wal]. Shirky defines it as *"socially created, typically flat name-spaces"*. An essential feature of these terms is their public nature, that allows users to instantly determine how others have used the same terms in categorizing their own content, and view terms others have added. This cycle of use and observation enables the community to shape the folksonomy, encouraging useful applications and eliminating useless ones [Shi]. As such, a good definition for folksonomy is given by Sturtz [D.N04]:

[4] Uniform Resource Identifier.

[5] As a quick example, recently, Google Mail has the notion of *labels* to tag mail messages and allows an hierarchisation of tags, similar to folders. Yet a mail message can have several labels.

In practical terms, a folksonomy is the complete set of tags - one or two keywords - that users of a shared content management system apply to individual pieces of content in order to group or classify those pieces for retrieval. Users are able to instantly add terms to the folksonomy as they become necessary for a single unit of content.

The value in this social tagging is derived from people using their own vocabulary and adding explicit meaning, which may come from inferred understanding of the information/object. People are not so much categorising, as providing a means to connect items (placing hooks) to provide their meaning in their own understanding.

Popular examples of systems that use such categorisation are Flickr[6], where users can tag their digital photographs while uploading them to the system and YouTube[7], where the same is allowed for uploaded videos.

Consequently, the proposed learning knowledge cycle uses these notions to categorise its elements, that is, its learning paths. Tags are used by the learners to both store and search for the information they need. Therefore, there is no forced categorisation or taxonomy imposed to the community. It is the community itself that shapes the way it wants the information to be categorised.

7.3.5 Collaborative environment requirements

The previous sections addressed the theoretical concepts and issues from where the author based his proposed work. As such, the proposed collaborative environment, poised to assist on framework learning, should satisfy the following requirements [FA10], to enable the persecution of its intended goals:

- **Seamlessly integrated.** Bringing collaboration into the learning process should not force the framework learner to leave her usual development environment, or, at least, her usual learning environment. Learning comes from documentation and code, which may, or may not, be accessed at the same place (i.e. application or IDE). Therefore, that should be the same media used to provide community help to the framework learner.

- **Non-intrusive, non-interruptive.** Ideally, capturing the learner's intrinsic knowledge should be implicit. That is, the learner should not be asked to explicitly provide any information (regarding that knowledge) to the system. In practice, a satisfactory solution would be not to disrupt the normal functions of the learner, asking very little of her.

[6] `http://www.flickr.com`
[7] `http://www.youtube.com`

- **Descriptive, not prescriptive.** The system should not tell the learner how to proceed, but instead should give possible directions on how to solve the task at hand. Although some unexperienced, novice learners would appreciate a *script-like* support of their learning tasks, it would rapidly become constraining, intrusive and non-effective. Not to mention the inability to provide the exact course of action to reach the solution that solved her specific problem.

- **Shareable knowledge.** The system should store and share all the relevant knowledge that helps the framework learning process. Not only the documentation artifacts and source-code (if applicable[8]), but the captured knowledge that helps guiding the learner throughout the learning process (e.g. learning paths).

- **Learning knowledge cycle support.** The previously presented four-step learning knowledge cycle § 7.3.3 (p. 119) should be supported, whether through one or a set of tools.

- **Extensible.** There should be potential for improvement and to integrate other useful tools that will provide increasing support to the learning process.

7.4 The DRIVER platform

With these requirements in mind, an instantiation of this environment was implemented, whose presentation and description is contents of the following sections. It was designed to address all the issues presented in § 7.3.5 (p. 122).

7.4.1 Setting

The task of providing a collaborative environment brings to mind several ideas on how to proceed, but, overall, there is a common ground that immediately settles in: the *web*. Looking into the *web* and finding out implementations of collaborative systems results in lots of available solutions. Selecting one is a challenge (see section 7.5). Mainly, we are dealing with documentation artifacts that should be readily available, easily searchable and prone to modification, extension and enhancement. Equally, they should be, somewhat, structured and presented in a familiar fashion without too much overhead on the medium by which they are delivered. Therefore, the author chose a *wiki*.

[8] Ultimately, the source code would be present in the documentation through *Graded Examples* [AD06a].

Wiki

Earlier in 1995, Ward Cunningham wrote a set of scripts that allowed collaborative editing of webpages inside the very same browser used to view them [Cun]. He named this system *WikiWikiWeb*, due to the analogy between the meaning of the word *wiki* — quick[9] — and the underlying philosophy of its creation: a *quick-web*. Since then, *wikis*[10] have gradually become a popular tool on several domains, including that of software development [Lou06], e.g., to assist the creation of lightweight software documentation [Agu03]. They ease collaboration, provide generalized availability of information, and allow the combination of different types of content, e.g., text, images, models, and code, in a common substrate [AD05b].

As such, and from the extensive list of available wiki engines[11], the author chose the *dokuwiki* engine [dok], mainly for the following reasons:

- **Lightweight.** A *dokuwiki* is simple, small (code-wise) and doesn't require much effort to install and use. It is written in PHP and doesn't require an auxiliary database as it resorts to a file-based architecture.

- **Semi-structured.** Most wiki engines exhibit an unbalanced measure of structure, that is, some provide a heavily structured, heavily constrained scaffold for placing information, while others have no structure beyond the notion of a wiki page (similar to a web page). *Dokuwiki* provided a lightweight structure form, having two levels of structure: **pages** and **sections** (inside a page). As it will be seen later, this provided the sufficient amount of structure for the development of the required framework documentation artifacts repository.

- **Extensible.** *Dokuwiki* is open-source and can be seamlessly extended by adding plug-ins, without tampering with the native wiki source-code.

- **Familiar.** The author had previous contact with this wiki engine and its familiarity would prove essential to enable a sustainable pace when developing the required tools for the proposed collaborative environment.

From the above stated, the wiki serves as a foundation for building the collaborative environment, providing a suitable set of building blocks to harbor, not only the documentation artifacts, but the toolset that will enhance the collaboration and support the learning knowledge cycle.

[9] the expression *wikiwiki*, in Hawaiian, means *quick*.
[10] After the creation of the *WikiWikiWeb*, several new sites and systems — or engines — emerged based on the same underlying principles, and are generally called *wikis*. Among them is the well-known *Wikipedia* [wik], based on the *MediaWiki* [med] engine.
[11] This list can be found at `http://www.wikimatrix.org`, where an extensive comparison between engines can be found.

Framework documentation artifacts repository

The contents of the wiki compose the framework documentation artifacts repository (*FDAR*). These artifacts follow a series of formats according to the related set of patterns PATTERNS FOR EFFECTIVELY DOCUMENTING FRAMEWORKS [AD11], referred to in § 6.5 (p. 85).

The wiki contents are, therefore, composed of instantiations of these artifacts related to a specific framework, available for the learner to read, browse and enhance. An example of a *Framework Overview* artifact present on the wiki is shown in Figure 7.3.

For an easier creation of these artifacts by the framework user, the wiki is enhanced with a plug-in[12] that allows the definition of a set of *templates* that pre-formats the artifact with the most common sections and content holders it bears. These are, obviously, optional, but will guide the user in creating the most suitable format to accommodate both his, and the audience needs. Consequently, the user can easily create new artifacts and promptly improve the documentation repository.

Additionally, there is a plug-in[13] that allows the tagging of pages, so that the user can build a folksonomy § 7.3.4 (p. 121) for the documentation. These tags can prove useful later on, helping on the tagging of learning paths.

Patterns

Both the patterns proposed by this dissertation and presented in Chapter 6, as well as the documentation artifacts patterns referenced in the previous section, are present in the wiki. The learner can read through both sets of patterns and become aware of their proposed solutions, as means to improve her learning experience.

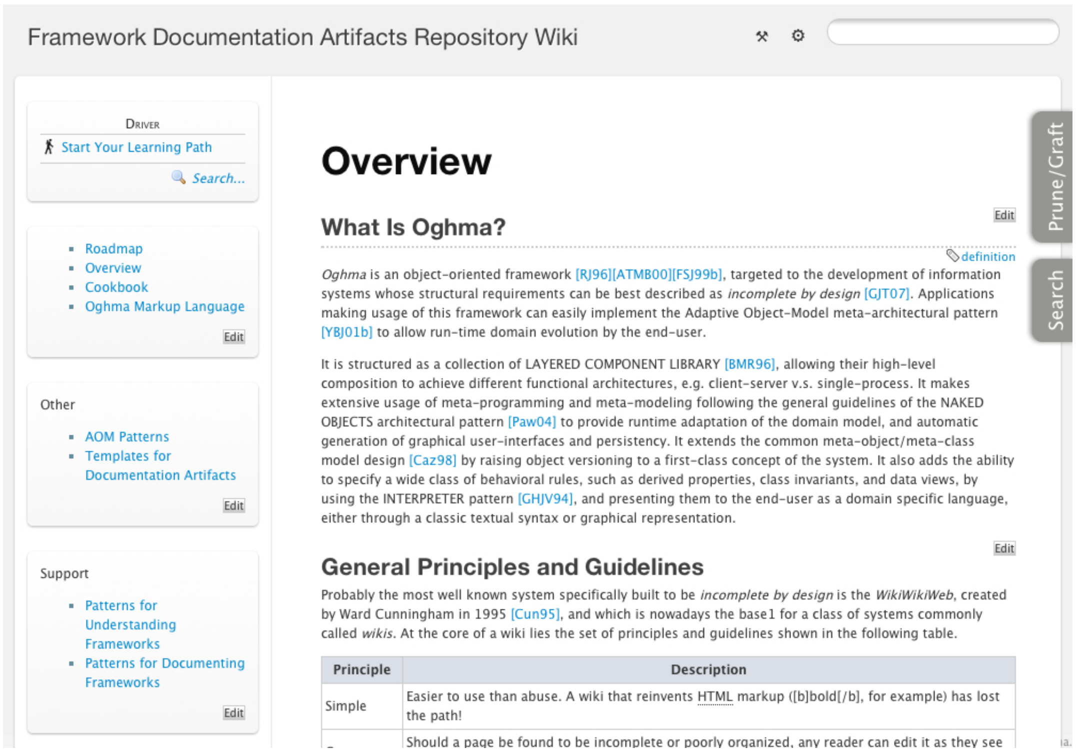
7.4.2 Components

The DRIVER toolset was developed as a bundle of dependent plug-ins for *dokuwiki*, with the sole purpose of supporting the learning knowledge cycle presented in § 7.3.3 (p. 119). Component-wise, it can be divided into 5 main complementary items, which are detailed below.

Capture

The **Capture** plug-in allows the system to become aware of the learning steps of the user, by tracking all the navigation along the wiki. All pages and sections the user reads are

[12] *snippets* plug-in, by Michael Klier (available at `http://www.dokuwiki.org/plugin:snippets`)
[13] *tag* plug-in, by Gina Häußge and Michael Klier (available at `http://www.dokuwiki.org/plugin:tag`)


Framework Documentation Artifacts Repository Wiki
Driver
Start Your Learning Path
Search...
Roadmap
Overview
Cookbook
Oghma Markup Language
Edit
Other
AOM Patterns
Templates for Documentation Artifacts
Edit
Support
Patterns for Understanding Frameworks
Patterns for Documenting Frameworks
Edit
Prune/Graft
Search
definition
Edit

Overview

What Is Oghma?

Oghma is an object-oriented framework [RJ96][ATMB00][FSJ99b], targeted to the development of information systems whose structural requirements can be best described as *incomplete by design* [GJT07]. Applications making usage of this framework can easily implement the Adaptive Object-Model meta-architectural pattern [YBJ01b] to allow run-time domain evolution by the end-user.

It is structured as a collection of LAYERED COMPONENT LIBRARY [BMR96], allowing their high-level composition to achieve different functional architectures, e.g. client-server v.s. single-process. It makes extensive usage of meta-programming and meta-modeling following the general guidelines of the NAKED OBJECTS architectural pattern [Paw04] to provide runtime adaptation of the domain model, and automatic generation of graphical user-interfaces and persistency. It extends the common meta-object/meta-class model design [Caz98] by raising object versioning to a first-class concept of the system. It also adds the ability to specify a wide class of behavioral rules, such as derived properties, class invariants, and data views, by using the INTERPRETER pattern [GHJV94], and presenting them to the end-user as a domain specific language, either through a classic textual syntax or graphical representation.

General Principles and Guidelines

Probably the most well known system specifically built to be *incomplete by design* is the *WikiWikiWeb*, created by Ward Cunningham in 1995 [Cun95], and which is nowadays the base1 for a class of systems commonly called *wikis*. At the core of a wiki lies the set of principles and guidelines shown in the following table.

Principle	Description
Simple	Easier to use than abuse. A wiki that reinvents HTML markup ([b]bold[/b], for example) has lost the path!
–	Should a page be found to be incomplete or poorly organized, any reader can edit it as they see

Figure 7.3: Example of a *Framework Overview* documentation artifact present in the wiki, as an overall depiction of the collaborative environment view.

logged and recorded, so that the user can, progressively, build her learning path. The start and end of this capture is signalled by the user, therefore the system has two global states: *capturing* and *not capturing*.

Search

The **Search** plug-in allows any user, at any time (no matter what state the system is in), to query the knowledge-base for learning paths that might prove useful. This query relies on tags that previous users have used to describe their learning paths[14], and the search results are sorted according to the rating score each learning path has. The user can then preview and *walkthrough* a specific queried learning path to find out if it's really helpful. Additionally, the user can rate any of the queried learning paths according to its usefulness or even change the scores of learning paths she has already rated. All of this is done without disrupting any previous navigation the user has made in the wiki, as the plug-in resides, graphically, in a different layer (side tab).

[14] The plug-in provides an *auto-completion* feature, e.g., while the user is typing her tags, the system provides a list of already existing tags in the knowledge-base

Prune/Graft

The **Prune/Graft** plug-in allows the user to build her (final) learning path for storage (sharing) purposes. At any time during the *capturing* state, the user can access all the steps of the learning path (captured so far) and re-arrange them to compose the relevant order by which she reached a solution. The composed learning path can then be tagged for indexing purposes and stored in the knowledge-base for access by the community of learners. The user can keep building her learning path (resuming her navigation on the wiki) without loosing the captured steps, until she signals the end of her learning activities and the system changes to a *not capturing* state. Even so, the captured steps won't be lost until a new *capturing* state is signalled.

Hint

The **Hint** plug-in allows the system to recommend possible directions the user might take towards her solution. This plug-in only operates when the system is in a *capturing* state. Depending on the present *position*[15] of the user, the system shows a list of possible directions the user might browse next, according to the existing learning paths in the knowledge-base. The user can then, if so wishing, navigate directly to those artifacts.

(Learning knowledge) Database

This plug-in is, non other than, the implementation of the learning path knowledge-base. It assembles and provides access to the database of learning paths. It was developed to accommodate the need to store and share the learning paths in a relational database structure, whereas the *dokuwiki* engine had no such infrastructure. Nevertheless, it relied on a file-based database technology to maintain simplicity.

7.4.3 Usage

This section shows a usage example of the toolset, as a means to provide a better understanding of how the learning knowledge cycle is supported and to show the implemented user interface. It is assumed that the framework learner has access to the wiki and has already sign in with her credentials, which give permission to view the documentation and use the plug-ins.

[15] The documentation artifact (i.e. wiki page) the user is currently browsing.

Start your learning path

As depicted in Figure 7.4, the learner has access to a small menu on the top left side of the wiki. This menu allows her to signal she's about to start her learning, so the system can begin capturing her navigation steps.

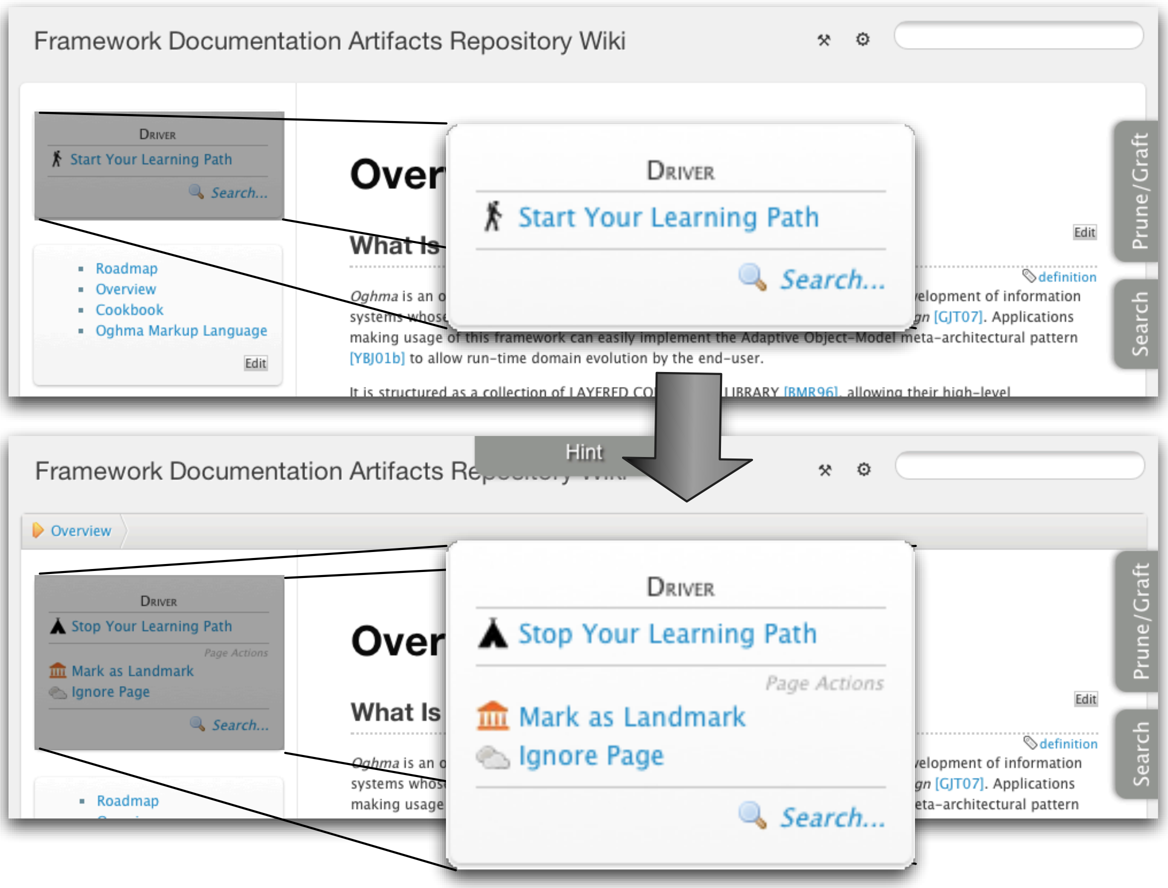


Figure 7.4: DRIVER menu when going from *not capturing* to *capturing* state.

After entering the *capturing* state, the system provides the following new actions to the learner while she is browsing the wiki:

- **Stop your Learning Path.** Used to signal the system that she no longer wants her browsing steps to be captured. The system moves to the *not capturing* state and the menu resumes its previous set of actions.

- **Mark as Landmark.** This action enables the learner to mark a page as *of noticeable importance* to her learning experience. If she reaches a partial conclusion (not yet the final answer), or realises something important to her knowledge needs (that she wants to point out and share), this feature allows her to do so.

- **Ignore Page.** This action marks the current page as *non-capturable*, i.e., every time the user returns or browses through this page, it is not added to the existing browsing steps. This is useful if the learner is recurrently going through a certain page, probably because it is a central page with many links or an index page.

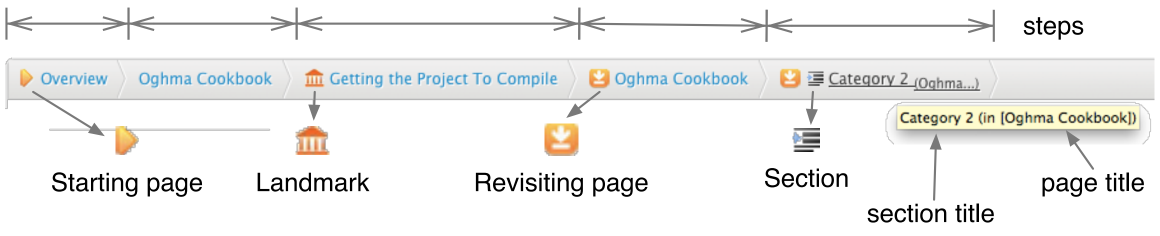


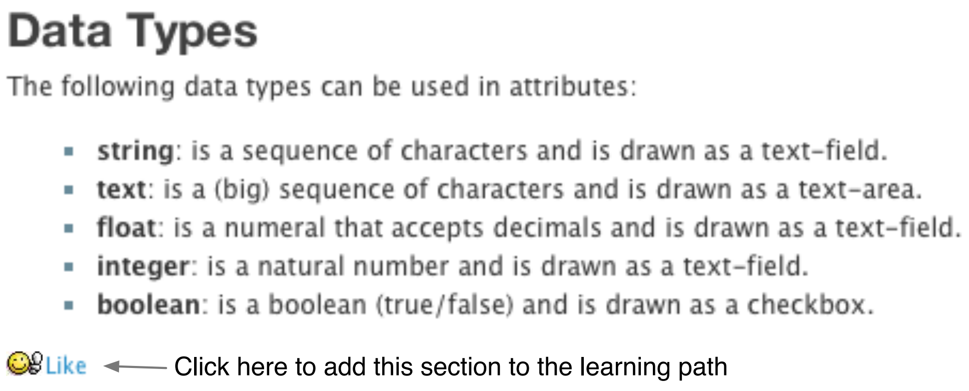
Figure 7.5: Depictions of the DRIVER's learning path trail of the last steps, explaining its constituent parts and icons.

Besides these new actions, the learner is presented, in the top of the page and at all times, with a *trail* of the last steps she took (and that were captured) while browsing the wiki. These steps can have associated markings or icons, as explained in Figure 7.5. Each cell or step is composed of an icon (optional) and the captured wiki page title. Both these elements are clickable anchors to their respective pages, so that the user can directly navigate to them. In the case of sections, the section title appears first, followed by an abbreviation of the page the section belongs to (subscript and bracketed). The tooltip shows the full name of the page.

Capturing a page being browsed is easy because the *dokuwiki* allows for the capturing of the navigation between wiki pages. The same is not true for sections. In the case of sections, the user has to manually mark the section as read. For this, each section has a *like* action link at the end of its contents. Consequently, the user can add that particular section to the captured learning path. Usually this enables parts of a documentation artifact to be emphasised, instead of the whole document. Later on, the learner can re-arrange the final learning path to indicate only these sections and not the whole document. This *like* action link only appears when the system is in *capturing* state (Figure 7.6).

Filtering and storing

At any time during the *capturing* state, the learner can filter and store her learning path. For that purpose, she uses the Prune/Graft plug-in, accessible by clicking on a tab on the right side of the wiki. This tab is visible at all times, despite the state the system. If the system is at a *not capturing* state, the available learning steps remain from the last capturing session.


Data Types

The following data types can be used in attributes:

- **string**: is a sequence of characters and is drawn as a text-field.
- **text**: is a (big) sequence of characters and is drawn as a text-area.
- **float**: is a numeral that accepts decimals and is drawn as a text-field.
- **integer**: is a natural number and is drawn as a text-field.
- **boolean**: is a boolean (true/false) and is drawn as a checkbox.

Figure 7.6: Adding sections to the learning path by hitting the *like* action link at the end of the section.

The Prune/Graft plug-in opens up on a different layer from the wiki itself, providing a set of features without conflicting with the current work session. The user interface can be seen in Figure 7.7. As such, the following functionalities are allowed:

- **Drag and drop learning steps.** The user is presented with all the learning steps she took during the learning session[16]. These can be dragged into a different area where she can compose the effective learning path she wishes to store and share. The steps can be re-arranged at will.

- **Preview learning step.** The learning steps can be previewed in an auxiliary pane so that the user can confirm, if her memory fails, the contents of that step. The user just have to click on the name of the step, which, at this point, loads that page into the pane. This functionality can be seen in Figure 7.8.

- **View similar existing learning paths.** While the user is dragging steps into the effective learning path area, the system searches for similar learning paths that already exist in the knowledge-base. This allows the user to check for relevant discrepancies between the learning path she is building and existing others. If the system encounters an exact match of the learning path the user is constructing, it does not duplicate when saving. Instead, it will merge the tags the user assigns to her learning path with the existing ones. This functionality can be seen in Figure 7.9.

- **Tagging.** At the bottom of the page, the user can tag the effective learning path using, again, a drag and drop behaviour. She has a list of tags that were present in the captured wiki pages and from where she can drag the tags she wants to assign

[16] All steps so far (if still in *capturing* state) or all steps from last session (if in *not capturing* state)

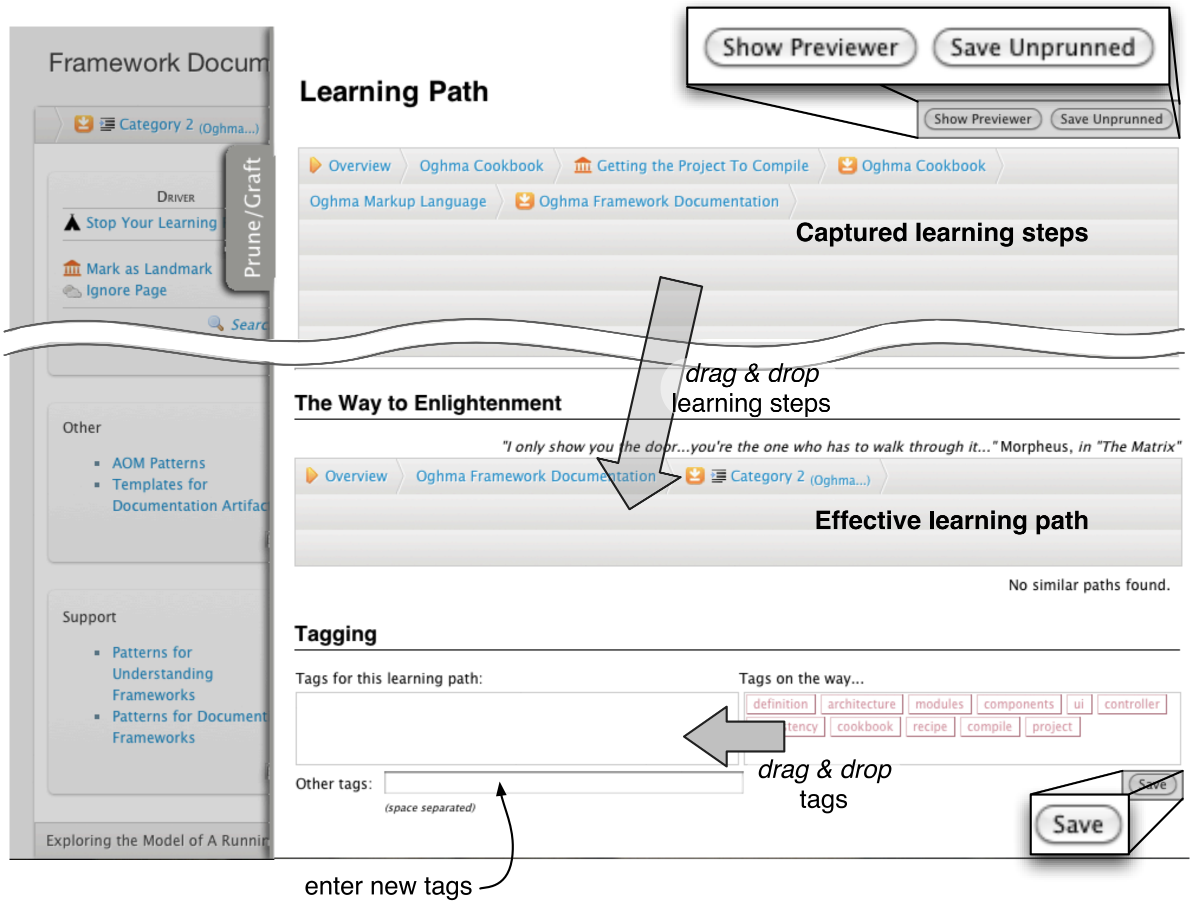


Figure 7.7: Prune/Graft plug-in user interface, showing the main functionalities

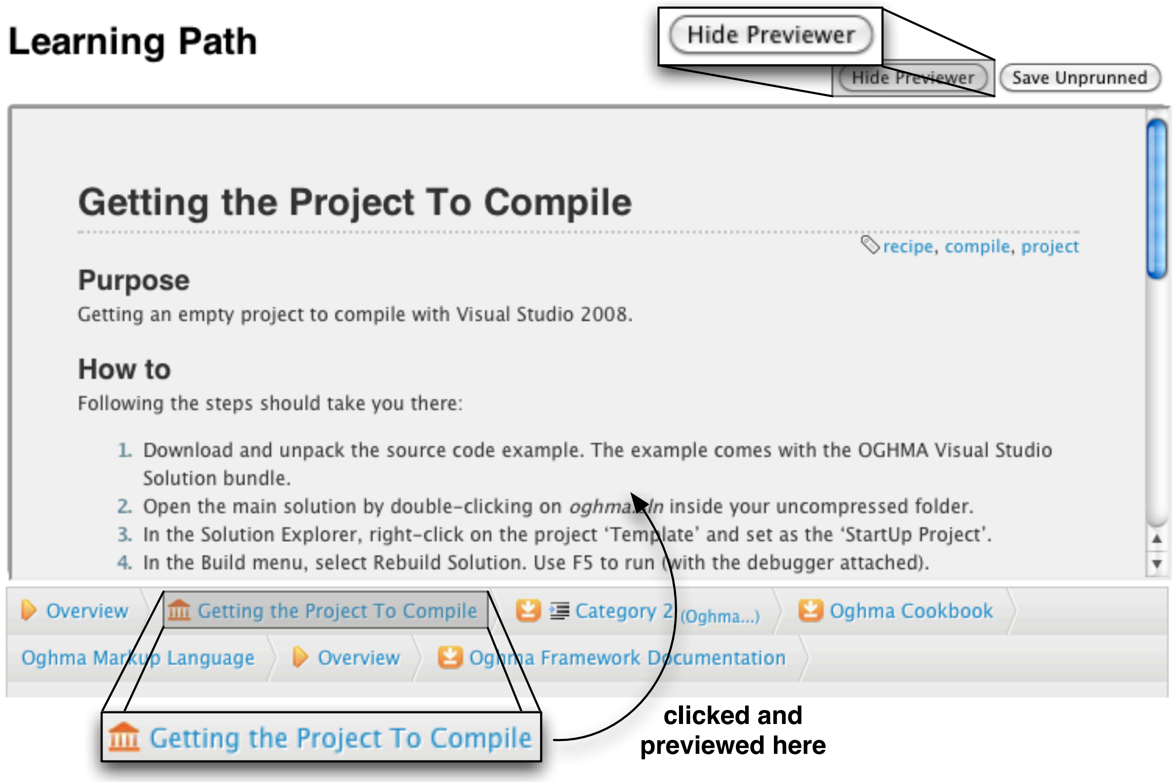
the learning path. There is also the possibility of adding new tags through a text box.

- **Save.** The user can save the effective learning path she has composed, tagged accordingly.
- **Save unprunned.** The user can save the learning path *as is*, i.e., exactly as it was captured, having only to tag it accordingly.

Searching and rating

The quickest way to access the knowledge-base of learning paths is to use the Search plug-in. For that, the learner just have to open the respective tab (visible at all times) on the right side of the wiki, or clicking on the *search* option on the menu on the left sidebar.

The interface is quite simple and straightforward (see Figure 7.10), if you are used



Getting the Project To Compile

recipe, compile, project

Purpose

Getting an empty project to compile with Visual Studio 2008.

How to

Following the steps should take you there:

1. Download and unpack the source code example. The example comes with the OGHMA Visual Studio Solution bundle.
2. Open the main solution by double-clicking on *oghma.sln* inside your uncompressed folder.
3. In the Solution Explorer, right-click on the project 'Template' and set as the 'StartUp Project'.
4. In the Build menu, select Rebuild Solution. Use F5 to run (with the debugger attached).

Figure 7.8: Prune/Graft plug-in user interface, showing the preview pane where the user can preview any step from his capturing session. The preview pane visibility can be toggled using the Show/Hide button on the top.

to the popular web search engines (Google[17], Yahoo[18] or Bing[19]). The user just have to enter[20] the tags by which she wants to query the knowledge-base and the matched learning paths will be listed.

The search heuristics finds all learning paths that have those tags, where the learning paths with the most matched tags appear first, sorted descendent by rating.

