On the design & preservation of software systems

van Gurp, Jilles

CHAPTER 1 *Introduction*

It is hard to imagine that only sixty years ago there were no computers. Yet, nowadays we are surrounded by computers. Nearly anything equipped with a power cord most likely also contains a microchip. These chips can be found in, for example, kitchen appliances, consumer electronics, desktop PCs, cars, (mobile) phones, PDAs. Consequently, ordinary things like making a phone call, heating a lasagna or booking a plane ticket involve the use of computers and software. Life as we know it today would be substantially different without computers and software.

The quick adoption of computer technology in the last half of the previous century was stimulated by the exponential increases in speed and capacity of computer chips. In 1965, Gordon Moore, co-founder of Intel (a leading manufacturer of micro chips), observed that engineers managed to double the amount of components that could be put on a microchip roughly every 18 months without increasing the cost of such chips. [Moore 1965]. This phenomenon was dubbed Moore's law by journalists. Microchips have doubled in capacity every 18 months ever since Moore observed this phenomenon. In 1965 chips typically had about 50 components. Today, a low-end Intel Pentium 4 processor contains approximately 42 million transistors. It is widely expected that Moore's law will continue to be applicable for at least another two decades.

The software running on these chips has also increased in size and complexity. Bill Gates, co-founder of Microsoft, is famous for allegedly having said once that "640 kilo-byte ought to be enough for anyone" (640 kilo-byte was the maximum amount of memory that MS DOS could use) [@640kb]. Currently, Microsoft recommends 128 mega-byte as the minimum amount of memory needed to run their Windows XP operating system.

In an article from 1996 [Rooijmans et al. 1996], the authors discuss the increase of software system size in Philips consumer products such as for, example, TVs. In the late nineteen eighties, such devices were typically equipped with less than 64 kilo byte of memory allowing for a limited amount of software. By 1996, the typical amount of memory had grown to more than 500 kilo byte. Also, the authors estimate the average size of software for such chips measured in lines of code (LOC) to be around 100.000 LOC, at that time.

The growth in memory size can be explained by Moore's law and according to this law the typical memory size in such devices should be several mega bytes by now (this is confirmed in a later study by [Van Ommering 2002]). The size of the software in these devices has grown in a similar fashion. In the late nineteen eighties, Philips typically assigned a handful of electrical

engineers to write the software for their consumer electronics. By 2000, [Van Ommering 2002] estimates that about 200 person years are needed to write software for a typical high-end TV.

This trend is likely to continue throughout the coming decades. The exponential growth of software systems has an effect on the way software systems are manufactured. Both processes and techniques are needed to do so effectively. The research field that studies these processes and techniques is generally referred to as software engineering.

Because of the omnipresence of computer hardware and software, software engineering has evolved into an engineering discipline that is just as important for society as other engineering disciplines such as for example electrical engineering, civil engineering and aeronautical engineering. Software has become a key factor in all sectors of our economy. The transport sector, financial world, telecommunication infrastructure and other sectors would all come to a grinding halt without software. The mere thought of their software failing motivated industries worldwide to invest billions of dollars to avoid being affected by the Y2K problem. Luckily, most software turned out to be robust enough to survive the turn of the millennium. However, the amount of money spent on the prevention (approximately 10 billion dollars in the Netherlands [@NRC]) of the potential catastrophe is illustrative of how important software and the development of software has become for society.

This thesis consists of a number of articles that make contributions to the research field of software engineering. This introduction places these contributions in the context of a number of predominant software engineering trends and present a set of research questions that can be answered using these results. The conclusion chapter at the end of this thesis will answer these questions using the results presented in the articles.

First, in Section 1 of this introduction we introduce the field of Software Engineering and give a brief overview of relevant topics and issues within this field. In Section 2 we highlight a few trends that form the context for our research. Research questions in the context of these trends are listed in Section 3. Finally, we explain our research method in Section 4. In Chapter 2, an overview of the articles in this thesis is given. The remaining chapters present the articles and finally, in Chapter 11, our conclusions are presented.

1 Software Engineering

The term *software engineering* was first coined at a NATO conference in 1968 [Naur & Randell 1969]. At this conference, attendees were discussing the so-called software crisis: as a result of ever progressing technology, software was becoming more and more complex and thus increasingly difficult to manage. As a solution to this crisis, it was suggested that engineering principles should be adopted in order to professionalize the development of software by applying the engineering practices that had been successful in other fields. In line with this vision, the IEEE (Institute of Electrical and Electronics Engineers) currently has the following standard definition for software engineering: *(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1)* [IEEE610 1990].

