Department of Computer Science
Series of Publications A
Report A-2013-13

Self-Organizing Software Architectures

Pietu Pohjalainen

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium XV, University Main Building, on 13th December 2013, at noon.

University of Helsinki
Finland

Supervisors
Jukka Paakki, University of Helsinki, Finland
Juha Taina, University of Helsinki, Finland

Pre-examiners
Görel Hedin, Lund University, Sweden
Jyrki Nummenmaa, University of Tampere, Finland

Opponent
Kai Koskimies, Tampere University of Technology, Finland

Custos
Jukka Paakki, University of Helsinki, Finland

Contact information

Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
FI-00014 University of Helsinki
Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/
Telephone: +358 9 1911, telefax: +358 9 191 51120

ISSN 1238-8645
ISBN 978-952-10-9424-8 (paperback)
ISBN 978-952-10-9425-5 (PDF)
Computing Reviews (1998) Classification: D.2.11, D.1.2, D1.5, D.2.2, D.2.3, D.2.7, D.2.13
Helsinki 2013
Unigrafia

Self-Organizing Software Architectures

Pietu Pohjalainen

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
pietu.pohjalainen@iki.fi

PhD Thesis, Series of Publications A, Report A-2013-13
Helsinki, December 2013, 114 + 71 pages
ISSN 1238-8645
ISBN 978-952-10-9424-8 (paperback)
ISBN 978-952-10-9425-5 (PDF)

Abstract

Looking at engineering productivity is a source for improving the state of software engineering. We present two approaches to improve productivity: bottom-up modeling and self-configuring software components. Productivity, as measured in the ability to produce correctly working software features using limited resources is improved by performing less wasteful activities and by concentrating on the required activities to build sustainable software development organizations.

Bottom-up modeling is a way to combine improved productivity with agile software engineering. Instead of focusing on tools and up-front planning, the models used emerge, as the requirements to the product are unveiled during a project. The idea is to build the modeling formalisms strong enough to be employed in code generation and as runtime models. This brings the benefits of model-driven engineering to agile projects, where the benefits have been rare.

Self-configuring components are a development of bottom-up modeling. The notion of a source model is extended to incorporate the software entities themselves. Using computational reflection and introspection, dependent components of the software can be automatically updated to reflect changes in the dependence. This improves maintainability, thus making software changes faster.

The thesis contains a number of case studies explaining the ways of applying

the presented techniques. In addition to constructing the case studies, an empirical validation with test subjects is presented to show the usefulness of the techniques.

Computing Reviews (1998) Categories and Subject Descriptors:
D.2.11 Software Architectures
D.1.2 Automatic Programming
D.1.5 Object-oriented Programming
D.2.2 Design Tools and Techniques
D.2.3 Coding Tools and Techniques
D.2.7 Distribution, Maintenance, and Enhancement
D.2.13 Reusable Software

General Terms:
Design, Economics, Experimentation, Reliability

Additional Key Words and Phrases:
engineering productivity, software maintenance, self-configuring components, bottom-up modeling, generative programming, domain-specific languages

Acknowledgements

This work would not have been possible without the support from many people at the university, industry and in private life. First I would like to thank Department of Computer Science, University of Helsinki for providing the environment for completing this thesis and Graduate School on Software Systems and Engineering for providing financial support for attending to a number of scientific workshops and conferences. On personal level, I would like to thank late professor Inkeri Verkamo for accepting me as a doctoral student regardless the unrealistic study plans. Dr. Juha Taina helped a lot when translating my fuzzy ideas into formal, scientific text. Professor Jukka Paakki aided this text to gain its final form. I would also like to express my gratitude to Ph.Lic Juha Vihavainen for teaching me the internals of compilers and MA Marina Kurtén for helping with the English language, often within tight deadlines.

Much of this work was done in context of industrial collaboration. For this reason, I would like to thank the companies involved, Digital Chocolate and Comptel Oyj. You have helped me to gain confidence that the things I am working on have significance also outside the academia. I would also like to thank people at National Institute for Health and Welfare for providing room for finishing this thesis. Additionally, I would like to thank Apple for making easy-to-use backup systems installed on their computers. Only three Apple-branded computers were destructed during preparation of this thesis.

Finally, I would like to thank my three girls, Eeva, Armi and Asta for giving me alternative ways of spending my time.

Helsinki, November 11th 2013
Pietu Pohjalainen

Contents

Original publications

This thesis consists of the following peer-reviewed publications and an introduction reviewing the area of study. In the thesis, the included publications are referred to by their Roman numerals.

I P. Pohjalainen: Restructuring optimizations for object-oriented mobile applications. *Proceedings of Workshop on New Horizons in Compiler Analysis and Optimizations*, Bangalore, India (2004).

II P. Pohjalainen: Object-Oriented Language Processing, *Proceedings of 7th Joint Modular Languages Conference*, Oxford, UK (2006).

III P. Pohjalainen: Bottom-up modeling for a software product line: An experience report on agile modeling of governmental mobile networks. *Proceedings of 15th Software Product Lines Conference*, Münich, Germany (2011).

IV P. Pohjalainen: Self-configuring user interface components. *Proceedings of 1st workshop on Semantic Models for Adaptive Interactive Systems*, Hong Kong, China (2010).

V P. Pohjalainen, J. Taina: Self-configuring object-to-relational mapping queries, *Proceedings of 6th International Symposium of Principles and Practice of Programming in Java*, Modena, Italy (2008).

VI P. Pohjalainen: Transparent persistency seems problematic for software maintenance - A randomized, controlled experiment. *Proceedings of 8th International Conference on Evaluation of Novel Approaches to Software Engineering*, Angers, France (2013).

In paper (V), J. Taina supported in formalization of the query inference system's description. The actual implementation was done by the author.

Chapter 1

Introduction

Improving productivity is an everlasting quest in software engineering. Being able to produce better software faster means opportunities to gain more profits for businesses, and to be able to serve customers better even when no commercial interest is in place, such as in governmental organizations. In the past, this target has been approached in many ways, such as via development of different software process models, improving managerial productivity, or by developments of e.g. better programming languages and development environments, targeting programmer productivity.

Developments in programming languages have not improved productivity much. The current mainstream of programming languages, such as Java [GJSB05] and C# [Mic07], have not been able to improve productivity in order of magnitudes over older languages [BH12, McC04, p. 62]. According to research, the way of using programming tools contributes more to productivity than the tool itself. Actually, it seems that software architecture is the driving force to alter productivity: well-suiting architecture is a boon for productivity while an ill-suited architecture results in a major decline in productivity due to the required rework effort [TLB+09]. Empirically, one study found the average team size to be the most important factor when determining variance in productivity and the architecture used to be the second in importance - while the programming language explains only one third of the variance compared to the architecture used [TMY+09]. As another example, the architecture of a given system, programmed in Java, can be monolithic or composed from modular pieces using a service-oriented architecture. In large systems, the service-oriented architecture can be a great boost for productivity [Jon08, p. 220]. These reasons give us the foundation to study software architectures in relation to software engineering productivity. Overall, there are two main reasons to study better ways of organizing software architecture:

1. Improved productivity enables cost savings and advantage in time-to-market.

2. Due to the specific nature of each software case, there is always room for improvement.

Reason number #1 deals with the competitive nature of the software market: unfit companies will be driven out of the market. Reason #2 gives the inductive clause for development: since there is always room for development, not seizing these opportunities makes the company risk losing its fitness. For these reasons, we need to broadly chase all techniques that take us further.

This thesis aims to improve engineering productivity by focusing on programmer productivity. The overall theme is characterized by the self-organization of software architecture on various levels. We describe various model-based approaches that we have employed in real industrial projects with real world constrains being applied. In many cases, we use advanced programming techniques, such as generative programming to achieve our target. We explore the traditional model-based engineering and combine those with program analysis techniques, producing a novel combination. In this thesis, we show that using the program under development as the source model can benefit the programmer in ways of improved type safety, reduced effects of internal dependencies, and improved module composability. These self-configuring components become architectural elements of the software. In this thesis, these properties are examples of approaches for making it easier to ship correctly working software in a shorter time period.

We propose a new way to combine model-driven development and generative programming. To increase industrial relevance, our solution takes the limitations of agile development models into account. Our treatment stems both from industrial experience and academic discussion of software development techniques. Although our goal is not to improve software engineering at the level of engineering processes or engineering team composition, all our proposed engineering techniques can be and have been implemented in current agile industrial environments.

The techniques are presented in a form that describes specific problems in specific programming environments. Our solutions of using specific tools and techniques in specific contexts can be seen as examples or larger applicability in other contexts and using other tools. This is a way to demonstrate the wide applicability of our proposition of building long-lasting software systems via introduction of self-organizing software components.

1.1 Summaries of original publications

In this section, we describe the included publications and show how they are related to the subject of the thesis.

1.1.1 Paper (I)

In paper (I) we describe a technique for building a software product line of mobile games for resource-constrained cell phones. In this context, the usual programming techniques, such as interfaces and abstract classes for building modular software architecture are problematic due to the extra overhead introduced by these classes. For this reason, many organizations in the industry have abandoned object-oriented programming practices. In this paper, we present an algorithm for reducing the architectural overhead of interface classes and abstract classes by removing those architectural classes that are used due to software product line derivation. This way, the company does not need to compromise between maintainability of its product line and single product resource consumption.

1.1.2 Paper (II)

Paper (II) discusses how a language processor can be implemented incrementally using a generative approach. The main contribution is an object-oriented construction, which allows parsing of object-oriented context-free grammars within object constructors, despite the fact that in most object-oriented languages the object constructor is not polymorphic. We present an incrementally extensible parser generator, which employs a variant of the Visitor pattern to improve the error messages produced by a strongly typed implementation language in case of incompatible changes to the processed grammar.

The paper can be seen as a traditional example of model-based generative programming. A parser generator, which uses a grammar as its input, generates a part of the program. For language processing, actions for grammar productions can be hooked to the generator. This structure avoids the inherent problems of generative programming, where it is difficult to interweave generated and handcrafted software parts.

1.1.3 Paper (III)

In paper (III), we document a case of building support for validating software product line instances. We approach the problem by re-using a well-known computational abstraction, regular expressions, to build a model

in a known analysis technique of feature-oriented domain analysis. We document application of a textual language as the source model and its implications in the context of an industrial software product line in the telecommunication provisioning domain.

Here we present a theoretical bridge from regular expressions to feature modeling. As a practical report, we also survey the model evolution that was experienced during development of a case study software product line.

1.1.4 Paper (IV)

In paper (IV) we explore the idea of using executable code as the model in a model-based software engineering technique. In an industrial project, we had identified a problem of copy-paste coding in connection with Java's standard component-based user interface building technology. In the paper, we present a tool for reducing internal dependencies in architectures using the standard Java web development stack.

The resulting self-configuring component allows developers to build web applications with fewer dependencies and improved support of producing more precise messages to be shown to the user. This technique is useful when considering the maintenance tasks of the software, as possible changes to the user interface layer are automatically reflected in its dependent behavior. The automatic reconfigurations are also useful in the case of product lines, where behavior changes based on the chosen configuration variant.

1.1.5 Paper (V)

Paper (V) presents another example case of using software code as the model for model-based software development. When using an object-to-relational mapping software component, a common problem is the dependency between a data-processing algorithm and its corresponding database query code. In this paper, we present a self-organizing database query component that analyzes properties of the algorithm and constructs the corresponding database query. This way, should the algorithm change, the fetching code is automatically updated as well.

Another benefit of this approach is that the need for re-defining the database queries is reduced, since the automated query constructor component can analyze and create corresponding queries for multiple processing sites.

1.1.6 Paper (VI)

In paper (VI) we document the results of a randomized, controlled experiment of performing maintenance tasks using the technique presented in paper (V). We assigned base software with a list of seven development tasks to randomly divided groups of students. The first group's base software was implemented by using the traditional object-to-relational mapping implementation. The second group's software used the technique presented in paper (V).

We used the number of correctly implemented maintenance tasks as the measure of success. The result of the experiment shows that using the self-organizing database queries resulted in a statistically significant difference between the groups with respect to code quality, as the members of the second group produced 22 correct submissions while the control group members were able to return only 5 correct submissions. Since the time consumed during the experiment did not vary between the two groups, it is evident that being able to produce four times as many correct submissions in the same amount of time is a great boon for productivity.

1.2 Thesis contributions

The papers introduced in the previous section are manifestations of our approach of building long-lasting software in practical settings. We have formulated our proposition to fit constraints mandated by e.g. agile software processes, since all the tools and techniques for productivity improvement can be and have been implemented in a sprintable form.

Model-driven engineering is a recent movement with a promise for improved productivity. Productivity gains in limited domains, such as compiler construction make the idea of raising the level of abstraction appealing. However, combining agile process with engineering approaches that include any significant investment, or up-front planning, before the engineering discipline can be employed can turn out to be problematic. Examples of this have been discovered when combining agile processes with product-line engineering [McG08] and requirements engineering [PEM03], and when applying architectural thinking with agile projects [ABK10, Mad10].

To fix these problems, we propose a flavor of model-driven engineering that takes into account the restrictions imposed by agile software development process models. This approach of bottom-up agile model-driven development recognizes smaller sub-domains within the software that are amenable for lightweight modeling. These small models can be used in traditional source-to-target generative programming or in some cases the

source code itself can be treated as the source model, thus reducing redundancy. The bottom-up modeling approach entails lighter initial investment than domain-specific modeling, and thus allows fast experimentation cycles within the limits of tightly time-boxed agile iterations.

We pay special attention to self-configuring software components. A self-configuring software component contains instructions for placing the component in question into different contexts. The component knows how to adapt its parameters, up to a certain degree, when the context it is residing in changes. The central idea is to reduce the harmful effects of internal dependencies, making it easier to maintain and further develop the software. When the architecture of a software utilizes such self-configuring components, the architecture can be said to be self-organizing.

Large software systems are full of interdependencies between modules and other elements. For example, when using arrays as storage for holding objects, the size of the array must be large enough. This is a dependency between the size of the array and the number of elements placed in the array. A self-configuring array analyzes its usage sites and automatically determines the required size. This way, if the usage pattern changes due to software evolution or for other reasons, the size of the created array is automatically updated to reflect the changed reality. This kind of configuration cannot always be done automatically - the problem is unsolvable in the general case. However, in many cases it is possible to automate the updating of dependencies, by using well-known software analysis techniques such as control flow graphs, variable def-use chains, class hierarchies, and other data structures that the compiler of a programming language already uses during its compilation task. Currently the results of these analyses are typically contained only within the compiler internals and are not available for application developers.

A self-configuring software component modularizes these kinds of analysis tools and makes it possible to employ existing information to better benefit in improving software maintainability, improve software architecture's support for implementing features and optimize established software product lines.

Our proposed techniques have long been used within homoiconic programming environments. Generative programming is a common practice in many software products using functional languages as their implementation tool. Our results propose that many of these techniques are applicable in a non-homoiconic environment as well.

Our concrete contributions include a number of tools for generating and optimizing architectural and code-level decisions in various domains,

namely the following:

- An optimizing compiler for the Java programming language that removes unnecessary abstract classes and interface classes, as documented in paper (I).
- A top-down parser generator for object-oriented context-free grammars using object constructors, which was previously believed to be impossible. As a side-product, we present software architecture for building extensible semantic processors for these parsers in paper (II).
- A fast technique for comparing feature models by reducing the compared models to regular expressions, as presented in an industrial context in paper (III).
- Two cases of using self-organizing software components in different non-homoiconic domains, as presented in papers (IV) and (V).
- An empirical validation of the tool presented in paper (V) is conducted in paper (VI).

Overall, the contribution of the thesis can be summarized as showing meta-programming and generative programming as viable vehicles for model-based self-organization of software systems, and showing that the approach is profitable in an industrial setting.

1.3 Thesis organization

We present a number of case studies in different contexts and their imposed constraints. In these cases we show how meta-programming, self-organization and generative programming can be used to improve the software engineering process and to produce better software that is easier to modify according to changing needs. Industrial experience is present in all cases; however, in some papers the industry-specific parts have been faded away in order to demonstrate wider applicability. Types of employed models vary between traditional, external models (such as UML) and internal models, where the program source code is used to self-configure another part of the program.

As a more specific list of thesis organization, we list the collection of papers under the following topics:

1. Industrial experience (IE) - whether the contribution of the paper has been applied in industrial context explicitly (exp), implicitly (impl) or not at all (none).

2. Empirical evaluation (EE) - whether the paper contains empirical evaluation; one star for a case study; two stars for industrial practice documentation; three stars for a randomized, controlled experiment.

3. Type of models (ToM) - whether the paper documents usage of external models (ext) or internal models (int)

4. Generative programming (GP) - whether the technique presented in the paper employs generative programming

	IE	EE	ToM	GP
Paper (I)	exp	**	int	x
Paper (II)	impl	*	ext	x
Paper (III)	exp	**	ext	x
Paper (IV)	impl	*	int	x
Paper (V)	impl	*	int	x
Paper (VI)	none	***	int	x

This thesis consists of the peer-reviewed publications and an introductory part. The introductory part is organized as follows. Chapter 2 explores the area of software engineering by first defining the concepts of software and productivity. Chapter 3 continues with a review of existing literature on related programming techniques of metaprogramming, literate programming and model-driven software development. Chapter 4 reviews the role of bottom-up modeling, as opposed to the traditional top-down modeling. Chapter 5 introduces the idea of using the program itself as the source and target models for model transformations. Chapter 6 concludes the introduction and discusses new research topics revealed in this thesis. The included publications follow the introductory part.

Chapter 2

Preliminaries

This chapter contains the required understanding of the background of this thesis. The preliminaries concentrate on various aspects of software engineering: agile engineering processes, non-functional requirements, productivity and the paradox of process improvement in an agile process. The rest of the thesis operates in the light cast in this chapter: we concentrate on the problems identified in this chapter.

The chosen viewpoints do not try to cover the full range of activities being done in software projects. Instead, these viewpoints are chosen for discussion because they tend to be sources of hard-to-fix problems for many software projects.

Section 2.1 discusses the general state of software engineering. Section 2.2 extends the discussion to currently popular agile software engineering and presents the Scrum process model. Section 2.3 introduces non-functional requirements with a special focus on maintainability and modularity, as these -alities are in the core of building long-lasting software products productively. Section 2.4 changes the focus to productivity in software engineering, which is the key for companies and other software organizations to be able to compete in the market place. In section 2.5, we develop a model for investing in software process improvement and introduce the paradox of process improvement in agile processes. Finally, section 2.6 explores the ways of resolving the paradox.

2.1 Software engineering

ISO/IEC/IEEE Standard 24765 defines software engineering as *"the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of*

engineering to software" [ISO10].

Software engineering as a discipline was born in the late 1960's as a response to the *software crisis*, a term used to describe the phenomenon of software developers and software tools failing to keep up with the rate of hardware development [Nat68]. Advances in hardware allowed more complex software to be developed, but the means for handling the increased complexity was lagging behind.

Since then, software engineers have explored ways to build larger and more complex systems correctly and in predictable schedules. Generally, the final solution is yet to emerge.

However, in recent years, the area of software engineering has been maturing at a fast rate. In 1994, researchers of the Standish Group reported that only 16% of software projects were completed in time and in budget [Cha94]. In 2003, this number had doubled to 34%, as reported in a follow-up text [Cha03]. By 2009, the success rate had stayed in about the same, reporting 32% success in delivering projects on time, within budget, and with required features and functions [Cha09]. Especially larger projects tend to be failure prone: according to productivity research, projects with over 10,000 function points[1] have an alarming cancellation rate of almost 50% with the remainder being delivered late and over budget [Jon08, p. 216].

There has been an impressive improvement in success rates since the early 1990's. Still, much room exists for doing better. During recent years, one project out of three can be seen to be successful in these terms. Even for a success rate of 80% there is the other side of the coin: one project out of five still exceeds its time or budget constraints. Although the Chaos reports have received negative feedback on their methodology and data validity [LEV10], the longitudinal data series is often cited as the definitive source of high-level project success in software engineering.

Yet another viewpoint to this subject is that succeeding to meet time and budget constraints is mainly the interest of the project manager of a given project [AR06]. For other stakeholders, the primary interest can be something else. For example, the client of the project is often happy to exceed the initial budget, when new opportunities for building better functionality are discovered during the project [SAR12]. For this reason, whether a project meets its time and budget constraints cannot be considered as the definitive success measure - but instead, the notion of success seems to be relative to the context and perception of the stakeholders [MM10].

[1] We will discuss the concept of function points in section 2.4

If we choose to dismiss the properties of project meeting its budget and completing the features on time as a definitive source of project success, the next question is what is the alternative definition. One answer is that we argue that *small* overcomes in budget and schedule do not matter; and being *slightly* short of the functionality can be corrected at the maintenance phase. Another view is to regard software as a way of operation; the software evolves along its surroundings. Then, the question is not about meeting the budget and schedule, but about being able to respond to change requests in a timely manner.

Software process models

Great deal of attention in software engineering has been given to find out the suitable process models for software construction. It was already quite early understood that software with any non-trivial complexity needs to be developed iteratively [Roy70] instead of the one-way process from requirements to coding and operational use, which some people call "the waterfall".

A classical way to characterize of different types of software systems is to divide them into three categories as follows [Leh80]:

- S-type programs, which are those that can be specified formally.
- P-type programs, which cannot be specified, but an iterative process is required for producing them.
- E-type programs, which interact with the real world, thereby changing it. A feedback is needed to evolve the program according to new needs.

The big majority of commercial software lies in the E-type category. New software enables new ways for organizations to work, which is reflected in organizational structures. Changes in the organizational structures then initiate changes to the software as well.

Scientific literature contains a number of different process models that try to embed the requirements of the E-type category. Well-known examples include the spiral model [Boe88] and the Rational Unified Process (RUP) model [Kru03]. Since nowadays the main use of these models is mainly limited to frightening sophomore-year students, we do not cover their specifics further. The interested reader can see the standard software engineering textbooks, such as [Som10, Pre10, Sch05] for a more thorough discussion of these software process models.

During the last two decades, the mainstream of software development has turned into using different kinds of *agile processes*.

2.2 Agile software engineering

The currently popular agile process methods help projects to avoid big mistakes of producing the wrong product to the wrong customer at the wrong time. Improved communication with the customer, learning effect within agile iterations and time-boxed development all contribute as process-level reinforcements to project work.

The agile manifesto [HF01] gives four high-level guidelines for agile software projects:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

These preferences are not to be regarded as mutually exclusive choices, but rather as experience-shown preferences on which to give priority in case of conflict. For example, a study on developer perspectives in software development reported a case where the project that developers felt to be the most successful was also reported to be most lacking in tool support [Lin99].

For a general overview of the various agile methodologies, the interested reader can read a comparative survey by Abrahamsson et al. [AWSR03]. Currently, the majority of agile development teams use the Scrum development model, with over 75% deployment rate reported in year 2011 [SS12].

Scrum

Scrum [Sch95] is an agile process model intended to fix deficiencies in its predecessors, the waterfall, spiral and previous iterative models. The main idea is to loosen the requirements for the process during iterative sprints: only preplanning and system architecture phases and the closure phase have a defined process. Inside the sprints - when the heat is on - there is minimal bureaucratic overhead. The name *Scrum* was first used to describe the way how Rugby players change their playing style according to events in the play field: during the game, a player cannot ask directions from

the coach, but he needs to take his own initiative. The same is thought to apply to software development as well: for maintaining flexibility, not every change needs to be accepted by the project management. Figure 2.1 gives an overview of the activities in the Scrum methodology.

