TUCS

Espen Suenson

How Computer Programmers Work

Understanding Software Development in Practise

Turku Centre *for* Computer Science

TUCS Dissertations
No 197, June 2015

How Computer Programmers Work

Understanding Software Development in Practice

Espen Suenson

To be presented, with the permission of the Department of Information Technologies at Åbo Akademi, for public criticism on 12th June 2015 at 12 noon in auditorium Armfelt, Arken, Fabriksgatan 2, Åbo.

Åbo Akademi
Finland
2015

Advisors

Iván Porres
Professor of Software Engineering
Åbo Akademi

Anna-Maria Åström
Professor of Nordic Ethnology
Åbo Akademi

Patrick Sibelius
Lecturer in Computer Science
Åbo Akademi

Reviewers and opponents

Pekka Abrahamsson
Professor of Software Engineering
Norges teknisk-naturvitenskapelige universitet

Robert Willim
Associate Professor of European Ethnology
Lunds universitet

ISBN 978-952-12-3226-8
ISSN 1239-1883

TILL ETT FRITT FOLK
VAPAALLE KANSALLE

Contents

Sammanfattning på svenska **9**

Tiivistelmä suomeksi **11**

Acknowledgements **13**

1 Introduction **17**
1.1 Motivation 17
1.2 Definition of programming 20
1.3 Research focus 22
1.4 Research problem and results 24
1.5 Scope of the dissertation 27
1.6 Audience of the dissertation 28
1.7 Overview of the dissertation 28

2 Related research **31**
2.1 Programming research 31
2.2 Cultural research 35
2.2.1 Ethnology 36
2.2.2 Sociology, anthropology, and other 37

3 Theory **43**
3.1 Choice of theory 43
3.2 Mainstream programming theories 45
3.2.1 Software engineering 46
3.2.1.1 The origins of software engineering . 46

3.2.1.2 Brooks 47
3.2.1.3 Process thinking 52
3.2.1.4 Systems thinking 55
3.2.1.5 Requirements engineering 60
3.2.2 Agile software development 62
3.2.2.1 The origins of the Agile movement . 62
3.2.2.2 The Agile manifesto 64
3.2.2.3 Scrum as an example 66
3.2.2.4 Programming practice 69
3.2.2.5 Work ethics 71
3.2.2.6 Relationship to software engineering 74
3.2.3 Computer science 79
3.2.3.1 Science, mathematics, formal systems 79
3.2.3.2 Algorithms and data structures . . . 81
3.2.3.3 Abstraction 83
3.2.3.4 Machine-orientation 86
3.2.3.5 Premises of argumentation 87
3.2.3.6 Work processes 89
3.2.4 Object-Oriented Programming 92
3.3 Hermeneutical theory 95
3.3.1 Introduction 95
3.3.2 What is hermeneutics 97
3.3.3 Prejudice and reason 98
3.3.4 Authority 100
3.3.5 Understanding 101
3.3.6 Effective history 103
3.3.7 Horizons of understanding 105
3.3.8 Application 109
3.3.8.1 The application of history 112
3.3.9 Ethical and technical knowledge 114
3.4 Cultural form theory 117
3.5 Rhetorical theory 121

4 Method **123**

4.1 Choice of cases 124
4.2 Participant observation 125
4.2.1 Interdisciplinary concerns 126
4.3 Interviews 126
4.4 Ethnological method 127
4.5 Analysis methods 128
4.5.1 Mainstream analysis 128
4.5.2 Hermeneutical analysis 128
4.5.3 Cultural form analysis 129
4.5.3.1 Comparative analysis 129
4.5.3.2 Safety critical case studies 129
4.5.4 Rhetorical analysis 130
4.6 Theory of science 130
4.7 Data 131
4.7.1 Game programming 131
4.7.2 Safety critical programming 133

5 Game programming 139
5.1 Case study 139
5.1.1 Tribeflame 139
5.1.2 The development team 141
5.1.3 The rhythm of work 142
5.1.4 Playing and planning 148
5.2 Analysis 151
5.2.1 Analysis with mainstream theories 151
5.2.1.1 Software engineering 152
5.2.1.2 Agile development 155
5.2.1.3 Computer science 159
5.2.2 Hermeneutical analysis 161
5.2.2.1 The work process 161
5.2.2.2 Horizons of understanding 164
5.2.2.3 Authority and tradition 165
5.2.2.4 To have fun 168
5.2.2.5 Summary 170

6 Safety critical programming 173
6.1 Introduction to safety critical development 173
6.2 Case studies 175
6.2.1 A large avionics company 175
6.2.2 A small farming systems company 180
6.2.3 The safety critical standard 183
6.3 Hermeneutical analysis 188

7 Cultural form analysis 201
7.1 Comparative analysis 201
7.2 Safety critical case studies and analysis 211

8 Rhetorical case study and analysis 227
8.1 Rhetorics and programming 227
8.2 Audience 230
8.3 Programming conventions 232
8.4 Tone and purpose 236
8.5 Rhetorical understanding 243

9 Conclusion 245
9.1 Results 245
9.2 Discussion 248
9.2.1 Ethnological significance 255
9.3 Validity and generalization 256

Bibliography 259

A Method and Fieldwork in a Hermeneutical Perspective 275

Sammanfattning

Hur programmerare arbetar - Att förstå mjukvaruutveckling i praktiken

Hur arbetar en framgångsrik programmerare? Uppgifterna att programmera datorspel och att programmera industriella, säkerhetskritiska system verkar tämligen olika. Genom en noggrann empirisk undersökning jämför och kontrasterar avhandlingen dessa två former av programmering och visar att programmering innefattar mer än teknisk förmåga.

Med utgångspunkt i hermeneutisk och retorisk teori och med hjälp av både kulturvetenskap och datavetenskap visar avhandlingen att programmerarnas tradition och värderingar är grundläggande för deras arbete, och att båda sorter av programmering kan uppfattas och analyseras genom klassisk texttolkningstradition. Dessutom kan datorprogram betraktas och analyseras med hjälp av klassiska teorier om talproduktion i praktiken - program ses då i detta sammanhang som ett slags yttranden.

Allt som allt förespråkar avhandlingen en återkomst till vetenskapens grunder, vilka innebär en ständig och oupphörlig cyklisk rörelse mellan att erfara och att förstå. Detta står i kontrast till en reduktionistisk syn på vetenskapen, som skiljer skarpt mellan subjektivt och objektivt, och på så sätt utgår från möjligheten att uppnå fullständigt vetande.

Ofullständigt vetande är tolkandets och hermeneutikens domän. Syftet med avhandlingen är att med hjälp av exempel demonstrera programmeringens kulturella, hermeneutiska och retoriska natur.

Tiivistelmä

Kuinka tietokoneohjelmoijat työskentelevät – ohjelmiston kehittämisen ymmärtäminen käytännössä

Kuinka menestyvä ohjelmoija työskentelee? On melko erilaista ohjelmoida pelejä ja teollisia, turvallisuuskriittisiä systeemejä. Huolellisen empiirisen tutkimuksen kautta tutkielma vertaa näitä kahta ohjelmoinnin muotoa ja osoittaa, kuinka ohjelmointi riippuu muustakin kuin teknisistä kyvyistä.

Käyttämällä hermeneuttisia ja retorisia teorioita ja hyödyntämällä kulttuurintutkimusta sekä tietojenkäsittelyoppia tutkielma osoittaa, että ohjelmoijien perinteet ja arvot ovat keskeisiä heidän työssään ja että kummankin tyyppistä ohjelmointia voidaan tarkastella ja analysoida samalla tavalla kuin klassisen tietämyksen mukaan tekstejä tulkitaan. Lisäksi tietokoneohjelmia voidaan tarkastella ja analysoida samalla tavoin kuin klassista teoriaa puheen muodostamisesta käytännössä – tässä suhteessa ohjelmat nähdään puheen esimerkkeinä.

Yhteenvetona tutkielma kehottaa palaamaan tieteen ytimeen, joka on jatkuvaa ja loppumatonta kehämäistä prosessia kokemisen ja ymmärtämisen välillä. Tämä on vastakohtana reduktionistiselle näkemykselle tieteestä, joka erottaa tarkasti subjektiivisen ja objektiivisen ja siten olettaa, että on mahdollista saavuttaa täydellinen tieto.

Epätäydellinen tieto on tulkinnan ja hermeneutiikan aluetta. Tutkielman tavoitteena on esimerkin kautta osoittaa ohjelmoinnin kulttuurinen, hermeneuttinen ja retorinen luonne.

Acknowledgements

This research has been made possible through the generous financial support of Svenska kulturfonden i Finland, Svenska litteratursällskapet i Finland, Gustaf Packaléns mindefond, det Suensonske familielegat, and Turku Center for Computer Science.

I would like to thank my advisors from Åbo Akademi for their help with, and their engagement and interest, in my research: Anna-Maria Åström, professor in Nordic ethnology; Iván Porres, professor in software engineering; and Patrick Sibelius, lecturer in computer science.

I would like to thank my closest research collaborators – my parents Susanne (public health, Københavns Universitet) and Thomas Suenson (ethnology, Københavns Universitet) – who have been actively involved in the entire research process. The most important things I know, I have learned from them.

Thanks to my mentor Hans Axel Kristensen who taught me to figure out what I want to achieve. Thanks also to my friend and colleague Ann-Helen Sund (ethnology, Åbo Akademi), who has been a great help during the writing process, and to Jeanette Heidenberg (software engineering, Åbo Akademi), with whom I had the pleasure to collaborate during data collection.

Heartfelt thanks to the many people who have helped me and encouraged me along the way, especially in the difficult first stages of my research. Among them are Jyrki Katajainen (computer science, Københavns Universitet), Fritz Henglein (computer science, Københavns Universitet), Thomas Højrup (ethnology, Københavns Universitet), Jukka Teuhola (computer science, Turun Yliopisto), Eija Karsten (management systems, Åbo Akademi), Markku Heikkilä (management systems, Åbo Akademi), Katariina Heikkilä (ethnology, Turun Yliopisto), Helena Ruotsala (ethnology, Turun Yliopisto), Thomas Geisnæs (philosophy, Københavns Universitet), Gry Søbye (ethnology, Københavns Universitet), and Jeppe Høst (ethnology, Københavns Universitet).

I would like also to express my gratitude to my friends from the student organization NEFA (Nordisk Etnologisk Folkloristisk Arbejdsgruppe). It was through their friendship that I came to find my way to Finland. Particular thanks to Karoliina Autere (ethnology, Turun Yliopisto), Päivi Salonen (ethnology, Turun Yliopisto), and Mikko Helminen (archaeology, Turun Yliopisto) – and to Jenny Viitanen (folkloristics, Turun Yliopisto), for helping me to find my home in Finland.

A very warm thanks to the owners and employees of Tribeflame, who took my observation of their work and the disturbance it caused in good spirit. Thanks to all the engineers and developers who allowed me to interview them in connection with the RECOMP research project. Thanks also to the programmers and IT professionals whose interviews have not been used in this research, but which were a great help nonetheless: Luka Milovanov, Henri Sara, Riku Mustila, and Olli Salli.

Thanks to the researchers and staff from the institutions I have been a part of at Åbo Akademi: The Department of Information Technologies, especially the Software Engineering Laboratory, and the Faculty of Arts, especially the Department of Nordic Ethnology.

Thanks to the participants in the ethnological research seminar at Åbo Akademi: Anna-Maria Åström, Ann-Helen Sund, Niklas Huldén, Anna-Liisa Kuczynski, Sanna Lillbroända-Annala, Katriina Siivonen, Ole Rud Nielsen, Kasper Westerlund, Camilla Hautala, Sonja Hagelstam, Marcus Lepola, and Evaristo López Ruano (cultural anthropology, Universitat de Barcelona).

Thanks to the participants in the weekly software research discussion group SEMPRE at the Department of Information Technologies, among whom are: Jeanette Heidenberg, Marta Olszewska (computer science, Åbo Akademi), Piia Hirkman (management systems, Åbo Akademi), Max Weijola (software engineering, Åbo Akademi), Tingting Lin (information systems, Turun Yliopisto), Henrik Nyman (management systems, Åbo Akademi), Samuel Rönnquist (computer science, Åbo Akademi), Johannes Eriksson (computer science, Åbo Akademi), and Natalia Díaz Rodríguez (computer engineering, Åbo Akademi).

Thanks to all of my programming colleagues and employers through the years, especially Casper Gjerris and my development team at NDS Denmark: Guus Oosterbaan, Nanna Glerup, Martin Lorensen, Søren Thomsen, Anders Petersen, and Kristian Knak. My practical experience with programming work has been crucial to my research.

Thanks to the ScrumPLoP community for inspiration, especially Dina Friis (computer science, Københavns Universitet) and Jim Coplien.

Thanks to all of my Finnish teachers, especially Jenni Laine and Pirjo Vuoriluoto.

I am grateful for all the dear friends and acquaintances I have made in Finland, each and every one of you. Thanks to Emilia Komulainen for showing me the *kokko* in Naantali at Juhannus. Thanks to Johanna Vikman & Matti Toivio for being like a family to me.

Thanks to my family for the love that we share – it is what gives my work meaning.

Above all, praise and thanks be to the Lord Almighty, God of our fathers, the Trinity, One in essence and Undivided. For Thine is the kingdom, and the power, and the glory, of the Father, and of the Son, and of the Holy Spirit, now and ever, and unto ages of ages. Amen.

Chapter 1

Introduction

1.1 *Motivation*

The rationale for this dissertation is to be found in a paradox in the scientific study of software and programming. The paradox is the difference between the view held within science on the current state of the art of software development, which tends to be a rather dismal and pessimistic view, and the obvious success and importance that software development enjoys in society. The dismal view is not new. Indeed, it seems to have been with us since the first days of programming research.

Carl H. Reynolds wrote in 1970 on the subject of computer programming management:

> "A common complaint among those who are charged with the responsibility of planning and managing computer program systems is that despite their best efforts the product is completed behind schedule, over budget, and below promised performance."[1]

[1]Reynolds 1970 p. 38.

Frederick P. Brooks, Jr. echoed these feelings in his 1975 book *The Mythical Man-Month*, one of the most well-known and influential texts on software engineering:

> "No scene from prehistory is quite so vivid as that of the mortal struggles of great beasts in the tar pits. ... Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems – few have met goals, schedules, and budgets."[2]

Another influential book on software development, *Peopleware*, was written by Tom DeMarco and Timothy Lister in 1987. These authors generally have a more optimistic approach to the state of the art of software development; yet, they open their book with a chapter heading stating, "Somewhere today, a project is failing". They explain:

> "There are probably a dozen or more accounts receivable projects underway as you read these words. And somewhere today, one of them is failing. Imagine that! A project requiring no real innovation is going down the tubes. Accounts receivable is a wheel that's been reinvented so often that many veteran developers could stumble through such projects with their eyes closed. Yet these efforts sometimes still manage to fail."[3]

A textbook in software engineering from 2000 simply starts its introduction with a list of "Software Engineering Failures", under headings such as "Interface misuse", "Late and over budget" and "Unnecessary complexity".[4] Søren Lauesen's 2002 book about software requirements puts it bluntly:

> "Most IT systems fail to meet expectations. They don't meet business goals and don't support users efficiently."[5]

[2]Brooks 1975 [1995] chp. 1, p. 4.
[3]DeMarco & Lister 1987 [1999] chp. 1, p. 3.
[4]Bruegge & Detoit 2000 [2010] chp. 1.1, p. 38.
[5]Lauesen 2002 back cover.

According to Hans van Vliet, in a 2008 textbook on software engineering, the "software crisis" first became apparent in the 1960s. But it is not over yet, as we still need:

> "Better methods and techniques for software development [that] may result in large financial savings, in more effective methods for software development, in systems that better fit user needs, in more reliable software systems, and thus in a more reliable environment in which those systems function."[6]

As we can see, the professions that study programming, and the software engineering profession in particular, do seem to take a negative view of software development, with a recurring chorus highlighting failure to meet schedule, budget, and user goals. This would be no cause for wonder if programming really were in a state of utter confusion and chaos. But how, then, do we account for the huge success of programming and computers more widely in society? After lamenting the current state of programming, van Vliet hints at the other side of the paradox:

> "On the positive side, it is imperative to point to the enormous progress that has been made since the 1960s. Software is ubiquitous and scores of trustworthy systems have been built. These range from small spreadsheet applications to typesetting systems, banking systems, Web browsers and the Space Shuttle software."[7]

Numbers alone demonstrate the enormous progress that has been made. For example, we can look at the development in the U.S.A., which is by most accounts the world leader in computer development. According to the U.S. Department of Commerce, the computer systems design and related services industry increased in value from 5 billion dollars in 1977 to 184 billion in 2010. This is an increase in

[6]van Vliet 2008 chp. 1, p. 5.
[7]Ibid.

share of GDP of 550%, and it means that the industry is now larger than the entire oil and gas industry.[8]

Our everyday experience chimes with the numbers in showing the success of computers and programming. Our lives are full of programmed devices, from the more obvious, such as mobile phones and personal computers, to the perhaps less conspicuous, such as washing machines, cars and airplanes. And these devices normally work satisfactory, by and large. This, then, is the paradox of the software crisis: even though computers and programming have been so hugely successful and have become such an integral part of daily life, the scientific study of programming talks about it in dire, almost dystopian, terms.

Where might we find the cause of the paradox of the software crisis? I believe that it has its roots in an insufficient understanding of the essential characteristics of programming. For example, when the IEEE Computer Society 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"[9] it expresses the hope that an engineering approach to software will result in better, cheaper, and faster development. But this dream has so far been elusive.

The obvious question then becomes: what kind of thinking should be employed in order to broaden the scientific understanding of programming? The answer offered here is cultural studies.

1.2 *Definition of programming*

The word program, or programme, comes from the Classical Greek ***prógramma*** (πρόγραμμα) which means "a public proclamation". It is derived from the verb ***grápho*** (γράφω), which means "write", and the

[8]U.S. Department of Commerce 2011: GDP by Industry.

[9]The Institute of Electrical and Electronics Engineers (IEEE) Standard Glossary of Software Engineering Terminology 1990 p. 67.

prefix *pró-*, which means "before", also in the sense of "forth" or "publicly". Thus, *prográpho* can mean either "write something before", at the head of a list, for example; or it can mean "give a public announcement".[10]

In the ordinary use of the word, a program is a sequence of steps that are carried out in order to achieve some purpose. An example is a program for an evening concert, where different musical pieces are performed by the musicians over the course of the evening. Another example is a training program meant to increase prowess in some sports disciplines, by specifying exercises that are to be carried out.

These examples are both of programs that are meant to be carried out by people. Programs can also be carried out by machines, if the machine is designed to be able to execute sequences of steps. We speak then of automation.[11] An example is the Jacquard attachment to the drawloom, which was developed during the period from 1725 to 1801.[12] Via a collection of punched cards, the machine could be programmed to produce a given woven pattern when the loom was operated; and the punched cards could be dismounted and saved for later, if the weaver wished to reuse a certain pattern.