Whether there ever was a software crisis and whether there still is a software crisis remains topic of debate. A few years ago, some of the attendees of the original NATO conference (among others Naur and Randell) in 1968 discussed this topic in a workshop [@Brennecke et al. 1996]. Although the debate was inconclusive, it should be noted that so far, software engineering practice has kept pace with the increase in hardware capacity. Ever larger teams of

software engineers build larger and more complex software. Elaborate techniques, best practices and methodology, help increase productivity and effectivity.

Our position in the debate regarding the software crisis is that rather than being in a constant software crisis, we are continually pushing the limit of what is possible. In order to be able to take advantage of hardware innovations, the practice of software engineering needs to evolve in such a way that we can do so cost effectively. In the thirty five years since the NATO conference the field of software engineering has evolved and matured substantially. New software development techniques and methods have been proposed (for examples, see Section 1.1) and subsequently adopted in the daily practice of developing software. Arguably, the state of the art in software engineering today allows us to build better, larger, more complex, more feature rich software than was possible in 1968.

In the remainder of this section, we will present an overview of some of the important research topics in the field of software engineering. An exhaustive overview would be beyond the scope of this thesis so we will limit ourselves to topics that are relevant in the context of this thesis.

1.1 Software Methodology

In order to make groups of software engineers work together efficiently to build a software product, a systematic way of working needs to be adopted [IEEE610 1990]. In [Rooijmans et al. 1996], the authors describe how before 1988 the development of embedded software for TVs had no visible software process since the implementation of this software was done by only two individuals. However, as the software became more complex, the need for a software process became apparent since, as the authors state in their article, "The need for the process's visibility throughout the organization emerged when software development became every project's critical path".

By 1993, the organization responsible for developing the TV software was certified as CMM (Capability Maturity Model) level 2. The CMM is a classification system for development processes that is often used to assess how mature the software development process in an organization is. It was created by the SEI which is a US government funded research institute [Paulk et al. 1993]. The CMM has five levels: initial, repeatable, defined, managed, optimizing. Very few organizations are certified as level 5 (optimizing) [@SEI CMM]. Organizations certified as level 2 (such as the organization described in [Rooijmans et al. 1996]) can develop software in a repeatable way. That means that given similar requirements and circumstances, software can be developed at a predictable cost according to a predictable schedule.

A wide variety of software development methodologies exists. Most of these methods are based on, or derived from the waterfall model proposed in [Royce 1970]. The waterfall model of software development divides the development process into a number of phases (see Figure 1). In each of these phases documents are produced which serve as input for the next phase. For example, during software requirements, a specification document is created. Based on the information in this document, an analysis document is created in the next phase.

Derivatives of the waterfall model often use different names for these phases or group phases or introduce new ones. In Figure 1, both the original waterfall model by Royce and the phases of the version we use in our own work are presented (see e.g. [Chapter 7] and [Bosch 2000] for discussions of such software methodologies).

Although the waterfall model is mostly interpreted as a purely sequential model (i.e. the phases are executed one after the other), Royce did foresee iterative applications of it and

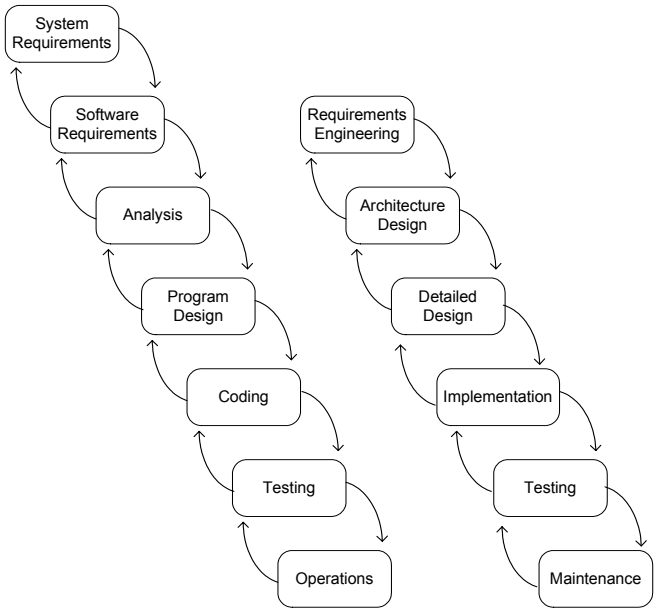

FIGURE 1. **The Waterfall Model. On the left is the model Royce defined in 1970, on the right is the version we mostly use in our work.**

even recommended going through some phases twice to get what he called "an early simulation of the final product".