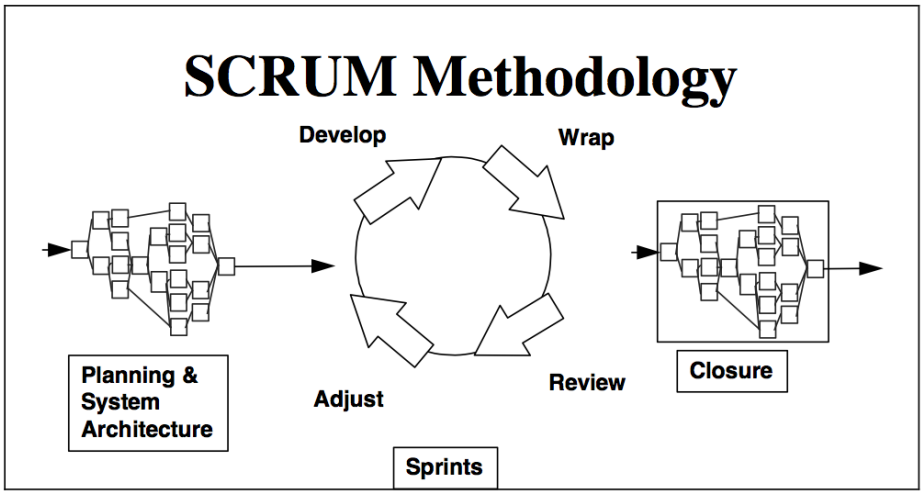

Figure 2.1: Scrum methodology [Sch95]

The interplay between well-defined, rigorous process and creativity-enabling freedom is balanced by having the planning and closure phases use defined processes. Here all processes, inputs and outputs are well defined and knowledge of how to execute is explicit. The flow is linear, with some iteration in the planning phase. The sprint phase is a different beast: it is an empirical process. Many activities in the sprint phase are unidentified or uncontrolled. Management-wise, it is treated as a black box, with only limited controlling interfaces, such as risk management given to prevent chaos, but otherwise targeting to maximizing flexibility [Sch95].

A sprint is a time-boxed set of development activities. The usual time limit ranges from one to four weeks; based on product complexity, risk assessment and degree of required oversight. Within the sprint, one or more teams concurrently execute phases of development, wrap-up, reviews and adjustment. Development consists of the actual doing: defining what is needed to be done in order to fulfill the development items in the backlog: domain analysis, design, development, implementation, testing and documentation. Wrap-up is the phase of building deliverable software. Review activities are the parts that require team communication to present work and review progress. Adjusting activities are those adjustments that are

discovered after reviews [Sch95].

It is important to note that these activities run sporadically within the iteration. During planning and system architecting, the general strategy for the next iteration is defined. However, if the situation changes during the sprint, the plans need to be adjusted 'on-the-fly'.

In the beginning of this section we reviewed how only a small fraction of software projects are able to deliver the required functionality within schedule and budget. Achieving this target is easy in Scrum. This is because all three components of this 'iron triangle' of required functionality, schedule and budget are defined in the project with the permission to redefine when the original estimates turn out to be overly optimistic.

Overall, the Scrum method promises throughout responsiveness to environment with a permission to use unlimited creativity within sprints - as opposed to cookbook approaches in the predecessor process models. The problem of knowledge transfer is thought to be handled through teamwork during the project.

From the productivity viewpoint, the Scrum model has been successful, especially in small projects. According to productivity research, Scrum and other agile software engineering methods are providing the best productivity rates amongst the compared technologies in applications of size 1,000 function points. For example, Scrum combined with object-oriented programming is said to achieve three times higher productivity than the traditional waterfall: Scrum+OO averages to 24 function points per staff month while the waterfall method averages to only 8 function points [Jon08, p. 220].

2.3 Non-functional requirements

When deciding whether a software product meets its specification, software professionals tend to primarily think about the functional scope of the product [AR06]. However, the causes for software to not meet its specification is seldom related to not being able to provide the required functionality; more often the root cause is related to poor quality [Jon95] or other non-functional requirement.

Non-functional requirements document non-behavioral aspects and constraints of the software. In addition to performance requirements, non-functional requirements include a number of "abilities", e.g. reliability, usability, security, availability, portability, and maintainability [Gli07].

The non-functional requirements are notoriously difficult to express in contracted software development [CdPL02, PKdWvV12]. In the view of

agile productivity, the first five abilities mentioned above can be regarded as normal requirements: regarding these, the software can be iteratively developed until the customer is happy. The last one, maintainability, has a different nature. It is the degree of maintainability that dictates how efficiently all other abilities and emerging new functional requirements can be implemented. For this reason, we will study maintainability a bit more.

Maintainability

According to Lehman's first law on software evolution, software is a living entity [BL76]. It is not enough to get a software product deployed into production once; instead the time of deployment is when a new software entity is born. The newborn baby starts to grow through maintenance tasks, which adjust the functionality of the software to fit the changing environment it lives in.

Maintenance tasks often consume the majority of resources spent during the lifetime of a software product. Software engineering knowledge cites the amount of maintenance effort to range from 66% to 90% of the overall spending in software projects [Pig96, Sch05, SPL03].

The importance of maintainability is especially relevant when operating in the agile mode. The repeated iterations with the proposition of embracing change can cause unbearable pressure if the software has not been built to be maintainable. Actually, some professional programmers advocate that programming should always be done in the maintenance mode [HT99, p. 27].

Evolvability has been mentioned as being on the top-level importance in computing [Den03]. Some researchers even argue that software evolution is the most important factor to influence productivity in any software development project [Nie02]. Especially in the area of agile software development, this statement is true due to short iterations and the need for ability to change the direction of development after any given iteration.

Maintenance-related tasks have a high relative weight in effort distribution charts. Thus, applying techniques to make maintenance-related tasks easier should be a key priority for software designers and in software architectures. For example, a central problem in software maintenance is in understanding all the internal dependencies in the system. Systematic refactoring can be applied to reduce the effort of changing the same code in many places [Fea04, p. 269-287]. However, this kind of advice is targeted to the maintenance programmer, who needs to understand all the dependencies in order to be able to refactor a better design. It would be better to design the system in the first place in such a way that the dependencies

do not hurt the maintenance programmer at all.

In general, maintenance refers to all the tasks that are required to keep an already shipped software product to fit in its changing environments. A usual classification of maintenance splits these tasks to *corrective maintenance*, *adaptive maintenance* and *perfective maintenance* [Swa76, LS80, p. 68].

Corrective maintenance consists of the tasks required to fixing the software bugs; all the work that could not be billed if the software was still under warranty. Adaptive maintenance refers to tasks where there is a need to change the software due to changes in data input or requested changes in output formats or interfaces. Perfective maintenance in turn consists of enhancements for users, improvements in software documentation and improvements in software performance [LS80, p. 68]. At later times, there has been a tendency to further classify parts of this effort into *preventive maintenance*, it being included as the fourth maintenance factor in the IEEE standard on software maintenance [ISO06]. Preventive maintenance consists of those tasks that are not performed due to bug fixing or user-requested enhancement, but to prevent failures in the future or to make the other types of maintenance tasks easier. In software context, many preventive maintenance tasks can be thought of being code refactorings without observable changes in functionality [Opd92].

Of all of these categories, the non-corrective maintenance tasks are the ones where most of the effort is being spent. Non-corrective maintenance has consistently accounted for over half of the effort spent in maintenance [AN93, DA10, LS80, p. 68]. In summary, this is the category of software work where the biggest share of software teams are spending their largest amount of time. A direct economic implication is that any techniques that can be used to reduce the absolute amount of effort spent in this share do significantly reduce overall software costs.

For maintaining existing software, there are textbooks that recognize common cases of maintenance-related refactoring tasks. Current development environments offer a range of automated refactoring tasks. E.g. [Fea04] documents 24 techniques for breaking dependencies in legacy object-oriented code that does not (yet) follow the current best practices of building unit tests, test-harnesses and design for testability.

Modularity

Building modular systems is a way to reduce the required effort in maintenance. Using correct tools to solve the right problems can yield big benefits. An early study reports that 89% among users of structured programming

reported improved maintainability of their code over unstructured programming [LS80, p. 7].

Programming languages contain a number of modularity techniques to improve productivity in software construction. For example, object-oriented programming and aspect-oriented programming are essentially ways to use suitable abstractions for developing reusable building blocks.

Program development by composing the program from a number of smaller building blocks is thought to be an efficient way of software engineering. However, the real problem is how to find the correct components and how to glue them together. *Object-oriented programming* introduces the notion of object as the basic building block. *Aspect-oriented programming* supports different aspects to be introduced to the program, allowing the functionality of the program to be defined from different angles. In both of these approaches, the joining of different modules is still done manually.

Designing modular architectures is the key when planning to support maintainability of software. There are a number of obstacles to hurdle when approaching a good decomposition of software to modules. However, the chosen decomposition greatly affects how well the software is modifiable, as was shown in an early study of two software implementations with different modular decompositions [Par72]. Although a good modular decomposition helps in further maintenance, it should be noted that in many cases the "perfect" decomposition does not exist, but instead any chosen decomposition is a compromise that favors one maintainability aspect over another. Any chosen way to modularize the software will serve some purposes on the expense of some other. This is known as the *tyranny of the dominant decomposition* [TOHS99].

The term *modularity* is unfortunately rather overloaded and requires clarification, since its meaning varies from one environment to other. For this section, we will use a metaphor of Lego® bricks to discuss modularity in a broad sense. These widely known plastic bricks can be seen as a metaphor for modularity. Each one of the little plastic bricks defines a certain connectivity interface through its upper and lower interfaces. When two bricks are connected through these interfaces, they form a new element, which can be regarded as a single, larger element with its own connectivity interface. Generation after generation, children's creativity is tickled with these simple but infinitely modifiable toys – there are hundreds of millions of ways to connect six 2x4 bricks [DE05].

One can reason about these two bricks and their corresponding composition algebra: two 2x2 bricks can be connected in 18 different positions[2].

[2] 72 different ways if the cheeks of the bricks are considered to be distinct

One of these combinations is shown in Figure 2.3 as an attempt to start building an infinite stream of steps.

Figure 2.2: Two 2x2 Lego® bricks connected

Adding a third 2x2 brick makes the compound model even more complicated: now there are 729 different ways the three pieces can be connected. When continuing with the attempt of building the stairway to heaven, a previously unknown force, *gravity*, steps in. Each object in our physical universe has mass, which causes all objects attract each other in force proportional to their mass. When the structure in Figure 2.3 is placed on an even surface, its center of gravity is outside the bottom surface area of the blue brick. Thus this structure crashes when no additional force is used to keep it in balance.

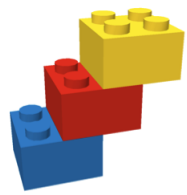

Figure 2.3: A three-step model with its internal balance out of bounds

Working in software development often reveals similar forces in the software. The software and its structure can work well in one configuration, but after a number of changes following the current architecture, its internal "center of gravity" is exceeded and the system crashes. Due to the nature of maintenance programming, the maintainers are not always fully aware of the forces defining the internal "center of gravity" of the maintained software.

Software is full of these kinds of hidden dependencies, where a seemingly innocent change can cause malfunction or crashing. Programmers have learned to defend themselves and their colleagues and customers from excessive rework by designing their software in a way that is resilient to future changes. One way is to document the anticipated ways of future

modifications.

A pattern of documenting the maintenance tasks [HH04] can be used to expose the software's internal dependencies and to guide maintenance personnel to the right direction when modifying the software's structure. For our bricks metaphor, a maintenance pattern would be stated as instructions to maintain the center of gravity in the overall structure, possibly by adding counter-weighting bricks to any structure-modifying addition.

It is easy to confuse intentional and accidental maintainability. Program designers and programmers often use constructs that are primarily intended to act as a functional construct, while their purpose is just to build safeguard constructs to improve maintainability. We can call this kind of design a case of accidental maintainability. We need to distinguish those constructs from intentional cases of a priori, explicitly designed maintainability.

In the Lego brick example, a case of accidental maintainability for maintaining the brick-structure's balance could be to place a certain number, say five, of counter-weight bricks on the first step. This way, the structure's center of gravity would stay within its constrained limits when new steps are added, up to five steps. However, this approach is problematic in two ways: first, it wastes resources when less than five steps is needed. Second, when there is a need for the sixth step, the system crashes.

A system of intentional maintainability in this example would identify the dependency between the number of steps and its implications to internal balance. Intentional maintainability would then instruct the stairway builder to add the required counterweight for any number of steps.

2.4 Productivity in software engineering

Productivity is a key issue in professional software development. In many software businesses, being able to produce more functionality in a given timeframe is advantageous: in a productive environment customers gain more value from software projects, and project professionals have higher job satisfaction. However, relatively little attention has been paid to actual productivity improvements in agile projects. This is surprising, given the fundamental nature of productivity and productivity improvement in the history of industrialized world.

The definition of productivity involves the ratio of outputs to inputs, such as material, labour, and capital. Thus, productivity is defined as follows [Boe87]:

$$productivity = outputs/inputs \tag{2.1}$$

An organization that can produce more products with less resources is more productive. Being able to produce more outputs with the same input makes an organization more productive; and producing the same output with less input again means better productivity. However, in the context of software engineering, the definition of productivity is problematic, mainly due to the definition of output part of the equation.

In many cases, the inputs can be estimated. This variable includes the resources consumed in the project. While some disagreement on the attribution of indirectly involved resources such as departmental secretary services is unavoidable, usually companies and organizations can settle to a rough agreement on division of these costs. The inputs can be in various forms, such as computing resources, network resources in addition to human resources. However, a conversion to present currency value can unify the consumption to a single number [Boe87].

The output part is the problematic to define [Boe87]. One suggestion is to define the outputs as the delivered functionality at certain quality level [Nie02]. However, this attempt does not solve the measurement problem: what is the unit of *functionality* or *quality*? In order to measure, there needs to be units of measurement. In the following, we review few traditionally used measures, the lines of code and function points.

Lines of code

Lines of code and its variants, such as source lines of code and delivered lines of code are the easiest variables to measure. Even crude tools, such as the *wc* utility can be used to give a crude estimate and more accurate tools can easily be obtained from open source software repositories. For this reason, lines of code is industrially used measure for software size.

Also some scientific studies, such as [AK04, MAP+08] use lines of code as the output measure. However, lines of code is a notoriously bad measure for productivity [FP98, pp. 405-412]. Actually, quite the contrary: researchers associate *less* code lines to be the key to productivity [TB03].

Bill Gates is quoted of saying, *"Measuring programming progress by lines of code is like measuring aircraft building progress by weight"*, which is a nice analogy in two ways: first, it emphasizes that the goal in building a good airplane is not to make it heavier, but to make it *lighter*. The second interpretation of the analogy is that big aircraft are distinguishably different from small aircraft; similarly, a software whose size is measured in tens of millions of lines of code is different from a software whose size is in tens of thousands.

In the micro scale, the measurement in lines of codes is irrelevant. As

an example, consider the code listing in Figure 2.4. This code loops for a specified number of times, and sleeps for one second. An equivalent code could be written by unrolling the loop. This means to remove the for-loop and to copy the loop body for the five times and substituting the loop variables with corresponding values and performing the string concatenations at coding time. In this case, the equivalent code would be as shown in Figure 2.5.

```
for(int i=1; i<=5; i++) {
    Thread.sleep(1000);
    System.out.println(i+" second" +(i==1?"":"s")+ " passed.");
}
```

Figure 2.4: Code for watching the time go by

```
Thread.sleep(1000);
System.out.println("1 second passed.");
Thread.sleep(1000);
System.out.println("2 seconds passed.");
Thread.sleep(1000);
System.out.println("3 seconds passed.");
Thread.sleep(1000);
System.out.println("4 seconds passed.");
Thread.sleep(1000);
System.out.println("5 seconds passed.");
```

Figure 2.5: Unrolled code for watching the time go by

The unrolled version is probably more efficient, since it does not spend time in calculating the loop conditions and constructing the current message object. However, very few professionals would prefer the unrolled version over the looping version in source code form: it violates the *do not repeat yourself* (DRY) principle, which has been found out to be a good guideline in constructing easily modifiable software [HT99, p. 27]. If we were to use line count as the numerator of the productivity equation (2.1), the unrolled version would yield a higher productivity index, since the first version line count is 4, whereas the unrolled version contains 10 lines to produce the same output.

The DRY-principle is not limited to this kind of mechanical loop unrolling, but applies also to questions of code style. To illustrate this idea, let

us consider a simple command interpreter, implemented in the C language as shown in Figure 2.6.

```
struct command
{
char *name;
  void (*function) (void);
};
struct command commands[] =
{
  { "quit", quit_command },
  { "help", help_command },
  ...
};
```

Figure 2.6: Interpreter implementation as an array of structs [SW11, p. 19]

The code in Figure 2.6 defines an interface for defining commands. Each command has a name and a pointer to a function implementing the command. By convention, each function is named after the command it implements.

This is a violation against the DRY-principle: the prefix in the function name in the command table repeats the name of the command [SW11, p. 19]. Thus, it is considered to be cleaner to factorize the command names as a common element, e.g. by using preprocessor directives. An equivalent code, without repetition could be formed as shown in Figure 2.7.

```
#define COMMAND(NAME) { #NAME, NAME ## _command }

struct command commands[] =
{
  COMMAND (quit),
  COMMAND (help),
  ...
};
```

Figure 2.7: Interpreter implementation by preprocessor concatenation [SW11, p. 19]

In Figure 2.7 the code uses the C preprocessor token concatenation operator [Ker88, p. 90] to define a macro for each of the commands. In

this example, after the preprocessing phase the source code is exactly the same as the code in Figure 2.6. The latter version can be argued to be cleaner and easier to understand.

Using line count as the productivity measure does not detect any difference between these two alternatives. However, when programs get bigger and more complex, the quality of the code starts to make a difference. A codebase with needless repetition and many dependencies can easily deteriorate into an unmaintainable state. Using the line count as a productivity metric can add great momentum to the demise of the project.

Function points

The problem with the number of lines as a measure of software development is well known. In order to better understand development progress and to be able to compare similar software written in different languages, the concept of function points has been developed [Alb79].

The idea with function points is to estimate the functionality of the software at higher level of abstraction than just the actual lines of code. When estimating the number of function points in a software, its functionality is dissected into small pieces that can be explained by the function point analysis method.

Function point analysis is based on estimating data functionality and transaction functionality. There are two types of data functionality: internal logical files (ILF) and external interface files (EIF). For transaction functionality, the estimation is based on three types of transactions: external inputs (EI), external outputs (EO) and external inquiries (EQ). Each occurrence is judged to be simple, average or complex. Figure 2.8 gives an overview of how application logic is estimated

For a given feature or component of a software system, the complexity-adjusted data and transaction functionalities are charted through function point analysis tables, which give out unadjusted function point values. Finally, this value is translated to the final function point value by applying the value adjustment factor, which reflects the non-functional requirements of the system. To get an estimate of the whole system, this process is repeated for all of its components.

Function point analysis is a lengthy process. A necessary pre-requirement is the functionality of the software to be well understood, since otherwise it would be impossible to enumerate all the components of the system. For this reason, function points are not widely used in software development methodologies that de-emphasize overly detailed upfront planning. Another problem with function points is that they suit only certain types

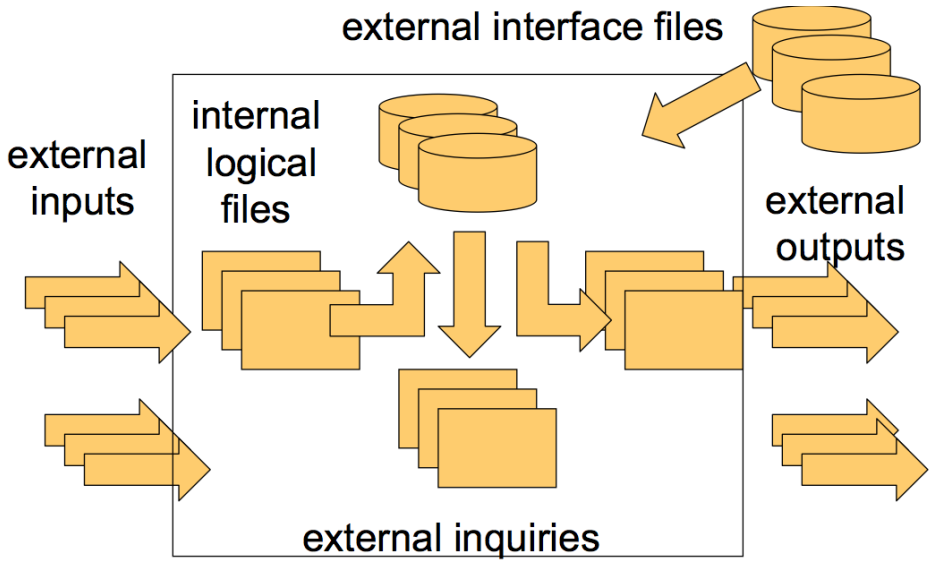

Figure 2.8: Elements of function point analysis

of application software. When applied in an inappropriate domain, such as operating systems, the result can be that functionally similarly looking operating systems targeted to the mobile phone and the desktop receive equal amount of function points, as happened when estimating the Apple Mac OS X full version and its mobile variant iOS [Jon08, p. 274].

Despite of not being a perfect measure, function points do provide a better view on sizing of software than lines of code, especially in sizing form-based information systems. Function points can be applied across different programming languages and development methodologies to gain understanding of whether investments to improved ways of working are delivering the promised benefits. For different programming languages, there are statistics-based equations for finding crude correspondences between lines of code and function points. These equations can be used to anticipate the software's implemented size when its functionality is known in function points; or to estimate the software's functionality in function points when its size in lines of code is known. However, it should be noted that these estimates are rather crude, as the software's architecture and the applied conventions in an individual project can have big effect to the actual results.

2.5 Economic model for investing in software development process

When investing in process improvement in software development, the obvious question is how to estimate the return-on-investment. Return-on-investment (ROI) is calculated by dividing the difference of benefits and costs of a change by the costs of the change [Phi94, pp. 12-13], as shown in formula (2.2).

$$ROI(\%) = \frac{Gain - Cost}{Cost} * 100 \tag{2.2}$$

In software development, the most important result is the working software system. As this result is abstract in nature, it is often hard to estimate causes and effects in the work producing the software. For this reason, also giving exact figures for estimating whether process improvement is justifiable is problematic in many cases. However, the general principles of economic thinking can be used to guide in decision making, although the exact numbers for a given decision might be impossible to calculate.

When a software development team is given a task to implement a software product, the total cost of the project can be calculated as stated in formula (2.3).

$$OC * OT \tag{2.3}$$

In this formula, OC stands for operational cost and OT for operational time. In a single project scope, operational costs are usually mandated by the organization. Operational time is the time that the project needs to produce the required functionality. The more efficient the development team, the shorter time is spent in development. Equation (2.3) thus is dependent on *total factor productivity* [CROB05, p. 3], which is productivity measure that encompasses all the factors in consideration.

For example, if the average cost of one man-month in the project, including the salaries, office spaces, cost of computer leases and so on, is 10 units, operating a software development team of ten people costs 100 units per month. If the software development project takes three months, the total cost of the project is 300 units.

Considering productivity improvement, the use of technologies for software development varies from organization to another. In traditional manufacturing context, the *production frontier* [CROB05, p. 3] refers to the curve of how many output units can be produced per one input unit when the production is scaled and the organization is using the best technology

for production. In production frontier, usually a larger scale yields less required input per output. An organization that is at the production frontier is said to be *technically efficient*.

Software development projects are seldom technically efficient. Capers Jones has presented an estimated an average of 35% efficiency in software development [Jon94, p. 228]. The requirements are typically defined during the project, which causes changes in the technologies for reaching the production frontier. Even if a project happened to be technically efficient, technical changes in development environment can cause the production frontier to move [CROB05, p. 4].