It is not always possible to distinguish clearly between the use of a machine and the programming of said machine. In the case of the drawloom, before the introduction of the Jacquard attachment a so-called drawboy would perform the instructions contained on the punched cards. The drawboy would then in effect be executing the pattern program by operating the loom. The degree of automation before the Jacquard attachment was of course lower, but even with the Jacquard attachment the drawloom was not fully automated. Thus, automation and programmability is a matter of degree rather than a question of either/or.

This treatise is concerned with a certain type of machine – namely, computing machines. In computers, the possibilities of computation and those of automation are combined in a way that makes the use

[10] Liddell, Scott, and Jones' lexicon.

[11] See Latour 1987 p. 130 f for a discussion of the concept of an automaton.

[12] Wilson 1979 [1982] p. 62 f.

of computers very flexible. It is of course possible to have computing devices that are not automatons. The abacus and the slide rule are examples of this. On the other hand, the Jacquard loom is an example of an automaton that does not perform any computing.

In a computer the aspects of computation and automation enhance each other, although practical uses of computers will often be inclined more towards one or the other aspect. A spreadsheet application, for example, can be said to be a use of computers that is dominated by the computational aspect, while a microcontroller for a car brake can be said to be dominated by the automation aspect.

It can be difficult to distinguish between using the computer and programming it. One example is given by spreadsheet applications, where the normal use of the application, specifying computations across rows and columns, can be viewed as essentially a limited form of programming.

With these remarks on the meaning of the term "programming", we can define the concept in the sense that it will be used in this treatise:

Programming is the act of instructing computing machines.

The term "coding" is often used to refer to the specific part of programming that is the writing down of the instructions, and we use the word in this sense here. The term "software development" can be used to refer to programming as well as a wider range of activities that are necessary to programming, for example sales. We use the term in this sense here, to denote programming and the supporting activities that are necessary to it.

1.3 *Research focus*

The approach of this dissertation is to apply cultural theory to the study of programming. More specifically, the approach and methods of the field of ethnology are adopted, and the theories employed are hermeneutical theory, cultural form theory, and rhetorical theory.

The importance of human factors are recognized within software engineering and other programming research today.[13] There is not yet a consensus on how best to study people within programming. Different approaches coexist, such as sociology, psychology, anthropology, and ethnography. Ethnology is a an approach that has been applied only sparsely.

A principle of this dissertation is that studies of programming must be founded on empirical research: theoretical development is crucial but it must be based on empirical observation so as to remain connected to the realities of programming.

A related principle is to focus on the practice of programming – that is, to study the actions of programmers in connection with the goals they are trying to achieve. The concept of practice will be explained further in section 3.4 (Cultural form theory).

This dissertation focuses on programming as work. This is natural insofar as programming historically is a product of professional development, and furthermore the work perspective on programming is shared by most of the existing research on programming in the technical fields.

The approach taken in this treatise is to apply more general theories in order to understand the more specific phenomenon of programming. The most general theory employed is an Aristotelian theory of culture, mainly inspired by the Danish ethnologist Thomas Højrup.[14]

To examine more specifically the work practices of programming, the theories of hermeneutics and rhetorics are employed. Hermeneutics is a general theory of how understanding takes place, while rhetorics is a general theory of expression; that is, of utterances. Classically, hermeneutics and rhetorics were confined to the more narrow phenomena of text and speech, but I use them in their modern versions, in which the theories are formulated so as to apply generally to

[13]See e.g. Glass 2002 p. 11.

[14]See section 3.4 (Cultural form theory).

any cultural phenomenon. In this I follow the philosophical hermeneutics of Hans-Georg Gadamer, and the “new rhetorics” of Chaïm Perelman and Lucie Olbrechts-Tyteca.[15]

In short, this treatise is a comparative empirical study of work practices that has philosophical hermeneutics as its epistemological and ontological point of departure. I am studying not only the cultural forms of programming, but the way these forms perceive themselves and how these perceptions are manifested in practice. My claim is that hermeneutical theory in particular allows the scholar to perceive the phenomenon of programming more clearly.

1.4 *Research problem and results*

The aeronautics engineer Walter Vincenti, in his 1990 book *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*, provides an historical exposition of episodes in U.S. aeronautical engineering. Vincenti analyzes how engineering knowledge is created and used, and points to the central role of the design process in engineering. His analysis shows the necessity of learning about the subjective, human element of a problem before everyday engineering and design tasks can commence. In engineering terms, this kind of problem can be posed as the need to examine an ill-defined problem in order to identify needs and translate them into specifiable criteria.[16] The needs and criteria cannot be specified beforehand, they must follow in the course of the research. This is the kind of research problem that is dealt with here.

The overarching research problem is this:

How can the practice of computer programming best be understood?

[15]For hermeneutics, see section 3.3 (Hermeneutical theory); for rhetorics, see section 3.5 (Rhetorical theory).

[16]Vincenti 1990 [1993] p. 51.

This means to examine the different forms the practice of computer programming can take and to research how the empirical forms of programming can best be understood theoretically. In order to accomplish this the dissertation contains the following major parts:

- Case studies that present empirically different forms of software development: game development and safety critical development.
- A presentation of mainstream programming theory (software engineering, Agile thinking, and computer science) and an application of the mainstream theory to the case study of game programming that demonstrates that mainstream theory has some shortcomings in analysis of empirical processes.
- A presentation of cultural theory: hermeneutical theory, cultural form theory, and to a lesser degree rhetorical theory. The theory is applied to the case studies in a series of analyses.
- In the conclusion the results of the cultural theory analyses are summarized and the consequences for programming practice are discussed.

The overarching research problem is answered with the help of three more concrete and specific research goals:

1. To explain the software development process with the help of hermeneutical analysis, and to a lesser degree rhetorical analysis.
2. To compare different software development practices with the help of cultural form theory.
3. To explore the practical implications of a hermeneutical and rhetorical understanding of software development.

The results of the dissertation are as follows: regarding the first research goal, it is demonstrated how the mainstream programming theories fall short in explaining the case study. The hermeneutical analysis reveals a circular epistomological structure in the process (the hermeneutical circle) that makes it significantly easier to understand. Safety critical development is also analyzed hermeneutically, and it is revealed how important bureaucratic structures are to the development process. The essential concepts are identified in analyzing game programming ("'fun") and safety critical programming ("safety"). The rhetorical analysis of game programming shows how the hermeneutical-rhetorical approach can be used to analyze not only work processes, but also work artifacts (i.e. source code).

Regarding the second research goal, a comparative form analysis of game programming and safety critical programming shows that form analysis can identify the constraints that programming practices are subject to. This analysis also demonstrates that programming practice can not be understood properly without taking into account the context it is carried out in. A second comparative analysis looks at variations within the safety critical industry and finds that these variations can be explained by cultural form theory partly as different strategies for adapting to the same circumstances, partly as reactions to variations in the constraints that limit the practices.

The third research goal is addressed through a discussion of the results of the first and second goals. One important finding is the necessity of context in the study of software development. Another is the relationship between scientific theory and practice; it turns out that theory must always be based in practice. The most important point made in the dissertation is an argument against reductionism: not everything can be controlled, indeed, not everything can be measured; however, hermeneutics can be used to find order in that which may seem chaotic.

1.5 *Scope of the dissertation*

In order to make any research project manageable, it is important to define what will not be included. This treatise studies programming as work. There are of course other reasons to program: for example as a hobby, as art, for the sake of education, or for fun. Though these reasons may be important, we will not consider them here.

Furthermore, only professional programmers are studied in this treatise. This excludes professionals in other areas that are programming as a part of their work, but who themselves set the task for the programming. Professional programmers are thus considered to be those programmers whose programs are not meant to be used by themselves: that is, their task is set by another party, directly or indirectly.

There are a number of aspects of programming that will only be considered to the extent that they have an influence on the work aspect, even though they would be interesting to study in their own right. The most important of these aspects are: what kind of people programmers are, and how they live their lives;[17] the management of programming, the economic and business aspects of programming; and the programmers' identification with their job. These aspects will be considered in this treatise, but they will be subordinate to the central theme of work.

To be truly complete, the section on mainstream programming theories (section 3.2) should also present ideas from the field of information systems science, which consists of literature with a largely sociological, economical, and managerial emphasis. It is with regret that I have omitted a treatment of information systems science – the justification is that it has arguably had less influence on the everyday practice of programmers than the fields of software engineering, computer science, and Agile methods, as it is more concerned with the site-wide implementation of information technology systems in large organizations than with programming practice.

[17]In ethnology this is sometimes referred to as life modes. See Christensen 1987.

The above mentioned research approaches and theoretical perspectives have not been left out of the thesis because they are irrelevant to the subject. The choice of topics to include has been made in order to be able to present clear and focused research results that are not mired down in too many issues that are interesting but tangential.

1.6 *Audience of the dissertation*

This dissertation is intended for a general academic audience. The ideal reader is interested in the subject of programming and familiar with the general principles of research and scholarship, but expertise in either ethnology or programming research is not a requirement to understand the arguments of the thesis. Naturally, some parts of the the thesis will be easier to understand for the reader with expertise in programming, and some parts will be easier to understand for the reader with expertise in ethnology, but care has been taken to explain the concepts used so that the argument can be followed also by readers without expertise, while at the same time making sure that the factual content of the thesis lives up to the expectations of experts in the fields.

1.7 *Overview of the dissertation*

The rest of the dissertation is organized as follows: Chapter 2 presents the current state of the art within research on programming as work. Section 2.1 concerns research coming from the programming fields while section 2.2 concerns research from the various fields of cultural studies.

Sections 3.2.1 (Software engineering), 3.2.2 (Agile software development), 3.2.3 (Computer science), and 3.2.4 (Object-Oriented Programming) present an overview of the mainsteam theories of programming that have predominantly shaped contemporary programming practice. Sections 3.3 (Hermeneutical theory), 3.4 (Cultural form theory), and 3.5 (Rhetorical theory) present the cultural theories that are applied to programming practice in this dissertation.

Sections 4.2 (Participant observation) and 4.3 (Interviews) describe the data collecting methods used in the dissertation. Section 4.5 present the different methods of analysis that the data are subjected to. Section 4.6 gives a brief account of the science theoretical foundation of the dissertation. Section 4.7 lists all the empirical data collected for this study; when it was collected and in which form.

Chapter 5 presents the case study in computer game programming. Section 5.1 is a description of the case. In section 5.2.1 the case study is analyzed with the use of mainstream theories of programming. In section 5.2.2 the case study is anlalyzed with hermeneutical theory.

Chapter 6 presents two case studies in safety critical programming. The case studies are described in sections 6.2.1 (A large avionics company) and 6.2.2 (A small farming systems company). Section 6.3 presents a hermeneutical analysis of the cases.

Chapter 7 presents cultural form analyses of the empirical material. In section 7.1 the case studies of chapter 5 and 6 are subjected to a comparative form analysis. In section 7.2 a number of smaller case studies in safety critical programming are presented and analyzed. The presentation and analysis in this section is done concurrently.

Chapter 8 presents a rhetorical analysis of source code from the case study of chapter 5 (Game programming). Again, the presentation and analysis is done concurrently.

Chapter 9 concludes the dissertation. In section 9.1 the results of the dissertation are presented. In section 9.2 the implications of the results for the practice of programming and for programming research are discussed. Section 9.3 addresses the validity and possible generalizations of the study.

Chapter 2

Related research

2.1 *Programming research*

Cultural approaches to programming are not unheard of, though they are scarce within academia. Outside of academia, the popular blogger Joel Spolsky, for example, has described the difference between Windows programming and UNIX programming as a difference between two programming cultures.[1]

Spolsky argues that UNIX programming evolved in a time and in an environment where most people who used computers were programmers, and very few were users without programming knowledge. Therefore the UNIX programming culture assumes that a user is also a programmer and knows how to modify the program to his liking. In contrast, the Windows programming culture developed among progammers who were making programs for users who are not programmers themselves. Therefore a typical Windows program is easier to use for a non-programmer, but a programmer will find it harder to modify for his own purposes.

[1]Spolsky 2004 chp. 18.

It seems that while there is a recognition within academia that it is important to study actual work processes, there is a lack of research on the subject, cf. Singer et al. 2008: "Software engineering is an intensely people-oriented activity, yet little is known about how software engineers perform their work."[2]

The cultural approach has been employed on occasion within the existing research literature on programming. For example, Carolyn Seaman, in an anthology about empirical software engineering research, describes participant observation as one of several data collection methods researchers can use.[3] Seaman also describes a method of data analysis known as the "constant comparison method", or "sense making", which essentially amounts to looking very carefully at the data, trying to make sense of it, and striving to find support for preliminary hypotheses in the data.[4]

This so-called ethnographic research method has been employed by several software researchers, for example Hugh Robinson and Helen Sharp[5], who have collected data by participant observation in a number of software companies with a focus on organisational culture and a certain way of working with software called "Extreme Programming".[6] Other researchers have employed ethnographic methods in studies of companies such as Microsoft and Siemens.[7]

The use of hermeneutics in the study of programming is a largely unexplored area. Hermeneutics has previously been applied to artificial intelligence, but without much insight or success.[8] Cole and Avi-

[2]Singer et al. 2008 in Shull et al. p. 9.

[3]Seaman 2008 in Shull et al. chp. 2.1, p. 37.

[4]Ibid. chp. 3.1.1, p. 49.

[5]Robinson & Sharp 2003: "XP Culture." Robinson & Sharp 2005 (I): "The Social Side of Technical Practices." Robinson & Sharp 2005 (II): "Organisational Culture and XP."

[6]Extreme Programming (XP) is a so-called Agile methodology, see section 3.2.2 (Agile software development).

[7]Begel & Simon 2008: "Struggles of New College Graduates in Their First Software Development Job." Herbsleb et al. 2005: "Global Software Development at Siemens." Ko et al. 2007: "Information Needs in Collocated Software Development Teams."

[8]Mallery et al. 1987: "Hermeneutics: From Textual Explication to Computer Understanding?" Massachusetts Institute of Technology Artificial Intelligence Laboratory.

son have employed hermeneutics in the field of information systems, though their work does not address programming. They raise the point that hermeneutics “is neither well accepted nor much practiced in IS [Information Systems] research”, which perhaps helps explain that it is even less practiced in programming research.[9]

The concept of understanding has a close connection to hermeneutics. Jorge Aranda’s doctoral dissertation from 2010 applies the perspective of understanding to programming, in what Aranda calls “a theory of shared understanding”. According to this approach, software development is not a simple transfer of data but a gradual process of mutual understanding. The software team has to discover all relevant aspects of the problem it intends to solve, and this requires interpretation.[10] Aranda emphasises that shared understanding develops slowly from practice, that it is dependent on context, and that it takes effort and practice to gain knowledge of the context of understanding.[11]

Aranda’s conclusions are very consistent with those that follow from a hermeneutical perspective on programming. The main difference between Aranda’s work and the approach taken in this treatise is that Aranda bases his theory on cognitive psychology rather than hermeneutics. In cognitive psychology, understanding consists of internal mental models;[12] whereas hermeneutical theory views understanding as an existential question of what truth is.[13]

With its basis in cognitive psychology, Aranda’s work can be seen as representative of the research field called "psychology of programming". While the theme of psychology of programming could be construed to be relevant to this dissertation, the literature, apart from Aranda’s work, does however seem to focus primarily on themes other than programming and work. Curtis and Walz wrote in 1990 that:

[9]Cole & Avison 2007 p. 820.
[10]Aranda 2010 p. 3, p. 89.
[11]Ibid. p. 111, p. 95.
[12]Ibid. p. 42 f.
[13]See sections 3.3.5 (Understanding) and 3.3.9 (Ethical and technical knowledge).

> "Programming in the large is, in part, a learning, negotiation, and communication process. These processes have only rarely been the focus of psychological research on programming. The fact that this field is usually referred to as the 'psychology of programming' rather than the 'psychology of software development' reflects its primary orientation to the coding phenomena that constitute rarely more than 15% (Jones, 1986) of a large project's effort. As a result, less empirical data has been collected on the team and organizational aspects of software development."[14]

More recent research seems to confirm Curtis and Walz's statement. For example, a paper from 2000 focuses more narrowly on programming language design,[15] while a paper from 2004 focuses on students and education.[16] A 2011 paper does address an aspect of software engineering as work, in that it investigates software engineer's motivation to do their work.[17] While the theme is relevant to this dissertation, the analysis in the mentioned paper is limited to a report of frequencies of questionnaire answers. As such, the work lacks a conceptual framework or theoretical foundations that could make it fruitful to engange in a discussion of its relevance to this dissertation.

Informatics professor Antony Bryant has argued for using the perspective of understanding in software engineering. In a formulation that is similar to Aranda's, Bryant writes that:

> "The process [of requirements engineering] is not one of passing or capturing information; the aim is for all those involved to work together to achieve a coherent and consistent view of the system context, an objective that is inherently difficult and requires collaborative input from all participants."[18]

[14]Curtis & Walz 1990 p. 267.
[15]Pane & Myers 2000.
[16]Mancy & Reid 2004.
[17]Sach, Sharp, and Petre 2011.
[18]Bryant 2000 p. 84.

As we shall see, this viewpoint is in harmony with a hermeneutical understanding of programming.

See also section 3.2 (Mainstream programming theories) for further academic research in programming that is relevant to the topic of this dissertation.

2.2 *Cultural research*

This purpose of this section is to outline the approaches taken by other cultural researchers to studies of work, programming, and new technology in general. For my purpose, the more relevant research is that which has been done by ethnologists, since ethnology is my primary research perspective and inspiration. For the sake of completeness, research done by other cultural researchers, such as sociologists and anthropologists, is also presented.

Not surprisingly, information technology and other new technology has been met with a great deal of interest by cultural and social researchers. The major part of this interest has been directed toward the users of technology and its influence on their lives. An example is Tina Peterson's ethnographic study of the role played by microwave ovens in people's daily lives, and their feelings toward the technology.[19]

A smaller amount of interest has been directed at the work aspects of new technology; that is, cultural studies of how technology is made. An illustrative example of this is Daniel Miller and Don Slater's ethnographic study of internet use in Trinidad. Although most of their study concerns internet use from a user perspective and the role of the internet in Trinidadian national consciousness, Miller and Slater include a chapter on doing web design as work, which focuses on the business aspects of the work.[20]

[19]Peterson 2009.

[20]Miller & Slater 2000 chp. 6.

2.2.1 *Ethnology*

Looking specifically at ethnological research, Robert Willim has written a book about a Swedish "new economy" software company, Framfab ("Future Factory").[21] Willim investigates the work practices, the company culture, and the business aspects of Framfab, showing how many of the company's practices can be seen as expressions of a more general wave of ideas in society about "the new economy" and "fast companies". He also shows how, in the end, Framfab was forced to conform more to traditional business practices than it originally desired.[22]

Willim's work, and the work presented in this treatise, fall within a long ethnological tradition of work studies. In recent years, many ethnological studies have taken a turn away from detailed research in work practices toward an interest in more general cultural aspects of the work, where the concrete work practices are discussed as a backdrop for social commentary. An example is Marianne Larsson's study of postal workers in Sweden between 1904-17, which focuses on the postmen's uniforms as a symbol of hierarchical power.[23] Another is Leena Paaskoski's study of Finnish forest workers in the second half of the 20th century, which focuses on gender issues.[24]

Earlier ethnological research, from the late 1970s and early 1980s, devoted more attention to work practices, although the research was often focused on the totality of the work situation. An example is Billy Ehn's study of factory workers in Stockholm in 1977-8, which includes detailed descriptions of work practices and their influence on the workers' existence.[25] Another example is Gudrun Gormsen's study of the diary of a Danish heath farmer from 1829 to 1857.[26]

[21]Willim 2002. Willim 2003 in Garsten & Wulff.