Each result item has 5 constituent elements:

- **Steps.** This shows the learning path steps, conveying to the standardised form of showing learning paths as depicted in Figure 7.5.
- **Tags.** On the top left corner of each result item, the tags associated with that learning path are listed, where the matched tags are shown in bold.

[17] `http://www.google.com`
[18] `http://www.yahoo.com`
[19] `http://www.bing.com`
[20] The text box where the tags are entered has an *auto-complete* feature that shows existing tags on the knowledge-base so that the user can maximise the hit ratio.

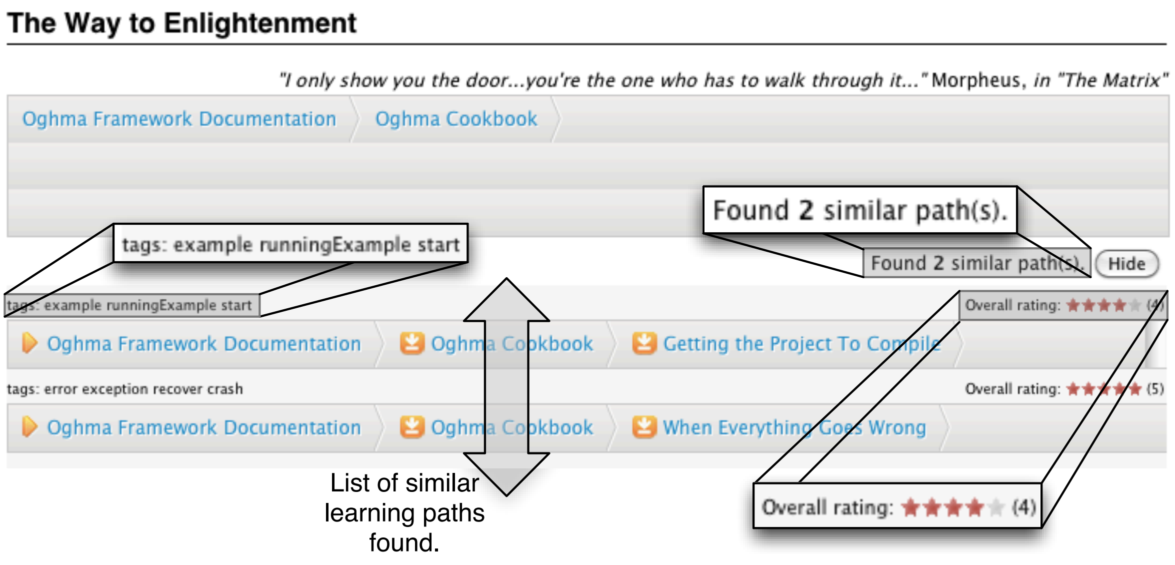

The Way to Enlightenment
"I only show you the door...you're the one who has to walk through it..." Morpheus, in "The Matrix"
Oghma Framework Documentation
Oghma Cookbook
Found 2 similar path(s).
Hide
tags: example runningExample start
Overall rating: ★★★★ (4)
Oghma Framework Documentation
Oghma Cookbook
Getting the Project To Compile
tags: error exception recover crash
Overall rating: ★★★★★ (5)
Oghma Framework Documentation
Oghma Cookbook
When Everything Goes Wrong
List of similar learning paths found.

Figure 7.9: Prune/Graft plug-in user interface, showing similar learning paths when found. The user can toggle the showing of this list. The shown elements are the same as if doing a search, except for the rating ability.

- **Overall rating.** On the top right corner of each result item, the overall rating of that learning path is shown. The rating value varies between 0 and 5 and follows a *star-like* graphical representation, complemented by its numerical value in brackets.

- **User rating.** On the lower right corner of each result item, the user can view and change her personal rating of that learning path. The *Rate* button commits the user's rating, entered by using a similar *star-like* interface.

- **Previewer.** On the lower left corner of each result item, there is a *Show Previewer* button that allows the user to preview the steps of the learning path, similarly to the same operation on the Prune/Graft plug-in. The only difference is that the preview pane appears below the respective learning path.

As such, the Search plug-in not only allows finding a suitable learning path, but enables the community to give their feedback on the usefulness of the learning paths, improving the effectiveness of the learning knowledge.

Recommending

While in the *capturing* state, the system can provide the learner with directions on possible next steps in her learning path. This is done using the Hint plug-in. As shown in Figure 7.11, the user has access to a tab at the top of the page that slides open to show a list of recommended next steps, based on the existing learning paths in the knowledge-base.

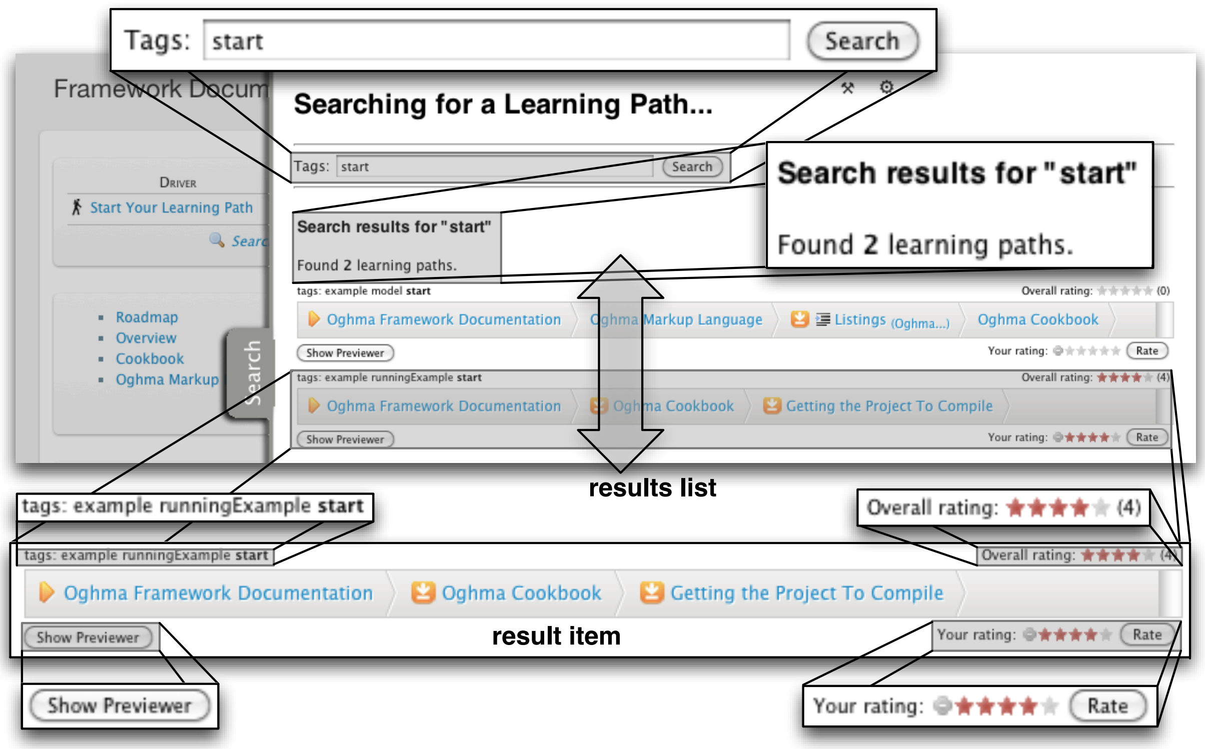


Figure 7.10: Search plug-in user interface, showing results for a query with the tag **start**. There is also a depiction of all the elements present in a result item. The *Show Previewer* button works similarly to the preview functionality in the Prune/Graft plug-in, showing a preview pane directly below the respective result item.

The list item is composed of clickable links to the page or section to where the user can directly navigate. Prefixed to each item is the number of learning paths (in brackets) that have the current page as the previous step.

This enables the learner to rely on the knowledge captured from the community to assist on her learning process. Generally, this only happens when the learner is at a loss, and seems to be *disoriented* in her quest for knowledge that might help her. The common behaviour is to fall back to the last *known* point, i.e. documentation artifact that appears to lead somewhere, and try to proceed from there. The Hint plug-in can, then, provide an *educated* guidance to possible solution directions. Of course, if the learner is hacking her way into virgin land, i.e. paving new learning directions, the Hint plug-in won't be of much use.

7.4.4 Limitations

The DRIVER toolset is still under development and improvement. At the writing of this dissertation, the current version still exhibits a few limitations, namely:

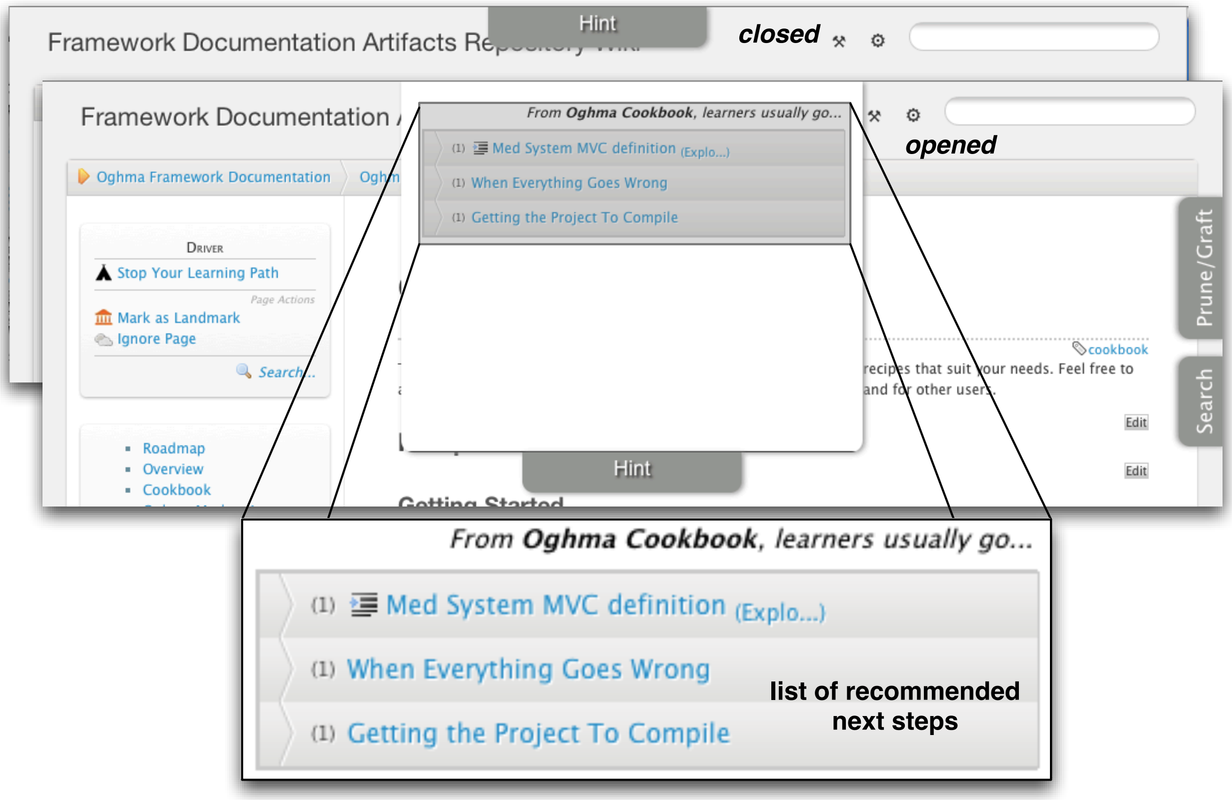

Framework Documentation Artifacts Repository Wiki
Hint
closed
Framework Documentation
opened
Oghma Framework Documentation
Driver
Stop Your Learning Path
Page Actions
Mark as Landmark
Ignore Page
Search...
Roadmap
Overview
Cookbook
Prune/Graft
Search
cookbook
recipes that suit your needs. Feel free to and for other users.
Edit
Getting Started
From Oghma Cookbook, learners usually go...
(1) Med System MVC definition (Explo...)
(1) When Everything Goes Wrong
(1) Getting the Project To Compile
list of recommended next steps

Figure 7.11: Hint plug-in user interface, showing recommended next steps, based on the current location of the user. The picture shows the hint tab located at the top of the page on a closed state and then on an opened state.

- **Knowledge-base maintenance.** Once a learning path is captured and stored, it remains unchanged, unless for its rating score. This is not a bad thing, nevertheless, it is error-prone, specially when dealing with novice users that might, inadvertently, store *poorly filtered* learning paths or even users that, accidentally, store a partially pruned learning path. As such, the knowledge-base would need a *garbage collector* to clean itself from these excessive learning paths (despite the community's rating-based mechanism to abandon these useless items).

- **Deleted documents.** If a page or document, that is present in the wiki at the time a learning path is captured, is later deleted or moved, the knowledge-base is not yet prepared to keep the learning paths synchronised with these changes in the wiki. Actually, there is still issues on how to deal with deleted documents and pages, and the effect it will have on a containing learning path.

7.4.5 Towards a collective knowledge system

As illustrated in Figure 7.12, the DRIVER toolset has all of the properties of a collective knowledge system introduced at § 7.2.3 (p. 115):

- **User generated content** - most of the content is created by the community. Whether the documentation artifacts in the wiki or the learning path knowledge-base, the information always originates from the users and can be evolved by them.
- **Human-machine synergy** - Automation of tasks such as collecting tags from the artifacts during browsing, presenting similar learning paths to the one just captured or providing recommendations, spares the user from collecting this information on her own.
- **Increasing returns with scale** - as more learners rate existing learning paths, the search tool brings up the most used and better rated results, contributing with higher valued information to the user.
- **Emergent knowledge** - the system offers recommendations for guidance through the artifacts, based on existing learning paths in the knowledge-base, inferring new knowledge from the existing data. Of course, the recommendation heuristic can be extended to improve the results and to more suitably provide effective recommendations. Nevertheless, the basis is there.

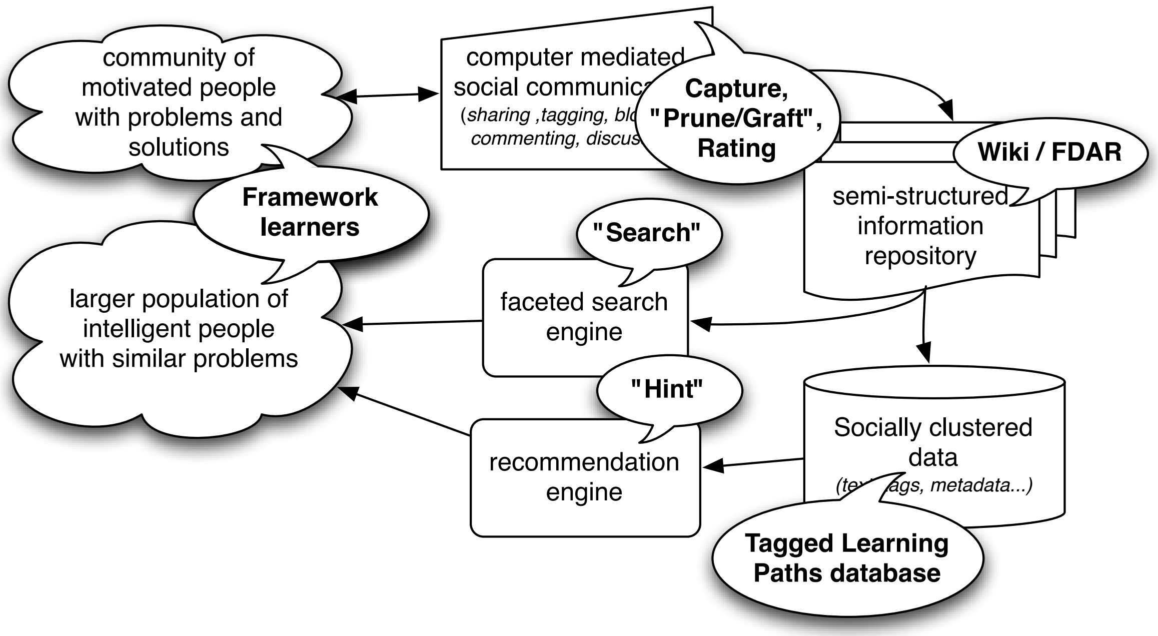

community of motivated people with problems and solutions
computer mediated social communic
(sharing ,tagging, blo commenting, discus
Capture, "Prune/Graft", Rating
Wiki / FDAR
semi-structured information repository
Framework learners
"Search"
larger population of intelligent people with similar problems
faceted search engine
"Hint"
recommendation engine
Socially clustered data
(te ags, metadata...)
Tagged Learning Paths database

Figure 7.12: The DRIVER toolset as a Collective Knowledge System.

In short, the proposed collaborative environment provides an extensible scaffold for building and extending a collective knowledge acquisition system, where the community of learners can share their insight without too much effort and benefit from its collective wisdom.

7.5 Other related tools

During the exploratory phase of development of the tools presented, the author was poised to find a suitable environment to develop the collaborative approach. Several tools emerged from the literature and state-of-the-art review, already focused in chapters 2 (p. 17), 3 (p. 37) and 4 (p. 53). From those, a couple stood out as strong candidates for the realisation of the collaborative approach. A brief explanation of the pros and cons of each and the reasons behind the choice for the wiki-based solution are detailed next.

7.5.1 Environment candidates

Evaluating existing environments suitable for the development of the collaborative approach and that would allow coverage of the requirements stated in § 7.3.5 (p. 122), was conducted based on three premisses: (i) it should be an (popular, commonly used) IDE, (ii) it should have a clear and viable documentation artifacts handling[21] infra-structure and (iii) it should already have work/task/team awareness features.

Three environments stood out: Microsoft Visual Studio [Cord], Rational Team Concert [RTC] and Eclipse [Ecl11], mostly due to their extensible nature and pluggable architecture. A deeper evaluation was then made of each candidate, assessing its suitability for the task at hand.

Microsoft Visual Studio/Team Foundation

Visual Studio[22] is one of the most popular IDEs currently in the development industry and it allows extension of its environment through *add-ins*. It provides a SDK[23] that allows the creation, development and deployment of these extensions for any Visual Studio user. If fitted[24] with the Team Foundation Server, it provides team awareness features. It has notions of team members, working items, artifacts, processes and relationships between them.

[21] CRUD-like (Create, Read, Update, Delete) features.
[22] At the time of the evaluation, the current Visual Studio release was 2008. Presently, Visual Studio 2010 as been released, but there are no relevant new features that hinder or threaten these statements.
[23] Software Development Kit.
[24] Depending on the bundle purchased.

IBM Rational Team Concert/Jazz

Rational Team Concert is IBM's excellence IDE. Its a team-aware software development platform that integrates work item tracking, builds, source control, and agile planning. It is built upon Jazz, a scalable, extensible team-collaboration platform that integrates tasks across the software lifecycle. The platform also provides useful building blocks and frameworks that facilitate the development of new products and tools. At its core is the Jazz Team Server that provides a variety of services (through RESTful web services) and allows extensions for new tools to integrate their own services. Rational Team Concert has built-in integrations with collaboration tools such as IBM Sametime® instant messaging and IBM Connections® social software.

Eclipse/MyLyn

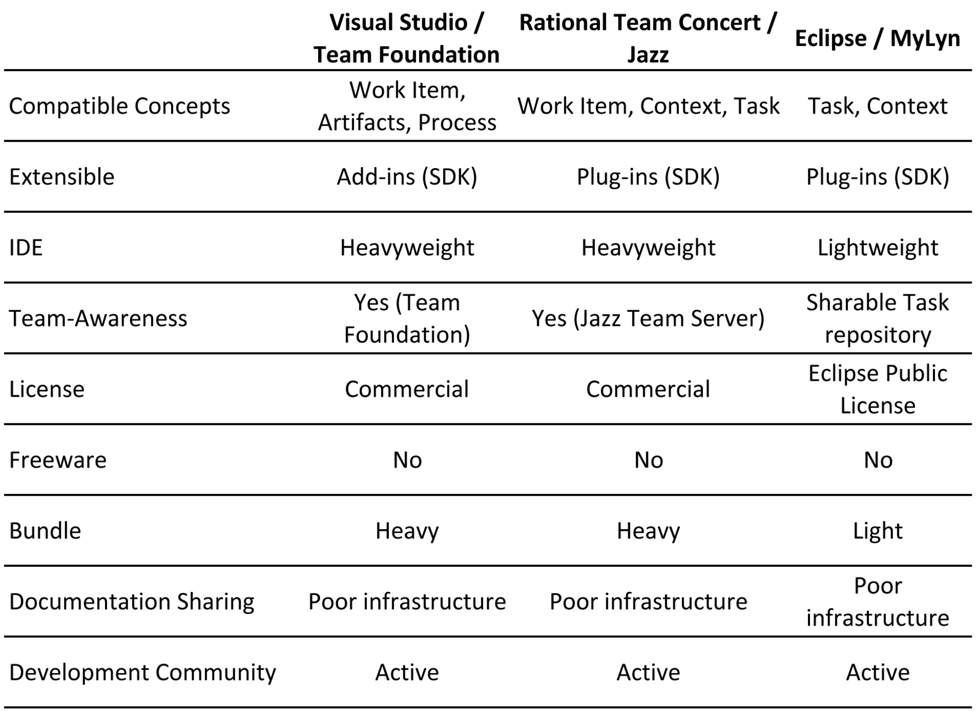
Eclipse is, similarly, one of the most used IDE's, mainly because it is freeware, open-source and has a *plug-in*-like architecture. It comes with MyLyn, a task-aware plug-in (now bundled within the IDE) that monitors and enhances the development process through the introduction of task contexts. Each task has a context. This context is composed of all the artifacts (documents) the user has opened and/or edited. It emphasises the relevance of the most edited and read[25], visually *declutering*[26] the development environment, thus focusing the developer on the important items.

7.5.2 Evaluation

All candidate environments were then evaluated for its suitability and viability to develop and validate the collaborative approach. A list of the pros and cons of each can be seen in Table 7.13.

Overall, the Eclipse/MyLyn option seemed a better candidate for a more simple, straightforward and quick implementation of the collaborative approach. Despite having a strong potential, both Visual Studio and Rational Team Concert were a commercial, heavyweight IDEs that would require additional effort and cost to setup for development and eventual validation and deployment of a solution. Eclipse was freeware, lightweight and easier to setup and, regarding the MyLyn plug-in, having a more active and focused community of developers. Therefore, the author selected this candidate as the environment for developing his approach.

[25] calculating what is called *degree-of-interest (DOI) [KM05]*.
[26] hiding non-relevant items.



	Visual Studio / Team Foundation	Rational Team Concert / Jazz	Eclipse / MyLyn
Compatible Concepts	Work Item, Artifacts, Process	Work Item, Context, Task	Task, Context
Extensible	Add-ins (SDK)	Plug-ins (SDK)	Plug-ins (SDK)
IDE	Heavyweight	Heavyweight	Lightweight
Team-Awareness	Yes (Team Foundation)	Yes (Jazz Team Server)	Sharable Task repository
License	Commercial	Commercial	Eclipse Public License
Freeware	No	No	No
Bundle	Heavy	Heavy	Light
Documentation Sharing	Poor infrastructure	Poor infrastructure	Poor infrastructure
Development Community	Active	Active	Active

Figure 7.13: Candidate environments comparison table.

Spike

After choosing to go for the Eclipse/MyLyn environment, a *spike* solution was attempted. Usually, in software development, to ascertain if a specific set of tools covers all the requirements of a system, a quick prototype[27] is developed with the sole purpose of having *proof of concept*, that is, a partial solution that covers all the technologies used and that checks if they can work together to reach all the required goals. This is called a *spike*[28].

The main idea of the prototype was to try to support the learning cycle, as simply as possible, using the concepts provided by the MyLyn component.

The Mylyn architecture defines the concept of *Task* and *Context*. A Task is an activity that a developer does regarding her work. A Context relates to a Task as the state of the development environment during the execution of the task. This context includes which files are opened, which artifacts were consulted, what changes were made to the system. MyLyn uses these concepts mainly to allow "context-changing", that is, the user can change from one task to another without loosing its context, being able to resume the work as it was left.

The prototype would extend these concepts so that the Task would become a *Learning Task* where, besides harnessing all the knowledge the Context already had, there would be

[27] In this context, also called *vertical prototyping*.
[28] This is a short name for *stalagmite*, serving as an allegory to symbolise a very focused, direct and protruded solution.

a special attention for artifacts relating to framework understanding (overviews, recipes, cookbooks, patterns), present within the current project. A *Learning Context* would then be pruned by the user before storage to filter out non-relevant information or artifacts that might have been used but revealed themselves worthless for the learning process. During the execution of the Learning Task, similar Learning Contexts would be recommended that might resemble the current one. This resemblance would be evaluated by looking at both contextual information (open artifacts, edited files) and "social" information (tags, ranking, etc). The user would be able to browse through recommended contexts and rank those that most helped her.

Appraisal

This prototype was developed during the course of a few weeks, and it rendered the following conclusions, regarding the Eclipse/MyLyn candidate:

- **Weak artifact representation.** The information regarding the artifacts within a task or context was shallow. It merely represented a local resource and not a GUID for a document. It would require a considerable amount of effort to improve that representation to support the learning cycle needs.

- **Context API limited.** The Context API that would allow manipulation of Context was built in a way that it could only manipulate the currently active context. This would jeopardise the ability to aggregate and manipulate a repository of contexts that would eventually be extended to contain learning knowledge. The developers community was questioned about this and it was regarded as a low priority feature, therefore the effort to change it would have to be on our side. This added a great deal of complexity to the prototype because this change would crosscut through the entire architecture of MyLyn, forcing it to be almost completely redesigned.

These hindrances led to reconsider the Eclipse/MyLyn option as a suitable candidate for the development of the collaborative approach. For all other caveats found, a suitable workaround was devised, nevertheless, it would always be found lacking. A new implementation approach was needed.

7.5.3 Comparing with the candidates

When comparing the proposed collaborative approach with presumed candidates, the author states the following:

- ***regarding Eclipse/MyLyn...*** The task context concept has a large potential for aiding learners, nevertheless it still lacks an effective, faceted, semantic context search tool. The user can import contexts from a repository with all the artifacts references and its *degree-of-interest* filtering, but there is no relation between tasks so that the learner may know what context(s) to import. The context is a great candidate for aggregating knowledge, but it still lacks learning semantics. The purpose of the *spike* was to introduce this feature into the context concept.

- ***regarding Rational Team Concert/Jazz...*** RTC/Jazz is an advanced version of Eclipse/Mylyn, scaled to larger systems, larger teams and covering the whole development process. It's sort of an enterprise version and, as such, it has all the pros and cons of the above candidate, plus the cons of being a commercial, heavyweight solution.
- ***regarding Visual Studio/Team Foundation...*** Team Foundation has a query-based search engine of work items, but these are not related through relevant semantics regarding learning usefulness. Their relationships are through process or team member ownership. The community contribution is through scattered means of communication that miss a focused aggregation point, thus unsuitable to help learners.

All candidates are suitable platforms for integrating the learning cycle support, but it would still require a considerable amount of effort to seamlessly provide the users with this support. The author's decision to choose the presented setting had simply to do with proving his hypotheses in an time-effective, focused way.

7.5.4 Moving to the web

At this point, the feeling was more to start from scratch and develop the minimum required infra-structure to support the learning cycle. The integration into a IDE could come later. The main goal was to provide a system that would allow validation of the devised approach. The focus then turned to the documentation artifacts that would serve as basis for the learning process.

Nevertheless, there was still the need to provide the learners with a familiar, suitable environment for their learning process. The choice was obvious. The most used application, nowadays, is the web browser. So a web-based system would be most suited to present familiar content to the learners. The next step was then to decide which kind of system would be most appropriate. It should be easy to use, lightweight and extensible.

Probing web solutions

The *wiki* solution immediately popped to mind. As already described in § 7.4.1 (p. 123), a wiki would provide a suitable environment from where to build the prototype that, eventually, evolved to the toolset presented in this chapter. Nevertheless, other possible solutions where screened for better suitability, if none other, to support the chosen wiki-based solution. Thus, the basic platform should support shared, editable documentation artifacts. Commercial products like Alfresco [Alf] or Confluence [Con] were quickly discarded, not only for their commercial (non-free) nature, but also for being heavyweight[29]. The focus then turned to platforms that were similar to wikis or that had (or integrated with) an embedded wiki. Possible candidates were Redmine [Red] and open-source CMS[30] solution like Drupal [Dru] or Joomla [Joo]:

- *Redmine.* Is a flexible project management web application[31]. Written using the Ruby on Rails framework, it is cross-platform and cross-database, having a bundled wiki and documentation management. It presented itself as a possible solution, not only for filling the pre-defined requisites, but because it had been previously worked with[32]. The reasons for not adopting Redmine where predominantly two: (1) a poorly structured wiki (no concept of sections) and (2) the wiki was decoupled from the documentation infrastructure.

- *Drupal/Joomla.* Both applications are PHP-based CMSs, open-source and extensible. Although presented in lightweight bundles, its basic installation still had to be stripped down of some unnecessary components. Although extensible, both Drupal and Joomla required some initial investment to learn how to use, and, although the extension architecture was somewhat similar to dokuwiki's[33] their content structure was too strict[34] to provide document content flexibility.

As such, the wiki-based solution became consolidated as a more quick and effective means to develop and deliver the proposed learning environment. A simple wiki should provide enough support and structure to harbour the documentation artifacts and it should be extensible so that only the minimum necessary features could be added on.

[29] The basic bundle would bring a reasonable amount of clogging, non-relevant features.
[30] Content Management System.
[31] Others similar applications exist but were discarded for not having documentation, or wiki, support.
[32] Supervised graduate students had developed extensions for Redmine.
[33] More Joomla, than Drupal.
[34] Without discussing the differences between a CMS and a wiki, that has been causing some stir on the blog community.

Proof-of-concept: from clickstreams to learning paths

As already explained in § 7.4.1 (p. 123), the *dokuwiki* was selected to provide the basis for the development of the DRIVER platform. From all the familiar features this wiki engine provided, it was still necessary to ascertain if it could implement the learning cycle § 7.3.3 (p. 119).

The concept of learning path can be compared to what is known as *clickstream*, a web analytics metric. According to WAA[35], Web Analytics is the measurement, collection, analysis and reporting of Internet data for the purposes of understanding and optimizing Web usage. Clickstream tracks by which order the visitor of a site navigated through its contents[36]. A learning path can be seen as constrained form of clickstream, where only the relevant clicks (navigating between documents and *liking* sections) are captured.

To provide this *proof-of-concept*, a dokuwiki plug-in called *directions*[37] was developed. It provides the wiki user with the whole wiki navigational graph, and shows, for the page the user is currently browsing, where the users mostly navigate to and from. This was the starting point for developing the DRIVER platform - a single developer project effort, under development for about 7 months. The two development technologies with the most impact were *PHP* and *Javascript*, generating a total of 9.3 KLOCs[38].

7.6 Summary

This chapter presented DRIVER, a collaborative environment that supports the framework learning process. Not only it relies on an easy, sharing, lightweight, editable platform (*wiki*), that provides documentation artifacts about the framework, but promotes knowledge acquisition by enabling capture, storage, sharing, rating and recommendation of learning knowledge.

This learning knowledge takes the form of *learning paths*. These show how other learners tackled with similar problems by presenting which documentation artifacts they went through and by which order. The presented toolset supports this kind of learning process through a series of plug-ins that seamlessly integrate the documentation infra-structure. These learning paths are stored and shared by the community of learners, who rate the level of usefulness this learning data has, allowing the information to mature and improve its quality and applicability throughout the community.

[35] Web Analytics Association
[36] The click stream is the sequence of mouse clicks the user performed on the contents of the site, but usually, only the navigation (link clicks) is recorded.
[37] Available at `http://www.dokuwiki.org/plugin:directions`
[38] *LOC* meaning *Lines of Code*

The presented approach tackles with the intrusiveness the learning process can have when directly asking for help, usually resulting in disregarding the request, exhibiting non-availability and, progressively forgetting useful learning knowledge (the issue of loosing tacit knowledge). Intrusiveness is thus mitigated by resorting to a shared and maturing knowledge-base of learning knowledge. Not only the *asking* learner isn't interrupting her colleagues, but the *knowledgeable* learner can, without disrupting his normal functions, capture and store the grafted learning path, as part of the common procedure of learning. This also contributes in diminishing the loss of tacit knowledge.

Part III

Validation & Conclusions

Chapter 8

Academic quasi-experiment

This chapter details a quasi-experiment conducted within a controlled experimental environment using the DRIVER platform. This study was intended to provide evidence that the presented collaborative approach helps novices and experts, improving their framework learning experience. The experiment took groups of similar MSc students and measured their performance, effectiveness and framework knowledge intake, while developing a set of tasks using a new framework. In parallel, a set of students already knew the framework, as to study the process of re-acquiring dormant framework knowledge. The final results support the hypothesis that the collaborative approach helps improving framework learning, specially for novices.