This is good news for productivity improvement: there is a lot of work to be done. However, application of technical changes and improvement proposals for moving towards the production frontier need to be justified. Seldom do these changes come for free: instead, all changes have an associated cost, and they take time to be implemented. On abstract level, the economic decision criterion for performing an improvement, using any given technical change or by adjusting the development process can be expressed as in equation (2.4), where the new term OT' stands for the new operational time after an investment to a changed way of working.

$$OC * OT > Cost + OC * OT' \tag{2.4}$$

In other words, the initial investment to implement new practice or employ new techniques within the project can be justified if the costs with new operations amortized over total operational time are smaller than the alternative of running the operations without changes.

Let us have a hypothetical development practice that gives 25% improvement on productivity with no other consequences to related variables. Implementing this practice has a fixed cost of 50 units and it is immediately implementable whenever the project chooses to. If available at the beginning of the project it should be employed, as the improvement on productivity shortens the required time to implement the project from three months to 2,4 months, thus promising an earlier delivery and overall savings on expenses. However, if the given practice becomes available at the beginning of the last month, it is no longer clear whether it is justifiable to employ the practice for this project. The improved productivity allows the work in the last month to be performed in 80% of the time. However, the saving of 20 units in operating cost does not warrant offsetting the initial cost of 50 units for this practice - but instead causes the overall development bill to exceed the budget by 30 units[3]. However, the improved

[3] 100+100+80+50 = 330

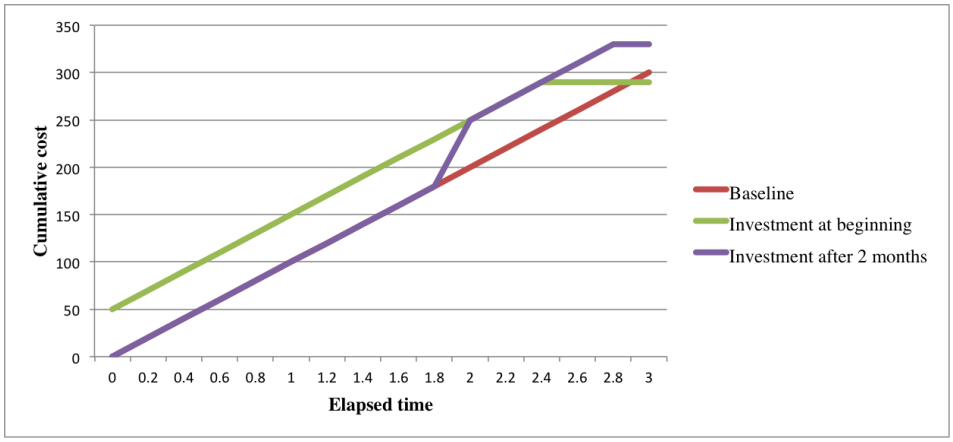

Figure 2.9: Expenditure graph

productivity allows to ship at a week earlier, so depending on the context the higher total cost might still be justified by faster time-to-market.

Figure 2.9 draws an expenditure graph for the three alternatives. The baseline spends all three months with linear spending curve. When the productivity upgrade is obtained in the beginning of the project, the initial cost is higher, but the improved productivity gives faster completion time and smaller expenses. The productivity upgrade, brought at near end of the project still gains faster shipping, but the overall cost is the highest.

In reality, this kind of reasoning for real software projects is very hard, maybe impossible. This reasoning does not take into account Parkinson's observation [Par57]. This observation refers to the phenomenon of work expanding to fill all the available time. Our example assumes a fixed amount of work, which seldom is the case. In practice, the 20% of the time saved in the last month would probably not be used to ship earlier, but instead be used in various, unaccountable ways, such as gold plating the software, which is a known anti-pattern of commercial software development [McC96, p. 65].

Another problem is that productivity rates vary between individuals and team mix-up in orders of magnitude [Gla02, pp. 14-15], and the hypothetical instant productivity boon option is just a project manager's daydream. Instead, various factors, such as the learning curve [Gla02, pp. 23-24], suitability of improvement into context and scalability of the improvement need to be taken into account. These are not easy to estimate beforehand.

However, the basic principle behind equation (2.4) is often encountered

in software development organizations: new practices and technologies are discovered, and a return-on-investment justification needs to be found for employing the practice in question. Unfortunately, most often the exact effect of the practice is not well understood; but instead, practicing professionals are faced with vague explanations of merits of the new technique. Even with a piloting implementation in the organization, it is very hard to measure the exact effect of the new technique, as distinguishing the new technique's effect from other productivity measures is mostly based on individual feelings rather than objective measures.

Process improvement in agile processes

The example in previous section discussed an extremely simple process model with a fixed amount of work to be done. Real software projects, especially the ones employing agile principles, are by nature more chaotic: accepting the fact that requirements are not known beforehand and the fact that the direction of development might change at any iteration, agile software teams agree with their customer on a time-boxed development model.

From the process development viewpoint, the principle of sudden change is problematic, as intra-project improvement cannot be justified via amortizing the investment costs to future iterations. This is because the deal with the customer does not last longer than to the end of the iteration: the next iteration materializes only if the shipping product in the end of the previous iteration pleases the customer well enough and assures him that the value brought by implementing the next iteration exceeds the cost. For this reason, all work in an agile project needs to be in sprintable form [Mad10].

This strict time-boxing of software development in agile process models introduces an additional constraint to the economic decision criterion of equation (2.4). Given a two-week sprint cycle, as exercised by many companies following the Scrum model [Sch95], any process improvement initiative needs to be fit in the same cycle. Traditional justification for return-on-investment calculations spans for the range of many months or up to a year [VS04].

At this point, we need to discuss about value of the work, i.e. what is gained by assigning an engineering team to a given problem. Given a baseline productivity, the value of the work is

$$value = OT * P \tag{2.5}$$

In equation 2.5, P refers to the productivity of the team, which is 100%

for the baseline. Equation 2.4 discusses whether productivity improvement can be justified by *cost savings*. Agile projects are more interested in delivering *value* to the customer. Thus, the economic decision criterion in agile process models translates to the following format:

$$OC * OT * P < Cost + OC * OT * P' \tag{2.6}$$

The difference between equations 2.4 and 2.6 is that in the former, an investment to software development process is justified by the shortened development time indirectly brought by productivity improvement. In the latter equation, the operating time is the same on both sides: regardless of productivity, the team commits to a single, time-boxed iteration at a time. Improved productivity brings more value to the customer: however, there is a paradox in that tighter time-boxing reduces the room for productivity improvement, since the cost of any improvement must be paid within the same time-boxing unit where it is to be taken into use. Otherwise, the team is not delivering full value to the customer.

It will be very hard for the suggested initiative to gain acceptance within the development team if it is not deliverable within agile iterations, as committing to the process improvement also endangers the continuity of the whole project. This happens not only because the external environment is setting the team's development tact to follow the time-boxed iterations, but also due to the internal pressure within the team to follow agile principles of meeting demands of the customer by the end of every iteration.

These reasons can be formulated as the paradox of agile process improvement: In an agile project, the closer the customer is watching the project, the harder it is to implement productivity improving practices.

On-site customer is a core extreme programming methodology principle. However, on productivity improvements angle this principle is contradictory: any investment that is expected to pay off after more than one iteration in improved productivity cannot be justified, because the agreement on the next iteration is based on the outcome of the current iteration. If the development team induces the customer to include additional process improvement efforts to a single iteration, the team is steering away from the agile principles of delivering value at the end of each iteration.

The paradox of agile process improvement is also related to issues of architectural problems in agile projects. As the agreed scope for a given project does not last over the end of the ongoing iteration, any effort spent on improving the internal working methods is perceived as lessening the available effort for value creating activities. Lack of time to consider design alternatives has been recognized to be a problem for many agile projects

[Bab09].

For this reason, the process improvement schemes available to a project using an agile process are limited to well-matured techniques with proven tool support. Examples of agile-associated techniques are e.g continuous integration, unit testing and version controlling. Each of these process techniques are clearly initially implementable within a very short timeframe, thus not endangering the value delivery promise within the iteration. Still, e.g the documentation for Selenium, a well-known web software test automation tool raises the question [HISMP+13]:

> "Is automation always advantageous? When should one decide to automate test cases? It is not always advantageous to automate test cases. There are times when manual testing may be more appropriate."

For Selenium, the authors find two example cases in which investment for building test automation is not beneficial: If it is known beforehand that the application structure is going to change considerably, building test cases would be wasted effort. Another case is that sometimes there simply is not enough wall-clock time or developer time to do this:

> "[S]ometimes there simply is not enough time to build test automation. For the short term, manual testing may be more effective. If an application has a very tight deadline, there is currently no test automation available, and it's imperative that the testing get done within that time frame, then manual testing is the best solution."

This viewpoint emphasizes the need to balance between short-term and long-term goals. Due to the fact that in agile projects the next deadline is always very soon, it is difficult to justify much investment in building e.g. test automation, as it does not bring direct value to the customer by the end of the on-going iteration. However, this kind of investment is mandatory when efficiency in development is a target goal.

In the running example at the beginning of this section, the hypothetical productivity implementation was assumed to be available immediately and at a fixed cost. In reality, this is a rare case to happen. Instead, new practices take time for the development team to get accommodated to, and benefits are often non-linear. For these reasons, project managers are often more reluctant to add new practices to projects when the next deadline is approaching. Which is always too soon.

The requirement to time-boxing forces the improvements to reside within the time-box as well. The unfortunate consequence of this requirement is

that the available improvement techniques are those that are well understood, general principles with proven tool support. No room for experimentation limits the possibilities to trivial projects that are executable regardless of the used process model.

2.6 Recovering from the paradox

Several authors have proposed ideas for shaping the state of software engineering in the future.

First of all, there is a tendency to relax the view that all activities should be financially or technically justified before implementation. On general level, even the most prominent proponents of measurement and control are now suggesting that all measuring is not reflected with an associated benefit [DeM09]. Despite the author's earlier motto of *"You can't control what you can't measure"* [DeM86, p. 1], now the cost of measurement needs to be taken into account: the benefits of control should overweight the costs of measurement. According to the author, there are many projects where this situation does not hold.

The lean software engineering movement also walks in the same direction [PP03]. Instead of giving fine-grained advice of e.g. how to organize the development work into pair programming and 40 hour working week, as is done in the extreme programming literature [BA04], the emphasis is on how to eliminate waste on a more general level. The important change in the mindset is that once a source of waste, such as too much resources being spent on repetitive manual testing, has been recognized, the repetitive parts should be automated.

In a tightly time-boxed iteration, deploying an automated test environment does not make economical sense, given the economic decision criterion of equation 2.6; for that given iteration, doing testing manually provides more value. But for the long run, automating the thing is the key for efficiency, since automating the tests reduces the unit-cost of executing a testing cycle to a fracture. So, the transition to post-agile, lean software development can be observed to happen in many software development organizations today.

This transition can be characterized by the swift of focus from intra-iteration optimization to a longer-term planning. Optimization target is changing to long-term efficiency, although it can mean that the customer does not get full, short-term value at all times. Lean development advocates take this idea even so far that they propose that *results are not the point* in software development [PP09, p. 199]. In this thinking the idea is to build

efficient teams whose overall productivity greatly surpasses the short-term optimizing team's performance.

Chapter 3

Software architectures, models and product lines

Discussing software development productivity is fruitless if concrete approaches to structure development work are not brought into the discussion. In this chapter, we review software architecture, software modeling and software product lines as tools for building better software faster, cheaper and with better quality.

Software architecture defines the bounding box for any given product to succeed: when the architecture is fit for its purpose, building the actual product becomes much easier. On the other hand, when the chosen architecture does not support the product, most of the effort will be spent in circumventing the deficiencies in the architecture [TLB+09].

Software modeling is a key tool in designing architecture. When modeling is combined with generative programming, parts of the software can be automatically generated, thus reducing the manual effort needed to complete the project.

Finally, software product lines are a way to scale software development from a single project to a number of similar projects, enabling productivity increase via shared components that encapsulate commonalities between the different products.

The rest of this chapter is organized as follows. Section 3.1 discusses software architecture and the difficulty of its exact definition. Section 3.2 reviews software modeling and model-driven engineering. Section 3.3 casts a view on software modeling in the context of agile processes. Section 3.4 relates software product line engineering to software architecture and software modeling. Section 3.5 elaborates on how self-configuring components can be employed to form self-configuring software architectures in model-driven software engineering and in software product lines.

3.1 Software architectures

Software architecture is a notoriously hard concept to define exactly. Software architecture researchers at Carnegie Mellon University have collected a listing of 29 different definitions from the literature [SEI13], which illustrates the diversity of the concept.

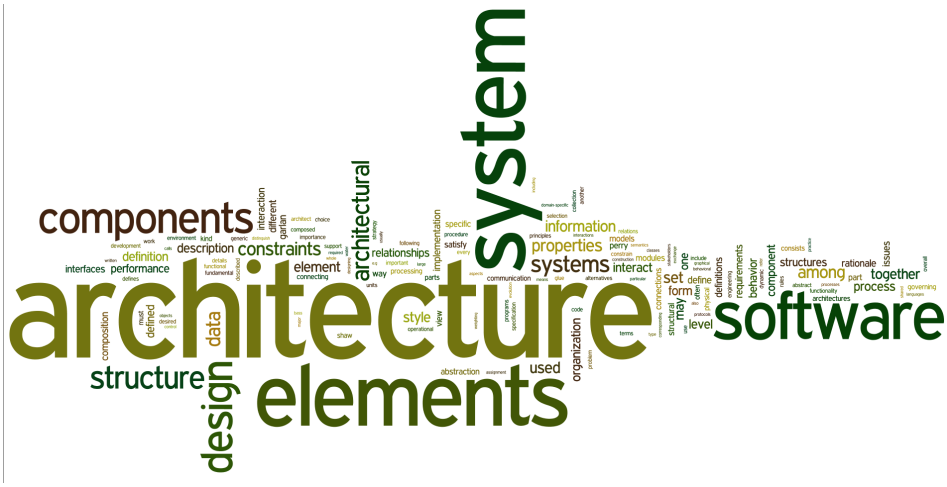

Figure 3.1: A word cloud generated by the different architecture definitions

To give a quick overview on the terms that are usually used to define software architecture, Figure 3.1 shows a word cloud generated from all of these definitions. Based on this cloud, it becomes apparent that software architecture deals with the elements of software systems. Architecture consists of components, structure and the architectural design; the constraints and properties and their relationships.

Many people do not take the term architecture seriously at all. For example, some agility researchers define software architecture as the result of work done by software architects. Software architects, in turn, are defined to be the people who create software architecture [Kru08]. This logic may be written partly in tongue-in-cheek style, but maybe only partly.

Although researchers seem to be unable to agree on the definition of software architecture, it is important to recognize that software always has its architectural design, even though it might not have been explicitly documented. However, a documented architecture helps in understanding the functionality of the software and makes further maintenance easier and thus more productive. Unfortunately, one of the dirty public secrets of software engineering is the lack of documentation. In open source projects,

people asking for documentation are usually answered 'RTSL' [1].

The problem lies in the area of duplication: when software is being changed, the primary locus of activity is to get the software working in the anticipated way. Updating of the documentation might get done, if time permits. Most often it does not.

In the past, a number of attempts to salvage the situation have been done. For instance, literate programming [Knu84] was proposed as a way to write such clean code that the documentation could automatically be generated from it. A more recent attempt is to make the artifacts previously intended primarily for documentation to be the actual source code. This turns our attention to model-driven software engineering.

3.2 Model-driven software engineering

In the past, raising the level of abstraction has been a successful approach to many software engineering problems. Model-driven software engineering is a further step in this direction. The idea is to allow majority of developers to concentrate on the task at hand, without the need to needlessly pay attention to technical, low-level details. This is supposed to yield improvements in three areas [BCT05]:

1. Productivity – developers can solve business problems faster.
2. Quality – use of automation results in more consistent outcomes.
3. Predictability – standardized transformations help to build a predictable development process.

Models are used in various forms. The most obvious form is the traditional boxes-and-arrows type of modeling, in which graphical notation is used to present concepts or ideas and relationships between them. In software engineering, currently UML is the most often used language for graphical representations of the developed software [Kro03, GKR+07]. However, the concept of modeling is not limited to drawing pretty pictures to the whiteboard or into a CASE tool - actually the opposite. Some academics dare even to say that UML is the worst thing to happen to model-driven development [Coo12] due to a number of reasons: It works on the same level of abstraction as program code; the fixed number of diagram types does not provide enough coverage; the language is too complex, but not expressive enough; the division of platform-independent models and platform-specific

[1] read the source, Luke

models is misguided and finally UML makes people to believe that models must be graphical [Coo12].

Models can also be presented in a textual form [GKR+07]. A number of benefits for both the language user and the language developer have been identified, including denser information content, speed of model development, easier integration between languages, platform independence and easier interoperability with version control systems, to name a few. An empirical study comparing the use of textual and graphical models found that participants who predominantly used text scored significantly better in communicating software architecture design decisions [HKC11].

The choice between graphical and textual modeling languages is not a mutually exclusive question. Both can be used at the same time. For example, the UML standard includes an XML-based format for model interchange called XMI [OMG07]. Although the graphical elements of UML are often first cited, the language's constraint rules language OCL [WK03] is primarily a text-based language. Thus, UML is actually a hybrid language. Researchers have produced prototypes for other kinds of graphical/textual model interaction as well [EvdB10].

Another, important architectural choice is the division between external and internal models. The normal interpretation is that models are external to the architecture of a software. In external modeling, the model artifacts are developed in an external tool and are incorporated into the software by the means of manual, semiautomatic or automatic translation.

When using internal modeling, the model is built as part of the software's architecture. For example, object-oriented modeling being done on the abstraction level of a programming language, using the programming language as the sole notation is an example of using an internal modeling language. One example of this approach is well documented as the domain-driven design paradigm [Eva03]. Other examples of internal modeling include the use of domain-specific languages.

Model-Driven Architecture

Model-driven architecture [MM03] approaches reusability by separating concepts into three layers: platform independent model (PIM), platform-specific model (PSM) and program code. Traversal between these layers is done via transformers: a platform independent model is translated to a platform-specific model by using a transformer, which augments the model with platform-specific attributes. A similar transformation is applied when translating the PSM into program code.

A typical platform independent model is expressed as a UML class di-

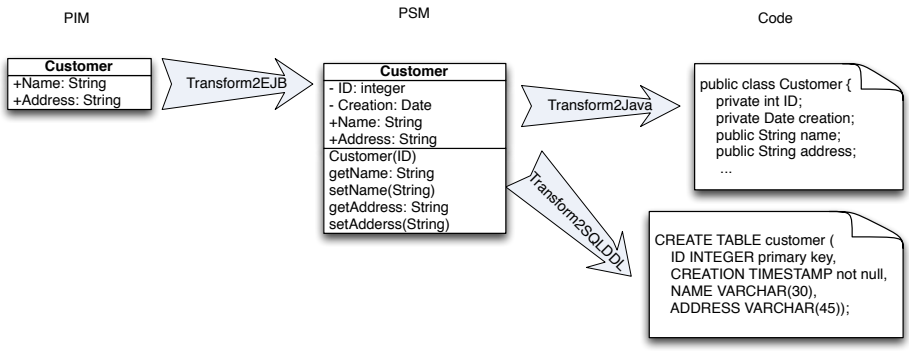

Figure 3.2: A UML model with transformations to Java and SQL

agram which contains only class attributes, possibly with programming language-level visibility information and data type annotations. A transformation creates corresponding programming language, e.g. Java classes, with accessing methods for each of the public attributes, or data-definition statements for a relational database.

Figure 3.2 represents a typical case, in which a highly abstracted class model (PIM) expressed in UML is first transformed to a platform-specific class model (PSM). In this transformation, the class is augmented with platform-specific features, such as accessor methods and constructors. This model is then further transformed to programming language code by the next transformation. Yet another transformation generates the corresponding database definition.

These transformations contain target-specific parametrization, as the transformation contains information about the target platform. In the UML-to-Java transformation, UML standard visibility rules are followed, but a data type transformation from UML integer to Java int is performed. In the UML-to-SQL transformation, similar platform-specific knowledge is being encoded. Most notably, the transformations also contain information about the system that is not shown in the source model. For example, the knowledge about different field sizes for Name and Address that have the same data type in the source model is encoded into the UML-to-SQL transformation.

Two distinct interpretations of using model-driven architecture tools have been identified. The first is the *elaborationist* approach [KWB03], in which the idea is not to even try to provide a complete set of operational transformations. Instead, the models are used as an initial kickstarting set, and the transformers produce a skeleton of the produced software. After

the initial generator run, programmers take over and do further modifications by hand. Obviously, this approach entails the round-trip engineering problem, since the initial models are useless after a few modifications to the generated code.

The second interpretation is the *translationist* approach [MB02], in which the target is to have the human modeler to work only in the model world and translations then generate the whole application. This is done, for instance, by introducing a new UML profile with a defined action semantics that is used in the application generation. This approach highlights one of the most fundamental problems with model-driven architecture: the inflexibility of toolsets and the (morbid) rigidity of extensions.

A notable shortcoming in using UML class diagrams to express the platform independent models is lack of extensibility [FGDTS06, SB05]. A class diagram can directly express only a limited set of parameters, such as visibility, data types, and default values. Further extensions require using UML profiles. A number of proposals for using profiles to express variability have been presented, e.g. [PFR02, KL07].

The problem with UML profiles relate to tool support and decreased interoperability and the dominance of chosen modularity. First, profiles may or may not be supported by the used toolset; toolset immaturity is one of the problems identified in the model-driven engineering literature [MD08]. This leads to decreased interoperability, as transferring models from one tool to another may lead to very unexpected results.

Poor interoperability between different tools is a problem related to toolset immaturity. The model interchange format XMI does specify how to define elements on the abstract level, but e.g. diagram layout is still being implemented via vendor-specific extensions [SB05]. Currently, tool interoperability is being built as point-to-point connections, e.g. by building a bridge from Eclipse to Microsoft modeling tools [BCC+10].

Other problems rise when trying to find the correct level of abstraction. For example, not all semantic connections in the source models are suitable for automatic transformations [BCT05]. Lack of automatic transformations is a big problem, since the initial promise of improved productivity is built on top of automation shoulders. Some researchers overcome this limitation by extending the base programming language to better accommodate for model-code interaction [SKRS05].

Another problem identified in the literature is the change in application structure: with the introduction of models, model transformations and code generation, the application logic is scattered to various places in the architecture [SB05]. This is argued to hinder general understandability of

the system, and thus to hinder maintenance.

Yet another problem is that given the current fast rate of change in technology choices and architectural evolution in software engineering, the model transformations provided by the chosen toolset probably do not match the current architectural needs of the developed software [SB05]. When this occurs, the development team has two choices: try to find an alternative, better suiting toolset or try to improve the existing toolset. The first option basically stalls development work, as the focus has changed to finding the right tool for the job instead of actually doing the job. The second alternative, if viable at all due to copyright reasons, requires specialized personnel who have the ability to modify the transformations used by the toolset. Since the development of the actual software cannot be delayed, the software's architecture evolves in parallel to transformation development. This reason gives a good chance that any given set of model transformations is already obsolete at its completion time.

It is also noted that in practice model-to-model mappings are complex and require careful design and implementation [BCT05]. While all software engineering needs careful design and implementation, it seems to be an even more relevant problem in the context of model-driven software engineering.