[22]Willim 2002 p. 139 ff.

[23]Larsson 2008 in *Ethnologia Scandinavica.*

[24]Paaskoski 2008 in *Ethnologia Scandinavica.*

[25]Ehn 1981.

[26]Gormsen 1982 in *Folk og Kultur.*

The research presented in this treatise can be said to fall within an even older tradition of classical ethnology that has work practices as its primary focus, although the totality of the work situation is always kept in mind. An example of this kind of research is Ole Højrup's 1967 book, *The Rural Dwelling Woman.*[27] Højrup studied a type of work – domestic female farm work – that was in decline, whereas the research presented here is concerned with a type of work that is on the rise. This results in a slight difference in the aims of the research, and in the methodological challenges, but the underlying ethnological perspective is essentially the same.

2.2.2 *Sociology, anthropology, and other*

Fields other than ethnology have engaged with work studies of new technology. Sociologists have been researching programming work since at least the 1970s. Sociological work studies were at this time dominated by Marxist theory and, consequently, the research focused on providing arguments in support of Marxist ideology. Philip Kraft's 1979 article about computer programmers is an example of this kind of research.[28] Later research challenged the Marxist studies. Sarah Kuhn's 1989 study of commercial bank programmers concludes that programming work is creative, one-of-a-kind intellectual work that cannot easily be understood using theories of assembly line factory work.[29]

In more recent sociological research, Karolyn Henderson in 1998 wrote about how engineers' use of diagrams affects their work practices.[30] Henderson uses the sociologist Susan Leigh Star's concept

[27]Højrup 1967: *Landbokvinden.*

[28]Kraft 1979: "The Industrialization of Computer Programming: From Programming to 'Software Production'."

[29]S. Kuhn 1989: "The Limits to Industrialization."

[30]Henderson 1998 chp. 3.

boundary object to analyze engineering diagrams.[31] The boundary object functions as a physical artefact that can communicate with different groups on different levels, and bring together people that need to cooperate despite holding differing understandings.

Star has herself studied the development of a large administrative software system.[32] Falling within the sociological tradition of informations systems research, the study focuses on organisational changes. Yet another example of sociological research is Seán Ó Riain's studies of Irish software companies. These studies focus primarily on day-to-day social interaction in the office,[33] and on the business aspects of the software industry.[34]

Leslie Perlow has produced an interesting study about software engineers, their use of time, and their working conditions.[35] We will return to Perlow in chapter 9 (Conclusion). Donald Schön has written about the work of creative intellectuals and how they gain knowledge, a process that is closely connected with hermeneutics theory.[36] Schön is discussed in more detail in section 5.2.2.1 (The work process).

Anthropological research has also produced a number of studies of new technology and programming work. In many such studies, anthropology is applied to management research, with a theoretical anchor in Edgar Schein's 1985 book *Organizational Culture and Leadership.* Two characteristics of Schein's thinking are worth noting. First, the organisational culture of a company is viewed as an entity that can be studied more or less separately. Secondly, an instrumental view of culture is presented, through the idea that organisational culture can and should be formed for specific purposes.

Susanne Ekman provides an example of anthropology applied to management research.[37] Her study focuses on journalists' perception of their work, and how this is connected with their identity and am-

[31]See e.g. Star & Griesemer 1989.
[32]Star & Ruhleder 1996.
[33]Ó Riain 2000.
[34]Ó Riain 1997.
[35]Perlow 1997.
[36]Schön 1983.
[37]Ekman 2010.

bitions. Another example is provided by Gideon Kunda's study of an anonymous high-tech engineering company.[38] Kunda focuses on the internal power-play and positioning that takes place within the company, and how the company's self-perception of its culture is expressed. Kunda views many of the activities of daily work life as ritual enforcements of the collective identity.

The phenomenon of free and open source software has received some attention from anthropologists. Magnus Bergquist has focused on the economic aspects of open source software development, and applied Marcel Mauss's classic anthropological theory of gift giving to this.[39] Gabriella Coleman has studied open source software "hackers".[40] She interprets the movement as both an extension of, and an opposition to, a liberal or neo-liberal political system; hence her focus is primarily on the ideological, political, and identity aspects of software development.

A number of anthropologists have conducted research associated with the Palo Alto Research Center of the well-known company Xerox. A common theme for these researchers has been an investigation of the interface between people and the machines they use in their work.[41] These include photocopy machines, but also, for example, color management on personal computers, and intelligent signs. Lucy Suchman, one of the more influential researchers, states: "Human-machine configurations matter not only for their central place in contemporary imaginaries [i.e. imagination] but also because cultural conceptions have material effects."[42] Notably, Suchman argues that people's use of machines is characterised less by pre-made plans than by so-called situated actions – actions that depend on situation and context and that can only be presented in a rational logical form with difficulty.

A crucial contribution from anthropology is the work of Bruno Latour, which investigates the internal workings of science and engi-

[38]Kunda 1992.

[39]Bergquist 2003 in Garsten & Wulff.

[40]Coleman 2005.

[41]Szymanski & Whalen (eds.) 2011.

[42]Suchman 1987 [2007] p. 1.

neering (what Latour calls “technoscience”) and their relationship to society at large.[43] Latour’s work provides some theoretical points that are important to this treatise; these are briefly mentioned in sections 3.5 (Rhetorical theory) and 9.2 (Discussion).

Sherry Turkle has written an excellent book about people’s personal relationship to computers, which examines many of the non-professional aspects of programming that are left out in this dissertation. Although Turkle is a psychologist, her research is as much a North American cultural study as it is psychological. One of the central themes of her book is how people imbue computers with meaning. She writes:

> “When people repair their bicycles, radios, or cars, they are doing more than saving money. They are surrounding themselves with things they have put together, things they have made transparent to themselves.”[44]

Turkle shows in her book that this is as true of computers as it is of bicycles and radios. She also examines programming styles in different situations and identifies three generalised types of programmers: “hard style” programmers are objective and abstract; “soft style” programmers are artistic and immediate; and “hacker style” programmers are primarily concerned with “winning”, and exclude themselves from non-programmers. We will not be using these categories in this treatise but they illustrate one way in which research can take an approach to programming that is not focused on work.

The research discussed above has in common that it is all concerned with actual practice, and based on empirical observation. A number of authors have taken a more speculative approach. Don Ihde has written a philosophy of technology that is critical of other philosophical writers, notably Heidegger.[45] Using a similar approach, Richard Coyne has written a philosophy called *Designing Information Technology in the Postmoderne Age*, containing speculation upon, and

[43]Latour 1987. Latour & Woolgar 1979.
[44]Turkle 1984 [2005] p. 176.
[45]Ihde 1979: *Technics and Praxis.*

critique of, many philosophical writers, including Heidegger and Derrida.[46]

Jay Bolter and Richard Grusin have written a philosophy of what they call *remediation*, a strategy of representing media within other media, which they claim is characteristic of new digital media.[47] In addition to a philosophical critique, they base their arguments largely on references to artworks, pictures, and websites; thus their approach comes to resemble literary critique or art history.

Authors such as Lev Manovich[48] and Friedrich Kittler[49] have written philosophies that come closer to essays in their style, and which explore the meaning of concepts such as software and media. A common trait for all the speculative writing mentioned here is that it has little to do with how programming work is carried out in practice.

This is also the case for the writings of Donna Haraway, which have attracted a large amount of attention within cultural studies of new technology. Haraway's work is, in her own words, ironic and blasphemous, which literally means that it is deliberately untrue.[50] Its stated goal is to advance socialist-feminist ideology. That this is a worthwhile goal is a matter of belief, and Haraway does not provide a justification as to why socialist-feminist ideology would be useful to technology workers or contribute to a better understanding of technology. As such, Haraway's work resembles the ideological Marxist sociology of the 1970s mentioned above.

[46]Coyne 1995.

[47]Bolter & Grusin 1998 [2000] p. 45.

[48]Manovich 2013.

[49]Kittler 2009.

[50]"This chapter is an effort to build an ironic political myth faithful to feminism, socialism, and materialism. Perhaps more faithful as blasphemy is faithful, than as reverent worship and identification. ... At the centre of my ironic faith, my blasphemy, is the image of the cyborg." Haraway 1991 chp. 8: "A Cyborg Manifesto: Science, Technology, and Socialist-Feminism in the Late Twentieth Century", p. 149

Chapter 3

Theory

3.1 *Choice of theory*

The approach is to use cultural theory, specifically ethnological theory, to build a theory of programming that encompasses the relevant human factors. Ethnology as defined in this dissertation is a cultural science of peoples' daily life. As such, is has a rich tradition of studying peoples' work practices and workplaces. This is an advantage when we want to study the phenomen of programming as a work practice, which is the first step toward a cultural understanding of programming.

There are many different approaches to cultural theory within the field of ethnology. The approach used in this dissertation is *cultural form theory*, which is best known from the variant *life-mode theory*,[1] a theory that originated in Copenhagen in the 1970's and since then has been developed by a number of ethnologists. The special advantages of this theory is that it is well suited to examine the relationship between a given cultural practice and the factors that are external to

[1]Højrup 1995.

it.[2] In this case, the cultural practice is software development, and the external factors are things such as higher management, market demands, corporate politics, *et cetera.*

On a practical level, a theory is needed to analyze programming as a cultural phenomenon in order to be able to relate source code, program design, archtitectural descriptions, and the like to the culture in which they are produced. Before the empirical investigations started, the intention was to use the body of theory called "new rhetorics", a revival of classical rhetorics that has been brought up to date as a modern theory of argumentation.[3] Viewing programming as comprised by arguments allows for analyzing the meaning of individual programming constructs, an important subgoal in a cultural description and an approach which is demonstrated in chapter 8 (Rhetorical case study and analysis).

As it turns out, the creation of work products and source code, though important, is not the paramount factor when investigating how software development practices are formed. As the case studies and the hermeneutical analyses of them amply demonstrate, programming practice is to a large extent shaped by how the programming problems are understood by software developers. Understanding is the domain of hermeneutics, and for this reason it is not suprising that the theory works well in describing the kind of intellectual, creative work that programming is an example of. In this respect, hermeneutics is chosen as an analytical theory because it suits the data well; in another respect, it is chosen because it negotiates the theoretical distance between the reflective approach of cultural form theory and the practical approach of rhetorics.

The fields of cultural studies offer many choices of theory other than cultural form theory and hermeneutics that could have been used in this dissertation. Some of these, briefly mentioned in section 2.2, have so far not produced research insights that explain programming work practice, and for this reason seem poor choices for this dissertation. Others might conceivably produce better results, however, to

[2]Højrup 2002, chp. 5.
[3]Perelman & Olbrechts-Tyteca 1958.

accurately assess whether such cultural theories actually offer a better alternative requires to do a full study using the theories, and this is a laborious task. The best argument for not considering other theoretical approaches in this dissertation is that the present choice of theories produced new insights, and the theories did not produce contradictory explanations of empirical phenomena or leave important phenomena unexplained.

3.2 *Mainstream programming theories*

This section discusses the schools of thought that arise from the programming communities themselves. Its purpose is to explain how the different programming traditions perceive programming: what they think programming is, and what kind of things they consider important to focus on when studying programming. These traditions largely make up the existing research on how to carry out programming, and it is these that are taught to would-be programmers at universities. The different schools have a large influence on the way programming is practised, and it is consequently important to understand these traditions of thought in order to be able to understand programming practice.

The most important schools of thought are computer science, software engineering, and Agile thinking. These are not exclusive schools of thought; for example, computer science has had a considerable influence on both software engineering and Agile thinking. It is not my intention to present a complete description of the fields, and there are many subfields and work carried out by individual researchers that will not be covered by the material here. Rather, my intention is to identify the most important and well-known ideas. For this reason, the scholars cited here are people whose views are respected within their fields. Some have had large influence on their fields, others are less well known but represent mainstream, and uncontroversial ideas.

For the sake of completeness, Object-Oriented Programming and Design is also presented, since it has had a large influence on the practice of programming. Its influence on work processes, however, has not been as significant as those of the above mentioned schools of thought.

3.2.1 *Software engineering*

3.2.1.1 *The origins of software engineering*

Software engineering is, as the name implies, a school of thought within programming that arose from engineering communities and is predominant among those software scholars and professionals who are inclined towards engineering. The first electronic computers constructed in the 1940s during the Second World War were built mainly by electrical engineers. In those early days there was not a great deal of distinction between the hardware engineers who built the machines, and those who programmed them. While computers slowly gained importance during the 1950s and 1960s, the engineers who had the task of programming them began to see themselves as a distinct group: software engineers.

The first electronic computers were developed for military use. Like other branches of engineering, for example aeronautical engineering, software engineering has since the beginning had strong ties to the military and consequently to the government. The origin of the term "software engineering" is attributed to two conferences sponsored by the NATO Science Committee, one in 1968 on "Software Engineering" and one in 1969 on "Software Engineering Techniques".[4] These conferences did not address the definition of the discipline, or its aim and purposes. The things that were discussed were the various problems and techniques involved in creating software systems, for example in connection with the design, management, staffing, and

[4]Naur & Randell 1969. Buxton & Randell 1970.

evaluation of systems, as well as for example quality assurance and portability.[5]

The connection with military engineering means that many prominent software engineers have worked for the military. The well-known researcher Barry Boehm, for example, worked on the Semi-Automated Ground Environment radar system for the U.S. and Canadian air defence in the beginning of his career in the 1950s.[6] Of the 12 represented in George F. Weinwurm's 1970 anthology *On the Management of Computer Programming*, at least half have had military careers.

However, after the war, computers soon found uses outside of the military. From the 1950s throughout the 1970s, these uses were primarily within the administration of other branches of government and in "big business". The big business sector was mostly served by large corporations such as IBM – the well-known software engineer Frederick P. Brooks, Jr. worked for IBM, and also later served the military as a member of the U.S. Defense Science Board.[7] The authors in Weinwurm's anthology without military careers have all had careers with either IBM or other large corporations, such as banks.

3.2.1.2 *Brooks*

In 1964-65 Frederick P. Brooks, Jr. was the manager of the development of the operating system of the IBM 360 computer, a technically innovative machine.[8] In 1975 he published the book *The Mythical Man-Month: Essays on Software Engineering* about his experiences of the project. This book has since become perhaps the most well-known book in the field of software engineering, as well as being known outside the field.

[5]Portability concerns the technical relationship between programs and the machines on which they run. If a program can easily be made to run on several different makes of machines, it is said to be portable.

[6]Boehm 2006.

[7]See for example the 1987 *Report of the Defense Science Board Task Force on Military Software.*

[8]Brooks 1975 [1995] preface. Patterson & Hennessy 1997 pp. 525–527.

In Brooks' approach to software engineering, the main concerns of development are management and organization. The technical part of development is also discussed, but falls outside of the main focus of the book. Brooks' thinking on management stresses the importance of estimation and the use of formal documents. The undisputed top priority of management is to be on time. Brooks poses the famous question "How does a project get to be a year late?" and gives the laconic answer " . . . One day at a time."[9]

In order to avoid slipping schedules, Brooks emphasizes the importance of good estimates of how long programming tasks will take. He also acknowledges that this is very difficult to achieve: nonetheless, managers must do their best, because without good estimates the project will end in catastrophe. Because of the difficulties inherent in estimation, Brooks advocates looking at data for time use in other large projects, to try to discover some general rules of how long programming takes depending on various factors, such as which programming language is used and the amount of interaction between programmers.

Good documentation is important to a successful project, for two reasons. When the manager writes down his decisions he is forced to be clear about the details, and when they are written down they can be communicated to others on the team. Producing a small number of formal documents is critical to the job of the manager. According to Brooks, the purpose of management is to answer five questions about the project: what?, when?, how much?, where?, and who? For each of the questions there is a corresponding document that answers it, and these formal documents form the core basis for all of the manager's decision making:[10]

[9]Brooks 1975 [1995] p. 153.

[10]"The technology, the surrounding organization, and the traditions of the craft conspire to define certain items of paperwork . . . [the manager] comes to realize that a certain small set of these documents embodies and expresses much of his managerial work. The preparation of each one serves as a major occasion for focusing thought and crystallizing discussions that otherwise would wander endlessly. Its maintenance becomes his surveillance and warning mechanism. The document itself serves as a

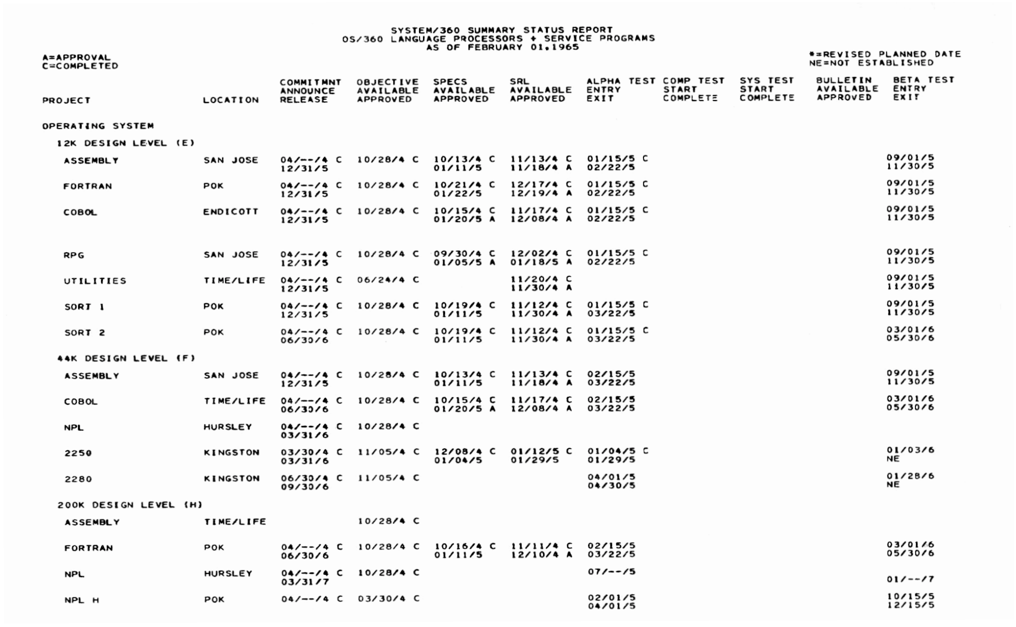
what?	is answered by	objectives
when?	"	schedule
how much?	"	budget
where?	"	space allocation
who?	"	organization chart

Besides estimation and documentation, a third important aspect of management is planning. Brooks' advice to managers is to use the Program Evaluation and Review Technique (PERT) developed by the U.S. Navy in the 1950s, which consists of providing estimates and start dates for all tasks, as well as listing which tasks depend on other tasks. The total duration of the project can then be calculated, as well as the critical tasks, which are those tasks that must not be delayed if the project is to finish on time. Figure 3.1 shows an example of a PERT chart given by Brooks.