Iterative methodologies have proven popular. An example of an iterative development methodology is the spiral model by [Boehm 1988]. It uses four phases, which are iterated until a satisfactory product has been created. The phases in this model are:

- Determine Objectives. During this phase, the objectives for the iteration are set.
- Risk Assessment. For each of the identified risks, an analysis is done and measures are taken to reduce the risk.
- Engineering/Development. During this phase, the development is done.
- Plan Next phase. Based on e.g. an assessment, the next iteration is planned

The Spiral model still goes sequentially through all phases, however. Each iteration effectively is one phase of the waterfall model. There also exist methodologies, which iterate over multiple or even all waterfall phases. Such methodologies are usually referred to as evolutionary. Examples of evolutionary methods that are currently popular are Extreme Programming [Beck 1999], Agile development [Cockburn 2002] and the Rational Unified Process [@Rup]. Especially agile development is known to have short development iterations (typically in the order of a few weeks) during which requirements are collected, designs are created, code is written and tested.

Specific methods for each waterfall model phase also exist such as Catalysis [D'Souza & Wills 1999] and OORAM (Object Oriented Role Analysis and Modeling) [Reenskaug 1996] that are intended for what we call the detailed design in Figure 1 (strictly speaking they also cover the rest of the waterfall model but the focus is mostly on the detailed design phase). SAAM (Software Architecture Analysis Method) [Kazman et al. 1994] is an example of an analysis method that can be used during the architecture design phase. Another example of a method that can

be used during this phase is the ATAM (Architecture Trade-off Analysis Method) [Clements et al 2002.]

Each of the phases in the waterfall model has its own (multiple) associated research fields, practices, tools and technologies. Highlighting all of these would be well beyond the scope of this introduction. For that kind of information we refer the reader to the many textbooks that were written on this subject: e.g. [Sommerville 2001][Van Vliet 2000].

1.2 Components

Along with the desire to apply engineering principles to software manufacturing also came the need to be able to breakdown software into manageable parts (i.e. components). The NATO conference in 1968 that is seen by many as the birth of the software engineering as a research field also resulted in the first documented use of the word *software component* [McIlroy 1969]. Similar to e.g. electrical components or building materials, the idea was that it should be possible to create software components according to some specification and subsequently construct a software product by composing various software components [Szyperski 1997].

Various component techniques such as Microsoft's COM [@Microsoft], CORBA [@OMG] and Sun Micro System's JavaBeans [@JavaBeans], which were all created during the last decade, have increased interest in Commercial Off The Shelf (COTS) components and Component Based Software Engineering (CBSE) [Brown & Wallnau 1999]. These techniques provide infrastructure to create and use components and form the backbone of most large software systems.

While these techniques are increasingly popular as a development platform for creating large, distributed software systems [Boehm & Sullivan 2000], they have failed to create a market for reusable COTS components based on these techniques [@Lang 2001]. Aside from niche markets such as the market for visual basic components (based on the COM standard), these techniques are generally used as a platform and not to create reusable components for third parties [Wallnau et al. 2002]. Each of these three component techniques has an associated set of software libraries and components that together form a platform that software developers use to create applications. Examples of such commercial component platforms are the .Net platform, which was recently introduced by Microsoft or the Java 2 platform.

The failure of component techniques such as CORBA or COM to create a COTS market of CORBA/COM components does not mean that there are no COTS components at all. In [Wallnau et al. 2002], the authors estimate that as many as 2000 new components per month were inserted in the market in 1996. Probably this number is much higher now. The market for COTS components is mostly focused on larger components such as e.g. operating systems, data bases, messaging servers, transaction servers, CASE (computer aided software engineering) tools, application frameworks and class libraries rather than specific COM/CORBA components [Boehm & Sullivan 2000]. Especially in the enterprise application market, there is a wide variety of such infrastructure components available.