In conclusion, given these reasons, unconstrained usage of model-driven architecture cannot be considered to be a good match for current agile development environments. However, we do not propose to canonically reject software development based on model-driven architecture. Our critique primarily bases on the combination of short-lived sprints of agile development and the uncertainty of toolsets and practices promised by MDA tool vendors. In cases where a toolset's abilities and limits are well known in advance, using the toolset-driven approach can be beneficial even in tightly time-framed situations.

Support for evolution in model-driven engineering

Support for evolution is often a recurring question in different flavors of model-driven engineering (MDE). Model-driven engineering is a larger concept than the model-driven architecture discussed in the previous chapter: MDA refers to the reference implementation that is trademarked by Object Management Group. Model-driven engineering refers to the general idea of using higher level models to drive software engineering.

The first rule in the MDE context is that developers are not supposed to modify the artifacts generated from models [SB05]. This is conceptually not a new idea, since in the classical edit - compile - run development cycle, engineers are not supposed to manually modify the assembler code

generated by the compiler. However, since some model transformations generate high-level programming language code, it seems to raise some confusion about the role of the generated artifacts.

Why would there be need to manually modify generated code in a software project? The reasons can be manyfold. Examples include the lack of semantical expressiveness in the source modeling language; non-optimizing transformations that produce sub-optimal target code with the consequence of performance problems; or just plain lack of required expertise in the project personnel.

For any tools that are targeted to real-world use, the need to support software evolution is not a new requirement. However, tools support this need using different approaches. For example, the AndroMDA tool emphasizes the use of subclassing for modifications [SB05]. This approach follows the "Generation Gap" pattern [Vli98, p. 85-101]. In some cases, this can be adequate, but a number of cases can be identified where inheritance is not good enough. For example, a study on scalable extensibility presents cases where inheritance is not a good choice for extending a set of classes when the corresponding objects interact in a certain way [NCM04]. Other experts complain about the lack of inheritance expressiveness [Lie86] and about the potential misuse of inheritance hierarchies [Tai96]. Also, it can be problematic to use inheritance when reusing and extending interdependent classes. It can be argued that the class hierarchies become unnecessarily big if all the generated classes have automatically generated base class, and the corresponding editable subclass.

In Paper (II) we describe an extension of the Visitor pattern [GHJV95, p. 331-344] for a set of generated classes. The Walkabout pattern [PJ98] uses reflection facilities to adjust the orchestration of visited nodes. We use its more efficient version called Runabout [Gro03] to generate a corresponding Visitor base class hierarchy for the corresponding model, and then rely on the type checking rules in the implementation language for finding inconsistencies introduced by evolutionary changes in the system.

Although the principle of not manually modifying generated code is well accepted in general, many researchers in the model-driven engineering community fail to take evolution into account. For example, a recent study describes a system for generating web applications based on a self-grown modeling language [BLD11]. The description does not support further evolution of the system: any changes to the system require a full re-generation of the produced software. As such, the approach can be classified to fall into the translationist category of the two model-driven engineering approaches. However, combined with the lack of expressiveness in the source modeling

language, the approach runs into problems when using it for any serious software engineering project.

3.3 Modeling in agile software process

Agile process improvement literature focuses mainly on process-level practices. Less emphasis is given to the actual software structures that can be designed in an agile process.

We argue that the classical model-driven architecture's approach is not very suitable for agile processes due to its heavy emphasis on tools and model transformations. Then we argue that agile model-driven development cannot be justified from productivity angle, as the lack of formality in agile models prevents the usage of automated handling. Domain-specific modeling is seen as a good trade-off between formality and agility, but it is still staying short of good agility due to its emphasis on specific tool usage.

Agile Model-Driven Development

Agile model-driven development [Amb04b] attacks the problems in model-driven architecture by relaxing the strong requirements on formality and tool support. Instead of using complex and extensive models, the approach emphasizes models that are *barely good enough* [Amb04a] for a given task. Modeling is mostly done top-down, although the approach does not exclude a bottom-up approach.

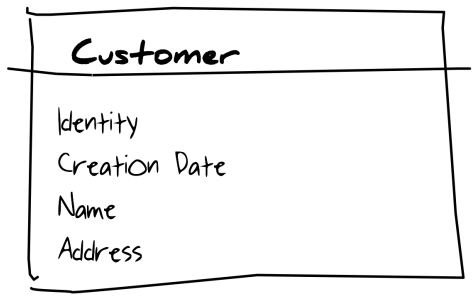

Figure 3.3: A hand-drawn sketch in agile UML modeling

Figure 3.3 shows an example of a sketch used in agile modeling. Although the attribute names are identical to the platform-specifig model shown in Figure 3.2, there is subtle difference in this modeling approach when compared to the previous example. Although the figure has been produced using a software drawing tool, the emulated hand-drawn style

suggests that the intent of the model is to enable communication and collaboration. This is in contrast to the technical representation used in the previous example.

According to this philosophy, created models should not affect the agility principles of a given process. As long as a model can be created and exploited within a production cycle (usually 1-3 weeks), it is suitable for agile modeling. This is a promising approach, but it does not state much about the possible modeling tools - actually, the approach de-emphasizes the need for tools, and focuses on people.

Due to this requirement, most modeling is based on high-level abstract modeling languages with little formalism. They are easy to learn, simple to use and fit well within a given time period. However, they offer more to easier problem domain abstraction than to automatic productivity increase. To our best knowledge, no productivity-related empirical validation has been done for agile model-driven development.

Model-driven architecture is a top-down approach to software engineering. Due to its heavy emphasis on tools and process, the approach is not a good fit for agile time-boxed iterations. Light-weight models are often a better alternative in agile development. Boxes and arrows on a whiteboard is a good start. However, we argue that in order to realize productivity gains, these models can and should be brought into software architecture level entities.

3.4 Software product lines

One answer to demands for easier utilization of software assets in reuse is the usage of software product lines. In a software product line, the idea is to organize software development into components that can be combined in defined, different ways. Domain analysis identifies core assets of a product line, which form a software platform. When needed, product-specific components can be included into a single product [LSR07, p. 6-8].

The aim of a software product line is to be able to support a number of different software products that share certain commonality between them. Typical examples of software product lines have emerged in areas of embedded software, such as cell phones [LSR07, Chapter 12] or avionics software [DBT11].

The starting point for defining the ways of possible combinations is the modeling of variability. Many variability modeling formalisms are built on top of feature-oriented domain analysis, FODA [KCH+90].

Modeling variability

Feature modeling [KCH+90] is a formalized way of building option spaces. The formalism allows to define structures with mandatory features, optional features and combinations of them.

A canonical example of using feature modeling was presented in [CE00]. The example elaborates on all basic operators in basic feature modeling. The model defines an option space for a car, which is shown in Figure 3.4. The car needs to have the body, the transmission, and the engine. The transmission can alternatively be automatic or manual. The engine can be electric or gasoline driven, or both. Optionally, there can be a trailer pulling hook. The original explanation of the formalism is mathematically rigorous. Fortunately, other introductory texts such as [AK09] kindly remind readers of the practical implications of the constructs, by associating the words *xor* and *or* to the optionality operators, as shown in the figure.

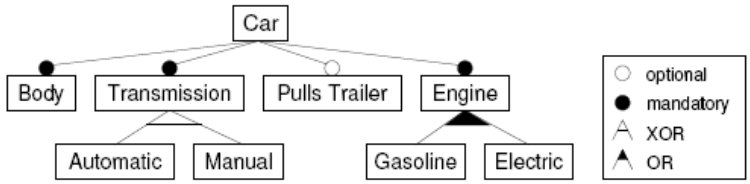

Figure 3.4: A feature model of a car [AK09]

Given this option space, the following 12 distinguishing configurations satisfy the model:

1.	Transmission: Automatic	Engine: Gas	
2.	Transmission: Automatic	Engine: Electric	
3.	Transmission: Automatic	Engine: Gas+Electric	
4.	Transmission: Automatic	Engine: Gas	Pulls Trailer
5.	Transmission: Automatic	Engine: Electric	Pulls Trailer
6.	Transmission: Automatic	Engine: Gas+Electric	Pulls Trailer
7.	Transmission: Manual	Engine: Gas	
8.	Transmission: Manual	Engine: Electric	
9.	Transmission: Manual	Engine: Gas+Electric	
10.	Transmission: Manual	Engine: Gas	Pulls Trailer
11.	Transmission: Manual	Engine: Electric	Pulls Trailer
12.	Transmission: Manual	Engine: Gas+Electric	Pulls Trailer

More advanced feature modeling variations allow the use of cardinality constraints, cross-tree constraints [Bat05], default values [SBKM09] and

other extensions.

In the example of Figure 3.4, a natural cross-tree constraint could be that when the trailer pulling option is selected, there needs to be a gas engine - a pure electric engine would not suffice for the added load of pulling the trailer. This could be expressed as *Pulls Trailer implies Gasoline Engine*. This constraint would rule out the configurations #5 and #11, thus reducing the number of valid configurations to ten.

The usage of default values allows the model to be augmented with guides on which values should be chosen if no additional information is given. In the example, the default values could be *Manual Transmission* and *Gasoline Engine*, which would result the configuration #7 to act as the starting point of engineering.

The example in Figure 3.4 represents a case where the software's data is modeled using feature modeling. However, the internal features of a software can also be modeled with the same mechanism. For example, the Linux kernel contains a domain-specific language, *Kconfig*, for configuring the features to be used. An example of the Kconfig model is given in Figure 3.5.

The excerpt of the actual kernel file *linux-3.6/arch/mips/Kconfig* is used to configure the kernels targeted to the MIPS processor architecture. The default target is the IP22 prompt, which was used in the once-popular SGI Indy machines. The alternative in the figure is IP27, which was used in a later models of Origin 200 and Origin 2000.

This small fragment of the feature model shows how the Kconfig language can be used to express alternation (choice), defaults (default), data types (bool), and cross-tree dependencies (depends) when configuring the kernel options to be used. The expressive power of the language provides for feature models with cross-tree constraints and default values [SLB+10]. However, as models in the language are involved with end-user interactive configuration software, it also contains instructions for displaying user help texts and other usability aiding tools.

To help in seeing what kind of support variability modeling can bring for software developers, researchers have started collecting an on-line repository of published variability models [MBC09]. This repository contains at the time of writing a collection of over 300 known, published models of variability in different domains and can be used to get a concrete feeling of how software variability modeling is currently being done.

```
config MIPS
choice
    prompt "System type"
    default SGI_IP22
    config SGI_IP22
        bool "SGI IP22 (Indy/Indigo2)"
        select BOOT_ELF32
        select SYS_HAS_CPU_R5000
        select DMA_NONCOHERENT
        help
          This are the SGI Indy, Challenge S and Indigo2.
    config SGI_IP27
        bool "SGI IP27 (Origin200/2000)"
        select BOOT_ELF64
        select SYS_HAS_CPU_R10000
        select DMA_COHERENT
        help
          This are the SGI Origin 200, Origin 2000 and
          Onyx 2 Graphics workstations.
endchoice
config CPU_R5000
        bool "R5000"
        depends on SYS_HAS_CPU_R5000
        select CPU_SUPPORTS_32BIT_KERNEL
        select CPU_SUPPORTS_64BIT_KERNEL
        help
          MIPS Technologies R5000-series processors.
 config CPU_R10000
        bool "R10000"
        depends on SYS_HAS_CPU_R10000
        select CPU_HAS_PREFETCH
        select CPU_SUPPORTS_32BIT_KERNEL
        select CPU_SUPPORTS_64BIT_KERNEL
        select CPU_SUPPORTS_HIGHMEM
        help
          MIPS Technologies R10000-series processors.
```

Figure 3.5: An except from the Linux kernel configuration at linux-3.6/arch/mips/Kconfig

Implementing variability

Implementing variability in a software product or system has many alternatives. However, before discussing about variability implementation mechanisms, the target of variability needs to be clarified. The model shown in Figure 3.4 represents a domain-analysis part of feature modeling: it could be contained for instance in a software system that manages purchase orders to a car factory. Another way of using feature modeling is to model the software structure, as is done in Figure 3.5.

In Paper (I) we present a case where software variability for different platforms is implemented by the means of object-oriented interfaces. In Paper (III) the discussion targets to implementing domain-analysis part of the model.

On software architecture level, the variability can be implemented with a number of techniques. Alternatives for implementation include at least the following:

- conditional compilation
- object-oriented composition
- aspect-oriented composition
- external rules engines
- generative scripting

In the following, we will shortly review these techniques.

Conditional compilation

Conditional compilation, also known as *#ifdef*s, is a widely used implementation mechanism of variability. In conditional compilation, the source code contains various preprocessor directives that are used to select out some parts of the unpreprocessed source. The variability-processed source is then fed to the regular compiler.

In the Kconfig context, this works as follows. Each configuration directive in the Kconfig files is exposed to the preprocessor by setting the CONFIG_- variables where the name of the configuration directive is appended to the preprocessor variable name. e.g. CONFIG_DMA_NONCOHERENT. Not all configuration directives are used in the source code, but some of them act as purely variables for guiding the configurational structure. For example, the SYS_HAS_CPU_R5000 preprocessor variable does not appear anywhere in the source code.

For example in the Linux kernel 3.6, a part of the file *arch/mips/mm/c-4k.c* reads as shown in Figure 3.6.

```
void __cpuinit r4k_cache_init(void)
{
[..]
#if defined(CONFIG_DMA_NONCOHERENT)
   if (coherentio) {
        _dma_cache_wback_inv = (void *)cache_noop;
        _dma_cache_wback = (void *)cache_noop;
        _dma_cache_inv = (void *)cache_noop;
    } else {
        _dma_cache_wback_inv = r4k_dma_cache_wback_inv;
        _dma_cache_wback = r4k_dma_cache_wback_inv;
        _dma_cache_inv = r4k_dma_cache_inv;
    }
#endif
    build_clear_page();
    build_copy_page();
    coherency_setup();
}
```

Figure 3.6: Excerpt from the Linux kernel source file arch/mips/mm/c-4k.c

In the excerpt, the DMA_NONCOHERENT selection in the configurator tool is shown as a compiler preprocessor directive variable. Choosing the R5000 processor implies the DMA_NONCOHERENT selection to be chosen as well, which leads to inclusion of the #ifdef part in the actual compilation cycle. In the case of the R10000 processor, this part is left out.

Using preprocessor directives has been the way to do parameterized compilation long before software product lines were invented. However, since conditional compilation is well supported in toolchains and the concept is well understood among software developers, it is an often used technique for implementing software product lines as well.

On the other hand, the simultaneous handling of configuration time and compilation time constructs hinders code understandability. Labeled as *#ifdef hell* [LST+06], software developed to be parameterized through conditional compilation is known to be harder to understand than software that does not use it. If not attributed by using other implementation mechanisms, the problem can be relaxed e.g. by improving tool support

[FKA+12].

Object-oriented composition

In object-oriented composition, interfaces are a natural point of variation. An interface class represents a variation point, and variability is implemented by changing the concrete implementation of the interface.

In paper (I) we demonstrated and empirically tested an optimizing compiler that works on a software product line using interfaces and abstract classes as the means to differentiate between mobile phone models. In the research setting, the overhead introduced by object-oriented constructs was feared to become preventive for employing good software architecture principles with suitable abstractions.

Some researchers believe that this kind of support for program evolution cannot be supported in resource-constrained environments [ZHKH07, ZHK07]. However, this view is based on incomplete understanding of the application building process: variability that is resolved before deployment time does not bring any runtime overhead.

Aspect-oriented composition

In aspect-oriented programming [KLM+97], the base software augmented with optional aspects. Specific weaving expressions that define the joining points are used to guide where the optional aspects should be inserted at compilation time or at runtime.

These join points can be used as the variation points: when an optional feature should be present by the product line configuration, the corresponding aspect is woven to the base software. Examples of this kind of implementation can be found at [AM04] and [GV09].

External rules engines

In some cases, the software product line is not implemented in the core software's architecture at all. Instead, the variation is outsourced to an external rule engine: all variation is expressed by changing the rules in the engine.

This approach has the benefit of moderating the changes to software architecture required by different customers and variations of the software. All the changes are encapsulated to their own environment.

The drawback in this approach is that the degree of variation is limited: the number of variation points is limited to the points where the base software transfers control to the external engine. Variation obviously is not

a first class entity in the architecture. Also, the question of how well the external engine supports variation is not answered in this approach: should the rules in the engine follow some variation approach presented in this chapter, or is all the variation implemented using ad hoc implementation techniques?

Generative scripting

Generative scripting is another approach to implementing product lines. In this approach, the idea is to use a higher level model as the product line definition. Scripts are then used to generate corresponding lower-level code, very much in the style of model-driven engineering presented in Section 3.2. Much of the discussion from that section applies to using models to drive software product line engineering as well.

An example of this kind of architecture is the generation of the NH90 helicopter product line, which has customers in several countries and military-grade restrictions in letting one country's version to interact with another country's version [DBT11]. In this environment, the big boost for productivity is the standard certification requirement in the avionics software development. In avionics software development, the created software needs to be certified by aviation authorities before it can be included in a type-certified aircraft. Standard object-oriented techniques, such as dynamic dispatching based on late binding, are known to be problematic to get certified due to their unknown runtime properties. In the NH90's case, the developers were able to certify the code generating Perl script as a way to do static dispatching at development-time binding.

3.5 Self-organizing and self-configuring software architectures

Both model-driven engineering and software product lines are approaches to resolve problems introduced by software variation. Variability can happen over time – i.e. software evolution – and over different contexts, such as variations tailored to different customers or product models. These techniques can help to manage the complexity introduced by variability, but many open problems still reside.

Self-organization is a way to reduce or minimize the need of explicitly managing the organized elements. For software architectures, self-organization has been applied in context of dynamic component discovery [MK96] and distributed systems development [GMK02] while ensuring

overall system structure consistency.

In this thesis, we propose that using self-organizing software architectures is a way to improve productivity and maintainability. These goals are even more important when addressing problems in several different contexts. For example, feature interaction is a core problem in software product lines: how to understand program behaviour in all possible configurations in different contexts and how to make sure that the chosen features do not interact in a wrong way.

However, using self-organizing software architectures is not limited to software product lines. They can be used, in general, as a simplifying technique for the architecture. Since the self-configuring components can be used to remove or reduce duplication, the architecture of a software becomes simpler. Before giving examples of how this is done, we will review what is meant by self-configuring software components and self-organizing software architectures.

Various kinds of self-something properties have been described for software systems. For a compact review, a dissection of the different self-* properties can be found in [BG09]. For our discussion, the definitions for self-organizing and self-configuring are the most interesting ones.

For self-organizing systems, a few different definitions exist. Anderson et al. define self-organization to require the system to maintain, or monotonically improve a function value involving the neighbors of the joining or the departing process in-between process joins and leaves [ADGR05]. Associated properties of self-organization are that the self-organization should happen in sublinear time and process joins and leaves should cause only local changes [DT09].

Now the question is that how does this relate to software architecture? Some parts of distributed software are targeted to handling process joins and leaves, but certainly it is in the core of software architecture only in few of the above mentioned definitions. To transfer these process-centered definitions, we should focus on the concepts in the software architecture: components, structure, design, constraints, properties and their relationships. For an architecture to be self-organizational, there should be a mechanism for maintaining or monotonically improving the software when new architectural elements are added, or old ones are removed or modified.

Self-configuration of a software system, in turn, refers to the ability to "change its configuration to restore or improve some [safety] property defined over configuration space" [BG09]. When comparing self-configuration and self-organization, the former requires the system to change its configuration when restoring, maintaining or improving some property of the

system after a change. The latter handles joining or removal of architectural elements. Thus, it follows that every self-organizing system is self-configuring, but the reverse is not true [BG09].

When discussing the joining and leaving of architectural elements, we need to think about the binding time of decisions. A usual division of binding times is to talk about design-time, coding-time, compilation-time, deployment-time and runtime decisions. For example, overall architectural design is clearly a design-time decision while the dynamic binding of method calls in an object-oriented language is a runtime decision. Choosing the database connection parameters is a typical deployment-time decision and tuning of the database engine to perform with a given workload is often runtime work.

Another dimension to the discussion is the amount of human interaction needed for making these decisions: fully automated decisions can be made without human intervention – but often, some kind of human interaction is needed. For example, in the world of refactorings [Opd92] we can see a continuum from manual refactorings to automated refactorings [KEGN01]. To be able to safely make automated decisions, they need to be formally specified: under certain conditions, a certain refactoring can be made. Stricter formality requirements lead to limited applicability, and often to a local scope. On the other hand, the amount of possible collateral damage is also limited. In manual refactorings, there is high possibility for collateral damage, but they can be widely applied and the human operator (the programmer) can use human judgement to make global changes. Often a high cost is involved due to high wages of programmers, at least when compared to electricity price. When the refactoring steps cannot be specified with enough precision, maintenance patterns [HH04] can be used to give the maintenance programmer some guidance on the workings of the system.

Figure 3.7 shows a few approaches on formality and binding time axis. Self-organizing software architectures are such software architectures that can adjust their functionality to changes in the architecture, such as adding or removal of components in the software. By definition, self-organization happens at late binding time: adjustments to the architecture at design and coding time are refactorings rather than examples of self-organization.

Application-level self-organization can be implemented by using self-configuring components. A self-configuring component contains program logic for adjusting the functionality of the component according to its local environment. In Paper (V) we show an example of a self-configuring component of database queries. Without such self-configuring component, the

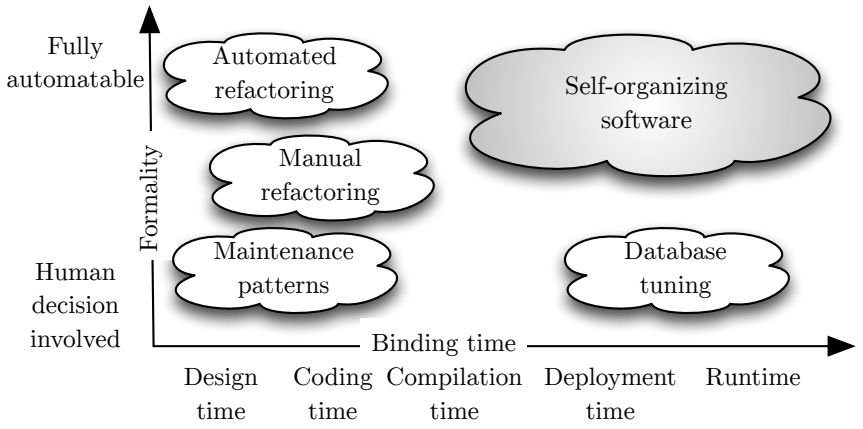

Figure 3.7: Chart of formality and binding time applicability of different modifications

database querying layer would need its own components for all database query contexts. These can be replaced by one self-configuring component which adjusts its functionality according to each context, thus simplifying the architecture.

These kinds of improved modularity and maintainability properties are useful in all software engineering, but they are especially useful when the resulting software experiences regular modifications, e.g. as is done in model-driven engineering when changing the model and in software product lines when deriving different variations out of the product line.

When designing a self-configuring component, the implementation needs reflectional access (self-description) to the context that the component is adjusting itself to. When the access is implemented via source code or other low-level access mechanisms, it can become cumbersome to design efficient self-configuration. If the accessed software exposes its functionality in higher level models, this task can become easier. In the next chapter we will explore bottom-up modeling, which can be used to overcome an often experienced conflict between high-level models and low-level implementation details.

Chapter 4

Bottom-up modeling

Model-driven architecture in its various forms gives a principle for increasing software development productivity by raising the level of abstraction. Model-driven development concentrates on models of the domain, thus allowing developers to concentrate on the actual domain instead of fighting against the compiler on the programming language level. However, traditional model-driven approaches are rather heavy-weight, as they require intensive support from various tools before meta and meta- meta-level constructs are defined precisely enough to be used. The end result appeals to architectural enthusiasts, also known as architectural astronauts [Spo04, pp. 111-114], but is seldom good enough for real software development.

