Supporting Software Evolution in the Organizations

Nicolas Anquetil

Université des Sciences et Technologies de Lille – Lille 1

Département de formation doctorale en informatique **École doctorale SPI Lille**
UFR IEEA

Supporting Software Evolution in the Organizations

THÈSE

présentée et soutenue publiquement le 09 May 2014

pour l'obtention de

Habilitation à Diriger des Recherches
de l'Université des Sciences et Technologies de Lille
(spécialité informatique)

par

Nicolas Anquetil

Composition du jury

Rapporteur :	M. Mark Harman, M. Paul Klint, Mme Julia Lawall
Examinateur :	M. Mel Ó'Cinnéide, M. Jacques Malenfant, M. Lionel Seinturier
Garant :	M. Stéphane Ducasse

Laboratoire d'Informatique Fondamentale de Lille — UMR USTL/CNRS 8022
INRIA Lille - Nord Europe

Numéro d'ordre: 41076

Contents

List of Figures

List of Tables

Acknowledgement

It's been a long way, West then South, and East then North. So many people contributed to this in so many ways: friends, teachers, colleagues, students, family members, judges. They were all part of this journey and they all made it what it is. Some carried me a long way, others showed me the beauty of the path. And even as I was carrying some, they were teaching me how to walk.

It's our building, unique as it is. What part is more important? The brick or the mortar? The windows or the roof? I cannot tell. Removing any part would make it something else.

Names and faces come to me that I should thank here. Then other names and faces follow, that I should not forget. And many more remain in the back, waiting for a small sign to resurface. I will not choose and let each one decide for himself his own contribution. Reaching back to the foundations of the building, I find the same difficulty and come to the same conclusion:

"Thanking you all, as much as you deserve, for all you brought me, is impossible. Words seem sometimes so small in front of what they should express. I wish I could stack them on, multiply them ... It's no use.

I am left with only one, so simple:

Thanks"

Abstract

Software systems are now so intrinsically part of our lives that we do not see them any more. They run our phones, our cars, our leisures, our banks, our shops, our cities ... This brings a significant burden on the software industry. All these systems need to be updated, corrected, and enhanced as the users and consumers have new needs. As a result, most of the software engineering activity may be classified as Software Maintenance, "the totality of activities required to provide cost-effective support to a software system".

In an ecosystem where processing power for computers, and many other relevant metrics such as disk capacity or network bandwidth, doubles every 18 months ("Moore's Law"), technologies evolve at a fast pace. In this ecosystem, software maintenance suffers from the drawback of having to address the past (past languages, existing systems, old technologies). It is often ill-perceived, and treated as a punishment. Because of this, solutions and tools for software maintenance have long lagged far behind those for new software development. For example, the antique approach of manually inserting traces in the source code to understand the execution path is still a very valid one.

All my research activity focused on helping people to do software maintenance in better conditions or more efficiently. An holistic approach of the problem must consider the software that has to be maintained, the people doing it, and the organization in which and for which it is done. As such, I studied different facets of the problem that will be presented in three parts in this document: *Software:* The source code is the center piece of the maintenance activity. Whatever the task (ex: enhancement or bug correction), it typically comes down to understand the current source code and find out what to change and/or add to make it behave as expected. I studied how to monitor the evolution of the source code, how to prevent it's decaying and how to remedy bad situations; *People:* One of the fundamental asset of people dealing with maintenance is the knowledge they have, of computer science (programming techniques), of the application domain, of the software itself. It is highly significant that from 40% to 60% of software maintenance time is spent reading the code to understand what it does, how it does it, how it can be changed; *Organization:* Organizations may have a strong impact on the way activities such as software maintenance are performed by their individual members. The support offered within the organization, the constraints they impose, the cultural environment, all affect how easy or difficult it can be to do the tasks and therefore how well or badly they can be done. I studied some software maintenance processes that organizations use.

In this document, the various research topics I addressed, are organised in a logical way that does not always respect the chronological order of events. I wished to highlight, not only the results of the research, through the publications that attest to them, but also the collaborations that made them possible, collaboration with students or fellow researchers. For each result presented here, I tried to summarised as much as possible the discussion of the previous state of the art and the result itself. First because, more details can easily be found in the referenced publications, but also because some of this research is quite old and sometimes fell in the realm of "common sense".

Part I

Software Evolution

Chapter 1

Introduction

1.1 Why Software Maintenance?

Software maintenance is the activity of modifying a software product after delivery for example to correct faults in it, to add new functionalities, or to adapt it to new execution conditions [ISO 2006]. It has a very distinctive place in the Information Technology domain. According to all accounts (*e.g.* [Pigoski 1997, Seacord 2003]), this is by far the most practised activity on software systems, yet it is traditionally ill-considered by practitioners and neglected by the academy. It is rarely taught in universities and most of research is devoted to developing new languages, new features and generally enhance the way we develop software. Yet more than 90% of the work in industry consists in modifying existing programs developed with past technologies.

I first came to work on software maintenance on the CSER (Consortium for Software Engineering Research) project of Prof. Timothy C. Lethbridge from University of Ottawa, Canada. In this project, we were working with a telecommunication company, trying to help it solve some of its difficulties in maintaining a large legacy system. Since then, and during the more than 15 years of research summarized in this document, I considered some of the many problems it raises.

1.2 Goals of this Document

In this document to obtain the "Habilitation à Diriger des Recherches"[1] I aimed at illustrating two complementary qualities that I consider necessary to supervise research:

- First, demonstrate successful experience in supervising novice researchers. This concerns typically PhD student direction, although in my case I also worked with many Master students.

[1] "Habilitation to supervise research"

- Second, demonstrate an ability to pursue long term research goals through succeeding short projects.

To this end, I have written the present document with a view to putting all my research results in perspective, showing how they contributed to higher objectives. The resulting order does not entirely follow the chronological order of the work presented, but it makes the logical connections clearer. I have also highlighted the contributions I had from students or colleagues in reaching these results. Throughout the document I switch between the singular and plural forms ("I" and "we") to better acknowledge work and results that were achieve in collaboration with others.

My career did not follow an entirely "traditional" path, which did impact some of my research work. It seems, therefore, appropriate to give a brief overview of it at this point. After a PhD at the University of Montreal, I first came to work on software maintenance as a research assistant in the KBRE group of Prof. Timothy C. Lethbridge at the University of Ottawa (Canada) for three years. I then worked 18 months as an invited professor at the Federal University of Rio de Janeiro (Brazil) before finding a permanent position at the Catholic University of Brasilia (Brazil). In this small university, we did not have a PhD program in computer science, which explains why a good deal of my research was conducted with Master students. On the more positive side, I had the opportunity to work with psychologists, economists, or management people which gave to some part of my research a much less technical point of view. Not all of the research I did in this position is reported here. Later, I took a position of research assistant at Ecole des Mines de Nantes (France) in the European research project AMPLE before coming finally to my current position as assistant professor at the university of Lille-1 (France). It is only in this last position that I had the opportunity to co-supervise PhD students in the RMod group under the direction of Prof. Stéphane Ducasse.

1.3 Organization of this Document

This document is organized along three main axes of software maintenance as illustrated in Figure 1.1: the *software* itself; the *people* that maintain it and their knowledge; and the *organization* and its maintenance processes. This categorization itself came out of our work on organizing the knowledge domains used in maintenance (see Section 7.2).

Following these three dimensions, this document has three main parts:

Part II—"Software System": In this part, I consider the software itself in the form of source code and explore three aspects of software

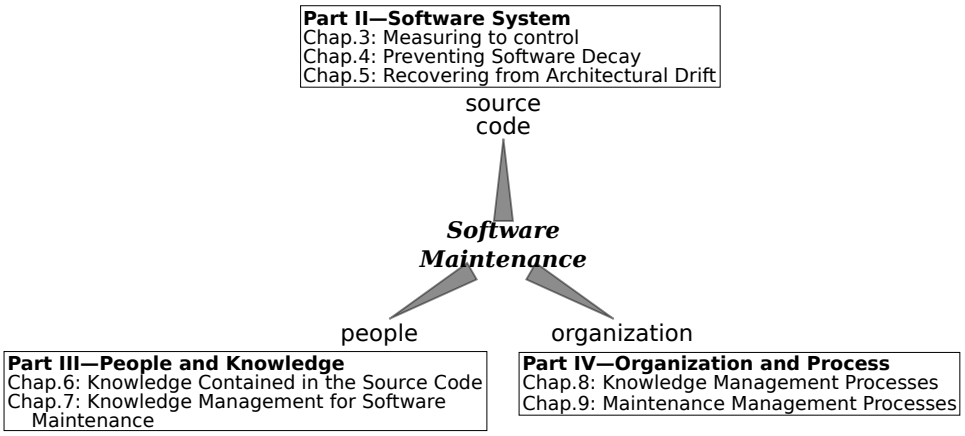

Figure 1.1: Three dimensions of software maintenance and where they are treated in this document

evolution:

- In Chapter 3, I discuss how one can *control* the evolution by monitoring different aspects of the code. This chapter includes work done with two PhD students, Cesar Couto (co-supervised with Prof. Marco T. Valente from Federal University of Minas Gerais, Brazil) and Karine Mordal-Manet (under the supervision of Prof. Françoise Balmas, from University of Paris 8, France), and a Master student, Cristiane S. Ramos [Anquetil 2011], [Couto 2012], [Mordal-Manet 2011], [Mordal 2012], [Ramos 2004];
- In Chapter 4, I study how to *prevent* software systems from losing their quality. This includes contributions from a post-doctoral fellow, Simon Allier, and a PhD student, André Hora Calvacante (co-supervised with Prof. Stéphane Ducasse) [Allier 2012], [Hora 2012];
- In Chapter 5, I look at possible actions to *remedy* architectural drift, a typical problem for legacy software. This comes out of my work in the CSER project, with a more recent contribution from a PhD student, Jannik Laval (co-supervised with Prof. Stéphane Ducasse) [Anquetil 1997], [Anquetil 1998a], [Anquetil 1998b], [Anquetil 1999a], [Anquetil 1999b], [Lethbridge 2002], [Anquetil 2003b], [Laval 2010], [Laval 2012].

Part III—"People and Knowledge": In this part, considering that software development and maintenance are knowledge intensive activities, we were interested in defining more precisely what knowledge was required:

- In Chapter 6, I consider whether this knowledge embedded in the source code is accessible to automated treatment. Can it be extracted? This research was done jointly with a postdoctoral fellow, Usman M. Bhatti [Anquetil 1998a], [Anquetil 2000a], [Anquetil 2001], [U.Bhatti 2012]
- In Chapter 7, I look at what knowledge do maintainers require? This is the sum of various research efforts led in collaboration with two Master students, Márcio G.B. Dias, Kleiber D. de Sousa, and also two postgraduates Marcelo F. Ramal and Ricardo de M. Meneses [Anquetil 2003a], [Anquetil 2006], [Dias 2003a], [Dias 2003b], [Ramal 2002]

Part IV—"Organization and Process": In the last part, I consider the organizations and how their processes can help software maintenance:

- In Chapter 8, I discuss how the knowledge required to maintain software can be extracted, stored and redistributed. This part also resulted from the work of several Master students Sergio C.B. de Souza, Kleiber D. de Sousa, Alexandre H. Torres [Anquetil 2007], [de Sousa 2004], [de Souza 2005], [Souza 2006]
- In Chapter 9, I come to other processes for the management and improvement of software maintenance. A part of this research was conducted within the AMPLE European project with many collaborations, another part is the result of the effort of a Master student, Kênia P.B. Webster [Webster 2005]

These three main parts are preceded and followed by:

Part I—"Software Evolution": In this introductory part, I present the document (Chapter 1) and the domain of software maintenance (Chapter 2);

Part V—"Perspectives": In the final part of the document, I summarize the discussion and propose some additional research directions (Chapter 10).

Chapter 2

Software Maintenance

2.1 Basic Facts on Software Maintenance

Software maintenance is the modification of a software product after delivery to correct faults, or to improve performance and other attributes [ISO 2006]. It is typically classified in four categories [ISO 2006]:

Adaptive: Modifying the system to cope with changes in the software environment.

When upgrading the operating system (OS) of a computer, or the database management system, it may occurs that the software running in this environment needs to be adapted. This would be an example of adaptive maintenance.

Perfective: Implementing new or changed user requirements that concern functional enhancements to the software.

User often wish that a system could do just a little bit more than what it currently offers. For example, they would prefer that an application could get its input data automatically from another system instead of having to enter it manually. This would be an example of perfective maintenance.

Corrective: Diagnosing and fixing errors.

Probably the best known of software maintenance categories. Corrective maintenance may be decomposed into emergency vs. normal corrective maintenance. Emergency corrective maintenance occurs when an error needs to be corrected without delay, typically because it impedes an important system to work.

Preventive: Increasing software maintainability or reliability to prevent problems in the future.

The most famous example of it was at the end of the 90's when many organizations had to fight against the Y2K problem[1]

[1] Year 2000 problem, when dates represented with only two digits would become "00", thus 2000, would be inferior to 1999, represented as "99".

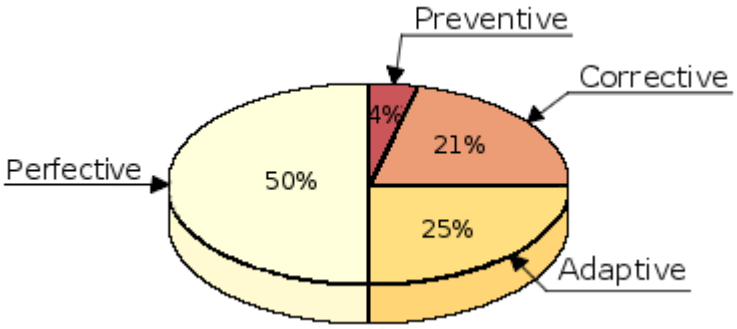

Figure 2.1: Relative importance of the software maintenance categories

Past studies (reported in [Pigoski 1997]) showed that, contrary to common belief, corrective maintenance represents only a small part of all maintenance (see Figure 2.1). Most software maintenance (50%) is done to add new features (perfective maintenance). The next most common kind is adaptive maintenance (25%) when some other part of the system (*e.g.* OS, database, hardware, third party library) changed. Corrective maintenance represents only a fifth (21%) of all maintenances while preventive maintenance makes up the smallest part (4%). Because of the small part of corrective maintenance, some researchers suggest changing the term of "software maintenance" to "software evolution". In this document, I use the two terms indifferently.

Studies show that software maintenance is, by far, the predominant activity in software engineering (90% of the total cost of a typical software [Pigoski 1997, Seacord 2003]). It is needed to keep software systems up to date and useful. Any software system reflects the world within which it operates. When this world changes, the software needs to change accordingly. Lehman's first law of software evolution (law of Continuing Change, [Lehman 1980, Lehman 1998]) is that "a program that is used undergoes continual change or becomes progressively less useful." Maintenance is mandatory, one simply cannot ignore new laws or new functionalities introduced by a concurrent. Programs must also be adapted to new computers or new operating systems, and users have ever growing expectations of what a given software system should be able to do for them (Lehman's law of Declining Quality: "The quality of systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes").

A corollary of Lehman's law of Continuing Change is that software maintenance is a sign of success: considering the costs associated to software maintenance,it is performed only for software systems which utility is perceived as more valuable than this cost. This is a conclusion that goes against

the usual perception of the activity, but maintenance actually means that a system is useful and that its users see a value in its continuing operation.

Maintenance differs from development "from scratch" for a number of reasons, including:

Event driven: Best practice recommends that development be requirement driven: one specifies the requirements and then plans their orderly implementation. Maintenance is event driven [Pigoski 1997], external events require the modification of the software, sometimes on very tight schedule, for example when discovering a critical bug. There is much less opportunity for planning.

Lack of documentation: Developers may rely on the fact that somebody (hopefully close by) knows one specific portion of the system, in the best cases there may even be a clear documentation. Maintainers must usually work from the source code to the exclusion of any other source of information.

Obsolete techniques: Developers may organize the system as best suits the requirements. In the best cases, they may even be able to choose the programming paradigm and language, the hardware platform, etc. Maintainers must cope with choices made by others in the past. The programming language may be old (e.g. COBOL), the system architecture may not fully support the modification they need to implement, or this architecture may even be so obfuscated by past modifications that there is no longer any clear architecture.

Differing processes: The process of maintenance is different from initial development in that it requires new activities up-front which do not exist in development. For example, maintenance requires to analyze in depth the system to be modified before any analysis of the task at hand starts.[2] Because of this, the bulk of the effort in a maintenance project is applied at the beginning of the project (between 40% to 60% of the time is spent analysing the source code to rediscover lost information about how it works ([Pfleeger 2002, p.475], [Pigoski 1997, p.35]), whereas, in a development project, most effort is applied towards the end (during the implementation and test[3]) [Grubb 2003]. In development, the initial activity (requirement elicitation) may be difficult, but it typically requires less effort than the implementation because fewer details are involved.

[2]The analysis of the system is made even more difficult by the usual lack of documentation on the system.

[3]In many maintenance projects, the test activity is crippled by the lack of previous test cases, the lack of detailed understanding of how the system should behave, and the pressure for a rapid delivery of the change.

System in operation: During maintenance, the system is in operation which may significantly increase the difficulty of altering the system while maintaining it operational. Particular conditions may only be available on the production environment (data from the full database, expensive hardware requiring a specific computing environment), making it difficult, for example, to replicate the conditions of a bug in the maintenance environment.

These characteristics make maintenance an activity quite different from software development.

2.2 Some Problems in Software Maintenance

Due to its characteristics, software maintenance suffers from problems that may be specific to it or generic to software engineering. I will expose some of them here that I have studied in my work.

A constant preoccupation of software engineering, that maintenance also shares, is the need to measure the quality of the programs, to be able to better plan future costs or understand how well the system is ageing. I will be discussing this issue in Chapter 3.

Due to the constant evolution of software systems, often in directions that were not foreseen when the systems were created, their quality slowly degrades over time, as stated by the Declining Quality law of Lehman (see preceding section). This phenomenon, called software decay, occurs at different levels of abstraction. In Chapter 4, I consider methods to prevent software decay.

I also focused on the architectural level, where the phenomenon is known as architectural drift or architectural erosion (*e.g.* [Murphy 2001, Rosik 2011]). In Chapter 5, I discuss my research to recover from architectural drift.

As described in the preceding section, software evolution is known to suffer from a perpetual lack of documentation. This results in a lack of knowledge in the maintenance team on many aspects of the system maintained: what it does, what is architecture is, why it was implemented that way, etc. We saw that about 50% of maintenance time is spent analysing the system to recover information that is needed to apply the needed modification. This is how the lack of documentation translates into a knowledge issue during maintenance, that will be discussed in Part III. In Chapter 6, I consider what knowledge can be found in the source code and how it could be extracted. This is then expanded in Chapter 7 where I will be looking at knowledge issues in software maintenance.

Finally, we saw in the preceding section that the process for software

maintenance is different than for usual software development. This is even more clear when we consider that one should include knowledge management activities. This will be the topic of Chapter 8 while Chapter 9 turns to other more "traditional" processes of software engineering that must be adapted to software maintenance.

Part II

Software System

Software maintenance is about modifying the system to correct it or adapt it to new needs (see Chapter 2). Typically, only the source code of a system will be available. It is therefore natural that I started my research looking at this source of information, trying to find ways to help the software maintainers in their tasks.

I considered several aspects of this task that I will report here: How to control the software to understand how it is evolving (Chapter 3); How to prevent some prejudicial situations to arise (Chapter 4); How to correct some issues, particularly at the architectural level (Chapter 5).

I will also show how this research led me to want to work with more abstract data than the sole source code. These consideration will lead us to the next part of the document.

Chapter 3

Measuring to Control

> *"When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind".* — Lord Kelvin

Following the advice of Lord Kelvin, the first step to improve a situation must be to measure it to provide the information required to make key decisions and take appropriate actions in given contexts [McGarry 2001]. In software maintenance, the quality of the system (*e.g.*, its source code, its architecture) can greatly impact the time it takes to understand it, or to introduce a new feature.

In this sense, a good part of my research has been dedicated to software quality metrics and models. I present in this chapter our contributions to the evaluation of software maintenance quality.

One first needs appropriate metrics that can quantify some property of interest in a way that is meaningful to practitioners. This is what I explore in Section 3.1. But many metrics already exist that, once validated, can be used to define quality models allowing to get a broader understanding of the software evolution phenomenon. In Section 3.2, I discuss the issue of creating specific quality models for specific problems (in software maintenance). In Section 3.2.2 I refine the notion of aggregating the results of different metrics on many software components into an overall evaluation.

3.1 Initial Experiment: Definition of Relevant Quality Metrics

The title of this section is misleading. The work presented here *should have been* the first step of the many studies (mine and other) that took place in the domain of architectural design quality. I must confess that my first research in this domain (presented in Chapter 5), took for granted some unproven assumptions. It was only after some time and results that were not entirely satisfactory that I perceived my mistake [Anquetil 1999a, Anquetil 2003b].

The experience, however, was not in vain and, in my following work, I was more careful to appropriately validate my tools before relying on them.

3.1.1 Software Architecture Quality Metrics

I present here a recent experiment [Anquetil 2011] that should have been the starting point of the research presented in this chapter and Chapter 5. Years after first realizing that it was missing, I finally performed it, considering that it was still lacking, impeding the community from making significant advances.

The problem. The architecture of a software system is a key aspect that one has to monitor during software evolution. Due to enhancements and modifications in directions not initially envisioned, systems suffer from architectural drift or architectural erosion. To fight against that, systems may go through a large restructuring effort. For example, the architecture of Eclipse (the IDE) was redesigned in version 3.0 to create the Eclipse Rich Client Platform (RCP, see Figure 3.1).

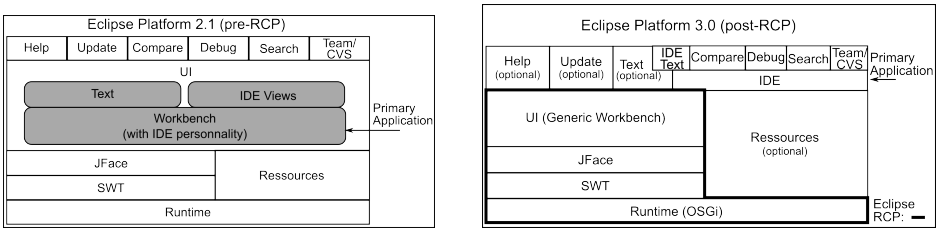

Figure 3.1: Architecture of the Eclipse platform before RCP (v. 2.1) and after RCP (v. 3.0). From a presentation at EclipseCon 2004[2]

Considering the importance that software engineers place on a good architecture, architecture erosion is a problem that must be monitored. An obvious candidate for that is to use some sort of quality metric (this is not the sole solution, see also Chapter 4). Yet there is little agreement on what is a good software architecture[3] and general belief sees it as a subjective topic.

[2] http://www.eclipsecon.org/2004/EclipseCon_2004_TechnicalTrackPresentations/11_Edgar.pdf, last consulted on: 08/08/13

[3] Nice answers to the question "what is a good software architecture?" may be found on http://discuss.joelonsoftware.com/default.asp?design.4.398731.10 (*e.g.,* "It's a bit like asking *What is beautiful?*"). Last consulted on 12/10/2012.

Previous State of the Art. People have relied on various cohesion and/or coupling metrics to measure the quality of software architecture. Good modularization counts as one of the few fundamental rules of good programming. The precept of high cohesion and low coupling was stated by Stevens *et al.* [Stevens 1974] in the context of structured development techniques. Since then the ideas have been transposed to OO programming and continue to hold, be it at the level of classes (*e.g.,* [Briand 1998]) or at the level of packages. According to this rule, systems must be decomposed into modules (or packages, subsystems, namespaces, etc.) that have high cohesion and low coupling. The heralded advantages of a modular architecture include [Bhatia 2006]: Handling complexity; independent design and development of parts of a system; testing a system partially; repairing defective parts without interfacing with other parts; controlling defect propagation; or, reusing existing parts in different contexts.

Many cohesion or coupling metrics may been found in the literature. They may apply to packages or classes. For example, a review of different cohesion or coupling metrics for packages may be found in [Ebad 2011]. Yet, some started to notice that we have little understanding of "how software engineers view and rate cohesion on an empirical basis" [Counsell 2005] (this was for classes); or that "it has been difficult to measure coupling and thus understand it empirically" [Hall 2005]. The same holds at the level of packages [Anquetil 2011].

But the idea that such metrics would be indicators of architectural quality has also been challenged. Brito de Abreu and Goulão stated that "coupling and cohesion do not seem to be the dominant driving forces when it comes to modularization" [Abreu 2001]. A statement with which Bhatia and Singh agreed [Bhatia 2006].