8.1 Experiment design

The use of empirical studies with students (*ESWS*) in software engineering helps researchers gain insight into new or existing techniques and methods. However, due mainly to concerns of external validity, these studies are often viewed skeptically by researchers and practitioners. Empirical studies with professionals, which are widely accepted by the above-mentioned also suffer from similar generalizability problems. Therefore, just like any other empirical studies, *ESWS*s can be valuable to the industrial and research communities if they are conducted in an adequate way, address appropriate goals, do not overstate the generalizability of the results and take into account threats to internal and external validity [CJMS10]. As *ESWS*s are often used to obtain preliminary evidence in support of or against research hypothesis, this experiment was designed as such.

The independent experimental validation of claims is not as common in Software Engineering as in other, more matured sciences. As such, the (quasi-)experiment here detailed was designed as an *experimental package* (available at [ESS11]), to be performed in different locations, and by different researchers, in order to enhance the ability to integrate the results obtained and allow further meta-analysis on them.

8.1.1 Subjects

The experiment subjects were 24 MSc students from the Integrated Master in Informatics and Computing Engineering, lectured at the University of Porto, Faculty of Engineering. They were part of a 4th year class, attending an optional course on "Architecture of Software Systems". Its syllabus deals strongly with frameworks and patterns, therefore it was more than suitable to integrate this experiment into their course work.

Group formation

The subjects were divided into 4 groups, each with its own purpose.

- **Baseline (*BL*).** This group established the baseline for the experiment, serving as the control group. Its subjects used the framework with no aids but the documentation. The experience protocol for this group was Pre-Questionnaire A § 8.2.2 (p. 154) , Treatment A § 8.2.3 (p. 154), Tasks § 8.2.4 (p. 155) and Post-Questionaire § 8.2.5 (p. 159).

- **Experimental Group 1 (*EG1*).** This group used the framework having, besides the documentation, the patterns (presented in Chapter 6, p. 81) as an aid. The purpose was to compare results with the baseline group and provide evidence for the usefulness of the patterns. The experience protocol for this group was Pre-Questionnaire A § 8.2.2 (p. 154), Treatment B § 8.2.3 (p. 154), Tasks § 8.2.4 (p. 155) and Post-Questionnaire § 8.2.5 (p. 159).

- **Experimental Group 2 (*EG2*).** This group used the framework having the DRIVER platform (documentation, patterns and wiki plug-ins) § 7.4 (p. 123) as aid. The purpose was to compare results with the baseline group and provide evidence that the proposed collaborative environment improves framework understanding. The experience protocol for this group was Pre-Questionnaire A § 8.2.2 (p. 154), Treatment C § 8.2.3 (p. 154), Tasks § 8.2.4 (p. 155) and Post-Questionnaire § 8.2.5 (p. 159).

- **Experts (*EX*).** This group used the framework having the DRIVER platform (documentation, patterns and wiki plug-ins) as aid. The purpose was to provide evidence that the proposed collaborative environment helps *Experts*[1] increase their knowledge of the framework. The experience protocol for this group was Pre-Questionnaire B § 8.2.2 (p. 154), Treatment C § 8.2.3 (p. 154), Tasks § 8.2.4 (p. 155) and Post-Questionnaire § 8.2.5 (p. 159).

Pre-experiment evaluation

For an experiment of this kind, it is important to assure that the subjects are similar and that their base skills don't pose a significant threat to the validity of the results. Therefore, they were scrutinised based on their academic track, by analysing their grades on a selected subset of courses. These courses were deemed relevant to the outcome of the experiment, namely: (i) Programming Fundamentals, (ii) Programming, (iii) Algorithms and Data Structures, (iv) Algorithm Design and Analysis, (v) Software Engineering, (vi) Software Development Laboratory, and (vii) Information Systems. Their grades can be found in Appendix A (p. 189), Table A.1. An independent samples t-test was conducted to compare the average students' grades (shown in Table 8.1) between the baseline and other experimental groups (*EG1*, *EG2*, *EX*).

As shown in Table 8.2, there was **no significant** difference in the scores for the Experimental Group 1 ($M = 16.93$, $SD = 1.35$) and Baseline ($M = 16.84$, $SD = 0.93$) conditions; $\rho = 0.902$, within a 95% confidence interval.

As shown in Table 8.3, there was **no significant** difference in the scores for the Experimental Group 2 ($M = 16.64$, $SD = 1.49$) and Baseline ($M = 16.84$, $SD = 0.93$) conditions; $\rho = 0.802$, within a 95% confidence interval.

As shown in Table 8.4, there was **no significant** difference in the scores for the Experts ($M = 16.73$, $SD = 1.02$) and Baseline ($M = 16.84$, $SD = 0.93$) conditions; $\rho = 0.839$, within a 95% confidence interval.

Group	N	Mean	Std. Deviation	Std. Error Mean
BL	6	16.838	0.9261	0.3780
EG1	4	16.928	1.3527	0.6763
EG2	4	16.943	1.4869	0.7435
EX	9	16.730	1.0213	0.3404

Table 8.1: Student grades group statistics.

[1] In this context, *Experts* stands for subjects that have already used the framework for a significant period of time (over 3 months), performing instantiation and evolution activities.

	F	Sig.	t	df	Sig. (2-tailed)
Eq. Var. Assumed	0.269	0.618	-0.13	8.000	0.902
Eq. Var. Not Assumed			-0.12	4.882	0.912

Table 8.2: Baseline vs. Experimental Group 1 Independent Samples Test. The first two columns are the Levene's Test for Equality of Variances, showing a significance greater than 0.05 (0.618). The other three columns are the t-test for Equality of Means. Since we can assume equal variances, the 2-tailed value of 0.902 allow us to conclude that there is no statistically significant difference between the two conditions.

	F	Sig.	t	df	Sig. (2-tailed)
Eq. Var. Assumed	0.607	0.458	0.259	8.000	0.802
Eq. Var. Not Assumed			0.234	4.569	0.825

Table 8.3: Baseline vs. Experimental Group 2 Independent Samples Test. The first two columns are the Levene's Test for Equality of Variances, showing a significance greater than 0.05 (0.458). The other three columns are the t-test for Equality of Means. Since we can assume equal variances, the 2-tailed value of 0.802 allow us to conclude that there is no statistically significant difference between the two conditions.

8.1.2 Framework selection

Choosing a framework to use in the experiment was an issue. One of the main concerns was finding a framework that would suit all the experimental groups. The experiment needed a framework that was unknown to most groups (*BL*, *EG1*, *EG2*) but known to the *Experts* (*EX*) group.

At first, a survey was devised to find out which frameworks, from a pre-selected subset, were known to the students. Afterwards, it was only a matter of grouping the students accordingly to the experiment's needs. Nevertheless, this survey would prove useless if the subset of frameworks rendered unsuitable results (inability to form the groups), not to mention the poor control over the reliability of the answers.

	F	Sig.	t	df	Sig. (2-tailed)
Eq. Var. Assumed	0.050	0.827	0.208	13.00	0.839
Eq. Var. Not Assumed			0.212	11.62	0.836

Table 8.4: Baseline vs. Experts Independent Samples Test. The first two columns are the Levene's Test for Equality of Variances, showing a significance greater than 0.05 (0.827). The other three columns are the t-test for Equality of Means. Since we can assume equal variances, the 2-tailed value of 0.839 allow us to conclude that there is no statistically significant difference between the two conditions.

At this point, a solution presented itself. A PhD colleague had been developing a new framework, called *OGHMA* [Fer11]. *OGHMA* is an object-oriented framework targeted to the development of information systems, on an industrial level, whose structural requirements can be best described as *incomplete by design*. It relies on the Adaptive Object-Model meta-architectural pattern [YBJ01] to allow run-time domain evolution by the user.

All of the students recruited for this experiment had attended the Software Development Laboratory course, although distributed by several distinct classes. One of those classes used the *OGHMA* framework. Coincidently, some of those students chose the optional "Architecture of Software Systems" course, which enabled the subject's pool to have students with *OGHMA* experience and others with no prior knowledge of the framework [2]. These conditions prompted *OGHMA* as the primary candidate for the selection. Choosing some other framework would imply training the *Experts* group *a priori*, whereas choosing *OGHMA* would enable a faster setup without jeopardising the experiment goals.

In future experiments, a previous, more extensive survey must be made to the subjects in order to find a suitable framework that can cope with all the constraints and where training of the *Experts* group might be required.

8.2 Experiment description

This section describes the experiment protocol and phases as depicted in Figure 8.1. After divided into groups, the students were submitted to a pre-experiment questionnaire to establish their initial state. Then, each group undertook a treatment phase to condition their experiment environment accordingly and, after going through a series of tasks, a post-experiment questionnaire was conducted to collect results.

8.2.1 Environment

According to [CJMS10], regarding ESWSs,

> *The study setting must be appropriate relative to its goals, the skills required and the activities under study.*

Considering this requirement, this section describes the experiment environment.

[2] Some students had heard of *OGHMA* through their colleagues, but had never used it or even knew its application domain.

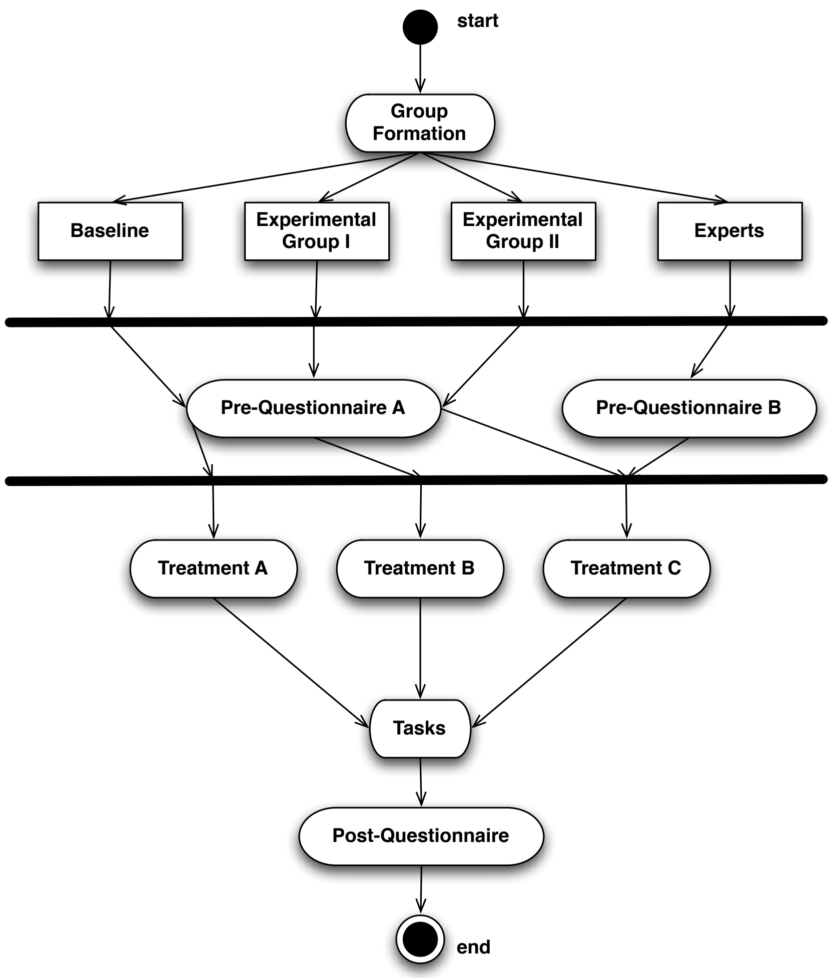


Figure 8.1: Experiment protocol and phases.

Setting

The experiment was conducted on a familiar setting to the students, as an attempt to minimise the external environmental factors that might threaten the validity of the results. It took place in laboratory classrooms, the same ones used by the students to attend classes or develop their course work.

To enforce group and collaborative work [CJMS10], the students were grouped into pairs with colleagues from their own experimental group and placed in two separate rooms (*BL* and *EG1* groups in one room and *EG2* and *EX* groups in another room). This was done to better monitor the experiment progress and to more easily conduct some phase activities, e.g. video projection (See Treatment C, section 8.2.3, p. 154). Each pair had a workstation.

They had limited internet access in order to minimise distractions (instant messaging, e-mail, etc.) and to control experimental variables. Nevertheless they had access to the necessary resources to conduct the experiment successfully. These resources included: (1) an experiment tracker wiki, (2) a documentation wiki and (3) an *OGHMA* Visual Studio Solution package.

Experiment tracker wiki

To monitor and control the experiment progress, a wiki[3] was provided for the students. This wiki contained instructions on how to start the experiment and description of the consecutive tasks to be performed, its requirements and goals. This wiki was also enhanced with a *tracker* plug-in that enabled a non-intrusive monitoring of the duration each pair took to complete each task. The students were warned that they should not proceed to the next task before finishing the current one, as that would mark the previous task as done. Not complying with this restriction would lead to invalid time readings.

Documentation wiki

To learn about the *OGHMA* framework, a documentation wiki was provided with enough contents to enable the effective coverage of the requirements presented by each task. This wiki was, in every way, an instantiation of the DRIVER platform described in § 7.4 (p. 123), except for the availability of the patterns and the learning cycle supporting plug-ins. During the Treatments phase § 8.2.3 (p. 154) each group would be given access to these resources according their experiment research goals.

OGHMA Visual Studio Solution

The *OGHMA* framework was develop in C#, and optimised for use on Microsoft's Visual Studio IDE. Therefore, a *Solution* package was provided to the students (and referenced by the documentation), hopefully, serving as a more quick and easy platform for development. It was assumed that all students were proficient using the IDE, as it was included in their academic track. Nevertheless, this aspect was screened through a pre-experiment questionnaire item (BG1.8).

8.2.2 Pre-questionnaires

The first phase of the experiment was to hand out a questionnaire to the students. The questionnaires were designed using a Likert scale [Lik32]. This psychometric bipolar scaling method contains a set of Likert items, or statements, which the respondent is asked to evaluate according to any kind of subjective or objective criteria, thus measuring

[3] a dokuwiki, the same wiki engine used to implement the DRIVER platform § 7.4.1 (p. 123).

either negative or positive response to the statement. For all the questionnaires in this experiment (both pre- and post-), the Likert items had a five-point format: (1) strongly disagree, (2) somewhat disagree, (3) neither agree nor disagree, (4) somewhat agree, and (5) strongly agree.

Pre-questionnaire A

The pre-experiment questionnaire A was used to ascertain the students background and general profile in order to screen out possible differences amongst the students regarding their basic skills. It also served to confirm their acquaintance with the *OGHMA* framework. Students of groups *Baseline (BL)*, *Experimental Group 1 (EG1)* and *Experimental Group 2 (EG2)* were submitted to this questionnaire (see Appendix B, p. 191), whose answers are detailed in Appendix C (p. 193), and further analysed in § 8.3 (p. 159).

Pre-questionnaire B

The pre-experiment questionnaire B extended pre-experiment questionnaire A to include items used to ascertain what knowledge the subjects had about the *OGHMA framework*, at the start of the experiment. Students of the *Experts (EX)* group were submitted to this questionnaire (see Appendix D, p. 195), whose answers are detailed in Appendix E (p. 199), and further analysed in § 8.3.8 (p. 175).

8.2.3 Treatments

After the pre-questionnaires phase, the students were subject to a treatment phase, where each group was introduced to their own experiment environment. This is where the groups actually diverge regarding their experiment research goals. Each treatment is described next.

Treatment A

This treatment introduced the experiment environment as described in § 8.2.1 (p. 151). The *Baseline (BL)* group undertook this treatment.

Treatment B

This treatment extended treatment A by allowing the students to have access to the patterns (Chapter 6, p. 81) and to spent time knowing about the patterns before advancing to the next phase. The *Experimental Group 1 (EG1)* undertook this treatment.

Treatment C

This treatment extended treatment B by providing the experiment environment with the whole DRIVER (Chapter 7, p. 109) platform. The DRIVER knowledge base already had a few learning paths captured in a previous *trial run* session where a framework expert performed the same experiment protocol. This expert had previous knowledge of the framework, but no acquaintance with the documentation wiki. The students were shown a demonstration video with a quick tutorial on how to use the plug-ins and their purpose, using a different case scenario. This way the students weren't biased by any clues on what documentation to look for in order to perform the experiment tasks. Both *Experimental Group 2 (EG2)* and *Experts (EX)* group undertook this treatment.

8.2.4 Tasks

At this point, all the groups were ready to start executing the tasks that would led them to use the framework. It wasn't disclosed how many tasks were there, merely the goal to be effective and as less time-consuming as possible.

The tasks were mainly focused on assessing how efficiently the students could incrementally build an information system within 4 iterations. These tasks had already been used in another experience [Fer11] regarding the *OGHMA* framework, so they were adapted to fit this experiment research goals. The description of each iteration is presented next.

Iteration 0

This iteration served only to contextualise the students with the problem domain where they were going to work and to explain how to conduct their work throughout the iterations. It also pointed out how to verify the effectiveness of the outcome of each iteration so that they might confidently proceed to the next one.

The following text was presented to the students:

> *Welcome to* **ASSO**CIATION [4] *Software! With your help, our company will most certainly thrive and reach higher grounds. Our customers are academic institutions and scholarly entrepreneurs.*
>
> *Your first assignment is to implement an information system for managing scientific conferences. After a careful requirements analysis, the engineers have concluded that the system should be implemented in several iterations. Having already planned all of the iterations, you'll find a detailed UML class diagram for each one, as you go along.*

[4] *ASSOCIATION* was a fake software company name made from the course acronym: *"ASSO"*

> *Due to the fact that the client wants to validate your system at the end of each iteration, you'll have to deliver a releasable product for each iteration. Each release should have a working Graphical User Interface and Persistency Engine. The user should be able to create, read, update and delete the modelled concepts.*
>
> *In order to achieve these goals, your are going to use the OGHMA framework. You will be given some documentation to help you. You can download the OGHMA Visual Studio Solution here.*
>
> *You may start whenever you feel ready.*
>
> *Good luck!*

Iteration 1

The first iteration was designed to yield a very simple system. Only a single screen would be needed to view and edit the information. There was no polymorphism, shared aggregations or any type of conditional rules and the whole system could be roughly stored in two database tables. The purpose was merely to make first contact with the framework with no excessive task complexity.

The following text and diagram depicted in Figure 8.2 were presented to the students:

> *In this iteration, you'll implement the basic concepts of a scientific conference. A conference is typically related to a specific* ***Scientific Area*** *(e.g., Computer Science or Software Engineering), but it is not uncommon to find conferences related to several areas.*
>
> *Each conference has several editions, normally once per year (e.g.* Pattern Languages Of Programs 2008*). Due to several factors, it is also common for conferences to be co-located with others. For example, the 2008 editions of "*Pattern Languages Of Programs*" and "*Object-Oriented Programming, Systems, Languages & Applications*" were co-located.*
>
> *In the end of this iteration, the user should be able to manage* ***Conferences*** *and* ***Editions****. Model elements tagged with the stereotype* entrypoint *represent main entry points to the system (e.g., the user should be able to invoke a list of Scientific Areas from the application's main menu or similar mechanism).*

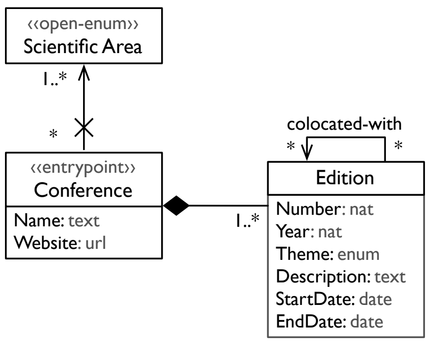


Figure 8.2: Iteration 1

Iteration 2

The second iteration was designed to make the students dig deeper into the framework. There was the need for more screens (due to indexation) and the number of database tables could grew from two to four, but with minimal modification to the existing artifacts.

The following text and diagram depicted in Figure 8.3 were presented to the students:

> *In this iteration, you'll extend your system with two extra features. The first one – **Indexes** – classifies conferences according to a rating system per year. For example, the index* "ISI Web of Knowledge" *rates thousands of conferences every year. The rating is given to a particular edition of a conference, so ISI could rate* "Pattern Languages of Programs" *as an A in 2015, and as a B in 2016. The second one - **Sessions** - allows the user to manage the program (contents) of a conference. There are two types of sessions: (a) **Presentations**, and (b) **Poster Sessions**. For example, the 2008 edition of* "Object-Oriented Programming, Systems, Languages & Applications" *had one (1) poster session, and twelve (12) presentations.*

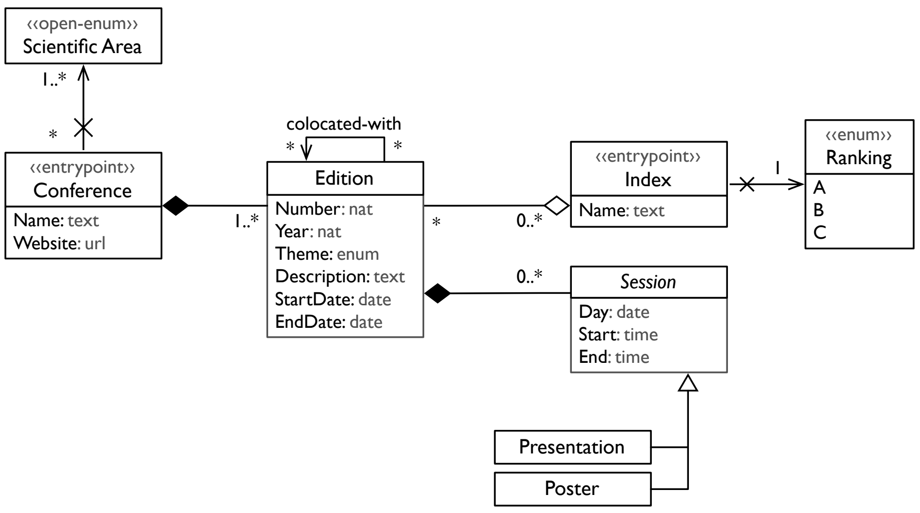


Figure 8.3: Iteration 2

Iteration 3

The third iteration merely increased the complexity of the system, requiring more elaborate user-interaction and conditional rules, involving shared many-to-many relationships and relationship properties. Again the students would have to go deeper into the framework in order to cope with these requirements.

The following text and diagram depicted in Figure 8.4 were presented to the students:

> *In this iteration, you'll provide your system by adding some remaining core concepts of scientific conferences. People that participate in conferences as authors have to submit at least one paper, either alone or with colleagues. In addition, they may be* ***Chairs****,* ***Organisers*** *or simple* ***Participants****. Different editions normally have different chairs and organisers.*

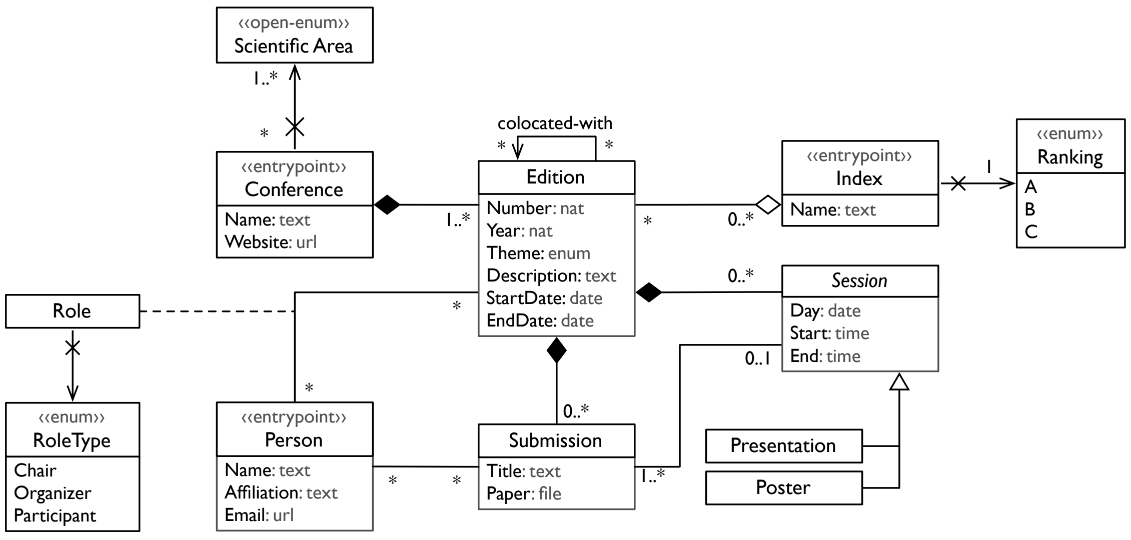


Figure 8.4: Iteration 3

Iteration 4

This final iteration posed a new challenge: framework evolution. The students had no *customisation* points to solve the proposed requirements. Therefore, they would have to go into the framework internal code and extend it to provide this new configuration ability. The goal was to put the students in direct contact with the framework code. Here is how the task was presented:

> *In this iteration, for reasons of practicability, the client has requested that whenever a paper is submitted to a conference, an* ***email*** *should be sent for the* ***Chairs*** *with the* ***Submission*** *and* ***Authors*** *relevant information.*

8.2.5 Post-questionnaire

At the end of the experiment, the questionnaire in Appendix F (p. 201) was handed out to the students. The answers from all the participants are detailed in Appendix G (p. 205), and further analysed in § 8.3.6 (p. 169).

8.3 Data Analysis

As mentioned in § 8.2.2 (p. 153), the subjects were given a pre-experiment questionnaire to screen out possible background and basic skills deviations. During the experiment, the task completion time was recorded and afterwards, a post-experiment questionnaire collected further data. This section presents a detailed analysis of the collected data in order to provide evidence of the validity of the assumptions presented by this dissertation.

Firstly, it will be shown that all groups have no significant background deviations and that acquaintance of the *OGHMA* framework is correctly assumed. Secondly, the analysis will focus on comparing results between the *Baseline (BL)* and *Experimental Groups* 1 (*EG1*) and 2 (*EG2*). Finally, the *Experts (EX)* group will be focused on.

8.3.1 Statistical relevance

To provide statistical relevance in the analysis of the questionnaires items, the results are interpreted as described next. Let the null hypothesis be denoted as H_0, the alternative hypothesis as H_1, the baseline group as G_b, the experimental group as G_e[5], and ρ the probability estimator of wrongly rejecting the null hypothesis. Then, the alternative hypothesis are either: (i) $H_1 : G_e \neq G_b$, the experimental group differs from the baseline, (ii) $H_1 : G_e < G_b$, the measure in the experimental group is lower than the baseline, or (iii) $H_1 : G_e > G_b$, the measure in the experimental group is greater than the baseline. The outcomes of the two treatments were compared for every answer using the non-parametric, two-sample, rank-sum Wilcoxon-Mann-Whitney [HW99] test, with $n_1 = 6$ and $n_2 = 4$[6]. The significance level for all tests was set to 5%, so probability values of $\rho \leq 0.05$ are considered *significant*, and $\rho \leq 0.01$ considered *highly significant*. The corresponding alternative hypothesis are further detailed for each question, and a summary of the base statistics and corresponding test values can be found in Appendices C (p. 193) and G (p. 205).

[5] The experimental group can be either *EG1* or *EG2* depending on what group is being analysed.
[6] Both *EG1* and *EG2* have 4 subjects.

8.3.2 Background

Although an objective comparison between the background of each group was already conducted using the subjects average grades in key courses § 8.1.1 (p. 149), this section rejects any subjective difference amongst the participants with respect to their basic skills.

	EXPERIMENTAL GROUP 1			BASELINE			STATISTICS		
	12345	$\bar{x}$	σ	12345	$\bar{x}$	σ	H_1	W	ρ
BG1.1		3.25	1.26		4.00	0.63	$\neq$	16.5	0.257
BG1.2		4.00	0.82		3.50	0.55	$\neq$	28.5	0.352
BG1.3		4.50	0.58		4.83	0.41	$\neq$	18.0	0.476
BG1.4		3.50	0.58		3.33	0.82	$\neq$	32.0	0.914
BG1.5		3.75	0.50		3.83	0.75	$\neq$	21.5	0.914
BG1.6		3.00	0.82		3.00	0.00	$\neq$	22.0	1.000
BG1.7		4.25	0.50		4.50	0.55	$\neq$	19.0	0.610
BG1.8		3.00	1.41		3.83	1.60	$\neq$	17.0	0.352
BG1.9		4.00	0.00		4.33	0.52	$\neq$	18.0	0.476
BG1.10		3.25	0.96		4.17	0.98	$\neq$	16.0	0.257
BG2		4.75	0.50		4.83	0.41	$\neq$	21.0	0.914

Table 8.5: Summary of Background results between the *Baseline (BL)* and *Experimental Group 1 (EG1)*, including the values of the non-parametric significance Mann-Whitney-Wilcoxon test.

BG1.1 I have considerable experience using frameworks

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 0.257$) in the scores for the experimental group 1 ($\bar{x} = 3.25$, $\sigma = 1.26$) and baseline ($\bar{x} = 4.00$, $\sigma = 0.63$) conditions, as seen in Table 8.5. Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 0.610$) in the scores for the experimental group 2 ($\bar{x} = 3.75$, $\sigma = 0.96$) and baseline ($\bar{x} = 4.00$, $\sigma = 0.63$) conditions, as seen in Table 8.6 (p. 161). As expected, the students have a fair amount of contact with frameworks throughout their academic track, therefore they feel comfortable around frameworks.

BG1.2 I have considerable experience analyzing and specifying information systems

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 0.352$) in the scores for the experimental group 1 ($\bar{x} = 4.00$, $\sigma = 0.82$) and baseline ($\bar{x} = 3.50$, $\sigma = 0.55$) conditions, as seen in Table 8.5. Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 0.352$) in the scores for the experimental group 2 ($\bar{x} = 4.00$, $\sigma = 0.82$) and baseline ($\bar{x} = 3.50$, $\sigma = 0.55$) conditions, as seen in Table 8.6 (p. 161). Almost since the beginning of their

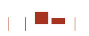
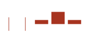
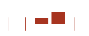
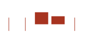
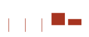
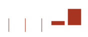
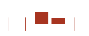
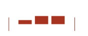
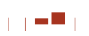
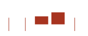
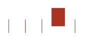
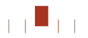
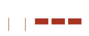
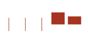
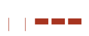
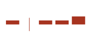
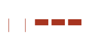
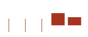
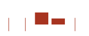
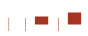
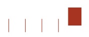
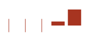
	EXPERIMENTAL GROUP 2			BASELINE			STATISTICS		
	12345	$\bar{x}$	σ	12345	$\bar{x}$	σ	H_1	W	ρ
BG1.1		3.75	0.96		4.00	0.63	$\neq$	19.5	0.610
BG1.2		4.00	0.82		3.50	0.55	$\neq$	28.5	0.352
BG1.3		4.50	0.58		4.83	0.41	$\neq$	18.0	0.476
BG1.4		3.50	0.58		3.33	0.82	$\neq$	32.0	0.914
BG1.5		3.25	0.96		3.83	0.75	$\neq$	18.0	0.476
BG1.6		3.50	1.00		3.00	0.00	$\neq$	27.0	0.257
BG1.7		4.25	0.96		4.50	0.55	$\neq$	20.5	0.762
BG1.8		4.00	0.82		3.83	1.60	$\neq$	21.0	0.914
BG1.9		4.25	0.96		4.33	0.52	$\neq$	22.0	1.000
BG1.10		3.50	0.58		4.17	0.98	$\neq$	17.0	0.352
BG2		5.00	0.00		4.83	0.41	$\neq$	31.0	0.762

Table 8.6: Summary of Background results between the *Baseline (BL)* and *Experimental Group 2 (EG2)*, including the values of the non-parametric significance Mann-Whitney-Wilcoxon test.

academic track, the students engage in the analysis and specification of information systems. At this point, all of them had gone through a course especially dedicated to that subject (namely *Information Systems*).

BG1.3 I have considerable experience with object-oriented architecture design and implementation

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 0.476$) in the scores for the experimental group 1 ($\bar{x} = 4.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 4.83$, $\sigma = 0.41$) conditions, as seen in Table 8.5 (p. 160). Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 0.476$) in the scores for the experimental group 2 ($\bar{x} = 4.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 4.83$, $\sigma = 0.41$) conditions, as seen in Table 8.6. As object-oriented development lies at the core of their academic track, it was relevant to ascertain their overall confidence in evaluating their own capability, to show that their answers were real and balanced. The high average results prove the reliability of the answers.