In [Boehm & Sullivan 2000], a strong case is made for the thesis that software economics more or less force developers to use large COTS components such as described above. Given the limited resources available, the time to market pressure and the competition with other software developing organizations, large COTS components are the only way to incorporate needed functionality. However there are still a few issues with COTS.

An important issue with components is the definition. Despite the many publications on software components, there is little consensus. A wide variety of definitions exists. Ironically, the

actual manifestation of COTS components described above, is well outside most commonly used definitions of what a component is. The Webster dictionary definition of a component is that of "a constituent part" (this of course also covers COTS components). However, in the context of software components there are usually additional properties and characteristics associated with components. Some definitions, for example, state that a component must have required interfaces (e.g. [Olafsson & Bryan 1996]); others insist on specification of pre and post conditions. Popular component techniques such as COM or CORBA, for instance, lack required interfaces and with infrastructure components such as described above it is even harder to distinguish between provided and required interfaces.

A definition that appears to represent some consensus in this matter is the following by [Szyperski 1997]: *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*. Even this definition is too strict to cover for instance operating systems or database systems. However, it does at least cover components that conform to e.g. COM or CORBA.

Another issue with the use of COTS is that the development process needs to be adjusted to deal with the selection, deployment, integration and testing of the COTS components. An extensive study by the NASA Software Engineering Laboratory, suggests that there are a number of problems with respect to cost-benefit here [NASA SEL 1998]. In [Wallnau et al. 2002], processes and approaches are discussed that may address such issues.

Finally, there is a growing consensus that in addition to specifying functionality of components, it is also important to specify non-functional attributes of component (e.g. performance, real-time behavior or security aspects) [Crnkovic et al. 2001]. While the importance of such specification is recognized, it is unclear how to create such specifications and what exactly needs to be specified [Crnkovic et al. 2001].

1.3 Software Architecture

Over the past few years, the attention has shifted from engineering software components towards engineering the overall structure of which the components are a part, i.e. the architecture. Increasingly the focus has shifted from reusing individual components and source code to reusing software designs and software architectures. If two systems have the same architecture, it is easier to use the same software components in both systems. One of the lessons learned from using components over the past few years is that without such a common infrastructure, it is hard to combine and use components [Wallnau et al. 2002][Bosch et al. 1999]. Components tend to have dependencies on other components and make assumptions about the environment in which they are deployed.

Similar to the term software engineering, the concept of software architecture was inspired by the terminology of another discipline: architecture. In [Perry & Wolf 1992], the authors try to establish software architecture as a separate discipline by laying out the foundations of the research field of software architecture by identifying research issues and providing a framework of terminology. Their work also includes a definition: *software architecture = {elements, form, rationale}.* However, despite this definition, there is little consensus within the software engineering community about what exactly comprises a software architecture. The Software Engineering Institute (SEI) at the Carnegie Mellon, maintains a list of software architecture definitions which is very extensive and includes textbook definitions, definitions taken from various articles about software architecture and even reader contributed definitions [@SEI software architecture]. The one thing that can be learned from this list is that there is a wide variety of mostly incompatible definitions.

However, since a few years there is an IEEE definition that appears to be increasingly popular: *Architecture: the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution* [IEEE1471 2000].

With this definition in place several research area's have emerged that focus on for example:

- The modeling of architectures (e.g. XADL [@xadl 2.0], Koala [Van Ommering 2002], UML [@OMG]).
- Assessing the quality of architectures (e.g. ATAM [Kazman et al. 1998] and ALMA [Bengtsson et al 2002]).
- Processes for creating architectures (see Section 1.1 for examples)
- Creating reusable architectures for software product lines (e.g. [Weiss & Lai 1999], [Jazayeri et al. 2000], [Clements & Northrop 2002], [Bosch 2000] and [Donohoe 2000]).

It would be beyond the scope of this introduction to highlight all these area's here, however. Where applicable, the chapters in this thesis elaborate on these issues.

In the mid-nineties, the interest in software architecture in both the software development community and the academic software engineering research community was accelerated by several publications on design patterns [Gamma et al. 1995] and architecture styles [Buschmann et al. 1996]. These patterns and styles present various design solutions and their rationale. These patterns and styles form a body of knowledge that software architects can use to build and communicate design solutions. Rather than presenting a concrete architecture, such patterns and styles communicate reusable design solutions and their associated rationale.