Other researchers considered a more theoretical point of view: "we conclude that high coupling is *not* avoidable—and that this is in fact quite reasonable" [Taube-Schock 2011]; Some coupling metrics have been found to be good predictors of fault proneness (e.g. [Binkley 1998, Briand 1997]), and a model including coupling metrics was shown to be a good predictor of maintenance effort [Li 1993]. But this does not relate directly to architectural design quality.

Cinnéide *et al.* [Cinnéide 2012] proposed a controlled experiment where they introduce random modification in the code and measure the impact on the metrics. They were able to show that the metrics do not agree one with the other. However, they cannot tell which metrics are relevant as they cannot automatically assess whether the random modification improved or hurt the architecture.

Contribution. In collaboration with the group of Prof. Marco T. Valente, we set out to validate cohesion/coupling metrics as architectural design quality metrics. The task is made difficult by the nature of the problem:

- Asking the opinion of experts would be costly on a realistic scale because architecture should be evaluated on large systems. One must also note that, in industry, young programmers are not asked to design the architecture of complex systems and similarly, evaluating an architectural design would require experienced designers, who are even harder and costlier to find ;
- Comparing to some golden standard raises the issue of subjectivity of the solution: for one system, there may be several possible, equally valid, architectures and the validation of any quality metric should take this into account.

To be of interest, evaluation of architectural design must be done on large, real, systems because architectural design presumably depends on many factors, other than cohesion and coupling [Abreu 2001], and the evaluation must consider these additional factors to bear any relevance. Evaluating a few packages out of context could cause an evaluator to base his opinion on too few parameters, thus leading to a possibly unrealistic evaluation.

We proposed a metric validation framework based on a practical approach [Anquetil 2011]. We say that a good architecture is one that is accepted as such by some expert software engineers (experts in the systems considered and in software architecture).

We hypothesize that the modular quality of a software system should improve after an explicit remodularization effort. A similar hypothesis was informally used by Sarkar *et al.* in [Sarkar 2007]. One of the validation of their metrics for measuring the quality of non-Object-Oriented software modularization was to apply them to "a pre- and post-modularized version of a large business application". Considering the time and effort one must invest in a remodularization, such task cannot be started lightly. As such we consider that an explicit remodularization will have a planned target architecture that must be the result of software engineers' best efforts. Even if it were not completely successful, it would still be the result of an expert best effort and as such would still give us a glimpse into what he thinks is a good architecture.

Considering real remodularization cases introduces "confounding factors". For example one must expect that other changes (bug fixes and feature addition) will also occur between the two versions considered. Bug fixes are localized and can therefore be ignored at the architectural level. New

features could be considered to impact the architecture, but those would happen within the context of the remodularization effort and therefore be taken into account in the new architecture. For example, when migrating Eclipse to the RCP, the OSGi framework was introduced in the core runtime (see Figure 3.1). But this was actually one of the goals of the remodularization.

To illustrate the use of this validation framework, I present in Figure 3.2 the results of an experiment with Eclipse and two well known metrics [Anquetil 2011]. The versions considered are: a bug fix version, v2.0 → v2.0.1; some local restructuring in preparation for the RCP migration, v2.0.1 → v2.1; the RCP migration, v2.1 → v3.0; and the maturation of the migration, v3.0 → v3.1. The metrics are those of the Bunch tool [Mancoridis 1999]. It shows that tested cohesion and coupling metrics did not behave as would be expected from the literature. If coupling diminished after both restructurings (v2.1, v3.0), cohesion did so too which is contrary to the traditional understanding of good architecture. The two metrics only show what would be considered an improvement in architecture quality for v2.0.1, a bug fix version.

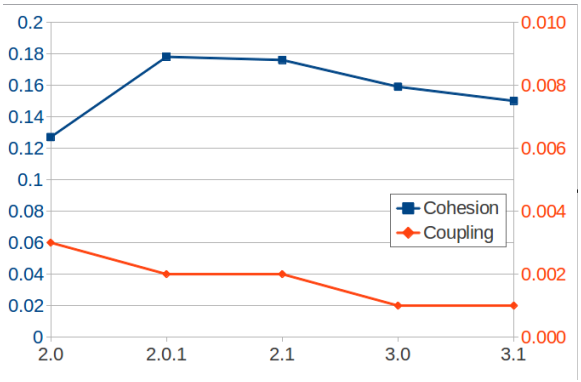

Figure 3.2: Variation of Bunch cohesion/coupling metrics on versions of Eclipse

Perspectives. This first experiment opened the way for more studies. First, the architectural design quality test bed must be amplified and strengthened to verify the early results. An extended experiment has been submitted for publication [Anquetil 2013].

With a better testing framework, one will be able to check all existing architectural design quality metrics and possibly invent and test new ones, for example based on concepts found in identifiers, or based on authors, co-modification of files, etc. A step in this direction has been made, very

recently, by Garcia *et al.* [Garcia 2013] by proposing a set of height architectures that they verified and can be used as ground truths to test and compare architecture recovery techniques.

3.1.2 Metrics as Bug Indicators

In another related case study, with the help of a PhD student co-supervised by me and Prof. Marco T. Valente, we studied a previous validation of some metrics that was flawed and for which we proposed a stronger one [Couto 2012].

The problem. Research on bug prediction indicators wishes to identify predictors that can be used to forecast the probability of having errors in the system. Having this, one would pay attention to keep these predictors below an alarming level.

Software quality metrics are natural candidates as bug predictors. The idea is that the decreasing quality of the source code, measured with the appropriate metrics, would reveal a software that is more difficult to maintain, which, itself, should lead to a higher probability of introducing errors when making changes to the system.

I present and discuss here the results of experiments conducted by a PhD student who I co-supervised with Prof. Marco T. Valente.

Previous State of the Art. Most experiments designed to evaluate bug prediction only investigate whether there is a linear relationship —typically Pearson correlation— between the predictor and the number of bugs. Yet, it is well known that standard regression models can not filter out spurious relations. Correlation does not imply causality. Numerous examples exist: there is a correlation between the number of firemen fighting a fire and the size of the fire but this does not mean that firemen cause an increase in the size of fire. In this case, it is rather the opposite. Other examples exist where there is no direct relationship between the two correlated variables but a third one influences them both.

A statistical test was proposed by Clive Granger to evaluate whether past changes in a given time series are useful to forecast changes in another series. The Granger Test was originally proposed to evaluate causality between time series of economic data (*e.g.*, to show whether changes in oil prices cause recession) [Granger 1969, Granger 1981]. Although more used by econometricians, the test has already been applied in bio-informatics (to identify gene regulatory relationships [Mukhopadhyay 2007]) and recently

in software maintenance (to detect change couplings that are spread over an interval of time [Canfora 2010]).

Testing Granger causality between two stationary time series x and y, involves using a statistical test — usually the F-Test — to check whether the series x helps to predict y at some stage in the future [Granger 1969]. If this happens (the p-value of the test is $< 5\%$), one concludes that x Granger-causes y.

Contribution. In this research we looked for more robust evidence toward causality between software metrics as predictors and the occurrence of bugs. We relied on a public dataset constructed by D'Ambros *et al.* to evaluate bug prediction techniques [D'Ambros 2010, D'Ambros 2012]. This dataset covers four real world Java systems, with a total of 4,298 classes, each system with at least 90 bi-weekly ($\simeq 3.5$ years) versions.

We added a new time series to D'Ambros dataset with the mapping of 5,028 bugs reported for the considered systems to their respective classes. This is done by mining commit messages in the software history to find bug fix changes. For this step one can either search for keywords such as "Fixed" or "Bug" in the commit message [Mockus 2000] or search for references to bug reports [Śliwerski 2005]. Identifying a bug fix commit allows one to identify the code changes that fixed the bug and therefore, where the bug was located in the source code. One defect is one change in the source code that was performed to fix a bug. One bug may correspond to several defects when one needed to change the code in various places to fix the bug. A class is related to a defect if the later occurs within the code of the class (*e.g.*, in one of its methods) [Kim 2007]. Table 3.1 summarizes the main properties of the defect time series. It shows the number of bugs we initially collected in the study (column B), the number of defects that caused such bugs considering all the classes included in the study (column D), and the average number of defects per bugs (column D/B). As can be observed in the table, on average each bug required changes in 2.87 defective classes. Therefore, in our experiment, bug fixes were not scattered.

In the experiment we considered several classical software quality metrics such as Number of Lines of Code (LOC), Depth of Inheritance Tree (DIT), Number Of Attributes (NOA), or Lack Of Cohesion in Methods (LCOM). We eliminated from the series classes that had less than 30 values (around 30% of the time series size) because they were too short lived to allow applying the test. For obvious reasons, we also eliminated series that had only null values (for example a class with no known bug). Other filters were applied to ensure that the series were not stationary, but I will not detail them here (see [Couto 2012] for more details).

Finally, we test whether a metric series Granger-causes the defect series

Table 3.1: Number of bugs (B), defects (D), and defects per bugs (D/B) in four Java system

System	B	D	D/B
Eclipse JDT Core	2398	7313	3.05
Eclipse PDE UI	1821	5547	3.05
Equinox	545	991	1.82
Lucene	264	564	2.14
Total	5028	14415	2.87

with a possible lag varying between 1 to 4. The test is applied with a significance level of 95% ($\alpha = 0.05$).

For 12% of Eclipse JDT Core classes (that main contain defects or not), we have not been able to detect a single causal relation from any of the seventeen series of metrics, for Eclipse PDE UI it is 36%, for Equinox 47%, and for Lucene 30%.

With this method, we have been able to discover in the history of metrics the Granger-causes for 64% (Equinox) to 93% (Eclipse JDT Core) of the defects reported for the systems considered in our experiment. Moreover, for each defective class we have been able to identify the particular metrics that have Granger-caused the reported defects.

Perspectives. This work was more a proof of concept than a real experiment. More work with a stronger experimental setting are still needed to fully validate these initial results.

3.2 Quality Models for Software Maintenance

Individual metrics, as the ones considered in the previous section, are important to monitor the evolution of a single property, but it is now understood that alone, they are not enough to characterize software quality. To cope with this problem, most advanced or industrially validated quality models aggregate metrics: for example, cyclomatic complexity can be combined with test coverage to stress the fact that it is more important to cover complex methods than getters and setters. The combination of various quality metrics to deal with multiple points of view, or more abstract quality properties give rise to the concept of quality models.

We identified two needs in the context of software maintenance. Although there are techniques to help defining quality models, they are difficult to apply. There is therefore a need to define and validate specific quality

models adapted to specific situations. As an example, we worked on the particular case of outsourced maintainers, that wish to price their services for systems they do not yet know (Section 3.2.1). Another issue considers the way quality models aggregate results of many metrics over many software components (Section 3.2.2).

3.2.1 Building a Quality Model from Metrics

The problem. As software maintenance outsourcing is becoming more common, outsourced maintainers are being faced with new challenges. One of them is the ability to rapidly evaluate a system to get an idea of the cost to maintain it. Depending on the type of maintenance contract offered, this cost may be linked to a number of variables: number of maintenance requests, urgency of these requests, their intrinsic difficulty, or the ability of the software system to endure modifications. Rapidly evaluating such variables without previous knowledge of a system is a difficult task that all outsourced maintainers must perform. Polo [Polo 2001] has highlighted this as a problem that was receiving little attention.

We worked on this with a Master student, in collaboration with an outsourced maintenance company.

Previous State of the Art. The metrics defined to evaluate the maintainability of a software system (e.g. in the international standard on software quality, ISO/IEC-9126 [ISO 2001]) were intended to be collected *during* maintenance as a mean to monitor the evolution of the system. Such model is not adapted to the necessity described above.

Misra and Bhavsar [Misra 2003] had "[investigated] the usefulness of a suite of widely accepted design/code level metrics as indicators of difficulty". They analysed 20 metrics over 30 systems "varying widely in size and application domains". Their research used Halstead's Difficulty metric as a basis against which to evaluate the value of other metrics that may be computed early in the development of a software system (design and implementation phases).

Polo *et al.* [Polo 2001] had considered a similar problem, trying to help outsourced maintainers evaluate the fault proneness of a system they do not know. They limited themselves to two metrics: the size of the systems in number of lines of code and in number of programs.

Pearse and Oman offered a model for measuring software maintainability [Pearse 1995]. After studying several large software systems at Hewlett-Packard, they came up with a four metric polynomial Maintainability In-

dex[4]. It seemed improbable to us that any "magic formula" could measure such a complex thing as maintainability. Also, the formula was fine-tuned for the Hewlett-Packard environment and an outsourced maintainer would have to craft a new one for every potential client. Finally, the formula was a black box and as such, it did little to help identifying specific reasons for the poor or good maintainability of the system, a feature which would be needed to negotiate prices with a client.

In collaboration with a Master student working in a private software company, we defined a specific model using the Goal–Question–Metric paradigm (GQM) [Basili 1992]. The GQM methodology helps identify Metrics required to answer Questions that will tell whether a specific Goal was attained.

Contribution. We used the GQM methodology to define a model to assess the complexity to maintain a legacy software system [Ramos 2004]. For an outsourced maintainer, the cost of maintaining a software system may come from several sources:

- High number of maintenance requests per month. As described in [Polo 2001], maintenance contracts may involve attending to an unspecified (or partially specified) number of maintenance requests per month: the more maintenance requests, the higher the maintenance cost.
- Urgency of maintenance requests. Urgent maintenance requests may cost more to tackle because they are not planned.
- Intrinsic difficulty of the maintenance requests. Some maintenance requests may be simple to address, requiring only one small change in a well known part of the system, while others may require restructuring the entire system.
- Difficulty of understanding and/or modifying the system. Depending on the documentation quality, system modularity, programming languages used, data model, etc., the same maintenance request can be easy to implement or on the contrary quite difficult.

In this research, we focused on the last of these issues. An important requisite of the measurement plan is that the metrics should be rapid to collect, giving preference to automated metrics when possible. We identified

[4] $MI = 171 - 5.44 * \ln(\overline{V}) - 0.23 * \overline{CC} - 16.2 * \ln(\overline{LoC}) + 50 * \sin(\sqrt{2.46 * \overline{LoCmt}})$ where $\overline{V}$ is the average Halstead volume metric, $\overline{CC}$ is the average extended cyclomatic complexity, $\overline{LoC}$ is the average number of line of code, and $\overline{LoCmt}$ is the average number of lines of comment.

two goals: (i) Assess the system documentation with respect to its completeness and consistency; (ii) Assess the source code with respect to the complexity to understand and modify it.

For each of the goals, we broke it into more specific questions and defined metrics to answer the questions.

Question 1.1: To what extent is the system documented? (4 metrics)

Question 1.2: Is the documentation coherent with the application domain and the application objectives? (4 metrics)

Question 1.3: Is the documentation coherent with itself? (9 metrics)

Question 2.1: What is the size of the system? (8 metrics)

Question 2.2: What is the amount of dependencies to data within the system? (4 metrics)

Question 2.3: What is the complexity of the code? (7 metrics)

Question 2.4: What is the complexity of the user interface? (3 metrics)

Question 2.5: What is the complexity of interface with other systems? (4 metrics)

The model was fine tuned using five real Cobol systems in maintenance in the company. The value of the metrics for each question were computed and their results compared to the opinion of two experts in the systems.

Perspectives. This problem is still very relevant today. We were contacted at the beginning of 2013 by an organization wishing to outsource its software maintenance and needing a quality model to validate the offers of the potential providers.

There are needs for specialized quality models, one example being a model to evaluate the difficulty of correcting some bug in specific situations (such as correcting architectural violations, or violation of good programming rules)

3.2.2 Metrics Aggregation Strategy for a Quality Model

One problem we did not explicitly address in the preceding work is that of aggregating and weighting metrics to produce quality indexes, providing a more comprehensive and non-technical quality assessment. We used a simple arithmetic mean over all measured components and all metrics.

This approach is simple and intuitive but may have adverse effects (see below). Other simple approaches such as a weighted average may also lead to abnormal situations where a developer increasing the quality of a software component sees the overall quality degrade. We studied this problems with a PhD student, in collaboration with a quality provider company and Prof. Alexander Serebrenik. We compared the Squale quality model to other methods found in the literature and identified some requirements for a metric aggregation method [Mordal-Manet 2011, Mordal 2012].

The problem. Assessing the quality of a software project raises two problems. First, software quality metrics, for example as proposed in the ISO 9126 standard [ISO 2001], are often defined for individual software components (i.e., methods, classes, etc.) and cannot be easily transposed to higher abstraction levels (i.e., packages or entire systems). To evaluate a project, one needs to *aggregate* one metric's results over all the software components assessed. Second, quality characteristics should be computed as a *combination* of several metrics. For example Changeability in part I of ISO 9126 is defined as "the capability of the software product to enable a specified modification to be implemented" [ISO 2001]. This sub-characteristic may be associated with several metrics, such as number of lines of code, cyclomatic complexity, number of methods per class, and depth of inheritance tree. Thus, combining the low-level metric values of all the individual components of a project can be understood in two ways. First, for a given component, one needs to compose the results of all the individual quality metrics considered, e.g., lines of code and cyclomatic complexity. Second, for a given quality characteristic, be it an individual metric or a composed characteristic as Changeability, one needs to aggregate the results of all components into one high level value. Both operations result in information loss to gain a more abstract understanding: individual metrics values are lost in the composed results, and the quality evaluation of individual components is lost in the aggregated result.

Previous State of the Art. The most common techniques used in industrial settings for aggregation of a software metric results are simple averaging and weighted averaging. These two techniques present problems such as diluting unwanted values in the generally acceptable results, or failing to reflect an improvement in quality.

More sophisticated techniques were proposed [Vasa 2009, Serebrenik 2010, Vasilescu 2011, Goeminne 2011] using econometric inequality indices (e.g. Gini [Gini 1921], Theil and mean logarithmic deviation (MLD) [Theil 1967], Atkinson [Atkinson 1970], Hoover [Hoover 1936], or Kolm [Kolm 1976]). The mathematical definition of these indices is given in Table 3.2. Finally,

Table 3.2: Mathematical definition of some econometrical inequality indices, $\bar{x}$ denotes the mean of $x_1, \ldots, x_n$ and $|x|$ denotes the absolute value of x

Index	Definition	Index	Definition
I_{Gini}	$\frac{1}{2n^2\bar{x}} \sum_{i=1}^{n} \sum_{j=1}^{n} \lvert x_i - x_j \rvert$	I_{Theil}	$\frac{1}{n} \sum_{i=1}^{n} \left(\frac{x_i}{\bar{x}} \log \frac{x_i}{\bar{x}} \right)$
I_{MLD}	$\frac{1}{n} \sum_{i=1}^{n} \left(\log \frac{\bar{x}}{x_i} \right)$	$I^{\alpha}_{Atkinson}$	$1 - \frac{1}{\bar{x}} \left(\frac{1}{n} \sum_{i=1}^{n} x_i^{1-\alpha} \right)^{\frac{1}{1-\alpha}}$
I_{Hoover}	$\frac{1}{2n\bar{x}} \sum_{i=1}^{n} \lvert x_i - \bar{x} \rvert$	I^{β}_{Kolm}	$\frac{1}{\beta} \log \left[\frac{1}{n} \sum_{i=1}^{n} e^{\beta(\bar{x} - x_i)} \right]$

the PhD student involved in this research also worked on an empirically defined, industrial model, Squale [Mordal-Manet 2009]. Squale uses an *ad-hoc* approach to compose metrics for a given software component. Typically this first result is normalized into a $[0, 3]$ interval (0=worst, 3=best). Then, for aggregation, the composed values are translated into a new space where low marks have significantly more weight than good ones, the actual aggregation is computed as the average of all transposed marks and the result is converted back to the original scale by applying the inverse weighting function.

Contribution. We contributed to this field by identifying requirements for the aggregation/composition of metrics in practice in industry; and by evaluating the existing solutions against these requirements.

I will not detailed the eleven requirements we identified, but only propose some of the more pertinent in an industrial context:

- *Highlight problems:* A quality model should be more sensitive to problematic values in order to pinpoint them, and also to provide a stronger positive feedback when problems are corrected;
- *Do not hide progress:* Improvement in quality should never result in a worsening of the evaluation. As a counterexample, it is known that econometric inequality indices will worsen when going from an "all equally-bad" situation to a situation where all are equally bad except one;
- *Composition before Aggregation:* Composition should be performed at the level of individual components to retain the intended semantics of the composition. For example, the comment rate assessment, composed from the number of commented lines and the cyclomatic complexity, would already be less meaningful at the level of a class, after aggregation of the two metrics, than at the level of individual methods: a class could have a very complex, poorly commented method and a very simple, over-documented one, resulting in globally normal

cyclomatic complexity and number of commented lines. It would much better to compute comment rate first, and then aggregate the results at the level of a class;

From these requirements (and others not shown here), we evaluated how well different aggregation approaches answered them.

An important requirement was that of aggregation being able to highlight problems. For this, we compared the reactions of different aggregation techniques in the presence of an increasingly large amount of problems. We used a controlled experiment where the amount of problems may be quantitative or qualitative, and we considered two independent variables:

- quantity of problems in a known quantity of good results;
- quality of the problems (badness degree) in a known quantity of perfect results.

The dependent variable was the final result of the aggregation technique.

All experiments started with a "perfect" case consisting of marks of 3 (in the $[0, 3]$ interval). We then introduced "problems" (mark < 3) up to 100% of problems in steps of 10%. To ease the interpretation of the results, we did several experiments limiting the variation of imperfections to $[2, 3[$; $[1, 2[$; $[0.5, 1[$; $[0.1, 0.5[$; and $[0, 0.1[$. The perfect and imperfect marks were randomly chosen from the results of the number of lines of code of Eclipse 2.0 methods normalized to the $[0, 3]$ interval according to the practice at Air France-KLM where the Squale model is in use. For each experimental set-up (percentage of imperfect marks, interval of imperfection) we present the mean result of 10 experiments.

I give here only the result of four aggregation possibilities: arithmetic mean, Squale approach, I_{Kolm} and I_{Theil}. To foster meaningful comparison, the results of the arithmetic mean are repeated in all other graphs in the form of a grey triangle in the background. One can see for example that even with 30% of very bad marks (imperfect methods in $[0, 0.1[$), the aggregated result for arithmetic mean is still ≥ 2, which does not answer the requirement of highlight problems. Squale has better properties with regard to this requirement. I_{Kolm} also behaves nicely as long as not too many marks are bad where, being an inequality index, it starts to improve (because all results are equally bad). I_{Theil} was chosen as an example of an aggregation method that does not perform well regarding this requirement. We do not see it as an issue since this situation is unlikely to occur in real situation.

Perspectives. For this research, we conducted laboratory experiments with randomly generated datasets (from a pool a real individual marks) to

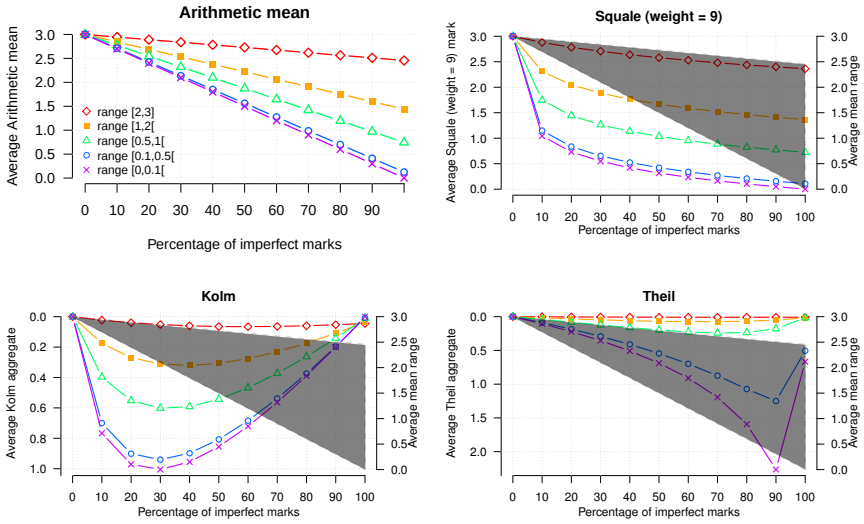

Figure 3.3: Results of experiments for four aggregation indexes (see text for explanation). The topmost left figure displays the common legend.

get a first idea of how the various aggregation solutions would compare. To be complete, it would be important to experiment further with real data and systems and the opinion of the maintainers of the system on the assessment made by the quality model.

Chapter 4

Preventing Software Decay

Measuring the quality of a software system is important to closely monitor its state and understand how it evolves. We explored this option in the previous chapter. Yet this is not enough to ensure sustainability of the evolution process. Another important step is to take actions to slow the decay process, for example by preventing the introduction of bugs into the system. In Section 3.1.2 we started to look at this by trying to use quality metrics as early indicators of software decay and probable bug apparition. Another possible tool is to use rule checkers [Flanagan 2002, Engler 2000, Johnson 1978, Roberts 1997] such as FindBugs, PMD, or smalllint. These are tools that can check good programming practice rules on the source code to prevent specific constructs that are known to be bad practice.