BG1.4. I have considerable experience with agile development methodologies

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 0.914$) in the scores for the experimental group 1 ($\bar{x} = 3.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 3.33$, $\sigma = 0.82$) conditions, as seen in Table 8.5 (p. 160). Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 0.914$) in the scores for the experimental group 2 ($\bar{x} = 3.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 3.33$, $\sigma = 0.82$) conditions, as seen in Table 8.6. This item served as an evaluation of the students' feelings towards pair-like iterative development that would characterise the experiment process. As such, the development methodology proved not to be a hindrance throughout the experiment.

BG1.5 I have considerable experience with classical development methodologies

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 0.914$) in the scores for the experimental group 1 ($\bar{x} = 3.75$, $\sigma = 0.50$) and baseline ($\bar{x} = 3.83$, $\sigma = 0.75$) conditions, as seen in Table 8.5 (p. 160). Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 0.476$) in the scores for the experimental group 2 ($\bar{x} = 3.25$, $\sigma = 0.96$) and baseline ($\bar{x} = 3.83$, $\sigma = 0.75$) conditions, as seen in Table 8.6 (p. 161). In conjunction with BG1.4 and BG1.6 this item merely served to screen out and confirm possible inconsistent answers regarding the development methodology.

BG1.6 I have considerable experience with formal development methodologies

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 1.000$) in the scores for the experimental group 1 ($\bar{x} = 3.00$, $\sigma = 0.82$) and baseline ($\bar{x} = 3.00$, $\sigma = 0.00$) conditions, as seen in Table 8.5 (p. 160). Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 0.257$) in the scores for the experimental group 2 ($\bar{x} = 3.50$, $\sigma = 1.00$) and baseline ($\bar{x} = 3.00$, $\sigma = 0.00$) conditions, as seen in Table 8.6 (p. 161). As some of the tasks relied on a formal definition of constraints and pre- and post-conditions, this item dispersed any possible aversion to that form of development.

BG1.7 I have considerable experience with UML class diagrams

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 0.610$) in the scores for the experimental group 1 ($\bar{x} = 4.25$, $\sigma = 0.50$) and baseline ($\bar{x} = 4.50$, $\sigma = 0.55$) conditions, as seen in Table 8.5 (p. 160). Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 0.762$) in the scores for the experimental group 2 ($\bar{x} = 4.25$, $\sigma = 0.96$) and baseline ($\bar{x} = 4.50$, $\sigma = 0.55$) conditions, as seen in Table 8.6 (p. 161). Consistent with their academic track, all groups exhibited a positive response, therefore the UML diagrams present in the tasks' description pose no thread to the validity of the experiment results.

BG1.8 I have considerable experience with Visual Studio IDE

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 0.352$) in the scores for the experimental group 1 ($\bar{x} = 3.00$, $\sigma = 1.41$) and baseline ($\bar{x} = 3.83$, $\sigma = 1.60$) conditions, as seen in Table 8.5 (p. 160). Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 0.914$) in the scores for the experimental group 2 ($\bar{x} = 4.00$, $\sigma = 0.82$) and baseline

($\bar{x} = 3.83$, $\sigma = 1.60$) conditions, as seen in Table 8.6 (p. 161). This item discarded the IDE as a validation threat to the reliability of the experiment results. This was expected due to the fact that Visual Studio is frequently used as an IDE during their academic track.

BG1.9 I have considerable experience using wikis

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 0.476$) in the scores for the experimental group 1 ($\bar{x} = 4.00$, $\sigma = 0.00$) and baseline ($\bar{x} = 4.33$, $\sigma = 0.52$) conditions, as seen in Table 8.5 (p. 160). Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 1.000$) in the scores for the experimental group 2 ($\bar{x} = 4.25$, $\sigma = 0.96$) and baseline ($\bar{x} = 4.33$, $\sigma = 0.52$) conditions, as seen in Table 8.6 (p. 161). As expected, all groups revealed familiarity with wikis. Their contact with this kind of collaborative technology starts early in their Integrated Master's study plan.

BG1.10 I have considerable experience with XML-based languages

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 0.257$) in the scores for the experimental group 1 ($\bar{x} = 3.25$, $\sigma = 0.96$) and baseline ($\bar{x} = 4.17$, $\sigma = 0.98$) conditions, as seen in Table 8.5 (p. 160). Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 0.352$) in the scores for the experimental group 2 ($\bar{x} = 3.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 4.17$, $\sigma = 0.98$) conditions, as seen in Table 8.6 (p. 161). To solve most of the tasks, the students had to use an XML-based language. This item discarded this as a posable threat to the validity of results.

BG2 I've never used or had contact with the OGHMA framework.

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 0.914$) in the scores for the experimental group 1 ($\bar{x} = 4.75$, $\sigma = 0.50$) and baseline ($\bar{x} = 4.83$, $\sigma = 0.41$) conditions, as seen in Table 8.5 (p. 160). Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 0.762$) in the scores for the experimental group 2 ($\bar{x} = 5.00$, $\sigma = 0.00$) and baseline ($\bar{x} = 4.83$, $\sigma = 0.41$) conditions, as seen in Table 8.6 (p. 161). This item confirmed the non-acquaintance with the *OGHMA* framework. This was a mandatory condition to the effective prosecution of the experiment goals. The few students that gave an answer not equals to 5 (*strongly agree*) were confronted and confessed to have heard of it by other colleagues, but confirmed never having used it or known its intent or application domain.

8.3.3 External factors

While designing the experiment, there was the concern of providing a neutral, familiar setting for the participants as to discard possible validation threatening environmental factors. But even in a common, usual working place there are aspects out of control (inter-group interaction, disturbances, noise, etc.) and that might pose as threats to the validity of the results. This section screens out those aspects.

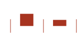
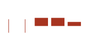
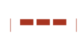
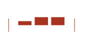
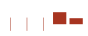
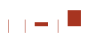
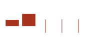
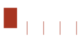
	EXPERIMENTAL GROUP 1			BASELINE			STATISTICS		
	12345	$\bar{x}$	σ	12345	$\bar{x}$	σ	H_1	W	ρ
EF1		3.00	1.15		3.33	1.37	$\neq$	20.00	0.762
EF2		3.00	0.82		3.00	0.89	$\neq$	22.00	1.000
EF3		4.50	0.58		4.50	0.84	$\neq$	21.00	0.914
EF4		1.50	0.58		1.00	0.00	$\neq$	27.00	0.256

Table 8.7: Summary of external factors results between the *Baseline (BL)* and *Experimental Group 1 (EG1)*, including the values of the non-parametric significance Mann-Whitney-Wilcoxon test.

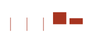
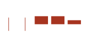
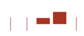
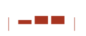
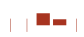
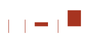
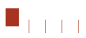
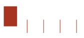
	EXPERIMENTAL GROUP 2			BASELINE			STATISTICS		
	12345	$\bar{x}$	σ	12345	$\bar{x}$	σ	H_1	W	ρ
EF1		4.50	0.58		3.33	1.37	$\neq$	26.00	0.171
EF2		3.75	0.50		3.00	0.89	$\neq$	27.00	0.257
EF3		3.50	0.58		4.50	0.84	$\neq$	14.00	0.114
EF4		1.00	0.00		1.00	0.00	$\neq$	22.00	1.000

Table 8.8: Summary of external factors results between the *Baseline (BL)* and *Experimental Group 2 (EG2)*, including the values of the non-parametric significance Mann-Whitney-Wilcoxon test.

EF1 I found the whole experience environment intimidating.

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 0.762$) in the scores for the experimental group 1 ($\bar{x} = 3.00$, $\sigma = 1.15$) and baseline ($\bar{x} = 3.33$, $\sigma = 1.37$) conditions, as seen in Table 8.7. Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 0.171$) in the scores for the experimental group 2 ($\bar{x} = 4.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 3.33$, $\sigma = 1.37$) conditions, as seen in Table 8.8. Overall, the scores discarded the experiment environment as a threat. Of course, there is always some degree of intimidation when we are prompted to participate in an experiment of this kind, but the overall levels of intimidation were within an acceptable range. Even so, and although not statistically significant, *EG2* exhibited a score a bit higher than expected. This deviation was probably caused by undertaking Treatment B § 8.2.3 (p. 154) and also from a more strong casual monitoring of their work. Nonetheless, this disturbance only occurred in the beginning of the experiment, so it can be discarded as a relevant threat to validity.

EF2 I enjoyed programming and developing in the experiment.

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 1$) in the scores for the experimental group 1 ($\bar{x} = 3.00$, $\sigma = 0.82$) and baseline ($\bar{x} = 3.00$, $\sigma = 0.89$) conditions, as seen in Table 8.7 (p. 164). Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 0.257$) in the scores for the experimental group 2 ($\bar{x} = 3.75$, $\sigma = 0.50$) and baseline ($\bar{x} = 3.00$, $\sigma = 0.89$) conditions, as seen in Table 8.8 (p. 164). This item measured the *fun factor* or the *novel factor*. The observed scores reveal that there was no negative feeling towards the experiment technical work, so this factor can be discarded as a threat to the whole experiment.

EF3 I would work with my partner again.

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 0.914$) in the scores for the experimental group 1 ($\bar{x} = 4.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 4.50$, $\sigma = 0.84$) conditions, as seen in Table 8.7 (p. 164). Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 0.114$) in the scores for the experimental group 2 ($\bar{x} = 3.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 4.50$, $\sigma = 0.84$) conditions, as seen in Table 8.8 (p. 164). It is said that "Birds of a feather flock together", that is, it was expected that when forming pairs, the participants would choose a colleague with whom they had already worked in the past. It was important to ascertain if the grouping factor was a threat and, consequently, the resulting scores discarded it.

EF4 I kept getting distracted by other colleagues outside my group.

Let $H_1 : G_{e1} \neq G_b$, there was **no significant** difference ($\rho = 0.256$) in the scores for the experimental group 1 ($\bar{x} = 1.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 1.00$, $\sigma = 0.00$) conditions, as seen in Table 8.7 (p. 164). Let $H_1 : G_{e2} \neq G_b$, there was **no significant** difference ($\rho = 1.000$) in the scores for the experimental group 2 ($\bar{x} = 1.00$, $\sigma = 0.00$) and baseline ($\bar{x} = 1.00$, $\sigma = 0.00$) conditions, as seen in Table 8.8 (p. 164). In a familiar, non-intimidating setting, it is easier to interact and to vocalise more, producing more noise and increasing the disturbance level. This item served to discard this factor, as shown by the low scores exhibited by all groups.

8.3.4 Overall satisfaction

This group of questions was intended to provide subjective validation to the thesis on an overall scope, by questioning subjects on their performance, comfort and feel for the presented collaborative environment.

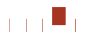
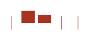
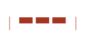
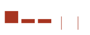
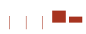
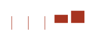
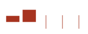
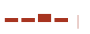
	EXPERIMENTAL GROUP 1			BASELINE			STATISTICS		
	12345	$\bar{x}$	σ	12345	$\bar{x}$	σ	H_1	W	ρ
OVS1		4.00	0.00		2.50	0.55	$>$	24.00	<u>0.005</u>
OVS2		2.75	0.96		1.50	0.84	$>$	29.50	<u>0.043</u>
OVS3		4.50	0.58		4.67	0.52	$<$	20.00	0.881
OVS4		1.75	0.50		2.50	1.05	$<$	16.50	0.953

Table 8.9: Summary of overall satisfaction results between the *Baseline (BL)* and *Experimental Group 1 (EG1)*, including the values of the non-parametric significance Mann-Whitney-Wilcoxon test.

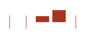
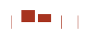
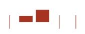
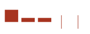
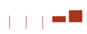
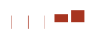
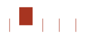
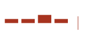
	EXPERIMENTAL GROUP 2			BASELINE			STATISTICS		
	12345	$\bar{x}$	σ	12345	$\bar{x}$	σ	H_1	W	ρ
OVS1		3.50	0.58		2.50	0.55	$>$	14.50	<u>0.048</u>
OVS2		3.00	0.82		1.50	0.84	$>$	27.00	<u>0.033</u>
OVS3		4.50	0.58		4.67	0.52	$<$	20.00	0.881
OVS4		2.25	0.50		2.50	1.05	$<$	20.00	0.738

Table 8.10: Summary of overall satisfaction results between the *Baseline (BL)* and *Experimental Group 2 (EG2)*, including the values of the non-parametric significance Mann-Whitney-Wilcoxon test.

OVS1 Overall, this particular setup was suitable for solving every task presented.

Let $H_1 : G_{e1} > G_b$, there was **a highly significant** difference ($\rho = 0.005$) in the scores for the experimental group 1 ($\bar{x} = 4.00$, $\sigma = 0.00$) and baseline ($\bar{x} = 2.50$, $\sigma = 0.55$) conditions, as seen in Table 8.9. Let $H_1 : G_{e2} > G_b$, there was **a significant** difference ($\rho = 0.048$) in the scores for the experimental group 2 ($\bar{x} = 3.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 2.50$, $\sigma = 0.55$) conditions, as seen in Table 8.10. The scores obtained by this item give strong evidence that the presented collaborative environment proved to be well suited for more easily learning about a framework. The score regarding *BL* vs. *EG1* is pretty clear to show that the patterns helped dealing with framework learning. The score regarding *BL* vs. *EG2* is not as strong but it is also significant. When presented with new tools, there is always a *learning curve* factor that attenuates the readily acceptance and recognition of a useful tool, especially in a time-constrained scenario such as this.

OVS2 I found the documentation available to be sufficient.

Let $H_1 : G_{e1} > G_b$, there was **a significant** difference ($\rho = 0.043$) in the scores for the experimental group 1 ($\bar{x} = 2.75$, $\sigma = 0.96$) and baseline ($\bar{x} = 1.50$, $\sigma = 0.84$) conditions, as seen in Table 8.9 (p. 166). Let $H_1 : G_{e2} > G_b$, there was **a significant** difference ($\rho = 0.033$) in the scores for the experimental group 2 ($\bar{x} = 3.00$, $\sigma = 0.82$) and baseline ($\bar{x} = 1.50$, $\sigma = 0.84$) conditions, as seen in Table 8.10 (p. 166). A typical issue of software development in general is that *there is never enough documentation*. In this experiment that is also the case. Nevertheless, the intent of this item was to perceive if the available tools would improve the usage and value of the available documentation, deemed sufficient to effectively undertake all the tasks presented. The exhibited scores support that assumption.

OVS3 I felt the need to have access to more information on how to use the framework.

Let $H_1 : G_{e1} < G_b$, there was **no significant** difference ($\rho = 0.881$) in the scores for the experimental group 1 ($\bar{x} = 4.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 4.67$, $\sigma = 0.52$) conditions, as seen in Table 8.9 (p. 166). Let $H_1 : G_{e2} < G_b$, there was **no significant** difference ($\rho = 0.881$) in the scores for the experimental group 2 ($\bar{x} = 4.50$, $\sigma = 0.58$) and baseline ($\bar{x} = 4.67$, $\sigma = 0.52$) conditions, as seen in Table 8.10 (p. 166). This item intended to measure how much did the provided collaborative environment supported the subjects cognitive needs. The collected scores show that there was still much to provide to fulfil their knowledge needs. It is believed that these scores were strongly influenced by the failure in completing iteration 4, that most subjects exhibited. The increased complexity of iteration 4 and the already long elapsed experiment time (tiredness) affected effectiveness during the last stages of the experiment. Therefore, the remaining feeling in the subjects minds was that they needed to know more about the framework to complete iteration 4, biasing the overall scores.

OVS4 Despite my experience, the tools available, excluding OGHMA, delayed my work considerably.

Let $H_1 : G_{e1} < G_b$, there was **no significant** difference ($\rho = 0.953$) in the scores for the experimental group 1 ($\bar{x} = 1.75$, $\sigma = 0.50$) and baseline ($\bar{x} = 2.50$, $\sigma = 1.05$) conditions, as seen in Table 8.9 (p. 166). Let $H_1 : G_{e2} < G_b$, there was **no significant** difference

($\rho = 0.738$) in the scores for the experimental group 2 ($\bar{x} = 2.25$, $\sigma = 0.50$) and baseline ($\bar{x} = 2.50$, $\sigma = 1.05$) conditions, as seen in Table 8.10 (p. 166). This item intended to screen out the hindrance level the presented tools might introduce during the experiment, specially affecting the time-related metrics. Despite not statistically relevant, the low scores tend to converge to, at least, a balance between the baseline and the experimental groups, so it is assumed that the tools didn't pose as a threat to the validity of the time-related results.

8.3.5 Development process

This category of items intended to ascertain how hard it was to complete each of the tasks presented and its evolution throughout the experiment, as means to measure the impact the collaborative environment had on the development process.

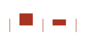
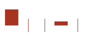
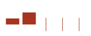
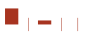
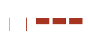
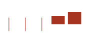
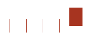
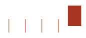
	EXPERIMENTAL GROUP 1			BASELINE			STATISTICS		
	12345	$\bar{x}$	σ	12345	$\bar{x}$	σ	H_1	W	ρ
DP1.1		2.50	1.00		1.67	1.21	$<$	26.00	0.971
DP1.2		1.75	0.50		1.50	0.84	$<$	29.50	0.833
DP1.3		3.75	0.96		4.17	1.17	$<$	18.50	0.795
DP1.4		5.00	0.00		5.00	0.00	$<$	22.00	1.000

Table 8.11: Summary of development process results between the *Baseline (BL)* and *Experimental Group 1 (EG1)*, including the values of the non-parametric significance Mann-Whitney-Wilcoxon test.

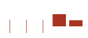
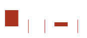
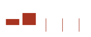
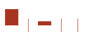
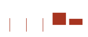
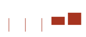
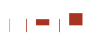
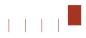
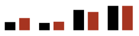
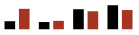
	EXPERIMENTAL GROUP 2			BASELINE			STATISTICS		
	12345	$\bar{x}$	σ	12345	$\bar{x}$	σ	H_1	W	ρ
DP1.1		4.25	0.50		1.67	1.21	$<$	22.50	1.000
DP1.2		1.75	0.50		1.50	0.84	$<$	29.50	0.833
DP1.3		3.75	1.26		4.17	1.17	$<$	19.00	0.853
DP1.4		4.00	1.15		5.00	0.00	$<$	16.00	1.000

Table 8.12: Summary of development process results between the *Baseline (BL)* and *Experimental Group 2 (EG2)*, including the values of the non-parametric significance Mann-Whitney-Wilcoxon test.

In none of the desired hypothesis ($H_1 : G_{e1} < G_b$ or $H_1 : G_{e2} < G_b$) did the scores produce any relevant statistical results, having some items, even, produced unexpected scores.

In an overall analysis, it can be stated that in the case of BL vs. EG1 , iteration 1 revealed to be an easy task, getting easier in iteration 2, but increasingly more difficult when it came to iteration 3 and 4 (where the non-completion of this iteration, led to the top score of 5). In the case of BL vs. EG2 , the results are similar to the above stated, with an increased score for iteration 1.

The overall scores of this group of questions was unexpected. This led to a follow-up informal interview with the students to try to understand their reasons for answering such scores. The main conclusion from this interview is that their interpretation of the items led them to answer not so much about the usage of the collaborative environment, but more about the complexity of the *OGHMA* framework and their subjective analysis of their own performance. As such, no evidence can be assumed from this group of questions, as the answers don't express its intended purpose. Nevertheless, when asked about iteration complexity, they noted that Iteration 1 (because it was the first) and Iteration 4 (no one was able to complete) were the most complex, ordering Iteration 3 as mildly complex and Iteration 2 as the less complex (especially after tackling with Iteration 1).

8.3.6 Framework knowledge

In order to measure the increase in framework knowledge, a set of 17 items was devised and presented to the subjects at the end of the experiment. These questions intended to ascertain how much correct information about the framework the participants had acquired. It was assumed that all groups (except the *Experts*) had no prior knowledge of the framework whatsoever, as corroborated by item BG2.

Categories

According to [AD05a], framework knowledge can be divided into layers, ranging from more abstract to more concrete information. Not only was it relevant to measure the amount of framework knowledge acquired, but also at what depth the subjects went in their learning of the framework. As such, framework knowledge can be divided into the following seven categories, each representing an abstraction layer, and giving examples of supporting documentation elements:

- **Overview (OV).** This category is intended to communicate the purpose of the framework to potential users, in a clear and concise way: framework overview, and snapshots.

- **Domain (DM).** This category defines the application domain covered by the framework, namely the products that can be developed with the framework, their variability aspects (hotspots), and how the framework can and should be reused: use cases, scenarios, examples, cookbooks, recipes, and design patterns.

- **Components (CP).** This category formally defines a black-box view, i.e. the properties and behaviour of the products that can be developed with the framework, and how they must interact with custom code: type specifications, operation specifications, state-charts, contracts, and design patterns.
- **Design (DN).** This category presents the design principles of the framework, and describe its micro-architectures and mechanisms of cooperation between components: technical architecture, application architecture, design patterns, design notebooks, and refinements from specification level to design level.
- **Public view of the implementation (PB).** This category represents an external view of the implementation of framework components: detailed use cases, collaborations, roles, interfaces, classes.
- **Protected view of the implementation (PT).** This category represents the view available for developers of components through extension of classes provided by the framework. In addition to public view, this protected view must include also subclass/superclass contracts: detailed use cases, collaborations, roles, interfaces, classes.
- **Private view of the implementation (PV).** This category presents a white-box view over the implementation of the framework, usually in the form of source code.

The items presented in the questionnaire tried to cover all these categories and had a true/false statement-like form (see Table 8.13). They were then shuffled, so its natural order wouldn't bias the subjects when answering.

Question	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Category	OV	OV	DM	DM	OV	CP	CP/DN	DN	PB	PT	PB	PT	PT	PV	PT	PV	PT/DN
Answer	T	F	T	F	F	F	T	T	T	T	F	F	T	F	T	T	T

Table 8.13: Framework knowledge items categorisation and answers. All items were presented as true-false statements. As for items 7 and 17, they both cover two categories as certain knowledge has relevance in several layers.

Results

The relevance of an item-to-item analysis of the scores isn't so much important as the total amount of knowledge the subjects acquired. So, the results are shown aggregated and processed in two ways: (i) total knowledge acquired and (ii) total knowledge acquired by category.

When answering a *true* or *false* statement, using a five-point format Likert scale § 8.2.2 (p. 153), the scores not only show the answer (*strongly disagree (1)* as *false* and *strongly agree (5)* as *true*) but also the confidence level of the respondent. The closer the answer gets to the boundaries of the scale, the more certain the subject is of the answer (being *neither agree nor disagree (3)* not knowing the answer). The scores were then processed and converted into distances from the correct answer, e.g., a score of 2 for a *true* statement (5) converts into a distance of 3 ($|5-2|=3$), whereas for a *false* statement it converts into a distance of 1 ($|1-2|=1$) and so forth. Items the subjects didn't know the answer (3) would always contribute the same distance (2). Finally, an average of the scores for each item was computed and, as done for the initial students' average pre-experiment evaluation § 8.1.1 (p. 149), an independent samples t-test was conducted to compare the averages of the items between the *Baseline (BL)* and *Experimental Groups 1* and *2*. These results can be seen in Table 8.14, Table 8.15 and Table 8.16. A comparison for the framework knowledge distances for each category can be seen in Figure 8.5 and Figure 8.6. All the aforementioned scores and processing step can be seen in Appendix G, in Tables G.3 (p. 206), G.4 (p. 207) and G.5 (p. 208).

Group	N	Mean	Std. Deviation	Std. Error Mean
BL	17	2.087	0.5598	0.1358
EG1	17	1.647	0.5234	0.1269
EG2	17	1.676	0.5431	0.1317

Table 8.14: Framework knowledge group statistics.

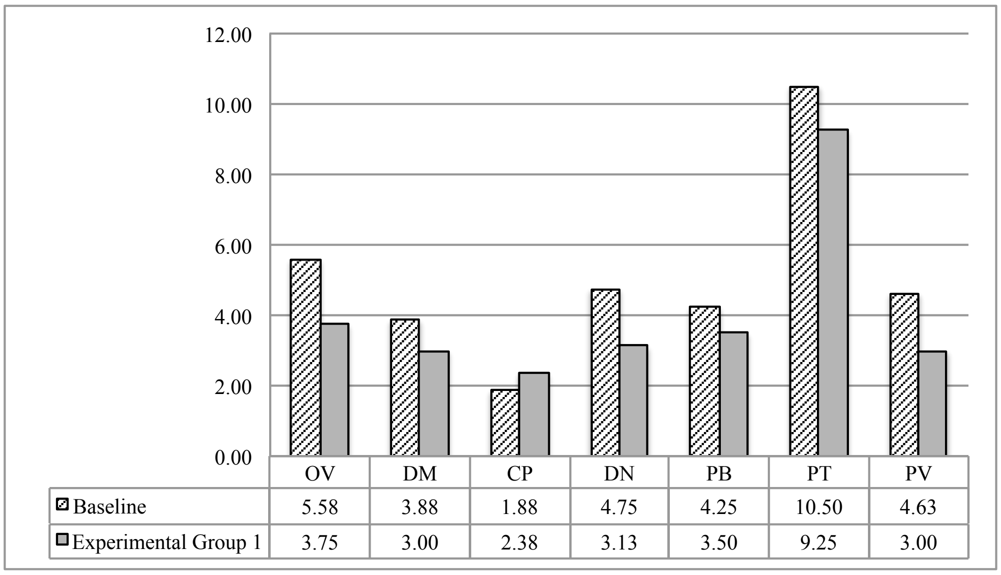
	F	Sig.	t	df	Sig. (2-tailed)
Eq. Var. Assumed	0.215	0.646	2.364	32.00	<u>0.024</u>
Eq. Var. Not Assumed			2.364	31.86	0.024

Table 8.15: Baseline vs. Experimental Group 1 Independent Samples Test for framework knowledge. The first two columns are the Levene's Test for Equality of Variances, showing a significance greater than 0.05 (0.646). The other three columns are the t-test for Equality of Means. Since we can assume equal variances, the 2-tailed value of 0.024 allow us to conclude that there is a **statistically significant difference** between the two conditions.

The results provide evidence that the collaborative environment contributes to an increase on framework knowledge acquisition, thus supporting the hypothesis that it helps novices on learning about a framework.

	F	Sig.	t	df	Sig. (2-tailed)
Eq. Var. Assumed	0.041	0.841	2.071	32.00	0.047
Eq. Var. Not Assumed			2.071	31.98	0.047

Table 8.16: Baseline vs. Experimental Group 2 Independent Samples Test for Framework Knowledge. The first two columns are the Levene's Test for Equality of Variances, showing a significance greater than 0.05 (0.841). The other three columns are the t-test for Equality of Means. Since we can assume equal variances, the 2-tailed value of 0.047 allow us to conclude that there is a **statistically significant difference** between the two conditions.



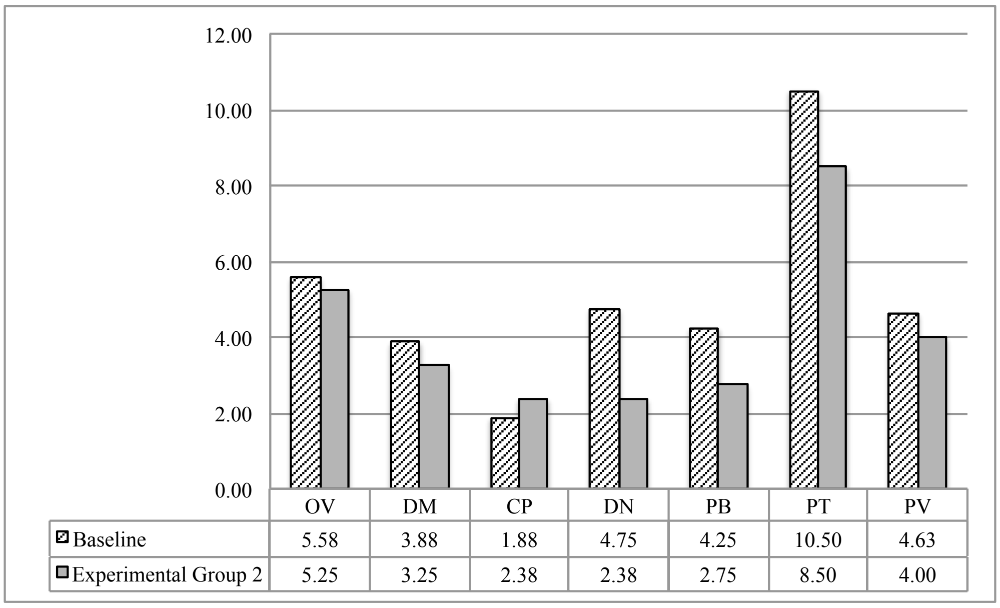
	OV	DM	CP	DN	PB	PT	PV
▨ Baseline	5.58	3.88	1.88	4.75	4.25	10.50	4.63
■ Experimental Group 1	3.75	3.00	2.38	3.13	3.50	9.25	3.00

Figure 8.5: Framework knowledge distances for *Baseline (BL)* and *Experiment Group 1 (EG1)*. For all categories, except *Components (CP)*, EG1 scored a better result than BL (lesser distance to the correct answer).

8.3.7 Objective measurement

During the experiment, the duration each group took to complete each iteration was recorded. At the end, these results were processed and corrected, considering the effectiveness of the deliverables, so that certain quality-related time deviations (e.g. failure to comply to requirements, code standards, implementation variations, etc...) could be minimised and the reliability of the results increased.

All deliverables were inspected for quality and effectiveness and graded, rendering a time penalty accordingly. The deliverables were graded with the following scale: *grade A (no time penalty)*, for deliverables that covered all the requirements and presented the expected implementations; *grade A- (5 minutes time penalty)* for deliverables that slightly deviated from the expected implementation, nevertheless covered all the requirements; *grade B, (10 minutes time penalty)* for deliverables that somewhat deviated from the



	OV	DM	CP	DN	PB	PT	PV
▨ Baseline	5.58	3.88	1.88	4.75	4.25	10.50	4.63
■ Experimental Group 2	5.25	3.25	2.38	2.38	2.75	8.50	4.00

Figure 8.6: Framework knowledge distances for *Baseline (BL)* and *Experiment Group 2 (EG2)*. For all categories, except *Components (CP)*, EG2 scored a better result than BL (lesser distance to the correct answer).

intended implementation, although still covering the requirements; *grade B-, (15 minutes time penalty)* for deliverables that failed to cover one or more requirements, although the coding of a possible solution was present. The grading of the deliverables can be seen in Table 8.17.

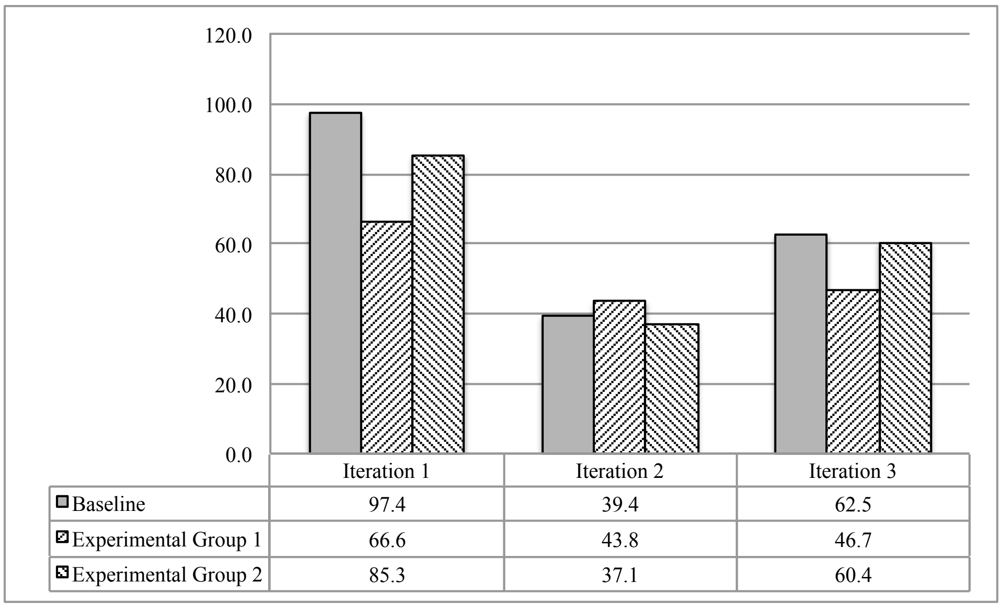
	Iterations			
	1	2	3	4
Baseline Pair 1	B	B	A-	-
Baseline Pair 2	B	B	B-	-
Baseline Pair 3	B	A-	B	-
Experimental Group 1 Pair 1	A-	A	A-	-
Experimental Group 1 Pair 2	B	B	A	-
Experimental Group 2 Pair 1	B	B	B	-
Experimental Group 2 Pair 2	A-	B	A-	-

Table 8.17: Deliverables grades.