The second major concern of Brooks' book is organization. In this he favors a strong hierarchy. The recommended team organization is what is known as a chief programmer team, though Brooks calls it the "surgical team". In this, the team consists of a chief programmer and his subordinate right hand, who together do all the actual programming on the project. They are supported by up to eight other people who do various technical and administrative tasks. According to Brooks, it is only possible to develop a coherent system if the number of people that contribute to the design is limited. The central idea of the chief programmer team is thus that it is an organization that allows a lot of people to work on the project in supporting roles while making sure that only the chief programmer contributes to the design.

Brooks also discusses how to organize the highest layer of project management. He maintains that it is extremely important to separate technical and administrative executive functions. He likens the technical manager to the director of a movie, and the administrative manager to the producer. Various possibilities for dividing the power between the two are considered, and Brooks finds that it is best if

check list, a status control, and a data base for his reporting." Brooks 1975 [1995] p. 108.



SYSTEM/360 SUMMARY STATUS REPORT
OS/360 LANGUAGE PROCESSORS + SERVICE PROGRAMS
AS OF FEBRUARY 01,1965

A=APPROVAL
C=COMPLETED

*=REVISED PLANNED DATE
NE=NOT ESTABLISHED

PROJECT	LOCATION	COMMITMNT ANNOUNCE RELEASE	OBJECTIVE AVAILABLE APPROVED	SPECS AVAILABLE APPROVED	SRL AVAILABLE APPROVED	ALPHA TEST ENTRY EXIT	COMP TEST START COMPLETE	SYS TEST START COMPLETE	BULLETIN AVAILABLE APPROVED	BETA TEST ENTRY EXIT
OPERATING SYSTEM										
12K DESIGN LEVEL (E)										
ASSEMBLY	SAN JOSE	04/--/4 C 12/31/5	10/28/4 C	10/13/4 C 01/11/5	11/13/4 C 11/18/4 A	01/15/5 C 02/22/5				09/01/5 11/30/5
FORTRAN	POK	04/--/4 C 12/31/5	10/28/4 C	10/21/4 C 01/22/5	12/17/4 C 12/19/4 A	01/15/5 C 02/22/5				09/01/5 11/30/5
COBOL	ENDICOTT	04/--/4 C 12/31/5	10/28/4 C	10/15/4 C 01/20/5 A	11/17/4 C 12/08/4 A	01/15/5 C 02/22/5				09/01/5 11/30/5
RPG	SAN JOSE	04/--/4 C 12/31/5	10/28/4 C	09/30/4 C 01/05/5 A	12/02/4 C 01/18/5 A	01/15/5 C 02/22/5				09/01/5 11/30/5
UTILITIES	TIME/LIFE	04/--/4 C 12/31/5	06/24/4 C		11/20/4 C 11/30/4 A					09/01/5 11/30/5
SORT 1	POK	04/--/4 C 12/31/5	10/28/4 C	10/19/4 C 01/11/5	11/12/4 C 11/30/4 A	01/15/5 C 03/22/5				09/01/5 11/30/5
SORT 2	POK	04/--/4 C 06/30/6	10/28/4 C	10/19/4 C 01/11/5	11/12/4 C 11/30/4 A	01/15/5 C 03/22/5				03/01/6 05/30/6
44K DESIGN LEVEL (F)										
ASSEMBLY	SAN JOSE	04/--/4 C 12/31/5	10/28/4 C	10/13/4 C 01/11/5	11/13/4 C 11/18/4 A	02/15/5 03/22/5				09/01/5 11/30/5
COBOL	TIME/LIFE	04/--/4 C 06/30/6	10/28/4 C	10/15/4 C 01/20/5 A	11/17/4 C 12/08/4 A	02/15/5 03/22/5				03/01/6 05/30/6
NPL	HURSLEY	04/--/4 C 03/31/6	10/28/4 C							
2250	KINGSTON	03/30/4 C 03/31/6	11/05/4 C	12/08/4 C 01/04/5	01/12/5 C 01/29/5	01/04/5 C 01/29/5				01/03/6 NE
2280	KINGSTON	06/30/4 C 09/30/6	11/05/4 C			04/01/5 04/30/5				01/28/6 NE
200K DESIGN LEVEL (H)										
ASSEMBLY	TIME/LIFE		10/28/4 C							
FORTRAN	POK	04/--/4 C 06/30/6	10/28/4 C	10/16/4 C 01/11/5	11/11/4 C 12/10/4 A	02/15/5 03/22/5				03/01/6 05/30/6
NPL	HURSLEY	04/--/4 C 03/31/7	10/28/4 C			07/--/5				01/--/7
NPL H	POK	04/--/4 C	03/30/4 C			02/01/5 04/01/5				10/15/5 12/15/5

Figure 3.1: PERT chart from Brooks' work on the IBM 360 in 1965. From Brooks 1979 [1995].

priority is given to the technical role, so that the technical manager is placed over the administrative manager, at least in projects of limited size. This arrangement mirrors the recommended organization of the chief programmer team, where a programmer is similarly placed in charge of administrative personnel.

In relation to technical issues, Brooks devotes some space to recommendations for trading off computation time for machine memory space, so that one has programs that are slower but also takes up less space. This was a sensible decision at the time. Nowadays, the decision it not so simple, but the choice between having either fast programs or small programs is ever relevant. Other than this, Brooks recommends that the machines and programs that are used during development are of the highest quality, and he lists some categories of programs that are useful in the programming work.

Regarding development methods, Brooks advocates top-down design, structured programming and component testing, as methods to avoid mistakes in the programming. Top-down design amounts to making a detailed plan for the system as a whole before beginning to design the individual components. Structured programming means enforcing a certain discipline in the programming in order to avoid common programmer mistakes and to make it easier to understand the programs. Component testing means testing each piece of the program separately before trying to see if the system as a whole works.

An interesting aspect of Brooks' work is that he appeals to Christian ontology to justify some of his fundamental assumptions. It is quite important in Brooks' argumentation to emphasize that programming is creative work, and the inherent creativity of humans is explained as a consequence of man being created in God's image.[11] He views communication problems as the root cause of problems in programming projects, and the problems are explained as analogous to the Biblical story of the Tower of Babel,[12] which he calls "the first engineering fiasco." And we receive a hint of a justification for the

[11]Brooks 1975 [1995] p. 7.
[12]Ibid. p. 74, p. 83.

central idea of the of the chief programmer team as he states, "A team of two, with one leader, is often the best use of minds. [Note God's plan for marriage]."[13] (Square brackets in the original.)

The interesting thing about these appeals to Biblical authority is that they connect Brooks' thinking to the mainstream Christian intellectual tradition of the Western world. This is in contrast to most contemporary literature in software engineering, where the fundamental ideas about humanity are instead taken from areas like psychology and sociology, which have a distinctly modernistic approach to ontology.

3.2.1.3 *Process thinking*

A prominent characteristic of the way in which software engineers think about development processes is that the process is divided into a number of distinct phases that follow each other. The exact number of phases varies between authors, though they usually follow a general trend. At the beginning of development is a phase where the problem is specified, followed by a phase of high level design, often called architecture. Then follows more detailed design, then coding, and finally testing, and actual use of the program.

The phases that have to do with specification, program architecture, and design clearly correspond to the design process in ordinary engineering. According to Vincenti, the historian of aeronautics engineering, the ordinary engineering design process consists of:[14]

1. Project definition.
2. Overall design.
3. Major-component design.
4. Subdivision of areas of component design.
5. Further division of categories into highly specific problems.

[13]Ibid. p. 232.
[14]Vincenti 1990 [1993] p. 9.

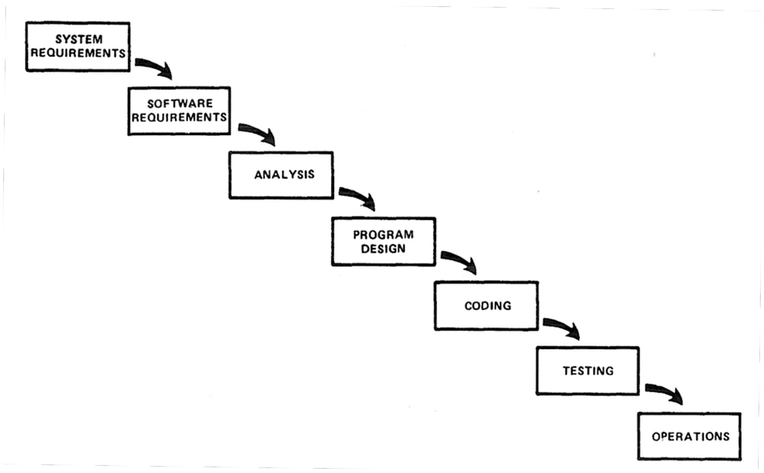


Figure 3.2: The waterfall model: "Implementation steps to develop a large computer program for delivery to a customer." From Royce 1970.

This process mirrors the software development phases, in which the activities also start with specification and definition, after which they progress to design phases that are at first very general, and further on continue with phases of more and more detailed design.

The idea of a distinct testing phase that follows the design and coding phases also corresponds to an ordinary engineering notion of testing. In ordinary engineering, the testing phase naturally follows after design and building phases; it is impossible to test something that has not yet been built. This leaves the coding phase as the only phase that is distinct to software engineering. It is often likened to a construction or building phase in ordinary engineering.

Process thinking in software engineering can be considered to revolve around two fundamental concepts: phases, and how to get from one phase to the next. Boehm writes: "The primary functions of a software process model are to determine *the order of the stages* involved in software development and evolution and to establish the *transition criteria* for progressing from one stage to the next."[15]

The transition criteria to which Boehm refers are what are variously called "documents", "work products", or "artefacts". These are more often than not documents in the usual sense of the word, i.e. the sense in which Brooks speaks of documents; but they can also be other kinds of work material in written form, for example source code or test result listings.

The best known model of software development by far is the so-called "waterfall" model, in which the phases of software development are arranged in a strict sequence such that one phase is completely finished before the next begins. In 1970 Winston W. Royce published an article describing this model, based on his experience of developing software for spacecraft missions. Royce's version of the waterfall model can be seen in Figure 3.2. The model has gained its name from the way the boxes and arrows in the picture resembles the flow of water in a waterfall.

It should be noted that Royce neither intended the model to be descriptive of how software development actually happens, nor did he find it realistic to have such a simplistic model as the goal for how development should be done. However, the model does reflect a conception of how programming would be "in an ideal world", if it were not hampered by the imperfections of people. Thus, many software models that are more complicated, and meant to be more realistic, are essentially embellishments upon the waterfall model. Note also that Royce did not think the model necessary at all for small programs – it is only intended for large projects.

The "V-model" is closely related to the waterfall model, but places much more emphasis on the testing of the program. Instead of having a single test phase, as in the waterfall model, the test phase is separated into a number of test phases, each corresponding to one of the design phases of the waterfall model. The V-model gained its name because of its shape: a version can be seen in Figure 3.3. This

[15]Boehm 1988 p. 61.

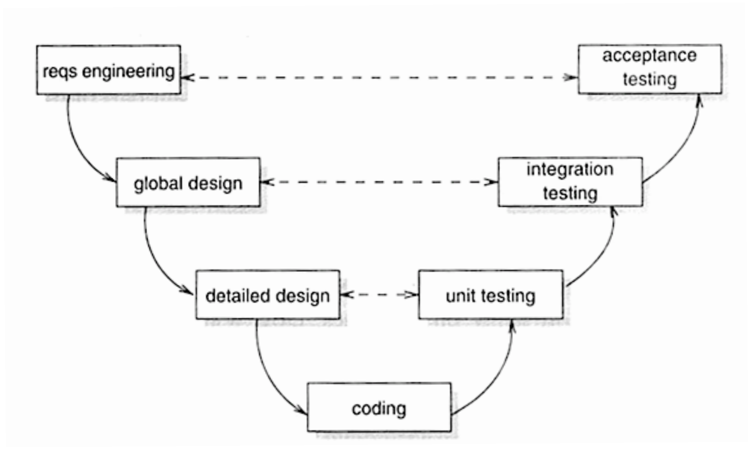


Figure 3.3: The V-model. From van Vliet 2008.

model plays an important role in safety critical programming; we shall therefore return to it in chapter 6 (Safety critical programming).

3.2.1.4 *Systems thinking*

Software engineering thinking is often preoccupied with the administrative aspects of management; that is, with planning, documenting and evaluating. The general interest of software engineering is in large, complex projects that need to be managed intensively.

An example is the "spiral model" of software development, published by Boehm in 1988 while he was working for the TRW Defense Systems Group. The model came about after Boehm's attempts to apply the waterfall model to large government projects, when he realized that projects do not proceed from start to finish, as in the waterfall model, but rather in a number of iterations of the project phases.[16]

The spiral model is shown in Figure 3.4. The waterfall model is embedded in the lower right quadrant of the spiral model, where we find all the phases of the waterfall model: specification (here called

[16]Boehm 1988 p. 64.

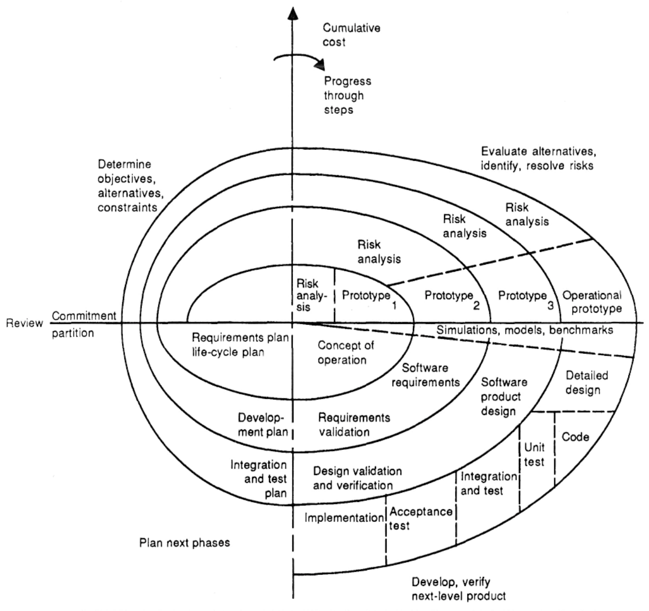


Figure 3.4: The spiral model. From Boehm 1988.

"requirements"), design, coding, testing, and operating (here called "implementation"). The prototype, simulation and benchmark phases in the model are somewhat optional.[17] Apart from the lower right quadrant, the largest part of the model is therefore concerned with administrative management: activities that concern documentation and planning (requirements planning, development planning, integration planning, commitment review, determining objectives, risk analysis and evaluation).

Robert Grady of the Hewlett-Packard Company has taken the emphasis on administrative management even further. In a 1997 book about software process improvement, he presents a model that is essentially a copy of Boehm's spiral model. Grady's model, shown in

[17]Boehm 1988 p. 65.

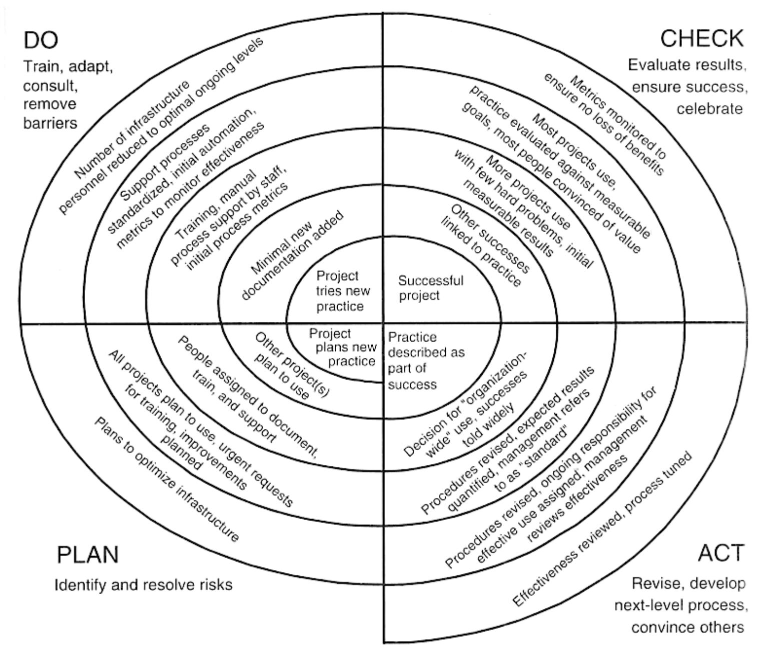

DO
Train, adapt, consult, remove barriers
CHECK
Evaluate results, ensure success, celebrate
Project tries new practice
Successful project
Project plans new practice
Practice described as part of success
Minimal new documentation added
Training, manual process support by staff, initial process metrics
Support processes standardized, initial automation, metrics to monitor effectiveness
Number of infrastructure personnel reduced to optimal ongoing levels
Other successes linked to practice
More projects use with few hard problems, initial measurable results
Most projects use, practice evaluated against measurable goals, most people convinced of value
Metrics monitored to ensure no loss of benefits
Other project(s) plan to use
People assigned to document, train, and support
All projects plan to use, urgent requests for training, improvements planned
Plans to optimize infrastructure
Decision for "organization-wide" use, successes told widely
Procedures revised, expected results quantified, management refers to as "standard"
Procedures revised, ongoing responsibility for effective use assigned, management reviews effectiveness
Effectiveness reviewed, process tuned
PLAN
Identify and resolve risks
ACT
Revise, develop next-level process, convince others

Figure 3.5: "Spiral model for process-improvement adoption." From Grady 1997.

Figure 3.5, is entirely removed from programming practice. It is oriented instead towards company organization and the bureaucracy needed to maintain a sizeable organization – meaning chiefly documentation, training, standardizing, and gaining company-wide support.

The Rational Unified Process is a process model currently promoted by IBM. It was originally created by Grady together with Ivar Jacobson of the telecommunications company Ericsson, and with James Rumbaugh of General Electric. The so-called "process structure" of the model can be seen in Figure 3.6. We recognize the usual phases of the waterfall model: requirements, design, implementation (cod-

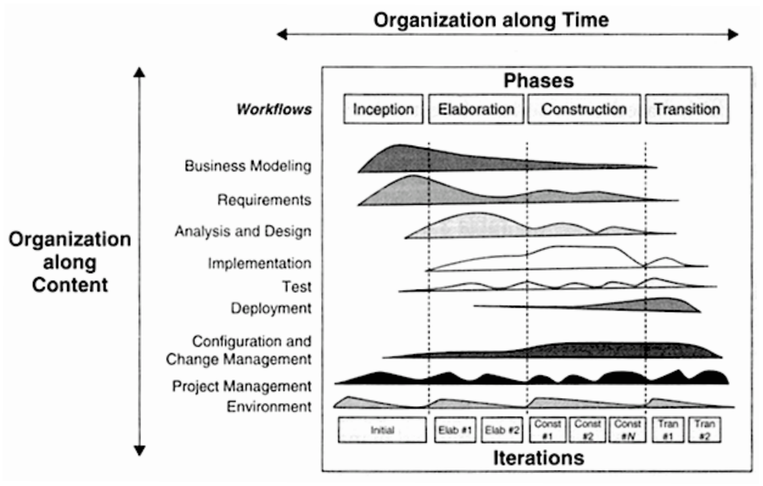


Figure 3.6: Process structure of the Rational Unified Process. From van Vliet 2008.

ing), test and deployment – except now they are termed “workflows”, and supplemented with a few more, such as business modelling and project management. The workflows are now executed concurrently in four phases, and each phase consists of a number of iterations. The irregular shapes in the diagram show the approximate intensity of each workflow in each of the iterations.