One of their original goals is to prevent bugs, yet, a high number of false positives is generated by the rules of these tools, *i.e.*, most warnings do not indicate real bugs for example Basalaj *et al.* [Basalaj 2006] studied the link between quality assurance C++ warnings and faults for snapshots from 18 different projects and found a correlation for 12 out of 900 rules (1.3%) other studies confirm these findings [Basalaj 2006, Boogerd 2008, Boogerd 2009, Couto 2012, Kim 2007, Kremenek 2004]. There are empirical evidences supporting the intuition that the rules enforced by such tools do not prevent the introduction of bugs in software. This may occur because the rules are too generic and do not focus on domain specific problems of the software under analysis.

One can try to improve their efficiency by finding better rules. We started to explore two solutions for this, first (Section 4.1) by looking at system specific rules, second (Section 4.2) by filtering out the warnings raised by these tools.

4.1 Relevance of System Specific Rules

Rule checking tools could, in theory, help preventing bugs in two ways. First, they could be used to avoid the introduction of specific errors; second, by improving the general quality of the code, they could improve its readability and favour its maintainability, thus lowering the probability of introducing

errors (as confirmed in [Nagappan 2005, Zheng 2006]). In a research led by a PhD student and reported here, we focused on the first approach.

The problem. Knowing that an ounce of prevention is worth a pound of cure, software engineers are applying code checkers on their source code to improve its quality. One of the original goals of these tools is to prevent bugs (hence the name FindBugs). Yet if it is usually acknowledged that if this approach can improve the general quality of the code, its efficiency in bug prevention is debatable.

Previous State of the Art. To understand how well rule checking tools can help prevent bug apparition, studies were conducted to see how the warnings they raise correlate with bug apparition. More specifically, previous research (e.g. [Boogerd 2008, Boogerd 2009, Couto 2012, Ostrand 2004]) studied whether warnings raised by rule checking tools occurred on the same classes, methods, or lines of code as bugs. For this, warnings can easily be ascribed to classes, methods or lines of code depending of the rule considered. For bugs, we used the technique described in Section 3.1.2

In possession of software entities (classes, methods or lines of code) marked with bugs and/or warnings, one is interested in the entities that have both markers. One calls a True Positive a warning collocated with a defect on an entity; an entity marked only as defective is a False Negative; if marked only with a warning it is a False Positive; and, with neither markers it is a True Negative.

The result of such experiment was that most warnings do not indicate real bugs and contain a high number of false positives.

On the other side, a study by Rengli [Renggli 2010] showed that rules defined specifically for a software system or for a domain had more impact (they were more corrected) on the practice than generic one.

Contribution. We hypothesized that the two findings are correlated and that code checking rules are too generic and are not focusing on domain-specific problems of the software under analysis. This is can also be related to studies indicating that the most prevalent type of bug is semantic or program specific [Kim 2006, Li 2006, Boogerd 2009].

Our contribution therefore was to test the efficiency of such system specific rules for defect prevention or reduction [Hora 2012]. We performed a systematic study to investigate the relation between, on one side, generic or domain specific warnings and, on the other side, observed defects:

1. Can domain specific warnings be used for defect prevention?

2. Are domain specific warnings more likely to point to defects than generic warnings?

Of course rules, and especially generic rules, may be created with other purposes than detecting bugs. This is why we performed two experiments answering the above research questions for two sets of rules. In the first experiment we consider all rules, in the second, we focus on "top rules", *i.e.* the ones that better correlate with the presence of bugs. This approach is also used by [Boogerd 2008, Boogerd 2009, Kim 2007] to assess the true positives. In fact, any random predictor, marking random entities with warnings, would, with a sufficient number of attempts, end up with a number of true positives higher than zero, but would not be very useful. Therefore, we can assess the significance of a rule by comparing it to a random predictor [Boogerd 2009]: the project is viewed as a large repository of entities, with a certain probability (p= #defective-entities / #entities) of those entities being defect related. A rule marks n entities with warnings. A certain number of these warnings (r) are successful defect predictions. This is compared with a random predictor, which selects n entities randomly from the repository. We can model the random predictor as a Bernoulli process (with probability p and n trials). The number of correctly predicted entities r has a binomial distribution; using the cumulative distribution function $P(TP \leq X \leq n)$ we compute the significance of the rule [Boogerd 2008]. When the random predictor has less than 5% probability[1] to give a better result than the rule, we call this one a *top rule*. We give some results with top rules in bold in Table 4.1.

The experiment was performed on Seaside, a real software system for which we had system specific rules already defined (the same system used in [Renggli 2010]).

From all methods tested, 77 had at least one top generic warning, from which 13 with at least one defect ($TP > 0$). 67 methods had at least one top domain specific warning, from which 17 with at least one defect. Applying a Mann-Whitney test we got a *p-value*= 0.047 (and effect size= 0.14). We concluded that there was a significant difference between both samples and we could reject the null hypothesis: it is better to use top domain specific warnings to point to defects than top generic warnings.

Perspectives. If we could establish the advantage of using system specific rules for bug prevention, they also suffer from a sever drawback: they must be defined by an expert of the domain under analysis, for each system. We are therefore pursuing this research line by trying to propose a method that

[1] Another threshold could have been used

Table 4.1: Rules with TP > 0. Rules in bold performed significantly better than random (top rules)

Rule	#Warning	#TP
GRAnsiCollectionsRule	8	1
GRAnsiConditionalsRule	**118**	**18**
GRAnsiStreamsRule	11	1
GRAnsiStringsRule	**40**	**10**
GRDeprecatedApiProtocolRule	56	3
GRNotPortableCollectionsRule	**7**	**4**
RBBadMessageRule	16	1
RBGuardingClauseRule	19	2
RBIfTrueBlocksRule	**7**	**2**
RBIfTrueReturnsRule	**14**	**3**
RBLawOfDemeterRule	**224**	**18**
RBLiteralValuesSpellingRule	232	10
RBMethodCommentsSpellingRule	216	8
RBNotEliminationRule	58	1
RBReturnsIfTrueRule	72	3
RBTempsReadBeforeWrittenRule	**16**	**3**
RBToDoRule	**38**	**6**

would help software engineers producing these rules. For this, we are looking for specific changes applied repetitively in the history of the system.

We also have indication that system specific rules could be used to help people migrating from a version of an API to a new one. This is a path that we plan to explore.

4.2 Better Generic Rule Filters

The problem. As we saw in the previous section, on a real sized system, general rule violations detected by the rule checking tools may be numbered by the thousands. Unfortunately, a high proportion (up to 91%) of these violations are false alarms, which in the context of a specific software, do not get fixed. One says they are un-actionable [Heckman 2011], they generate no action from the developers. Huge numbers of false alarms (un-actionable violations) may seriously hamper the finding and correction of true issues (actionable violations) and dissuade developers from using these tools. This can be measured by the *effort* metric [Kremenek 2003]: the average number of violations to manually inspect to find an actionable one, where the closest to 1 is the optimum because it means all violations are actionable. With the assistance of a post-doctoral fellow, we looked for ways to identify actionable violations [Allier 2012].

Previous State of the Art. The literature provides different violation ranking algorithms that try to compute the probability of a violation being truly actionable. Eighteen algorithms were reviewed by Heckman *et al.* in [Heckman 2011] and described according to different criteria like information used, or algorithm used. But they were never compared between themselves, so we had no means to decide which one to use. We needed to formally establish which of the violation ranking algorithms is best and in what conditions.

Contribution. We established a formal framework for comparing different violation ranking approaches [Allier 2012]. It uses a benchmark covering two programming languages (Java and Smalltalk) and three rules checker (FindBugs[2], PMD[3], and SmallLint[4]).

This framework compares violation ranking algorithms on three criteria: effort metric (*i.e.* average number of violations to manually inspect to find an actionable one); whether it is best to work on rules (all violation of a

[2]http://findbugs.sourceforge.net/
[3]http://pmd.sourceforge.net/
[4]http://c2.com/cgi/wiki?SmallLint

Table 4.2: Results for the effort metric for the first 20%, 50%, and 80%. Bold values show the best results (closer to 1).

	PMD			FindBugs			SmallLint		
Threshold	20%	50%	80%	20%	50%	80%	20%	50%	80%
AWARE	**1.01**	**1.03**	**1.36**	**1.13**	**1.14**	**1.35**	**1.01**	**1.02**	**1.03**
FeedBackRank	1.08	1.04	1.75	1.19	1.22	1.45	**1.01**	**1.02**	1.04
RPM	1.9	2.02	2.46	1.17	1.19	1.47	**1.01**	**1.02**	1.11
ZRanking	4.05	4.44	4.2	1.27	1.3	1.84	1.71	1.69	1.67
AlertLifeTime	4.06	3.79	3.72	2.05	1.65	1.76	1.04	1.49	1.69
EFindBugs	2.7	3.01	3.22	1.3	1.28	1.58	1.02	1.14	1.35

given rule are considered equal) or individual violations; and whether the algorithm used is statistical or more *ad-hoc* (characterization from Heckman [Heckman 2011]). For each of these three criteria, we established a formal hypothesis and described how to test the significance of the results.

We then ran our framework on six real systems in Java and Smalltalk; in different domains (*e.g.*, in Java we have JDOM, to manipulate XML data, and Runtime, a plugin in the Eclipse platform, for Smalltalk we have Seaside, a web application framework, and Pharo-Collection, a part of the kernel on collections); and of different size (at least an order of magnitude in both languages).

We compared five violation ranking algorithms (Aware, FeedBackRanking, RPM, ZRanking, and AlertLifeTime) described in [Heckman 2011] plus EFindBugs which is more recent.

I give here only two examples of all the results which can be found in [Allier 2012]. Table 4.2 gives the results of the effort metric for the first 20%, 50%, and 80% of the filtered violations for the six violation ranking algorithms studied. One can first see that the best results give an excellent effort, very close to 1. So for example when filtering violations with the Aware algorithm, if one takes up to 50% of the filtered violations there are good chances that almost all violations are actionable (effort ≤ 1.03 for two rule checking tools, $= 1.14$ for the third).

Another interesting result was considering the violation in order of probability to be actionnable (as reported by the individual algorithms tested) and plotting the percentage of all actionable violations found (*i.e.* recall) versus the number of violations considered (Figure 4.1). It shows that, on violations provided by PMD, three violation ranking algorithms (notably ZRanking) can be worse than randomly filtering, or that FeedBackRank is very good at first, filtering all un-actionable violations, but then reaches a

plateau and starts accepting almost only un-actionable ones.

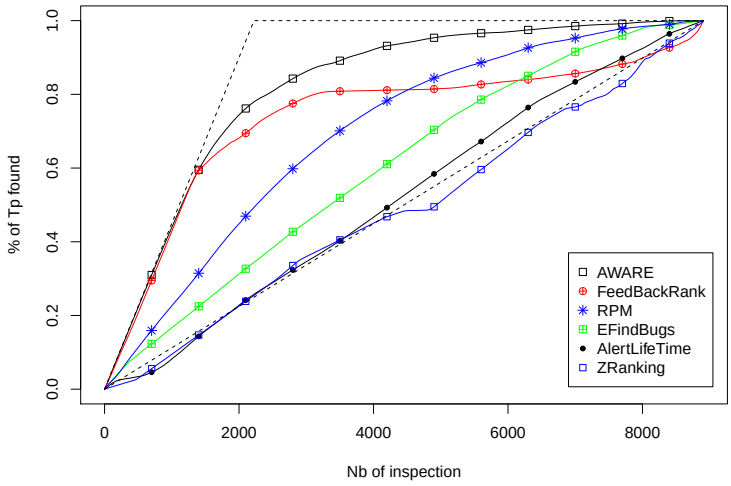

Figure 4.1: Fault detection rate of the violation ranking algorithms tested for the PMD rule checker. Dotted lines represent a random ranking algorithm (lower dotted line) and a perfect ranking one (upper dotted line)

The behaviour of the violation ranking algorithms was not exactly the same for the three rule checking tools but we could establish that:

- For the violations provided by two rule checking tools (SmallLint and PMD), the Aware and FeedBackRank algorithms are significantly better than the other filtering methods while the ZRanking and AlertLifeTime algorithms are significantly worse;
- Overall one should opt for violation algorithms that work on individual violations rather that rules, but one may also consider that the training of the violation ranking algorithms requires more data and therefore effort (training implies that somebody manually check whether violations are actionable or not);
- we found no evidence in favour or against statistical versus *ad-hoc* algorithms.

Perspectives. In the previous section, we successfully showed that more specific rules (system specific) could give a higher percentage of violations pointing to actual bugs. Such violations are by definition actionable, since they point to real bug, one would want to correct them. In this section we

looked at methods to filter the violation candidates on other criteria than just the source code of the system to improve the probability of them being actionable.

An immediate following step would be to verify that actionable violations are also better bug predictor. The two notions are not equivalent, an actionable violation could be corrected for other reason that preventing an error. May be the alert ranking algorithms select violations fitted for another purpose than bug prevention. In either case, the result would be interesting as it would allow people to better understand what can be done with rule checking tools.

Another perspective would be to explore other possible usages of the rules. For example we said in the previous section that system specific rules could be used to help when migrating the API of a library. Another example would be that identifying a good bug prevention rule would immediately lead to the search for a good bug fixing rule.

Chapter 5

Recovering from Architectural Drift

The first two chapters of this part considered methods to control the evolution of a software system by monitoring its quality (Chapter 3); and how one could try to slow the software decay process by preventing the introduction of bugs in systems (Chapter 4). But there is one aspect of a software system that is difficult to prevent from degrading: its architecture.

The solutions proposed in the preceding chapters may be useful at the micro level but they cannot solve all the problems at the macro, architectural, level. Systems are initially conceived with a given goal and scope in mind. As they evolve and new functionalities are added, this goal and scope will typically drift in unforeseen directions. After some time, it becomes necessary to straighten things by redefining a new architecture, taking into account the new scope of the system.

In this chapter, I first present a study of existing methods in software remodularization (Section 5.1) followed by an experiment in the specific case of removing cyclic dependencies (Section 5.2).

5.1 Initial Experiments: Clustering Algorithm to Define Software Architecture

The research described here was performed a long time ago. It's results may appear outdated now, however it must be noted that the problem is still actual and as of today there is still no satisfactory solution.

The problem. The initial assumption in this field of research is that the software engineers are not satisfied with the current architecture (or lack of) of the system. For example, we worked with a company where all the source code was organized as a very large set of files (> 2000) in a single directory. If the organization is a provider of software solutions to others, it will have difficulties to identify what source file each client uses. This can become a real issue, for example when distributing upgrades, because

clients are wary of putting new programs into production, as new programs are always a potential source of new errors.

Previous State of the Art. At the time of this research, a technique often used to help software engineers re-architecture their system was clustering. It was used to gather software components into modules meaningful to the software engineers. For example, Figure 5.1 shows how an hierarchical clustering algorithm gathers individual software elements (e.g. functions in a C program, or classes in a Java one) in a hierarchy of clusters. By cutting the hierarchy at a given level, one obtains a partition of the system where clusters are considered as modules.

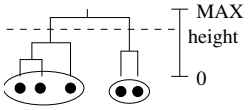

Figure 5.1: A hierarchy of clusters and how to cut it to get a partition of the software elements

Clustering, however, is a sophisticated research domain in itself. There are many alternative methods, the choice of which depends on such factors as size and type of the data, a priori knowledge of the expected result, etc. At the time of this research, reverse engineering was a younger research domain, with still unclear goals [Clayton 1998] and methods that were somewhat *ad-hoc*. In this context, clustering was often used without a deep understanding of many of the issues involved.

A previous publication [Wiggerts 1997] had started to address this issue by presenting a summary of literature on clustering. It listed the possible decisions one needs to make and gave insights about advantages and drawbacks associated with various alternatives. However, Wiggerts' paper only offered conclusions drawn from the general clustering literature, some of which did not apply to the domain of software remodularization.

Another limitation of previous approaches is that they considered the content of the source code as the sole source of reliable information on the system. This approach defined interrelationships between files, and then clustered together the files with strong interrelationships [Lakhotia 1997]. Examples of such interrelationships included calls from routine to routine [Müller 1993, Tzerpo 1997], uses in one file of variables or types defined in another [Müller 1993], as well as inclusion of a particular file by others [Mancoridis 1996, Tzerpo 1997].

Some had started to show that an approach solely based on the source code was impeded by the very low level of information it was based on.

For example, in [Carmichael 1995], Carmichael reported his difficulties in extracting correct design information from a "reasonably well designed", medium sized (300,000 LOCs) recently-developed system. The experiment used inclusion between files to deduce subsystems, but it turned out that each file included many other files, some of these relationships crossing important architectural boundaries. In another paper, Biggerstaff [Biggerstaff 1994] advocated a more human-oriented approach that would actually help the user to relate structures in the program to his "human oriented conceptual knowledge" (see Chapter 6). But it was not clear yet how this could be put into practice on a large scale.

Contribution. With the help of a Master student, we methodically explored the numerous options available when trying to automatically identify high level abstractions (grouping of software elements) in the source code [Anquetil 2003b]. We first set up a conceptual framework where we identified three main points of variation: (i) How to describe the entities to be clustered; (ii) What similarity metrics to use to quantify the coupling between these entities, and; (iii) What clustering algorithms to use.

From this study, we could offer practical suggestions on how to perform "software clustering". This research was very well received and earned two awards: the "Mather Premium", awarded in 2004 by the journal IEE Proceedings—Software (now IET Software), and the "Ten Years Later, Most Influential Paper of WCRE'99" awarded at WCRE'09.

The first point was to define how to describe the entities to be grouped. We compared different description characteristics, some based on the compiled code (*e.g.* routines called by the functions, or header files included by other files), other based on less formal description characteristics such as references to words in identifiers or references to words in comments. At the time, work based on documentation was scarce as a result of assuming that documentation was often outdated.

As a sample of our results, Figure 5.2 allows to compare two description characteristics. The X-axis gives the height of the cut in the hierarchy of clusters (see also Figure 5.1). For each height, we look at the number of software elements not clustered (bottom, light grey), belonging to the largest cluster (top, mid-grey), or belonging to other intermediate clusters (center, dark-grey). In this example, clustering C files according to the header files they included (left hand-side) gave better results than according to the routine that are called from within the files (right hand-side). This was, and is, significant as there is a strong assumption that routine calls are an adequate cohesion/coupling indicator in software architecture whereas here it did not performed very well. This topic as been discussed earlier in Section 3.1.

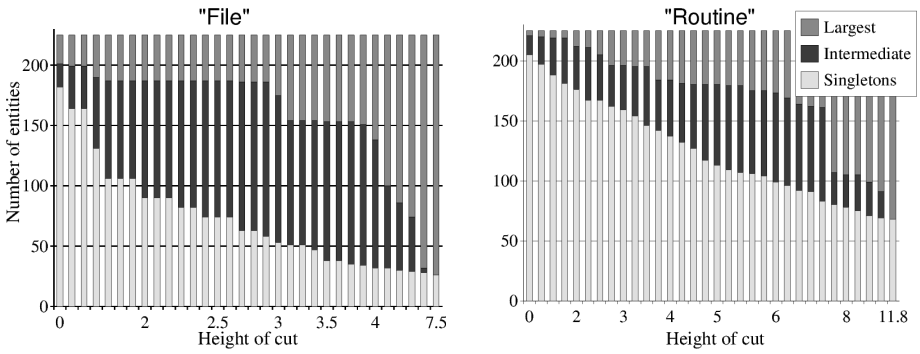

Figure 5.2: Comparison of two entity description characteristics on the size of the clusters. The system used here is the Mosaic web browser

We called a *black hole* configuration when the algorithm tends to create one big cluster that grows regularly during the clustering process and drags, one after the other, all entities to it. We called *gas cloud* configuration when the algorithm create very small clusters and then suddenly clusters all of them into one big cluster near the top of the hierarchy. The right graphic in Figure 5.2 exhibits this undesirable condition. The best configuration is exemplified on the left graphic, where clusters grow evenly during the process.

A second part of the study considered some of the many similarity metrics that may be used to define how close two entities or two groups of entities (i.e. clusters) are one from each other. Clustering algorithm are numerical techniques, they use metrics to decide what entities to group together. We will not enter in details on this issue but were able to propose some suitable metrics, an important conclusion being that a metric should not consider the absence of a characteristic as significant. By this we mean that when clustering animals, the absence of feather is a significant characteristic, whereas in software clustering the fact that some routine does not call another one is not significant because there is too many other routines that share this characteristic.

The last part of the study regarded the clustering algorithm used. We mainly experimented with agglomerative hierarchical clustering (e.g. see Figure 5.1); there are four such algorithms and we identified the best suited depending on whether one wanted to promote cohesion of the clusters or a balance between good cohesion and good coupling.

Yet I believe the main contribution of this research was to identify a fundamental flaw in the whole research area. It was during this research that I first perceived the problem of evaluating the results of automatic re-architecturing. This was already discussed in this document as it led to the

research reported in Section 3.1. This is a message that went unnoticed and research continued for more than a decade without significant results.

Perspectives. This research is at the same time old and still very relevant. The problem of re-architecturing a software system is still an important one. For example, a recent publication [Garcia 2013] reviews some of the technique that were developed during the years. An important conclusion of the paper is that "on the whole, all of the studied techniques performed poorly".

Personally, this work, however successful it was, led me to the conclusion that this research direction, as it was performed, would not succeed. One part of the problem has been treated in a previous chapter (§3.1), we were relying on metrics whose quality was un-tested and that, I now believe, were misleading. Another part is that the goal the community set to itself was probably not realistic. Designing the architecture of a system depends on many conflicting forces, most of which are not directly related to the source code. For example, one can structure a system according to the market segments it addresses, the hardware it requires, the teams that maintain it, historical considerations, plans for the future of the system, etc. Information on these forces cannot be found in the source code and therefore an automatic tool does not have all the data needed to come up with a proper architectural design proposal.

5.2 Approaching a Concrete Case

After the first experiments reported in the preceding section on automated definition of a new architecture, I gained a much better understanding of the domain and acquired the conviction that the research direction was inherently flawed. I chose to go for solutions that would better integrate the semantics of the domain. This will be reported in Part III—People and Knowledge. Yet more recently, I came back to the topic of automatic architecture improvement with new perspectives: First, I am going to the root of the problem by considering the architectural design quality metrics and how we could improve them. This was reported in Section 3.1. Second, with a PhD student, we explored a better defined problem for which quality metrics were easier to define: the removal of package cyclic dependencies [Laval 2012]. I will now describe this research.

The problem. When a large system is well structured, its structure simplifies its evolution. A possible good structure is the layered architecture. A layered architecture is an architecture where entities in a layer may access

only entities in the layers below it. A layered architecture eases software evolution because the impact of a change can be limited to layers above, it also offers good properties of modifiability and portability [Bachmann 2000]. Moreover, using a layered view on a software system is a common approach to understand it [Ducasse 2009].

At first sight, it might seem easy to create a layered organization from an existing program: (i) compute dependencies between software entities; and (ii) put the entities in layers such that no lower layer accesses a layer above it. However, in practice, cyclic dependencies are common in large software applications [Falleri 2011]. For example, the largest cycle of ArgoUML contains almost half of the system's packages, and for JEdit, almost two thirds of the packages are in the largest cycle [Falleri 2011]. The naive algorithm above would place all entities in one cycle (e.g. two third of JEdit's packages) in the same layer. Hence, cyclic dependencies should be dealt with before trying to create a layered organization.

Previous State of the Art. Three main approaches existed to extract a layered structure from package dependencies. To illustrate their results, we consider an hypothetical example (Figure 5.3, left part). The right part of the figure proposes a plausible decomposition into layers. This is actually the one proposed by our solution, so we will come back to it later.

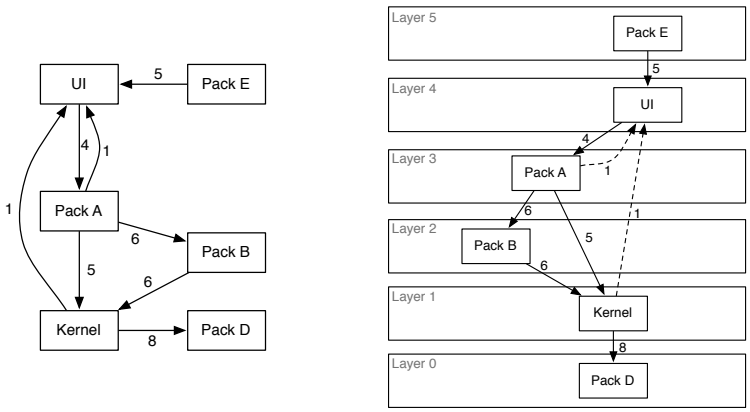

Figure 5.3: On the left, an example of cycles between entities. On the right, a plausible decomposition in layers. The dashed arrows are those that we "removed" to create a proper layer architecture. (Copied from [Laval 2012])

A first solution consisted in leaving all cycles out of the layered structure (Figure 5.4, left) as in the NDepend tool[1] Entities that depend, directly or

[1] http://www.ndepend.com

indirectly, on entities in a cycle are also excluded. The problem of this approach is that, if a cycle involves a core entity (as Kernel in our example), most of the other entities will end up outside the layered organization.