1.4 Reuse & Object Orientation

Both software components and software architecture are often associated with object oriented programming and object oriented (OO) frameworks. Object Oriented programming was first introduced in the language Simula in 1967 [Holmevik 1994]. However, it was the Xerox Palo Alto Research Lab that picked up the idea of object orientation and developed it further, resulting in the first completely object oriented language: Smalltalk [Kay 1993]. The creators of Smalltalk made an effort to ensure that Smalltalk was entirely object oriented and subsequently used it to create the first graphical user interface (an invention which was later adopted by Apple) and many other interesting innovations [Kay 1993]. Later, other programming languages such as Java, C++ and Delphi further popularized the use of object orientation. Today many popular programming languages support object oriented concepts such as classes, objects and inheritance.

The strength and the key selling point of object orientation has always been that it is allegedly easy to reuse objects and classes. Whether the adoption of object oriented techniques actually improves reusability, remains a topic of debate, however. A distinction can be made between opportunistic and systematic reuse. When reusing opportunistically, the existing software is searched for reusable parts when they are needed but no specific plan to reuse parts exists when the parts are created [Wartik & Diaz 1992][Schmidt 1999].

Opportunistic reuse only works on a limited scale and typically does not allow for reuse across organizations or domains [Schmidt 1999]. For that a more systematic approach to reuse is needed [Griss 1999]. Object oriented frameworks provide such an approach. Object oriented frameworks were invented along with object oriented programming. When Kay et al. developed Smalltalk, an elaborate framework of classes and objects was included with the compiler.

However, it was not until the late 1980s until the usefulness of object oriented frameworks was recognized and popularized. The use of OO frameworks was popularized by publications such as [Johnson & Foote 1988], [Roberts & Johnson 1996] and [@Nemirovsky 1997].

An OO framework is an abstract design consisting of abstract classes for each component [Bosch et al. 1999] and possibly a number of component classes that hook into this abstract design. OO frameworks can be seen as an example of a *software architecture*. In [Roberts & Johnson 1996] an approach is suggested to use object oriented frameworks to reuse both implementation code and design.

Nemirovsky makes a distinction between application frameworks, support frameworks and domain specific frameworks, depending on the type of functionality they support. Application frameworks typically provide reusable classes for common functionality such as user interfaces, database and file system access, etc; support frameworks encapsulate reusable functionality for e.g. working with sound or 3D graphics hardware and domain specific frameworks provide reusable design and functionality for applications within specific, usually industry specific domains.

Object oriented frameworks have proven a successful way of reusing functionality and development environments such as Java [@Javasoft] and .Net [@Microsoft] are commonly bundled with application and support frameworks. Building domain specific frameworks, however, has proven much harder.

There are several issues that make this hard:

- Several (at least three) applications need to be created to find out what they have in common [Roberts & Johnson 1996]. For many companies that presents a chicken egg problem: they need a framework to build applications and need to build applications before they can build a framework.
- Integration and composition issues with legacy software make it hard to adopt a third party, domain specific framework that was not explicitly designed to work together with the legacy software [Bosch et al. 1999].
- Evolution of object oriented frameworks triggers evolution of derived applications. Consequently, domain specific frameworks are more resistant to changes because any changes would require subtantial changes in derived frameworks and applications.

These issues also apply to application and support frameworks. However, the (typically) larger userbase of these types of frameworks makes it more attractive to build them.

1.5 Summary

In this section, we gave a brief overview of some important topics in Software Engineering. We discussed the influence the 1968 NATO conference has had on the field, identifying both the need for the application of engineering principles to software manufacturing and the need for software components. In addition, we highlighted how effective use of components requires some sort of software architecture (for example an object oriented framework). These research fields form the background for the work presented in this thesis.

2 Trends & Motivation

The work presented in this thesis is motivated by a number of software engineering trends that we have observed. In this section, we will highlight these trends.

2.1 More, larger and more complex software

Because of hardware developments, the average size of software is increasing exponentially. On top of that, more software systems are needed because the amount of devices that is equipped with computer hardware is increasing [Moore 1965].