As is apparent, the diagram in Figure 3.6 is not of great practical use alone. It has to be supplemented by a wealth of other diagrams, and the Rational Unified Process is arguably not meant to be used without a large number of computer programs that support working with it, developed by IBM. It is a complicated model, meant for projects with complex administration.

We can now begin to see why software engineering thinking is so preoccupied with documents. A process model, in Boehm’s words, consists of a number of stages arranged in order, along with transition criteria for moving from one stage to the next. Documents are important because they serve as transition criteria. When a document has been produced and approved, the process can move to the next stage and development can progress.

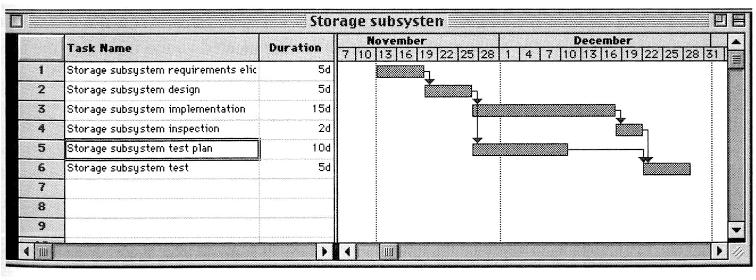
An extension of this way of thinking is to regard documents as input and output to the stages. The documents said to be input to a stage are those that need to be approved before the stage begins. The output are the documents that have been produced when the stage is declared to have ended.

As mentioned before, documents are often called work products or artefacts, and they are not restricted to be documents in the usual sense, but can also, for example, be the source code for programs. The most important aspect of a document in this sense is that it serves as a transition criterion that is tangible and can be checked. Thus the purpose of an individual document is not regarded as its most significant trait – executable code, structure diagrams and budget plans are all work products, regardless of their widely differing purposes.

If we turn our attention to the planning side of software engineering, we find that the documents that serve as criteria have a temporal counterpart, namely milestones. Milestones are the dates that separate phases from each other. When the documents that are associated with a given milestone have been approved, the milestone has been cleared and the project can progress.

The dominant trend of project management within software engineering is thus concerned with meeting the appropriate milestones by completing the required phases to produce the associated documents. This is ensured by allocating sufficient resources to each phase. Resources can be material things, such as machines, but they are first and foremost qualified personnel. A common way of administrating milestones, phases and resources is by way of the so-called Gantt chart, shown in Figure 3.7, which is a variant of the PERT method espoused by Brooks (a PERT chart is shown in figure 3.1).

The Gantt chart shows the various tasks belonging to a project phase and how the tasks are connected to each other. By estimating how long each task will take and how many resources it requires, the



	Task Name	Duration
1	Storage subsystem requirements elic	5d
2	Storage subsystem design	5d
3	Storage subsystem implementation	15d
4	Storage subsystem inspection	2d
5	Storage subsystem test plan	10d
6	Storage subsystem test	5d
7		
8		
9		

Figure 3.7: Gantt chart. From Bruegge & Dutoit 2000 [2010].

Gantt chart allows the engineers to compute the completion date of each milestone and hence the time when the project documents will be finished. If a task is delayed, it will push back all subsequent tasks that depend on it, with the result that the whole project is delayed. When the final milestone is has been met, the project is finished.

3.2.1.5 *Requirements engineering*

The requirements, or specification, of a software project tell programmers and managers what the users, or buyers, want the software to do. The requirements are thus of singular importance to a project; how could it be possible to make a program if one were not aware of what it is supposed to do? Important as they are, specifying the requirements brings about a host of problems, for, in Brooks' words, "the hardest single part of building a software system is deciding precisely what to build."[18]

Consequently, requirements engineering is a subfield of software engineering concerned with requirements. The most obvious approach to deciding what to build is simply to ask the intended users what they want. However, this is rarely sufficient in practice says Brooks, "for the truth is, the clients do not know what they want."[19] A common approach when the users have stated what they think they

[18]Brooks 1986 [1995] p. 199.
[19]Ibid.

want is to try to separate the "what" into two questions: "why" they want it, which is then their business goals, and on the other hand "how" it can be achieved, which is then a technical problem left for the engineers to solve.[20]

If we look at the software engineering process models, we see that requirements are generally relegated to a separate phase in the beginning of the process (with the exception of the Rational Unified Process, in which requirements are a separate workflow). The ideal is that, first, the requirements are discovered out and carefully written down in a requirements specification, after which the development can proceed unhindered by doubts about what the system is really supposed to do.

However, this ideal of separating the requirements specification from the rest of the development process is just that – an ideal – and it is seldom met in practice. C.A.R. Hoare, in his 1980 Turing Award Lecture, stated that "the design of a program and the design of its specification must be undertaken in parallel by the same person, and they [the designs] must interact with each other."[21] In the original article describing the waterfall process, Royce states that: "For some reason what a software design is going to do is subject to wide interpretation even after previous agreement."[22]

When writing a requirements specification, it is of fundamental importance that the software engineer has sufficient understanding of what the user is trying to do. Otherwise, the requirements will express what the user literally says, but not what the user wants. This kind of knowledge of the user situation is commonly called "domain knowledge".[23] It is often contrasted with software engineering knowledge, or programming knowledge, which is knowledge about how to make the programs that is implicitly supposed to be independent of the domain in which the program is to work.[24]

[20]Lauesen 2002 p. 29.
[21]Hoare 1980 [1981] p. 79.
[22]Royce 1970 p. 335.
[23]Lauesen 2002 p. 20.
[24]Ibid. p. 26.

In accordance with the general software engineering focus on documents, the literature on requirements engineering devotes much attention to notation systems and formats for writing things down. Søren Lauesen's 2002 textbook on requirements engineering, for example, describes writing down requirements in the form of state diagrams, state-transition matrices, activity diagrams, class diagrams, collaboration diagrams, and sequence diagrams. Not surprisingly, most of these forms of diagrams are originally made for program design and thus meant to describe computer programs.

3.2.2 *Agile software development*

3.2.2.1 *The origins of the Agile movement*

The concept of Agile software development refers to a very diverse set of approaches, whose underlying philosophies loosely share common themes. The software engineering approaches, as described in section 3.2.1 (Software engineering), are to a large degree institutionalised in university departments, professional associations, well-defined processes and standards, and so on. In contrast, the Agile movement is less organized, making it difficult to specify exactly what is Agile and what is not. Approaches that are "outside" the movement are also sometimes considered Agile so long as they share the same basic attitude to development.

The basic characteristics of Agile methods is that they focus on rapid change and "lightweight" processes. This contrasts with the traditional software engineering focus on "heavyweight" processes and "plan-driven" development. The Agile movement emerged among programmers who were familiar with software engineering thinking, and thus Agile development is both derived from software engineering and formed in opposition to it. Understanding this tension within Agile thinking is important for understanding the underlying philosophy.

The Agile movement started well before it gained its current name. For example, the influential book *Peopleware* from 1987[25] brought to light a focus on the importance of teamwork that is central to many Agile methodologies, and during the 1990s well-known Agile methodologies, such as Dynamic Systems Development Methodology and Extreme Programming, were developed. The movement gained its name in 2001, when a group of influential software professionals collaborated on the Agile manifesto, in which they agreed on some core principles of their very different methodologies.

Before the publication of the Agile manifesto, the movement received little attention and was, in general, not taken seriously within academia and software engineering.[26] Although the movement gained more acceptance and interest from 2001, it continued for some years to hold academic conferences separately from mainstream software engineering.[27] However, by 2014 Agile development has become successful, as large companies such as the telecommunications firm Ericsson are seeking to become more agile.[28] As with all successful movements, claiming to be "agile" has now become fashionable.[29]

[25]DeMarco & Lister 1987.

[26]E.g.: " ... academic research on the subject is still scarce, as most of existing publications are written by practitioners or consultants." Abrahamsson et al. 2002 p. 9.

[27]"In a short time, agile development has attracted huge interest from the software industry. ... In just six years the Agile conference has grown to attract a larger attendance than most conferences in software engineering." Dybå & Dingsøyr 2008 p. 5.

[28]Auvinen et al. 2006: "Software Process Improvement with Agile Practices in a Large Telecom Company."

[29]"Even if the Extreme Programming philosophy advocates a complete development methodology that seems to make sense, in practice it's often just used as an excuse by programmers to avoid designing features before they implement them." Spolsky 2004 p. 243.

3.2.2.2 *The Agile manifesto*

The Agile manifesto[30] is shown in Figure 3.8. It consists of four pairs of values, where each pair has a preferred value. After that follows 12 principles that give some methodological consequences of the values. It is clear that the values and principles as they are written are open to different interpretations – the creators intended it that way. This means that the manifesto alone is not enough to understand Agile practices.

For example, the phrase "we value individuals and interactions over processes and tools" can be interpreted in a number of ways. However, this does not mean that tools can be ignored; some methodologies place great emphasis on tools. In Extreme Programming, for example, tools for automated testing and version control are important. Rather, the sentence says something about the way tools should be used.

By referring to themselves as anarchists ("Seventeen anarchists agree … "), the creators of the manifesto show that they are conscious of a break with mainstream software engineering traditions. At the same time they are hinting that the Agile movement is a diverse and non-dogmatic movement.[31]

The break with tradition is also seen in the way in which the core Agile values – individuals and interactions, working software, customer collaboration, responding to change – are contrasted with what can be considered the foundational values of software engineering: processes and tools, comprehensive documentation, contract negotiation and following a plan. However, the creators do not dismiss the traditional software engineering values: on the contrary, they acknowledge their worth (" … while we value the items on the right

[30]The manifesto was published on the Agile Alliance's website in 2001. *Dr. Dobb's Journal* also published it online in August 2001 with comments by Martin Fowler and Jim Highsmith. An edited version with comments is published as an appendix in Cockburn 2001.

[31]The story about being diverse and non-dogmatic is repeated in the comments to the manifesto, in which the creators stress that they are "fascinated" that they could agree on anything. Fowler & Highsmith 2001. Also that they "hoped against hope" that they could agree on something. Cockburn 2001 p. 177.

The Manifesto for Agile Software Development
Seventeen anarchists agree:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

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

That is, while we value the items on the right, we value the items on the left more.

We follow the following principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.

Figure 3.8: The Agile manifesto. This is the version published in Dr. Dobb's Journal*, August 2001. (Continued on the next page.)*

- Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity–the art of maximizing the amount of work not done–is essential.
- The best architectures, requirements and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

—**Kent Beck**, **Mike Beedle**, **Arie van Bennekum**, **Alistair Cockburn**, **Ward Cunningham**, **Martin Fowler**, **James Grenning**, **Jim Highsmith**, **Andrew Hunt**, **Ron Jeffries**, **Jon Kern**, **Brian Marick**, **Robert C. Martin**, **Steve Mellor**, **Ken Schwaber**, **Jeff Sutherland**, **Dave Thomas**
www.agileAlliance.org

Figure 3.8: The Agile manifesto. (Continued from the preceding page.)

. . . "). Thus, they are not only conscious of a break with tradition, but also of having thir roots within that tradition and of being indebted to it.

3.2.2.3 *Scrum as an example*

It is easier to understand the reasoning behind Agile through a concrete example than through discussing the Agile values in the abstract. We will therefore take a look at the Scrum methodology, a well-known Agile methodology that was developed principally by Ken Schwaber and Jeff Sutherland during the 1990s.[32] The word "Scrum"

[32]The presentation in this section is based mainly on Ken Schwaber's 2003 book: *Agile Project Management with Scrum.*

itself comes from rugby football; it means that all players huddle together around the ball to restart play following an interruption.

There are three roles in a Scrum project, and anyone that is not in one of these three roles is regarded as a spectator or an interested party without direct authority. The roles are: Product Owner, ScrumMaster, and Team.

The Product Owner is the representative of the customers or the users of the software. It is his responsibility to provide the Team with a list of features to implement, to take business decisions, and to prioritise the features for the Team so that they know what they should work on first.

The ScrumMaster is responsible for ensuring that the Scrum process is being followed. The ScrumMaster is the representative of the Team and acts in the Team's interest to make sure that it has the necessary conditions needed for doing the work. The ScrumMaster is also responsible for teaching the Product Owner how to use the Scrum process to get the maximum business value from the Team's work. The ScrumMaster role can be filled by anyone except the person acting as Product Owner.

The Team comprises the people doing the work. The optimal Team size is around seven persons. If the Team is many fewer than seven, Scrum is not needed; if there are many more , it is best to organize them in several Teams working together on a project. The Team must be composed of competent individuals and there is no hierarchy of authority within it. The Team manages its work itself with the help of the input it gets from the Product Owner, and no one outside the Team can dictate its work – not even the ScrumMaster.

The team works in periods of 30 days, called Sprints. Before each Sprint, the Team meets with the Product Owner, who presents the work he would like to be done and prioritizes it. The Team then selects an amount of work that it thinks it can handle within one Sprint and commits to doing it. After the Sprint the team again meets with the Product Owner and other interested parties and presents all the work it has completed during the last Sprint. The Team and the

Product Owner plan the following Sprint based on what the Product Owner now thinks are the most important features.

Before starting a new Sprint, the Team also holds a meeting in which it reviews the last Sprint and assesses whether the working process has any problems and whether it can be improved. This Sprint review meeting is the main mechanism for continuous process improvement in Scrum.

During each Sprint, the Team holds a 15-minute meeting every day, called the Scrum meeting. Each member of the Team answers three questions during the meeting: What have you worked on since last meeting? What will you work on until next meeting? Are there any hindrances to your work? Only the Team members and the ScrumMaster are allowed to talk during the Scrum meeting.[33] Other people can observe, but they must remain silent.

These are the basic rules of working with Scrum. The complete set of rules runs to a mere seven pages.[34] Thus, the rules necessarily leave many practical questions unanswered. Scrum does not provide detailed rules for how the Team should work; rather, it provides a basic frame of rules around which the Team itself has to figure out how to work.

This means that in order to apply the Scrum rules correctly, one has to know the motivation for why the rules are there. Scrum is built on three core principles: transparency, inspection, and adaption. All of the rules and practices of Scrum are designed to facilitate one or more of these three principles; if a rule is used in a way that goes against the principles, the Scrum process does not work as intended.

For example, Schwaber describes a Team that was following the letter of the Scrum rules, but it was "hiding" behind task descriptions that were so abstract that it was not possible to tell how much work

[33]And in principle also the Product Owner.

[34]Schwaber 2003, Appendix A. For comparison, the software engineering DOD-STD-2167 Military Standard for Defense System Software Development is 90 pages long – U.S. Department of Defense 1985. The software requirements for the safety critical engineering process IEC 61508-3 are 112 pages long – International Electrotechnical Commission 2009.

was actually required to complete the tasks.[35] Thus, the Team violated the principle of transparency, which dictates that all relevant information must be clear and accessible to everyone involved. In turn, the violation of the transparency principle meant that the team lost the ability to follow the other principles, inspection and adaption. Thus, while the letter of the rules is not always important in Scrum, even a small deviation from the core principles can have a detrimental effect on the project.

My own interpretation of Scrum is that the methodology is, on one hand, oriented towards the practical day-to-day tasks of development. On the other hand, it rests on a subtle but coherent philosophical system that means that the details of the methodology must be understood and cannot be changed haphazardly. The philosophical system in turn rests on a certain work ethic. Scrum demands that the Team members have an attitude to their work that is consistent with the underlying values and principles of the methodology.

3.2.2.4 *Programming practice*

As discussed above, the Agile movement consists of a host of different approaches. Some of the approaches considered to be Agile are Extreme Programming, Adaptive Software Development, Feature-driven Development, Dynamic System Development Methodology, Lean Development, Adaptive Software Development, Scrum, and the Crystal methodologies.[36] There are also approaches that, even though they are not part of the Agile movement as such, can be considered Agile in their spirit, either because they have influenced the Agile movement or because they are similar in their practices to Agile methodologies.

The 1987 book *Peopleware* is not a development methodology, but the authors wrote about the importance of teamwork and self-organization in software development, themes that are central to Agile methodology, and their ideas have influenced later Agile approaches.[37]

[35]Schwaber 2003 p. 96 ff.

[36]See Fowler & Highsmith 2001; Highsmith 2002; and Cockburn 2001.

[37]DeMarco & Lister 1987. For *Peopleware's* influence on Agile methods, see for example Cockburn 2001 p. 56, p. 80.

The popular weblog writer Joel Spolsky has developed a "Joel Test" of 12 easily checked, practical working conditions, which can be said to embody many of the same values as Agile development.[38] Andrew Hunt and David Thomas are both co-creators of the Agile manifesto, though they "have no affiliation to any one method."[39] Their influence is through the book *The Pragmatic Programmer* which, as the title suggests, contains practical advice on a fairly concrete level.[40]

Thus, there is a large diversity within the Agile movement. The same ideas are often found in methodologies that otherwise differ quite much, and the sources of inspiration for the Agile methodologies are varied, even eclectic. These aspects of the Agile movement are all due to the fact that Agile is very much grounded in programming practice. Programming practice develops with its own local particularities and quirks, and this is reflected in the Agile movement.

To an extent, Agile is based on common sense, on what the programmer would do in any case – at least, this is how the Agile movement sees itself. In the words of the Agile practitioner Jim Coplien: "Everyone will like Scrum; it is what we already do when our back is against the wall."[41] Schwaber relates a story about a product manager that applies Scrum, not in the form of the usual rules, but in the form of simple common sense.[42]

The emphasis on practicality and common sense has some consequences in practice that distinguishes the Agile movement from software engineering. For example, as we saw in section 3.2.1.2 (Brooks), Brooks took the documentation for a project to be the starting point for thinking about how the project should be managed, stating that five formal documents is the basis for all management decisions.

In contrast to this, Alistair Cockburn repeatedly stresses that documentation merely has to be good enough for its purpose, and that how good this is always depends on the situation.[43] In many situ-

[38]Spolsky 2004 chp. 3. First published online August 2000.
[39]Cockburn 2001 p. 177.
[40]Hunt & Thomas 1999.
[41]Schwaber & Sutherland 2010 p. 2.
[42]Schwaber 2003 p. 60.
[43]Cockburn 2001 p. 18, p. 63, p. 147 f.

ations the important information is best conveyed in person, and a formal document would be superfluous and even undesirable. This is of course a pragmatic approach that in each particular situation takes into account what the purpose of the documentation is.