A second approach treated cycles as features, as in Lattix [Sangal 2005] (center of Figure 5.4). Each cycle is put in a separate layer. This may lead to erroneous architecture if a cycle is really a mistake and not a feature. This is the case of Kernel and UI being in the same layer in our example.

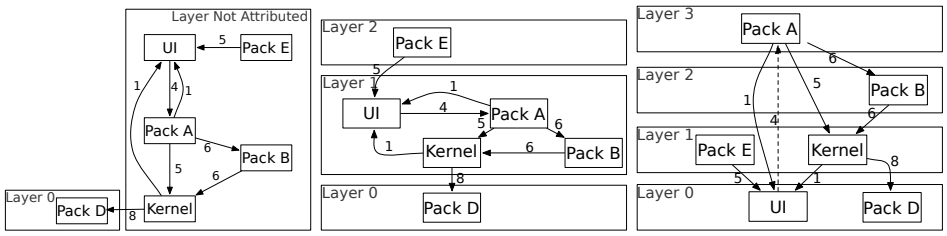

Figure 5.4: Layered organization obtained by different strategies when applied to the system described in 5.3: left=cycles out, center=cycle in one layer, right=MFAS. (Copied from [Laval 2012])

Finally one could try to break cycles using the *Minimum Feedback Arc Set* (MFAS) algorithm as in the Structure101 tool (right of Figure 5.4). In graph theory, a minimum feedback arc set is the smallest set of edges that, when removed from the graph, leaves an acyclic graph. This approach could produce good results because it guarantees to make minimal modifications to the software structure to break cycles. However, it does not take into account the semantics of the software structure: what the entities mean, why they have these dependencies. Optimizing a graph is not equivalent to identifying the layered architecture of an existing software system. In our example, this solution places UI at the bottom of the layer organization, and Kernel in a higher layer, which does not fit common understanding in software engineering.

Contribution. Our approach is based on two intuitions for finding dependencies that, once "removed" (*i.e.,* not considered in building the layered architecture), will allow generating a layered structure. We consider that dependencies between entities have weights. For example, when a package A depends on a package B, it might be because only one class of A inherits from one class of B (weight=1) or because there are 10 invocations of methods in B from methods in A (weight=10).

Our first heuristic states that, in a direct cycle (between two packages), the lightest dependency might be a design defect or a programming error and therefore should be removed. Concretely, if the software engineers wish

to actually put the layered architecture proposed into practice, the lightest dependency often requires the least amount of work to remove. If the two dependencies in direct cycles have the same or a similar weight the tool can only choose randomly the one to remove.

Our second heuristic is based on the occurrence of shared dependencies in minimal cycles involving three or more packages. A shared dependency is one that belongs to several cycles. Shared dependencies have a strong impact on the system structure because removing them may break several cycles at the same time. A minimal cycle is one with no node appearing more than once. We must focus on them to make sure the shared dependency really belongs to different cycles and not simply the same cycle repeated several times.

These hypotheses might not always hold true, so software engineers with knowledge on the system are given the opportunity to overrule wrong decisions made by our tool by flagging individual dependencies either as *undesired* (forces removal) or *expected* (impedes removal).

We validated our approach with a study of two large open-source software systems: the Moose [Nierstrasz 2005] and Pharo [Black 2009] projects. We selected these two projects because they are large, contain realistic package dependencies, and are open-source and available to other researchers. In addition, both systems are more than 10 years old. Most importantly, we selected them because we could get feedback on our results from members of each project community. Engineers of these two projects manually validated the results reported by our tool.

Compared to other approaches, ours contained fewer false-positive and false-negative results producing a layer organization corresponding to the intended structure of the software. This was possible because the software engineers knowing the system were able to give feedback to the tool by flagging as *undesired* some dependencies that the tool first decided to keep (mainly in the case of random removal when two dependencies in direct cycles have similar weights).

We also compared our algorithm with the *Minimum Feedback Arc Set* (MFAS) algorithm on the set of 1328 dependencies of Pharo that were manually classified (expected or undesired) by its developers. The results are that the precision for our approach (dependencies proposed for removal that were classified *undesired*) is a bit better with 64% for our tool and 61% for MFAS. Furthermore our tool allows the developers to fine-tune the results which MFAS does not. In addition, we also got a better recall (proportion of *undesired* dependencies proposed for removal) with 44% for our tool against 41% for MFAS. Hence, our approach performed better than MFAS on this example.

Perspectives. A first continuation on this research would be to put more semantics on the dependencies. For example, on a dependency between package A and package B, it might not be the same to say that a class in A inherits from a class in B or that a method in A invokes a method in B. By taking the individual links into account, we might be able to further improve the results we already obtained.

Another possible research direction would consist in targeting other specific architectural goals such as aiming for a layered architecture with a well defined semantics for each layer: client/server or three-tiers.

Part III

People and Knowledge

In Part II—"Software System" I discussed how I tried to ease software evolution considering solely the software system. One of the conclusions of this part of my research was that some tasks, like evaluating the quality of an architectural design, depended on information that is of a different nature. Biggerstaff [Biggerstaff 1994], see below, called this information "human-oriented concepts" (as opposed to "computer oriented" ones). For example, one may want to decompose a system according to the various market segments it attacks (*e.g.,* hotels, hospitals, or schools), or according to internal divisions in the organization using it (*e.g.,* sales department, logistics department, etc.).

In this part (Part III—"People and Knowledge"), I report on attempts to work with more abstract concepts like what the software does and for who.

Chapter 6

Knowledge Contained in the Source Code

This chapter presents the results of a transitional research, where I still tried to extract data from the software itself. The idea was that if a system is devoted to a given market segment (*e.g.,* hotels), then this might appear inside the code in the form of comments or in identifiers.

In the next chapter (Chapter 7) I will be considering another evolution of this reflection where I tried to work with the knowledge of the developers themselves.

6.1 Initial Experiments: Domain Concepts Found in Identifiers

My first experiment in this new line of research [Anquetil 1998a] was to verify that it was viable. For this, I had to demonstrate that the source code contained information that could be reliably used to get access to concepts relevant to the higher level of abstraction software maintainers deal with. I also qualified the kind of information that could be extracted from this new source of information [Anquetil 2001].

The problem. To work with more abstract information I needed a reliable source of this information. Following my previous experiments (see Section 5.1), I turned to comments and identifiers in the source code. They intuitively represent a possible source of human-oriented concepts since they are intended for human readers. For example, showing all the identifiers appearing in a method gives a better clue as to its goal than giving the source code but obfuscating the identifiers (see an example in Figure 6.1). However, I wished to verify whether this could be reliably used to make inferences on the concepts used.

Previous State of the Art. In a well received article [Biggerstaff 1994], Biggerstaff had coined the expression "concept assignment problem" as the

m1(C1 v1) { C2 v2 = "\\+"; C2 v3 = " "; C3 v4 = C3.m2(v2); C4 v5 = v4.m3(v1); return v5.m4(v3); }	removeDuplicateWhitespace CharSequence inputStr String patternStr String replaceStr Pattern pattern Pattern compile patternStr Matcher matcher pattern matcher inputStr matcher replaceAll replaceStr

Figure 6.1: One piece of code obfuscated in two different ways (Java example found on http://www.tutorialspoint.com/javaexamples/regular_whitespace.htm)

action of relating source code to the *human-oriented* concepts they represented and manipulated. The human-oriented concepts were defined as living "in a rich context of knowledge about the world", and "designed for succinct, intentionally ambiguous communication". They were opposed to the *computer oriented* concepts, "designed for automated treatment" using "vocabulary and grammar narrowly restricted". This was the first explicit identification that the software itself was not all and that other points of view were required.

What I planned to do was slightly different, as I wanted to extract the human-oriented concepts from the source code. There was no clear indication whether this could be done or even what these concepts were exactly.

For example, in the case of identifiers, opinions were contradictory: Sneed [Sneed 1996] reported that "programmers often choose to name procedures after their girlfriends or favourite sportsmen," or "data attributes of the same structure may have different names from one program to another". On the other hand, other researchers (*e.g.,* [Burd 1996], [Cimitile 1997], [Newcomb 1995]) had tried to find synonymous structured types based on their definitions, thereby implicitly assuming that structured type identifiers were not significant (synonymous records) whereas their field names were.

The closest to my goal was a very preliminary study [Clayton 1998] that identified all the "knowledge atoms" that would be necessary to understand a very small program (102 lines of Fortran code). They classified these atoms in three knowledge types: domain knowledge, language knowledge (FORTRAN), and programming knowledge, to which they added five knowledge atoms not related to any of three types. Compared to them, I was more interested in domain knowledge.

Contribution. My first goal was to clarify whether the identifiers and the comments could be reliably used to extract human oriented concepts. This implied defining more precisely what reliability meant in this context. I was basing this work on the assumption that some naming convention was used in developing software through a process whereby software engineers

informally followed each others' lead by creating new names inspired from the ones already existing in the code.

Ideally, one could have required that a naming convention was reliable if there was a strict equivalence between the names of the software artefacts and the human oriented concepts they implement. But that would have been overly restrictive and I limited myself to checking that two software artefacts with the same name did implement the same concept.

Another issue was to verify objectively that two human oriented concepts were the same. I chose to work with structured data types, for which it was easier to compare the implementation and establish a notion of similarity between them.

I was working with a proprietary telecommunication legacy system that was over 15 years old and about 2 MLOCs of Pascal code. The system contained over 7000 structured type definitions (*records* in Pascal) of which 2666 were global, non-anonymous, record definitions, and 97 had a non-unique name.

I first established that the field names were reliable (*i.e.,* two identical field names have identical types) by comparing the names and the declared type of the fields (see Table 6.1). A high proportion (94.8%) of synonymous fields have the same type (same name ⇒ same implemented concept). The proportion of non synonymous fields with different types (89%) is also good (opposite implication, different names ⇒ different implemented concepts). The significance of this result is confirmed by a χ^2 test. I thus established that inside the subset of structured types that have non unique names, the field naming convention is reliable.

		Field types		
		=	≠	total
Field	=	73(95%)	4(05%)	77
names	≠	52(11%)	421(89%)	473

Table 6.1: Paired comparison of fields' names and fields' types within synonymous structured types

I used this first result to compare structured types themselves, by computing a distance between the names of the structured types identifiers and comparing this to the distance between the structured types definitions (words found in the names of the fields, this is possible because we showed that the identifiers are significant). The results were that only 10% of the 77 pairs have a conceptual similarity inferior to 0.6 (from 0, completely different, to 1, equals). Only two pairs of structured records were completely different, they were utility types.

This experiment was also replicated on Mosaic, one of the very first web browsers. I tested variables' names, structured types' names and fields' names and got similarly positive results (although, I did find a few peculiar examples such as: "mo_here_we_are_son", "mo_been_here_before_huh_dad", etc.)

I concluded that, in the legacy software studied, the names of structured types and their fields could be significantly used to extract semantic information from identifiers. Given this result I was interested in finding out whether and how human-oriented concepts could be extracted from the identifiers [Anquetil 2001].

For this I had to establish a classification of knowledge that suited my needs. Based on Biggerstaff [Biggerstaff 1994] and Clayton [Clayton 1998], I proposed three main domains of knowledge: Application domain, Computer science domain and general domain (the rest). Given these domains of knowledge, I studied how well they were represented in software artefacts identifiers and in comments. The study was conducted on the Mosaic system[1].

Then I extracted concepts from the source of information: (i) decompose in words the identifiers and correct misspelled words in comments; (ii) remove utility words with a standard stop list; (iii) associate words to "concept" (*e.g.* the words "alloc", "allocate", "allocator" and "allocation" are all associated to the same concept); (iv) classify each concept in a domain. The first step was challenging, particularly to split identifiers containing unknown acronyms into the proper set of words. This is a problem that I also studied in other work [Anquetil 1997, Anquetil 1998b, Anquetil 1999b] and that is still an active domain of research as of today (e.g. [Feild 2006, Dit 2011]). The last two step were manual and more straightforward. The paper [Anquetil 2001] describe the rules used to classify words into knowledge domains.

I found close to 6200 global identifiers containing 2900 words, and 7818 different words in the comments. After applying the stop list and normalization into concepts I ended up with 1020 different concepts (see Table 6.2). I got 939 concepts common to both documentation sources, the identifier concepts being almost included in the comment concepts with only 14% of concepts only found in identifiers. There was a very high proportion of General concepts (80% for comments, 60% for identifiers) with Computer science and application domain concepts being at similar levels (10% in comments and 20% in identifiers). This was not a welcome discovery as it challenged one's ability to extract relevant application domain concept from the identifiers and/or comments.

[1] One of the very first web browsers

Knowledge domains	Comments #	Comments Avg.	Identifiers #	Identifiers Avg.
Application	286/10%	48.6	226/22%	22.7
Comp. science	323/11%	83.3	190/19%	18.2
General	2389/80%	26.6	604/59%	12.5
All	2998/100%	34.8	1020/100%	15.9

Table 6.2: Number of concepts found in comments and identifiers in Mosaic (left) and average repetition (right). Repetition computed as: For comments, total number words referring to the concept; For identifiers, number of identifiers referring to the concept.

Perspectives. These un-satisfactory results as well as the extreme difficulty I had to manually classify concepts in their respective domains of knowledge convinced me that it would be very difficult if at all possible to extract automatically concepts from the identifiers and comments in the source code. This will be discussed in the next chapter.

There is still a need to be able to relate computer oriented concepts in the source code to human oriented concepts. This is the very essence of reverse engineering and program comprehension. For example, with the growing use of Software Product Line Engineering, work is still on progress on how to generate a software product line from a set of applications from the same domain. This implies "aligning" the different applications so as to identifies their commonalities and the variation points.

As already mentioned, decomposing identifiers into a set of words is still an active domain of research [Feild 2006, Dit 2011].

6.2 Tools and Techniques for Concepts Extraction

Concurrently to this research on where to find the concepts we wanted to extract from the code and its comments, I also studied how this could be done. A possible solution was to use Lattice of Concepts, also known as Formal Concept Analysis.

The problem. To achieve the goal of program comprehension, reverse engineering must be able to abstract details out of the source code of systems and present to the user the important concepts this code contains and deals with. A natural hypothesis is that these important concepts are more

often encountered that mere details[2] such as the font size of a label or the validation of a street address.

Previous State of the Art. To find these important concepts, several grouping techniques exist, such as clustering (we saw examples of this in Section 5.1) of Formal Concept Analysis (FCA). With these techniques, concepts are groups of entities, that may be described by their properties. Wille [Wille 1992] defines a concept as having three components: A name, a list of attributes describing the concept (called the *intent*) and a list of entities belonging to the concept (called the *extent*). A concept can be indifferently referred to by any of these three components.

Formally, one considers *couples* (pairs) of entities and attributes. A concept {E},{I} is a (named) couple with an extent E= <entity-list> and an intent I=<attribute-list>. In a couple, all objects in E possess all attributes in I. To be a concept, a couple must also respect the following requirements:

- there is no entity outside of E that has all the attributes in I;
- there is no attribute outside of I that belongs to all entities in E.

Given the data in Table 6.3 (a context) the couple {F2.c,F3.c},{fopen,free} is not a concept because printf also belongs to F2.c and F3.c which violates the second requirement. {F2.c,F3.c},{fopen,printf,free} is a concept.

Table 6.3: Description of some C files with the routines they refer to

File	Description
F1.c	{sizeof, malloc, realloc, free}
F2.c	{fopen, printf, fprintf, fclose, free}
F3.c	{fopen, fscanf, printf, malloc, free}

Note that one can easily define a generalization/specialization relation between such concepts. Given two concepts $C_1 = (E_1, I_1)$ and $C_2 = (E_2, I_2)$, we will say that C_1 is more general than C_2 iff: $I_1 \subset I_2$ (or $E_2 \subset E_1$).

Formal Concept Analysis is a technique to extract all concepts from a given context: a set of objects and their attributes. The concept are arranged in a Galois lattice structure as shown in Figure 6.2. In the figure,

[2]Note, however, that this hypothesis was never formally tested. Section 6.1 reports on my work in this direction. Also, in [Anquetil 2000b] I had some success with another hypothesis based on opposition rather than repetition.

the concepts are simplified by taking advantage of "inheritance", a property defines in a super-concept is not repeated in a sub-concept; an entity belonging in a sub-concept is not repeated in a super-concept.

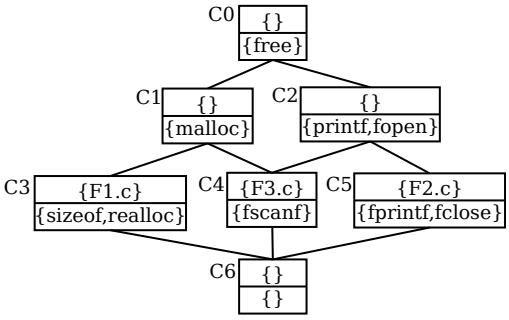

Figure 6.2: Lattice of concepts for the data set in Table 6.3

This structure proved useful for example:[3] for browsing purposes [Godin 1993]; to extracts objects from procedural code; to mine aspects [Kellens 2007]; etc.

As we explain in [U.Bhatti 2012], one of the strengths of FCA for program comprehension and software reengineering is the wide range of contexts (entities and their attributes) that can be used. For each different context, the method provides different insights into reengineering opportunities. Yet it also presents two significant drawbacks. The first one is the number of concepts it extracts, it usually outputs much more information (concepts) than what it was given in input. The ratio can go up to hundreds of times more information output; thus completely missing the point of abstracting important information. The second difficulty is that previously existing approaches left the analysis work to the user. Because Galois lattices are complex abstract constructs, and because they are so big, they often prove difficult to interpret. Sahraoui *et al.* [Sahraoui 1999] recognized this problem and proposed a tool to help analysing the lattices.

In [Anquetil 2000a] I discussed the first issue, while the second was treated more recently in [U.Bhatti 2012] in a collaboration with two other researchers (Dr. Marianne Huchard and Dr. Usman Bhatti).

Contribution. Given a context, *i.e.,* a set of objects and their attributes, there is a finite set of concepts that can be extracted from it. This is the set extracted by the FCA technique, no other concepts can be found. This allowed us to see the result of FCA as a search space where any concept extraction method could look for important concepts. This contributes to set a common base on which to compare all the methods. For example, a

[3](we list some more examples in [U.Bhatti 2012])

method's ability of abstraction could be simply measured in the percentage of concept from the lattice of concepts it generates.

In [Anquetil 2000a] I listed some methods to reduce the number of concepts in the lattice. Using a real legacy software (Mozilla), I compared them according to the number of concepts extracted and the "design quality" of the concepts. This research was largely inspired by my previous work for example as described in Section 5.1.

In [U.Bhatti 2012], we attacked the problem from another direction. We wanted to propose a solution to help people interpret a lattice of concepts. We proposed a catalogue of patterns that represent interesting constructs in concept lattices from the software engineer point of view. These patterns help the user in two ways. First, they help to reduce the work of understanding a complex lattice to interpreting a few node and graph patterns. Hence, the work is reduced to look for these patterns and understand their interpretation. Second, because we consider generic sub-graph patterns, a tool can be built to extract these patterns from a lattice, which will greatly simplify the analysis.

The patterns are defined as specific topology of nodes and edges, sometimes accompanied by a specific type of node for one or more members of the pattern. Some of these patterns are illustrated in Figure 6.3 to help the reader visualize them. The grey upper half of a node shows that the node introduces attributes that its super-concepts do not have, but all its entities appear in sub-concepts. This is the case of C0, C1, and C2 in Figure 6.2. Such a node must have more than one direct super-concept ("multiple inheritance" in OO parlance). In OO parlance, this would be an abstract class that introduces properties but does not have instances of its own. Such node must have more than one direct sub-concept. The black lower half of a node (a concept) shows that the node has entities of its own (that no sub-concept has), but "inherits" all its attributes from its super-concepts.

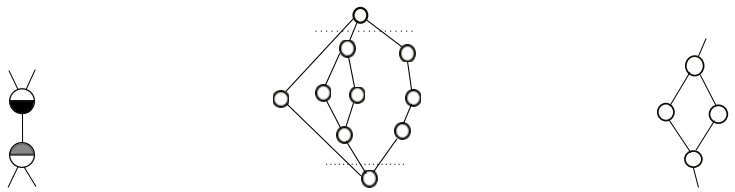

Figure 6.3: Some of our patterns in concept lattices, from left to right: *Irreducible Specialization*, *Horizontal Decomposition*, and *Module*. The black lower half indicates that the concept has no attributes of its own; the grey upper half indicates that the concept has no entities of its own (an "abstract class").

The illustrated examples are (from left to right): an *Irreducible Special-*

ization where the two nodes should not be merged; an *Horizontal Decomposition* where cutting along the doted lines would produce three disjoint sub-lattices; and a *Module* that represents a sub-lattice.

In [U.Bhatti 2012], we propose some patterns, we explain them and explain their meaning in some specific cases. We also proposed a prototype to automatically discover such patterns in a lattice of concepts.

Perspectives. Formal concept Analysis is still a very dynamic field of research. For example, in [U.Bhatti 2012] we identified all possible context uses for reverse engineering source code: what entities could be used and what their attributes could be. As far as we know, only half of the possibilities have been explored which leaves plenty of opportunities to find new interesting applications for this technique.

Chapter 7

What knowledge is Needed in Maintenance?

7.1 Initial Experiment: Observing the Maintainers at Work

The problem. In a preceding work (reported in Section 6.1), I studied the concepts that could be encountered when analysing the comments and identifiers in the source code. This study confirmed that different domains of knowledge could be found. It was, therefore, important to understand what domain of knowledge would be the most important to work on. Different domains of knowledge had been highlighted in past research: Biggerstaff [Biggerstaff 1994] insisted on the necessity of application domain knowledge. Van Mayrhauser and Vans [von Mayrhauser 1994], focused on design decisions (i.e. knowledge about software development applied to the transformation of knowledge on the application domain to produce the source code). Jørgensen and Sjøberg [Jørgensen 2002] showed that sheer maintenance experience is not enough to reduce the frequency of major unexpected problems after maintenance, whereas application experience does.

Previous State of the Art. There was very little work on the knowledge required by software maintainers. I already listed some of it in Section 6.1: the work of Biggerstaff [Biggerstaff 1994] insisted on the difference between human and computer oriented concepts. It implicitly assumed that application domain concepts were the most needed, but this was based on his perception rather than actual study.

Clayton [Clayton 1998] identified the "knowledge atoms" that would be necessary to understand a very small program (102 lines of Fortran code), but this was a theoretical analysis not based on actual or concrete maintenance needs. Also, working on a single small program, allowed them to easily understand the program and application domain, which is hardly ever the case with real world applications, where maintenance is performed with only a partial understanding of both.

Von Mayrhauser and Vans (e.g. [von Mayrhauser 1994, Mayr 1995]) performed some empirical studies of software engineers performing real maintenance tasks. But their goal was to formalize a mental model of how the software engineers understand programs. They were not directly concerned with the knowledge the software engineers use, but rather with the "process" of program comprehension.

Contribution. With the help of two Licence students working in two IT organizations, we performed a field study with real software maintainers during their normal activity [Ramal 2002]. Before starting some maintenance task, a software engineer would call one of the two field experimenters to does a session. The field experimenter would then sit behind this software engineer and record everything he says (micro-tape recorder) or do (pen and paper).

The session usually ended with the end of the maintenance task. In a few occasions, the session ended with the end of the working day and was not completed afterwards (either because the task itself was delayed past the end of the study or because the field experimenter could not be present at the end of it).

Each session was then documented as follows (see also Figure 7.1): The *context* of the maintenance task is described; the type of maintenance operation according to the classical decomposition (corrective, adaptive, perfective or preventive) is specified; the profile (newcomer, intermediate, etc.) of the software engineer who realized this session is given; finally the various "steps" of the task are written down with their approximate time marker. Each step is also numbered sequentially inside its session so that it can easily be referenced afterwards. Figure 7.1 shows the first steps of a session.

Once a session was transcribed to text, we analysed it to identify and classify the knowledge atoms it contained. We used the same high level knowledge domains as in our previous study: Business knowledge, Computer science knowledge, and General knowledge. But they were divided into new sub-domains (e.g. programming, programming language, development environment). I will not detail here these sub-domains, as a more formal organization of knowledge will be presented in Section 7.2.