The final results can be seen in Figure 8.7. An analysis iteration-wise follows.

Iteration 1

The exhibited results for Iteration 1, give strong evidence that the presented tools helped in the first contact with the framework. Both *EG1* and *EG2* took less time completing



	Iteration 1	Iteration 2	Iteration 3
▢ Baseline	97.4	39.4	62.5
▨ Experimental Group 1	66.6	43.8	46.7
▩ Experimental Group 2	85.3	37.1	60.4

Figure 8.7: Iteration completion time results (average per group). Units in minutes.

this iteration when compared to the *Baseline*. The longer time *EG2* took when compared to *EG1* can be explained by the overhead taken using a new tool (the DRIVER plug-ins).

Iteration 2

For Iteration 2, there is a more even set of results, where, oddly *EG1* took a slightly bit more than both *Baseline* and *EG2*. It is believed that the short finishing time they took in Iteration 1 may have rendered them overconfident for tackling with this iteration, thus affecting their time performance. On the other hand, despite slim, EG2 performed better than *Baseline*.

Iteration 3

Here the results again support the usefulness of the presented tools where both *EG1* and *EG2* perform better than the *Baseline* group.

Iteration 4

There were no deliverables or time results for this Iteration, due to the fact that only a couple of groups finished within the expected time frame for the experiment, rendering their results useless for analysis. This is explained by the increased complexity of the iteration, combined with the already experiment duration and tiredness of the groups. Even so, it served to force the subjects to dwell deeper into the framework code as to provide data for ascertaining how much framework knowledge the subjects could intake, which has already been analysed in § 8.3.6 (p. 169). The experiment ended at a pre-defined time for all subjects.

Overall

As an overall analysis, the results indicate that there were better time performances from *Experimental Groups 1* and *2*, in comparison to the *Baseline* group. It is interesting to observe that when complexity and extensiveness increase, the results are better. This can be seen in Iteration 1, when there is a first contact with the framework and then again in Iteration 3. The complexity gap between Iteration 1 and 2 isn't so great, so the performance is somewhat similar. But when one *turns up the heat*, (from Iteration 2 to Iteration 3) the tools step in to aid on performing better.

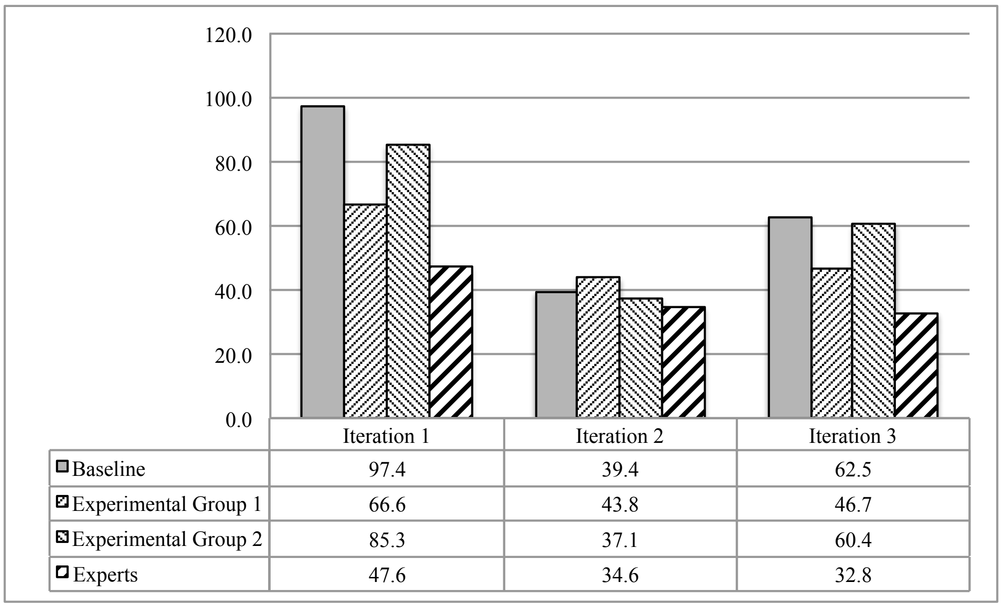
8.3.8 Experts group analysis

The purpose of the *Experts* group was to study the impact the collaborative approach might have on subjects with prior knowledge of the framework, although not actively using it. For this experiment, the subjects had been in contact with the framework around 6 months prior to the experiment and during a period of 3 months. During this time, they engaged in instantiation and evolution tasks. The inactivity period allowed the framework knowledge acquired to decay, therefore, regaining contact with the framework, after a while, would generate different cognitive needs than the novice users that had never used the framework.

Initially, during the experiment design, this group was not supposed to exist. But while selecting the subjects for the experiment, the profile of several students presented an opportunity to study this scenario where frameworks users regain contact with a previous framework. The study wasn't so much concerned about the time issue (although it would be monitored), but with the knowledge acquisition and usage of the collaborative approach (both patterns and toolset).

As such, the *Experts* group undertook Pre-Questionaire B § 8.2.2 (p. 153) not only to screen their basic skills and background, but also to take a snapshot of their prior knowledge of the framework. They answered the same questions about the framework before and after (Post-Questionnaire § 8.2.5 (p. 159)) the experiment, to measure the knowledge acquisition metric.

The results prove inconclusive as to where the collaborative approach helps experts on improving framework learning. Despite the expected better time performance when compared with the other groups (Figure 8.8), the knowledge acquisition results don't indicate (Table 8.18 and Table 8.19) that the approach helped the *Experts* group in their (re-)learning of the framework.



	Iteration 1	Iteration 2	Iteration 3
■ Baseline	97.4	39.4	62.5
▨ Experimental Group 1	66.6	43.8	46.7
▧ Experimental Group 2	85.3	37.1	60.4
▨ Experts	47.6	34.6	32.8

Figure 8.8: Comparing iteration completion time results (average per group) between the *Experts* group and all others. The deliverables were also graded, as can be seen in Appendix § H (p. 211) and the results equally processed. Units in minutes.

Group	N	Mean	Std. Deviation	Std. Error Mean
Experts before experiment	17	1.522	0.5620	0.1363
Experts after experiment	17	1.556	0.5474	0.1328

Table 8.18: Framework knowledge acquisition by the *Experts* group

Despite these results, other indicators where captured, namely the usage of the collaborative approach to capture their own knowledge of the framework. Although not disclosed to them, they quickly discovered that some (if not all) the subjects of the *Experts* group were sharing the same knowledge base of learning paths, so they spent some time adding their own and rating others. It is believed that, without the burden of the overhead of learning a new framework, the subjects spent some time using the DRIVER platform to capture their learning paths and improve the knowledge-base with their own expertise. In a later follow-up session, they provided lots of useful feedback on how to improve the collaborative environment.

	F	Sig.	t	df	Sig. (2-tailed)
Eq. Var. Assumed	0.002	0.965	-0.176	32.00	0.861
Eq. Var. Not Assumed			-0.176	31.98	0.861

Table 8.19: Experts group Independent Samples Test for framework knowledge. The first two columns are the Levene's Test for Equality of Variances, showing a significance greater than 0.05 (0.965). The other three columns are the t-test for Equality of Means. Since we can assume equal variances, the 2-tailed value of 0.861 allow us to conclude that there is no statistically significant difference between the two conditions.

8.4 Validation threats

The outcome of validation is to gather enough scientific evidence to provide a sound interpretation of the results. Validation threats are issues and scenarios that may distort that evidence and thus incorrectly support (or discard) expected results. Each validation threat should be expected and addressed *a priori* in order to yield unbiased results or, at least, minimised *a posteriori* with effective counter-measures.

This section addresses expected validation threats and how these were discarded, while others should be attentively focused in future experiments.

- **Misunderstanding of the given tasks.** Because the tasks relied on textual specifications and UML diagrams, is was necessary to ensure that the participants correctly interpret them. This threat was discarded by the pre-experiment evaluation of their basic skills and through the pre-experiment questionnaire (specially item BG1.7, regarding UML).

- **Insufficient skills to execute the tasks.** The tasks required participants to have the necessary skill to build and evolve information systems, namely knowing how to work with the given programming language, IDE and database engine. Once again, this threat was discarded by both pre-experiment evaluation and pre-experiment questionnaire, through items BG1.2, BG1.3, BG1.8 and BG1.10.

- **Overhead due to lack of experience with the new tools.** When presented with new tools, these will, unavoidably, introduce overhead into the development process. This overhead was expected and compensated when interpreting the observed results. Nevertheless, the evaluation of the participants subjective feel over this overhead had to be discarded so that the tools would not pose as a relevant threat beyond expectations. This discarding was made through the post-experiment questionnaire, namely item OVS4.

- **Experiment-related factors.** Knowingly being part of an experiment, changes the mood and may be an inhibitor of normal development. The performance may be conditioned by the feel of being observed and *judged*. The results of item EF1 allow this threat to be discarded.

- **Team factors.** Despite the forming of pairs being handed to the participants as to alleviate the possibility of having *conflicting* partners, it was necessary to make sure that the final grouping wasn't a threat to validity. Item EF3 discards this threat.

- **Lack of motivation.** Due to the length of the tasks (experiment went beyond 3 hours), and the fact that there was no compensation to individuals participating in the experiment, the lack of motivation could hinder the outcome. This threat is discarded by item EF2.

- **Inter-group competition.** In an open-space setting and knowing all groups are undertaking the same experiment, the ability to have feedback on how others are performing may influence groups differently. Different people react differently to pressure. Therefore, there was the need to discard this threat by ascertaining if this pressure was an issue. Item EF4 served this purpose.

The following threats were not completely discarded, and should be the focus of future studies:

- **Assertion of task completion.** One difficulty the subjects exhibited during the experiment was making sure the developed solution covered all the requirements for that iteration. There was no deterministic mechanism (e.g., automated tests) that would quickly allow the groups to effectively verify the effectiveness of their solution. They had to manually conduct a series of interaction tests to ascertain the quality of their work. This threatened the time results and so forced a thorough investigation of the deliverables to compensate for the effectiveness deviations that might had occurred. In future experiments, this deterministic mechanism should exist to assure the effectiveness of the deliverables. It could even automatically advance the group to the next iteration, on full coverage of the presented requirements.

- **OGHMA Framework aversion.** This threat happened only in the *Experts* group. It was noted that some subjects of this group stated that they had a *not so good experience* with the *OGHMA* framework in the past. Therefore, their motivation and willingness to provide good results was somewhat fragile. Although motivated *in loco* to the tasks of the experiment, this threat might happen in future experiments when the selection of the framework causes feelings of dislike to the participants and, therefore, affect the outcome of the experiment. Of course, this can be discarded by the item EF2, nevertheless, its importance is reason enough to point it out.

The power of this study could also be improved by (i) increasing the number of participants, and (ii) switching the participants roles, where individuals in the experimental groups would undergo the baseline process and *vice-versa*.

8.5 Summary

This chapter detailed a quasi-experiment conducted within a controlled experimental environment using the collaborative approach presented in previous chapters.

There were two major groups: one of novices and another of *Experts*. The novices were then divided into three groups (*Baseline*, *Experimental Group 1* and *Experimental Group 2*) to which their background and basic skills were screened through a pre-experiment questionnaire, guaranteeing no statistical deviation.

All three groups went through the development of the same technical tasks using a previously unknown framework (*OGHMA*), all using different development settings to enable a comparison between having, or not having, the proposed collaborative approach. The *Experts* group already knew the framework, but served to assess the usefulness of the collaborative approach.

A post-questionnaire and their time track were used to assess the outcome of the experiment.

The final results support the hypothesis that the collaborative approach helps novices to more effectively learn about a framework. Although more evident between the *Baseline* and *Experimental Group 1*, than *Experimental Group 2*, both experimental groups fared better at both time and knowledge intake, when compared to the *Baseline* group. No evidence was collected when it comes to experts, but it is believed that, it allows experts to easily capture and share their learning knowledge about the framework.

Some threats to this validation were identified and further discarded by analysing the results in pre- and post-experiment questionnaires and due to the nature of the experimental setting. Not all original hypothesis were supported, though and some borderline threats which emerged after the experiment can help refine new studies.

Chapter 9

Conclusions

This dissertation focused on the issues behind the understanding of frameworks, i.e., acquiring knowledge on how to use a framework for a specific task at hand.

Frameworks are a powerful technique for software reuse, but with its power comes its complexity and difficulty in learning how to use. Mitigating factors can come from suitable documentation and proper training, but these are often neglected and provide insufficient assistance to the framework learner. Hence, framework understanding, as a sub-domain, shares most of the trends and challenges of the program comprehension domain, adding its own flavour according to its specific aspects and characteristics.

Frameworks are a cornerstone of, nowadays, software development - an activity that is achieving, more and more, a social emphasis. Software is developed by groups of people, ranging from a handful of elements, to an entire community of developers. With the advent of the Web and improvement of communication infra-structures, the concept behind developing software, and keeping the knowledge generated by such an activity, has evolved. Sharing becomes essential, and collaboration proves vital to keep a sustainable pace and to provide effective results with satisfactory quality levels.

Using the framework philosophy of *reusing experience*, the work presented in this dissertation intended to provide the framework learner with tools and techniques (***patterns***) to improve its learning experience and attenuate the learning curve it takes to understand a framework. Acknowledging the community of developers and learners, it devised a ***process*** and developed supporting ***tools*** to nurture the collaboration between these elements and further enhance the learning experience, thus, improving framework understanding.

Evidence was collected that verifies the benefits behind these contributions and helps on the validation of the presented work.

9.1 Key contributions

Briefly, the main contributions of the work presented in this dissertation are:

- **Elicitation of the best practices of framework understanding, documented in a pattern form.** Through *historical* and *observational* methods, the author mined, gathered and analysed experiences and identified the recurrent best practices when understanding a framework. These best practices (9 patterns) were then compiled into a set of patterns as a suitable form to communicate the empirical knowledge they hold. The learner is presented with a guiding tool to help focusing and personalising the learning process to achieve quicker and effective results.

- **Definition of a process that supports learning a framework collaboratively.** Collaboration is good if it is mutually beneficial. But if it becomes intrusive, it fails. Helping learners might sound like a unilateral deal and can become a tiresome activity if not properly moderated and dealt with. Therefore, the author defined a collaborative process of helping learners that tries to attenuate the intrusiveness such an activity might bring. This process enables the capturing of intrinsic knowledge that, otherwise, would be lost. Through a cycle of capture, filter, share, rate and recommend, the process allows the seamless collaboration of the community of learners, grasping its collective knowledge and expertise and, thus, improving the learning experience.

- **Tool support for the learning process in a collaborative environment.** For every process to be effective, it needs to be suitably supported. As such, the author developed a set of tools that provide support and allow the community of developers and learners to follow and apply the devised collaborative learning process without much effort. The tools are presented in a popular, collaborative environment: a *wiki*, garnished with framework documentation artifacts and extension capabilities.

- **Impact study of the key benefits of the best practices and learning process through a repeatable (quasi-)experiment.** Insight on the impacts that both patterns and the collaborative learning process have on learners and developers was ascertained through a controlled (quasi-) experiment. This experiment was conceived as a repeatable package, allowing other researchers to conduct and aggregate further results. Such empirical studies provide supporting evidence for theories and techniques regarding software engineering and, besides serving as a validation strategy, they raise other issues that may spur further research and define directions on forthcoming work.

9.2 Future work

Scientific research is always a work in progress and new directions emerge constantly, whether to improve the present findings or to explore new possibilities. The research paths described in the next sections are deemed, by the author, worth of pursuit.

9.2.1 Improve and enhance the DRIVER platform

The DRIVER platform, presented in Chapter 7 (p.109) is an on-going project that already has a forthcoming plan of development, as to improve and extend its functionalities. As such, the next steps of development will focus on the following goals:

- **Improve recommendation heuristics.** The *Hint* plug-in relies on a simple recommendation algorithm that uses a single-step *match position and look-ahead* heuristic. This algorithm can be improved by extending the matching, not only to the present position, but the trail already being captured. As the categorisation of the information becomes more *semantic* (see last item on this list), the recommendation heuristics can rely on a stronger relationship between the learning paths and provide better results.

- **Improve learner's profile awareness.** The main goal here is to make the community converge into a social network of learners, with the purpose of using the profile similarity to achieve and recommend better results. Not only will it matter how it was rated, but who rated it. Users would have expertise ranks according to their behaviour and this *reward* system would motivate learners to provide real, effective feedback on the usefulness of the learning paths.

- **Converge to the semantic web.** Can the *folksonomy* evolve to an *ontology*[1]? Enriching the categorisation and relationship between learning paths, learners and artifacts might bring new emerging knowledge and improve the reliability of results.

Besides these next steps in development, it is expected to consider all the suggestions given by the (quasi-)experiment subjects (Chapter 8, p. 147) to improve the usability and

[1] According to Tom Gruber [Gru93], and *ontology* is a "*formal, explicit specification of a shared conceptualisation*", i.e., a definition of concepts or objects, their properties and relationships, rendering a shared vocabulary or taxonomy.

effectiveness of the platform. Furthermore, there is also a plan to develop a bridge between the platform and Eclipse IDE, as a means to progressively integrate its functionalities in a coding environment.

9.2.2 Refine and extend the patterns

Although patterns are intended to be timeless[2], thoughts on the epistemology of patterns [KP09] invokes us to convey that:

> *As Buschmann, Henney and Schmidt [BHS07] point out, patterns and pattern descriptions evolve over time. Including new findings, e.g. new relevant forces, new consequences, new contexts or limitations, means to get a better understanding of the nature of a particular pattern. Therefore, pattern descriptions should be open towards inspiration by scientific progress, for instance the discovery of new materials, new procedures or new findings in human-computer interaction.*

As such, we must always be on the outlook for new empirical knowledge that might enhance, enrich and refine our own best practices, or, that might replace it, if that is the case.

9.2.3 Further studies

The performed experiment presented in chapter 8 (p. 147), provided supporting evidence for the benefits brought by the proposed contributions. Even so, further studies are required to solidify the results and consolidate the research.

Industrial settings

Empirical studies should be performed in an professional, non-academic setting, with developers engaged in full-scale software projects with defined time frames and development process. These case-studies should engage in critical reflexion of results with periodic interviews, questionnaires and focus groups over an extended period of time. Lessons should be learned that might be useful to help others and to act as agents of change in a real-life problem setting. As this dissertation is written, efforts are underway to undertake these studies and provide further results for analysis.

Enlarge the Community

So far, the studies have constrained the community of developers to a well-defined group of people, i.e., the development team. What happens if we extend the community beyond

[2] Unless replaced by new, better ones.

the team and embrace the web community? Insights on the impact this might have on the proposed contributions would provide relevant data and would raise, amongst others, scalability issues.

Impact of collaboration in framework understanding

Will the collaborative learning of frameworks change the way developers look at a framework? Will documentation change and adapt to a more collaborative learning experience. Will new formats or document types emerge from such as effect? These questions should elicit further studies.

Impact of collaboration in software learning in general

Developers gain by learning a framework collaboratively. But can we generalise these findings to software in general? Or are frameworks too specific? Broadening the learning cycle concept to software in general could bring new insights into ways of collaboratively enhancing the general software learning experience, beyond that of frameworks.

9.3 Final remarks

To end this dissertation, I would like to go back to the start.

When I started off my PhD adventure, my supervisor, Ademar, gave a seminar lecture about *the path of a PhD*. This lecture was intended to students, such as myself at the time, starting their PhDs. He intended to give insight and share his thoughts and experience on how his PhD (that he had recently finished) had gone through.

Purposely or not, he based his presentation on a set of patterns by Joseph Bergin, entitled *Patterns for the Doctoral Student* [Ber]. The context of the patterns read: *"You are considering a Doctoral level degree or are engaged in one. You have many sub goals, but one overriding goal: completing your doctorate and getting on with your life. How should you proceed?"*. Then, Ademar proceeded pointing out his pattern sequence that enabled him to successfully finish his PhD.

His presentation was very joyful, but tremendously insightful. I got caught up with the pattern form as a way to, quickly and effectively communicate a considerable amount of information (from a topic completely outside software) in a simple and straightforward manner. I had a map and a compass on how to get my PhD done.

Looking back, I could now make my own presentation (that would take another chapter) but I will, merely, point out three *patterns* that I observed and experienced while pursuing my PhD, and that I will strongly advise others, starting their PhDs:

- *Have passion for the program, to avoid burnout.* Motivation fluctuates as obstacles arise. If you like what you set up to do, it will help to keep your motivation levels high when faced with these obstacles.

- *Avoid complications in your life.* As my supervisor, humorously, said: *Don't change jobs, don't get married, don't have kids.* You might not think so, but every major change in your life will postpone the completion of your PhD. This doesn't mean *don't have a life*, but be aware of the impact big changes will have on your work.

- *Always maintain a sustainable effort, even if each step is tiny.* Continuous progress is hard, but small steps can be taken when large ones are elusive. Come up with simple metrics to measure your progress, daily (e.g., papers read or printed, words written, planning done, reasoning time, etc). These will give you feedback and will monitor your progress. Even very slightly, you'll feel you are closer to your goal than yesterday.

In retrospective, a PhD is like *learning how to drive* (learning how to do research). Once you get your *driver's license* (i.e. PhD degree), you'll be *driving real cars* (researching) for the rest of your life.

To close, I would like to leave the words of Joseph Bergin, referring the QWAN[3] in his patterns, and to which I subscribe completely.

> *Your life and work should make a contribution to the betterment of the world. The doctorate is one way to get yourself started on that path, but getting the degree can be a barrier. Rather, you want to think of it as a gate that opens you to the tools by which you can accomplish great things. But you must pass through that gate in order to do so, and you must move beyond the gate as well. The doctorate is not an end in itself, but a beginning. Its intent is for your growth and the improvement of the world as well. You don't receive the degree purely on your own work and merits, but on the work of a vast number of people (and resources) including your family, society as a whole, and other scholars stretching back to antiquity and beyond. Honour their work as we honour yours.*

Thank you for reading.

[3] QWAN is the Quality Without a Name [Ale79]. In short, this quality, which is known to everyone but cannot be named, is that quality of things that enhances human life and potential. In other contexts it is sometimes described as *"goodness, truth, and beauty"* but it goes beyond these things.

Appendices

Appendix A

Pre-experiment subject data

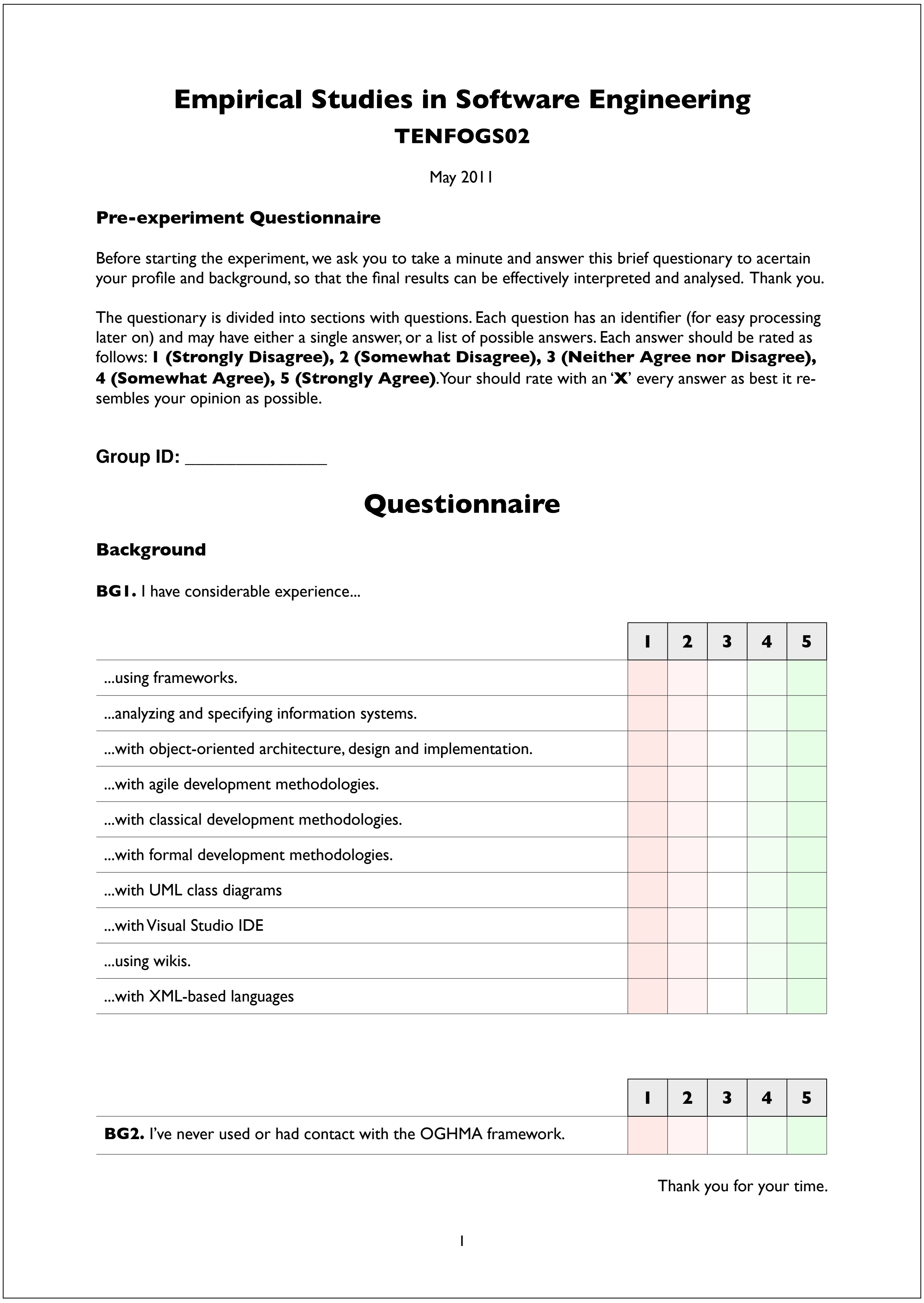
	I	II	III	IV	V	VI	VII	$\bar{x}$	σ
BL Subject A	16	14	14	18	15	17	18	16.0	1.7
BL Subject B	18	18	20	17	14	18	18	17.6	1.8
BL Subject C	18	17	16	18	17	18	18	17.4	0.8
BL Subject D	18	18	19	19	15	18	18	17.9	1.3
BL Subject E	-	-	19	17	14	18	15	16.6	2.1
BL Subject F	13	13	14	18	16	18	17	15.6	2.2
EG1 Subject A	17	16	17	18	15	18	19	17.1	1.3
EG1 Subject B	18	17	18	19	16	18	16	17.4	1.1
EG1 Subject C	18	18	19	19	16	18	19	18.1	1.1
EG1 Subject D	13	12	17	14	14	18	17	15.0	2.3
EG2 Subject A	14	15	15	18	14	18	16	15.7	1.7
EG2 Subject B	12	15	15	18	14	18	16	15.4	2.1
EG2 Subject C	16	17	15	17	16	18	18	16.7	1.1
EG2 Subject D	18	19	20	19	19	18	18	18.7	0.8
EX Subject A	18	16	18	18	14	17	16	16.7	1.5
EX Subject B	16	17	17	18	14	17	19	16.9	1.6
EX Subject C	16	17	16	16	15	17	16	16.1	0.7
EX Subject D	17	17	18	18	15	18	15	16.9	1.3
EX Subject E	12	15	14	18	16	18	17	15.7	2.2
EX Subject F	15	17	16	18	15	18	16	16.4	1.3
EX Subject G	18	19	19	20	19	18	18	18.7	0.8
EX Subject H	15	12	14	17	15	16	17	15.1	1.8
EX Subject I	16	18	18	18	15	18	18	17.3	1.3

Table A.1: Student grades for all participating groups. Each column represents the following courses: (I) Programming Fundamentals, (II) Programming, (III) Algorithms and Data Structures, (IV) Algorithm Design and Analysis, (V) Software Engineering, (VI) Software Development Laboratory, and (VII) Information Systems.

Appendix B

Pre-questionnaire A

The following is a copy of the anonymous questionnaire handed to the subjects of *Baseline*, *Experimental Group*s *1* and *2* at the beginning the experiment.



Empirical Studies in Software Engineering

TENFOGS02

May 2011

Pre-experiment Questionnaire

Before starting the experiment, we ask you to take a minute and answer this brief questionary to acertain your profile and background, so that the final results can be effectively interpreted and analysed. Thank you.

The questionary is divided into sections with questions. Each question has an identifier (for easy processing later on) and may have either a single answer, or a list of possible answers. Each answer should be rated as follows: **1 (Strongly Disagree), 2 (Somewhat Disagree), 3 (Neither Agree nor Disagree), 4 (Somewhat Agree), 5 (Strongly Agree)**.Your should rate with an '**X**' every answer as best it resembles your opinion as possible.

Group ID: _______________

Questionnaire

Background

BG1. I have considerable experience...

	1	2	3	4	5
...using frameworks.					
...analyzing and specifying information systems.					
...with object-oriented architecture, design and implementation.					
...with agile development methodologies.					
...with classical development methodologies.					
...with formal development methodologies.					
...with UML class diagrams					
...with Visual Studio IDE					
...using wikis.					
...with XML-based languages					

	1	2	3	4	5
BG2. I've never used or had contact with the OGHMA framework.					

Thank you for your time.

1

Appendix C

Pre-questionnaire A answers

	Baseline						$\bar{x}$	σ	EG1				$\bar{x}$	σ	H_1	W	$\rho \neq$	$\rho <$	$\rho >$
BG1.1	5	4	4	4	3	4	4.00	0.63	2	3	3	5	3.25	1.26	$\neq$	16.5	0.257	0.110	0.919
BG1.2	4	4	3	3	3	4	3.50	0.55	3	5	4	4	4.00	0.82	$\neq$	28.5	0.352	0.262	0.928
BG1.3	5	5	4	5	5	5	4.83	0.41	4	5	4	5	4.50	0.58	$\neq$	18.0	0.476	0.333	0.967
BG1.4	3	4	3	4	2	4	3.33	0.82	3	4	3	4	3.50	0.58	$\neq$	32.0	0.914	0.548	0.738
BG1.5	4	3	3	4	4	5	3.83	0.75	3	4	4	4	3.75	0.50	$\neq$	21.5	0.914	0.619	0.667
BG1.6	3	3	3	3	3	3	3.00	0.00	3	4	2	3	3.00	0.82	$\neq$	22.0	1.000	0.733	0.734
BG1.7	4	5	5	4	4	5	4.50	0.55	4	5	4	4	4.25	0.50	$\neq$	19.0	0.610	0.452	0.929
BG1.8	5	5	4	3	1	5	3.83	1.60	4	1	4	3	3.00	1.41	$\neq$	17.0	0.352	0.162	0.895
BG1.9	4	5	4	4	5	4	4.33	0.52	4	4	4	4	4.00	0.00	$\neq$	18.0	0.476	0.333	1.000
BG1.10	5	5	3	3	5	4	4.17	0.98	4	3	2	4	3.25	0.96	$\neq$	16.0	0.257	0.148	0.938
BG2	5	5	5	4	5	5	4.83	0.41	5	4	5	5	4.75	0.50	$\neq$	21.0	0.914	0.667	0.866

Table C.1: Pre-experiment questionnaire A results for *Baseline* and *Experimental Group 1 (EG1)*, each line representing the data of a single question for both groups, with corresponding means and standard deviation values. It includes the values of the non-parametric significance Mann-Whitney-Wilcoxon test; see § 8.3.2 (p. 160).