As outlined earlier, Moore's law has enabled exponential growth of hardware capacity over the past few decades and is very likely to continue to do so for at least the next few decades. This has two consequences for software engineering:

- The growing hardware capacity makes it possible to run larger and more complex software. Embedded hardware, for instance, has traditionally been seen as relatively limited compared to hardware in e.g. desktop PCs or mainframe computers. Embedded hardware typically has limited memory capacity and limited performance. However, embedded hardware is also subject to Moore's law and the hardware now shipping in e.g. TV's or mobile phone's compares quite favorably to consumer PC's of only a few years ago in terms of performance [Rooijmans et al. 1996][Van Ommering 2002].
- Because of the low cost of computer chips, they are mass-produced and deployed. A car for instance has over 50 microprocessors. Consequently, not only is the software on these chips becoming larger and more complex but also more software is needed.

2.2 Commoditization

In order to cope with the increasing demand for software (see Section 2.1), an increasing amount of commercial of the shelf COTS hardware and software is used [Wallnau et al. 2002].

In the past few years we have worked with various industrial partners such as Axis AB (Sweden), Thales Naval BV and Philips Medical BV (The Netherlands). These companies all build embedded software systems. Nowadays, they use off the shelf hardware and software components as well as internally manufactured hardware and software components. However, in the seventies and eighties, all or most of the components in their products, including hardware, operating systems, etc., were proprietary. Because of exponential growth of hardware and software (also see Section 2.1), this has become increasingly infeasible and all of these companies have since adopted third party hardware and software components.

What has happened, and continues to happen, at Axis, Philips, Thales and almost any software developing organization is that proprietary components such as hardware and software are replaced by off the shelf components as they become more common. The reason for this is simple: these common components no longer represent the value added to whatever product is being manufactured. Therefore, it is more cost effective to outsource the development of such commodity components to specialized third parties and focus on the parts of the product where the added value of the product is created. This has the following consequences:

- Standardization. Once third parties start selling commoditized software to multiple clients that previously developed similar software internally, these commoditized components become industry standards.
- Accumulation of commodity software. There is an ever-growing amount of commodity software that software developers can use to create new software products. Much of the recently popularized open source software falls into this category.

- The special purpose software of today may become a commodity tomorrow.

2.3 Variability

Because software is gradually becoming more complex and larger, it takes more time to develop it. However, at the same time there is a pressure to deliver software early in order to meet time to market demands. This contradiction leads to a situation where existing software must be reused in order to be able to meet time to market demands. Writing all needed software from scratch simply takes to long. Also when writing new software, future reuse of this software is already taken into account.

Unfortunately, reusing special purpose software is hard and usually adaptations to the software are needed because the requirements are slightly different. By anticipating such differences in requirements and building in variability into their software, developers can facilitate the future reuse of their software. Variability can take many forms and there are many ways to implement it in software. Some examples of techniques that can be used to offer variability are the use of user configurable parameters, plugin mechanisms and generative programming [Czarnecki & Eisenecker 2000].

With respect to variability a few trends can be observed

- Variability that used to be handled in hardware (e.g. using dip switches, different hardware components, etc.), is increasingly handled in software. For example, some modern mobile phones can adapt to different mobile networks (e.g. GSM and CDMA).
- Increasingly, variability is moved to the run-time level to provide end-users of the software more flexibility. For example, some mobile phones run a small Java virtual machine so that users can download new features to their phones. Older phones do not have this ability and users generally need to replace their phones when new features are needed.

2.4 Erosion

As pointed out earlier, the improvements in technology make it possible to make larger software systems. So, increasingly the investment represented by these software products is getting larger as well. The consequence of this is that because software represents a significant investment, companies will be reluctant to abandon it when new requirements come along.

Large software systems tend to have long life cycles, sometimes decades, during which new requirements are imposed on the system and adaptive maintenance is performed on the system. A phenomenon we have observed and report on in this thesis is that of design erosion (see Chapter 9).

The many small adaptations that are made to a software system have a cumulative effect on the system and over time, the changes may be quite dramatic even if all the individual changes are small. At any point during the evolution of a software system, such changes are taken in the context of all previous changes (i.e. the system as it is at that moment in time) and expectations about possible future changes that may need to be made. It is inevitable that errors of judgment are made with respect to future requirements. Consequently, the system may evolve in a direction where it is hard to make necessary adjustments. The larger a system and the longer it lives, the harder it is to detect such eroding changes.