Each individual knowledge unit identified was called a *knowledge atom*. For example, step 1 in Figure 7.1 is classified as Development Environment knowledge, step 2 is Application Implementation knowledge, and step 3 is Development Environment knowledge (to look for the class, one need to know how to do this in the environment) + Application Implementation knowledge (to know that there is a method to consult the database, one must have some knowledge of the application).

SESSION 1

Context: A query was received to modify the program responsible for sending e-mails in the "Protocolo" system. Currently, this system reads e-mails to be sent in a table called "Emails to send". However, this mechanism is extremely resource demanding for the database (the mechanism is based on the use of triggers). The new implementation uses "stored procedures" which will return the data to be sent. The necessary stored procedures have already been created and the software engineer only needs to modify the client part.

Perfective Maintenance

Profile: A

	Time	Steps
1	08:15	The SE opens Visual Basic
2	08:15	The SE opens the project for this system
3	08:16	The SE *looks for* the class containing a method to consult the DB
4	08:16	The SE opens this class ...
		... and goes to the method
5	08:17	The SE removes the part of this method that references the old tables
6	08:18	The SE starts to write the new code
7	08:20	The SE consults the DB model ...
		... to know the name of the stored procedures s-he will need

Figure 7.1: Description of a software maintenance session (SE=Software Engineer)

The classification in knowledge domains is also marked as positive when the software engineer was using some knowledge, or negative when he was lacking this knowledge. For example, Figure 7.1 step 7, the software engineer "consults the DB model" is classified as positive Diagramming knowledge (he knows how to read the DB model) + "to know the name of the stored procedures" is classified as negative Application Implementation knowledge (he ignores this detail of the implementation).

The two field experimenters were professionals working in a bank (organization 1) and a public administration (organization 2). Six software engineers working in these organizations accepted to participate in the study on a purely voluntary basis.

To get a better insight on the experimental conditions, we established each subject's profile[1]

A: 4 years experience as a programmer; 1 year experience in the organization.

B: 2 years experience as a database administrator (DBA), 8 years experience as a programmer; 1.5 year experience in the organization.

C: 2 years experience as a programmer; 6 months experience in the organization, recently contracted.

D: 3 years experience as a programmer; recently contracted.

F: 2 years working experience; 2 years experience in the organization.

G: 2 years working experience; 1.5 year experience in the organization.

We realized 13 sessions, five in organization 1 (with subjects F and G) and eight in organization 2 (with subjects A, B, C, and D). The sessions recorded short punctual maintenance operations, the average length being of a 1 hour and 15 minutes (minimum: 23 min., maximum: 2 hours 27 min.).

Some of the results were that:

- We identified very few negative knowledge items, mostly concentrated in the Application Implementation domain.
- The positive knowledge most used for each organization are: Programming, Application functionalities, Development Environment, Application Implementation for organization 1; and Development Environment, Application Implementation, Programming, Additional Tools (email, web browser) for organization 2.

[1]There was no subject E

- There was little business knowledge used.

We also studied the relation between the use of a knowledge domains and the moment in a session when it happens. The hypothesis was that Business knowledge could be more important at the beginning of a session when the software engineer tries to understand the system and what he must do, whereas Programming or Programming Language could be more important latter in the session when the software engineer is implementing the solution. From our results, however, it was not possible to conclude that any type of knowledge would be more useful at any given time in a session.

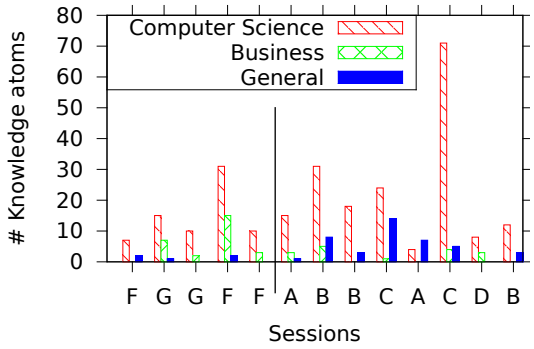

Figure 7.2: Number of positive knowledge atoms identified for each main knowledge domain. Sessions are identified by their subject (left Organization 1, right Organization 2)

AS mentioned above, we were surprised by the small representation of Business knowledge, be it positively or negatively. The only sub-domain really appearing was Application Functionalities knowledge in Organization 1, and this was actually due to two particular sessions. Business knowledge never amounted to more than 33% of the knowledge atoms whereas Computer Science knowledge was below 60% of all knowledge atoms in only one session. Overall, on 345 positive knowledge atoms, only 12% were from the Business domain, 74% were from the Computer Science domain, and 13% from the General domain (see Figure 7.2).

Perspectives. This experiment seriously undermined the idea that Business knowledge was the most needed when doing maintenance. Note that this is not saying that there is no interest in being able to extract such knowledge from the systems maintained. The software maintainers studied did need some business knowledge (of course). Yet it shed doubts on a common assumption, and this showed the need to investigate further this topic. This is what will be discussed in the next section along slightly different lines.

7.2 Formalizing the Knowledge Needed

The problem. Emerging from the research reported in the preceding chapter and this one, was the idea that software maintenance is a knowledge intensive activity [Anquetil 2007]. This was actually implied in studies (cited in [Pfleeger 2002, p.475] or [Pigoski 1997, p.35]) showing that 40% to 60% of maintenance activity involves trying to understand the system. Maintainers need knowledge of the system they work on, its application domain, the organization using it, past and present software engineering practices, different programming languages (in their different versions), programming skills, etc. In software maintenance, the need for knowledge is all the more important because it is no longer acceptable to ask users to re-explain detailed aspects of the application domain, whereas the documentation created during the initial development effort may be either lacking, outdated, incomplete, or lost [Pressman 1994].

Before trying to propose new tools to help software maintainers acquire, persist, recover, or reuse the needed knowledge, one needs to have a clear map of what this knowledge is. For this it was natural to turn to ontologies, *i.e.* descriptions of entities of a given domain, their properties, relationships, and constraints [Uschold 1996] Informally in software engineering, an ontology would be similar to the data model of a domain (*e.g.* a class model).

Previous State of the Art. The problem we encountered in this domain was that considering software maintenance from a knowledge management point of view is not a common perception and very little can be found on this. I already cited (Section 7.1) some work that could give initial insights.

Kitchenham [Kitchenham 2002] proposed an ontology to help classify software maintenance research but it was too restrictive, its goal being to offer a formal framework to classify research in software engineering so as to help compare different results.

Deridder [Deridder 2002] proposed an ontology and a tool to populate this ontology that would help record important concepts, associations, etc. during the development of a software system. This store of information could then be used to help maintenance. The ontology defined was a very high level one. Deridder's objective seemed to focus on the representation of the knowledge and possibly automatic reasoning on it rather than considering what knowledge would be useful.

Another related project by Ruiz *et al.* [Ruíz 2004] was developed in parallel with ours and we were not aware of it at the time of this research. It had a strong emphasis on the aspect of software maintenance process, to the detriment of other domains that we considered.

For this research, we used existing literature on how to define an ontology, for example [Uschold 1996]. There are various methodologies to design an ontology, all considering basically the following steps: definition of the ontology purpose, conceptualization, validation, and finally coding. Conceptualization is the longest step and requires the definition of the scope of the ontology, definition of its concepts, description of each one (through a glossary, specification of attributes, domain values, and constraints). It represents the knowledge modelling itself.

Contribution. I supervised a Master student who defined an ontology using the steps listed above. The *purpose* was to define an ontology describing the knowledge relevant to the practice of software maintenance. The *conceptualization* step was based on a study of the literature and the experience of the authors. We identified *motivating scenarios* and *competency questions* (*i.e.*, requirements in the form of questions that the ontology must answer [Uschold 1996]. It resulted in a set of all the concepts of the ontology and their organization in knowledge sub-domains. The *validation* contemplated quality of the ontology itself (how clear it is, how complete, concise, etc.), and how useful the concepts were for maintenance.

The entire ontology is too extensive to be fully presented here, it has 98 inter-related concepts grouped in five domains of knowledge (see also Table 7.1): knowledge about the Software *System* itself; knowledge about the Maintainer's *Skills*; knowledge about the *Maintenance Activity* (or process); knowledge about the *Organizational Structure*; and knowledge about the *Application Domain* (see Figure 7.3). A better description can be found in [Anquetil 2006] or [Dias 2003a]

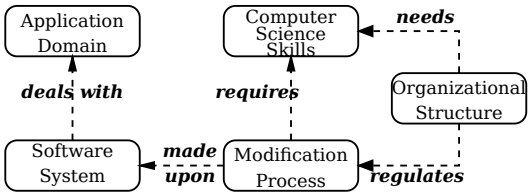

Figure 7.3: Knowledge for software maintenance ontology

Some of the main concepts are: for the System domain, the *system* itself and a description of all its components (*documents* or *software artifacts*); for the Skills domain, the *programming* or *modelling* languages used, the different *CASE tools* (*e.g.* for testing, documenting, etc.) and *IDEs*, also *directives* and *techniques*; for the Activity domain, the *human resources* involved, the *process activities*, and the possible *origin* of the *modification request*; for the Organizational Structure domain, the *organization* and its *human resources*; and for the *Application Domain* domain, the *concepts*

manipulated, their *properties* and the *tasks* that apply to them.

To validate this ontology, we first asked four experts to study the ontology and fill a quality assessment report composed of several questions for different criteria: consistency (absence of contradictory information), completeness, conciseness, clarity, generality, and robustness (ability to support changes). This first evaluation gave good results, on a scale of 0 (bad) to 4 (good), no criterion scored below 3 on average [Anquetil 2003a, Dias 2003b].

Besides the expert assessment, we also validated the completeness, usefulness, and conciseness of the ontology by instantiating its concepts in real situations [Anquetil 2003a, Dias 2003b]. First, we manually instantiated the concepts from the documentation (coming from the development and the maintenance of the system) of a real software system. This validation resulted in the instantiation of 73 concepts out of 98 (Table 7.1, column "Doc.").

Table 7.1: Results of three validations of the ontology on the knowledge used in software maintenance in number of concepts instantiated: from the documentation of a system ("Doc."), during five maintenance sessions using the Think-Aloud protocol ("TA"), and after 17 maintenance sessions filling a questionnaire about the concepts used ("Quest.")

	Concepts	Validations		
Domains	Defined	Doc.	TA	Quest.
Skill	38	28	15	26
Application domain	4	2	1	2
Modification	30	24	16	23
System	23	16	9	13
Organizational structure	3	3	2	3
Total	98	73	43	67

Second, we followed two maintainers during five short maintenance sessions (26 minutes per session on average) using the think-aloud protocol.[2] These sessions were recorded and later analysed to identify the concepts used. The results are given in Table 7.1, column "TA"). The results are not so good, but there were few sessions and they were very short ones.

Third, we asked four software engineers to fill in, every day, a questionnaire on the concepts they used. This form consisted of all the concepts we had instantiated in the first validation ("Doc.") and the list of their instances (as we identified them). The maintainers were simply asked to tick the instances they had used during the day. They could not add new instances. They filled 17 forms in 11 different days over a period of 10 weeks.

[2]The maintainers were asked to say everything they did and why they did it

The results (Table 7.1, column "Quest.") are good considering that the reference is the concepts instantiated in the first validation (column "Doc.").

Perspectives. This part of my research is still very relevant to today state of the art. Tools are needed to help people communicate their understanding of an application to each other, either through documentation or direct communication. There is still no satisfying solutoin on how to extract, gather, store, and then reuse information about a system, what it does, for who, or how it is implemented.

For example, one could think about new issue trackers that would help people fill-in the needed information [Breu 2010, Sillito 2008], or contextual help in an IDE [Ko 2007]. In this case, our ontology could serve different purposes:

- Organization and formalization of the knowledge needed when performing maintenance to serve as a common basis for information exchange;
- Identification of the scope of the knowledge needed to allow checking the completeness and coverage of some information source;
- Definition of concepts that may be used as an indexing scheme to access relevant sources of information;
- Identification of the knowledge needed to ground a search for more information, to identify the most pressing needs, and to categorize possible sources of information according to the needs they may fulfil.

Part IV

Organization and Process

After studying the software systems maintained (Part II—"Software System") and the people that are performing maintenance on them (Part III—"People and Knowledge"), I turn now to the third, and last, part of the software maintenance ecosystem: the organization within which the people work and the working processes this organization enforces.

The organization and the process are an important aspect of any software engineering work and therefore of software maintenance. For example, they make up a significant part of the ontology of the knowledge needed in maintenance (see Chapter 7).

Chapter 8

Knowledge management processes

This chapter presents three studies on processes for knowledge management in software maintenance. First, we carried out a preliminary study to understand how software maintainers do their work (Section 8.1). This study yielded some results on the knowledge used by the people studied. Second, we defined a process to extract, store, and recover specific knowledge during maintenance activities (Section 8.2). And third, we defined an agile maintenance process with a study of the minimal documentation required to do maintenance on the system afterwards (Section 8.3).

8.1 Initial Experiment: Studying the Practice

The research reported in this section is part of the work done jointly with Prof. Timothy C. Lethbridge and Dr. Janice Singer to provide better tools to software maintainers. We mainly worked with the group in a telecommunication company that was maintaining a large system which was one of the key products of the company. The system included a real-time operating system and interacted with a large number of different hardware devices. It contained several million lines of code in over 8000 files. It was also divided into numerous layers and subsystems written in a proprietary high-level language. The group studied comprised 13 people actively working on various aspects of the system. Over 100 people had made changes to the source code during the life of the system.

The problem. To more effectively help software maintainers performing their work, one must first understand precisely what they are doing on a day to day basis, what are their needs, and how current tools fulfill these needs. In our case, we were called on by a telecommunication company to help them improve the maintenance activity of a very large system they were selling.

One human-computer interaction approach to the design of tools is to study the cognitive processes of programmers as they attempt to understand

programs [von Mayrhauser 1995, von Mayrhauser 1996]. Such studies were intended to provide the basis for designing better tools. However they presented deficiencies that I will now explain.

Previous State of the Art. There were three problems with the previous approach:

First, most research had been conducted with graduate and advanced undergraduate students serving as expert programmers. It was not clear (and is still not) that these subjects accurately represented the population of industrial programmers. Consequently, the results of studies involving students could not be generalized to programmers in industry.

Second, to control extraneous variables, researchers had used very small programs (both in terms of lines of code and logic) relative to industrial software. We already referred to this problem in Sections 6.1 and 7.1. This poses a generalization problem as well: it is not clear that approaches to comprehending small programs scale up to the comprehension of very large programs.

Third, there is an assumption that understanding the programmer's mental model is an efficient route to designing effective tools. However, it is not at all obvious how to design a tool given a specification of the programmer's mental model. It does not tell us what kind of tool to build, or how to integrate that tool into the workplace or the programmer's work.

Contribution. The first thing that struck us when starting this research was that we did not know exactly what it was that the software engineers did on a day-to-day basis. That is, we knew neither the kinds of activities they performed, nor the frequency with which these various activities took place. As far as we could tell, there were many hypotheses about the kinds of things they did, but no clear cataloguing as such of exactly how they went about solving problems. Consequently, we decided to begin our study of work practices by finding out what it is that software engineers did in their work.

Our approach to this problem was to implement different data collection techniques and see if the evidence from each converged. We collected five basic types of work practice data. First, using a web questionnaire, we simply asked the software engineers what they did. Second, we followed an individual software engineer for 14 weeks as he went about his work. This person joined the company short time after the project started and it was a great opportunity to understand how newcomers integrated into the organizational culture and got to know the system they were working on. At that time, estimation of the managers was that it took six months for a new

Table 8.1: Questionnaire results of work practices (6 subjects)

Activity	% of people
Read documentation	66%
Look at source	50%
Write documentation	50%
Write code	50%
Attend meetings	50%
Research/identify alternatives	33%
Ask others questions	33%
Configure hardware	33%
Answer questions	33%
Fix bug	33%
Design	17%
Testing	17%
Review other's work	17%
Learn	17%
Replicate problem	17%
Library maintenance	17%

employee to become knowledgeable enough to be fully productive. Third, we individually shadowed eight different software engineers for one hour as they worked. Fourth, we performed a series of interviews with software engineers. Finally, we obtained company-wide tool usage statistics.

Results from the first data collection technique (web questionnaire) are reported in Table 8.1. Six of the 13 software engineers of the group filled in the questionnaire. These answers were self-reported by the software engineers and could not be much relied upon. Also, these results only show what software engineers did, but not with what frequency and how long.

Shadowing the newcomer gave some expected and unexpected results. For example searching, interacting with the hardware and looking at the source code were the three events most likely to occur in his daily activity. All this is normal as the software engineer was getting accustomed to the system at this period of time. On the other hand, he only looked at documentation on 2 of the 14 days, thus indicating that this was not a primary source of information for him.

Shadowing eight other software engineers (not newcomers) likewise gave expected and unexpected results. Looking at the source code, searching (with the "grep" Unix command, or the editor functionality), and interacting with other developers were the events that occurred on most days, and issuing a UNIX command or editing source code were the events occur-

ring most. Again, reading the documentation, although done by 6 of the 8 software engineers studied, occurred rarely overall.

This seemingly lack of interest in documentation prompted us to conduct another study that will be reported in Section 8.3.

Finally, the results for the fifth data collection technique (company-wide tool usage statistics) are reported in Figure 8.1. Overall, 79,295 separate tool calls were logged from the Sun operating system.

The overwhelming finding from the company data is that search (using primarily the Unix "grep" command or one of its derivatives: fgrep, egrep, ...) is done far more often than any other activity. It must be noted that compiler calls, which accounted for 41% of all calls are not included here. This is because the compiler data include all the automatic software builds done nightly and by the various testing and verification groups. These data are therefore not representative of the real work practices of the software engineers studied.

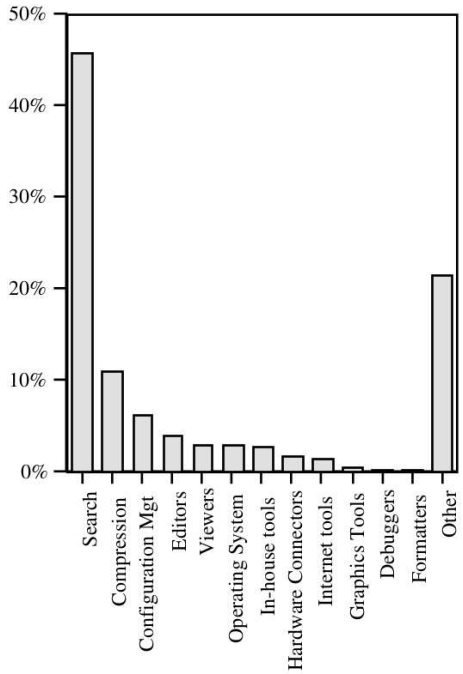

Figure 8.1: Proportion of all tool calls accounted for by each tool type[1]

Compression and un-compression tools were also used often although we never actually observed anyone using these tools.

As a result of this study, we observed that almost all the software en-

[1] At the time of this research the internet was still a relatively new tool, little used and with little resources.

gineers studied spent a considerable proportion of their total working time in the task of trying to understand source code prior to making changes. This is in agreement with other research that I reported in Chapters 2 or 7 stating that maintenance includes a large part of code understanding and knowledge acquisition. We called this approach to software maintenance: Just in Time Comprehension.

Just in Time Comprehension occurred regardless of the task performed, whether defect fixing or feature addition. In either case the maintainers had to explore the system to identify where to apply the modifications. A second factor that seemed to make relatively little difference to the way the task is performed was the level of expertise of the developer: novice or expert. Novices were not familiar with the system and had to learn it at both the conceptual and detailed level; experts knew the system well, but were still not able to maintain a complete-enough mental model of the details. The main differences between novices and experts were that novices were less focused.

Perspectives. One direct result of this study was to show the importance of knowledge discovering techniques during software maintenance. It precedes and justifies all the work that was reported in Part III—"People and Knowledge".

This study is now 20 years old, one cannot expect that all its results are still relevant today considering the tremendous changes that we have witnessed in software engineering (*e.g.* Model Driven Development, Service Oriented Architectures, Cloud) or in general (*e.g.* popularization of the Internet). There is therefore a renewed need to perform such a study to understand the new software maintenance environment, and the activities it imposes on software engineers.

8.2 Collecting Maintainers' Knowledge

Having organized all the knowledge items that were useful to software maintenance (Section 7.2) and knowing the difficulties to extract it automatically from the source code (Section 6.1) we turned to another path and explored the possibility of getting the needed information from the maintainers themselves. With the help of a Masters student, we defined a process to collect knowledge from the experience of the software engineers.

The problem. It should be clear to the reader by now that developing and maintaining software systems is a knowledge intensive task. One needs knowledge of the application domain of the software, the problem solved by

the system, the requirements for this problem, the architecture of the system and how the different parts fit together, how the system interacts with its environment, etc. More often than not, this knowledge is not documented and lives only in the head of the software engineers. It is, therefore, volatile and an organization may need to repeatedly pay professionals to rediscover knowledge it previously acquired and lost.

Previous State of the Art. To help managing this knowledge, techniques such as the Postmortem Analysis (PMA) were used in software projects [Pfleeger 2002, Birk 2002]. PMA is a technique by which a team gathers after the end of a project to identify which aspects of the project worked well, and which worked badly [Stålhane 2003]. These positive and negative aspects of the project are then recorded to help in future projects. PMA was used, for example, in the Experience Factory [Briand 1994] to improve a software process.

A study of the literature (see for example [Birk 2002, Rising 1999, Stålhane 2003]) showed that PMA had been mainly used in software development projects with a particular view on process improvement. We proposed to use the same technique for software maintenance projects [de Sousa 2004, Anquetil 2007] not only to improve the maintenance process, but also to gain more knowledge about the system maintained. One of the great advantages of the technique is that it may be applied on a small scale with ew resources (*e.g.*, a two hour meeting with all the members of a small project team, plus one hour from the project manager to formalize the results), or a larger scale, with a complete team in meeting over a period of several days [Collins 1996].

Contribution. To define our PMA model for software maintenance, we had to consider three aspects: (i) when to insert PMA during the execution of a typical maintenance process; (ii) what knowledge should we look for in the PMA; and, (iii) how to best extract this knowledge from the software engineers.

First, maintenance projects may be of widely varying size, they may be short in the correction of a very localized error, or very long in the implementation of a new complex functionality, or correction of a very diluted problem (e.g. the Y2K bug). For small projects, one may easily conduct a PMA at the end of the project without risking losing (forgetting) important lessons. But for larger projects (as proposed by Yourdon [Yourdon 2001]), it is best to conduct several PMAs during the project so as to capture important knowledge before it becomes so integrated in the participants' mental models that they cannot clearly remember the details. As a generic process with which to experiment, we decided to use the ISO 12207 maintenance process [ISO/IEC 1995] (see also Figure 8.2) and identified moments where

intermediary PMAs could be inserted:

- One intermediary PMA after the analysis of the modification, which includes the first two activities (Process implementation and Problem and modification analysis) and the initial tasks of the third activity (Modification implementation: requirement analysis).
- Another intermediary PMA after the implementation of the modification, which includes the rest of the third activity (Modification implementation).

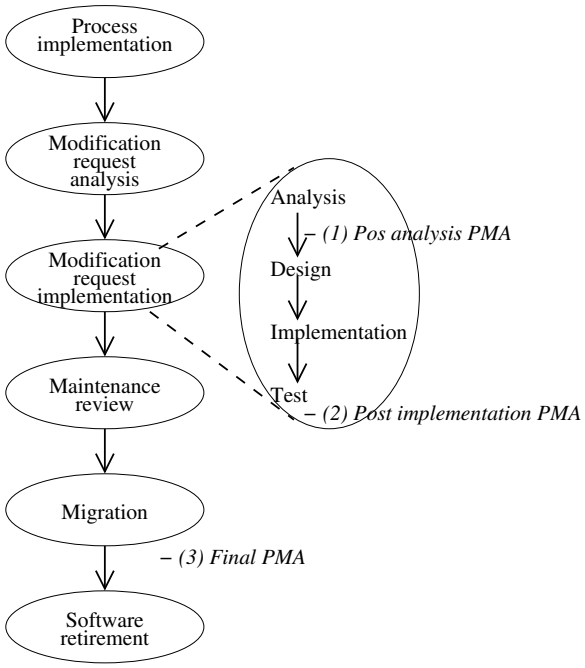

Figure 8.2: Overview of the ISO 12207 Maintenance process [ISO/IEC 1995] with the intermediary and final PMAs

A last PMA could then be conducted at the end of the project to review all its aspects and the most recent activities not yet considered in the intermediary PMAs.