The following sections are organized as follows. Section 4.1 briefly discusses why top-down modeling is problematic, especially in agile context. Section 4.2 represents the alternative: modeling in a bottom-up way. Section 4.3 explains the bottom-up modeling concept with an example in context of building a data model for general aviation maintenance application. Section 4.4 discusses when the bottom-up modeling approach is applicable in software development. Section 4.5 reviews related work.

4.1 Problems of top-down model engineering

When starting modeling in a top-down fashion, it is often unclear whether the chosen modeling formalism can handle the task. If the formalism is too weak, modelers cannot express their intentions well enough. For example, in a critique of MDA, researchers found out that plain UML is not expressive enough for software development. Tools use vendor-specific profiles, which limits interoperability and can cause vendor lock-in [SB05]. On the other hand, if the formalism is too strong, it is probably also too complex. This

steepens the learning curve and makes modeling less lucrative.

The top-down approach assumes a complete and correct understanding of the system being developed. This is seldom the case in software development. An early advice for software engineering once stated that *build one to throw away* [Bro95]. Following this route can help in building understanding of what is needed to build the system. Unfortunately, with the current demands of faster time-to-market, only few teams can afford to spend extra time in the beginning to build something that is going to be thrown away anyway.

Another problem is that with the top-down approach, the referred designs are treated as 'black boxes' that are to be further specified and developed in the following step-wise refinements. The black boxes may fail to clarify the internal workings of fundamental elements in the lower abstraction level [EHS10, p. 34]. For an uninitiated designer, the seemingly simple structure can then turn out to be unimplementable at the lower level.

4.2 Bottom-up model-driven engineering

To overcome difficulties with the top-down approach, it is often possible to change to a bottom-up approach. Traditional model-driven engineering is essentially a top-down method, which recursively decomposes the problem to smaller pieces, until any given piece is small enough to be solved with available tools. The alternative to top-down methods is to use a bottom-up approach which identifies smaller sub problems and develops solutions to these. When this cycle is repeated, gradually the solution for the whole emerges. Given a problem domain, the bottom-up approach identifies subdomains that are amenable to modeling.

Bottom-up model driven development [BS13] recognizes the problems of working with abstract artifacts that hide the concrete results of the modeling work. They propose to change the point of view from fully synthesizing model-driven software development to use partial models and partial synthesis. Where-ever a model is recognized to be useful, based on understanding gained from working with concrete code elements, higher level models can be developed. Since the models do not comprise the whole software, the generated code is also based on partial synthesis.

Agile bottom-up modeling constrains the model identification process to such tools and techniques that can be applied in an agile process model. The application of bottom-up modeling is thereby limited to a small number of tools, which can be evaluated and applied within a tightly time-boxed iteration. Yet, although this search-and-discover approach theoretically

produces non-optimal solutions, it guarantees that progress is not stalled while searching for the optimal solution. This way, a bottom-up approach to modeling can avoid the heavy up-front planning phase associated with traditional model-driven engineering. The need to build complex modeling languages with associated tool support can be avoided or the impact of the tool-building effort can be damped to be included within the time-boxed sprints. To further help, existing languages can be reused or a domain-specific modeling tool can be used. The obvious downside of the approach is that repetitive application of ad-hoc modeling constructs might gradually erode the software's overall architecture.

The main message with bottom-up modeling is to constantly look for sub-domains that are amenable to lightweight formalization. Crafting meta-models and tools for such a small domain is definitely easier than building an all-inclusive supermodel of the whole world, which is argued to be needed in the case of top-down model-driven engineering [BJV04].

We argue that productivity gains chased with model-driven engineering should be combined with agile development models by examining the productivity problems that are encountered on project level. If a given productivity problem gives a feeling that its root cause is associated with the problem of lack of abstraction, or incorrect level of abstraction, then that particular part of the software could be a possible candidate for building a higher level model.

We call this approach as bottom-up agile model-driven development (BUAMDD). In this approach bottom-up models are a way to introduce light-weight (possibly formal) modeling to an agile development process.

Traditional model building deals with object-oriented models, represented in the source code using the implementation language's abstraction mechanisms. The Model-View-Controller [Ree79] design pattern, maybe decorated with the Fluent Interfaces pattern [Fow10], is a classical example of this approach. The problem with object-oriented modeling resides in the lack of increased abstraction: the level of handling is essentially the same as the implementation language's level of abstraction – and breaking the productivity barriers induced lack of abstraction is the primary reason for applying formalized modeling.

An essential requirement of bottom-up modeling in an agile project is that the building of model languages and models can be decomposed into sprintable form. We mention both model languages and models because the essence of bottom-up modeling is to find suitable abstractions to the problem at hand, and often this means inventing a new language or reusing an existing language for modeling. This notion is contradictory to the com-

mon wisdom of using the best existing tool for the job at hand. However, given the large number of different tools and techniques available on the market, it is not possible to carry out a thorough tool evaluation within the time frame of an iteration. For this reason, agile teams often need to build their own abstractions for modeling.

These abstractions or languages are not necessarily complex, meaning that there is no mandatory need to building complex modeling languages with associated tool support. Instead, existing languages can be piggybacked and reused as is common with domain-specific languages [MHS05]. Preliminary research shows that this can be the most efficient way to implement domain-specific languages [KMLBM08], meaning that it can be a good fit in an agile environment. Also a domain-specific modeling tool can be employed, once an initial understanding of the problem at hand has emerged.

Bottom-up modeling does not limit the format of source models, as long as they are expressible in machine-readable form. This means that the modeling language does not need to be a graphical boxes-and-arrows -type tool. Actually, although the traditional boxes-and-arrows kind of modeling can be beneficial in the drafting board, the lack of exact interpretation for the used symbols hinders productivity when forwarding these models to any type of automatic code generation or runtime interpretation.

External domain-specific languages are often used, but an interesting alternative is to use the source code as the source model. In practice, this option is viable because using the source code as the source model for further transformations increases robustness against modifications into the software. This option is investigated further in Chapter 5.

4.3 An example

In order to illustrate the idea of using the bottom-up approach to modeling, we build an example case of a general aviation operations software. In this example, we have a hypothetical aircraft operator who wants to improve its business efficiency by building an operation supporting software suite. The software suite handles various aircraft of different type that are being operated by this business. In the first iteration, we build the data model for the aircraft database. In a subsequent iteration we then build a new module for handling different configurations of individual aircraft.

Although this section carries on with the motivational example, we refer to essential literature along the discussion.

Data modeling

Data modeling is often at the very core of enterprise applications. Data is stored into a database, and is retrieved from there for viewing and further modification. With this background, it is not surprising that class diagram is the most often used diagram type when modeling data with UML [DP06, FGDTS06].

Our approach to this problem is to exploit the nature of bottom-up problem solving. Re-using the notion of piggy-backing existing languages, we can choose to use e.g. XML schema [TBMM04] for data modeling. For example, the XML schema definition for the data model of an engine type that we are interested in could be as shown in Figure 4.1. This presentation is prettified: the actual syntax is not this clean. In practice, the IDE handles a lot of the inconvience presented by the cluttered syntax.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:dsm="http://acmebar.com/DomainSpecificSchema">
  <xs:annotation>
      <dsm:engine name="O-320" />
  </xs:annotation>
  <xs:element name="OverhaulInterval" type="xs:duration"
                  fixed="P2000H"/>
  <xs:element name="LastOverhaulTime" type="xs:date"/>
  <xs:element name="CurrentRunningHours" type="xs:integer"/>
</xs:schema>
```

Figure 4.1: Data model for example engine O-320 presented in the XML Schema language

Here we define an engine model O-320, which is the normally aspirated, air-cooled, carbureted four-cylinder boxer engine manufactured by Lycoming. As a link to the surrounding software, we annotate the schema file with our own, domain-specific schema annotation of *dsm:engine*. The standard data modeling part is done in the XML Schema namespace *xs*. The engine overhaul interval is defined to be a duration of 2000 hours in the field *OverhaulInterval* using the notation *P2000H*, meaning a period of 2000 hours, as specified in the XML Schema language. Finally, the two fields available to engine instances are *LastOverhaulTime* and *CurrentRunningHours*, for recording the time of the last overhaul and the number of flown hours since the last overhaul.

In top-down engineering, using a predefined data modeling language has the consequence of a need to implement corresponding support for the language at the implementation level. For example, UML does not provide a concise way of defining the simple options modeled in the XML schema document above. By using an existing language, with known implementations, the implementation part is easier since the modeling language already defines the available data types [BM04].

With this approach, the data modeling is done by defining XML Schema models. The expressive power of the schema language greatly overpasses the one offered by standard UML class diagrams [MNSB06]. Another benefit is that standard XML tools can be used to validate data transmissions. The semantics of the XML Schema language is well understood. As a formal language, the source documents can also be used for further model transformations; the schema language contains a well-defined variation point holder for defining new features for the data modeling tool.

Similarly to traditional model-driven engineering, the data model can be a subject to model transformations, for example to define the corresponding database creation clauses as shown in Figure 4.2.

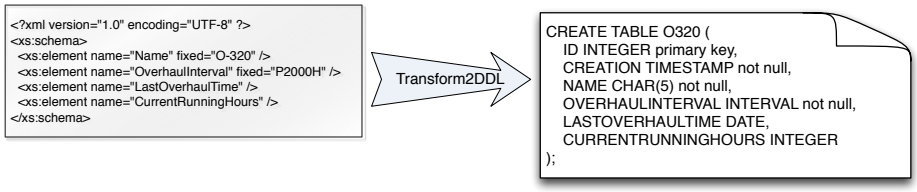

Figure 4.2: Transformation from XML schema to SQL DDL

However, the use of the model is not limited to development time transformations. Since the model is expressed in programmatically accessible form, it can also be used to drive e.g. user interface generation and input form validation at runtime.

In Section 3.2 we argued that the limited extensibility of UML class diagrams is a problem for developing domain-specific semantics. In this approach, this limitation can be relieved. The XML Schema language contains a specifically designed variation point place in the annotation field. This can be used to elaborate the model with arbitrary extensions, based on domain-specific needs.

However, the approach is not without downsides either. We argue that the predefined set of data types is good for implementation. Now the question is what if these data types are not good enough for the given need?

At that point the implementation runs into the same problem as when using the top-down approach; the situation needs to be handled separately. Fortunately, given the stronger expressiveness, this case does not happen as often and therefore is less problematic in practice.

Feature modeling

For our discussion, the interest lies in how to implement functionality to handle these kinds of models. In model-driven architecture, the standard approach is to use transformations to bring the source model into the streamline of standard modeling languages. Thus, a transformation for translating the feature model into an UML model is needed. Literature presents various ways for doing this. Several researchers [CJ01, Gom04, GFd98, CCC+11] have presented different flavours of using stereotypes for representing variability in UML. [VS06] suggests that changes in the UML metamodel are needed to fully support variability.

These studies contain many fine points for implementing beautiful models of variability using the standard technologies. However, for practical software development cases, variability is just one of the dozens, hundreds or thousands issues that a development team needs to tackle. It can be impractical to start discussing about the academically correct way of implementing this variability, since it can be hard to demonstrate how this discussion brings value to the end customer. Due to the economic reasons discussed in Section 2.5, it probably never will.

An alternative is to work with this specific problem, using the standard tools offered by the implementation environment. For the variability example, in Paper (III) we have documented a way of using standard regular expressions for modeling variability. This approach tries to combine good parts from both formal modeling and agile product development. The approach has the benefit that the customer can be shown steady progress, since modeling is concentrated on small subdomains. On the other hand, since the models are implemented using the standard implementation environment structures, the mismatch between modeling environments and implementation is kept at minimum.

In our example, next we define the semantics of the software concept *aircraft*. As an implementation vehicle, we use regular expressions to define the variability model as shown in Figure 4.3. In this definition, we define what different parts can form an airplane. The definition reads as follows:

- The fuselage can be either pressurized or unpressurized.
- There are always two wings, the empennage and the tail.

- The propeller is either a fixed pitch or variable pitch.
- The engine type is one of: O-320, TSIO-360 or TAE-125.
- There can be up to 3 radio (COM) units, each with the optional VHF navigation (NAV) ability.
- There can be navigation equipment group for automatic direction finder (ADF), distance measuring equipment (DME), VHF omnidirectional range (VOR) and global positioning system (GPS).
- The avionics is either the traditional six-pack avionics or a glass-panel G1000 model.
- The landing gear can be either conventional or tricycle.
- The landing gear can optionally be a retractable undercarriage.

```
aircraft consists of:
  (pressurized | unpressurized) fuselage
  left wing right wing empennage tail
  (fixed pitch propeller | variable pitch propeller)
  (O-320 | TSIO-360 | TAE-125) engine
  (no | (COM NAV?){0-3}) radio
  (no | (ADF | DME | VOR | GPS)+) navigation
  (G1000 | traditional) avionics
  (conventional | tricycle) gear
  (retractable undercarriage)?
```

Figure 4.3: Variability model of aircraft

In the case of radio and navigation equipment, there is the option of the aircraft not having any of these: in that case the 'no radio' and 'no navigation' options are chosen. Each of these elements can contain the corresponding data model definition. For example, the data model definition for the engine O-320 was given in Figure 4.1.

This is a very concise way of using higher level abstraction to bring benefits of modeling into implementation level. For example, if we want to validate a given aircraft configuration, we can feed the proposed configuration to a regular expression matcher, which is available in all modern programming languages.

However, using this kind of modeling language specific translation to the implementation language raises some questions. For example, many

feature modeling formalisms allow the definition of cross-tree constraints. In the example model, we could not specify the linkage between a VHF omnidirectional range equipment and a navigational radio. In a general aviation aircraft, the VOR display receives navigational signals from the NAV radio. If the aircraft does not contain a NAV radio, the VOR display cannot function either. When using regular expressions to represent feature models, these kind of cross-tree constraints are problematic. Many programming language implementations offer the use of back references that can be used up to some degree. Another option is to implement the cross-tree constraints in general programming language code. One of the main ideas in BUAMDD is to iteratively find the optimal, desired balance between the models and the code.

Using UML models gives the possibility to scale the scope of modeling to include also the attributes of modeled entities, for example, if there is a need to specify the size or power of the engine being modeled. In a class diagram, it is very straightforward to add the attribute in question to the class model. But using regular expressions to model even five to ten different engines would soon prove to be cumbersome. However, this does not mean that we could not extend our little language to a bigger one. Quite the contrary: when we identify the need for such a semantic extension to regular expressions, we can choose to adopt e.g. attribute grammars [Paa95] to handle our increased needs.

How in practice that extension would be implemented depends on the environment and individual preferences of the designer. In Section 3.2 we discussed the difference between internal and external modeling. Using an external modeling with a dedicated generator/extension suite, such as JastAdd [EH07] could be a good choice in a Java-based environment. If the operating environment happens to have a good support for internal implementation of attribute grammars, such as the Kiama library in a Scala environment [SKV11], the lightweightness of a pure embedding could work well in practice.

Emerging modeling support

This way, bottom-up modeling can use a mix-and-match approach for selecting the suitable tools for situations rising in development projects. In the example in this section, the developers chose to use regular expressions for modeling variability and the XML schema for data modeling. Combining these two allowed the developers to use models as first-class artefacts in their product, since both of the used modeling languages were supported in the programming environment. Equally important, the developers were

able to show steady progress towards the customer, since there were no delays due to the team doing tool evaluations and comparisons.

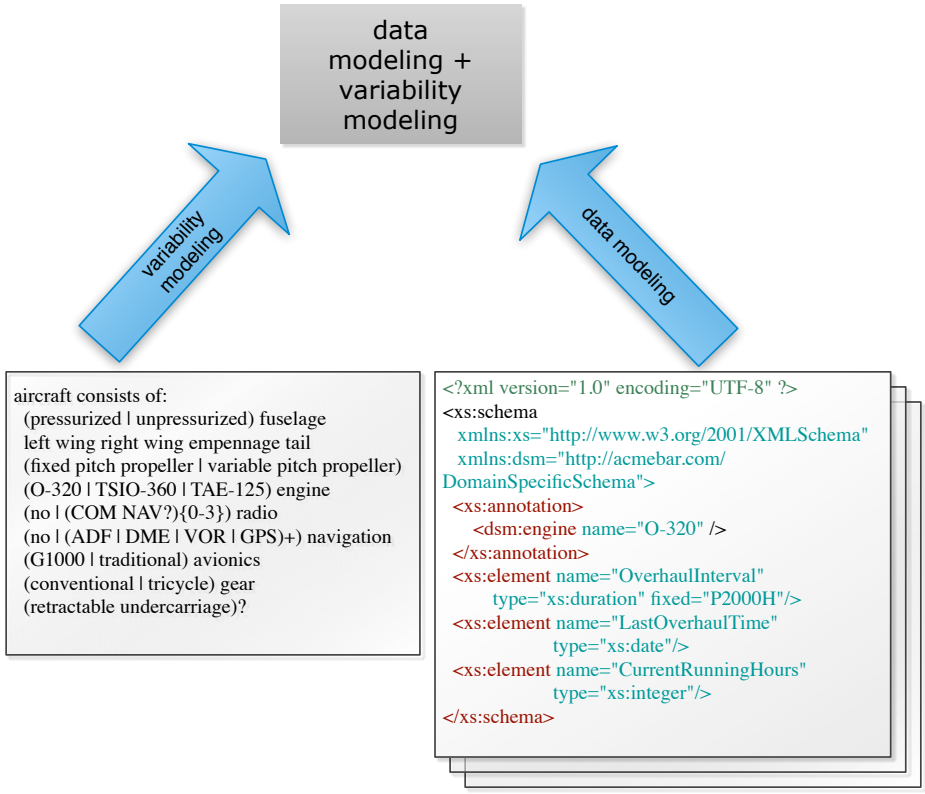

Figure 4.4: Combining two modeling languages

Figure 4.4 shows a conceptual picture of the idea of combining two modeling languages into a meaningful entity. On the right hand side, the schema language defines the data model, and on the left hand side, the variability is modeled using regular expressions. In the middle, these chosen modeling formalisms are merged to a meaningful whole. Should further modeling needs emerge, the structure is open for extension by new modeling formalisms and their combinations.

In Paper (III) we have documented a case of using bottom-up modeling to build a telecommunication provisioning software product. A crucial feature of the developed software was to ensure that only valid combinations of network services were provisioned to the network. We used piggybacking on the regular expressions language to build a validator for checking whether a given combination of telecommunications network services is valid. This

solution can be characterized as a use of a specialized expression language to handle the network models in format of regular expression, but using specialized semantics of feature-oriented domain expressions. This was driven by a feature model, which was used to configure the network service combination validator.

In another part of the software, each of the network services' internals were implemented using XML Schema. Again, we used piggybacking on an existing, well-documented language as a base formalism. The user interface on the product was built based on the model expressed in XML Schema; and user input validations for each specific service were partly implemented by using out-of-box XML validator tools.

4.4 When to use bottom-up modeling

Although we have shown a few cases where bottom-up modeling can be used, it should not be considered as the silver bullet to software engineering problems. Introducing new abstractions always creates initial mental burden to developers and raises the barrier of entry to newcomers into the project. Users of this approach should be aware that they have a home-grown, ad hoc modeling environment, which can cause problems when new people are introduced to the project and/or when the project transits from development to maintenance. For these reasons, it is worthwhile to carefully analyze whether the increased complexity on modeling level can be justified by savings in implementation, documentation and maintenance.

In general, the use of bottom-up approaches involves the drawback of having limited overall vision [EHS10, p. 34]. Since each component is specified and implemented in its own sandbox, there is no overall guidance on how the models should be formulated and whether the models can seamlessly interoperate with each other.

The strength of bottom-up modeling lies in the ability to flexibly adopt suitable modeling tools by embracing new formalisms or by extending existing, well-known formalisms. When the programming environment supports the formalism, such as regular expressions, it can be lightweight to add a new way to use modeling formalism, since there is no requirement for overall structure changes. Also, existing formalisms can be extended or adjusted as needed.

However, if these features are not needed in a project, it is not wise to use bottom-up modeling. For example, if the project already has deployed a suitable MDA toolset that supports all the required technologies and has enough expressiveness for the needs of that project, then there is no need

for the added flexibility and extensibility. Project personnel's (dis-)ability can also be a barrier to adoption: if the project staff does not include personnel with sufficient theoretical knowledge, it can be hard to introduce any improvements.

Another case when bottom-up modeling should not be employed is when the given environment already supports modeling and/or implementation work on sufficient level. For example, in Paper (IV) we show a case of building self-configuring user interface components using Java Server Faces (JSF) [HS07], the standard user interface building technology for Java Enterprise. JSF offers good support for modeling page navigation, i.e. the transitions between different pages in a web application. In that case, it would not be useful to re-invent a new modeling mechanism for handling a case that is already supported by the environment. In general, when given a problem, it is a good idea first to evaluate whether it can be solved by your existing tools: don't reinvent the wheel just because you can.

Designers need to be aware of the fact that after building a bottom-up modeled solution, they have a homegrown, ad hoc modeling environment. Meticulous attention is needed to be prepared to refactor back to mainstream technologies. In the case of refactoring, the good news is that there already is an implemented set of models and an idea of semantics of artifacts in your model. It might be that these models can automatically or semi-automatically migrated to the mainstream solution. For example, many automated tools can transform XML schema documents to UML.

4.5 Related work

Other researchers have also proposed various forms of composing software components in a bottom-up way. In this section, we review some relatives to bottom-up modeling, namely the framelet approach to framework construction and the aggregation of modeling languages.

Frameworks and framelets

The quest for finding reusable solutions to building software has resulted in an uncountable number of frameworks being developed to all possible software domains. The main idea in a software framework is to collect an abstract design of a solution into a coherent set of classes, providing the desired service [JF88]. When the framework needs to be varied, it is done by parametrizing the framework with situation-specific classes that the framework calls at specified points. This is called *inversion of control*, or the Hollywood Principle of "do not call us, we will call you" [Swe85].

Using frameworks to build software is essentially a top-down approach: the framework defines the overall structure (top) and the application developer fills in the missing pieces by providing his own classes (down). When the framework supports the task at hand, this can be a good boost for productivity.

However, in practice we seldom find frameworks that fit exactly to the task and environment [Cas94, MB97, KRW05, LATaM09]. Instead, software is typically composed from a number of frameworks, each handling different domains of the software. A basic web application can contain frameworks for HTTP communications, user interface building, security, and persistency just to name a few. When each one of these wants to be the one who defines the control, problems are bound to be born.

One solution to this problem is to source full application development stacks instead of individual frameworks with the presumption that the stack developer has a thought out framework hierarchy to provide a consistent functionality. The downside is that the scope of a software stack is even more focused that of the individual framework's.

A bottom-up alternative can be called framelets [PK00]. A framelet is a small solution to a specific problem that does not assume main control of the application. Framelets have a defined interface, and they provide variation points for application-specific functionality. They can be seen as a midway between framework-related design patterns and full-fledged frameworks.

Domain-specific modeling

Domain-specific modeling is an activity of recognizing the relevant entities of the problem domain and to use a dedicated editor to model the software system's behavior, by the terms used in the domain [SLTM91, KT08]. On surface level, it might seem that bottom-up modeling is the same approach as domain-specific modeling. However, a number of differences exist.

First of all, domain-specific modeling is a tool-specific activity. The principle is to use a special tool for crafting models that precisely describe the target domain. Introducing new tools to a project can sometimes be problematic for many reasons. Sources of problems can be in the areas of compatibility to existing toolset, licensing politics, and increased complexity to understand the new tool, to name a few.

The bottom-up approach, on the contrary, can be applied without external tool support, as was shown in the example in Section 4.3. However, evidence from previous experience, such as the case reported in Paper (III), suggests that a home-grown modeling language introduces its own com-

plexity. In this case, when the number of modeling elements grew over 20, developers would have benefited from improved tool support. Thus, these models without tool support should probably be used only for prototyping and for bootstrapping development.

Second, domain-specific modeling concentrates on specialized models that are positioned in the target domain. The aim is often to help non-project people to comprehend the models and to allow better collaboration between technical and non-technical people. In contrast, in bottom-up modeling the idea is to find existing, general-purpose formalisms. Rather than giving tools to the customer, as is done in domain-specific modeling, the idea is to give tools for efficient implementation of target domain constructs. However, these models can be used to communicate to non-project people as well. Depending on the communication target, prettifying transformations to simplified graphs and reports can prove to be beneficial.

Aggregated modeling languages

While framelets are an answer to modeling the structure of a software in a bottom-up way, they do not cover data and functionality modeling. Other research has concentrated on reusing existing computational formalisms to build complete modeling environments. Combining aggregated modeling languages from a set of base formalisms, using automation to produce a coherent modeling environment, is a related concept [LV02].

Using these ideas, a demonstration of building an Android application from a set of base formalisms has been done. The base formalisms include modular definitions of the execution platform's properties, such as the device's features and screen navigation model, and the application's functionality model encoded in a state chart. These models are combined to produce a complete application that is executable on the corresponding mobile device [MV12].

Chapter 5

Programs as models

In the previous two chapters we discussed the notion of models in software engineering. We distinguished external models, where the model is an external entity to the software, from internal models, where the model is placed inside the software by terms of object-oriented modeling, language binding, or by other means. In this chapter, we further extend the notion of internal modeling to include also the software itself. We do this in order to show how it can be beneficial to create internal model-driven transformations in order to build resilient software that is easy to change since the resiliency features reduce the effort needed in updating software's internal dependencies in case of changes.

In Section 5.1 we discuss metaprogramming and generative programming as tools of building resilient software. Section 5.2 views program annotations as hooks for attaching external semantics to program fragments. Section 5.3 extends the notion of software models to include the software itself as a model. Section 5.4 reviews related work to the idea of regarding software code as the model in model-driven engineering.

5.1 Metaprogramming and generative programming

In advanced software engineering, experienced programmers use metaprogramming and generative programming as their tools. Generative programming is a discipline of creating programs that produce other programs as their output. The term metaprogramming refers to the task of creating programs that use and modify other programs as their input and/or output.

Many authors in the literature claim that efficient use of these tools is

the key to enormous gains in productivity [TB03, BSST93, JJ05, SS03]. For example, it is claimed that the success of the first commercially successful e-commerce web-based startup company, Viaweb, is mostly explained by the use of the LISP programming language and its metaprogramming facilities [Gra04].

LISP is an example of language that has a low barrier for metaprogramming. It is a homoiconic language, meaning that the programs in the language are represented as data structures in the language itself, using its native data types. This property has made it natural for LISP programs to generate parts of the program at runtime.

In non-homoiconic environments, the means of metaprogramming vary from environment to environment. Many modern languages provide some support for computational reflection, meaning that programs can access their structure and execution state through a reflectional interface. When the program only observes its structure and state, it is called to be an introspective program. If the program also modifies its structure, it is said to be an intercessing program. Both of these reflectional ways require that the program's structures are reified for runtime access.

When the environment (e.g. programming language) lacks proper support for computational reflection, program designers have developed a number of techniques to overcome the limitations of the environment. For example, implementing automated memory management in systems not natively supporting such notion is a good example.

Automated memory management

Automated memory management is a term for employing techniques that allow the program to be designed without explicitly considering the memory allocation and deallocation sites in the program flow. Often, the use of automated memory management causes certain runtime overhead. However, since manual memory management is error prone and tedious, automated memory management can provide more a secure way to manage allocations and de-allocations. In many business sectors programmer productivity is of higher importance, and thus automated memory management gets deployed to practice.

Software engineering wisdom states that in order to build complex systems efficiently, the two most important issues to handle are abstraction and modularity. A given system can be decomposed into modules using different criteria, each decomposition resulting in different properties for performance and maintainability [Par72]. Researchers in garbage collection techniques argue that explicit memory management is an unnecessary

burden in many cases. The unnecessary book-keeping of low-level memory structures detains the focus from more relevant parts of the code [JL96, p. 9-11]. In other words, the manual book-keeping of memory references introduces internal dependencies that violate the modularity aspects, which in turn makes the software less maintainable.

There are many ways to achieve automated memory management in a software system. A common approach is to use a separate thread of execution inside the virtual machine executing the code. The garbage collection thread maintains a list of referred objects, and whenever it is evident that a certain object cannot be accessed or will not be accessed in the future execution, the space used by the object is freed. This is the model used by many current virtual machine based execution environments, such as Java and C#.

However, in other environments some other techniques can be used. For example, the C and C++ language environments do not offer an automated memory management at standard level. To overcome this limitation in these environments, designers often build their own, home-grown memory management systems by using concepts of reference counting, smart pointers or other techniques. As a non-trivial development task, these solutions tend to lack the required technical maturity for building production software. For example, the memory management in applications written for the Symbian OS is known to be horrendous. As a result, a study has found that three out of four times of device freezings could be attributed to memory access violations or heap management problems [CCKI07]. The demise of that operating system in popularity can partly be attributed to its poor support for building applications [TSL11].

Instead of building project-specific garbage collection mechanisms, a library-provided solution can be used. For example, a replacement of the standard allocation and deallocation functions doing automatic memory management has been available for decades [BW88]. This solution can be used in many traditional programs without any modifications; the only difference in many cases is that the existing program's memory leaks are fixed by the use of the library.

Non-standard heap-allocation can be used to perform automatic memory management in virtual machine based environments as well. For example, for the Java environment much research has been conducted to provide escape analysis of objects. If an object can be proven to be local for its execution thread and execution flow, performance benefits can be realized by allocating its space from the stack and reducing the needs for locking [CGS+03]. In this approach, the standard execution environment is modi-

fied to analyze the program flow, and to use a different allocation scheme for local objects.

The analysis of program's code can also be done with external tools. For example, researchers have documented a case of a Scheme compiler which instruments the generated code with functionality to detect memory leaks and to visualize the heap state [SB00].

As can be seen from this section, by just scratching the surface of the research done for a small sub-area of software engineering, we have identified a number of approaches for implementing automated memory management. The prevailing virtual machine based approach is complemented by a number of other techniques that use the source program as the model for configuring the way how memory is allocated and deallocated. These solutions range from project-specific ones, which tend to be poorly generalizable, to generic, library-based solutions. As a bottom line, this discussion shows that it is actually not very uncommon for a software structure to contain self-configuring components.

5.2 Program annotations

Annotating programs to provide hooks for different viewpoints is a popular way to embrace multiple meanings for software components. For example, the Scala program in Figure 5.1 is a tail recursive way for deciding whether its input is a balanced list of parentheses.