As we saw with Scrum in section 3.2.2.3 (Scrum as an example), Agile methodologies rest partly on an underlying philosophical system and partly on principles that derive from concrete programming practice. Each Agile methodology contains a particular mix of philosophy and development practice. Scrum is quite explicit about its core values, without becoming a treaty on philosophy. On the other hand, Extreme Programming is very much based on concrete practices, and demands that programmers always work together in pairs, that all program code is tested every day, and that everyone is sitting in the same room or building – things that the slightly more abstract rules of Scrum leave up to the developers themselves.

In general, Scrum instructs people *how* they should work, not *what they should do* in their work. What they concretely have to do is something they have to know already – it is something that stems from their previous experience, their tacit knowledge, and their culture.[44] This necessary pre-existing experience and ability is treated differently by the different Agile methodologies. Extreme Programming, for example, is more particular about "engineering excellence" than is Scrum.[45]

In a nutshell, Scrum can be summed up as a project management philosophy that is based primarily on programming practice instead of on software engineering theory. To some extent, this is true of all Agile methodologies.

3.2.2.5 *Work ethics*

Just as the philosophical foundation of Scrum rests on a certain work ethics, so doother Agile methodologies. Agile methodologies require

[44]Cockburn in particular points to the importance of the developers' culture. Cockburn 2001 p. 30, p. 55, p. 92. Bear in mind that their knowledge of programming is part of their culture, among other things.

[45]Schwaber 2003 p. 107.

that participants care about their work, and that they care about it in a certain way. Doing a good job is its own reward, and to be able to help other people out is the highest purpose of a job well done.

Ken Schwaber writes of a company that he helped to implement Scrum: "When I last visited Servicelst, it was a good place to visit. I could watch people striving to improve the organization, the teams, themselves, and their profession. I was proud to be associated with them. ... What more can you ask from life?"[46] The quote expresses the sentiment that the work ethic – striving to improve – is the most important part of the changes brought about by Scrum, and that merely being associated with these working people is reward enough for Schwaber.

The opening sentence of Jim Highsmith's explanation of Agile development is a quote from a project manager: "Never do anything that is a waste of time – and be prepared to wage long, tedious wars over this principle."[47] This stresses that Agile methodologies are not for people who do not care for their work, or who take the path of least resistance; and it indicates that to uphold certain work ethics comes with a cost. Agile practices can be difficult to implement because the benefits do not come without hard choices.

Alistair Cockburn writes of the kind of people that make projects successful, who Agile methodologies try to encourage: "Often, their only reward is knowing that they did a good deed, and yet I continually encounter people for whom this is sufficient."[48] He relates a story of the teamwork in a hugely successful project: "To be able to get one's own work done and help another became a sought-after privilege."[49]

In *The Pragmatic Programmer* a similar attitude to caring about one's work is displayed:

[46]Schwaber 2003 p. 118.

[47]Highsmith 2002 p. 4.

[48]Cockburn 2001 p. 63.

[49]Dee Hock, 1999, on the first VISA clearing program. Quoted in Cockburn 2001 p. 64.

> "Care about your craft. We feel that there is no point in developing software unless you care about doing it well. ... In order to be a Pragmatic Programmer we're challenging you to think about what you're doing while you're doing it. ... If this sounds like hard work to you, then you're exhibiting the *realistic* characteristic. ... The reward is a more active involvement with a job you love, ... "[50]

The Agile movement is of course not alone in having an ethical foundation. Software engineering, for example, is founded on a professional ethic, in contrast to the work ethic of Agile. The software engineering professional ethic is directed less toward the work being its own reward, than it is toward upholding a professional community. The core values of the software engineering ethic are: to be rigorous, to be scientific, and to do good engineering.

In Scrum, the value of working in a certain way is only justified by the value of the outcome. To work according to scientific principles, for example, is only valued if the situation calls for it. The rules of Scrum are really an insistence on the absence of unnecessary rules. Ideally there would be no rules – only competent, cooperating individuals with common sense. However, since rules cannot be avoided altogether, the rule set at least has to be minimal, such that if someone tries to impose new rules, one can reasonably balk, by objecting, "that's not Agile!"

The consequence of the Scrum rule set is of course to give executive power to the programmers. They determine how to do their job, and it is the ScrumMaster's task to ensure that no one interferes with them. Fittingly, Schwaber likens the ScrumMaster to a sheepdog, so the ScrumMaster should not really be seen as a master who commands, but more as one who serves and protects his flock.[51]

The Agile insistence on giving the programmers executive power parallels Brooks' advocacy for giving programmers executive power

[50]Hunt & Thomas 1999 p. xix f.

[51]Schwaber 2003 p. 16.

by placing them at the top of the decision making hierarchy – see section 3.2.1.2 (Brooks). However, while Brooks' model has never really caught on, Scrum has enjoyed widespread success. The main reason is probably that unlike in the hierarchical model, with Scrum the programmers' power is strictly delimited. They reign absolutely, but only over their own work process. The Product Owner has a clear right and mechanisms to decide what they should work on, and he has frequent inspection meetings to see that his wishes are carried out.

In Scrum, along with the power to decide over their own work, the programmers also receive the full responsibility of organizing themselves, alongside the demand that they are mature enough to do this. For this reason, the organization of a team of peers without formal hierarchy is an important part of Scrum. In Agile thinking this is called self-organization, and it is a central concept.

Agile methodologies are advertized as being more effective and deliver more value. However, in Agile thinking the effectiveness is a consequence of the ethics: it is not that the most effective system was found, with an ethical part added as an afterthought. Non-Agile methodologies can be effective, but if their ethics are very different, the results of their effectivity will be different as well.

3.2.2.6 *Relationship to software engineering*

As discussed above, the Agile methodologies are both derived from software engineering and developed in opposition to it. Proponents of Agile methodologies sometimes downplay this opposition in order to make their methodologies more palatable to traditional software engineers.

For example, Kent Beck, one of the creators of Extreme Programming, pictures a continuous evolution that proceeds as a matter of course from the earliest software engineering model, the waterfall model (see Figure 3.2), over iterative development, and ending in Extreme Programming.[52] Beck's illustration of this evolution is depicted

[52]Beck 1999.

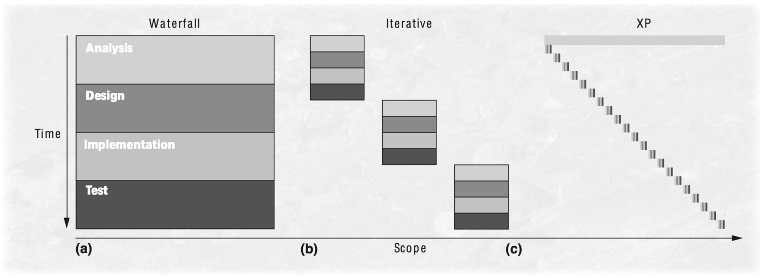


Figure 3.9: "The evolution of the Waterfall Model (a) and its long development cycles (analysis, design, implementation, test) to the shorter, iterative development cycles within, for example, the Spiral Model (b) to Extreme Programming's (c) blending of all these activities, a little at a time, throughout the entire software development process." From Beck 1999.

in Figure 3.9. With this evolutionary model, Beck is portraying his Agile methodology as a continuation of software engineering practices, and stressing the commonality between software engineering and the Agile movement.

The evolutionary model conceals the philosophical differences between software engineering and the Agile movement. Seen from the perspective of the underlying philosophy, Agile is not so much a gradual change in tradition as it is a radical shift in thinking.

Nevertheless, Agile has clear ties to the engineering tradition of software engineering and this is an important part of the movement's self-perception. Both Schwaber and Highsmith conceptualize the difference between Agile and software engineering's plan-driven methods as being between "empirical processes" and "defined processes".[53] The concepts of empirical and defined processes come from chemical engineering process theory.[54]

The Lean approach to development has even stronger ties to traditional engineering, because it was originally developed in the Toyota

[53]Schwaber 2003 p. 2. Highsmith 2002 p. 4.
[54]Schwaber 2003 p. 146.

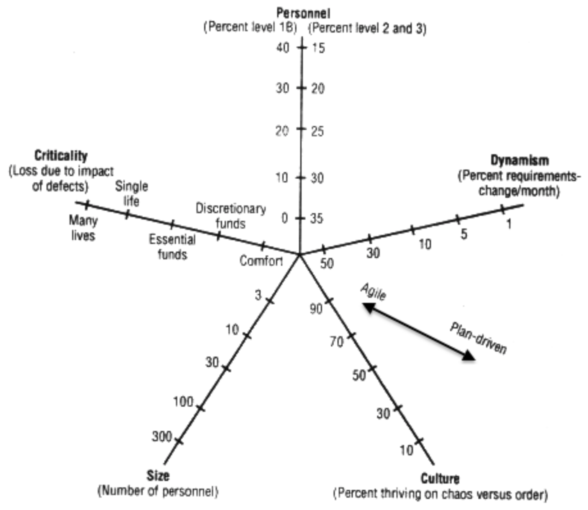


Figure 3.10: Agile and plan-driven homegrounds. From Boehm & Turner 2003.

Production System for car manufacturing in the 1980s. The system was later adapted to an Agile software methodology, primarily by Mary and Tom Poppendieck.[55]

As the popularity of Agile methodologies has grown, so has the interest from software engineering researchers in explaining these methodologies in software engineering terms. One example of this is a model proposed by Boehm and Turner in the 2003 article "Using Risk to Balance Agile and Plan-Driven Methods". This model, shown in Figure 3.10, essentially presents Agile and plan-driven methods as two alternatives, with the choice between them depending on five factors. Three of the factors are in reality matters of opinion rather than factual assessments.

[55]Heidenberg 2011 p. 10 f.

One of these three factors is whether the personnel “thrives on chaos”. This is a statement so vague that it cannot be assessed factually. The second is what proportion of requirements changes per month. Requirements change can be measured, but measuring lines changed, number of features changed, or estimated amount of rework changed will result in different figures – and in the case of measuring estimates, the measure in practice becomes guesswork. The third factor is the competence of the personnel, but again, the percentage that is “able to revise a method, breaking its rules to fit an unprecedented situation”[56] is a criterion so vague that it becomes a matter of opinion.

That leaves two factual factors: the number of personnel, and the criticality of the software. It is indeed the case that, generally speaking, larger and more critical projects require a larger bureaucracy. Cockburn’s Crystal family of methodologies is an Agile attempt to address these points.[57] However, to make the choice of methodology a matter of size, as Boehm and Turner do, is to overlook the fundamental differences in philosophy between Agile and software engineering thinking. From an Agile point of view, the answer to larger projects and bureaucracy is to make the bureaucracy more Agile – not to discard the philosophy in favor of a software engineering model.

The traditional software engineering focus on planning and documentation is not surprising regarding the history of computer use. Computers were first developed in settings that traditionally involved strong bureaucracy and hierarchy: military staff and governmental administration. According to Max Weber, one of the founders of modern sociology, bureaucracy is characterized by making explicit rules and upholding them through the use of well-defined duties, authorities, and qualifications.[58] The emphasis on these concepts is apparent in software engineering, where they are often implemented as processes (rules), roles (duties), standards (authority), and certifications (qualifications).

[56]Boehm & Turner 2003 p. 60.
[57]Cockburn 2001 chp. 6.
[58]Weber 1922 [2003] p. 63.

This bureaucratic setting is often in conflict with the fact that programming as an activity is intensely creative. The importance of the creative aspect of programming is noted both by Brooks[59] and by Peter Naur,[60] famous for his contribution to the Algol 60 programming language and winner of the Turing Award for computer science. If the programmer spends most of his time administrating bureaucratic rules, he will be creative in rule-making, not in programming. The Agile movement can be seen as the creative programmers' reaction to the bureaucratic traditions of software engineering.

However, the tension between bureaucracy and creativity in programming has deeper roots than the conflict between Agile and software engineering regarding project management philosophy, for the very origins and *raison d'être* of computers lie in bureaucratic organizations. The bureaucratic origins of programming are thus at odds with its inherent creativity.

As an example of the practical consequences of this tension, we saw in section 3.2.1.4 (Systems thinking) that the various documents in a software engineering process are, in principle, all equal. From a bureauratic point of view, their value lies in their function as transition markers, which allow the process to proceed to the next step. The actual content of the work products is not irrelevant, but secondary – it is less important, as long as it lives up to the criteria for transition.

By contrast, a core Agile principle is that working, demonstrable program code is the primary measure of success. Other work products only have meaning through their contribution to the working code.[61] Thus, in Agile methods, a piece of documentation is usually not produced unless there is a compelling reason to make it. Conversely, in software engineering a piece of documentation will generally be made unless there is a provision in the process to skip it.

The tension between bureaucracy and creativity in programming highlights a fundamental difficulty of software development. Software engineering thinking maintains that software development can and

[59]Brooks 1975 [1995] p. 7.
[60]Naur 1985 [2001] p. 194 f.
[61]Cockburn 2001 p. 37.

should be equivalent to ordering a car part from a factory.[62] From an Agile perspective, software development is a collaboration with the customer, and the customer needs to be involved as much as the programmers. In chapter 6 (Safety critical programming) we will take a closer look at what software development and car manufacturing have in common in practice.

3.2.3 *Computer science*

3.2.3.1 *Science, mathematics, formal systems*

Computer science is the most prominent of the academic fields that study computers and programming. It was proposed in 1959[63] and created as a university field in the U.S. a few years afterwards. Software engineering is sometimes regarded as a subfield of computer science and this is true to a degree but the two schools of thought rest on somewhat different academic traditions. Until around 1970 most people who had anything to do with programming computers were scientists, mathematicians, or electrical engineers.[64] While software engineering has its origins in the intellectual tradition of electrical engineers, computer science is dominated by the perspectives of scientists, who need computations, and mathematicians.

Howard Aiken, who developed some of IBM's influential first automatic computers, imagined in 1937 that computers would be used for scientific calculation in, for example, theoretical physics; and his idea was to use computers for things that required a great deal of calculation and tabulation, for example numerical differentiation.[65] This vision of computers was of specialized machinery for narrow scien-

[62]Czarnecki & Eisenecker 2000 p. 1 f.

[63]Fein 1959: "The Role of the University in Computers, Data Processing, and Related Fields."

[64]Mahoney 1990 p. 328.

[65]Aiken 1964 p. 192 ff, p. 196.

tific fields, a long way from the general appeal of applications such as spreadsheets and accounting.

In 1981, a programming guide for microprocessors gives laboratory work as an example of what computers might be used for. This is not the specialized scientific calculations envisioned by Aiken, but a practical device that can ease the laboratory worker's daily life: "There are many applications for division, but one of the most common is in taking the average of a set of numbers – perhaps the results of a series of laboratory tests."[66]

The famous Dutch computer scientist Edsger W. Dijkstra regards programming as a branch of applied mathematics and, at the same time, a branch of engineering. This makes for a neat division of computing into computer science on one hand and various flavours of engineering on the other. The emphasis on mathematics is shared by Lawrence C. Paulson, professor of programming languages, who advocates a form of programming – functional programming – that consists of expressions that obey mathematical laws.[67] Functional programming has met with limited success in the programming industries, but has been popular within academia.

For Paulson, the aim of functional programming is to make programs easier to understand by mathematical reasoning.[68] In addition, the use of mathematical analysis is said to break bad habits that programmers form through the use of low-level programming languages,[69] where "low-level" means that the programming languages have a fairly simple and direct relationship between how code is written and what the computer does.[70] In other words, direct manipulation of the machine results in bad habits, and mathematical reasoning is the way to combat this.

[66]Scanlon 1981 p. 117.

[67]Paulson 1991 [1996] p. 2 f.

[68]Ibid. p. 10.

[69]Ibid. p. 12.

[70]This does not mean, however, that they are simple to use. Many of the complexities that "high-level" programming languages introduce are things that make the programming task easier for the programmer.

Some computer scientists place great emphasis on formal logic and formal systems, and regard these as the essential characteristics of computer science. According to one textbook of logic in computer science:

> "The aim of logic in computer science is to develop languages to model the situations we encounter as computer science professionals, in such a way that we can reason about them formally. Reasoning about situations mean constructing arguments about them; we want to do this formally, so that the arguments are valid and can be defended rigorously, or executed on a machine."[71]

The reasoning here is that formal arguments lead to valid arguments that can be defended rigorously. Another example of thinking that is centered on formal systems comes from a textbook in type theory. Type systems is a technique that is used in every modern programming language; and this textbook states that type systems are light-weight formal methods that can be applied even by programmers unfamiliar with the underlying theory.[72] To view type systems in this light emphasizes formal systems much more than as a natural development of programmers' need to keep different kinds of numbers and letters separate from each other.

3.2.3.2 *Algorithms and data structures*

Computer science's primary study object is programs, or programming languages. Therefore, it makes sense to try to establish how programs are perceived by computer scientists: that is, what do they mean when they speak of a program? The authors of the famous textbook *The C Programming Language* write that C consists of data types and structured types, mechanisms for fundamental control flow, and mechanisms for separating computation into functions and for

[71]Huth & Ryan 2000 [2004] p. 1.

[72]Pierce 2002 p. 1.

separate compilation.[73] Schematically, a programming language, and by extension a program, can thus be described as:

> structured data + fundamental control flow + modular mechanisms.

Data structures are descriptions of how things, i.e. numbers and text, are placed in the machine memory. Control flow is the mechanism to describe algorithms: an algorithm is a sequence of steps that the computer carries out; it is "what the program does". The things that an algorithm operates upon are the data. Modular mechanisms tie smaller program modules together into larger programs.

Charles Simonyi, a programmer who is famous for his work on Microsoft Word and Excel, has said that programming is a science, an art, and a trade.[74] The science is knowledge of the best algorithms to use; the art the ability to imagine the structure of the code; the trade is knowing the details of writing efficient code.

Bjarne Stroustrup, the creator of the widely used C++ programming language, has written that procedural programming – what he calls "the original programming paradigm" – amounts to the imperative: "Decide which procedures you want; use the best algorithms you can find."[75] He continues: "The focus is on the processing – the algorithm needed to perform the desired computation." This sums up neatly the perspective of much of classical computer science: it is first and foremost a science of algorithms.

A complementary view is presented by Naur, who places the emphasis not on algorithms but on their counterparts, data structures. For Naur, computer science is simply the description of data and data processes. For this reason, Naur wanted computer science to be called "datalogy" – "The datalogist describes data and data processes in a way similar to how the botanist describes plants."[76] The name caught on in the Nordic countries, where it is used to this day (In Danish: *datalogi*. In Swedish: *datavetenskap*, "data science").

[73]Kernighan & Ritchie 1978 [1988] p. 1.
[74]Lammers 1986 p. 15.
[75]Stroustrup 1985 [1997] p. 23.
[76]Naur 1995 p. 9.

Apart from the traditionally close relationship between science and mathematics, there is a concrete reason that science and mathematics unite in computer science around the study of algorithms. Scientific computing applications often consist of small pieces of code that execute millions of times.[77] This means that any change to the central piece of code that makes it more efficient can result in saving time, and thereby reducing costs. In turn, this has led to a focus on efficient algorithms – an area where mathematics has had early and large successes in the field of computing.