Second, on the knowledge that should be looked for in each PMA, we obviously used as a basis our ontology (Section 7.2). The knowledge categories to consider in each PMA are listed in Table 8.2. They were based on the tasks and activities preceding them.

Finally, we had to define a method that would help the software engineer remember all they could have learned in the various knowledge domains considered (process, system, application domain, etc.). For this we decided

Table 8.2: The three maintenance PMAs and the knowledge categories they focus on

PMA	Knowledge category
(1) Post analysis	Details on the modification request
	Organizational structure using the software
	Options for implementing the modification
	Negotiation of time limit to do the modification
	Effort estimation for the modification
	Documents modified
	Requirement elicitation technique used
	Tools used
	Application domain
	Details on the requirements
(2) Post implementation	Programming languages & tools used
	Programming techniques used
	Software components modified
	Systems interrelationship
	Analysis/design inconsistencies
	Re-engineering opportunities detected
	Artifacts traceability
	Database design
	Design patterns used
	Testing technique used
	Process and support documentation modified
(3) final	Negotiations with other technological departments
	Modification monitoring
	Maintenance process
	Application of the PMAs

to perform the PMAs in two steps: First, we designed a questionnaire as a means to pre-focus their minds on the bits of information we wanted to discover. This questionnaire was distributed among the software engineers that would participate in a PMA session. In a second step, we conducted a PMA session where the same topics were brought up again to effectively discover the lessons learned.

We experimented with this process on six real maintenance projects from the industry. Yet, one must realize that validating a knowledge management approach in general is a difficult thing as the results only appear on the long run, and even then, it may be difficult to pinpoint a single event that clearly shows the benefit of the approach. The experiments were realized in a public organization, where the software maintenance group includes about 60 software engineers (managers, analysts, programmers, DBA, etc.). The methodology was tested on a specific group of 15 people, responsible for the maintenance of seven legacy systems. The maintainers had been briefed before hand on the goals of the PMAs, particularly that it was not intended to be a witch-hunt. Actually some experimental PMAs had already been conducted in the organization previously with the same group (for example, one of the validations of the ontology reported in Section 7.2).

We applied semi-structured postmortem interviews to four short maintenance projects hat involved few maintainers (one or two):

- Project 1: Perfective maintenance, involved 2 maintainers during 6 days for a total of 27 man-hours of work.
- Project 2: Perfective maintenance, involved 1 maintainer during 5 days for a total of 17 man-hours of work.
- Project 3: Perfective maintenance, involved 2 maintainers during 5 days for a total of 47 man-hours of work.
- Project 4: Perfective maintenance, involved 2 maintainers during 2 days for a total of 10 man-hours of work.

In Table 8.3 we present an overview of the number of concepts that could be instantiated during the PMAs (This can be compared to the previous experiment we did, reported in Section 7.2). For example, of the 23 concepts in the System sub-ontology, 11 were instantiated, which means that at least one concrete example of these concepts was mentioned during the PMAs as something that was learned and worth remembering.

From Table 8.3, we can see that the Process sub-ontology is the one that was the most instantiated, in number of concepts (21) and number of instances (135). The organization had just undergone (2 or 3 months before) a major redefinition of its working practices and the issue was still a fresh and

Table 8.3: Concepts from the ontology instantiated during the four PMAs

sub-ontology	Number of concepts	Instantiated concepts	Number of instances
System	23	11	80
Process	30	21	135
CS skills	38	05	09
Organization	03	03	22
Application domain	04	04	68

sensitive one. Some concepts from this sub-ontology were not instantiated due to the characteristics of the projects. For example all four projects were perfective maintenance, therefore the concept Corrective maintenance could not be instantiated in these cases.

This is also the case for many CASE[2] sub-concepts (there are 16 in the sub-ontology) which were not used or do not exist in the case considered (e.g. concept Debugger).

With only four maintenance projects, we were able to instantiate almost half of the concepts from the System sub-ontology (11 instantiated for a total of 23) with many instances (80). We saw it as a very positive sign.

Because of the typical conditions of legacy software systems (foremost the lack of system documentation), many concepts from the System sub-ontology could not be instantiated. This is the case of many Document sub-concepts (there are 16 in the sub-ontology).

Knowledge on the Application Domain and the Organizational Structure was gained during the PMAs. All concepts from these two sub-ontologies were instantiated and, more importantly, many instances were found, especially in the case of the Application Domain sub-ontology (68 instances).

Finally, the lesser results for the Computer Science Skills sub-ontology is seen as natural and with little impact. From the 9 instances found, 4 were to mention interviews as instance of the concept Requirement elicitation technique. The other instances, refer to the discovery of the minus operator in a relational database environment (concept Programming technique); the use of a new class from the programming language library (concept Programming technique); two instances of new testing approaches (concept Testing techniques) and the discovery of a functionality of the modification request management tool ClearQuest (concept Supporting CASE). We felt it was natural that experienced software engineers discovered new knowledge about computer science techniques or CASE tools.

[2]Computer Aided Software Engineering

Perspectives. Recovering knowledge from software maintenance projects to help future maintenance is still a very important topic. However, considering the cost of realizing PMAs and the lack of awareness in industry about these issues, it is very difficult to implement solutions such as the ones described here. We did not pursue much further this line of research ([Torres 2006]). At this moment, it seems preferable to look for more automated solutions (see Chapter 6) even if their results can never be as rich as the one we got in the experiments described here.

8.3 An Agile Maintenance Process

This research was conducted with the assistance of a Masters student.

The problem. From our previous research, I concluded that it was necessary to store and redistribute knowledge on the software systems maintained, the business rules automated by these systems, and the organizations that use them. A natural and well known solution to store knowledge in software engineering is to use the various documentation artefacts that were proposed over the years by the different software development approaches.

Yet in another study (Section 8.1) we noticed that this documentation did not seem to be considered important by developers, who spent very little time consulting it, preferring to look at the source code. We thus needed to find out how to document software systems in a way that would better fit the needs and habits of software developers. Ultimately, we wanted to focus on documents that they were familiar with, that they would be willing to consult, and that would not be heavy to maintain so that it can be kept up to date at a reasonable cost.

Previous State of the Art. Among all the recommended practices in software engineering, software documentation long had a special place. It is one of the oldest recommended practices and yet is renowned for its absence (e.g. [Sousa 1998]). For years, the importance of documentation has been stressed by educators, processes, quality models, etc. but it is not generally created or maintained (e.g. [Kajko-Mattsson 2001]). According to Ambler [Ambler], software documentation responds to three necessities: (i) contractual, for example when the client requires a given CMMi accreditation from a software company; (ii) support a software development project by allowing team members to gradually conceive the solution to be implemented; and (iii) allow a software development team to communicate implementation details across time to the maintenance team. This view was shaken by the advent of agile processes proposing an approach to software development

that greatly reduced the need for documentation as an helper to software development. Using informal communication (between developers and with users), code standardization, or collectivization of the code, agile methods propose to realize the communication necessary to a software development project on an informal level. However, they do not remove the need for documentation as a communication tool through time, that allow developers to communicate important information on a system to future maintainers.

Better defining what document(s) software maintainers need had already been considered in other studies:

- Tilley [Tilley 1992, Tilley 1993], stressed the importance of a document describing the hierarchical architecture of the system;
- Cioch *et al.* [Cioch 1996] differentiated four stages of experience (from newcomer to expert) with different needs: newcomers need a short general view of the system; apprentices need the system architecture; interns need task-oriented documents such as requirements descriptions, process descriptions, examples, or/and step by step instructions; finally, experts need low level documentation as well as requirements descriptions, and design specifications;
- Rajlich [Rajlich 2000] proposed a re-documentation tool that allowed to gather the following information: notes on the application domain, dependencies among classes, detailed descriptions of a class' methods;
- Ambler [Ambler] recommended documenting the design decisions, and a general view of the design: requirements, business rules, architecture, etc;
- In a workshop organized by Thomas and Tilley at SIGDoc 2001, they stated that "no one really knows what sort of documentation is truly useful to software engineers to aid system understanding" [Thomas 2001];
- Forward and Lethbridge [Forward 2002], in their survey of managers and developers, found the specification documents to be the most consulted whereas quality and low level documents to be the least consulted.
- Grubb and Takang [Grubb 2003, pp.103-106], identified some information needs of maintainers according to their activities although few specific documents were listed: managers need decision support information such as the size of the system and/or the cost of the modification; analysts need to understand the application domain, the requirements of the system and have a global view of the system; designers need

architectural understanding (functional components and how they interact) and detailed design information (algorithms, data structures); programmers need a detailed understanding of the source code as well as a higher level view (similar to the architectural view).

- Finally, according to Teles [Teles 2004, p.212], the documents that should be generated at the end of an Extreme Programming project are: user stories, tests, data models, class models, business process descriptions, user manuals, and project minutes.

Contribution. In this research, we wanted to identify a small set of documentation artefacts, that would be easy to keep up to date or to recreate if they were missing [de Souza 2005, Souza 2006]. For this, we distributed a questionnaire to software maintainers asking them to rate the importance of various documentation artefacts in helping understanding a system maintained. The questionnaire was available on paper and on the Internet. The selection of the subjects was done by convenience on a voluntary and anonymous basis.

The main part of the questionnaire asked the subjects to answer the following question for a list of documentation artefacts: "Based on your practical experience, indicate what importance each documentation artefact has, in the activity of understanding a software to be maintained". Four levels of importance where proposed: 1="no importance", 2="little importance", 3="important", and 4="very important". The subjects could also indicate that they did not know the artefact.

The documentation artefacts were divided by activities of a typical development process, discriminating for each activity artefacts specific to the structured analysis (e.g. context diagram), object-orientation (based on the Unified Process, e.g. use case diagram), or both (e.g. Entity-Relationship Model). The complete list of 34 artefacts, as they were presented in the questionnaire is the following:

Requirement elicitation: For *structured analysis:* (1) requirements list, (2) context diagram, (3) requirement description. For *OO development:* (4) vision document, (5) use case diagram. For *structured and OO:* (6) conceptual data model, (7) glossary.

Analysis: For *structured analysis:* (8) functions derived from the requirements, (9) hierarchical function diagram, (10) data flow diagram. For *OO development:* (11) use cases specifications, (12) class diagram, (13) activity diagram, (14) sequence diagram, (15) state diagram. For *structured and OO:* (16) non functional prototype, (17) logical data diagram (MER), (18) data dictionary.

Design: For *structured analysis:* (19) architectural model, (20) general transaction diagram, (21) components specification. For *OO development:* (22) collaboration diagram, (23) components diagram, (24) distribution diagram. For *structured and OO:* (25) physical data model, (26) functional prototype.

Coding: For *structured and OO:* (27) comments in source code, (28) source code

Test: For *structured and OO:* (29) unitary test plan, (30) system test plan, (31) acceptance test plan.

Transition: For *structured and OO:* (32) data migration plan, (33) transition plan, (34) user manual.

Seventy-six software maintainers, from various parts of Brazil, answered the questionnaire. They formed an heterogeneous population with 20 managers (26%), 48 analysts (63%), 5 programmers (7%), and 3 consultants (4%). Twenty-two of them only knew structured development (29%), 6 only knew OO development (8%), and 48 knew both (63%). Seventeen had 1-3 years of experience (22%), 19 had 3-5 years (25%), 17 had 5-10 years (22%), and 23 had more than 10 years (31%). Twenty-six had worked on 1-5 systems (34%), 15 worked on 6-10 systems (20%), 15 worked on 11-20 systems (20%), and 20 worked on more than 20 systems (26%).

Tables 8.3 give the result of the survey for structured analysis and object-orientation development paradigms. They are ranked in decreasing order of percentage of "very important" answers among all those who expressed an opinion.

Overall, and merging the two approaches, the most important documentation artefacts to help understanding a system prior to maintaining it seem to be source code and comments, then a data model, and information about the requirements (requirement list/description, use case diagram/description, acceptance tests). It was a surprise to see that general views of the system (e.g. architectural model, vision document) ranked low (structured analysis: 18th, 19th, and 23rd; object-orientation: 18th, 22nd, 24th).

Perspectives. This survey first confirms the importance of the source code and comments as the primary source of information on the system. This is something I already discussed in Section 8.1.

It also highlights the importance of a view on the data model. Knowing this, one could endeavour to extract a data model from the types and variables used in a software system. There is work to group individual data

Table 8.4: Importance of documentation artefacts 76 software maintainers. First column marks artefacts specific to OO development, Structured Analysis (ST) or common to both.

	Structured analysis artefact	Not known	Important? no	little	yes	very	% very imp.
both	Source code	2	0	0	5	69	93%
both	Comments	1	0	5	11	59	79%
both	Logical data model (MER)	3	0	3	15	55	75%
OO	Class diagram	13	0	3	20	40	63%
both	Physical data model	4	0	2	26	44	61%
ST	Requirement description	4	3	7	18	44	61%
OO	Use case diagram	12	0	7	21	36	56%
ST	Requirement list	5	6	9	17	39	55%
OO	Use case specification	15	1	4	25	31	51%
both	Acceptance test plan	6	6	9	20	35	50%
both	Data dictionary	4	1	11	25	35	49%
both	Conceptual data model	5	4	8	27	32	45%
both	User manual	4	6	11	24	31	43%
both	System test plan	4	4	11	27	30	42%
both	Implantation plan	3	5	10	28	30	41%
both	Unitary test plan	7	5	13	25	26	38%
both	Data migration plan	7	7	11	26	25	36%
ST	Data flow diagram	6	5	12	29	24	34%
OO	Sequence diagram	12	0	6	37	21	33%
both	Functional prototype	6	7	15	26	22	31%
OO	Activity diagram	14	3	8	32	19	31%
ST	Component specification	10	4	10	32	20	30%
ST	Architectural model	11	5	15	26	19	29%
OO	Vision document	20	2	13	25	16	29%
ST	Context diagram	7	4	25	23	17	25%
ST	Hierarchical function diagram	10	5	15	30	16	24%
both	Glossary	4	5	21	29	17	24%
ST	Functions derived from requ.	22	4	17	21	12	22%
both	Non functional prototype	9	8	13	32	14	21%
ST	General transaction diagram	21	5	14	25	11	20%
OO	Component diagram	18	5	12	31	10	17%
OO	State diagram	17	6	17	28	8	14%
OO	Distribution diagram	19	5	18	30	4	7%
OO	Collaboration diagram	21	6	17	29	3	5%

into classes (*e.g.* [Falleri 2008]), but this should be improved and extended, for example by looking for relationship between the data.

The importance of requirement description is also made clear. Again this is an interesting research issue. There has been plenty of work on feature assignment (mapping source code to the user features it implements, *e.g.* see [Poshyvanyk 2007]) but this still requires a lot of manual work.

Chapter 9

Maintenance management processes

In this chapter, I will consider other processes for software maintenance. We will consider how to establish and maintain traceability links between the source code and development documents (Section 9.1), and an identification and categorization of the main risks encountered during software maintenance (Section 9.2).

9.1 Managing traceability links

Another approach to help software developers understand how a system was designed and conceived is to establish traceability links between all the artefacts of the software development process, down to the source code.

The problem. Traceability [Cleland-Huang 2003, Dömges 1998, Egyed 2002, Ramesh 1998, Ramesh 2001] — *i.e.,* the possibility to trace software artefacts forward and backwards along the software lifecyle — is an important and practical aspect of software engineering. The main advantages of traceability are: (i) to relate software artefacts and corresponding design decisions, (ii) to give feedback to architects and designers about the current state of the development, allowing them to reconsider alternative design decisions, and to track and understand errors, and (iii) to ease communication between stakeholders.

Traceability is often mandated by professional standards, for example, for engineering fault critical systems, such as medical applications. However, many existing tools and approaches were limited to requirements management (for instance RequisitePro or works in [Egyed 2002, Ramesh 2001]), they relied on using and integrating various tools [Asuncion 2007, Dömges 1998], they proposed very limited analysis [Moon 2006], or they were not scalable. Additionally, industrial approaches and academic prototypes did not address end-to-end traceability yet, *i.e.,* spanning the full software engineering lifecycle. The use of traceability is considered a factor of success for soft-

ware engineering projects. However, traceability can be impaired by various factors ranging from social, to economical, to technical [Asuncion 2007].

This research was conducted with various European colleagues in the context of the AMPLE research project on Software Product Line Engineering (SPLE) [Pohl 2005], Model Driven development and Aspect Oriented Development.

A software product line is a software system aimed at producing a set of software products (applications) by reusing a common set of features, or *core assets*, that are shared by these products. In SPLE a substantial effort is made to reuse the core assets, by systematically planning and controlling their development and maintenance. Thus, a peculiarity of SPLE is variability management [Berg 2005, Mohan 2007, Moon 2006], that is, the ability to identify the variation points of the family of products and to track each product variant. In contrast to single system software engineering, SPLE yields a family of similar systems, all tailored to fit the wishes of a particular market niche from a constrained set of possible requirements. The software product line development process consists of two main activities (see Figure 9.1): *Domain engineering* and *Application engineering*. These activities are performed in parallel, each with a complete development cycle, consisting of, for example, requirements engineering, architecture design and implementation. The complexity of SPLE poses novel problems (*e.g.,*, variability management) and also increases the complexity of traditional software engineering activities, such as software architecture and traceability.

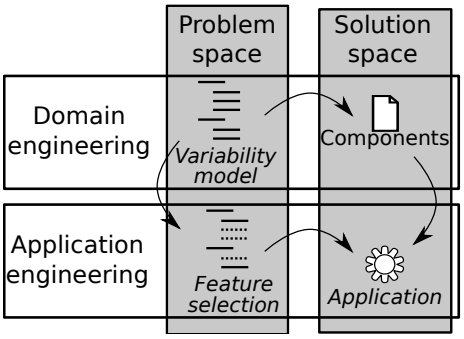

Figure 9.1: Domain and Application Engineering in Software Product Line

Previous State of the Art. We conducted a survey on industrial tools that support some degree of traceability. The goal of the survey was to investigate the current features provided by existing tools to assess their strengths and weaknesses and their suitability to address SPL development and maintenance. The tools were evaluated in terms of the following criteria: (i) management of traceability links, (ii) traceability queries, (iii) traceability

views, (iv) extensibility, and (v) support for SPL, MD Engineering (MDE) and AOSD. These criteria are important for this kind of tool as they provide the basic support to satisfy traceability requirements (which are the creation of trace information and the ability to query the trace links), easier variability management, adaptability to projects specific needs [Dömges 1998], or concerns regarding evolution of these tools and SPL development.

In terms of "link management", the tools allowed defining them manually, but also offered the possibility to import them from other existing documents, such as, MS-Word, Excel, ASCII and RTF files. For the "queries" criterion, the tools typically allowed to query and filter various types of artefacts, mainly the requirements. We also found the ability to use advanced query mechanisms, such as detecting some inconsistencies in the links or artefacts definition, establishing an impact analysis report, or detecting of orphan code. The traceability tools offered different kinds of "views", such as, traceability graphical tree and diagram, and traceability matrix. All of them also allowed navigating over the trace links from one artefact to another. In terms of "extensibility", some allowed specifying new types of reports or even creating new types of links and personalized views. The main ones provided support to save and export trace links to external databases. However, as could be expected, these tools did not support SPLE explicitly, yet.

Two of the leading industrial tools in SPL development, GEARS[1] and pure::variants[2], had some extensions to allow integration with other commercial traceability tools. However, they lacked the ability to deal explicitly with SPL development specificities such as, managing and tracing commonalities and variabilities for different SPL artefacts, or dealing with change impact analysis.

To complete our analysis, we reviewed the academic approaches supporting traceability for product lines or system families. Only three of them provide some sort of tool support [Ajila 2004, Jirapanthong 2005, Mohan 2002]. But none of these approaches provides a clear and comprehensive view of the trace links in an SPLE development.

From this survey we conclude that a thorough analysis of the dimension in SPL was needed, with specific emphasis on variability and versioning.

Contribution. We defined a set of orthogonal traceability dimensions to manage traceability in software product lines. The analysis was based on the traceability needs identified in the literature.

We started by reusing the two traditional traceability dimensions (Re-

[1] www.biglever.com
[2] www.pure-systems.com

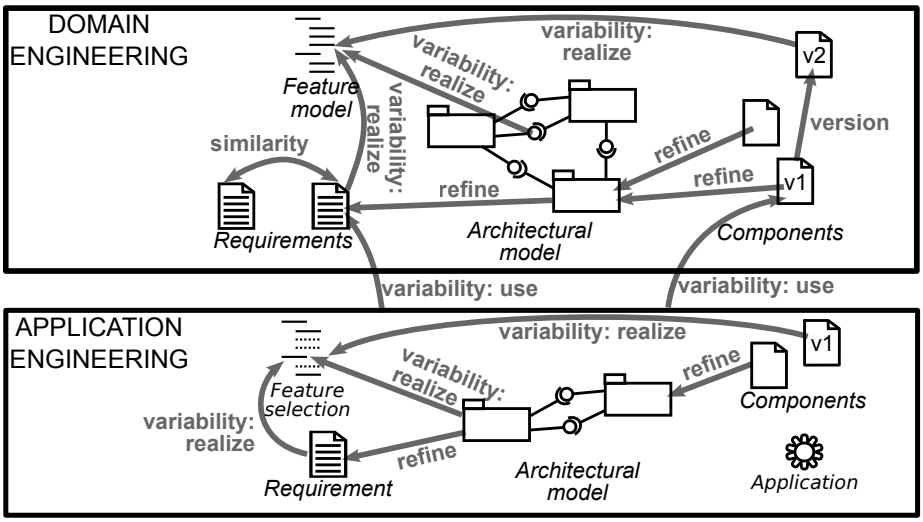

Figure 9.2: Examples of the four orthogonal traceability dimensions (grey arrows) in the two processes of a software product line.

finement and Similarity traceability, see below). We added a dimension to account for variability traceability as suggested by many. Finally, we also found a need for tracing the evolution of artefacts (*i.e.*,, Software Configuration Management).

I now summarize the four traceability dimensions, which are illustrated in Figure 9.2.

- *Refinement traceability* relates artefacts from different levels of abstraction in the software development process. It goes from an abstract artefact to more concrete artefacts that realize the abstract one. For example, a design model *refines* a software requirement. Such links may happen in either of the two development stages of software product lines: domain engineering or application engineering.

- *Similarity traceability* links artefacts at the same level of abstraction. For example, UML components and class diagrams can specify the SPL architecture at different levels of detail but at the same level of abstraction (software design). The trace links defined between these artefacts can be used to understand how different classes, for example, are related to specific components (or interfaces) of the SPL architecture. The links are inside either of the two software product line processes: domain engineering or application engineering.

- *Variability traceability* relates two artefacts as a direct consequence of variability management. For example, at the domain engineering level,

a variability traceability link would relate a variant with the artefact that "realizes" (or implements) it. Or, an application artefact (application engineering level) would be related to its underlying reusable artefact at the domain engineering level. For example, a use case model and a feature model can be related to illustrate which functional requirements are responsible to address the SPL common and variable features. Such traceability links allow understanding how SPL features are materialized in requirements and find potential related candidates during a maintenance task.

- *Versioning traceability* links two successive versions of an artefact.

These ideas were implemented using a model driven approach to integrate with the other tools of the AMPLE project. We modelled our ideas in ECore (the Eclipse implementation of Model Driven Engineering) and generated a Java implementation as an Eclipse plugin. Our traceability metamodel basically defines `TraceableArtefacts` and `TraceLinks` between these artefacts as fundamental elements. The TraceableArtefacts are references to actual artefacts that live in some kind of source or target model or just arbitrary elements created during the development phases of an application like, a requirement in a requirements document. TraceableArtefacts and TraceLinks are typed to denote the semantics of the elements themselves. Using Eclipse gave us flexibility in adding plugins around this core implementation. Several plugins were developed independently by the members of the project: `Trace Register`, to populate a repository; `Trace Query`, to implement complex advanced queries that should be of interest to end-users (e.g., impact analysis); and `Trace View`, to visualize the result of a query.

Perspectives. The two research topics considered in this section (traceability and SPL) are important for software maintenance research.

As explained above, traceability is useful to keep trace of requirements, design decisions, and others and their consequences on the source code. So it is a knowledge management tool that helps software maintainers understand how the system was implemented and why it is so. Unfortunately, keeping strict traceability is very expensive and difficult. Very few systems have this information available. In software maintenance, there has been some research on how to recreate some of these links automatically, for example by relating a class to the textual documents describing it (*e.g.* [Antoniol 2000, Antoniol 2002]), or relating a feature (functional requirement) of a system to the source code that implements it (*e.g.* [Antoniol 2005, Chen 2000, Eisenbarth 2001]). These early attempts are far from having exhausted the subject. One could try, for example, to relate some source code to a UML model of it, even if they do not match perfectly.