```
@tailrec
def balance(chars: List[Char], level: Int): Boolean = {
  if (level < 0) return false
  if (chars.isEmpty) return level == 0

  if (chars.head == ')')
      return balance(chars.tail, level - 1)
  else if (chars.head == '(')
      return balance(chars.tail, level + 1)
  else
      return balance(chars.tail, level);
}
```

Figure 5.1: Compiler-enforced annotation to tail-recursiveness

In this example, the @*tailrec* annotation ensures that the program re-

ally is tail recursive - it tells the compiler to reject the code if it cannot be compiled to be executed in constant stack space.

Annotations are not limited for the compiler use only. Actually, often the primary target of annotating programs is to guide the software's frameworks and external tools about the meaning of the program. For example, when using an object-to-relational mapping (ORM) component to build a persistency layer, it is convenient to annotate class members with instructions to the ORM tool on how to persist the member to the database. The alternative is to use external configuration files to do the same.

Using an external configuration file decouples the actual definitions and corresponding semantic guides. This alternative is a good idea when there are many possible semantic interpretations in the given aspect, e.g. due to reconfiguration needs. However, when there is only one meaningful interpretation, it probably is a better idea to place the instructing guide as a program annotation.

Programming environments usually support a standardized way to use annotations with external tools. For example, the Java 5 platform provides a possibility to annotate classes, class members, interfaces and other language constructs (but not all, e.g. local variables). An external tool [Sun04] provides hooks for external projects to plug into the processing of annotations. The external tools can be used to modify the structure of the processed source code, based on the purpose of the plug-in in question.

Program annotations can be seen as a complementary way of providing program reification. In addition to the program's dominant meaning, the alternate aspects can be guided via annotations. This way, program annotations can be used to decouple different aspects of the software's structure into meaningful modules. For example, consider the code fragment in Figure 5.2 written in Scala:

```
@Entity
class PersistentBean = {
  @BeanProperty
  @Id
  var id;
}
```

Figure 5.2: A class with multi-aspect annotations

The code introduces the class *PersistentBean*. The class is annotated to be an *@Entity*, which tells the persistency framework that this class is to be persisted. The class has a single attribute, *id*. There are two annotations

to this attribute: *@BeanProperty* and *@Id*. The former annotation tells the Scala compiler to automatically add getter and setter functions for the attribute. The latter annotation tells the persistency framework to treat the attribute as the identity field of the objects belonging to this class.

Although the persistency framework advocates the use of transparent persistency and the Scala language tries to avoid unnecessary introduction of get and set methods to all public attributes, the presence of persistency and the tradition of Scala's predecessor have "sneaked" into this design. This kind of mixup, known as *annotation hell* [RV11] when unrestrictedly used for larger number of annotated domains is not uncommon in large software systems. Sometimes this kind of rectifying from the past is just convenient, but sometimes we wish to have better means to control the forces that drive us to use annotations.

5.3 Program fragments as the source models

Although program annotations can be useful in many situations, there are a number of cases where the use of annotations cannot be justified.

First of all, the annotations were originally developed to help in injecting aspect-specific processing instructions to program elements. As promoted in aspect-oriented programming [KLM+97], different aspects should be modularized into decomposable units. The use of annotations injects aspect-specific instrumentation to the target site, which breaks the principle of modularity apart since the dependence target needs to be modified according to the needs to the dependent.

Another scenario where annotations are problematic is the case of multiple aspects being attached to a single entity. In this case, each aspect would inject its own semantic instructions to the instrumented site, which can prove to cause more problems than the injection mechanism can actually solve.

Fortunately, there is an alternative: to build the software architecture to be aware of its own structure. By building self-organizing software components into the software architecture, there is no need to use artificial annotations to guide the different components into their semantic meaning.

As an example, we will build a simple command processor. Let us consider the Java code in Figure 5.3. It first registers three objects for handling different commands, and then repeatedly reads in a command and dispatches following arguments to the given command.

For our discussion, the interesting property in this code lies in how the

```
class CommandProcessor {
    static Map<String, Cmd> funcs =
        new HashMap<String, Cmd>() {{
            put("print", new PrintCmd());
            put("noop", new NoopCmd());
            put("quit", new QuitCmd());
        }};
   private static Scanner scanner = new Scanner(System.in);

   public static void main(String a[]) {
        while(true) {
           String cmd = scanner.next("\\w+");
           String args = scanner.nextLine();
           funcs.get(cmd).Execute(args);
        }
    }
}
```

Figure 5.3: Code for a command line processor

processor uses a dynamic data structure as the storage for the registered commands. Using a dynamic structure makes it easy to add new commands at later time. In contrast to implementing the same functionality by using e.g. a switch-case construct and hard coding the possible commands into the structure of the command processor, this dynamic solution makes the program easier to modify.

This flexibility is gained with the minor runtime cost of using a dynamically allocated data structure with every command fetching being routed through the hashing function of the object. Although the runtime cost is small, it still adds some memory and runtime overhead, since the generic hashing implementation cannot be optimized for this specific use case. For example, the standard Java implementation for HashMap allocates the default value of 16 entries in the internal array implementing the map. In this case, only three of the entries are used, as shown in Figure 5.4. Also, when fetching the command object for a given command, a generic hashing function is used, which also gives room for optimization.

With this discussion, we can see characteristics of accidental maintainability in our example. With accidental maintainability we mean that in this case the solution uses a dynamic data structure for handling a case that does not actually require a dynamic solution. Namely, the set of available

0	null
1	null
2	null
3	null
4	null
5	null
6	NoopOb
7	null
8	null
9	null
10	null
11	null
12	null
13	QuitOb
14	PrintOb
15	null

Figure 5.4: HashMap default layout

commands is a property that is bound at design time, but implemented using a structure that employs runtime binding. There are a number of reasons for implementing the command processor in this way. The map implementation is available in the standard class library, its use is well known and understood among programmers and usually the induced overhead is negligible. Yet another reason can be the lack of viable alternatives. In cases where any overhead should be minimized, introducing this dynamic structure purely due to comfort of the implementer would not be good use of scarce resources.

An alternative solution to this example is to create a specific implementation of the map interface that is statically populated to contain all the required elements. This would make it possible to use context-specific knowledge of the structure in implementing the command fetching system: instead of using a fully generic hashing table, more memory and runtime efficient, specific hash table and hashing functions for the three commands could be implemented.

The self-configurator component resolves this problem by introducing a configurator component to this structure. Figure 5.5 illustrates the configuration process. The self-configuring component reads the static list of commands and generates a specific hashing function, using e.g. *gperf* [Sch90] for this set of commands to be used. Now the runtime and memory

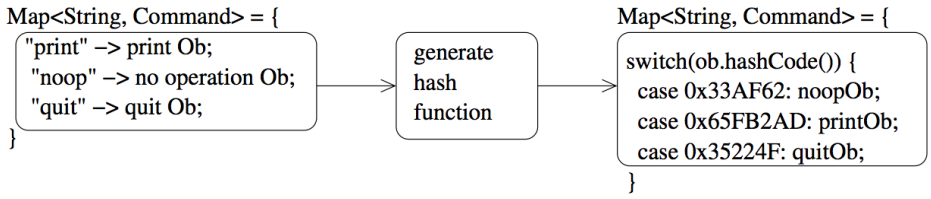

Figure 5.5: Self-configuring function for the command processor

overhead of generic hashing is avoided. The hash generation function is bound (i.e. executed) at the same time as all other parts are compiled. This way, the runtime overhead can be minimized. However, the design-time allocation of command names and associated functions still enjoys the flexibility of defining the command mapping as a well-understood, standard Map interface.

There is a degree of freedom in placing this generative part in the binding time continuum. The hash generating function and associated hash map generation can take place as part of the normal compilation process, or they can be delayed up until first use of the command processor object. As usual, earlier binding time gives opportunities for optimizing for that special case, while dynamic binding gives more flexibility and possibilities to use contextual information to determine the behavior.

Applicability

There are many situations where self-configuring components can prove to be useful. First of all, the pattern is applicable when you are using dynamic structures to guard against changes that a future developer might be performing. In the example in the previous section, the dynamic mapping structure defines a clear place for implementing additional commands. However, this flexibility is gained by introducing additional runtime cost.

Another scenario where you can find this pattern useful is when there is a need to provide characteristics of one code site to parameterize another routine. An example of this case can be e.g. a dependency between a set of different algorithms performing a computation upon data that is held in the database. Each algorithm requests certain set of data, but you want to separate the database fetching code from the algorithm's processing code. In this case, you can introduce a self-configuring component to analyze each specific algorithm and to automatically produce optimized queries for each of them without introducing a dependency between the query site and the algorithm.

These types of applications have a dependency between the data that is read from the database and the algorithm performing the calculations. Within the object-oriented style of programming, an additional object layer is built on top of a relational database, creating an additional problem of object/relational mismatch. An approach of building object-to-relational mapping frameworks, such as Hibernate [BK06], proved to be popular as a bridge between object-oriented application code and relational persistence structures. In order to provide a fluent programming environment for the object-oriented design, transparent persistence is one of the key phrases. The promise of transparent persistence means that objects can be programmed as objects, without paying attention to the underlying relational database.

One of the tools for achieving transparent persistence is the usage of the Proxy design pattern [GHJV95, pp. 207-217] to hide if an object's internal state is stored in the database, or whether it is already loaded to the main memory. However, in many cases this delayed fetching hides symptoms of bad design: the program relies on the slow, runtime safety net implemented with the proxy. A better design would be to explicitly define which object should be fetched. If the objects to be processed within a certain algorithm can be known beforehand, the usage of the Proxy pattern can be classified as a design fault.

Optionally, the pattern can also expose details of the processed dependency via a dependency interface, which allows programmatic access to characteristics of this dependency. In the previous example, this kind of dependency lies between the statically allocated list of commands and the command-line processing loop.

Implementation

In order to analyze a code site for configuring its dependents, there needs to be a way to access the source data. When using compilation-time configuration, all the source code is available for analysis. For instantiation time and runtime configurations the analysis interface is defined by the execution environment characteristics: some environments, known as homoiconic, such as the LISP language, expose the full structure of the program for further analysis; but many current environments do not. In the latter case, the implementor needs to use his own reification strategy. Popular alternatives range from byte-code analysis, such as the BCEL library [Dah99] in the Java environment, to standardized API access to program definition, as implemented in .NET Expression trees, a data structure available in C# since its third version [Mic07].

Regardless of the used access method, the configurator component analyzes the dependent source. Based on this analysis, the dependent is configured to adhere to the form that is required by the source site. In the previous example, a possible configuration could be a generation of a minimal perfect hashing table for the different registered commands.

Often the required target configuration varies from one context to another. What is common in different variations is the built-in ability for the architecture to adapt to changes between architectural elements, which help both in maintenance and in gaining understanding of the overall system.

Drawbacks

Fred Brooks has asserted that *"There is no single development, in either technology or management technique, which by itself promises even one order of magnitude [tenfold] improvement within a decade in productivity, in reliability, in simplicity"* [Bro87]. This work does not claim to be such, either.

When building self-configuring software architectures, there are problems in pre-emphasizing the future needs. The self-configurator ought to be able to adjust its structure to meet the needs of future enhancements. Unfortunately, those people who have been gifted with clairvoyance ability do not end up being software developers. So, the main bulk of architectural work is done based on best guesses.

For example, the self-configurator component presented in Paper (V) can automatically reorganize database queries based on the algorithm accessing the database as long as the component can recognize the analyzed algorithm's structure. If the implementation changes from the structure expected by the self-configurator, then it obviously fails in its job.

Another problem is that writing self-aware code is often considered difficult. In many cases, it seems to be outside the scope of project personnel. Although we have argued that in the past we have been able to implement self-organizing components in agile projects with strict time-boxing limits, it might be the case that this property is not generalizable over all software engineering organizations. In many places, even the term metaprogramming might be unknown concept. In these kinds of organizations, it can be better to start improvements by employing the more classical, well-matured productivity improving techniques.

Empirical results

The roots of self-configuring components are based on industrial software engineering setting: a software product line developed in telecom sector needed improved configurability. Papers (III), (IV) and (V) discuss different aspects of building these components. As this work was done in one company, in one development team using one set of technologies, one can argue that the success of the techniques relied upon the skilled engineers who were doing extraordinary work in that one particular environment. To prove wider applicability, the techniques should be able to demonstrate usefulness in other contexts as well.

Controlled experiments are a way to gain insights to this kinds of phenomena. Widely used in many fields of science, they can be employed in computer science as well. One of the most straightforward ways of performing controlled experiments is the A/B testing. The idea is to divide the test population to two groups: the first group acts as the control group: they experience a traditional, or baseline treatment. The second group is exposed to an alternative, slightly varied treatment. Finally, changes in outcomes is observed.

In order to understand how well the self-configuring components work in new contexts, we performed a randomized, controlled experiment as documented in Paper (VI). Although the number of test subjects was low, the initial results were impressive: test subjects using the self-configuring components outperformed the traditional object-to-relational mapping users by a factor of three in number of functionally correct submissions. As the result is statistically significant, we have a reason to believe the approach to be useful in other contexts as well.

5.4 Related work

The theme of reducing maintenance effort via metaprogramming support can be seen in many existing software projects. In the context of self-configuring software components, the special interest lies in the area of introspective metaprogramming that allows the software to reconfigure its components as maintenance changes are being made.

Self-configuring components by reflection

Using reflection to build self-configuring components has long been known as a way to build resilient software. For example, the standard class library in Java since version 1.3 has included the concept of dynamic proxy classes

for interfaces. Other people have also proposed extending this functionality to classes as well [Eug03].

Similar techniques can be applied to arbitrary code entities. For example, in section 4.5 we discussed framelets as a bottom-up way to building frameworks. Using reflectional access to the interface of a framelet has been shown to be a viable way of automating component gluing [PAS98].

Automatic compiler generation

Generative programming has long been used in constructing the front-end of a compiler: *lex* [LS75], *yacc* [Joh78], and their descendents and look-alikes are routinely used in building the front-end. However, this solves only a fraction of the problem; some estimate the scanning and parsing phases to account for 15% of the whole task [Wai93].

Tim Barners-Lee is quoted of saying: *"Any good software engineer will tell you that a compiler and an interpreter are interchangeable"*. The idea behind this quote is that since the interpreter executes code in the interpreted language, it necessarily has the required knowledge for producing the equivalent lower level code. Also the other way applies: the compilation routines for a given language can also be harnessed to build an equivalent interpreter.

The top-down approach to increasing the amount of automatically generated parts of a compiler is to introduce stronger formalisms for describing the functionality of the compiler. Larger parts of the compiler can be generated by using e.g. attribute grammars to describe the compiler's semantics [KRS82] .

This interchanging process can also be seen as a self-configuration component. This has been applied e.g. to build compilers for embedded domain-specific languages [EFM00] and to produce portable execution environments for legacy binaries [YGF08].

For example, consider the code in Figure 5.6 as an interpreter of a language. The interpreter is given a list of function pointers to the instructions of the executed program. After each function pointer invocation, the instruction pointer is incremented to point to the next memory location where the next instruction resides. When the instruction pointer becomes null, execution terminates.

This is a compact way to implement an interpreter. However, the real problem resides in implementing the actual instructions. The usual way for implementing semantics of the language, as employed e.g. in Paper (II) is to hand-write a code generator to emit the lower-level instructions. The example used in the Paper shows an implementation for generating code

```
int *ip;
void interpreter() {
  while(ip) {
   ( (void (*)())*ip++ )();
  }
}
```

Figure 5.6: Function-pointer hack for running an interpreted language

for addition and multiplication, as shown in Figure 5.7

```
1  void visitDigit(Digit d) {
2    emit("iconst"+ d.value);
3  }
4
5  public void visitOperator(Operator oper) {
6    if("+".equals(oper.value)) {
7      emit(" iadd");
8    } else if("*".equals(oper.value)) {
9      emit(" imul"); }
10   }
11 }
```

Figure 5.7: Emitter code for integer addition and multiplication (Paper II)

The code emitter works as a Visitor pattern [GHJV95, p. 331-344] over an abstract syntax tree. The lower-level byte code to be emitted is hand-written as calls to *emit* function, which handles the actual output. This approach is known as functional decomposition solution to the expression problem [OZ05]. In Paper (II)'s example, the addition and multiplication operations are also implemented in the host language for interpreter work, as shown in Figure 5.8.

The solution for the interpreter to work uses object-oriented decomposition to encapsulate operation semantics to different subclasses. The problem in combining two different decomposition methods is duplication: the semantics for addition and multiplication operations is defined once for the compilation context and once for interpreter context. If one of these happens to change, there is no guarantee that the other will be changed as well.

A self-configuring approach can be used to remove the duplication here,

```
abstract eval();
expression lhs, rhs;
operator() {
  if (token == '+') { eval <- plus.eval; }
  else if (token == '*') { eval <- times.eval; }
}

// methods plus.eval and times.eval
plus.eval() {
     return lhs.eval() + rhs.eval();
}
times.eval() {
     return lhs.eval() * rhs.eval();
}
```

Figure 5.8: Interpreter code for integer addition and multiplication (Paper II)

as is shown in [YGF08]. When we examine the code produced by the standard compiler for the method *plus.eval*, as shown in Figure 5.9, we see clear correspondence to the Visitor method in Figure 5.7. The *iadd* instruction on the line 7 in the Java listing corresponds to the byte code instruction at index #2 in the reverse engineered version of the class.