Neil Jones, professor of programming languages, provides a good example of a view of computer science that combines mathematics, algorithms, and formal systems. He regards a programming language, L, as a mathematical function that maps programs to their meanings:[78]

$$[\![\ ]\!]^L : \text{Programs} \to \text{Meanings}$$

An individual program, p, is similarly regarded as a mathematical function from input to output data:

$$[\![p]\!]^L : \text{Input} \to \text{Output}$$

This view of programs and programming languages leads, unsurprisingly, to a research focus on the mathematical functions that can be expressed by a computer. In mathematical terminology, these are called computable functions – in this context, this is another name for algorithms.

3.2.3.3 *Abstraction*

A common theme in computer science thinking is abstraction; both as an ideal and as an explanation of progress. The standard explanation for the evolution of the high-level programming languages of our day is thus:[79] In the beginning programs were written directly in binary

[77] Patterson & Hennessy 1997 p. 86.
[78] Filinski et al. 2005 chp. 1, p. 7.
[79] Patterson & Hennessy 1997 pp. 6–8.

machine code. From this evolved the slightly more abstract assembly code, in which programmers do not have to remember the numbers of locations in the machine's memory but can use names instead. After that came low-level programming languages which are even more convenient for the programmer to use and regarded as more abstract. Finally, there are the high-level programming languages, which are seen as the most abstract and which a programmer can use without knowing many of the technical details of the machines he is programming for. Thus, historically, what is today known as low-level programming languages was once thought of as high-level.

Abstraction in this sense means that the programming language does not closely describe what the machine does, but instead presents a symbolic notation – an abstraction – that is seen as easier to use. Abstract programming languages are considered to lead to a more natural way of thinking as well as being more productive, because the programmer has to write fewer lines of code.[80] Conversely, assembly language and low-level programming languages are considered to lead to an unnatural style of programming: "Assembly ... [is] forcing the programmer to think like the machine."[81]

According to Stroustrup, a programming language is a vehicle for machine actions, plus concepts to use when thinking.[82] This viewpoint is a variant of an underlying idea common to much of computer science: that a programming language is in some sense a way of thinking, and that abstraction is the ideal for thinking. Moreover, it is a common viewpoint that "every high-level language provides abstraction of machine services."[83] Combined with Stroustrup's definition, this implies that a good, high-level programming language provides abstract concepts to think about machine actions that are themselves presented as abstractions.[84] Jones writes that high-level programming

[80]Ibid.

[81]Ibid. p. 6.

[82]Stroustrup 1985 [1997] p. 9.

[83]Pierce 2002 p. 6.

[84]The general view described here is not exactly Stroustrup's own position. He advocates that the programming language, *sans* the concepts for thinking, should be "close to the machine" i.e. non-abstract. Stroustrup 1985 [1997] p. 9.

languages abstract away from any particular machine;[85] this viewpoint of course works well with an orientation of research away from engineering and towards mathematics.

The ideal of abstraction is not only applied to program code but also to the textbooks and articles that explain programming languages. To give an example, here is a textbook definition of what a type system is:

> "A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they produce."[86]

The definition is so abstract and general that it hardly enlightens anyone who does not already know what a type system is. A definition in plain language could be: a type system is a mechanism in a programming language that makes sure that different kinds of values do not accidently get mixed up; for example numbers and letters, or integers and fractional numbers.

According to the same textbook, the history of type systems begins in the 1870's with Gottlob Frege's works on formal logic.[87] However, Frege did not have programming languages in mind: type systems were first introduced to programming languages in the 1950s for reasons of efficiency.[88] To place the beginnings of type systems with Frege has the effect of maximising the emphasis on formal, logical, and mathematical aspects at the cost of downplaying practical aspects.

Finally, we should note with caution that abstraction is sometimes given an unrealistic importance, as if "more abstract" is the same as "better". In reality, to regard a program or programming language as abstract is simply a perspective. Any given program must run as concrete instructions on a concrete computer in order to be useful, no matter how "abstract" the programming language is. From the

[85] Filinski et al. 2005 chp. 1, p. 3.
[86] Pierce 2002 p. 1.
[87] Ibid. p. 11.
[88] Ibid. p. 8 f.

perspective of the machine that executes the program, the high-level programming language program is every bit as concrete as binary machine code – the layers of programs that make up a high-level programming language are perhaps thought of as abstractions, but they exist in a concrete form.

3.2.3.4 *Machine-orientation*

Despite, or perhaps because of, the emphasis placed on mathematics and logic in computer science, the thinking is remarkably oriented towards the machines, meaning that the world is seen almost exclusively in terms of what the computers can do, rather than what people do. We read in a quote above that "Reasoning about situations means constructing arguments about them; we want to do this formally, so that the arguments are valid and can be defended rigorously, or executed on a machine."[89] This quote can also be read as indicating that computer scientists should work with formal systems precisely because they can be executed by a machine. Thus, the notion of an abstract machine comes to dictate the focus of computer science.

Within computer science, the study of programming languages is generally divided into syntax and semantics, where syntax is the study of programming notation and semantics the study of the meaning of programs. This notion of meaning, however, includes only the mathematical formalization of a program, which is also what can be executed on a machine.[90] The usual sense of semantics – which is the meaning a program has to a human – is lost.

Paulson writes that an engineer understands a bicycle in terms of its components, and therefore a programmer should also understand a program in terms of components.[91] This thinking is a kind of reification: mistaking concepts for concrete things. Machine-oriented thinking easily leads to this kind of philosophical mistake, because focusing solely on the machine means that the inherent ambiguity of human thinking is ignored. The comparison between bicycle and

[89]Huth & Ryan 2000 [2004] p. 1.
[90]Filinski et al. 2005 chp. 1, p. 2.
[91]Paulson 1991 [1996] p. 59.

program is fine as a metaphor – if it is extended with additional explanation – but it cannot stand alone as an argument in itself.

3.2.3.5 *Premises of argumentation*

There is within computer science an unfortunate tendency to ignore the premises of the mathematical systems upon which the research focuses. Naur calls it a pattern in ignoring subjectivity in computer science.[92] By "subjectivity", Naur largely means the consequences for the people who are to use the programs, that is, the programs' usefulness. According to Naur, computer science publications commonly postulate a need for a new notation for some purpose. The notation is then presented in thorough technical details; next, it is claimed in the conclusion that the need is now met, without any evidence.

As an example, we will look at the doctoral dissertation of Ulla Solin *Animation of Parallel Algorithms* from 1992. This is not badly-executed research – on the contrary, it is exemplary in fulfilling the expectations of a dissertation in computer science and precisely because of that it serves well as an example.

In the introduction to the dissertation, Solin briefly explains what the subject is: making animations out of programs.[93] She then proceeds in seven chapters to demonstrate a detailed method of constructing animated programs, complete with mathematical proof. The dissertation's conclusion begins with the sentence: "It is obvious that animation has a wide range of potential use and should become an important tool for testing and analysing algorithms."[94]

The problem is that it is not at all possible to conclude this from the dissertation, which concerns only the details of a solution to the problem that has been posed. The statement that animation should be an important tool is actually the direct opposite of a conclusion: it is the premise for the whole dissertation. Without assuming animation of programs to be important, it makes little sense to devote

[92]Naur 1995 p. 10.

[93]That is, making a computer-generated animated film clip out of a computer program to illustrate how that program is behaving.

[94]Solin 1992 p. 101.

a dissertation to it. Nor is it obvious that animation is as useful as claimed – as it is, history so far has shown that animating programs is not actually an important tool for programming.[95]

An example on a larger scale comes from Jones, who conducted research in automatic compiler generation since 1980.[96] Twenty-five years later he writes that: "It is characteristic of our grand dream [of automatic compiler generation] that we are just now beginning to understand which problems really need to be solved."[97] However, Jones does not call for an examination of the premises of the dream, but rather advocates more mathematics: "What is needed is rather a better fundamental understanding of the theory and practice of binding time transformations … ".[98]

The focus on mathematics and logic to the exclusion of the premises of research sometimes has the consequence that overly specialized subfields in computer science develop an unrealistic sense of their own importance. For example, Glynn Winskel, a researcher in the field of programming language semantics – which is the mathematical study of programming languages – writes that semantics is useful for "various kinds of analysis and verification" and can also reveal "subtleties of which it is important to be aware."[99] Moreover, semantics is arguably not of great practical use, since "it is fairly involved to show even trivial programs are correct."[100] Overall, this does not give the impression that semantics is a field of extreme importance for the whole of computer science. Yet the fact that program semantics is regarded as belonging to the foundations of computer

[95]It is not for want of trying. So-called visual programming is a long-standing idea held by computer scientists. See for example the August 1985 issue of IEEE Computer magazine devoted to "Visual Programming", including an article on "Animating Programs Using Smalltalk".

[96]Jones & Schmidt 1980: "Compiler Generation from Denotational Semantics."

[97]Filinski et al. 2005 chp. 1, p. 22.

[98]Ibid.

[99]Winskel 1993 p. xv.

[100]Ibid. p. 95.

science[101] indicates the emphasis placed upon the mathematical perspective.

3.2.3.6 *Work processes*

We will now look at what computer science has to say about programming as an activity – that is, about the work processes of programming. According to Dijkstra, programming is essentially a matter of effective thinking.[102] Dijkstra sees the programming field as being in an ongoing transition from a craft to a science. The goal of using computers, and of thinking, is to reduce the amount of reasoning that is necessary in order to arrive at effective, intellectually elegant solutions to problems. What Dijkstra means by "reasoning" is essentially calculation, since it is something a computer can do.[103] It is not difficult to discern the ideals of a mathematician in Dijkstra's approach to programming.

For Simonyi, the central things in programming is imagining and writing code to maintain invariances in data structures. "Maintaining invariances" is a concept that comes from mathematical analysis of data structures. Simonyi writes that: "Writing the code to maintain invariances is a relatively simple progression of craftmanship, but it requires a lot of care and discipline."[104]

Stroustrup has a simple phase model of development that is reminiscent of software engineering's waterfall model. According to Stroustrup, the development process consists first of analysis, then design, and finally programming.[105] There is a marked difference from software engineering, however, in that Stroustrup emphasises that many of the details of the problem only become understood through programming. This aspect of the model seems more in line with Agile thinking, which discourages programmers from planning too far

[101]Winskel's book is published in the Massachusetts Institute of Technology series "Foundations of Computing".
[102]Dijkstra 1975.
[103]Dijkstra 1975 p. 6 f.
[104]Lammers 1986 p. 15.
[105]Stroustrup 1985 [1997] p. 15.

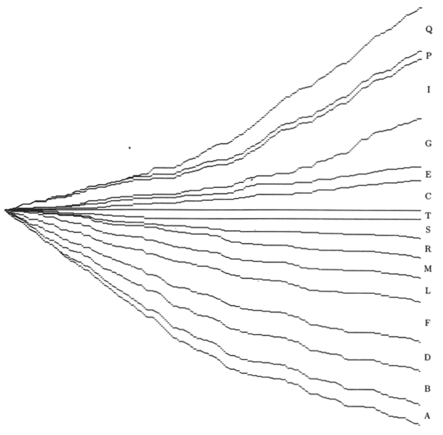


Figure 3.11: "Accumulated errors of TEX78 divided into fifteen categories." From Knuth 1989.

ahead according to the view that only working code can show if the program solves the right problem.

Since, as we have seen, computer science is preoccupied with mathematics and a machine-oriented perspective, it has little to say regarding the programming work process compared to software engineering and Agile thinking. In general, according to computer science the way a programmer works is much the same as the way a stereotypical mathematician works. One notion is that he works on the basis of genius inspiration, which is basically inexplicable; another notion is that he follows a very personal and *ad hoc* process that cannot meaningfully be systematized.

The famous computer scientist Donald E. Knuth can serve as an illustrative example of the "genius" perception of programming processes in computer science. Knuth is the author of the colossal, unfinished, and highly acclaimed *The Art of Computer Programming.* His article "The Errors of TEX" tells the story of one man (the genius of the story) who worked alone for 10 years on a typesetting program that later went on to become a huge success.[106] He then publishes a list of all the errors that he discovered and corrected in the program during the 10 years, thinking that somehow a list of more than 850 former errors in a program that he alone has worked on could be of interest to other people.

As it is, the list has no practical use, but it does serve as a monument to Knuth's lonely programming effort. Most interesting is Knuth's classification of the errors: some are classified according to technical criteria such as "algorithm" (A), "data" (D), and "robustness" (R), but others are classified in purely personal categories: "blunder" (B), "forgotten" (F), and "surprise" (S). (See Figure 3.11.) It is of course not surprising that there were errors in the program – every programming effort is a process of trial and error. Nor is it surprising that the programmer's personality and emotions play a part in the programming process. It is more remarkable that Knuth seems to regard his strictly personal experience of the process as something that can meaningfully be communicated to others in schematic form.

Naur writes that programming is primarily a question of achieving a certain kind of insight.[107] He calls this building a theory of the matters at hand. Programming is therefore a process of gaining insight, and it is accordingly not possible to find any "right method" for programming in general since every situation demands a unique insight. What Naur calls "theory" is in my opinion close to what is, in hermeneutical theory, termed "understanding"; and as we shall see in chapter 5.2 (Analysis), Naur's view of the programming process appears to be quite close to a hermeneutical understanding of programming.

[106]Knuth 1989.
[107]Naur 1985 [2001] p. 186.

3.2.4 *Object-Oriented Programming*

Object-Oriented Programming is a collection of programming techniques and a design philosophy that was primarily developed during the 1980s and became successful, perhaps even dominant, during the 1990s. The spread of Object-Oriented Programming was largely due to the popularity of the programming languages C++ and later Java, which in many branches of the programming industries became *de facto* standards.

Each Object-Oriented programming language has its own distinctive features, so it is difficult to give a precise definition of what constitutes Object-Orientation. However, if one looks at the basic features of Object-Orientation there are a few traits that seem to be common. Object-Oriented programming techniques are primarily characterized by the use of so-called "classes" and "inheritance". In addition to these, Object-Oriented programming languages are founded upon a distinctive philosophy of Object-Oriented Design.

An Object-Oriented class is essentially the same as what is more generally, in computer science, called an abstract data type. It means that the programmer is not limited to working with the data types that can be represented directly by the machine, such as letters and numbers, but can also construct more complex data structures such as lists and trees. Additionally, classes use "information hiding", or "encapsulation", which means that the details of the class are shielded from any code outside the class, so that the programmer is not tempted to use bad low-level habits in his code.

What makes classes different from abstract data types is the other defining Object-Oriented technique: inheritance. Inheritance is essentially a way of diverting the control flow of a program. Classes can be divided into an inheritance hierarchy, in which some classes "inherit" the other classes. Whenever a programming operation is requested, the inheritance hierarchy then decides which of the classes' program code is executed.

Stroustrup has the following view of what constitutes the Object-Oriented programming process (with my explanations in parentheses):[108]

1. Decide on classes (define the data structures to be used).
2. Provide operations for classes (write the program code).
3. Make commonality explicit with inheritance (modify the control flow).

Object-Oriented Programming is, as mentioned above, founded upon an Object-Oriented Design philosophy; however in practice, the programming techniques and the design philosophy are distinct approaches that are not always encountered together. The first principle of Object-Oriented Design is to "use inheritance to classify objects into taxonomies"[109] (The "objects" spoken of here have nothing to do with the usual sense of the word "object", but refers to instances of Object-Oriented classes.) Stroustrup writes: "One of the most powerful intellectual tools for managing complexity is hierarchical ordering."[110]

The second principle of Object-Oriented Design is to use inheritance to exploit commonality in the program code – this is also called code reuse. However, there is a difference between conceptual commonality, which somehow reflects some aspect of reality, and artificial similarities.[111] Obviously, a true Object-Oriented design build on the former while avoiding the latter.

Object-Oriented Design has been criticized for being a far-fetched design philosophy, while Object-Oriented Programming generally is taken for given. Lauesen writes:

> "The elusiveness of transferring functions from domain to product makes it difficult to use the object approach for requirements. . . .

[108]Stroustrup 1985 [1997] p. 39.
[109]Bruegge & Dutoit 2000 [2010] p. 347.
[110]Stroustrup 1985 [1997] p. 15.
[111]Ibid. p. 732.

> There are many other problems with modeling the domain by objects, one being that users have difficulties seeing that a hotel stay or an invoice can perform operations, such as object modeling suggests. Object modeling also suggests that users are objects with operations, which users find strange.
>
> Another problem is that class models are not suitable for describing how objects co-operate to support the user tasks. . . .
>
> However, there are systems where the object approach works well for analysis and requirements . . . In these cases we either model a computer artefact or we model something that really *is* a collection of physical objects. For good reasons, object people prefer these kinds of systems as examples.
>
> These observations apply to object-oriented analysis and requirements specification. Object-oriented *programming* is another issue. . . . because many programming tools and packages are object-oriented, OO-programming is often a must."[112]

Stroustrup himself admits that "In many cases there is no real advantage to be gained from inheritance."[113] A textbook on Object-Oriented software engineering points out that " . . . inheritance is such a powerful mechanism that novice developers often produce code that is more obfuscated and more brittle than if they had not used inheritance in the first place."[114] Here, a critic of Object-Oriented Design might point out that it is perhaps not on account of inheritance being too powerful that it produces brittle and obfuscated code, but due to this not being a very good design principle in the first place.

Another problem that Object-Oriented Programming and Design shares with the rest of computer science[115] is that its fundamental

[112]Lauesen 2002 p. 184 f.
[113]Stroustrup 1985 [1997] p. 727.
[114]Bruegge & Dutoit 2000 [2010] p. 348.
[115]See section 3.2.3.4 (Machine-orientation)

principles carry with them a danger of reification – that is, a danger of mistaking programming concepts for real things. This is because the Object-Oriented approach is very strongly based upon the notion that a program is a model of reality, and not just a useful tool for some purpose. According to Stroustrup: "The most fundamental notion of object-oriented design and programming is that the program is a model of some aspect of reality."[116]

The following example is taken from the influential book *Design Patterns: Elements of Reusable Object-Oriented Software.* The authors write about how the program code for a document editing program should be structured: "Logically there is [a programming] object for every occurrence of a given character in the document: Physically, however, there is one shared ... object per character".[117] This is, of course, absurd; physically, there is only the computing machine. Any "object" in the program code will be physically present only in the form of electrical currents in the machine. This goes to show that when the whole design philosophy is based on the object concept, it is easy to forget that the objects are metaphorical and not real.[118]

3.3 *Hermeneutical theory*

3.3.1 *Introduction*

Hermeneutics is a theory both of how understanding takes place and of what understanding is. Hermenutical theory is used in this treatise in order to interpret what software developers do and to understand better why work processes take the form that they do.

[116]Stroustrup 1985 [1997] p. 732.

[117]Gamma et al. 1994 p. 196.

[118]Latour observes that reification happens all the time in science and has explained it as a consequence of the rush to develop unassailable theories and establish facts. Latour 1987 p. 91 f.