Software product line engineering (SPLE) received great interest in the software engineering area, both from practitioners and researchers. For software maintenance, SPLE brought interesting questions, such as how to automatically extract an SPL description from various systems in the same application domain. This is still an unresolved issue.

9.2 Managing the Risks in Maintenance

Departing a bit from the knowledge management line of research that I presented up to now, I also worked on the management of maintenance projects themselves. This research was performed with the help of a Masters student.

The problem. A risk, in a software project, is an event whose occurrence is uncertain and that could impact the project (typically negatively, but one also speaks of positive risks) if it should occur [Jalote 1999, Jalote 2002, McManus 2004, Schneidewind 2002].

Risk management for software is considered as one of foremost best practices in software projects [Addison 2002, Brown 1997]. As such, methods for risk identification (e.g. [Carr 1993, Keil 1998, Machado 2002]), or for risk management (e.g. [Boehm 1989, Jalote 2002, Houston 2001, Kwak 2004, McManus 2004, Murphy 1996]) received lot of attention. However, there had been very little study on risk management for software maintenance projects [Charette 1997].

Maintenance presents specificities that set it apart from software development [Charette 1997, Grubb 2003]. In terms of risk management, this implied that existing methodologies might not be adequate for maintenance projects, whereas reports suggested that risk management would contribute to improve the quality and efficiency of software evolution [CCTA 2000, Schneidewind 2002].

Previous State of the Art. Keil *et al.* [Keil 1998] studied various systems after delivery and established that many problems encountered could have been avoided if risk management had been applied during the projects. To increase the chances of success of a software project, one must constantly monitor possible risk factors, foresee possible solutions, detect as early as possible the occurrence of a risk, evaluate its severity and actual impact, and apply the needed correction.

Risk identification consists in enumerating all possible risks for a project (before they occur) [Jalote 2002]. There are different ways to identify risk

factors (*e.g.*, brainstorming sessions, fishbone diagrams, etc.) but a popular one is to use a taxonomy of possible risks from which one identifies those that may apply to a particular project.

Contribution. To offer a first help in managing risk of software maintenance projects, we defined a taxonomy of risk factors [Webster 2005]. This was necessary as even Charette *et al.* who stated that risk factors for software maintenance are different [Charette 1997], still used the taxonomy of risk factors proposed by Carr *et al.* [Carr 1993] which focused software development.

We conducted a survey of risk management for software development and found 382 risk factors in 18 publications (not listed here for the sake of space). As a comparison, we found only one reference for software maintenance [Schneidewind 2002] listing 19 risk factors.

First, we analysed the list of 382 risk factors for software development found in the literature. We removed duplicates to get a list of 198 risk factors. Second, from these, we removed all risk factors that were not referenced by at least two authors. The idea was eliminate those that were possibly proposed in a too specific context. Our list of risks was now down to 56 risk factors for software development. Third, studying the relevant literature, we established a list of 91 risk factors and problems for software maintenance and again removed the duplicates. As there were much fewer risk factors for software maintenance, we did not filter them on the number of references. This gave us 54 risk factors extracted from maintenance problems. Fourth, we integrated the two lists of risk factors resulting from steps 2 and 3 to establish a final list of 92 risk factors for maintenance.

To complete the creation of the taxonomy, we organized these 92 risk factors into categories. We followed the organization scheme proposed by the SEI taxonomy [Carr 1993] since this is the work that was the most cited. This taxonomy is organized in three levels (names were adapted to software maintenance): The Product Engineering class (technical risks) consists of the activities required to maintain the product; the Maintenance Environment class (methodological risks) is concerned with the project environment, where the software is being maintained; the Program Constraints class (organizational risks) consists of the risks external to the project, factors that are outside the direct control of the project but can still have major effects on its success.

Perspectives. This research in itself did not close the topic of managing risks in software maintenance projects. Even though risk identification for software maintenance can be improved with our taxonomy, the risks

encountered may have a different importance in maintenance than development (for example a stronger impact) because of the differing conditions in which maintenance is performed. More work could be done, for example in studying the "typical" impact and probability of each risk in many projects. This would help maintenance managers get gradually acquainted with the taxonomy by initially focusing on the risks that are typically most critical.

Part V

Perspectives

Chapter 10

Closing considerations

In this document I summarized most of my research in the past years on software maintenance. I considered three main aspects of the activity: the software system maintained, the people that maintain it, and the organization in which this is taking place. I showed how each of these three dimensions were addressed.

For the software system, I presented solutions to control its evolution with software metrics, prevent its decay with rules, and rejuvenate (its architecture) when evolution had to happen in new directions.

One conclusion of this aspect of my research was that we cannot work solely at the level of the source code and must deal with more abstract data that reside in the mind of the software engineers working on the systems. I discussed propositions to recover the knowledge gained by the software engineers in their work, and organize it in well defined categories.

Finally I also treated the problem at the level of the organization, considering the software processes used, and how they could be improved: first to help managing the knowledge on software maintenance and the systems, but also more basic process for managing maintenance itself.

I am closing this document with some longer term research directions, some of which have already been mentioned in the previous chapters.

An important problem that I hardly touched in this document but that I could perceive while working with professionals is that software maintenance suffers from a very negative perception in the workplace. It is recognize as an important activity in volume, but typically considered as less rewarding that development of new software. One must also admit that academia seems to have some responsibility: Research focuses on finding new development methodologies (*e.g.* aspects, model driven development, software product lines), new languages, improved tools, but rarely considers the problems related to maintaining legacy software (*e.g.* developed in Cobol).

Also, software maintenance is rarely taught in typical courses, which may contribute to give the impression that it is not an important activity. This is an important issue because it hampers the search for better solutions. The solution for this would require better teaching of the activity but primarily a better understanding of its reality. This in turn means more study of

the practice of software maintenance to understand the typical problems it involves. This is something we alluded to in Section 8.1.

As discussed in the previous chapters, tools are needed to help understand software systems at a high level (concept location, feature location), but software maintainers also need help for other activities.

There are no tools to help doing large refactorings, for example to split a package in two, moving classes, introducing new abstractions, or reorganizing classes. Modern IDEs do offer localized refactoring operations (renaming a class, changing the signature of a method), but these would need to be chained in the right order to achieve larger refactorings, with risks of forgetting something, inadvertently doing operations in the wrong order, or getting lost in the chain of operations to perform.

In a similar line of research, developers need tools to help them break very long methods (or functions). In industry, cases of methods with thousands of lines of code can be found and there are very few propositions on how to refactor these. This problem is more complex than the previous one because, on top of the necessity to identify and separate different concerns, it requires a fine understanding of the data and control flows which involves extremely complex analyses of data flow, especially in the presence of pointers.

Lastly, I started to look at model driven engineering and how models could help in achieving the old dream of round-trip engineering: from the source code to an abstract representation that is modified and then re-implemented into source code.My experiments with the Moose software analysis platform (based on MDD) show that this idea is flawed with an intrinsic incompatibility. The interest of models here would lie in the fact that they give an abstract view on the system that would allow to concentrate on the important aspects. Apart the fact that we do not know yet how to tell what is irrelevant from what is important, the problem is that working from the source code of a running system is mainly interesting if we can take advantage of this code and all the details it contains. Therefore one would like at the same time abstract away the details, but also keep them to be able to reimplement the system! There is a fundamental difficulty here that will be difficult, if ever possible, to solve.

Bibliography

[Abreu 2001] Fernando Brito Abreu and Miguel Goulão. *Coupling and Cohesion as Modularization Drivers: Are We Being Over-Persuaded?* In CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, pages 47–57, Washington, DC, USA, 2001. IEEE Computer Society.

[Addison 2002] Tom Addison and Seema Vallabh. *Controlling Software Project Risks - an Empirical Study of Methods used by Experienced Project Managers.* In Proceedings of SAICSIT, pages 128–140, Republic of South Africa, 2002. South African Institute for Computer Scientists and Information Technologists. ISBN:1-58113-596-3.

[Ajila 2004] Samuel Ajila and Badara Ali Kaba. *Using Traceability Mechanisms to Support Software Product Line Evolution.* In Du Zhang, Éric Grégoire and Doug DeGroot, editeurs, Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, IRI, pages 157–162. IEEE Systems, Man, and Cybernetics Society, 2004.

[Allier 2012] Simon Allier, Nicolas Anquetil and Andre Hora Stéphane Ducasse. *A Framework to Compare Alert Ranking Algorithms.* In Proceedings of the 19th International Working Conference on Reverse Engineering (WCRE'12), 2012.

[Ambler] Scott W. Ambler. Agile/lean documentation: Strategies for agile software development. http://www.agilemodeling.com/essays/agileDocumentation.htm, last accessed on Jul. 25, 2013.

[Anquetil 1997] Nicolas Anquetil and Timothy C. Lethbridge. *File clustering using naming conventions for legacy systems.* In Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, CASCON'97, pages 184–195. IBM Press, November 1997.

[Anquetil 1998a] Nicolas Anquetil and Timothy C. Lethbridge. *Assessing the relevance of identifier names in a legacy software system.* In Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research, CASCON'98, pages 213–222. IBM Press, 1998.

[Anquetil 1998b] Nicolas Anquetil and Timothy C. Lethbridge. *Extracting Concepts from File Names: a New File Clustering Criterion.* In

Proceedings of the 20th international conference on Software engineering, ICSE'98, pages 84–93, Washington, DC, USA, 1998. IEEE Computer Society.

[Anquetil 1999a] Nicolas Anquetil and Timothy Lethbridge. *Experiments with Clustering as a Software Remodularization Method.* In Proceedings of Working Conference on Reverse Engineering (WCRE'99), pages 235–255, 1999.

[Anquetil 1999b] Nicolas Anquetil and Timothy C. Lethbridge. *Recovering Software Architecture from the Names of Source Files.* Journal of Software Maintenance: Research and Practice, vol. 11, pages 201–21, 1999.

[Anquetil 2000a] Nicolas Anquetil. *A Comparison of Graphs of Concept for Reverse Engineering.* In Proceedings of the 8th International Workshop on Program Comprehension, IWPC '00, pages 231–, Washington, DC, USA, 2000. IEEE Computer Society.

[Anquetil 2000b] Nicolas Anquetil. *Concepts + Relations = 'Abstract Constructs'.* In WCRE'00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00), Los Alamitos, CA, USA, 2000. IEEE Computer Society. Due to an error of the editor, the paper does not appear in the proceedings.

[Anquetil 2001] Nicolas Anquetil. *Characterizing the Informal Knowledge Contained in Systems.* In WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01), pages 166–175, Washington, DC, USA, 2001. IEEE Computer Society.

[Anquetil 2003a] Nicolas Anquetil, Káthia Marçal de Oliveira, Márcio Greyck Batista Dias, Marcelo Ramal and Ricardo de Moura Meneses. *Knowledge for Software Maintenance.* In Proceedings of the Fifteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2003), pages 61–68, 2003.

[Anquetil 2003b] Nicolas Anquetil and Timothy Lethbridge. *Comparative study of clustering algorithms and abstract representations for software remodularization.* IEE Proceedings - Software, vol. 150, no. 3, pages 185–201, 2003.

[Anquetil 2006] Nicolas Anquetil, Káthia M. de Oliveira and Márcio G. B. Dias. *Software Maintenance Ontology.* In Ontologies for Software Engineering and Software Technology, chapitre 5, pages 153–173. Springer-Verlag New York, Inc., 2006.

[Anquetil 2007] Nicolas Anquetil, Káthia M. de Oliveira, Kleiber D. de Sousa and Márcio G. Batista Dias. *Software maintenance seen as a knowledge management issue.* Information Software Technology, vol. 49, no. 5, pages 515–529, 2007.

[Anquetil 2011] Nicolas Anquetil and Jannik Laval. *Legacy Software Restructuring: Analyzing a Concrete Case.* In Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11), pages 279–286, Oldenburg, Germany, 2011.

[Anquetil 2013] Nicolas Anquetil, André Hora, Hani Abdeen, Marco Tulio Valente and Stéphane Ducasse. *Assessing the Quality of Architectural Design Quality Metrics.* Journal of Systems and Software (JSS), 2013. In submission.

[Antoniol 2000] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza and Andrea De Lucia. *Information Retrieval Models for Recovering Traceability Links between Code and Documentation.* In Proceedings of the International Conference on Software Maintenance (ICSM 2000), pages 40–49, 2000.

[Antoniol 2002] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia and Ettore Merlo. *Recovering Traceability Links between Code and Documentation.* IEEE Transactions on Software Engineering, vol. 28, no. 10, pages 970–983, 2002.

[Antoniol 2005] Giuliano Antoniol and Yann-Gaël Guéhéneuc. *Feature Identification: a Novel Approach and a Case Study.* In Proceedings of the IEEE International Conference on Software Maintenance (ICSM'05), pages 357–366, Los Alamitos CA, September 2005. IEEE Computer Society Press.

[Asuncion 2007] Hazeline U. Asuncion, Frédéric François and Richard N. Taylor. *An end-to-end industrial software traceability tool.* In ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 115–124, New York, NY, USA, 2007. ACM.

[Atkinson 1970] A. B. Atkinson. *On the measurement of inequality.* Journal of Economic Theory, vol. 2, no. 3, pages 244–263, 1970.

[Bachmann 2000] Felix Bachmann, Len Bass, Jeromy Carriere, Paul Clements, David Garlan, James Ivers, Robert Nord, Reed Little, Norton L. Compton and Lt Col. *Software Architecture Documentation in Practice: Documenting Architectural Layers*, 2000.

[Basalaj 2006] Wojciech Basalaj and Frank van den Beuken. *Correlation Between Coding Standards Compliance and Software Quality.* Rapport technique, Programming Research, 2006.

[Basili 1992] Victor R. Basili. *Software modeling and measurement: the Goal/Question/Metric paradigm.* Rapport technique, College Park, MD, USA, 1992.

[Berg 2005] Kathrin Berg, Judith Bishop and Dirk Muthig. *Tracing software product line variability: from problem to solution space.* In SAICSIT'05: Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries, pages 182–191, Republic of South Africa, 2005. South African Institute for Computer Scientists and Information Technologists.

[Bhatia 2006] Pradeep Bhatia and Yogesh Singh. *Quantification Criteria for Optimization of Modules in OO Design.* In Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006, volume 2, pages 972–979. CSREA Press, 2006.

[Biggerstaff 1994] Ted J. Biggerstaff, Bharat G. Mitbander and Dallas E. Webster. *Program Understanding and the Concept Assignment Problem.* Communications of the ACM, vol. 37, no. 5, pages 72–82, May 1994.

[Binkley 1998] Aaron B. Binkley and Stephen R. Schach. *Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures.* In ICSE '98: Proceedings of the 20th international conference on Software engineering, pages 452–455, Washington, DC, USA, 1998. IEEE Computer Society.

[Birk 2002] Andreas Birk, Torgeir Dingsoyr and Tor Stalhane. *Postmortem: Never leave a project without it.* Software, IEEE, vol. 19, no. 3, pages 43–45, 2002.

[Black 2009] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker. Pharo by example. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[Boehm 1989] Barry W. Boehm. Software risk management. IEEE Press, 1989. ISBN:0-8186-8906-4.

[Boogerd 2008] C. Boogerd and L. Moonen. *Assessing the Value of Coding Standards: An Empirical Study.* In International Conference on Software Maintenance, pages 277 –286, 2008.

[Boogerd 2009] Cathal Boogerd and Leon Moonen. *Evaluating the Relation Between Coding Standard Violations and Faults Within and Across Software Versions.* In Working Conference on Mining Software Repositories, pages 41–50, 2009.

[Breu 2010] Silvia Breu, Rahul Premraj, Jonathan Sillito and Thomas Zimmermann. *Information needs in bug reports: improving cooperation between developers and users.* In Proceedings of the 2010 ACM conference on Computer supported cooperative work, pages 301–310. ACM, 2010.

[Briand 1994] Lionel C Briand, Victor R Basili, Yong-Mi Kim and Donald R Squier. *A change analysis process to characterize software maintenance projects.* In Software Maintenance, 1994. Proceedings., International Conference on, pages 38–49. IEEE, 1994.

[Briand 1997] Lionel Briand, Prem Devanbu and Walcelio Melo. *An investigation into coupling measures for C++.* In ICSE '97: Proceedings of the 19th international conference on Software engineering, pages 412–421, New York, NY, USA, 1997. ACM.

[Briand 1998] Lionel C. Briand, John W. Daly and Jürgen K. Wüst. *A Unified Framework for Cohesion Measurement in Object-Oriented Systems.* Empirical Software Engineering: An International Journal, vol. 3, no. 1, pages 65–117, 1998.

[Brown 1997] Alan W. Brown and Kurt C. Wallnau. *Enginnering of Component-Based Systems.* In Alan W. Brown, editeur, Component-Based Software Engineering, pages 7–15. IEEE Press, 1997.

[Burd 1996] E. Burd, M. Munro and C. Wezeman. *Extracting Reusable Modules from Legacy Code: Considering the Issues of Module Granularity.* In Working Conference on Reverse Engineering, pages 189–196. IEEE Comp. Soc. Press, Nov 1996.

[Canfora 2010] Gerardo Canfora, Michele Ceccarelli, Massimiliano Di Penta and Luigi Cerulo. *Using multivariate time series and association rules to detect logical change coupling: an empirical study.* In 26th International Conference on Software Maintenance (ICSM), pages 1–10. IEEE Comp. Soc. Press, 2010.

[Carmichael 1995] Ian Carmichael, Vassilios Tzerpos and Rick C. Holt. *Design Maintenance: Unexpected Architectural Interactions.* In International Conference on Software Maintenance (ICSM), pages 134–140. IEEE CS, 1995.

[Carr 1993] M. J. Carr, S. L. Konda, I. Monarch, F. C. Ulrich and C. F. Walker. *Taxonomy-based risk identification*. Rapport technique, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, USA, 1993. CMU/SEI-93-TR-6.

[CCTA 2000] CCTA. Risk handbook. Central Communications and Telecommunications Agency - UK Civil Service, 2000.

[Charette 1997] R. N. Charette, K. M. Adams and M. B. White. *Managing Risk in Software Maintenance*. IEEE Software, vol. 14, no. 3, pages 43–50, may/jun. 1997.

[Chen 2000] Kunrong Chen and Václav Rajlich. *Case Study of Feature Location Using Dependence Graph*. In Proceedings IEEE International Conference on Software Maintenance (ICSM), pages 241–249. IEEE Computer Society Press, 2000.

[Cimitile 1997] A. Cimitile, A.D. Lucia, , G.D. Lucca and A.R. Fasolino. *Identifying Objects in Legacy Systems*. In 5^th International Workshop on Program Comprehension, IWPC'97, pages 138–147. IEEE Comp. Soc. Press, 1997.

[Cinnéide 2012] M.Ó. Cinnéide, L. Tratt, M. Harman, S. Counsell and I.H. Moghadam. *Experimental Assessment of Software Metrics Using Automated Refactoring*. In Proc. Empirical Software Engineering and Management (ESEM), September 2012. to appear.

[Cioch 1996] Frank A Cioch, Michael Palazzolo and Scott Lohrer. *A documentation suite for maintenance programmers*. In Software Maintenance 1996, Proceedings., International Conference on, pages 286–295. IEEE, 1996.

[Clayton 1998] R. Clayton, S. Rugaber and L. Wills. *Incremental Migration Strategies: Data Flow Analysis for Wrapping*. In Proceedings of WCRE '98, pages 69–79. IEEE Computer Society, 1998. ISBN: 0-8186-89-67-6.

[Cleland-Huang 2003] Jane Cleland-Huang, Carl K. Chang and Mark Christensen. *Event-Based Traceability for Managing Evolutionary Change*. IEEE Transactions on Software Engineering, vol. 29, no. 9, pages 796–810, September 2003.

[Collins 1996] Jason A. Collins, Jim E. Greer and Sherman X. Huang. *Adaptive Assessment using Granularity Hierarchies and Bayesian Nets*. In Proceedings of the Third International Conference on Intelligent Tutoring Systems, pages 569–577, 1996.

[Counsell 2005] Steve Counsell, Stephen Swift and Allan Tucker. *Object-oriented cohesion as a surrogate of software comprehension: an empirical study.* In Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation, pages 161–172, 2005.

[Couto 2012] Cesar Couto, Christofer Silva, Marco T. Valente, Roberto Bigonha and Nicolas Anquetil. *Uncovering Causal Relationships between Bugs and Software Metrics.* In Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12), page *to appear*, 2012.

[D'Ambros 2010] Marco D'Ambros, Michele Lanza and Romain Robbes. *An extensive comparison of bug prediction approaches.* In 7th Working Conference on Mining Software Repositories (MSR), pages 31–41, 2010.

[D'Ambros 2012] Marco D'Ambros, Michele Lanza and Romain Robbes. *Evaluating defect prediction approaches: a benchmark and an extensive comparison.* Journal of Empirical Software Engineering, vol. 17, no. 4-5, pages 531–77, 2012.

[de Sousa 2004] Kleiber D. de Sousa, Nicolas Anquetil and Káthia Marçal de Oliveira. *Learning Software Maintenance Organizations.* In Advances in Learning Software Organizations, 6th International Workshop, LSO 2004, volume 3096 of *Lecture Notes in Computer Science*, pages 67–77. Springer, 2004.

[de Souza 2005] Sergio Cozzetti B. de Souza, Nicolas Anquetil and Káthia M. de Oliveira. *A study of the documentation essential to software maintenance.* In Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information, SIGDOC '05, pages 68–75, New York, NY, USA, 2005. ACM.

[Deridder 2002] Dirk Deridder. *A concept-oriented approach to support software maintenance and reuse activities.* In Proceedings of the 5th Joint Conference on Knowledge Based Software Engineering, 2002.

[Dias 2003a] Márcio G. B. Dias, Nicolas Anquetil and Káthia M. de Oliveira. *Organizing the Knowledge Used in Software Maintenance.* Journal of Universal Computer Science, vol. 9, no. 7, pages 641–658, 2003.

[Dias 2003b] Márcio Greyck Batista Dias, Nicolas Anquetil and Káthia Marçal de Oliveira. *Organizing the Knowledge Used in Software Maintance.* In Ulrich Reimer, Andreas Abecker, Steffen Staab and Gerd Stumme, editeurs, WM 2003: Professionelles

Wissesmanagement - Erfahrungen und Visionen, Beiträge der 2. Konferenz Professionelles Wissensmanagement, volume 28, pages 65–72, 2003.

[Dit 2011] Bogdan Dit, Latifa Guerrouj, Denys Poshyvanyk and Giuliano Antoniol. *Can better identifier splitting techniques help feature location?* In Program Comprehension (ICPC), 2011 IEEE 19th International Conference on, pages 11–20. IEEE, 2011.

[Dömges 1998] Ralf Dömges and Klaus Pohl. *Adapting traceability environments to project-specific needs.* Commun. ACM, vol. 41, no. 12, pages 54–62, 1998.

[Ducasse 2009] Stéphane Ducasse and Damien Pollet. *Software Architecture Reconstruction: A Process-Oriented Taxonomy.* IEEE Transactions on Software Engineering, vol. 35, no. 4, pages 573–591, July 2009.

[Ebad 2011] Shouki A. Ebad and Moataz Ahmed. *An Evaluation Framework for Package-Level Cohesion Metrics.* International Proceedings of Computer Science and Information Technology, vol. 13, pages 239–43, 2011.

[Egyed 2002] Alexander Egyed and Paul Grünbacher. *Automating Requirements Traceability: Beyond the Record & Replay Paradigm.* In ASE, pages 163–171. IEEE Computer Society, 2002.

[Eisenbarth 2001] Thomas Eisenbarth, Rainer Koschke and Daniel Simon. *Aiding Program Comprehension by Static and Dynamic Feature Analysis.* In Proceedings of ICSM '01 (International Conference on Software Maintenance). IEEE Computer Society Press, 2001.

[Engler 2000] Dawson Engler, Benjamin Chelf, Andy Chou and Seth Hallem. *Checking system Rules Using System-specific, Programmer-Written Compiler Extensions.* In Symposium on Operating System Design & Implementation, pages 1–16, 2000.

[Falleri 2008] Jean-Rémy Falleri, Marianne Huchard and Clémentine Nebut. *A Generic Approach for Class Model Normalization.* In ASE, pages 431–434, 2008.

[Falleri 2011] Jean Rémi Falleri, Simon Denier, Jannik Laval, Philipe Vismara and Stéphane Ducasse. *Efficient Retrieval and Ranking of Undesired Package Cycles in Large Software Systems.* In Proceedings of the 49th International Conference on Objects, Models, Components, Patterns (TOOLS'11), Zurich, Switzerland, June 2011.

[Feild 2006] Henry Feild, David Binkley and Dawn Lawrie. *An empirical comparison of techniques for extracting concept abbreviations from identifiers.* In Proceedings of IASTED International Conference on Software Engineering and Applications (SEA 2006). Citeseer, 2006.