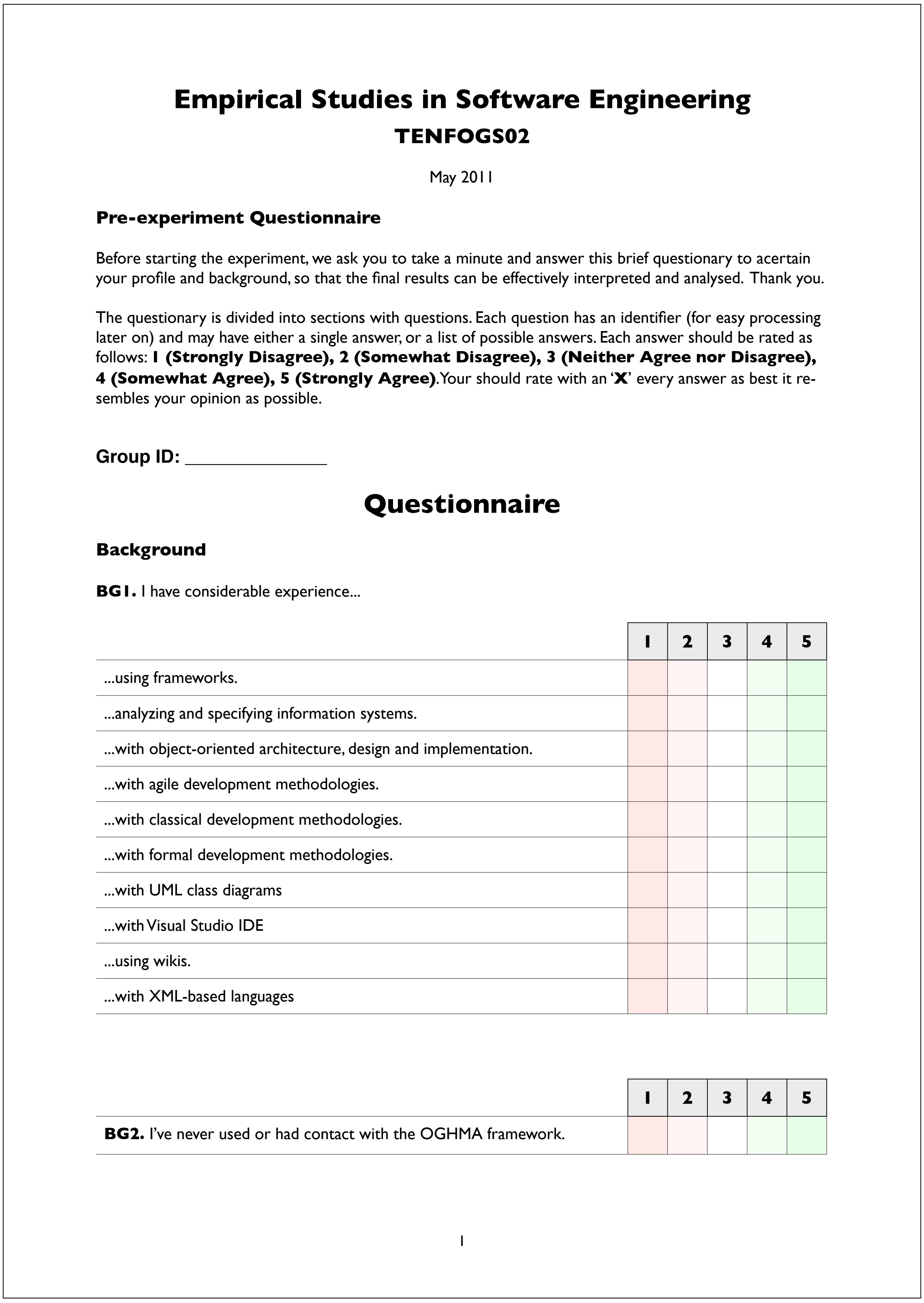
	Baseline						$\bar{x}$	σ	EG2				$\bar{x}$	σ	H_1	W	$\rho \neq$	$\rho <$	$\rho >$
BG1.1	5	4	4	4	3	4	4.00	0.63	3	3	4	5	3.75	0.96	$\neq$	19.50	0.610	0.319	0.824
BG1.2	4	4	3	3	3	4	3.50	0.55	3	4	4	5	4.00	0.82	$\neq$	28.50	0.352	0.262	0.928
BG1.3	5	5	4	5	5	5	4.83	0.41	4	4	5	5	4.50	0.58	$\neq$	18.00	0.476	0.333	0.967
BG1.4	3	4	3	4	2	4	3.33	0.82	3	3	4	4	3.50	0.58	$\neq$	32.00	0.914	0.548	0.738
BG1.5	4	3	3	4	4	5	3.83	0.75	4	4	3	2	3.25	0.96	$\neq$	18.00	0.476	0.257	0.886
BG1.6	3	3	3	3	3	3	3.00	0.00	4	4	4	2	3.50	1.00	$\neq$	27.00	0.257	0.129	0.876
BG1.7	4	5	5	4	4	5	4.50	0.55	4	3	5	5	4.25	0.96	$\neq$	20.50	0.762	0.452	0.738
BG1.8	5	5	4	3	1	5	3.83	1.60	3	4	5	4	4.00	0.82	$\neq$	21.00	0.914	0.490	0.624
BG1.9	4	5	4	4	5	4	4.33	0.52	3	4	5	5	4.25	0.96	$\neq$	22.00	1.000	0.548	0.595
BG1.10	5	5	3	3	5	4	4.17	0.98	3	3	4	4	3.50	0.58	$\neq$	17.00	0.352	0.205	0.881
BG2	5	5	5	4	5	5	4.83	0.41	5	5	5	5	5.00	0.00	$\neq$	31.00	0.762	0.600	1.000

Table C.2: Pre-experiment questionnaire A results for *Baseline* and *Experimental Group 2 (EG2)*, each line representing the data of a single question for both groups, with corresponding means and standard deviation values. It includes the values of the non-parametric significance Mann-Whitney-Wilcoxon test; see § 8.3.2 (p. 160).

Appendix D

Pre-questionnaire B

The following is a copy of the anonymous questionnaire handed to the subjects of the *Experts* group at the beginning the experiment.



Empirical Studies in Software Engineering

TENFOGS02

May 2011

Pre-experiment Questionnaire

Before starting the experiment, we ask you to take a minute and answer this brief questionary to acertain your profile and background, so that the final results can be effectively interpreted and analysed. Thank you.

The questionary is divided into sections with questions. Each question has an identifier (for easy processing later on) and may have either a single answer, or a list of possible answers. Each answer should be rated as follows: **1 (Strongly Disagree), 2 (Somewhat Disagree), 3 (Neither Agree nor Disagree), 4 (Somewhat Agree), 5 (Strongly Agree)**.Your should rate with an '**X**' every answer as best it resembles your opinion as possible.

Group ID: _______________

Questionnaire

Background

BG1. I have considerable experience...

	1	2	3	4	5
...using frameworks.					
...analyzing and specifying information systems.					
...with object-oriented architecture, design and implementation.					
...with agile development methodologies.					
...with classical development methodologies.					
...with formal development methodologies.					
...with UML class diagrams					
...with Visual Studio IDE					
...using wikis.					
...with XML-based languages					

	1	2	3	4	5
BG2. I've never used or had contact with the OGHMA framework.					

1

About the OGHMA framework...

	1	2	3	4	5
AF1. The OGHMA framework is suitable to automatically generate graphical user interfaces.					
AF2. Most of the components in the OGHMA framework are only used in the deployment phase of a project..					
AF3. I would use the OGHMA framework to build information systems.					
AF4. The OGHMA framework is not suitable for systems that keep changing their domain model..					
AF5. I would use most of the components in the OGHMA framework to build multi-agent systems.					
AF6. Persistency's component cannot be changed to support different technologies (relational databases, distributed key-value DBs, etc).					
AF7. Every action issued by the end-user is concatenated into a Transaction, which is committed when he decides to Save.					
AF8. The domain-specific language is implemented by following the Interpreter pattern.					
AF9. The Entity and Thing form the basis of the TypeSquare pattern, along with Property and PropertyType.					
AF10. Attributes and Relations are different specializations of Properties.					
AF11. Invariants are defined as bool-returning expressions and never apply to an Entity.					
AF12. Every run-time object derives from the class Entity.					
AF13. To add a new concept to the infrastructure, I would sub-class Thing.					
AF14. State-based and operation-based commits take fundamentally different code paths.					
AF15. MergedContainer is a special type of container used to ensure a certain level of atomicity, consistency, isolation and durability.					
AF16. Every user has a MutableChangeset that collects his operations, before the transaction is commited.					
AF17. Serialization takes place by considering the State object of the Thing being serialized.					

Thank you for your time.

Appendix E

Pre-questionnaire B answers

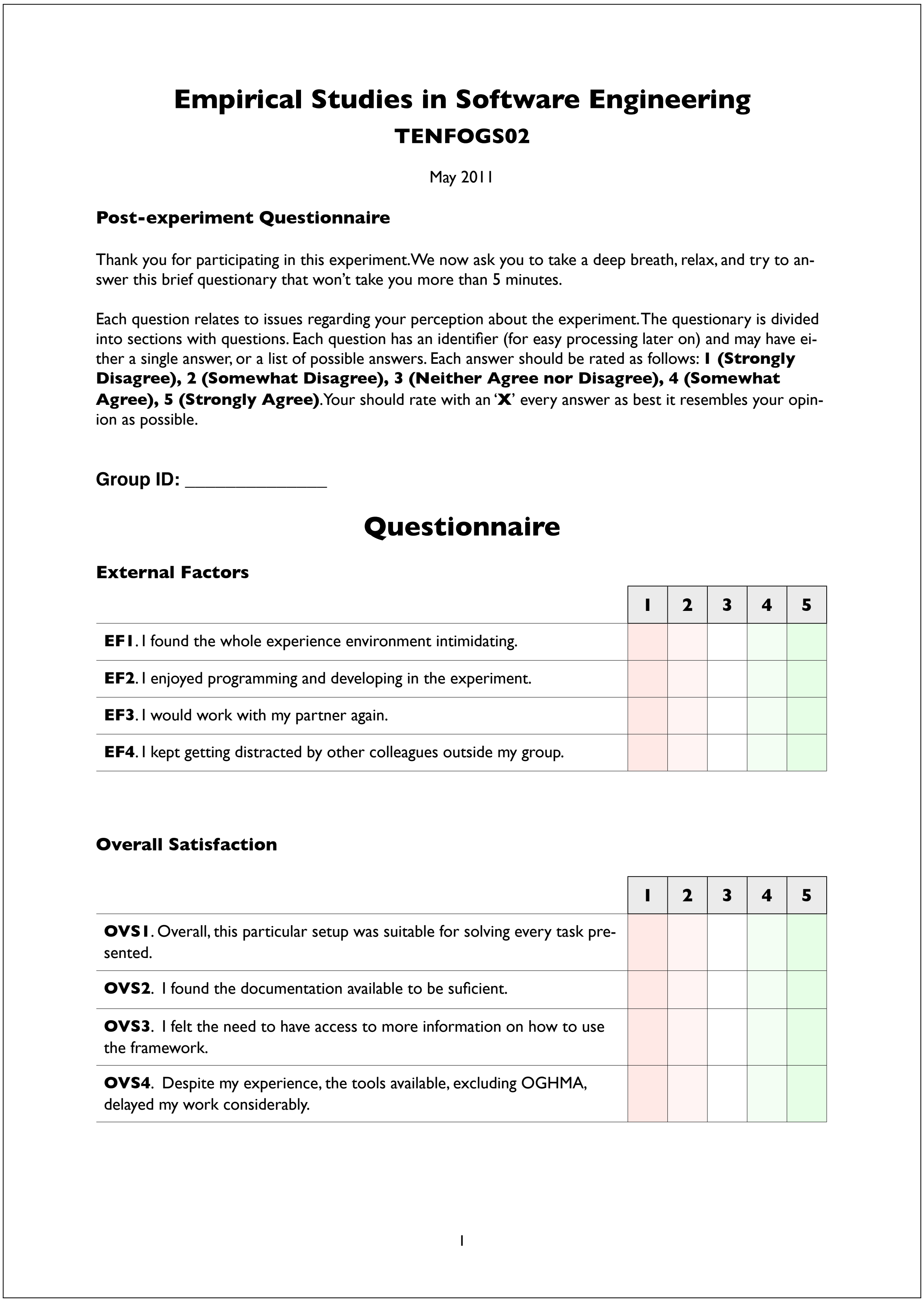
	Answers									$\bar{x}$	σ	=	Distances									$\bar{x}$	σ
AF1	3	4	5	5	3	5	4	4	4	4.11	0.78	5	2	1	0	0	2	0	1	1	1	0.89	1.49
AF2	2	2	3	4	3	4	4	5	2	3.22	1.09	1	1	1	2	3	2	3	3	4	1	2.22	1.10
AF3	3	3	5	4	1	2	2	1	4	2.78	1.39	5	2	2	0	1	4	3	3	4	1	2.22	1.58
AF4	2	1	5	5	2	1	2	5	1	2.67	1.80	1	1	0	4	4	1	0	1	4	0	1.67	1.71
AF5	2	1	3	3	2	3	1	1	3	2.11	0.93	1	1	0	2	2	1	2	0	0	2	1.11	0.88
AF6	1	2	1	1	2	1	4	3	1	1.78	1.09	1	0	1	0	0	1	0	3	2	0	0.78	1.03
AF7	5	4	5	5	3	5	4	5	5	4.56	0.73	5	0	1	0	0	2	0	1	0	0	0.44	1.60
AF8	3	4	4	4	3	4	4	4	3	3.67	0.50	5	2	1	1	1	2	1	1	1	2	1.33	1.25
AF9	3	2	4	4	3	3	3	4	5	3.44	0.88	5	2	3	1	1	2	2	2	1	0	1.56	1.37
AF10	3	4	4	4	4	5	2	3	4	3.67	0.87	5	2	1	1	1	1	0	3	2	1	1.33	1.42
AF11	3	1	3	3	4	3	3	3	3	2.89	0.78	1	2	0	2	2	3	2	2	2	2	1.89	0.79
AF12	3	4	4	4	4	5	3	2	2	3.44	1.01	1	2	3	3	3	3	4	2	1	1	2.44	1.06
AF13	4	4	3	3	4	3	3	4	4	3.56	0.53	5	1	1	2	2	1	2	2	1	1	1.44	1.23
AF14	3	3	3	3	3	4	3	3	3	3.11	0.33	1	2	2	2	2	2	3	2	2	2	2.11	0.47
AF15	4	3	3	3	3	3	3	3	4	3.22	0.44	5	1	2	2	2	2	2	2	2	1	1.78	1.10
AF16	3	4	5	5	4	3	3	5	4	4.00	0.87	5	2	1	0	0	1	2	2	0	1	1.00	1.51
AF17	4	4	3	3	3	3	3	3	4	3.33	0.50	5	1	1	2	2	2	2	2	2	1	1.67	1.15

Table E.1: Pre-experiment questionnaire B results for the *Experts* group. Left side of the table shows the answers, while the right side of the table shows computed distances to the correct answer (stated in column "="). See § 8.3.8 (p. 175). Background items were discarded due to its lower relevance to the experiment.

Appendix F

Post-questionnaire

The following is a copy of the anonymous questionnaire handed to all subjects at the end of the experiment.



Empirical Studies in Software Engineering

TENFOGS02

May 2011

Post-experiment Questionnaire

Thank you for participating in this experiment. We now ask you to take a deep breath, relax, and try to answer this brief questionary that won't take you more than 5 minutes.

Each question relates to issues regarding your perception about the experiment. The questionary is divided into sections with questions. Each question has an identifier (for easy processing later on) and may have either a single answer, or a list of possible answers. Each answer should be rated as follows: **1 (Strongly Disagree), 2 (Somewhat Disagree), 3 (Neither Agree nor Disagree), 4 (Somewhat Agree), 5 (Strongly Agree)**. Your should rate with an '**X**' every answer as best it resembles your opinion as possible.

Group ID: _______________

Questionnaire

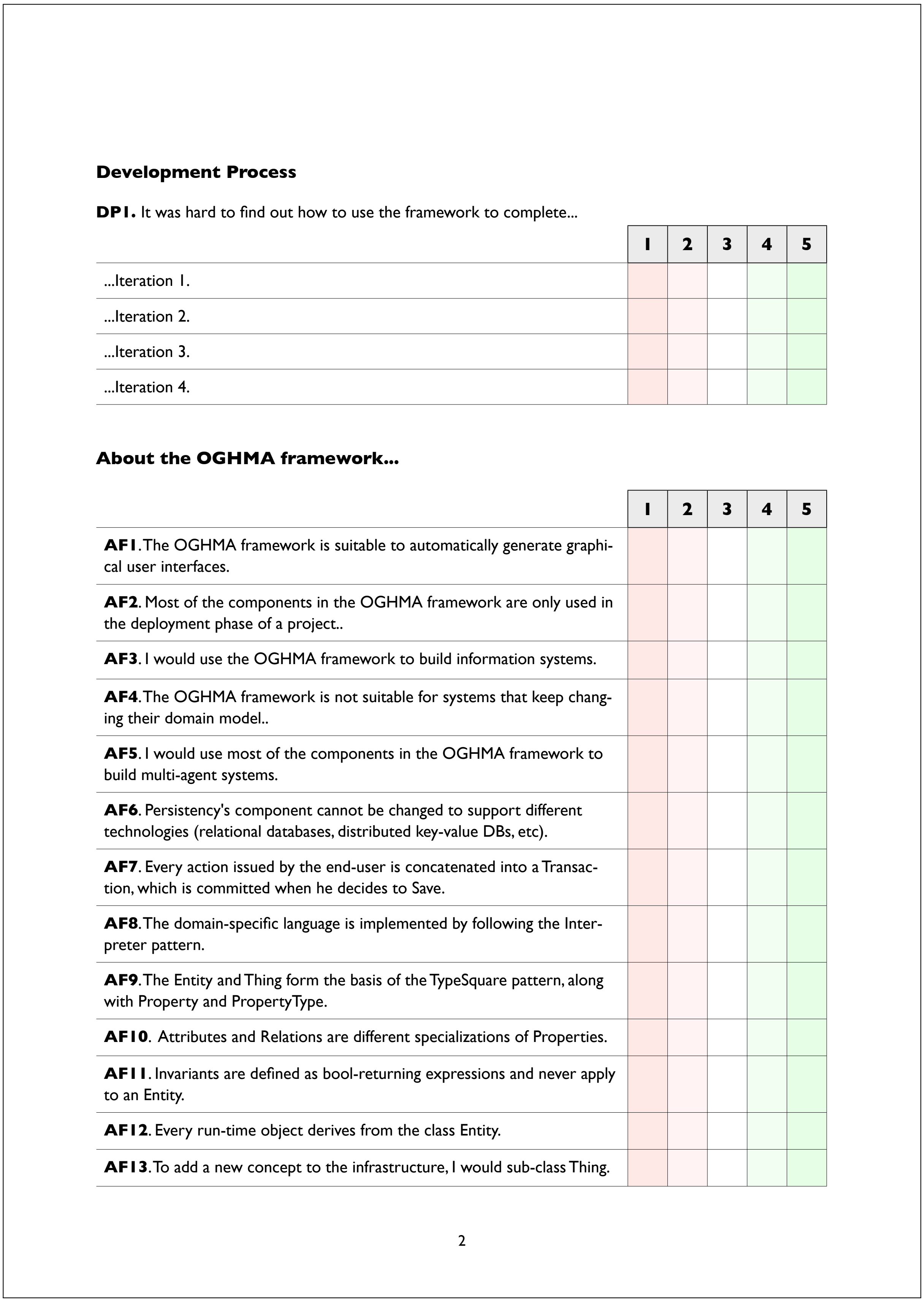
External Factors

	1	2	3	4	5
EF1. I found the whole experience environment intimidating.					
EF2. I enjoyed programming and developing in the experiment.					
EF3. I would work with my partner again.					
EF4. I kept getting distracted by other colleagues outside my group.					

Overall Satisfaction

	1	2	3	4	5
OVS1. Overall, this particular setup was suitable for solving every task presented.					
OVS2. I found the documentation available to be suficient.					
OVS3. I felt the need to have access to more information on how to use the framework.					
OVS4. Despite my experience, the tools available, excluding OGHMA, delayed my work considerably.					

1



Development Process

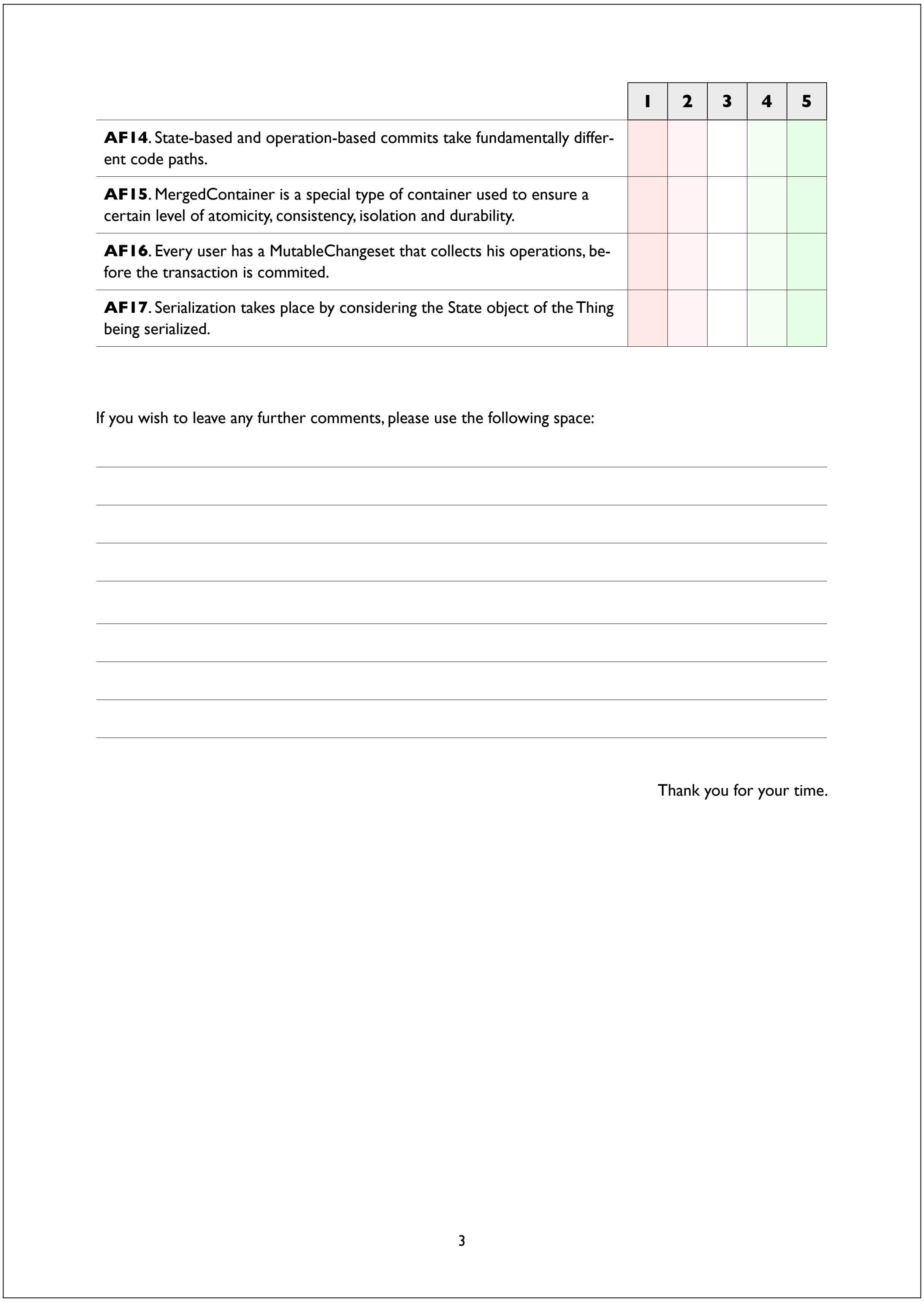
DP1. It was hard to find out how to use the framework to complete...

	1	**2**	**3**	**4**	**5**
...Iteration 1.					
...Iteration 2.					
...Iteration 3.					
...Iteration 4.					

About the OGHMA framework...

	1	**2**	**3**	**4**	**5**
AF1. The OGHMA framework is suitable to automatically generate graphical user interfaces.					
AF2. Most of the components in the OGHMA framework are only used in the deployment phase of a project..					
AF3. I would use the OGHMA framework to build information systems.					
AF4. The OGHMA framework is not suitable for systems that keep changing their domain model..					
AF5. I would use most of the components in the OGHMA framework to build multi-agent systems.					
AF6. Persistency's component cannot be changed to support different technologies (relational databases, distributed key-value DBs, etc).					
AF7. Every action issued by the end-user is concatenated into a Transaction, which is committed when he decides to Save.					
AF8. The domain-specific language is implemented by following the Interpreter pattern.					
AF9. The Entity and Thing form the basis of the TypeSquare pattern, along with Property and PropertyType.					
AF10. Attributes and Relations are different specializations of Properties.					
AF11. Invariants are defined as bool-returning expressions and never apply to an Entity.					
AF12. Every run-time object derives from the class Entity.					
AF13. To add a new concept to the infrastructure, I would sub-class Thing.					

2



	1	2	3	4	5
AF14. State-based and operation-based commits take fundamentally different code paths.					
AF15. MergedContainer is a special type of container used to ensure a certain level of atomicity, consistency, isolation and durability.					
AF16. Every user has a MutableChangeset that collects his operations, before the transaction is commited.					
AF17. Serialization takes place by considering the State object of the Thing being serialized.					

If you wish to leave any further comments, please use the following space:

Thank you for your time.

3

Appendix G

Post-Questionnaire answers

	Baseline						$\bar{x}$	σ	EG1				$\bar{x}$	σ	H_1	W	$\rho \neq$	$\rho <$	$\rho >$
EF1	4	5	3	3	4	1	3.33	1.37	2	2	4	4	3.00	1.15	$\neq$	20.0	0.762	0.338	0.695
EF2	2	3	4	4	3	2	3.00	0.89	3	4	2	3	3.00	0.82	$\neq$	22.0	1.000	0.652	0.653
EF3	5	5	5	5	3	4	4.50	0.84	4	4	5	5	4.50	0.58	$\neq$	21.0	0.914	0.548	0.666
EF4	1	1	1	1	1	1	1.00	0.00	2	1	2	1	1.50	0.58	$\neq$	27.0	0.256	0.133	1.000
OVS1	2	2	3	2	3	3	2.50	0.55	4	4	4	4	4.00	0.00	$>$	24.0	0.010	0.005	1.000
OVS2	1	1	3	1	2	1	1.50	0.84	3	4	2	2	2.75	0.96	$>$	29.5	0.067	0.043	1.247
OVS3	5	5	4	5	4	5	4.67	0.52	5	4	4	5	4.50	0.58	$<$	20.0	0.762	0.548	0.881
OVS4	1	3	3	4	2	2	2.50	1.05	2	1	2	2	1.75	0.50	$<$	16.5	0.257	0.190	0.953
DP1.1	1	1	1	1	4	2	1.67	1.21	2	4	2	2	2.50	1.00	$<$	26.0	0.171	0.067	0.971
DP1.2	1	1	1	1	3	2	1.50	0.84	2	1	2	2	1.75	0.50	$<$	29.5	0.476	0.262	0.833
DP1.3	5	5	4	5	4	2	4.17	1.17	5	4	3	3	3.75	0.96	$<$	18.5	0.476	0.262	0.795
DP1.4	5	5	5	5	5	5	5.00	0.00	5	5	5	5	5.00	0.00	$<$	22.0	1.000	1.000	1.000

Table G.1: Post-experiment questionnaire results for *Baseline* and *Experimental Group 1 (EG1)*, each line representing the data of a single question for both groups, with corresponding means and standard deviation values. It includes the values of the non-parametric significance Mann-Whitney-Wilcoxon test; see § 8.3 (p. 159).

	Baseline						$\bar{x}$	σ	EG2				$\bar{x}$	σ	H_1	W	$\rho \neq$	$\rho <$	$\rho >$
EF1	4	5	3	3	4	1	3.33	1.37	4	4	5	5	4.50	0.58	$\neq$	26.0	0.171	0.114	0.972
EF2	2	3	4	4	3	2	3.00	0.89	3	4	4	4	3.75	0.50	$\neq$	27.0	0.257	0.167	0.976
EF3	5	5	5	5	3	4	4.50	0.84	4	3	3	4	3.50	0.58	$\neq$	14.0	0.114	0.057	0.986
EF4	1	1	1	1	1	1	1.00	0.00	1	1	1	1	1.00	0.00	$\neq$	22.0	1.000	1.000	1.000
OVS1	2	2	3	2	3	3	2.50	0.55	4	3	4	3	3.50	0.58	$>$	14.5	0.670	0.048	1.000
OVS2	1	1	3	1	2	1	1.50	0.84	2	3	3	4	3.00	0.82	$>$	27.0	0.038	0.033	0.996
OVS3	5	5	4	5	4	5	4.67	0.52	5	4	5	4	4.50	0.58	$<$	20.0	0.762	0.548	0.881
OVS4	1	3	3	4	2	2	2.50	1.05	2	2	2	3	2.25	0.50	$<$	20.0	0.762	0.405	0.738
DP1.1	1	1	1	1	4	2	1.67	1.21	4	5	4	4	4.25	0.50	$<$	22.5	0.019	0.019	1.000
DP1.2	1	1	1	1	3	2	1.50	0.84	2	2	1	2	1.75	0.50	$<$	29.5	0.476	0.262	0.833
DP1.3	5	5	4	5	4	2	4.17	1.17	4	4	5	2	3.75	1.26	$<$	19.0	0.610	0.376	0.853
DP1.4	5	5	5	5	5	5	5.00	0.00	5	5	3	3	4.00	1.15	$<$	16.0	0.257	0.133	1.000

Table G.2: Post-experiment questionnaire results for *Baseline* and *Experimental Group 2 (EG2)*, each line representing the data of a single question for both groups, with corresponding means and standard deviation values. It includes the values of the non-parametric significance Mann-Whitney-Wilcoxon test; see § 8.3 (p. 159).