Empirical evidence for this phenomenon is provided in [Eick et al. 2001]. In this work, the authors present a statistical analysis of change management data of a large telecommunication system. One of the important conclusions of the authors is that *"code decay is a generic*

phenomenon". However, erosion of software was identified much earlier by for instance [Perry & Wolf 1992] who speak of architectural drift and erosion in their paper on software architecture. Later, [Parnas 1994] speaks of software aging and compares software aging to aging of humans. An interesting point that he identifies with this analogy is that, like human aging, software aging cannot be stopped but that we can fight the symptoms to prolong the life.

A few trends can be observed with respect to design erosion:

- Fixing design erosion can be expensive. So expensive, in fact, that we have been able to find several examples of software or software components that were replaced rather than fixed.
- An eroded software system may become an obstacle for further development.

2.5 Summary

In this section, we have outlined a set of trends in the field of software engineering that together form the context for this thesis. Software engineers have to deal with ever-growing amounts of ever more complex software. Doing so requires that they use a growing amount of commoditized software components and focus their development efforts on adding value to those commoditized components. On top of that they need to add variability so that their efforts are not lost for future generations of their software product. Also, they need to keep an eye out for future changes and try to dodge the effects of design erosion.

3 Research Questions

As mentioned before, this thesis consists of a number of articles. Each of these articles of course has its own goals and research questions. The goal of this section is not to merely rephrase these goals and questions but instead to connect the articles by putting these articles in the context of more general research questions.

The overall research question that motivates this thesis is:

> Given the fact that new, potentially unexpected requirements will be imposed on a software system in the future, how can we prepare such a system for the necessary changes?

In addition to this main research question, three more research questions have been specified as well as a number of more detailed ones.

> RQ 1 How can we prepare an object oriented framework for future changes and make it as reusable as possible?
>
> > RQ 1.1 What exactly is an OO framework?
> >
> > RQ 1.2 How can reusability of OO framework classes be improved?
> >
> > RQ 1.3 What are good practices for creating OO frameworks?
> >
> > RQ 1.4 How can we assess in an early stage whether a framework is designed well enough for its quality requirements?

RQ 2 Given expected (future) variations in a software system, how can we plan and incorporate the necessary techniques for facilitating these variations

RQ 2.1 What is variability and what kind of terminology can we use to describe variability?

RQ 2.2 How can variation points be identified?

RQ 2.3 What kinds of variability techniques are there and can they be organized in a taxonomy?

RQ 2.4 How can an appropriate variability technique be selected given a taxonomy such as in RQ 2.3?

RQ 3 Can design erosion be avoided or delayed?

RQ 3.1 What is design erosion and why does it occur?

RQ 3.2 Why do so many software projects suffer from the consequences of design erosion?

RQ 3.3 What type of design changes are the most damaging?

RQ 3.4 What can be done to limit the impact of such damaging changes?

The articles included in this thesis are organized into three parts, each bundling articles that are related to one of the three research questions.

4 Research Approach

Research in the field of software engineering is somewhat different from research in other fields of computer science. The main difference is the presence of the human factor. It does not suffice to just consider the technical side of a problem without considering how the problem affects the software engineering process; without considering the issues of how to integrate solutions to this problem in the practice of software engineering and without considering how to get people to agree that a particular technical solution is in fact a good solution.

In Figure 2, an overview is provided of our view of how software engineering relates to other fields. First of all, the problem domain is that of improving the practice of software manufacturing. Problems are identified by observing and analyzing how software is engineered in practice. Solutions to these problems consist of tools, techniques as well as methodology to apply them effectively. In order to provide such solutions, technical solutions from computer science research may be used. However, a mathematical proof of the correctness of these solutions is not enough to convince software practitioners to adopt such solutions. Software practitioners need to be convinced that a particular solution solves their problems, does not create new problems and that the solution is indeed feasible in their context. For this, research methodologies such as case studies, surveys and action research are more appropriate. This way of doing research is more common in empirical sciences such as sociology and psychology than it is in natural sciences such as computer science.