```
$ javap -c Plus
Compiled from "Plus.java"
public class Plus extends java.lang.Object{
[..]
public int eval(int, int);
  Code:
   0: iload_1
   1: iload_2
   2: iadd
   3: ireturn
}
```

Figure 5.9: Compiled code for method plus.eval

The instructions at indices 0 and 1 in the compiled code load their arguments. The instruction 2 is correspondent to the emit-method call in Figure

5.7. Finally, the instruction 3 returns the result back to the caller. To build a self-configured version of the interpreter/compiler, the self-configurator can build a compiler from the interpreter by analyzing each opcode definition of the interpreter and by emitting each opcode's corresponding code as the code generation step. This is a way to treat software's methods as the source model of a model-driven compiler engineering.

Software renewal via code transformations

In the heart of software maintenance justification lies the fact that rewriting software is a dangerous business decision. Existing software investment encodes countless special cases and their handling rules in its corresponding environment. Evolution can happen via modular changes, but full system changes are rare and inherently dangerous. This is why the text in flight tickets is still written in upper case letters.

Sometimes, in cases of low competition, companies might be able to renew their main operating software. An example of a documented survival case for a company transiting from a mainframe-based software to a newer Windows-based system can be found from Finnish Agricultural Data Processing Centre [NJGG10]. In this case, the transition from the 30-year old mainframe-based legacy information processing environment to modernized relational database system was made possible only by low pressure from competition.

In environments of higher competition, software businesses very seldom can enjoy the liberty of rewriting their old codebase. For example, the main reason for Netscape to lose the so-called "browser war" and ultimately its independence is attributed to the business decision to rewrite their old code [Spo04, p. 183-187]. Based on this example, companies are usually reluctant to make business decisions to a full-rewrite, and they probably are wise in that.

Figure 5.10 draws a picture of what customers experience when a company decides to rewrite its software from scratch. The company can ship the old version of the software while work is undergoing to produce the new version (*old version functionality*). Customers experience no new features or other development while development of the new version is in progress. It is going to be a long time before the recoding achieves the level of functionality provided by the old version (*recode progress*). It should be noted that unlike people often assume, the total effort in redoing the software probably is not smaller than when doing the previous version [Spo04, p. 186].

The next option is not to throw the old codebase away, but place it in

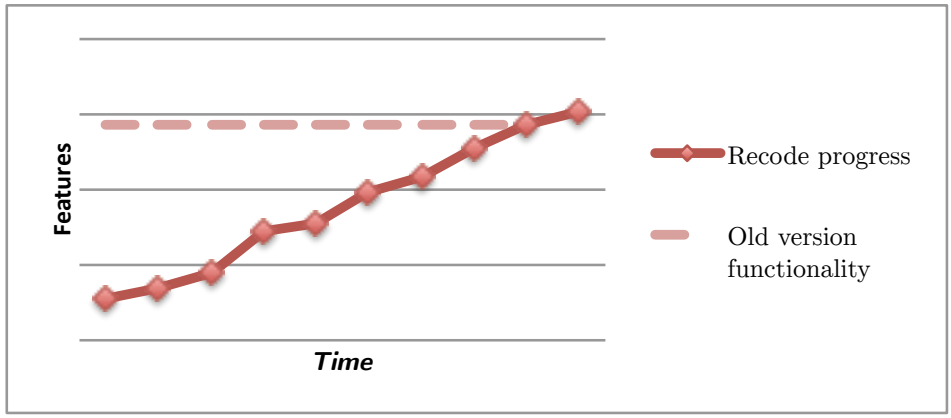

Figure 5.10: Rewriting the software from scratch: time-to-market is long [KB12]

maintenance mode until the new version has been completed. The split-versions strategy can work for companies with large resources to throw at a given renewal project. Depending on competition situation, the time-to-market can be shortened by selecting only the most important features of the software to be implemented first. This strategy is what Apple used when transitioning the QuickTime software from 32-bit version 7 to the 64-bit version QuickTime X. The initial release of version X was severely limited: it allowed only the very basic video playback operations, and the older version was still offered for any serious work [Sir09].

This strategy is shown in Figure 5.11. Customers experience a slowed-down development of the old codebase, and the new development is slowed down due to the need to support the old codebase. It can be questioned whether the rewrite will ever get ready in this case.

Unfortunately, not all companies have similar resources and strong grip of their customers, and they also need to renew their software offering without such major disruptions that QuickTime users experienced. Code transformations can be used to help in code renewal in a similar way to self-configuring components.

An example case of using code transformations for code renewal has been presented in industrial context [KB12]. In this case, the company has built a ship structural modeling software NAPA, based on Fortran and self-developed BASIC-flavour, with the total line count ranging in 2 million source lines of code in Fortran and about the same in the in-house BASIC. The company developers have estimated the existing codebase to account for 150 man years of work.

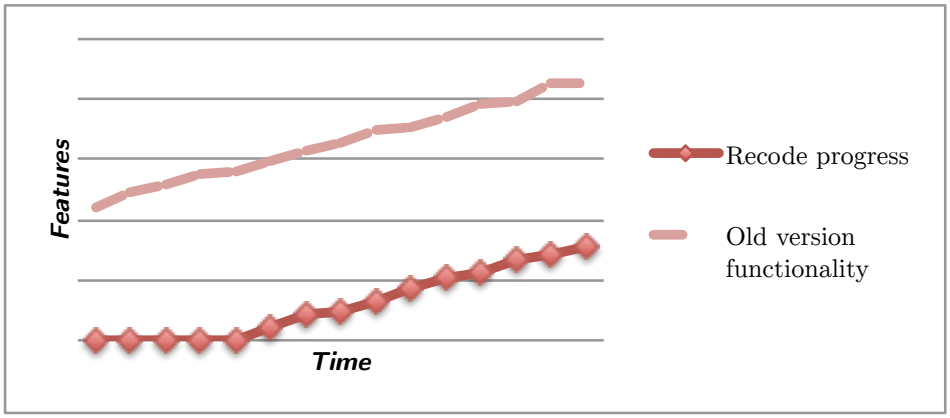

Figure 5.11: Recode on the side: split focus hinders all development [KB12]

Since not many computer science curricula include Fortran as a major educational building block and the used BASIC language is company-specific, the company is experiencing a major struggle in driving software development. The business environment gives a continuos pressure to keep adding features to the product, mandates maintenance development for existing customers and requires custom projects of special features for key customers. Competition ensures that "freezing" of feature development is not an option for a major technology overhaul [KB12].

Using code transformations similar to self-configuring components can be used to build a roadmap of technological transition. In the NAPA case, the work consisted of a BASIC-parser written in ANTLR [PQ94] and few thousand lines of pattern matching code, written in Mathematica [Wol03]. In this kind of renewal the idea is to iteratively find generic rules for translating from the legacy system to the modern technology [Bro10]. Similar approaches have been used previously in legacy system renovations, for example in renewal of COBOL-based banking systems [vD99].

In principle, the renewal progresses as shown in Figure 5.12. The coverage of translation rules can be supposed to experience a law of diminishing returns: initially few basic rules can handle large amounts of legacy source code. The leftovers are harder and harder to handle using systematic pattern matching rules. When the gap between legacy functionality and the code covered with transformations is deemed small enough, the development team makes a "jump of faith" to generate a modernized version of the legacy code, and all further development stays in that version.

In this version, customers see development continuing with normal, or a bit slower pace in the legacy codebase. With iterative enhancements to

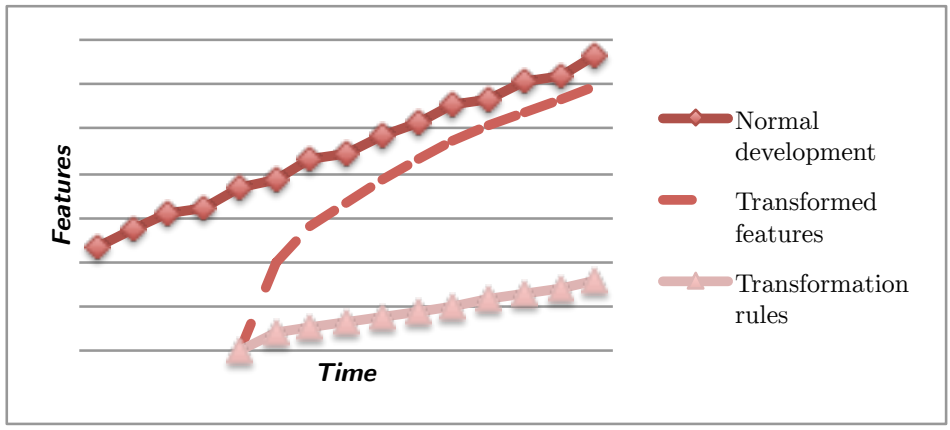

Figure 5.12: Code transformations are iteratively developed [KB12]

the transformation rules and legacy codebase refactoring, the amount of correctly transformed code in the legacy platform grows. After the transformation rules is deemed good enough, all development is transitioned to the renewed platform, which consists of automatically generated code from the legacy codebase.

However, it should be noted that the development speed charts in Figures 5.10, 5.11, and 5.12 are hypothetical, generated by using a pseudo-random number generator. It should be investigated whether this kind of approach works in practice. Initial reports suggest that in the example case company the transformation has been successful [Bro13], but more rigorous studies should be performed to gain better understanding of code transformation mechanics in practice.

Software productivity via code transformations

Code transformations can be useful not only in modernizing legacy code, but in building multi-platform software as well. For example, when producing software to be run on both Android and iOS platforms, several teams at Google use a Java-to-ObjC translator tool, J2ObjC [Bal13].

The tool has implemented a semantic translation from business-level Java code to corresponding Objective C constructs. This way, the business code can be written only once in Java for the Android platform, and then translated to iOS. By automating this translation, the development teams do not encounter synchronization problems between two code bases, but instead can be sure that the business functionality is based on the same source. Figure 5.13 shows a schema of this tool usage.

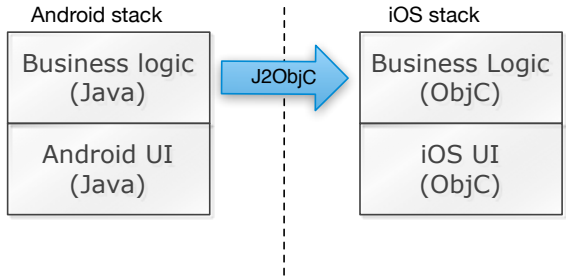

Figure 5.13: Semantic translation from Java to ObjC used to synchronize business logic to two platforms

The tool tackles the problems of full code translation by limiting itself purely to the business domain, meaning that the user interface level needs to be written separately to both of the platforms. This is understandable since the user interface principles are different in the platforms.

Contrary to the one-time translation approach used in renewing technology as presented in the previous section, the target of J2ObjC is to provide a continuous integration between the two platforms. This is a way to provide programming productivity in the case of multi-platform development. Projects can complete faster because they do not need to duplicate the business logic to both platforms. When a new, previously only Android-based project wants to join in, the translator tool usually needs to adjust to handle the new special cases introduced by the joining project. This way, over time the translator tool evolves to handle larger and more complex cases. Projects can meanwhile enjoy improved productivity and concentrate to polish the user experience on the different platforms.

The difference between the examples in this and the previous section resemble the difference between translationalist and elaborationalist approaches to model-driven engineering discussed in Section 3.2. The approach to build an initial code transformation suite to lessen the gap when jumping from the legacy codebase to a modern platform resembles the elaborationistic approach. The code transformation suite is being incomplete and can be compensated by manual coding. On the other hand, introducing the code translation tool as a part of everyday build cycle resembles the translationalistic approach: the target is to support all possible source code constructs.

Chapter 6

Conclusions

Improving programming productivity is the main theme of this thesis. We have focused on two related approaches: first by using bottom-up modeling to build project-specific modeling environments, and second by using the software's own source components as source models in model translations to build improved resilience. Resilience in software means that changes can be made locally, without dependency-based rippling effects breaking its overall structure. In this chapter, we discuss the implications of these approaches.

This chapter proceeds as follows. Section 6.1 relates the material in this introductory part to the papers that follow the introductory part and review contributions of the thesis. Section 6.2 reviews the limitations of the thesis. Section 6.3 outlines possible future research.

6.1 Contributions of the thesis

In Papers (I) and (II) we deploy the traditional software engineering techniques to build an optimizing compiler and a language generator. In these papers, we show incremental evolution to existing approaches.

In Paper (III) we build a project-specific modeling language with a binding to the existing body of computational knowledge. The modeling language is exposed as a runtime entity to the project structure, allowing computational access in other parts of the software. This leads to the idea of using the software's own structures as computational elements as well. This is called the *self-configurator* component, as the self-configurator reads in the software's definition and uses its characteristics to configure some other part of the software. In Papers (IV) and (V) we have applied the self-configurator component in two different contexts. Paper (VI) conducts

an empirical experiment in a randomized, controlled environment with the result of finding the self-configurator variant to be more effective in building working code.

In Paper (IV) we express the case of component-based software engineering that is widely employed in the area of user interface composition. User interface widgets can be developed as stand-alone components, and the interface of a new application can be built by composing from a palette of these ready-made components. Pioneered in Visual Basic, the approach has been adapted to numerous application areas.

One of the drawbacks in component-based user interface composing is the need for duplicated binding expressions when programmatically defining multiple properties of user interface components. For example, when defining whether a user interface component is active or not, a corresponding tooltip should be placed. Without sufficient support for cross-referencing to other binding expressions, providing this kind of conceptual coherence in the user interface requires cloning of the behavior defining expressions.

We have built a prototype for analyzing these binding expressions in the standard Java environment for building web interfaces. By exposing the structure of the binding expressions to backend code, we were able to reduce the amount of cloned binding expressions by a factor of 3 in a demo application presented in Paper (IV).

In Paper (V) we consider typical database applications: they read data from a database to the main memory, perform an algorithm on the data, and then write the result back into the database. In the paper we have documented the usage of self-configured database queries as a tool to improve the runtime properties of this case. In this design, a code analyzer reads in the byte-code of a given algorithm and deducts the required queries for prefetching the needed data from the database. This design improves maintainability: should the algorithm change for some reason, the fetching code is automatically updated to reflect the change. Another benefit is that on the architectural level, the number of database-accessing components is reduced, since this one component can configure itself for multiple cases.

In Paper (VI) we report a controlled, randomized experiment on students to determine whether the self-configurator component is useful for performing maintenance tasks. We built two versions of a simple database application. The first version uses the self-configuring component presented in Paper (V), and the second version uses the transparent persistency. Although the two versions differ only in a few lines of code, based on the results the few lines seem to be crucial for performing maintenance tasks.

The result suggests that unlimited separation of concerns might be harmful for rapid maintenance tasks: the members in a control group, using the self-configuring query component, performed statistically significantly better in the experiment in terms of correct submissions.

6.2 Limitations of the thesis

In the included papers, we have contributed to software engineering techniques in a number of areas, including mobile software development, object-oriented grammar implementation, mobile network development, database development, and user interface development. The usefulness of one of the contributions, the database component, was empirically validated.

It can be seen as a limitation of the thesis that not all of the contributions have been rigorously validated, but their applicability is based on observations of industrial practice rather than randomized, controlled experiments. Unfortunately, performing controlled trials is expensive and time-consuming. Thus, more thorough empirical evaluation was not possible in the scope of this work.

The empirical validation presented in Paper (VI) was performed by using students of the conducting university as test subjects. The test group can be argued to be small, since only 16 students attended the test. For more reliable results, larger populations should have been used.

6.3 Future directions of research

The main message of this thesis can be summarized as follows: in order to build maintainable software, an increasing number of software entities must be addressable by software.

This means that existing abstractions need to be made more accessible through introspective and reflective interfaces. For new abstractions, designers should think ahead how the new abstractions would be programmatically accessed - can the entity in question be a so-called first-class citizen in the system.

This is an everlasting quest, since both academia and the industry seem to be endlessly creative in creating new structures, frameworks and systems to be used. Since the number of possible combinations is unlimited, we have no fear that maintenance work of these would come to an end.

For the short term, the work could expand to cover new application areas for self-configurative components. It would be best to start in domains that involve a high frequency of application and are currently manually

done. The work presented in this thesis concentrated on a traditional software engineering setting, where teams of programmers, designers and architects work on a software product, using general-purpose languages. Using self-configurative components could be employed in this area to speed up development work and to allow clever programmers to concentrate on harder problems, instead of wasting their time in doing bulk, automatable work. In order to validate the usefulness of these efforts, longitudinal studies on productivity would be needed.

We could also broaden the viewpoint from pure software engineering to all software creation. Most of software creation lies outside of software engineering scope: for every professional programmer, there are at least ten business majors who are doing scripting in their spreadsheet programs. If errors in enterprise software are bad, the errors in spreadsheet-based business decisions can be disastrous. For example, famous economists have studied the relation between a nation's gross domestic product and the debt. Their main result is that median growth rates for countries with public debt over 90 percent of GDB are roughly one percent lower than otherwise and average growth rates are several percent lower [RR10]. This all makes sense, except that the conclusion is based on errors in the researchers' spreadsheet: they forgot to include a number of countries that happened to have high public debt and better GDB growth rates [HAP13].

These kinds of problems could be mitigated with self-configuring extract-transform-load (ETL) components. When extracting data from a data source, the current transformations usually evaluate the possible source statements. Often the target system could as well evaluate the expressions, but successfully carrying the semantic transformations from the source system to the target system can be cumbersome. By using self-configuring programs, the transformations could be automatically carried to the target system, e.g. to be used in a spreadsheet. This kind of improvement would be a good target, since ETL programs are widely used in the industry, and are known to be tedious to build and cumbersome to modify. Thus, a self-configuring Excel-sheet is the next thing to do.

In the long term, building improved resilience as a routine work of software engineering is needed. More complex software is being produced with the need to address more specific customer needs. Software product lines, mass customized software and software factories make the space of possible software configurations much larger than programmers, designers and architects have used to handle. Software engineers are so busy with providing value that there is no room for any activities that will be wasted: in the future, less time will be spent on formal reviews, using the UML or

using sophisticated metrics to understand productivity. Customers want their software and they want it now. Cheaply. The key to achieving this is to building in quality assurance: continuous, automated testing is implemented to reduce waste.

The logical next step is to change the software's configuration from manually handled configuration management to automatically managed. Self-configuring software components are a tool for implementing automatically configuring, change-resilient software architectures within the tightly budgeted projects that are today's norm in the software industry.

When self-configuration as a tool to provide self-organization in software architectures is common knowledge, the next steps take us higher: how do self-configuring components interact and how should they interact. Can there be self-configuring components that configure other self-configuring components? Will there be a need for higher-order self-configuration? Questions like this are to be answered in the future.

References

[ABK10] Pekka Abrahamsson, Muhammad Ali Babar, and Philippe Kruchten. Agility and architecture: Can they coexist? *IEEE Software*, 27(2):16–22, 2010.

[ADGR05] Emmanuelle Anceaume, Xavier Défago, Maria Gradinariu, and Matthieu Roy. Towards a theory of self-organization. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *9th International Conference on Principles of Distributed Systems OPODIS 2005, Revised Selected Papers*, volume 3974 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2005.

[AK04] Pekka Abrahamsson and Juha Koskela. Extreme programming: A survey of empirical data from a controlled case study. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, ISESE '04, pages 73–82, Washington, DC, USA, 2004. IEEE Computer Society.

[AK09] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, July/August 2009. Refereed Column.

[Alb79] Aj Albrecht. Measuring application development productivity. In I. B. M. Press, editor, *IBM Application Development Symposium*, pages 83–92, October 1979.

[AM04] Michalis Anastasopoulos and Dirk Muthig. An evaluation of aspect-oriented programming as a product line implementation technology. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 141–156, 2004.

[Amb04a] Scott Ambler. Agile model driven development is good enough. *IEE Software*, 21(5):71–73, 2004.

[Amb04b] Scott Ambler. *The Object Primer: Agile Model-Driven Development with UML 2.0*. Cambridge University Press, 3rd edition, 2004.

[AN93] Alain Abran and Hong Nguyenkim. Measurement of the maintenance process from a demand-based perspective. *Journal of Software Maintenance: Research and Practice*, 5(2):63–90, 1993.

[AR06] Nitin Agarwal and Urvashi Rathod. Defining 'success' for software projects: An exploratory revelation. *International Journal of Project Management*, 24(4):358 – 370, 2006.

[AWSR03] Pekka Abrahamsson, Juhani Warsta, Mikko T. Siponen, and Jussi Ronkainen. New directions on agile methods: a comparative analysis. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 244–254, Washington, DC, USA, 2003. IEEE Computer Society.

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[Bab09] Muhammad Ali Babar. An exploratory study of architectural practices and challenges in using agile software development approaches. In *WICSA/ECSA*, pages 81–90. IEEE, 2009.

[Bal13] Tom Ball. Personal communication, 2013.

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th international conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[BCC+10] Hugo Bruneliere, Jordi Cabot, Cauê Clasen, Frédéric Jouault, and Jean Bézivin. Towards model driven tool interoperability: Bridging Eclipse and Microsoft modeling tools. In Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and François Terrier, editors, *ECMFA*, volume 6138 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2010.

[BCT05] Alan Brown, Jim Conallen, and Dave Tropeano. Practical insights into model-driven architecture: Lessons from

the design and use of an MDA toolkit. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-Driven Software Development*, pages 403–431. Springer Berlin Heidelberg, 2005.

[BG09] Andrew Berns and Sukumar Ghosh. Dissecting self-* properties. In *2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems*, pages 10–19, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[BH12] Andrew Binstock and Peter Hill. The comparative productivity of programming languages. *Dr. Dobb's Journal*, 2012. http://www.drdobbs.com/jvm/the-comparative-productivity-of-programm/240005881.

[BJV04] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the need for megamodels. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[BK06] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.

[BL76] László Belady. and Meir Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, September 1976.

[BLD11] Mario Bernardi, Giuseppe Di Lucca, and Damiano Distante. A model-driven approach for the fast prototyping of web applications. In *Proceedings of 13th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 65–74, Sepember 2011.

[BM04] Paul V. Biron and Ashok Malhotra, editors. *XML Schema Part 2: Datatypes*. W3C Recommendation. W3C, second edition, October 2004.

[Boe87] Barry W. Boehm. Improving software productivity. *Computer (IEEE)*, 20(9):43–57, September 1987.

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[Bro87] Frederick Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[Bro95] Frederick Brooks. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Bro10] Robert Brotherus. Transforming a legacy GUI to WPF and IronRuby. Chapter 10 in Porto Carrero / IronRuby In Action (unpublished), 2010.

[Bro13] Robert Brotherus. Personal communication, 2013.