	Answers						$\bar{x}$	σ	=	Distances						$\bar{x}$	σ
AF1	5	4	4	4	5	4	4.33	0.52	5	0	1	1	1	0	1	1.75	0.52
AF2	4	4	2	3	4	3	3.33	0.82	1	3	3	1	2	3	2	2.33	0.82
AF3	4	3	3	4	2	2	3.00	0.89	5	1	2	2	1	3	3	2.75	0.89
AF4	1	1	3	2	2	4	2.17	1.17	1	0	0	2	1	1	3	1.13	1.17
AF5	3	4	2	2	3	2	2.67	0.82	1	2	3	1	1	2	1	1.50	0.82
AF6	1	1	2	2	2	3	1.83	0.75	1	0	0	1	1	1	2	0.88	0.75
AF7	5	3	3	4	4	5	4.00	0.89	5	0	2	2	1	1	0	2.00	0.89
AF8	5	3	3	3	4	2	3.33	1.03	5	0	2	2	2	1	3	2.50	1.03
AF9	3	3	3	5	3	3	3.33	0.82	5	2	2	2	0	2	2	2.50	0.82
AF10	3	3	3	4	5	4	3.67	0.82	5	2	2	2	1	0	1	2.25	0.82
AF11	3	3	3	3	3	3	3.00	0.00	1	2	2	2	2	2	2	1.75	0.00
AF12	3	3	3	4	3	2	3.00	0.63	1	2	2	2	3	2	1	1.75	0.63
AF13	5	3	3	4	3	2	3.33	1.03	5	0	2	2	1	2	3	2.50	1.03
AF14	4	3	3	3	4	3	3.33	0.52	1	3	2	2	2	3	2	2.00	0.52
AF15	3	3	3	3	3	3	3.00	0.00	5	2	2	2	2	2	2	2.75	0.00
AF16	3	3	3	4	3	3	3.17	0.41	5	2	2	2	1	2	2	2.63	0.41
AF17	3	3	3	4	3	4	3.33	0.52	5	2	2	2	1	2	1	2.50	0.52

Table G.3: Post-experiment questionnaire framework knowledge items results for *Baseline*. Left side of the table shows the answers, while the right side of the table shows computed distances to the correct answer (stated in column "="). See § 8.3.6 (p. 169)

	Answers				$\bar{x}$	σ	=	Distances				$\bar{x}$	σ
AF1	4	5	4	5	4.50	0.58	5	1	0	1	0	0.50	0.58
AF2	2	3	2	3	2.50	0.58	1	1	2	1	2	1.50	0.58
AF3	3	3	4	2	3.00	0.82	5	2	2	1	3	2.00	0.82
AF4	2	1	2	3	2.00	0.82	1	1	0	1	2	1.00	0.82
AF5	3	3	2	3	2.75	0.50	1	2	2	1	2	1.75	0.50
AF6	2	2	4	2	2.50	1.00	1	1	1	3	1	1.50	1.00
AF7	4	4	2	3	3.25	0.96	5	1	1	3	2	1.75	0.96
AF8	3	4	3	3	3.25	0.50	5	2	1	2	2	1.75	0.50
AF9	3	5	3	3	3.50	1.00	5	2	0	2	2	1.50	1.00
AF10	3	3	3	3	3.00	0.00	5	2	2	2	2	2.00	0.00
AF11	3	3	3	3	3.00	0.00	1	2	2	2	2	2.00	0.00
AF12	4	4	3	3	3.50	0.58	1	3	3	2	2	2.50	0.58
AF13	4	2	2	3	2.75	0.96	5	1	3	3	2	2.25	0.96
AF14	3	3	3	3	3.00	0.00	1	2	2	2	2	2.00	0.00
AF15	3	3	3	3	3.00	0.00	5	2	2	2	2	2.00	0.00
AF16	4	5	4	3	4.00	0.82	5	1	0	1	2	1.00	0.82
AF17	4	4	4	4	4.00	0.00	5	1	1	1	1	1.00	0.00

Table G.4: Post-experiment questionnaire framework knowledge items results for *Experimental Group 1.* Left side of the table shows the answers, while the right side of the table shows computed distances to the correct answer (stated in column "="). See § 8.3.6 (p. 169)

	Answers				$\bar{x}$	σ	=	Distances				$\bar{x}$	σ
AF1	4	5	4	4	4.25	0.50	5	1	0	1	1	0.75	0.50
AF2	4	4	4	4	4.00	0.00	1	3	3	3	3	3.00	0.00
AF3	4	3	4	4	3.75	0.50	5	1	2	1	1	1.25	0.50
AF4	3	3	3	3	3.00	0.00	1	2	2	2	2	2.00	0.00
AF5	2	3	2	3	2.50	0.58	1	1	2	1	2	1.50	0.58
AF6	3	2	2	3	2.50	0.58	1	2	1	1	2	1.50	0.58
AF7	3	3	3	4	3.25	0.50	5	2	2	2	1	1.75	0.50
AF8	4	3	4	5	4.00	0.82	5	1	2	1	0	1.00	0.82
AF9	4	4	3	4	3.75	0.50	5	1	1	2	1	1.25	0.50
AF10	3	3	3	3	3.00	0.00	5	2	2	2	2	2.00	0.00
AF11	3	3	1	3	2.50	1.00	1	2	2	0	2	1.50	1.00
AF12	3	3	3	3	3.00	0.00	1	2	2	2	2	2.00	0.00
AF13	3	3	3	3	3.00	0.00	5	2	2	2	2	2.00	0.00
AF14	3	3	3	3	3.00	0.00	1	2	2	2	2	2.00	0.00
AF15	3	3	3	3	3.00	0.00	5	2	2	2	2	2.00	0.00
AF16	3	3	3	3	3.00	0.00	5	2	2	2	2	2.00	0.00
AF17	3	4	5	4	4.00	0.82	5	2	1	0	1	1.00	0.82

Table G.5: Post-experiment questionnaire framework knowledge items results for *Experimental Group 2*. Left side of the table shows the answers, while the right side of the table shows computed distances to the correct answer (stated in column "="). See § 8.3.6 (p. 169)

	Answers									$\bar{x}$	σ	=	Distances									$\bar{x}$	σ
AF1	3	4	5	5	3	5	4	4	4	4.11	0.78	5	4	4	5	5	5	2	4	4	4	4.11	0.93
AF2	2	2	3	4	3	4	4	5	2	3.22	1.09	1	4	4	4	4	3	3	3	2	1	3.11	1.05
AF3	3	3	5	4	1	2	2	1	4	2.78	1.39	5	2	2	4	4	3	1	1	4	4	2.78	1.30
AF4	2	1	5	5	2	1	2	5	1	2.67	1.80	1	3	4	1	4	1	4	5	2	1	2.78	1.56
AF5	2	1	3	3	2	3	1	1	3	2.11	0.93	1	3	3	3	3	1	1	1	1	3	2.11	1.05
AF6	1	2	1	1	2	1	4	3	1	1.78	1.09	1	4	1	1	5	1	2	2	2	1	2.11	1.45
AF7	5	4	5	5	3	5	4	5	5	4.56	0.73	5	3	4	4	5	5	5	5	4	5	4.44	0.73
AF8	3	4	4	4	3	4	4	4	3	3.67	0.50	5	3	4	3	3	3	4	3	3	3	3.22	0.44
AF9	3	2	4	4	3	3	3	4	5	3.44	0.88	5	4	3	4	4	3	4	4	3	5	3.78	0.67
AF10	3	4	4	4	4	5	2	3	4	3.67	0.87	5	3	4	4	4	5	4	3	2	3	3.56	0.88
AF11	3	1	3	3	4	3	3	3	3	2.89	0.78	1	3	2	3	4	3	3	3	3	3	3.00	0.50
AF12	3	4	4	4	4	5	3	2	2	3.44	1.01	1	4	3	4	5	4	4	3	3	3	3.67	0.71
AF13	4	4	3	3	4	3	3	4	4	3.56	0.53	5	5	3	4	4	3	1	2	3	4	3.22	1.20
AF14	3	3	3	3	3	4	3	3	3	3.11	0.33	1	2	3	3	3	3	3	3	3	2	2.78	0.44
AF15	4	3	3	3	3	3	3	3	4	3.22	0.44	5	3	4	3	3	3	3	3	3	4	3.22	0.44
AF16	3	4	5	5	4	3	3	5	4	4.00	0.87	5	4	4	4	4	3	4	5	3	4	3.89	0.60
AF17	4	4	3	3	3	3	3	3	4	3.33	0.50	5	5	4	4	4	3	3	5	3	4	3.89	0.78

Table G.6: Post-experiment questionnaire framework knowledge items results for the *Experts* group. Left side of the table shows the answers, while the right side of the table shows computed distances to the correct answer (stated in column "="). See § 8.3.8 (p. 175).

Appendix H

Time and effectiveness results

	Iterations			
	1	2	3	4
Baseline Pair 1	B	B	A-	-
Baseline Pair 2	B	B	B-	-
Baseline Pair 3	B	A-	B	-
Experimental Group 1 Pair 1	A-	A	A-	-
Experimental Group 1 Pair 2	B	B	A	-
Experimental Group 2 Pair 1	B	B	B	-
Experimental Group 2 Pair 2	A-	B	A-	-
Experts Group Pair 1	A-	A-	A-	-
Experts Group Pair 2	A-	A-	A-	-
Experts Group Pair 3	A	A-	A	-
Experts Group Pair 4	A-	B	A-	-
Experts Group Pair 5	A	A	A	-

Table H.1: Deliverables grades, including the *Experts* group analysis.

	Iterations			
	1	2	3	4
Baseline Pair 1	69.2	44.5	79.3	-
Baseline Pair 2	86.8	48.3	64.5	-
Baseline Pair 3	136.1	25.5	43.6	-
Experimental Group 1 Pair 1	74.7	39.8	71.8	-
Experimental Group 1 Pair 2	58.4	47.7	21.5	-
Experimental Group 2 Pair 1	103.1	42.1	60.0	-
Experimental Group 2 Pair 2	85.3	37.1	60.4	-
Experts Group Pair 1	57.6	29.5	37.1	-
Experts Group Pair 2	63.3	31.3	50.5	-
Experts Group Pair 3	56.1	50.6	29.3	-
Experts Group Pair 4	34.9	45.3	30.5	-
Experts Group Pair 5	25.9	16.1	16.4	-

Table H.2: Iteration time results, including the *Experts* group analysis. Experts Group Pair 2 had only one element, due to the odd number of subjects from the *Experts* group. Values in minutes.

Glossary

.NET	Microsoft .NET framework for software solutions development [dot].
AJAX	Acronym for **A**syncronous **J**avascript **A**nd **X**ML. A group of interrelated web development methods used on the client-side to create asynchronous web applications [Gar05].
AOP	Acronym for **A**spect **O**riented **P**rogramming [KLM$^+$97].
API	Acronym for **A**pplication **P**rogramming **I**nterface.
CDE	Acronym for **C**ollaborative **D**evelopment **E**nvironment [BB03].
COBOL	Acronym for **CO**mmon **B**usiness-**O**riented **L**anguage. Its one of the oldest programming languages. Its primary domain is business, finance, and administrative systems for companies and governments..
CRUD	Acronym for **C**reate, **R**ead, **U**pdate, and **D**elete.
DSML	Acronym for **D**omain **S**pecific **M**odeling **L**anguage [DSM].
ESWS	Acronym for **E**mpirical **S**tudies **W**ith **S**tudents.
FORTRAN	A general-purpose, procedural, imperative programming language that is especially suited to numeric computation and scientific computing..
FOSD	Acronym for **F**eature-**O**riented **S**oftware **D**evelopment [AK09].
GUI	Acronym for **G**raphical **U**ser **I**nterface.
HTML	Acronym for **H**yper**T**ext **M**arkup **L**anguage.
IDE	Acronym for **I**ntegrated **D**evelopment **E**nvironment.
J2EE	Java Platform, Enterprise Edition. A platform for server programming in the Java programming language.
kLOC	Acronym for **k**ilo **L**ines **O**f **C**ode — effectively thousands of LOC.
LOC	Acronym for **L**ines **O**f **C**ode.
MVC	Acronym for **M**odel-**V**iew-**C**ontroller [KP88].
OO	Acronym for **O**bject-**O**riented.
PHP	Acronym for **H**ypertext **P**re**P**rocessor. A general-purpose server-side scripting language originally designed for web development to produce dynamic web pages..
QWAN	Acronym for **Q**uality **W**ithout **A** **N**ame [Ale79].
SDK	Acronym for **S**oftware **D**evelopment **K**it.
UML	Acronym for **U**nified **M**odeling **L**anguage [OMG11].

URI	Acronym for **U**niform **R**esource **I**dentifier.
VCS	Acronym for **V**ersion **C**ontrol **S**ystem.
XML	Acronym for e**X**tended **M**arkup **L**anguage.

References

[AC06] M. Antkiewicz and K. Czarnecki, *Framework-specific modeling languages with round-trip engineering*, Proc. of the Model Driven Engineering Languages and Systems, 2006, pp. 692–706. Cited on pp. 49 and 50.

[Ack00] M. Ackerman, *The intellectual challenge of cscw: The gap between social requirements and technical feasibility*, Human-Computer Interaction (2000), no. 15, 179–203. Cited on pp. 55 and 59.

[AD05a] A. Aguiar and G. David, *Patterns for documenting frameworks – part i*, VikinPLoP'2005 (Helsinki, Finland), September 2005. Cited on pp. 94, 96, 97, 98, 105, and 169.

[AD05b] ———, *Wikiwiki weaving heterogeneous software artifacts*, Proceeding of the WikiSym'05 - International Symposium on Wikis, 2005, pp. 67–74. Cited on p. 124.

[AD06a] ———, *Patterns for documenting frameworks – part ii*, EuroPLoP'2006 (Irsee, Germany), July 2006. Cited on pp. 90, 94, 96, 97, 98, and 123.

[AD06b] ———, *Patterns for documenting frameworks – part iii*, PLoP'2006 (Portland, Oregon, USA), October 2006. Cited on pp. 90, 92, 94, 96, 97, 100, 102, and 103.

[AD11] Ademar Aguiar and Gabriel David, *Patterns for effectively documenting frameworks*, Transactions on pattern languages of programming II (James Noble and Ralph Johnson, eds.), Springer-Verlag, Berlin, Heidelberg, 2011, pp. 79–124. Cited on pp. 50, 85, 86, and 125.

[AG08] R. Ahuja and A. Goel, *A study of the effect of logging using aop*, The 2008 International Conference on Software Engineering Research and Practive (SERP'08), 2008. Cited on p. 23.

[Agu03] A. Aguiar, *Framework documentation – a minimalist approach*, Ph.D. thesis, FEUP, September 2003. Cited on pp. 2, 3, 4, 5, 37, 76, 97, and 124.

[AIS77] C. Alexander, S. Ishikawa, and M. Silverstein, *A pattern language: Towns, buildings, construction*, Oxford University Press, 1977. Cited on pp. 47, 73, 82, and 83.

[AK03] C. Atkinson and T. Kuhne, *Model-driven development: a metamodeling foundation*, IEEE Software **20(5)** (2003), 36–41. Cited on p. 24.

[AK09] S. Apel and C. Kastner, *An overview of feature-oriented software development*, Journal of Object Technology **8(5)** (2009), 49–84. Cited on pp. 23 and 213.

[AKT07] S. Apel, C. Kastner, and S. Trujillo, *On the necessity of empirical studies in the assessment of modularization mechanisms for crosscutting concerns*, 1st International Workshop on Assessment of Contemporary Modularization Techniques, ICSE Workshops, ACoM'07, 2007. Cited on p. 23.

[Ale79] *The timeless way of building*, Oxford University Press, 1979. Cited on pp. 82, 186, and 213.

[Alf] *Alfresco official site*, http://www.alfresco.com/ [Online; accessed September 2011]. Cited on pp. 34 and 142.

[And04] J. Anderson, *Cognitive psychology and its implications*, Worth Publishers, 2004. Cited on pp. 48 and 114.

[APP04] S. I. Ahamed, A. Pezewski, and A. Pezewski, *Towards framework selection criteria and suitability for an application framework*, Proceedings of the international Conference on information Technology: Coding and Computing (Itcc'04), vol. 1, 2004, pp. 424–428. Cited on pp. 88 and 99.

[ASFF11] P. Alves, A. Santos, E. Figueiredo, and F. Ferrari, *How do programmers learn aop?*, Latin American Workshop on Aspect-Oriented Software Development, 2011. Cited on p. 23.

[Bas] *Project management software, online collaboration: Basecamp*, http://basecamphq.com [Online; accessed September 2011]. Cited on p. 62.

[BB03] G. Booch and A. W. Brown, *Collaborative development environments*, Advances in Computers **59** (2003), 2–29. Cited on pp. 60, 62, and 213.

[BBvB$^+$] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas, *Manifesto for agile software development*, http://agilemanifesto.org [Online; accessed August 2011]. Cited on p. 23.

[BE96] T. Ball and S. G. Eick, *Software visualization in the large*, IEEE Computer **29** (1996), 33–43. Cited on p. 33.

[BE98] B. Boehm and A. Egyed, *Software requirements negotiation: Some lessons learned*, 20th International Conference on Software Engineering (ICSE'98) (Japan), 1998, pp. 503–507. Cited on p. 61.

[Ben05] J. Bentley, *Laziness, impatience, hubris: Personality traits of a great programmer*, Proceedings of the 13th Anual SouthEast SAS Users Groups (SESUG) Conference, 2005, pp. 1–9. Cited on p. 24.

[Ber] Joseph Bergin, *Patterns for the doctoral student*, Pace University, http://csis.pace.edu/~bergin/patterns/DoctoralPatterns.html[Online;accessed October 2011]. Cited on p. 185.

[BGS92] M. S. K. Brade, M. Guzdial, and E. Soloway, *Whorf: A visualization tool for software maintenance*, Proceedings 1992 IEEE Workshop on Visual Languages, 1992, pp. 148–154. Cited on p. 31.

[BHS07] F. Buschmann, K. Henney, and D. Schmidt, *Pattern oriented software architecture - volume5: On patterns and pattern languages*, vol. 5, John Wiley and Sons, 2007. Cited on p. 184.

[BK01] S. Bassil and R.K. Keller, *Software visualization tools: survey and analysis*, Proceedings of the 9th International Workshop on Program Comprehension (IWPC), 2001. Cited on p. 31.

[BKM00] G. Butler, R. Keller, and H. Mili, *A framework for framework documentation*, ACM Computing Surveys **32** (2000). Cited on pp. 4, 39, and 76.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila, *The semantic web*, Scientific American Magazine (2001). Cited on p. 24.

[BLS05] M. Barnett, K. Leino, and W. Schulte, *The spec programming systenm: An overview*, Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, 2005, pp. 49–69. Cited on p. 23.

[BM02a] R. Braga and P. Masiero, *A process for framework construction based on a pattern language*, Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC)., 2002, pp. 615–620. Cited on pp. 47 and 48.

[BM02b] ______, *The role of pattern languages in the instantiation of object-oriented frameworks*, Advances in Object-Oriented Information Systems (2002), 403–410. Cited on p. 48.

[BM03] ______, *Building a wizard for framework instantiation based on a pattern language*, Object-oriented Information Systems, 2003, pp. 95–106. Cited on pp. 47 and 50.

[BMMB99] J. Bosch, P. Molin, M. Mattsson, and P.O. Bengtsson, *Framework – problems and experiences*, Building Application Frameworks, M.Fayad, D.Schmidt, R.Johnson, Wiley, 1999. Cited on p. 37.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern oriented software architecture - a system of patterns*, John Wiley and Sons, 1996. Cited on pp. 23, 37, 100, and 101.

[BMW93] T. J. Biggerstaff, B. W. Mitbander, and D. Webster, *The concept assignment problem in program understanding"*, Proceedings of the 15th International Conference on Software Engineering, 1993, pp. 482–498. Cited on p. 26.

[BNR+03] C. Boldyreff, D. Nutter, S. Rank, M. Smith, P. Wilcox, and R. Dewar, *Enviroments to support collaborative software engineering*, Proc of the 2nd Workshop on Cooperative Supports for Distributed Software Engineering Processes, 2003, pp. 25–28. Cited on p. 60.

[Boo94] G. Booch, *Designing an application framework*, Dr.Dobb's Journal **19(2)** (1994). Cited on p. 4.

[Bra95] J .M. Brant, *Hotdraw*, Master's thesis, University of Illinois, 1995. Cited on p. 45.

[Bro83] R. Brooks, *Towards a theory of the comprehension of computer programs*, International Journal of Man-Machine Studies (1983), 543–554. Cited on p. 19.

[BS93] C.F. Bertholf and J. Scholtz, *Program comprehension of literate programs by novice programmers*, Empirical Studies of Programmers: 5th Workshop, 1993. Cited on p. 29.

[BS94] I. Z. Ben-Shaul, *Oz: A decentralized process centered environment*, Ph.D. thesis, Department of Computer Science: Columbia University, December 1994. Cited on p. 62.

[BS00] D. Brugali and K. Sycara, *Frameworks and pattern languages: an intriguing relationship*, ACM Computing Surveys (CSUR) **32 (1ed)** (2000), 2. Cited on p. 47.

[BS05] R. I. Bull and M-A. Storey, *Towards visualization support for the eclipse modeling framework*, A Research-Industry Technology Exchange at EclipseCon, 2005. Cited on p. 34.

[BSKH92] I. Z. Ben-Shaul, G. E. Kaiser, and G. T. Heineman, *An architecture for multi-user software development environments*, ACM SIGSOFT 92: 5th Symposium on Software Development Environments (Tyson's Corner, Virginia), 1992, pp. 149–158. Cited on p. 62.

[BSM06] M. Bruch, T. Schäfer, and M. Mezini, *Fruit: Ide support for framework understanding*, OOPSLA Eclipse Technology Exchange, 2006. Cited on pp. 45 and 104.

[BT96] G. A. Bolcer and R. N. Taylor, *Endeavors: a process system integration infrastructure*, 4th International Conference on the Software Process (ICSP'96) (Brighton, UK), 1996, pp. 76–89. Cited on pp. 61 and 62.

[Bug] *The bugzilla guide - 4.1.2 development release*, `http://www.bugzilla.org/docs/tip/en/pdf/Bugzilla-Guide.pdf` [Online; accessed August 2011]. Cited on p. 61.

[But98] G. Butler, *A reuse case perspective on documenting frameworks*, APSEC '98 Proceedings of the Fifth Asia Pacific Software Engineering Conference, 1998. Cited on pp. 4, 70, 76, and 85.

[CDMS02] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider, *Source transformation in software engineering using the txl transformation system*, Journal of Information and Software Technology **(44)13** (2002), 827–837. Cited on p. 34.

[CFL06] M. Cortés, M. Fontoura, and C. Lucena, *Framework evolution tool*, Journal of Object Technology **5(8)** (2006), 101–124. Cited on p. 93.

[CHSV97] W. Codenie, K. Hondt, P. Steyaert, and A. Vercammen, *From custom applications to domain-specific frameworks.*, Communications of the ACM **40(10)** (1997), 71–77. Cited on p. 92.

[CJMS10] J. C. Carver, L. Jaccheri, S. Morasca, and F. Shull, *A checklist for integrating student empirical studies with research and teaching goals*, Empirical Software Engineering **15(1)** (2010). Cited on pp. 148, 151, and 152.

[CLNC01] S. Chan, P. Lee, V. Ng, and A. Chan, *Syncronous collaborative development of uml models on the internet*, Concurrent Engineering **9 (2)** (2001), 111–119. Cited on p. 61.

[CMSB05] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, *Hipikat: A project memory for software development*, IEEE Transactions on Software Engineering **31** (2005), 446–465. Cited on p. 33.

[Cod] *Codase - source code search*, `http://www.codase.com/` [Online; accessed September 2011]. Cited on p. 33.

[Col] *Collabnet official site*, `http://www.collab.net/` [Online;accessed September 2011]. Cited on p. 34.

[Col95] K. Coleman, *Groupware technology and applications*, Prentice Hall, 1995. Cited on p. 54.

[Con] *Confluence site*, `http://www.atlassian.com/software/confluence/overview` [Online; accessed September 2011]. Cited on p. 142.

[Cop96] *Software patterns*, SIGS, 1996. Cited on p. 82.

[Cora] IBM Corporation, *Ibm software - doors product line*, `http://www-01.ibm.com/software/awdtools/doors/productline/` [Online; accessed August 2011]. Cited on p. 61.

[Corb] Imagix Corporation, *Imagix - source code analysis*, `http://www.imagix.com` [Online; accessed August 2011]. Cited on p. 31.

[Corc] Microsoft Corporation, *Office.com*, `http://office.microsoft.com/` [Online; accessed August 2011]. Cited on p. 62.

[Cord] ______, *The official site of visual studio 2010*, `http://www.microsoft.com/visualstudio/en-gb/` [Online; accessed August 2011]. Cited on pp. 23 and 137.

[Core] Rational Software Corporation, *Rational method composer*, `http://www-01.ibm.com/software/awdtools/rmc/` [Online; accessed August 2011]. Cited on p. 62.

[Cor89] E.S. Cordingley, *Knowledge elicitation principles, techniques and applications*, ch. Knowledge acquisition techniques for knowledge-based systems, Ellis Horwood Limited, 1989. Cited on p. 114.

[Cre11] *Creately - online diagramming and design*, 2011, http://creately.com/ [Online; accessed September 2011]. Cited on p. 61.

[Cun] Ward Cunningham, *Wikiwikiweb*, http://c2.com/cgi/wiki [Online;accessed August 2011]. Cited on p. 124.

[CW01] C.L. Corritore and S. Wiedenbeck, *An exploratory study of program comprehension strategies of procedural and object-oriented programmers*, International Journal of Huma-Computer Studies (2001). Cited on p. 22.

[CZvD$^+$09] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, *A systematic survey of program comprehension through dynamic analysis*, IEEE Transactions on Software Engineering **35(5)** (2009), 684–702. Cited on p. 32.

[Dav98] *Working knowledge: How organizations manage what they know*, Harvard Business School Press, 1998. Cited on p. 114.

[DB92] P. Dourish and V. Bellotti, *Awareness and coordination in shared workspaces*, Proceedings of the 1992 ACM conference on Computer-supported cooperative work (New York, NY, USA) (ACM Press, ed.), 1992. Cited on pp. 56 and 62.

[Dét01] F. Détienne, *Software design – cognitive aspects*, Springer Practitioner Series, 2001. Cited on pp. 18, 22, and 25.

[Deu89] L. P. Deutsch, *Design reuse and frameworks in the smalltalk-80 system*, Software reusability: vol. 2, applications and experience **2** (1989), 57–71. Cited on p. 2.

[DK99] S. Dossick and G. Kaiser, *Chime: a metadata-based distributed software development environment*, Proceedings of the 7th European Software Engineering conference help jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 1999, pp. 464–475. Cited on p. 60.

[DKCR05] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson, *Towards understanding programs through wear-based filtering*, Softvis, 2005. Cited on pp. 24 and 33.

[DL06] Esther Derby and Diana Larsen, *Agile retrospectives: Making good teams great*, Pragmatic Bookshelf, 2006. Cited on p. 23.

[D.N04] D.N.Sturtz, *Communal categorization: The folksonomy*, Essay for a Library and Information Science course at Drexel University., December 2004. Cited on p. 121.

[dok] *Dokuwiki*, http://www.dokuwiki.org [Online; accessed August 2011]. Cited on p. 124.

[dot] *Microsoft .net website*, http://www.microsoft.com/net [Online; accessed September 2011]. Cited on p. 213.

[Dou03] P. Dourish, *The appropriation of interactive technologies: Some lessons from placeless documents*, Computer Supported Cooperative Work (Kluwer Academic Publishers, ed.), vol. 12, 2003, pp. 465–490. Cited on p. 56.

[Dro11] *Dropbox - simplify your file (website)*, 2011, http://www.dropbox.com [Online; accessed September 2011]. Cited on p. 62.

[Dru] *Drupal - open source cms*, http://drupal.org [Online; accessed September 2011]. Cited on p. 142.

[dSCdW$^+$06] I. A. da Silva, P. H. Chen, C. V. der Westhuizen, R. M. Ripley, and A. van der Hoek, *Lighthouse: Coordination through emerging design*, Proc. of the 2006 OOPSLA workshop on eclipse technology eXchange, 2006. Cited on p. 62.

[DSE06] M. Desmond, M-A. Storey, and C. Exton, *Fluid source code views for just in-time comprehension*, Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT'06), 2006. Cited on p. 31.

[DSM] *Dsm forum : Domain specific modeling*, `http://www.dsmforum.org/` [Online; accessed September 2011]. Cited on pp. 49 and 213.

[Dus03] L. Dusseault, *Webdav: Next-generation collaborative web authoring*, Prentice Hall PTR, 2003. Cited on p. 62.

[ea95] R. Baecker et al., *Readings in human-computer interaction: toward the year 2000*, Morgan Kaufmann Publishers, 1995. Cited on pp. 31 and 57.

[Ecl] *Eclipse uml plug-in*, `http://www.visual-paradigm.com/solution/eclipseuml/` [Online; accessed September 2011]. Cited on p. 33.

[Ecl11] *Eclipse project*, August 2011, `http://www.eclipse.org` [Online; accessed August 2011]. Cited on pp. 23, 32, and 137.

[EGR93] C. A. Ellis, S. J. Gibbs, and G. L. Rein, *Groupware some issues and experiences*, Readings in groupware and computer-supported cooperative work, Baecker, Morgan Kaufmann, San Francisco, 1993, pp. 9–28. Cited on pp. 54 and 55.

[End95] M. Endsley, *Toward a theory of situation awareness in dynamic systems*, Human Factors **(37)1** (1995), 32–64. Cited on pp. 58 and 62.

[Ent] *Entreprise architect*, `http://www.sparxsystems.com/` [Online; accessed September 2011]. Cited on p. 33.

[ES98] K. Erdös and H. M. Sneed, *Partial comprehension of complex programs (enough to perform maintenance)*, Proceedings of the 6th International Workshop on Program Comprehension, 1998, pp. 98–105. Cited on p. 27.

[ESJ92] S. G. Eick, J. L. Steffen, and E. E. Sumner Jr., *Seesoft – a tool for visualizing line oriented software statistics*, IEEE Transactions on Software Engineering **(28)4** (1992), 396–412. Cited on p. 62.

[ESS11] *http://softeng.fe.up.pt/essewiki*, August 2011, `http://softeng.fe.up.pt/esseWiki` [Online; accessed August 2011]. Cited on p. 148.

[FA08] N. Flores and A.Aguiar, *Patterns for framework understanding*, 15th Pattern Languages of Programming Conference (PLoP'08) (Nashville, USA), October 2008. Cited on p. 107.

[FA10] N. Flores and A. Aguiar, *Understanding frameworks collaboratively : Tool requirements*, International Journal on Advances on Software **vol. 3** (2010). Cited on p. 122.

[Fav01] J.-M. Favre, *Gsee: A generic software exploration enviroment.*, Proceedings of the 9th International Workshop on Program Comprehension (IWPC), 2001, pp. 233–244. Cited on p. 31.

[FD04] J. Froehlich and P. Dourich, *Unifying artifacts and activities in a visual tool for distributed software development teams*, Proceedings of the 26th International Conference on Software Engineering, 2004, pp. 387–396. Cited on p. 34.

[Fer11] H. Ferreira, *Adaptive object-modelling: Patterns, tools and applications*, Ph.D. thesis, University of Porto, Faculty of Engineering, 2011. Cited on pp. 151 and 155.

[FGS06] G. Fairbanks, D. Garlan, and W. Scherlis, *Design fragments make using frameworks easier*, OOPSLA, 2006. Cited on pp. 46 and 104.

[FHLS97] G. Froehlich, H. Hoover, L. Lui, and P. Sorenson, *Hooking into object-oriented application frameworks*, Proceedings of the 19th International Conference on Software Engineering, 1997, pp. 491–501. Cited on pp. 37, 45, 90, and 103.

[FHLS00] G. Froehlich, H. Hoover, L. Lui, and P. Sorenson, *Choosing an object-oriented domain framework*, ACM Computing Surveys **32 (1)** (2000). Cited on p. 88.

[FKAL09] J. Feigenspan, C. Kastner, S. Apel, and T. Leich, *How to compare program comprehension in fosd empirically - an experience report*, Proc. Int'l Workshop on Feature-Oriented Software Development, 2009, pp. 55–62. Cited on p. 23.

[Flo06] Nuno Flores, *Engenharia reversa de padrões em arquitecturas reutilizáveis*, Master's thesis, Faculty of Engineering, 2006. Cited on pp. xvi and 71.

[Fos] Sam Walter Foss, *The calf-path*, http://holyjoe.org/poetry/foss3.htm [Online; accessed August 2011]. Cited on p. 117.

[FPR01] M. Fontoura, W. Pree, and B. Rumpe, *The uml profile for framework architectures*, Addison-Wesley Professional, 2001. Cited on pp. 48 and 91.

[FS88] R. Felder and L. K. Silverman, *Learning and teaching styles in engineering education*, Engineering Education **78(7)** (1988), 674–681. Cited on p. 94.

[FS97] M. Fayad and D. Schmidt, *Object-oriented application frameworks*, Communications of the ACM **40(10)** (1997), 32–38. Cited on p. 2.

[FS05] R. Felder and J. Spurlin, *Applications, reliability, and validity of the index of learning styles*, International Journal of Engineering Education **21(1)** (2005), 103–112. Cited on pp. 6, 50, and 93.

[FSJ99] M. Fayad, D. Schimdt, and R. Johnson, *Building application frameworks*, Wiley, 1999. Cited on pp. 37, 39, 86, and 90.

[Fua07] M. Muztaba Fuad, *An autonomic software architecture for distributed applications*, Tech. report, 2007. Cited on p. 23.

[GA07] M. Goulao and F. Brito Abreu, *Modeling the experimental software engineering process*, QUATIC'07: Proceedings of the 6th International Conference on Quality of Information and Communications Technology (Washington, DC, USA) (IEEE Computer Society, ed.), 2007, pp. 77–90. Cited on p. 79.

[Gar05] J. J. Garrett, *Ajax: A new approach to web applications*, February 2005, http://www.adaptivepath.com/publications/essays/archives/000385.php [Online; accessed August 2011]. Cited on pp. 23 and 213.

[GB04] N. Gold and K. Bennett, *Program comprehension for web services*, Proceedings of the 12th IEEE International Workshop on Program Comprehension, 2004. Cited on p. 22.

[GCS05] D.M. German, D. Cubranic, and M-A. Storey, *A framework for describing and understanding mining tools in software development*, ACM SIGSOFT Software Engineering Notes **30(4)** (2005), 1–5. Cited on p. 30.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns — elements of reusable object-oriented software*, Addison-Wesley, 1995. Cited on pp. 23, 37, 44, 46, 50, 83, 102, and 103.

[GHL$^+$11] A. Guzzi, L. Hattori, M. Lanza, M. Pinzger, and A. van Deursen, *Collective code bookmarks for program comprehension*, Proceedings of the 19th International Conference on Program Comprehension, 2011, pp. 101–110. Cited on p. 34.

[GM95] D. Gangopadhyay and S. Mitra, *Understanding frameworks by exploration of exemplars*, Proceedings of CASE-95 (IEEE Computer Society, ed.), 1995, pp. 90–99. Cited on p. 37.

[GP92] T.R.G. Green and M. Petre, *When visual programs are harder to read than textual programs*, Human-Computer Interaction: Tasks and Organization, Proceedings (ECCE)-6 (6th European Conference Cognitive Ergonomics), 1992. Cited on p. 31.