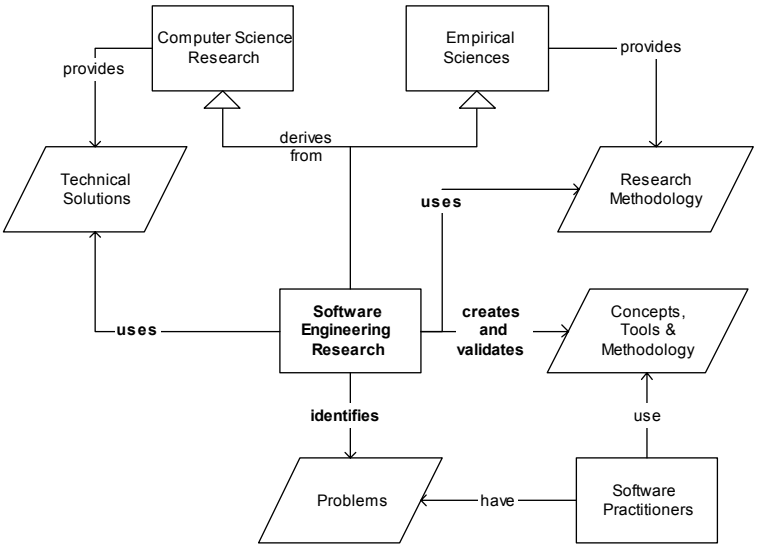

FIGURE 2. Software Engineering Research

The application of empirical research methods is increasingly popular in software engineering. In his editorial for the journal of empirical software engineering [Basili 1996], Victor Basily makes a plea for the use of empirical studies to validate theories and models that are the result of software engineering research. In a more recent publication [Basili et al. 2002], he gives an overview of how empirical research has benefited NASA's Software Engineering Lab.

In this thesis, we rely on our experience with several industrial cases for providing us with examples for our theories and for validating our approaches. By working together with industrial partners and by conducting surveys and interviews, we have learned a great deal about what problems software developing organizations encounter in practice and how it is affecting them. In the articles that comprise this thesis, we refer to these cases extensively and whenever possible we use examples from these cases.

When doing empirical research, a distinction can be made between qualitative empirical studies and quantitative studies. The latter type of studies is very useful for validating solutions to specific problems. However, when trying to establish what the issues are, such studies are less feasible because of a lack of quantifiable data. The approach, advocated by Basili in [Basili 1996] and [Basili et al. 2002], can be characterized as mostly quantitative. As can be seen in [Basili et al. 2002], collecting quantitative data is a labor intensive process that needs to be tightly integrated with the development process. In a setting like NASA, where reliable, dependable software is required this is feasible. The results of the quantitative empirical research are used to optimize the development processes.

Qualitative data, on the other hand, is relatively easy to obtain and has the advantage of providing more explanatory information [Seaman 1999]. As is noted in [Seaman 1999], neither quantitative nor qualitative empirical research can prove a given hypothesis. Empirical research can only be used to support or refute a hypothesis. A combination of both is the best way of supporting a hypothesis.

Most of our explorative case studies are of a qualitative nature. Over the past few years, we have been in contact with several industrial parthers who have cooperated with us by providing us access to internal documentation and by discussing their work with us.

Although industrial validation of new approaches is the best way to demonstrate their suitability, doing so is easier said than done. Industrial validation requires the cooperation of industrial partners, which poses inherent time and money constraints on a study. Consequently, we sometimes have to resort to the usage of smaller, non industrial cases such as, for instance the framework, described in Chapter 3, which was re-used in Chapter 9.

5 Summary & Remainder of this thesis

In this introduction, we have sketched how the research field of software engineering has developed over the past few decades. In addition, we introduced a number of software engineering topics and listed a number of predominant software engineering trends.

Due to the ever expanding capacity and proliferation of computer hardware, there is a constant pressure on the research field of software engineering to enable the creation of more, and larger software systems. Also, because software systems. Increasingly developers are resorting to Commercial Of The Shelf (COTS) hardware and software components to create software systems. Also to be able to reuse existing pieces of software effectively, variability techniques are adopted to make the software more versatile. Finally, due to the increasing economic value of these ever larger software systems represent, companies are increasingly reluctant to replace them with new and improved versions. However, many systems erode and become increasingly harder to maintain due to the cumulative effect of adaptations to new requirements.

We listed a number of research questions that fit in this context and presented a discussion of the research method that is used to answer those questions. The remainder of this thesis consists of an overview of the included articles (Chapter 2), eight articles organized into three parts. Finally, the research questions that were formulated in Section 3, are answered and some concluding marks are presented in Chapter 11.