[BS13] Hamid Bagheri and Kevin J. Sullivan. Bottom-up model-driven development. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *Proceedings of 35th International Conference on Software Engineering, ICSE'13*, pages 1221–1224. IEEE / ACM, 2013.

[BSST93] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. *SIGSOFT Software Engineering Notes*, 18(5):191–199, December 1993.

[BW88] Hans-Jürgen Böhm and Mark Weiser. Garbage collection in an uncooperative environment. *Software, Practice & Experience*, 18(9):807–820, 1988.

[Cas94] Eduardo Casais. An experiment in framework development - issues and results. In *Architectures and Processes for Systematic Software Construction. FZI-Publication, Forschungszentrum Informatik*, 1994.

[CCC+11] Antonio Contieri, Guilherme Correia, Thelma Colanzi, Itana de Souza Gimenes, Edson Oliveira, Sandra Ferrari, Paulo Masiero, and Alessandro Garcia. Extending UML components to develop software product-line architectures: Lessons learned. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Proceedings of 5th European Conference on Software Architecture*, volume 6903 of *Lecture Notes in Computer Science*, pages 130–138. Springer, 2011.

[CCKI07] Marcello Cinque, Domenico Cotroneo, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. How do mobile phones fail? A failure data analysis of Symbian OS smart phones. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN '07*, pages 585–594, June 2007.

[CdPL02] Luiz Marcio Cysneiros and Julio Cesar Sampaio do Prado Leite. Non-functional requirements: From elicitation to modelling languages. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 699–700, New York, NY, USA, 2002. ACM.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CGS+03] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876–910, November 2003.

[Cha94] The CHAOS report. Technical report, Standish Group International, 1994.

[Cha03] The CHAOS report. Technical report, Standish Group International, 2003.

[Cha09] The CHAOS report. Technical report, Standish Group International, 2009.

[CJ01] Matthias Clauß and Intershop Jena. Modeling variability with UML. In *GCSE 2001 Young Researchers Workshop*, 2001.

[Coo12] William Cook. UML is the worst thing to ever happen to MDD. Fortunately many people now realize this ... should I list the reasons? UML was created to model OO designs. It effect you are modeling the code of a system, not the system's behavior. UML is at wrong level. 2) the idea that 7 (or 13) diagram formats in UML can cover everything is crazy. What about GUIs, web wireframes, authorization, etc. ??? 3) UML has encouraged the idea that models must be

graphical. Ridiculous! Text and graphic models are both useful and often interchangeable. 4) UML is at once too large and complex and at the same time very limiited. stereotype and profiles are not effective for usable extensions. 5) the PIM/PSM distinction is misguided. The purpose of high-level models is not platform independence. It is about "what" versus "how". Tweet, Jun 2012.

[CROB05] Timothy J. Coelli, D. S. Prasada Rao, Christopher J. O'Donnell, and George E. Battese. *An introduction to efficiency and productivity analysis*. Springer, New York, NY, second edition, 2005.

[DA10] Jean-Marc Desharnais and Alain April. Software maintenance productivity and maturity. In *Proceedings of the 11th International Conference on Product Focused Software*, PROFES '10, pages 121–125, New York, NY, USA, 2010. ACM.

[Dah99] Markus Dahm. Byte code engineering. In Clemens H. Cap, editor, *Java-Informations-Tage 1999 JIT'99*, Informatik aktuell, pages 267–277. Springer Berlin Heidelberg, 1999.

[DBT11] Frank Dordowsky, Richard Bridges, and Holger Tschope. Implementing a software product line for a complex avionics system. In *Proceedings of the 15th International Software Product Line Conference*, SPLC '11, pages 241–250, Washington, DC, USA, 2011. IEEE Computer Society.

[DE05] Bergfinnur Durhuus and Soren Eilers. On the entropy of LEGO, April 2005. http://arxiv.org/abs/math.CO/0504039.

[DeM86] Tom DeMarco. *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1986.

[DeM09] Tom DeMarco. Software engineering: An idea whose time has come and gone? *IEEE Software*, 26(4):96, 95, 2009.

[Den03] Peter J. Denning. Great principles of computing. *Communications of the ACM*, 46(11):15–20, November 2003.

[DP06] Brian Dobing and Jeffrey Parsons. How UML is used. *Communications of the ACM*, 49(5):109–113, May 2006.

[DT09] Shlomi Dolev and Nir Tzachar. Empire of colonies: Self-stabilizing and self-organizing distributed algorithm. *Theorerical Computer Science*, 410(6-7):514–532, 2009.

[EFM00] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. In *Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, SAIG '00, pages 9–27, London, UK, UK, 2000. Springer-Verlag.

[EH07] Torbjörn Ekman and Görel Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.

[EHS10] Basem S. El-Haik and Adnan Shaout. *Software Design for Six Sigma: A Roadmap for Excellence*. Wiley Publishing, 1st edition, 2010.

[Eug03] Patrick Th. Eugster. Dynamic proxies for classes: Towards type-safe and decoupled remote object interaction. Technical Report IC 200317, École Polytechnique Fédérale de Lausanne, 2003.

[Eva03] Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[EvdB10] Luc Engelen and Mark van den Brand. Integrating textual and graphical modelling languages. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 253(7):105–120, September 2010.

[Fea04] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[FGDTS06] Robert B. France, Sudipto Ghosh, Trung Dinh-Trong, and Arnor Solberg. Model-driven development using UML 2.0: promises and pitfalls. *Computer*, 39(2):59–66, February 2006.

[FKA+12] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering*, pages 1–47, May 2012.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, first edition, October 2010.

[FP98] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998.

[GFd98] Martin Griss, John Favaro, and Massimo d'Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 76–85, Washington, DC, USA, 1998. IEEE Computer Society.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Text-based modeling. In *4th International Workshop on Software Language Engineering (ATEM 2007)*, 2007.

[Gla02] Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, October 2002.

[Gli07] Martin Glinz. On non-functional requirements. In *Proceedings of 15th IEEE International Requirements Engineering Conference, RE '07.*, pages 21–26, 2007.

[GMK02] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 33–38, New York, NY, USA, 2002. ACM.

[Gom04] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[Gra04] Paul Graham. Beating the averages. In *Hackers and Painters: Big Ideas from the Computer Age*, pages 165–180. O'Reilly Media, Inc., 2004.

[Gro03] Christian Grothoff. Walkabout revisited: The runabout. In Luca Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 103–125. Springer Berlin Heidelberg, 2003.

[GV09] Iris Groher and Markus Völter. Aspect-oriented model-driven software product line engineering. In Shmuel Katz, Harold Ossher, Robert France, and Jean-Marc Jézéquel, editors, *Transactions on Aspect-Oriented Software Development VI*, volume 5560 of *Lecture Notes in Computer Science*, pages 111–152. Springer Berlin Heidelberg, 2009.

[HAP13] Thomas Herndon, Michael Ash, and Robert Pollin. Does high public debt consistently stifle economic growth? A critique of Reinhart and Rogoff. Working paper 322, Political Economy Research Institute, University of Massachusetts Amherst, April 2013.

[HF01] Jim Highsmith and Martin Fowler. The agile manifesto. *Software Development Magazine*, 9(8):29–30, 2001.

[HH04] Imed Hammouda and Maarit Harsu. Documenting maintenance tasks using maintenance patterns. In *Proceedings of 8th European Conference on Software Maintenance and Reengineering*, CSMR'04, pages 37–47, 2004.

[HISMP+13] Dave Hunt, Luke Inman-Semerau, Mary Ann May-Pumphrey, Noah Sussman, Paul Grandjean, Peter Newhook, Santiago Suarez-Ordonez, Simon Stewart, and Tarun Kumar. Selenium documentation [15.06.2013]. Technical report, 2013. http://seleniumhq.org/docs/index.html.

[HKC11] Werner Heijstek, Thomas Kuhne, and Michel R. V. Chaudron. Experimental analysis of textual and graphical representations for software architecture design. In *Proceedings*

of the 2011 International Symposium on Empirical Software Engineering and Measurement, ESEM'11, pages 167–176, 2011.

[HS07] James Holmes and Chris Schalk. *JavaServer Faces: The Complete Reference*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2007.

[HT99] Andrew Hunt and David Thomas. *The pragmatic programmer: From journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[ISO06] ISO/IEC/IEEE. International Standard ISO/IEC 14764 IEEE Std 14764-2006 - Software Engineering - Software Life Cycle Processes – Maintenance. pages 1–46, 2006.

[ISO10] ISO/IEC/IEEE. International Standard ISO/IEC/IEEE 24765 - Systems and software engineering - Vocabulary. 2010.

[JF88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Object-Oriented Programming*, 1(2), 1988.

[JJ05] Yang Jun and Stan Jarzabek. Applying a generative technique for enhanced genericity and maintainability on the J2EE platform. In Robert Glück and Michael Lowry, editors, *Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 237–255. Springer Berlin Heidelberg, 2005.

[JL96] Richard Jones and Rafael D Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[Joh78] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, 1978.

[Jon94] Capers Jones. *Assessment and control of software risks*. Yourdon Press, Upper Saddle River, NJ, USA, 1994.

[Jon95] Capers Jones. Patterns of large software systems: Failure and success. *Computer*, 28(3):86–87, mar 1995.

[Jon08] Capers Jones. *Applied Software Measurement: Global Analysis of Productivity and Quality*. McGraw-Hill Companies, Incorporated, 2008.

[KB12] Antti Karanta and Robert Brotherus. Renewal of Napa software GUI technologies and architecture with language legacy transformation and architectural refactoring. In *Industry Day presentation at WICSA/ECSA 2012 Conference*, 2012.

[KCH+90] Kyo C. Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute, November 1990.

[KEGN01] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*, pages 736–743, Florence, Italy, November 6–10, 2001.

[Ker88] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[KL07] Birgit Korherr and Beate List. A UML 2 profile for variability models and their dependency to business processes. In *Proceedings of the 18th International Conference on Database and Expert Systems Applications*, DEXA '07, pages 829–834, Washington, DC, USA, 2007. IEEE Computer Society.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, chapter 10, pages 220–242. Springer Berlin / Heidelberg, Berlin/Heidelberg, 1997.

[KMLBM08] Toma Kosar, Pablo E. Martinez López, Pablo A. Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 50(5):390–405, April 2008.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27:97–111, 1984.

[Kro03] John Krogstie. Evaluating UML using a generic quality framework. In Liliana Favre, editor, *UML and the Unified Process*, pages 1–22. IGI Publishing, Hershey, PA, USA, 2003.

[KRS82] Kai Koskimies, Kari-Jouko Räihä, and Matti Sarjakoski. Compiler construction using attribute grammars. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 153–159, New York, NY, USA, 1982. ACM.

[Kru03] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley, Boston, 3 edition, 2003.

[Kru08] Philippe Kruchten. What do software architects really do? *Journal of Systems and Software*, 81(12):2413–2416, 2008.

[KRW05] Douglas Kirk, Marc Roper, and Murray Wood. Identifying and addressing problems in framework reuse. In *Proceedings of 13th International Workshop on Program Comprehension*, IWPC'05, pages 77–86, 2005.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley-IEEE Computer Society, March 2008.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* The Addison-Wesley Object Technology Series. Addison-Wesley, 2003.

[LATaM09] Sérgio Lopes, Francisco Afonso, Adriano Tavares, and Jo ao Monteiro. Framework characteristics - A starting point for addressing reuse difficulties. In *Fourth International Conference on Software Engineering Advances*, ICSEA '09, pages 256–264, 2009.

[Leh80] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060 – 1076, September 1980.

[LEV10] Johan Laurenz Eveleens and Chris Verhoef. The rise and fall of the chaos report figures. *IEEE Software*, 27(1):30–36, January 2010.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 214–223, New York, NY, USA, 1986. ACM.

[Lin99] Kurt R. Linberg. Software developer perceptions about software project failure: a case study. *Journal of Systems and Software*, 49(2-3):177 – 192, 1999.

[LS75] Michael. E. Lesk and Eric Schmidt. Lex – a Lexical Analyzer Generator. Technical report, Bell Laboratories, 1975. CS Technical Report No. 39.

[LS80] Benent Lientz and Burton Swanson. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations.* Addison-Wesley, Reading (MA), 1980.

[LSR07] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[LST+06] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 191–204, New York, NY, USA, 2006. ACM.

[LV02] Juan de Lara and Hans Vangheluwe. Computer aided multiparadigm modelling to process Petri-nets and statecharts. In *Proceedings of the First International Conference on Graph Transformation*, ICGT '02, pages 239–253, London, UK, UK, 2002. Springer-Verlag.

[Mad10] James Madison. Agile architecture interactions. *IEEE Software*, 27(2):41–48, March 2010.

[MAP+08] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In Bertrand Meyer, Jerzy R. Nawrocki, and Bartosz

Walter, editors, *Balancing Agility and Formalism in Software Engineering*, pages 252–266. Springer-Verlag, Berlin, Heidelberg, 2008.

[MB97] Michael Mattsson and Jan Bosch. Framework composition: Problems, causes and solutions. In *Proceedings of the Tools-23: Technology of Object-Oriented Languages and Systems*, TOOLS '97, pages 203–214, Washington, DC, USA, 1997. IEEE Computer Society.

[MB02] Stephen Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Object Technology Series. Addison-Wesley, 2002.

[MBC09] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.P.L.O.T.: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 761–762, New York, NY, USA, 2009. ACM.

[McC96] Steve McConnell. *Rapid development: Taming wild software schedules*. Microsoft Press, 1996.

[McC04] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, 2004.

[McG08] John D. McGregor. Agile software product lines, deconstructed. *Journal of Object Technology*, 7(8):7–19, 2008.

[MD08] Parastoo Mohagheghi and Vegard Dehlen. Where is the proof? - A review of experiences from applying MDE in industry. In *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA '08, pages 432–443, Berlin, Heidelberg, 2008. Springer-Verlag.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[Mic07] Microsoft Ccorporation. C# standard, version 3.0. Technical report, 2007.

[MK96] Jeff Magee and Jeff Kramer. Self organising software architectures. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, ISAW '96, pages 35–38, New York, NY, USA, 1996. ACM.

[MM03] Joaquin Miller and Jishnu Mukerji. MDA Guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[MM10] Laurie McLeod and Stephen MacDonell. Stakeholder perceptions of software project outcomes: An industry case study. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 32:1–32:4, New York, NY, USA, 2010. ACM.

[MNSB06] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and complexity of XML Schema. *ACM Transactions on Database Systems*, 31(3):770–813, September 2006.

[MV12] Raphael Mannadiar and Hans Vangheluwe. Modular artifact synthesis from domain-specific models. *Innovations in Systems and Software Engineering*, 8(1):65–77, March 2012.

[Nat68] *NATO Software Engineering Conference*, 1968.

[NCM04] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 99–115, New York, NY, USA, 2004. ACM.

[Nie02] Oscar Nierstrasz. Software evolution as the key to productivity. In *Proceedings Radical Innovations of Software and Systems Engineering in the Future*, pages 274–282. Springer-Verlag, 2002.

[NJGG10] Uolevi Nikula, Christian Jurvanen, Orlena Gotel, and Donald C. Gause. Empirical validation of the classic change

curve on a software technology change project. *Information and Software Technology*, 52(6):680 – 696, 2010.

[OMG07] OMG. *XML Metadata Interchange (XMI)*. OMG, 2007.

[Opd92] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[OZ05] Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In *Proceedings of the 12th International Workshop on Foundations of Object-Oriented Languages*, FOOL 12, January 2005. `http://homepages.inf.ed.ac.uk/wadler/fool`.

[Paa95] Jukka Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[Par57] C. Northcote Parkinson. *Parkinson's Law Or The Pursuit of Progress*. Penguin business library. Penguin, 1957.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[PAS98] Wolfgang Pree, Egbert Althammer, and Hermann Sikora. Self-configuring components for client/server applications. In *Proceedings of 9th International Workshop on Database and Expert Systems*, pages 780–783, 1998.

[PEM03] Frauke Paetsch, Armin Eberlein, and Frank Maurer. Requirements engineering and agile software development. In *Proceedings of the Twelfth International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, WETICE '03, pages 308–313, Washington, DC, USA, 2003. IEEE Computer Society.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product line annotations with UML-F. In *Lecture Notes in Computer Science*, pages 188–197. Springer-Verlag, 2002.

[Phi94] J.J. Phillips. *In Action: Measuring Return on Investment*. In Action. American Society for Training and Development, 1994.

[Pig96] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Software Investment.* John Wiley & Sons, Inc., 1996.

[PJ98] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference*, COMPSAC '98, pages 9–15, Washington, DC, USA, 1998. IEEE Computer Society.

[PK00] Wolfgang Pree and Kai Koskimies. Framelets - small and loosely coupled frameworks. *ACM Computing Surveys*, 32(1es), March 2000.

[PKdWvV12] Eltjo Poort, Andrew Key, Peter de With, and Hans van Vliet. Issues dealing with non-functional requirements across the contractual divide. In *Proceedings of 2012 Joint Working Conference on Software Architecture and 6th European Conference on Software Architecture*, WICSA-ECSA.212.52, pages 315–319. IEEE, 2012.

[PP03] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[PP09] Mary Poppendieck and Tom Poppendieck. *Leading Lean Software Development: Results Are Not the Point.* Addison Wesley Signature Series. Addison-Wesley, 2009.

[PQ94] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25:789–810, 1994.

[Pre10] Roger Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2010.

[Ree79] Trygve Reenskaug. Models-views-controllers. *Technical note, Xerox PARC, December*, 1979.

[Roy70] Winston W. Royce. Managing the Development of Large Software Systems. In *Proceedings of IEEE Wescon*, pages 1–9, August 1970.

[RR10] Carmen M. Reinhart and Kenneth S. Rogoff. Growth in a time of debt. Working Paper 15639, National Bureau of Economic Research, January 2010.

[RV11] Henrique Rocha and Marco Tulio Valente. How annotations are used in Java: An empirical study. In *SEKE*, pages 426–431. Knowledge Systems Institute Graduate School, 2011.

[SAR12] Paula Savolainen, Jarmo J. Ahonen, and Ita Richardson. Software development project success and failure from the supplier's perspective: A systematic literature review. *International Journal of Project Management*, 30(4):458 – 469, 2012.

[SB00] Manuel Serrano and Hans-Jürgen Böhm. Understanding memory allocation of scheme programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 245–256, New York, NY, USA, 2000. ACM.

[SB05] Tilman Seifert and Gerd Beneken. *Model-Driven Software Development*. Springer, 2005.

[SBKM09] Juha Savolainen, Jan Bosch, Juha Kuusela, and Tomi Männistö. Default values for improved product line management. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 51–60, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[Sch90] Douglas C. Schmidt. GPERF: a perfect hash function generator. In *USENIX C++ conference proceedings: C++ Conference, San Francisco, California*, pages 87–102, 1990.

[Sch95] Ken Schwaber. SCRUM development process. In *Proceedings of OOPSLA'95 Workshop on Business Object Design and Implementation*, pages 117–134, 1995.

[Sch05] Stephen R. Schach. *Classical and Object-Oriented Software Engineering*. McGraw-Hill, 6 edition, 2005.

[SEI13] SEI, Software Engineering Institute. Published software architecture definitions, May 2013. sei.cmu.edu/architecture/start/glossary/published.cfm.

[Sir09] John Siracusa. Mac OS X 10.6 Snow Leopard: the Ars Technica review. 2009. http://arstechnica.com/apple/2009/08/mac-os-x-10-6/6/.

[SKRS05] Stefan Sarstedt, Jens Kohlmeyer, Alexander Raschke, and Matthias Schneiderhan. A new approach to combine models and code in model driven development. In *International Conference on Software Engineering Research and Practice, International Workshop on Applications of UML/MDA to Software Systems*, SERP'05, 2005.

[SKV11] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. A pure embedding of attribute grammars. *Science of Computer Programming*, 2011.

[SLB+10] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The variability model of the Linux kernel. In *Proceedings of 4th International Workshop on Variability Modelling of Software-intensive Systems*, VAMOS'10, Linz, Austria, January 2010.

[SLTM91] Kari Smolander, Kalle Lyytinen, Veli-Pekka Tahvanainen, and Pentti Marttiin. MetaEdit – A flexible graphical environment for methodology modelling. In *Advanced Information Systems Engineering*, pages 168–193. 1991.

[Som10] Ian Sommerville. *Software Engineering.* Addison-Wesley, Harlow, England, 9. edition, 2010.

[SPL03] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems Software Technologies, Engineering Processes, and Business Practices.* Addison-Wesley, 2003.

[Spo04] Joel Spolsky. *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune Or Ill Luck, Work with Them in Some Capacity.* Apress Series. Apress, 2004.

[SS03] Raul Silaghi and Alfred Strohmeier. Better generative programming with generic aspects. In *OOPSLA 2003 Workshop in Generative Techniques in the Context of Model Driven Architecture*, 2003.

[SS12] Jeff Sutherland and Ken Schwaber. The Scrum papers: Nuts, Bolts, and Origins of an Agile Process, Version 1.1, April 2012.

[Sun04] Sun Microsystems. Annotation Processing Tool (apt), 2004.

[SW11] Richard M. Stallman and Zachary Weinberg. *The C Preprocessor for GCC version 4.7.2*, 2011.

[Swa76] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[Swe85] Richard E. Sweet. The Mesa programming environment. *SIGPLAN Notices*, 20(7):216–229, June 1985.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[TB03] Dave Thomas and Brian M. Barry. Model driven development: The case for domain oriented programming. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 2–7, New York, NY, USA, 2003. ACM.

[TBMM04] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures. World Wide Web Consortium, Recommendation REC-xmlschema-1-20041028, October 2004.

[TLB+09] Thomas Tan, Qi Li, Barry W. Boehm, Ye Yang, Mei He, and Ramin Moazeni. Productivity trends in incremental and iterative software development. In *Proceedings of 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM'09, pages 1–10, 2009.

[TMY+09] Masateru Tsunoda, Akito Monden, Hiroshi Yadohisa, Nahomi Kikuchi, and Kenichi Matsumoto. Software development productivity of Japanese enterprise applications. *Information Technology and Management*, 10(4):193–205, December 2009.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM.

[TSL11] David Tilson, Carsten Sorensen, and Kalle Lyytinen. The paradoxes of change and control in digital infrastructures: The mobile operating systems case. In *Proceedings of the 2011 10th International Conference on Mobile Business*, ICMB '11, pages 26–35, Washington, DC, USA, 2011. IEEE Computer Society.

[vD99] Arie van Deursen. Software renovation. *ERCIM News*, 36, 1999.

[Vli98] John Vlissides. *Pattern hatching: design patterns applied.* Addison-Wesley Longman Ltd., Essex, UK, 1998.

[VS04] Rini Van Solingen. Measuring the ROI of software process improvement. *IEEE Software*, 21(3):32–38, 2004.

[VS06] Valentino Vraniae and Jan Snirc. Integrating feature modeling into UML. In Robert Hirschfeld et al., editors, *Proceedings of NODe 2006*, LNI P-88, pages 3–15, Erfurt, Germany, September 2006. GI.

[Wai93] William M. Waite. Beyond lex and yacc: How to generate the whole compiler. Technical report, University of Colorado, 1993.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, second edition, 2003.

[Wol03] Stephen Wolfram. *The Mathematica book (5. ed.).* Wolfram-Media, 2003.

[YGF08] Alexander Yermolovich, Andreas Gal, and Michael Franz. Portable execution of legacy binaries on the Java virtual machine. In *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, PPPJ '08, pages 63–72, New York, NY, USA, 2008. ACM.

[ZHK07] Weishan Zhang, Dong Han, and Thomas Kunz. Object-orientation is evil to mobile game: Experience from industrial mobile RPGs. In Yann-Hang Lee, Heung-Nam Kim, Jong Kim, Yongwan Park, Laurence Tianruo Yang, and Sung Won Kim, editors, *ICESS*, volume 4523 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2007.

[ZHKH07] Weishan Zhang, Dong Han, Thomas Kunz, and Klaus Marius Hansen. Mobile game development: Object-orientation or not. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 601 –608, July 2007.

Reprints of original publications