[Gre88] I. Greif, *Computer-supported cooperative work: a book of readings*, Morgan Kaufmann Publishers, 1988. Cited on p. 55.

[Gre91] *Computer-supported co-operative work and groupware*, Academic Press Ltd., London, 1991. Cited on p. 55.

[Gri99] R. Grinter, *Systems architecture: Product designing and social engineering*, ACM Conference on Work Activities Coordination and Collaboration (WACC'99) (San Francisco, California), 1999, pp. 11–18. Cited on p. 58.

[GRS04] C. Gutwim, R.Penner, and K. Schneider, *Group awareness in distributed software development*, ACM CSCW, 2004, pp. 72–81. Cited on p. 26.

[Gru88] J. Grudin, *Why cscw applications fail: problems in the design and evaluation of organization of organizational interfaces*, Proceedings of the 1988 ACM conference on Computer-supported cooperative work (New York, NY, USA) (ACM Press New York, ed.), 1988. Cited on p. 56.

[Gru93] T. Gruber, *A translation approach to portable ontology specifications*, Knowledge Acquisition **5(2)** (1993), 199–220. Cited on p. 183.

[Gru94] J. Grudin, *Computer-supported co-operative work: History and focus*, Computer **(27)5** (1994). Cited on p. 54.

[Gru07] T. Gruber, *Collective knowledge systems: Where the social web meets the semantic web.*, Journal of Web Semantics (2007). Cited on pp. 115 and 116.

[HCRP04] S. Hupfer, L.-T Cheng, S. Ross, and J. Patterson, *Introducing collaboration into an application development environment*, Proceedings of the ACM Conference on Computer Supported Cooperative Work, 2004, pp. 444–454. Cited on pp. 34 and 62.

[HH01] D. Hou and H. J. Hoover, *Towards specifying constraints for object-oriented frameworks*, Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research (IBM Press, ed.), 2001, p. 5. Cited on p. 49.

[HHG90] R. Helm, I. Holland, and D. Gangopadhyay, *Contracts: specifying behavioral compositions in object-oriented systems*, Proceedings of the European conference on object-oriented programming (ECOOP'90), 1990, pp. 169–180. Cited on p. 49.

[HIBK97] T. Hendrix, J.H. Cross II, L. Barowski, and K.Mathias, *Tool support for reverse engineering multi-lingual software*, Proceedings of the 4th Working Conference on Reverse Engineering (WCRE'97), 1997, pp. 136–143. Cited on p. 31.

[HK06] J. Hautamäki and K. Koskimies, *Finding and documenting the specialization interface of an application framework*, Software-Practice and Experience **36(13)** (2006), 1443–1465. Cited on pp. 48 and 50.

[HL11] D. Hou and L. Li, *Obstacles in using frameworks and apis: An exploratory study of programmers' newsgroups discussions*, IEEE 19th International Conference in Program Comprehension (ICPC), 2011. Cited on p. 42.

[HOB05] A. R. Haydarlou, B. J. Overeinder, and F. M. T. Brazier, *A self-healing approach for object-oriented applications*, Proceedings of the 3rd International Workshop on Self-Adaptive and Autonomic Computing Systems, 2005, pp. 191–195. Cited on p. 23.

[Hoh96] *Journey of the software professional: The sociology of software development*, Prentice Hall, 1996. Cited on p. 18.

[Hou08] D. Hou, *Investigating the effects of framework design knowledge in example-based framework learning*, IEEE International Conference on Software Maintenance, 2008, pp. 37–46. Cited on pp. 42 and 94.

[HW99] M. Hollander and D. A. Wolfe, *Nonparametric statistical methods*, Wiley-Interscience, January 1999. Cited on p. 159.

[HWH05] D. Hou, K. Wong, and H. J. Hoover, *What can programmer questions tell us about frameworks?*, Proceedings of the 13th International Workshop on Program Comprehension (IPWC'05), 2005, pp. 87–96. Cited on p. 41.

[IK02] I.Hammouda and K.Koskimies, *A pattern-based j2ee application development environment*, Nordic Journal of Computing **(3)9** (2002), 248–260. Cited on p. 44.

[Ima] *Imagix4d official website*, `http://www.imagix.com` [Online; accessed September 2011]. Cited on p. 33.

[ISO87] S. Isoda, T. Shimomura, and Y. Ono, *Vips: A visual debugger*, IEEE Software (1987). Cited on p. 31.

[JF88] R. E. Johnson and B. Foote, *Designing reusable classes*, Journal of Object-Oriented Programming **1(2)** (1988), 22–35. Cited on p. 2.

[Jir] *Jira official website*, `http://www.atlassian.com/software/jira/overview` [Online; accessed September 2011]. Cited on p. 34.

[JK02] J.Hannemann and G. Kiczales, *Design pattern implementation in java and aspectj*, Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications, 2002, pp. 161–173. Cited on p. 44.

[JLJL90] P. Johnson-Lenz and T. Johnson-Lenz, *Rhythms, boundaries, and containers: Creative dynamics of asyncronous group life*, Research Report 4, Awakening Technology, April 1990. Cited on p. 54.

[JN99] E. E. Jacobson and P. Nowack, *Frameworks and patterns: Architectural abstractions*, Building Application Frameworks, John Wiley and Sons, 1999, pp. 29–54. Cited on pp. 101 and 104.

[Joh88] *Groupware: Computer support for business teams*, The Free Press, 1988. Cited on pp. 54 and 57.

[Joh92] R. Johnson, *Documenting frameworks using patterns*, Proceedings of the OOPSLA'92, SIGPLAN notices, vol. 27(10), 1992, pp. 63–76. Cited on pp. 37, 44, 47, and 50.

[Joh97] ———, *Components, framework, patterns*, SIGSOFT Software Engineering Notes **22(3)** (1997), 10–17. Cited on pp. 38, 42, and 45.

[Joo] *Joomla!*, `http://joomla.org` [Online; accessed September 2011]. Cited on p. 142.

[Kad92] R. Kadia, *Issues encountered in building a flexible software development environment*, ACM SIGSOFT 92: 5th Symposium on Software Development Environments (Tyson's Corner, Virginia), 1992, pp. 169–180. Cited on p. 62.

[KAKB+08] B. Kitchenham, H. Al-Khilidar, M. Ali Babar, M. Berry, K. Cox, J. Keung, F. Kurniawati, M. Staples, H. Zhang, and L. Zhu, *Evaluating guidelines for reporting empirical software engineering studies*, Empirical Software Engineering **13 (1)** (2008), 97–121. Cited on p. 79.

[Kli91] R. Kling, *Co-operation, co-ordination and control in computer-supported work*, Communications of the ACM **(34)12** (1991). Cited on p. 55.

[KLM+97] G. Kizcales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J-M Loingtier, and J. Irwin, *Aspect oriented programming*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP), June 1997. Cited on pp. 23 and 213.

[Klo] *Klocwork official website*, `http://www.klocwork.com/` [Online; accessed September 2011]. Cited on p. 33.

[KM05] M. Kersten and G. Murphy, *Mylar: a degree-of-interest model for ide's*, International Conference on Aspect Oriented Software Development, 2005, pp. 159–168. Cited on pp. 24, 33, and 138.

[KMCA06] A.J. Ko, B.A. Myers, M.J. Coblenz, and H.H. Aung, *An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks*, IEEE Transactions on Software Engineering (2006), 971–987. Cited on p. 24.

[KP88] G. E. Krasner and S. T. Pope, *A cookbook for using the model-view-controller user interface paradigm in smalltalk-80*, Journal of Object-Oriented Programming **1(3)** (1988), 26–49. Cited on pp. 37, 43, and 213.

[KP09] C. Kohls and S. Panke, *Is that true...? - thoughts on the epistemology of patterns*, PLoP '09 : Proceedings of the 16th Conference on Patterns Languages of Programs, 2009. Cited on pp. 82 and 184.

[KRW05] D. Kirk, M. Roper, and M. Wood, *Identifying and addressing problems in framework reuse*, Proceedings of the 13th International Workshop on Program Comprehension (IPWC'05), 2005, pp. 77–86. Cited on pp. 40 and 94.

[KTC+92] S. M. Kaplan, W. J. Tolone, A. M. Carroll, D. P. Bogia, and C. Bignoli, *Supporting collaborative software development with conversationbuilder*, ACM SIGSOFT 92: 5th Symposium on Software Development Environments (Tyson's Corner, Virginia), 1992, pp. 11–20. Cited on p. 62.

[LAD+94] P. Linos, P. Aubet, L. Dumas, Y. Helleboid, P. Lejeune, and P. Tulula, *Visualizing program dependencies: An experimental study*, Software-Practice and Experience **24(4)** (1994), 387–403. Cited on p. 31.

[LB03] C. Larman and V. Basili, *Iterative and incremental development: A brief history*, Computer **36(6)** (2003), 47–56. Cited on p. 58.

[LD01] M. Lanza and S. Ducasse, *A categorization of classes based on visualization of their internal structure: the class blueprint*, Proceedings for OOPSLA 2001 (2001), 300–311. Cited on p. 32.

[Lea94] D. Lea, *Chistopher alexander: An introduction for object-oriented designers*, Software Engineering Notes **19(1)** (1994), 39–45. Cited on p. 82.

[Lik32] R. Likert, *A technique for the measurement of attitudes*, Archives of Psychology 22 **140** (1932), 1–55. Cited on p. 153.

[Lio92a] Y.I. Liou, *Collaborative knowledge acquisition*, Expert Systems with Applications **5** (1992), no. 1-2, 1–13. Cited on p. 115.

[Lio92b] ———, *Knowledge acquisition: issues, techniques and methodology*, SIGMIS Database 23, vol. 1, 1992, pp. 59–64. Cited on p. 113.

[LOJW98] B. S. Lerner, L. J. Osterweil, Stanley M. Sutton Jr., and A. Wise, *Programming process coordination in little-jil toward the harmonious functioning of parts for effective results*, European Workshop on Software Process Technology, 1998. Cited on p. 61.

[Lon10] G. Longworth, *Concise encyclopedia of philosophy of language and linguistics*, ch. Definitions: Uses and Varieties, Elsevier, 2010. Cited on p. 121.

[Lou06] P. Louridas, *Using wikis in software development*, IEEE Software **23** (2006), 88–91. Cited on p. 124.

[LPLS86] D.C Littman, J. Pinto, S. Letovsky, and E. Soloway, *Mental models and software maintenance*, Proceedings of the 1st Workshop on Empirical Studies of Programmers, 1986, pp. 80–98. Cited on p. 20.

[LVD03] T. D. LaToza, G. Venolia, and R. DeLine, *Maintaining mental models: A study of developer working habits*, Proc. of the International Conference of Software Engineering (ICSE'06) (Shanghai, China), 2003. Cited on pp. 7, 71, 96, and 111.

[MC94] T. W. Malone and K. Crowston, *The interdisciplinary study of coordination*, ACM Computing Surveys (CSUR) **26(1)** (1994), 87–199. Cited on pp. 33 and 59.

[McA00] Donald R. McAndrews, *Team software process: An overview and preliminary results of using disciplined practices, the*, Cmu/sei-2000-tr-015, CMU/SEI, 2000. Cited on p. 23.

[MD85] S. Mittal and C.L. Dym, *Knowledge acquisition from multiple experts*, Al Magazine **6(2)** (1985), 32–36. Cited on p. 113.

[med] *Mediawiki - the free wiki engine, 2007*, `http://www.mediawiki.org` [Online; accessed August 2011]. Cited on p. 124.

[MFH02] A. Mockus, R. Fielding, and J.D. Herbsleb, *Two case studies of open source software development: Apache and mozilla*, ACM Transactions of Software Engineering and Methodology **11(3)** (2002), 309–346. Cited on p. 26.

[MFM03] A. Marcus, L. Feng, and J.I. Maletic, *Comprehension of software analysis data using 3d visualization*, Proceedings of the IEEE International Workshop on Program Comprehension (IWPC2003), 2003, pp. 105–114. Cited on p. 33.

[M.H91] M.H.Brown, *Zeus: A system for algorithm animation and multi-view editing*, Proceedings of the IEEE 1991 Workshop on Visual Languages, 1991, pp. 4–9. Cited on p. 31.

[MHK+01] M.Hakala, J. Hautamäki, K.Koskimies, J.Paakki, A.Viljamaa, and J.Viljamaa, *Annotating reusable software architectures with specialization patterns*, Proceedings of the Working IEEE/IFIPConference on Software Architecture (WICSA'01), 2001, p. 171. Cited on pp. 44 and 50.

[MK88] H. Muller and K. Klashinsky, *Rigi – a system for programming-in-the-large*, Proceedings of the 10th International Conference on Software Engineering (ICSE'10), 1988, pp. 80–86. Cited on pp. 31 and 32.

[ML05] A. Murray and T. Lethbridge, *On generating cognitive patterns of software comprehension*, Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research, vol. 200-211, 2005. Cited on p. 28.

[MMM95] H. Mili, F. Mili, and A. Mili, *Reusing software: Issues and research directions*, Software Engineering **21(6)** (1995), 528–562. Cited on p. 1.

[MN96] S. Moser and O. Nierstrasz, *The effect of object-oriented frameworks on productivity*, IEEE Computer (1996), 45–51. Cited on p. 37.

[MNS95] G.C. Murphy, D. Notkin, and K. Sullivan, *Software reflexion models: Bridging the gap between source and high-level models*, Proceedings of Foundations of Software Engineering, 1995, pp. 18–28. Cited on p. 32.

[MRB97] R. Martin, D. Riehle, and F. Buschmann, *Pattern languages of program design*, vol. 3, Addison-Wesley, 1997. Cited on p. 82.

[MRS02] M. Morisio, D. Romano, and I. Stamelos, *Quality, productivity, and learning in framework-based development: An exploratory case study*, IEEE Transactions on Software Engineering **28(9)** (2002), 876–888. Cited on pp. 40 and 50.

[MS95] A. Mendelson and J. Sametinger, *Reverse engineering by visualizing and querying*, Software – Concepts and Tools **16** (1995), 170–182. Cited on p. 31.

[MVS09] G.C. Murphy, P. Viriyakattiyaporn, and D. Shepherd, *Using activity traces to characterize programming behaviour beyond the lab*, Proceedings of the International Conference on Program Comprehension, 2009, pp. 90–94. Cited on p. 24.

[NA05] N.Flores and A.Aguiar, *Jfreedom: a reverse engineering tool to recover framework design*, Proceedings of the 1st International Workshop on Object-Oriented Reengineering, ECOOP'05, 2005. Cited on pp. 46 and 50.

[NAT68] NATO, *Software engineering conference*, October 1968. Cited on p. 17.

[NBM95] J. F. Nunamaker, R. O. Briggs, and D. D. Mittleman, *Electronic meeting systems: Ten years of lessons learned*, Groupware: Technology and Applications (D. Coleman and R. Khanna, eds.), Prentice-Hall, Englewood Cliffs, NJ, 1995. Cited on p. 54.

[Net] *Netbeans ide*, `http://netbeans.org/` [Online; accessed August 2011]. Cited on p. 23.

[NK02] J. Noerbjerg and P. Kraft, *Software practice is social practice*, Social Thinking, Software Thinking, Software Practice, MIT Press, 2002, pp. 205–222. Cited on p. 110.

[NOY00] K. Nakakoji, K. Ohira, and Y. Yamamoto, *Computational support for collective creativity*, Knowlegde-Based Systems Journal, Elsevier Science **13(7-8)** (2000), 451.458. Cited on p. 110.

[NYY06] K. Nakakoji, Y. Yamamoto, and Y. Ye, *Supporting software development as knowledge community evolution*, Proceedings of the CSCW Workshop on Suporting the Social Side of Large Scale Software Development, 2006. Cited on pp. 7 and 71.

[OBM05] C. O'Reilly, D. Bustard, and P. Morrow, *The war room command console (shared visualizations for inclusive team coordination)*, Softvis, 2005. Cited on p. 34.

[OC99] A. Ortigosa and M. Campo, *Smartbooks: A step beyond active-cookbooks to aid in framework instantiation*, Technology of Object-Oriented Languages and Systems, Prentice Hall, 1999. Cited on p. 44.

[OH03] M. Ohki and Y. Hosaka, *A program visualization tool for program comprehension*, Human Centric Computing Languages (2003). Cited on p. 31.

[OLDR11] F. Olivero, M. Lanza, M. D'Ambros, and R. Robbes, *Enabling program comprehension through a visual object-focused development environment*, IEEE Symposium on Visual Languages and Human-Centric Computing, 2011, pp. 127–134. Cited on p. 34.

[OMG] OMG, *Object constraint language specification*, http://www.omg.org/spec/OCL/ [Online; accessed August 2011]. Cited on p. 23.

[OMG10] *Model driven architecture (mda)*, 2010, http://www.omg.org/mda [Online; accessed August 2011]. Cited on p. 23.

[OMG11] OMG, *Unified modeling language specification*, 2011, http://www.uml.org/ [Online; accessed August 2011]. Cited on pp. 23 and 213.

[Pac05] M. Pacione, *A novel software visualization model to support object-oriented program comprehension*, Ph.D. thesis, University of Strathclyde, Glasgow, 2005. Cited on p. 31.

[PBT97] M. Petre, A. Blackwell, and T.Green, *Cognitive questions in software visualization*, Software Visualization: Programming as a Multi-Media Experience, MIT Press, 1997, pp. 453–480. Cited on pp. 22 and 31.

[Pen] *Penumbra := eclipse assmalltalkplugin*, http://www.info.ucl.ac.be/~jbrichau/penumbra.html [Online; accessed September 2011]. Cited on p. 33.

[Pen87] N. Pennington, *Stimulus structures and mental representations in expert comprehension of computer programs*, Cognitive Psychology **19** (1987), 295–341. Cited on pp. 20, 22, 29, and 31.

[Pen92] D. A. Penny, *The software landscape: A visual formalism for programming-in-the-large*, Ph.D. thesis, University of Toronto, 1992. Cited on p. 33.

[PJ05] N. Pillay and V.R. Jugoo, *An investigation into student characteristics affecting novice programming performance*, ACM SIGCSE Bulletin **37(4)** (2005), 107–110. Cited on p. 24.

[P.N99] P.N.Robillard, *The role of knowledge in software development*, Communications of the ACM **42(1)** (1999), 87–92. Cited on p. 110.

[PPI11] *Psycology of programming interest group*, 2011, http://www.cs.york.ac.uk/ppig2011/ [Online; accessed September 2011]. Cited on p. 25.

[Pre94] W. Pree, *Design patterns for object-oriented software development*, Addison-Wesley, 1994. Cited on pp. 45, 49, 90, and 103.

[Pre95] ______, *Framework development and reuse support*, Visual Object-Oriented Programming, Concepts and Environments, Prentice Hall, 1995. Cited on p. 43.

[PSK07] M. Di Penta, R.E.K. Stirewalt, and E. Kraemer, *Desigining your next empirical study on program comprehension*, Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07), 2007, pp. 281–285. Cited on p. 26.

[RCM04] M. Robillard, W. Coelho, and G. Murphy, *How effective developers investigate source code: An exploratory study*, IEEE Transactions on Software Engineering **30 (12)** (2004). Cited on pp. 92, 95, and 106.

[RCWP91] G.-C. Roman, K. C. Cox, C. D. Wilcox, and J. Y. Plun, *Pavane: A system for declarative visualization of concurrent computations*, Technical report wucs-91-26, Washington University, St. Louis, 1991. Cited on p. 31.

[RDL90] V. Rajlich, N. Damskinos, and P. Linos, *Vifor: A tool for software maintenance*, Software-Practice and Experience **20(1)** (1990), 67–77. Cited on p. 31.

[Red] *Redmine.org*, http://redmine.org [Online; accessed September 2011]. Cited on p. 142.

[Rei01] S. Reiss, *An overview of bloom*, Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program analysis for software tools and engineering, 2001, pp. 2–5. Cited on p. 33.

[Rie00] D. Riehle, *Framework design: A role modelling approach*, Ph.D. thesis, Swiss Federal Institute of Technology, 2000. Cited on p. 49.

[Ris86] R. S. Rist, *Plans in programming: Definition, demonstration, and development*, Proceedings of the 1st Workshop on Empirical Studies of Programmers, 1986. Cited on p. 29.

[RJ97] D. Roberts and R. E. Johnson, *Evolving frameworks: A pattern language for developing object-oriented frameworks*, Pattern Languages of Program Design 3, Addison-Wesley, 1997. Cited on pp. 86 and 93.

[RL07] R. Robbes and M. Lanza, *Characterizing and understanding development sessions*, Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07), 2007, pp. 155–166. Cited on p. 24.

[RM03] M.P. Robillard and G. Murphy, *Feat: A tool for locating, describing, and analyzing concerns in source code*, Proceedings of the 25th International Conference on Software Engineering, 2003, pp. 822–823. Cited on p. 33.

[RSVW94] W. Reinhard, J. Schweitzer, G. Volksen, and M. Weber, *Cscw tools: concepts and architectures*, Computer **27(5)** (1994), 28–36. Cited on p. 56.

[RTC] *Rational team concert*, `http://www-01.ibm.com/software/rational/products/rtc/` [Online; accessed September 2011]. Cited on p. 137.

[SAA+00] A.Th. Schreiber, J. Akkermans, A. Anjewierden, R. De Hoog, N. Shadbolt, W. Van De Velde, and B. Wielinga, *Knowledge engineering and management: The commonkads methodology*, MIT Press, 2000. Cited on p. 114.

[Sar05] A. Sarma, *A survey of collaborative tools in software development*, Tech. report, ISR - Institute for Software Research, University of California, 2005. Cited on pp. 60, 61, and 62.

[SB92] K. Schmidt and L. Bannon, *Taking cscw seriously*, Computer Supported Cooperative Work **1** (1992), 7–40. Cited on p. 56.

[SBdL92] P. Schorn, A. Brungger, and M. de Lorenzi, *The xyz geobench: Animation of geometric algorithms*, Animations for Geometric Algorithms: A Video Review, Digital Systems Research Center, Palo Alto, California, 1992. Cited on p. 31.

[Sca89] D. A. Scanlan, *Structured flowcharts outperform pseudocode: An experimental comparison*, IEEE Trans. Soft. Eng. (1989). Cited on p. 31.

[SCG05] M-A. Storey, D. Cubranic, and D. M. German, *On the use of visualization to support awareness of human activities in software development: A survey and a framework*, In Proc. of the 2005 ACM symposium on Software Visualization, 2005, pp. 193–202. Cited on p. 30.

[SD96] S.Tilley and D.B.Smith, *Coming attractions in program understanding*, Technical Report 96-TR-019, CMU/SEI, 1996. Cited on pp. 21 and 30.

[SE84] E. Soloway and K. Erlich, *Empirical studies of programming knowledge*, IEEE Transactions on Software Engineering **10(5)** (1984), 595–609. Cited on pp. 19 and 29.

[Sec] *Second life*, `http://secondlife.com` [Online; accessed September 2011]. Cited on p. 64.

[SES05] J. Singer, R. Elves, and M-A. Storey, *Navtracks demonstration: Supporting navigation in software space*, International Workshop on Program Comprehension, 2005. Cited on p. 33.

[SFC94] M-A Storey, F. Fracchia, and S. Carpendale, *A top down approach to algorithm animation*, Technical Report CMPT 94-05, Simon Frasier University, Brunaby, B.C., Canada, 1994. Cited on p. 31.

[SFM97] M-A Storey, F. Fracchia, and H. Muller, *Cognitive design elements to support the construction of a mental model during software visualization*, Proceedings of the 5th International Workshop on Program Comprehension (IWPC'97) (Dearborn, Michigan), 1997, pp. 17–28. Cited on pp. 21, 31, and 95.

[SG96] M. Shaw and D. Garlan, *Software architecture – perspectives on an emerging discipline*, Prentice Hall, 1996. Cited on p. 101.

[SG07] I. Safer and G.C.Murphy, *Comparing episodic and semantic interfaces for task boundary identification*, Proceedings of the Conference of the Center of Advanced Studies on Collaborative Research, no. 229-243, 2007. Cited on p. 24.

[Shi] C. Shirky, *Folksonomy*, `http://many.corante.com/archives/2004/08/25/folksonomy.php` [Online; accessed August 2011]. Cited on p. 121.

[Shn77] B. Shneiderman, *Measuring computer program quality and comprehension*, International Journal of Man-Machine Studies **9** (1977), 465–478. Cited on p. 29.

[SK10] K. Sirbi and P. J. Kulkarni, *Stronger enforcement of security using aop and spring aop*, Journal of Computing **2(6)** (2010). Cited on p. 23.

[SLB00] F. Schull, F. Lanubile, and V. Basil, *Investigating reading techniques for object-oriented framework learning*, IEEE Transactions on Software Engineering **26(11)** (2000). Cited on pp. 39, 42, 94, and 95.

[SLVA97] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, *An examination of software engineering work practices*, Proceedings of CASCON'97, 1997, pp. 209–233. Cited on pp. 27 and 31.

[SM79] B. Shneiderman and R. Mayer, *Syntatic/semantic interactions in programmer behavior: A model and experimental results*, International Journal of Computer and Information Science **8(3)** (1979), 219–238. Cited on p. 19.

[SMV06] J. Sillito, G. Murphy, and K. De Volder, *Questions programmers ask during software evolution tasks*, Proceedings of the 14th ACM SIGSOFT international Symposium on Foundations of Software Engineering, 2006. Cited on pp. 94 and 106.

[SNvdH] A. Sarma, Z. Noroozi, and A. van der Hoek, *Palantír – raising awareness among configuration management workspaces*, Proceedings of the 25th International Conference on Software Engineering, pp. 444–454. Cited on p. 62.

[Sof11] *Software visualization symposium*, 2011, `http://www.softvis.org` [Online; accessed September 2011]. Cited on p. 26.

[SPL+88] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert, *Designing documentation to compensate for delocalized plans*, Communication of the ACM **31(11)** (1988), 1259–1267. Cited on p. 20.

[SS00] S.E. Sim and M-A. Storey, *A structured demonstration of program comprehension tools*, Proceedings of the 7th Working Conference in Reverse Engineering (WCRE), 2000, pp. 184–193. Cited on p. 26.

[SSP95] A. Schappert, P. Sommerlad, and W. Pree, *Automated framework development*, Symposium on Software Reusability (SSR'95) (ACM Software Engineering Notes, ed.), 1995. Cited on p. 43.

[SSS07] F. Shull, J. Singer, and D.I.K. Sjøberg, *Guide to advanced empirical software engineering*, Springer-Verlag, 2007. Cited on p. 79.

[Sto03] M-A. Storey, *Designing a software exploration tool using a cognitive framework of design elements*, Software Visualization (2003). Cited on pp. 32 and 33.

[Sto05] ______, *Theories, methods and tools in program comprehension: Past, present and future*, Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC) (St. Louis, MO) (IEEE Computer Society Press, ed.), 2005, pp. 181–191. Cited on pp. 17, 22, 25, 29, 32, and 95.

[Sub] *Subethaedit - collaborative text editing*, `http://www.codingmonkeys.de/subethaedit/` [Online; accessed September 2011]. Cited on p. 34.

[Sur04] J. Surowiecki, *The wisdom of crowds: Why the many are smarter than the few and how collective wisdom shapes business, economies, societies and nations*, Anchor Publishing, 2004. Cited on pp. 78, 97, and 111.

[Tei09] T. Teixeira, *Web collaboration for software engineering*, Mscthesis, Faculty of Engineering of University of Porto (FEUP), July 2009. Cited on p. 62.

[THP92] W.F. Tichy, N. Habermann, and L. Pretchelt, *Summary of the dagstuhl workshop on future directions in software engineering*, ACM SIGSOFT Software Engineering Notes **18(1)** (1992), 35–48. Cited on p. 9.

[TKS01] T.Eisenbarth, R. Koschke, and D. Simon, *iding program comprehension by static and dynamic feature analysis*, Proceedings of the IEEE International Conference on Software Maintenance, 2001. Cited on p. 32.

[TM04] T. Tourwé and T. Mens, *Automated support for framework-based software evolution*, Proceedings of the International Conference on Software Maintenance, 2004, p. 148. Cited on p. 45.

[TN99] S. Terzis and P. Nixon, *Building the next generation groupware: A survey of groupware and its impact on the virtual enterprise*, Technical Report TCD-CS-1999-08, Trinity College Dublin, Department of Computer Science, 1999. Cited on p. 55.

[Tog11] *Together - visual modeling for software architecture design*, 2011, `http://www.borland.com/us/products/together/` [Online; accessed September 2011]. Cited on p. 33.

[Tou02] T. Tourwé, *Automated support for framework-based software evolution*, Ph.D. thesis, Vrije Universiteit, 2002. Cited on p. 45.

[Tra] *The trac project*, `http://trac.edgewall.org/` [Online; accessed September 2011]. Cited on p. 61.

[TW07] A. Turner and C. Wang, *Ajax: Selecting the framework that fits*, Dr. Dobb's Journal (2007). Cited on p. 88.

[Ves85] I. Vessey, *Expertise in debugging computer programs: A process analysis*, International Journal of Man-Machine Studies (1985). Cited on p. 25.

[VLH11] *Ieee symposium on visual languages and human-centric computing*, 2011, `http://www.cs.cmu.edu/~vlhcc2011/` [Online; accessed September 2011]. Cited on p. 25.

[vMV93] A. von Maryhauser and A. Vans, *From code understanding needs to reverse engineering tool capabilities*, Proceedings of CASE'93, 1993, pp. 230–239. Cited on pp. 27 and 31.

[vMV95] ———, *Program comprehension during software maintenance and evolution*, IEEE Computer (1995), 44–55. Cited on p. 20.

[Vol06] K. De Volder, *Jquery: A generic code browser with a declarative configuration language.*, Practical Aspects of Declarative Languages (2006), 88–102. Cited on p. 46.

[Wal] T. W. Wal, *You down with folksonomy?*, `http://www.vanderwal.net/random/entrysel.php?blog=1529` [Online; accessed August 2011]. Cited on p. 121.

[Wat86] *A guide to expert systems*, Addison-Wesley, 1986. Cited on p. 113.

[WBWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing object-oriented software*, Prentice Hall, 1990. Cited on p. 97.

[WCR11] *Working conference in reverse engineering (wcre)*, 2011, `http://www.cs.wm.edu/semeru/wcre2011/` [Online; accessed October 2011]. Cited on p. 26.

[Wei98] *The psychology of computer programming: Silver anniversary editions*, Dorset House Publishing Company, September 1998. Cited on p. 53.

[Whi07] J. Whitehead, *Collaboration in software engineering: A roadmap*, Future of Software Engineering within the International Conference on Software Engineering (Washington, DC) (IEEE Computer Society, ed.), 2007, pp. 214–225. Cited on pp. 58, 59, and 60.

[wik] *Wikipedia, the free encyclopedia, 2010.*, `http://www.wikipedia.org` [Online; accessed August 2011]. Cited on p. 124.

[Wil91] P. Wilson, *Computer supported cooperative work: An introduction*, Kluwer Academic Pub (1991). Cited on p. 55.

[WKB09] M. Weimer, A. Karatzoglou, and M. Bruch, *Maximum margin matrix factorization for code recommendation*, Proceedings of the 3rd ACM Conference on Recommender Systems., 2009, pp. 309–312. Cited on p. 33.

[WL07] R. Wettel and M. Lanza, *Program comprehension through software habitability*, Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07), 2007, pp. 231–240. Cited on p. 33.

[Won00] K. Wong, *The reverse engineering notebook*, Ph.D. thesis, University of Victoria, 2000. Cited on p. 31.

[YBJ01] Joseph Yoder, Federico Balaguer, and Ralph Johnson, *Adaptive object models for implementing business rules*, Urbana (2001). Cited on pp. 23 and 151.

[YR11] A.T.T. Ying and M. Robillard, *The influence of the task on programmer behaviour*, Proceedings of the International Conference on Program Comprehension, 2011, pp. 31–40. Cited on p. 24.

[ZA05] U. Zdun and P. Avgeriou, *Modelling architectural patterns using architectural primitives*, OOPSLA, 2005. Cited on pp. 46 and 101.

[ZGH07] L. Zou, M.W. Godfrey, and A.E. Hassan, *Detecting interaction coupling from task interaction histories*, Proceedings of the International Conference on Program Comprehension, 2007, pp. 135–144. Cited on p. 24.

[ZL00] I. Zayour and T. C. Lethbridge, *A cognitive and user centric based approach for reverse engineering tool design*, Proceedings of the CASCON 2000, 2000. Cited on pp. 28 and 31.

[ZW98] M.V. Zelkowitz and D.R. Wallace, *Experimental models for validating technology*, IEEE Computer **31(5)** (1998), 23–31. Cited on pp. 9 and 10.