[Flanagan 2002] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe and Raymie Stata. *Extended Static Checking for Java.* In Conference on Programming Language Design and Implementation, pages 234–245, 2002.

[Forward 2002] Andrew Forward and Timothy C Lethbridge. *The relevance of software documentation, tools and technologies: a survey.* In Proceedings of the 2002 ACM symposium on Document engineering, pages 26–33. ACM, 2002.

[Garcia 2013] Joshua Garcia, Igor Ivkovic and Nenad Medvidovic. *A comparative analysis of software architecture recovery techniques.* In Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pages 486–496. IEEE, 2013.

[Gini 1921] C. Gini. *Measurement of Inequality of Incomes.* The Economic Journal, vol. 31, pages 124–126, 1921.

[Godin 1993] Robert Godin and Hafedh Mili. *Building and Maintaining Analysis-Level Class Hierarchies using Galois Lattices.* In Proceedings OOPSLA '93 (8th Conference on Object-Oriented Programming Systems, Languages, and Applications), volume 28, pages 394–410, October 1993.

[Goeminne 2011] M. Goeminne and T. Mens. *Evidence for the Pareto principle in Open Source Software Activity.* In Proc. Int'l Workshop SQM 2011. CEUR-WS workshop proceedings, 2011.

[Granger 1969] Clive Granger. *Investigating causal relations by econometric models and cross-spectral methods.* Econometrica, vol. 37, no. 3, pages 424–38, 1969.

[Granger 1981] Clive Granger. *Some properties of time series data and their use in econometric model specification.* Journal of Econometrics, vol. 16, no. 6, pages 121–30, 1981.

[Grubb 2003] Penny Grubb and Armstrong A. Takang. Software maintenance concepts and practices. World Scientific, second edition édition, 2003.

[Hall 2005] Gregory A. Hall, Wenyou Tao and John C. Munson. *Measurement and Validation of Module Coupling Attributes.* Software Quality Control, vol. 13, no. 3, pages 281–296, 2005.

[Heckman 2011] Sarah Heckman and Laurie Williams. *A systematic literature review of actionable alert identification techniques for automated static code analysis.* Inf. Softw. Technol., vol. 53, pages 363–387, apr 2011.

[Hoover 1936] E. M. Hoover. *The measurement of industrial localization.* The Review of Economic Statistics, vol. 18, no. 4, pages 162–171, 1936.

[Hora 2012] Andre Hora, Nicolas Anquetil, Stéphane Ducasse and Simon Allier. *Domain Specific Warnings: Are They Any Better?* In Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12), 2012.

[Houston 2001] D. Houston, G.T. Mackulak and J. Collofello. *Stochastic Simulation of Risk Factor Potential Effects for Software Development Risk Management.* Journal of Systems and Software, vol. 59, no. 3, pages 247–57, 2001.

[ISO 2001] ISO. *International Standard – ISO/IEC 9126-1:2001 – Software engineering – Product quality.* Rapport technique, ISO, 2001.

[ISO 2006] ISO. *International Standard – ISO/IEC 14764 IEEE Std 14764-2006.* Rapport technique, ISO, 2006.

[ISO/IEC 1995] ISO/IEC. *ISO/IEC 12207 Information technology – Software life cycle pro- cesses*, 1995.

[Jalote 1999] Pankaj Jalote. Cmm in practice: Processes for executing software projects at infosys, chapitre 8, pages 159–74. Addison-Wesley, 1999. ISBN: 0201616262.

[Jalote 2002] Pankaj Jalote. Software project management in practice, chapitre 6, pages 93–108. Addison-Wesley, 2002. ISBN: 0201737213.

[Jirapanthong 2005] Waraporn Jirapanthong and Andrea Zisman. *Supporting Product Line Development through Traceability.* In Proc. of the 12th Asia-Pacific Software Engineering Conference (APSEC), pages 506–514, Taipei (Taiwan), 2005.

[Johnson 1978] S.C. Johnson. *Lint, a C Program Checker.* In UNIX programmer's manual, pages 78–1273. AT&T Bell Laboratories, 1978.

[Jørgensen 2002] Magne Jørgensen and Dag I. K. Sjøberg. *Impact of experience on maintenance skills.* Journal of Software Maintenance, vol. 14, no. 2, pages 123–146, 2002.

[Kajko-Mattsson 2001] Mira Kajko-Mattsson. *The state of documentation practice within corrective maintenance.* In Software Maintenance, 2001. Proceedings. IEEE International Conference on, pages 354–363. IEEE, 2001.

[Keil 1998] Mark Keil, Paul E. Cule, Kalle Lyytinen and Roy C. Schmidt. *A framework for identifying software project risks.* Communications of the ACM, vol. 41, no. 11, pages 76–83, nov. 1998.

[Kellens 2007] Andy Kellens, Kim Mens and Paolo Tonella. *A Survey of Automated Code-Level Aspect Mining Techniques.* Transactions on Aspect-Oriented Software Development, vol. 4, no. 4640, pages 143–162, 2007.

[Kim 2006] Sunghun Kim, Thomas Zimmermann, Kai Pan and E. James Jr. Whitehead. *Automatic Identification of Bug-Introducing Changes.* In International Conference on Automated Software Engineering, pages 81–90, 2006.

[Kim 2007] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr. and Andreas Zeller. *Predicting Faults from Cached History.* In ICSE '07: Proceedings of the 29th international conference on Software Engineering, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[Kitchenham 2002] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam and Jarrett Rosenberg. *Preliminary guidelines for empirical research in software engineering.* IEEE Trans. Softw. Eng., vol. 22, no. 8, pages 721–734, 2002.

[Ko 2007] Andrew J. Ko, Robert DeLine and Gina Venolia. *Information needs in collocated software development teams.* In Proceedings of the 29th International Conference on Software Engineering, pages 344–353. IEEE Computer Society, 2007.

[Kolm 1976] S.-C. Kolm. *Unequal inequalities I.* Journal of Economic Theory, vol. 12, no. 3, pages 416–442, 1976.

[Kremenek 2003] Ted Kremenek and Dawson Engler. *Z-ranking: using statistical analysis to counter the impact of static analysis approximations.* In Proceedings of the 10th international conference on Static analysis, SAS'03, pages 295–315, Berlin, Heidelberg, 2003. Springer-Verlag.

[Kremenek 2004] Ted Kremenek, Ken Ashcraft, Junfeng Yang and Dawson Engler. *Correlation exploitation in error ranking.* In Proceedings of

the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, SIGSOFT '04/FSE-12, pages 83–93, New York, NY, USA, 2004. ACM.

[Kwak 2004] Y.H. Kwak and J. Stoddard. *Project risk management: lessons learned from software development environment.* Technovation, vol. 24, no. 11, pages 915–920, nov. 2004.

[Lakhotia 1997] A. Lakhotia. *A unified framework for expressing software subsystem classification techniques.* Journal of Systems and Software, pages 211–231, March 1997.

[Laval 2010] Jannik Laval, Nicolas Anquetil and Stéphane Ducasse. *OZONE: Package Layered Structure Identification in presence of Cycles.* In Proceedings of the 9th edition of the Workshop BElgian-NEtherlands software eVOLution seminar (BENEVOL'10), Lille, France, 2010.

[Laval 2012] Jannik Laval, Nicolas Anquetil, Usman Bhatti and Stéphane Ducasse. *Package Layers Identification in the presence of Cyclic Dependencies.* Science of Computer Programming, 2012. Special issue on Software Evolution, Adaptability and Maintenance.

[Lehman 1980] Manny Lehman. *Programs, Life Cycles, and Laws of Software Evolution.* Proceedings of the IEEE, vol. 68, no. 9, pages 1060–1076, September 1980.

[Lehman 1998] Manny Lehman, Dewayne Perry and Juan Ramil. *Implications of Evolution Metrics on Software Maintenance.* In Proceedings IEEE International Conference on Software Maintenance (ICSM'98), pages 208–217, Los Alamitos CA, 1998. IEEE Computer Society Press.

[Lethbridge 2002] Timothy C. Lethbridge and Nicolas Anquetil. *Approaches to clustering for program comprehension and remodularization.* In Advances in software engineering, pages 137–157. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[Li 1993] W. Li and S. Henry. *Object Oriented Metrics that predict maintainability.* Journal of System Software, vol. 23, no. 2, pages 111–122, 1993.

[Li 2006] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou and Chengxiang Zhai. *Have Things Changed Now? An Empirical Study of Bug Characteristics in Modern Open Source Software.* In Proceedings of the 1st workshop on Architectural and system support for improving software dependability, pages 25–33, 2006.

[Machado 2002] C.A. Machado. A-risk: Um método para identificar e quantificar risco de prazo em desenvolvimento de software. Master's thesis, PPGIA, Universidade Católica do Paraná, 2002.

[Mancoridis 1996] Spiros Mancoridis and Richard C. Holt. *Recovering the structure of Software Systens Using Tube Graph Interconnection Clustering.* In International Conference on Software Maintenance, ICSM'96, pages 23–32. IEEE Comp. Soc. Press, Nov 1996.

[Mancoridis 1999] Spiros Mancoridis, Brian S. Mitchell, Y. Chen and E. R. Gansner. *Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures.* In Proceedings of ICSM '99 (International Conference on Software Maintenance), Oxford, England, 1999. IEEE Computer Society Press.

[Mayr 1995] Ernst W. Mayr and Claude Puech, editeurs. Proceedings STACS'95, volume 900 of *LNCS*. Springer-Verlag, Munich, Germany, March 1995.

[McGarry 2001] J. McGarry. *When it comes to measuring software, every project is unique.* IEEE Software, vol. 18, no. 5, page 19, 2001.

[McManus 2004] John McManus. Risk management in software development projects. Elsevier Butterworth-Heinnemann, 2004. ISBN: 0-7506-5867-3.

[Misra 2003] S.C. Misra and V.C. Bhavsar. *Measures of software system difficulty.* Software Quality Professional, vol. 5, no. 4, pages 33–41, 2003.

[Mockus 2000] Audris Mockus and Lawrence G. Votta. *Identifying Reasons for Software Changes using Historic Databases.* In International Conference on Software Maintenance, pages 120–130, 2000.

[Mohan 2002] Kannan Mohan and Balasubramaniam Ramesh. *Managing Variability with Traceability in Product and Service Families.* In Proceedings of the 35th Hawaii International Conference on System Sciences, pages 1309–1317, 2002.

[Mohan 2007] Kannan Mohan and Balasubramaniam Ramesh. *Tracing variations in software product families.* Communications of ACM, vol. 50, no. 12, pages 68–73, 2007.

[Moon 2006] Mikyeong Moon and Heung S. Chae. *A Metamodel Approach to Architecture Variability in a Product Line.* In Springer-Verlag, editeur, Proceedings of the Reuse of Off-the-Shelf Components, 9th International Conference on Software Reuse, volume 4039 of *LNCS*, pages 115–126, 2006.

[Mordal-Manet 2009] Karine Mordal-Manet, Françoise Balmas, Simon Denier, Stéphane Ducasse, Harald Wertz, Jannik Laval, Fabrice Bellingard and Philippe Vaillergues. *The Squale Model – A Practice-based Industrial Quality Model.* In Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'09), pages 94–103, Edmonton, Canada, 2009.

[Mordal-Manet 2011] Karine Mordal-Manet, Jannik Laval, Stéphane Ducasse, Nicolas Anquetil, Françoise Balmas, Fabrice Bellingard, Laurent Bouhier, Philippe Vaillergues and Thomas J. McCabe. *An empirical model for continuous and weighted metric aggregation.* In Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11), pages 141–150, Oldenburg, Germany, 2011.

[Mordal 2012] Karine Mordal, Nicolas Anquetil, Jannik Laval, Alexander Serebrenik, Bogdan Vasilescu and Stéphane Ducasse. *Practical Software Quality Metrics Aggregation.* Journal of Software Maintenance and Evolution: Research and Practice, 2012.

[Mukhopadhyay 2007] Nitai D. Mukhopadhyay and Snigdhansu Chatterjee. *Causality and pathway search in microarray time series experiment.* Bioinformatics, vol. 23, no. 4, pages 442–49, 2007.

[Müller 1993] Hausi A. Müller. *Software Engineering Education should concentrate on Software Evolution.* In Proceedings of National Workshop on Software Engineering Education, pages 102–104, May 1993. University of Victoria (Canada).

[Murphy 1996] Richard L. Murphy, Christopher J. Alberts, Ray C. Williams, Ronald P. Higuera, Audrey J. Dorofee and Julie A. Walker. Continuous risk management guidebook. Software Engineering Institute, Carnegie Mellon University, 1996.

[Murphy 2001] Gail C. Murphy, Robert J. Walker, Elisa L. A. Baniassad, Martin P. Robillard, Albert Lai and Mik A. Kersten. *Does aspect-oriented programming work?* Commun. ACM, vol. 44, no. 10, pages 75–77, 2001.

[Nagappan 2005] Nachiappan Nagappan and Thomas Ball. *Static Analysis Tools as Early Indicators of Pre-release Defect Density.* In International Conference on Software Engineering, pages 580–586, 2005.

[Newcomb 1995] Philipp Newcomb and Paul Martens. *Reengineering procedural into data flow program.* In Proceedings of WCRE (Working Conference on Reverse Engineering), pages 32–39. IEEE CS, 1995.

[Nierstrasz 2005] Oscar Nierstrasz, Stéphane Ducasse and Tudor Gîrba. *The Story of Moose: an Agile Reengineering Environment.* In Michel Wermelinger and Harald Gall, editeurs, Proceedings of the European Software Engineering Conference, ESEC/FSE'05, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[Ostrand 2004] Thomas J. Ostrand, Elaine J. Weyuker and Robert M. Bell. *Where the Bugs Are.* In ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 86–96, 2004.

[Pearse 1995] T. Pearse and P. Oman. *Maintainability measurements on industrial source code maintenance activities.* In International Conference on Software Maintenance, 1995. Proceedings.,, pages 295–303. IEEE Comp. Soc. Press, 1995.

[Pfleeger 2002] Shari Lawrence Pfleeger and Joanne M. Atlee. Software engineering - theory and practice. Pearson Education, 2nd édition, 2002.

[Pigoski 1997] T. Pigoski. Practical software maintenance. best practices managing your software investment. John Wiley and Sons, 1997.

[Pohl 2005] Klaus Pohl, Günter Böckle and Frank van der Linden. Software Product Line Engineering - Foundations, Principles, and Techniques. Springer Verlag, Heidelberg, 2005.

[Polo 2001] Macario Polo, Mario Piattini and Francisco Ruiz. *Using Code Metrics to Predict Maintenance of Legacy Programs: A Case Study.* In ICSM, pages 202–208, 2001.

[Poshyvanyk 2007] Denys Poshyvanyk and Andrian Marcus. *Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code.* In ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension, pages 37–48, Washington, DC, USA, 2007. IEEE Computer Society.

[Pressman 1994] Roger S. Pressman. Software engineering: A practitioner's approach. McGraw-Hill, 1994.

[Rajlich 2000] Vaclav Rajlich and Keith Bennett. *A Staged Model for the Software Life Cycle.* IEEE Computer, vol. 33, no. 7, pages 66–71, 2000.

[Ramal 2002] M. Fenoll Ramal, Ricardo de Moura Meneses and Nicolas Anquetil. *A Disturbing Result on the Knowledge Used during Software Maintenance.* In 9th Working Conference on Reverse Engineering (WCRE 2002), pages 277–. IEEE Computer Society, 2002.

[Ramesh 1998] Balasubramaniam Ramesh. *Factors influencing requirements traceability practice.* Communications of the ACM, vol. 41, no. 12, pages 37–44, 1998.

[Ramesh 2001] Balasubramaniam Ramesh and Matthias Jarke. *Toward Reference Models for Requirements Traceability.* IEEE Transactions on Software Engineering, vol. 27, no. 1, pages 58–93, 2001.

[Ramos 2004] Cristiane S. Ramos, Káthia Marçal de Oliveira and Nicolas Anquetil. *Legacy Software Evaluation Model for Outsourced Maintainer.* In 8th European Conference on Software Maintenance and Reengineering (CSMR 2004), pages 48–57. IEEE Computer Society, 2004.

[Renggli 2010] Lukas Renggli, Tudor Gîrba and Oscar Nierstrasz. *Embedding Languages Without Breaking Tools.* In Theo D'Hondt, editeur, Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10), volume 6183 of *LNCS*, pages 380–404. Springer-Verlag, 2010.

[Rising 1999] Linda Rising. *Patterns in postmortems.* In Computer Software and Applications Conference, 1999. COMPSAC'99. Proceedings. The Twenty-Third Annual International, pages 314–315. IEEE, 1999.

[Roberts 1997] Don Roberts, John Brant and Ralph E. Johnson. *A Refactoring Tool for Smalltalk.* Theory and Practice of Object Systems (TAPOS), vol. 3, no. 4, pages 253–263, 1997.

[Rosik 2011] Jacek Rosik, Andrew Le Gear, Jim Buckley, Muhammad Ali Babar and Dave Connolly. *Assessing architectural drift in commercial software development: a case study.* Software: Practice and Experience, vol. 41, no. 1, pages 63–86, 2011.

[Ruíz 2004] Francisco Ruíz, Aurora Vizcaíno, Mario Piattini and Félix García. *An ontology for the management of software maintenance projects.* International Journal of Software Engineering and Knowledge Engineering, vol. 14, no. 03, pages 323–349, 2004.

[Sahraoui 1999] H. A. Sahraoui, H. Lounis, W. Melo and H. Mili. *A Concept Formation Based Approach to Object Identification in Procedural Code.* Automated Software Engineering Journal, vol. 6, no. 4, pages 387–410, 1999.

[Sangal 2005] Neeraj Sangal, Ev Jordan, Vineet Sinha and Daniel Jackson. *Using Dependency Models to Manage Complex Software Architecture.* In Proceedings of OOPSLA'05, pages 167–176, 2005.

[Sarkar 2007] Santonu Sarkar, Girish Maskeri Rama and Avinash C. Kak. *API-Based and Information-Theoretic Metrics for Measuring the Quality of Software Modularization.* IEEE Trans. Softw. Eng., vol. 33, no. 1, pages 14–32, 2007.

[Schneidewind 2002] Norman F. Schneidewind. Advances in software maintenance management: Technologies and solutions, chapitre VII - Requirements Risk and Maintainability. IDEA Group Publishing, 2002. ISBN: 1591400473.

[Seacord 2003] R.C. Seacord, D.A. PLAKOSH and G.A.A. LEWIS. Modernizing legacy systems: Software technologies, engineering processes, and business practices. The SEI Series in Software Engineering. ADDISON WESLEY Publishing Company Incorporated, 2003.

[Serebrenik 2010] A. Serebrenik and M. G. J. van den Brand. *Theil index for aggregation of software metrics values.* In Int. Conf. on Software Maintenance, pages 1–9. IEEE, 2010.

[Sillito 2008] J. Sillito, G.C. Murphy and K. De Volder. *Asking and Answering Questions during a Programming Change Task.* IEEE Transactions on Software Engineering, vol. 34, no. 4, pages 434–451, jul 2008.

[Śliwerski 2005] Jacek Śliwerski, Thomas Zimmermann and Andreas Zeller. *When Do changes Induce Fixes?* In Proceedings of International Workshop on Mining Software Repositories — MSR'05, Saint Lous, Missouri, USA, 2005. ACM Press.

[Sneed 1996] Harry M. Sneed. *Object-Oriented COBOL Recycling.* In Working Conference on Reverse Engineering, pages 169–178. IEEE Comp. Soc. Press, Nov 1996.

[Sousa 1998] Maria João C Sousa and Helena Mendes Moreira. *A survey on the software maintenance process.* In Software Maintenance, 1998. Proceedings. International Conference on, pages 265–274. IEEE, 1998.

[Souza 2006] Sergio Cozzetti B. de Souza, Nicolas Anquetil and Káthia M. de Oliveira. *Which documentation for software maintenance?* Journal of the Brazilian Computer Society, pages 31–44, 2006.

[Stålhane 2003] Tor Stålhane, Torgeir Dingsøyr, Geir Kjetil Hanssen and Nils Brede Moe. *Post mortem–an assessment of two approaches.* In Empirical Methods and Studies in Software Engineering, pages 129–141. Springer, 2003.

[Stevens 1974] W. P. Stevens, G. J. Myers and L. L. Constantine. *Structured Design.* IBM Systems Journal, vol. 13, no. 2, pages 115–139, 1974.

[Taube-Schock 2011] Craig Taube-Schock, Robert J. Walker and Ian H Witten. *Can we avoid high coupling?* In Proceedings of ECOOP 2011, 2011.

[Teles 2004] Vinícius Manhães Teles. *Extreme programming.* Novatec Editora Ltda, Rua cons. Moreira de Barros, vol. 1084, pages 02018–012, 2004.

[Theil 1967] H. Theil. Economics and Information Theory. North-Holland, 1967.

[Thomas 2001] David Thomas and Andrew Hunt. Programming ruby. Addison Wesley, 2001.

[Tilley 1992] Scott R. Tilley, Hausi A. Müller and Mehmet A. Orgun. *Documenting software systems with views.* In Proceedings of the 10th annual international conference on Systems documentation, pages 211–219. ACM, 1992.

[Tilley 1993] Scott R. Tilley and Hausi A. Müller. *Using Virtual Subsystems in Project Management.* In Proceedings of CASE '93 6th International Workshop on Computer-Aided Software Engineering. IEEE Computer Society, July 1993.

[Torres 2006] Alexandre H. Torres, Nicolas Anquetil and Káthia M. de Oliveira. *Pro-active dissemination of knowledge with learning histories.* In Proceedings of the Eighth International Workshop on Learning Software Organizations, pages 19–27, 2006.

[Tzerpo 1997] Vassilios Tzerpo and R. C. Holt. *The Orphan Adoption Problem in Architecture Maintenance.* Reverse Engineering, Working Conference on, vol. 0, page 76, 1997.

[U.Bhatti 2012] Muhammad U.Bhatti, Nicolas Anquetil, Marianne Huchard and Stéphane Ducasse. *A Catalog of Patterns for Concept Lattice Interpretation in Software Reengineering.* In Proceedings of the 24th International Conference on Software Engineering & Knowledge Engineering (SEKE 2012), pages 118–24, 2012.

[Uschold 1996] Mike Uschold and Michael Gruninger. *Ontologies: principles, methods and applications.* The Knowledge Engineering Review, vol. 11, pages 93–136, 1996.

[Vasa 2009] Rajesh Vasa, Markus Lumpe, Philip Branch and Oscar Nierstrasz. *Comparative Analysis of Evolving Software Systems Using the Gini Coefficient.* In Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009), pages 179–188, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[Vasilescu 2011] B. Vasilescu, A. Serebrenik and M. G. J. van den Brand. *You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics.* In Int. Conf. on Software Maintenance. IEEE, 2011.

[von Mayrhauser 1994] A. von Mayrhauser and A.M. Vans. *Dynamic code cognition behaviors for large scale code.* In Program Comprehension, 1994. Proceedings., IEEE Third Workshop on, pages 74–81, 1994.

[von Mayrhauser 1995] Anneliese von Mayrhauser and A. Marie Vans. *Program Comprehension During Software Maintenance and Evolution.* IEEE Computer, vol. 28, no. 8, pages 44–55, 1995.

[von Mayrhauser 1996] A. von Mayrhauser and A.M. Vans. *Identification of Dynamic Comprehension Processes During Large Scale Maintenance.* IEEE Transactions on Software Engineering, vol. 22, no. 6, pages 424–437, June 1996.

[Webster 2005] Kenia P. Batista Webster, Kathia M. de Oliveira and Nicolas Anquetil. *A Risk Taxonomy Proposal for Software Maintenance.* In Proceedings of the 21st IEEE International Conference on Software Maintenance, pages 453–461, Washington, DC, USA, 2005. IEEE Computer Society.

[Wiggerts 1997] Theo Wiggerts. *Using Clustering Algorithms in Legacy Systems Remodularization.* In Ira Baxter, Alex Quilici and Chris Verhoef, editeurs, Proceedings of WCRE '97 (4th Working Conference on Reverse Engineering), pages 33–43. IEEE Computer Society Press, 1997.

[Wille 1992] Rudolf Wille. *Concept lattices and conceptual knowledge systems.* Computers & mathematics with applications, vol. 23, no. 6, pages 493–515, 1992.

[Yourdon 2001] Edward Yourdon. *Minipostmortems.* COMPUTERWORLD, march, vol. 19, 2001.

[Zheng 2006] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl and Mladen A. Vouk. *On the Value of Static Analysis for Fault Detection in Software.* Transactions on Software Engineering, vol. 32, no. 4, pages 240–253, apr 2006.