Hermeneutical concept	**Characteristics**
Prejudice	Prejudice is a prerequisite for understanding – it can have either a positive or negative influence.
Authority	These are the sources of reason that are recognized as valid.
Received tradition	All understanding builds upon some kind of tradition, and carries this tradition onwards itself while also contributing to it.
Personal reverence	This is the basis for authority. Since understanding is done by persons, personal reverence is crucial.
Pre-understanding	This is the kind of factual understanding that is necessary for understanding, but leaves relatively little room for interpretation – for example to know the language of a text one wishes to read.
Understanding	In hermeneutical theory, understanding is productive and existential. Briefly put, this means that understanding has consequences.
Hermeneutical circle	A concept that expresses the relationship between, for example, part and whole, or between action and reflection.
Effective history	A concept that expresses the historical nature of knowledge and understanding.
Question	All interpretation and understanding is driven by some kind of question.
Horizon of understanding	One's horizon of understanding is the amount of things that are understandable given the present state of one's knowledge and prejudice. Phenomena beyond the horizon of understanding appear meaningless.
Application	Since understanding has consequences, it is crucial to what end the understanding takes place. The intended application of an effort of understanding has an impact on the understanding itself.

Figure 3.12: Some hermeneutical concepts used in the analyses of programming work process.

In Figure 3.12 are listed some key hermeneutical concepts that will be applied in the analyses in this treatise, along with some very short explanations of the concepts. The use of each concept and its more precise meaning is explained in the following sections.

3.3.2 *What is hermeneutics*

Briefly put, hermeneutics is the scholarly discipline that studies interpretation. Originally, hermeneutics referred specifically to interpretation of the Bible because for most of European intellectual history, theology was considered the highest form of study. Over time, hermeneutics came to be extended to other fields of study.

Thus, the German philosopher Hans-Georg Gadamer has developed a coherent view that has subsequently come to be known as "philosophical hermeneutics". Gadamer published his *magnum opus* called *Wahrheit und Methode* (*Truth and Method*) in 1960 at the age of 60. In this work Gadamer summarizes and engages in a discussion with more than 2,000 years of European thinking, from Aristotle through the developments of the Middle Ages up to and including the science of his own time.

According to Gadamer, classical hermeneutics is divided into three branches: theological, judicial, and philological hermeneutics, which are concerned with interpreting the Bible, llegal texts, and works of great literature respectively . During the 18th and 19th centuries the branches of classical hermeneutics were supplemented by historical hermeneutics. The growing professionalization and objectivism of historians meant that they were faced with pertinent questions about how to ensure a true interpretation of history; they turned to hermeneutics for assistance with their task.

Since Gadamer's work is a reaction to the historicism of the 19th century, he is first and foremost concerned with how interpretation can overcome the barrier of historical distance. However, for our purposes we will assume that this distance in time might as well be a distance in culture. After all, Gadamer's central problem concerns how it is possible to understand an author whose historical distance makes him strange to us. The strangeness, however, is still present when we try to bridge a contemporary, but cultural, distance. Philosophical hermeneutics applies equally well to historical and cultural differences because culture is in essence historical.

It is important to understand that Gadamer's philosophy is general, in the sense that it applies to all true understanding, not only scientific understanding or understanding in particular situations. This is the case despite the fact that the situations that Gadamer mostly uses as examples are those of the scholar, the historian, and the judge. Gadamer uses hermeneutical understanding as a model of all true understanding. By saying what true understanding is, he is implicitly also defining false understanding: namely, an understanding that disregards the fundamental restrictions of hermeneutics, thereby fooling itself.

3.3.3 *Prejudice and reason*

Prejudice[119] is a central concept in philosophical hermeneutics. Prejudice is a prerequisite for understanding.[120] As such, reason and prejudice are intimately connected, because reasoning can only take place on the foundation of prejudice. Reason cannot justify itself; there will always be some axioms, basic assumptions, or *a priori* knowledge that lay the ground rules for reason, and these assumptions are part of prejudice.

The Enlightenment philosophers and the romanticism movement have tried to replace prejudice by reason. However, the very notion that it is possible to do so is itself a form of prejudice.[121] An archetypical proponent of this kind of thinking is Descartes, who refused to

[119] "Vorurteil".
[120] Gadamer 1960 [1965] II.II.1.a.α, p. 254 ff.
[121] Ibid. II.II.1.a.β, p. 260.

accept anything as a basis for truth other than that which he could reason about the world. The only basis for his reasoning was his own ability to reason, as expressed in his famous aphorism "I think, therefore I am". Of course, as others have pointed out, Descartes did not adhere to his own strict methodology.[122]

To explain why prejudice is inevitable, consider that as you are reading this text you believe that I have something to tell you, that I am not lying or trying to deceive you, and that I am not saying the opposite of what I mean – in other words, that my words have meaning, and that this meaning can be understood. Your belief is a part of your prejudice, and indeed it is necessary in order to understand the text.

A radical rejection of prejudice is at the same time a rejection of the meaningfulness of the outside world. The archetype of a person rejecting all prejudice is a person suffering from paranoid schizophrenia: because he refuses to see meaning in the outside world, it becomes unintelligible to him, and in turn his own internal world becomes contingent, random, and meaningless.

Prejudice is thus a prerequisite for understanding, but it does not follow from this that all prejudice is equally good, or that prejudice is good just because it is prejudice. There exists false and true prejudice; prejudice that helps understanding and prejudice that hinders understanding. The task of hermeneutics is to determine true prejudice from false.[123]

[122] "[D]espite his austere recommendations about the methods of discovery and demonstration, he hardly ever followed those methods, hardly ever wrote in the same genre twice". Amelie Oksenberg Rorty 1983 as quoted in McCloskey 1985 [1998] p. 59.

[123] "Es ist diese Erfahrung, die in der historischen Forschung zu der Vorstellung geführt hat, daß erst einem gewissen geschichtlichen Abstande heraus objektive Erkenntnis erreichbar werde. ... Nichts anderes als dieser Zeitenabstand vermag die eigentlich kritische Frage der Hermeneutik lösbar zu machen, nämlich die *wahren* Vorurteile, unter denen wir *verstehen*, von den *falschen*, unter denen wir *mißverstehen*, zu scheiden." Gadamer 1960 [1965] II.II.1.c, p. 282.

3.3.4 *Authority*

The claim made by Enlightenment philosophy that reason is limitless has its opposite in the claim made by the Christian intellectual heritage: that man is fundamentally limited by existence and by history.[124] By existence because he cannot do whatever he wants; by history because his knowledge is always given in a certain historical context. From this it follows that tradition is always a part of understanding and history.[125] There is always an already existing understanding and an existing way of doing things that must be evaluated in the course of understanding.

When evaluating tradition, the crucial concept is authority. Tradition has an authority of its own, such that we will often not accept changes to tradition without good reason. The authority of tradition ultimately rests on other sources of authority, and rejection of tradition requires authority as well. Appeals to authority are not undesirable: on the contrary, they are an integral part of understanding.

Of course, not all appeals to authority are equally good. The good appeal to authority is the appeal to reasonable authority: that is, authority that is exercised in accordance with its purpose and limitations. False authority is authority that is misused: the authority of the tyrant or the con-man. Reasonable authority is the basis for personal reverence, and personal reverence is again the basis for prejudice. Thus we see why authority is important to understanding – we form our prejudices on the basis of those whom we believe in, and those who we believe in are determined by those to whom we ascribe authority.

The personal aspect of reverence is important; the understanding mind must have a personal relationship to authority in order to form prejudice. This precludes that Descartes' limitless individual reason could be substituted with a limitless collective reason of humanity, such as Rousseau's "volonté générale". For example, a scientist may

[124] Ibid. II.II.1.a.β, p. 260.

[125] "In Wahrheit ist Tradition stets ein Moment der Freiheit und der Geschichte selber." Ibid. II.II.1.b.α, p. 265.

defer to scientific authority in the abstract, but this would not have told him, at the time, whether to believe Bohr or Einstein, Lavoisier or Priestley, or Edison or Westinghouse.

3.3.5 *Understanding*

Understanding[126] is composed of two parts: pre-understanding[127] and understanding proper.[128] Pre-understanding is the knowledge of the facts of the case that is necessary before the meaning of the case can begin to be understood. For example, knowledge of Classical Greek is part of the pre-understanding for understanding the meaning of Aristotle's original works. Note that pre-understanding, the knowledge of bare facts, is never in itself enough to form understanding.

The concept of understanding is not a way to perceive subjectivity – that is, it is not "subjective" understanding of some "objective" reality. Understanding is an exchange between oneself and another.[129] Both sides have subjective and objective perspectives. They share one physical, existing reality, but they do not necessarily share their perception of reality.

Understanding takes place in a circular movement between part and whole.[130] To take one example: the part can be a sentence and the whole can be a book. To understand a sentence (the part), one has to have an understanding of the book (the whole). But, on the other

[126] "Verstehen".

[127] "Vorverständnis".

[128] "Auch hier bewährt sich, daß Verstehen primär heißt: sich in der Sache verstehen, und erst sekundär: die Meinung des anderen als solche abheben und verstehen." Gadamer 1960 [1965] II.II.1.c, p. 278.

[129] "*Das Verstehen ist selber nicht so sehr als eine Handlung der Subjektivität zu denken, sondern als Einrücken in ein Überlieferungsgeschehen,* in dem sich Vergangenheit und Gegenwart beständig vermitteln." Ibid. II.II.1.b.*β*, p. 274 f.

[130] "Die Antizipation von Sinn, in der das Ganze gemeint ist, kommt dadurch zu explizitem Verständnis, daß die Teile, die sich vom Ganzen her bestimmen, ihrerseits auch dieses Ganze bestimmen." Ibid. II.II.1.c, p. 275.

hand, an understanding of the whole is only accessible through the understanding of its constituent parts. This means that the effort of understanding is always shifting, now focusing on understanding the part in context, now focusing on revising the context in the new light of the part. This insight is so important that is has come to be known as "the hermeneutical circle".

The hermeneutical circle is not a method; method alone cannot bring about understanding.[131] Reading a pile of books is a method for understanding, but whether or not the reading actually results in understanding depends on the specific situation in which the reading takes place. There is no way to proscribe a method that will with certainty result in understanding.

Understanding is not merely reproductive – that is, carrying intact knowledge around from mind to mind. Rather, understanding is *productive*, in that when I understand another person's utterance, my understanding is never exactly the same as his, since our respective contexts are different. Therefore understanding will bring about something new in the present situation, having a productive effect.[132]

Understanding is existential,[133] meaning that understanding must result in some kind of consequence for the one who understands. For if there were no consequences, any other interpretation might do as well as the one arrived at; it would not matter to the interpreter – consequently the interpretation would not matter, meaning that it is not true understanding.[134]

[131] "Der Zirkel des Verstehens ist also überhaupt nicht ein 'methodischer' Zirkel, sondern beschreibt ein ontologisches Strukturmoment des Verstehens." Ibid. II.II.1.c, p. 277.

[132] "Daher ist Verstehen kein nur reproduktives, sondern stets auch ein produktives Verhalten. ... Verstehen ist in Wahrheit kein Besserverstehen, weder im Sinne der sachlichen Besserwissens durch deutlichere Begriffe, noch im Sinne der grundsätzlichen Überlegenheit, die das Bewußte über das Unbewußte der Produktion besitz. Es genügt zu sagen, daß man *anders* versteht, *wenn man überhaupt versteht.*" Ibid. II.II.1.c, p. 280.

[133] "Denn erst von der ontologischen Wendung, die Heidegger dem Verstehen als einem 'Existenzial' verlieh, und der temporalen Interpretation, die er der Seinsweise des Daseins widmete, aus konnte der Zeitenabstand in seiner hermeneutischen Produktivität gedacht werden." Ibid. II.II.1.c, p. 281.

[134] "*Verstehen erwies sich selber als ein Geschehen,* ..." Ibid. II.II.2.a, p. 293. This point expresses in a different way the insight encapsulated by the American philosopher C.S.

Understanding is a connection between tradition[135], understood in a broad sense, and the interpreter.[136] Thus, hermeneutics acts to connect the strange and the familiar; or, seen from another perspective, acts to connect objectivity and tradition.[137]

3.3.6 *Effective history*

Effective history[138] is Gadamer's concept for explaining the relationship between understanding and history. It is so named to emphasise that understanding has effects, in history as well as in the present day. To be conscious of effective history is to be simultaneously aware of historical reality and of the historical nature of understanding.[139]

Peirce's "pragmatic maxim": "Hence is justified the maxim, belief in which constitutes pragmatism; namely, *In order to ascertain the meaning of an intellectual conception one should consider what practical consequences might conceivably result by necessity from the truth of that conception; and the sum of these consequences will constitute the entire meaning of the conception.*" Peirce 1905 [1931-1958] vol. 5, ¶ 9.

[135] "Überlieferung."

[136] "Der Zirkel ist also nicht formaler Natur, er ist weder subjektiv noch objektiv, sondern beschreibt das Verstehen als der Ineinanderspiel der Bewegung der Überlieferung und der Bewegung des Interpreten. Die Antizipation von Sinn, die unser Verständnis eines Textes leitet, ist nicht eine Handlung der Subjektivität, sondern bestimmt sich aus der Gemeinsamkeit, die uns mit der Überlieferung verbindet. Diese Gemeinsamkeit aber ist in unserem Verhältnis zur Überlieferung in beständiger Bildung begriffen. Sie ist nicht einfach eine Voraussetzung, unter der wir schon immer stehen, sondern wir erstellen sie selbst, sofern wir verstehen, am Überlieferungsgeschehen teilhaben und es dadurch selber weiter bestimmen." Gadamer 1960 [1965] II.II.1.c, p. 277.

[137] "Die Stellung zwischen Fremdheit und Vertrautheit, die die Überlieferung für uns hat, ist das Zwischen zwischen der historisch gemeinten, abständigen Gegenständlichkeit und der Zugehörigkeit zu einer Tradition. *In diesem Zwischen ist der wahre Ort der Hermeneutik.*" Ibid. II.II.1.c, p. 279.

[138] "Wirkungsgeschichte."

[139] "Eine sachangemessene Hermeneutik hätte im Verstehen selbst die Wirklichkeit der Geschichte aufzuweisen. Ich nenne das damit Geforderte 'Wirkungsgeschichte'. Verstehen ist seinem Wesen nach ein wirkungsgeschichtlicher Vorgang." Gadamer 1960 [1965] II.II.1.c, p. 283.

For a hermeneutic theory to be effective, it has to be a theory that is conscious that understanding is itself a part of historical reality.[140] We demand of science that it is methodical; thus, we demand of scientific hermeneutics not only that it is conscious, but also that it is methodical in its consciousness. Effective history is a methodical way of expressing this consciousness, because consistently employing the concept of effective history in the hermeneutical process ensures that the historical consciousness is not merely *ad hoc*, but can be carried out in a systematic manner.[141]

What effective history essentially expresses is this: that the immediate separation in time or in culture between the subject and the interpreter is not the whole of the truth. In addition to the observable differences there is a difference in perspective; namely that subject and interpreter have different aims in application.[142] A subject who writes a text does not have the same ends in mind as the interpreter who later tries to understand the text.

Effective history is a form of self-insight.[143] It is an awareness that when I try to understand something, I am placed in a hermeneutical

[140] "Ein wirklich historisches Denken muß die eigene Geschichtlichkeit mitdenken." Ibid. II.II.1.c, p. 283.

[141] "Daß das historische Interesse sich nicht allein auf die geschichtliche Erscheinung oder das überlieferte Werk richtet, sondern in einer sekundären Thematik auch auf deren Wirken in der Geschichte (die schließlich auch die Geschichte der Forschung einschließt), gilt im allgemeinen als eine bloße Ergänzung der historischen Fragestellung, ... Insofern ist Wirkungsgeschichte nichts Neues. Daß es aber einer solchen wirkungsgeschichtlichen Fragestellung immer bedarf, wenn ein Werk oder einer Überlieferung aus dem Zwielicht zwischen Tradition und Historie ins Klare und Offene seiner eigentlichen Bedeutung gestellt werden soll, das ist in der Tat eine neue Forderung – nicht an die Forschung, aber an das metodische Bewußtsein derselben – die sich aus der Durchreflexion des historischen Bewußtseins zwingend ergibt." Ibid. II.II.1.d, p. 284.

[142] "Wenn wir aus der für unsere hermeneutische Situation im ganzen bestimmenden historischen Distanz eine historische Erscheinung zu verstehen suchen, unterliegen wir immer bereits den Wirkungen der Wirkungsgeschichte. Sie bestimmt im voraus, was sich uns als fragwürdig und als Gegenstand der Erforschung zeigt, und wir vergessen gleichsam die Hälfte dessen, was wirklich ist, ja mehr noch: wir vergessen die ganze Wahrheit dieser Erscheinung, wenn wir die unmittelbare Erscheinung selber als die ganze Wahrheit nehmen." Ibid. II.II.1.d, p. 284.

[143] "Wirkungsgeschichtliches Bewußtsein ist zunächst Bewußtsein der hermeneutische *Situation*. ... Der Begriff der Situation ist ja dadurch charakterisiert, daß man sich nicht

situation that I cannot rise above. In the same way as I cannot choose to let my body disobey the laws of physics, I also cannot choose to let my understanding disregard the limits to knowledge that is given by the concrete point in history in which I find myself.

It is a consequence of the nature of historical existence that a reflection on effective history can never be complete; that is, we can never arrive at the unequivocal true effective history of a phenomenon.[144] This is simply another way of saying that there is no such thing as complete knowledge. Our understanding is always limited by different perspectives: of some of the perspectives we are aware, of others we are not. The concept of effective history is a way to guide our awareness to discover those perspectives that best serve our purpose – our purpose being, ultimately, truth.

3.3.7 *Horizons of understanding*

As stated earlier, the task of hermeneutics is to tell true prejudice from false. In order to do this, the prejudices have to be evaluated. For this to happen, they must be engaged, and the engagement and evaluation of prejudices happens when a question is posed that demands that understanding is extended.[145]

Prejudice limits our point of view, in the sense that it limits the understanding that we are able to form based on the available facts. But at the same time prejudice serves as a solid ground that is the foundation for meaning, for as we have seen prejudice is a prereq-

ihr gegenüber befindet und daher kein gegenständliches Wissen von ihr haben kann." Ibid. II.II.1.d, p. 285.

[144] "Auch die Erhellung dieser Situation, d.h. die wirkungsgeschichtliche Reflexion ist nicht vollendbar, aber diese Unvollendbarkeit ist nicht ein Mangel an Reflexion, sondern liegt im Wesen des geschichtlichen Seins, das wir sind. *Geschichtlichsein heißt, nie im Sichwissen aufgehen.*" Ibid. II.II.1.d, p. 285.

[145] "Alle Suspension von Urteilen aber, mithin und erst recht die von Vorurteilen, hat, logisch gesehen, die Struktur der *Frage.*" Ibid. II.II.1.c, p. 283.

uisite for meaning.[146] This unavoidable limitation of understanding that is effected by prejudice is called, by Gadamer, a horizon of understanding.[147]

The double role of prejudice as both limit and foundation means that prejudice must constantly be tested. The determination of what is true and false prejudice is not an act that is done with once and for all; rather, it is an ongoing process. As prejudice is in turn founded on received tradition, the ongoing testing of prejudice also means that our understanding of received tradition must constantly be tested.[148]

Our point of view is limited by our horizon of understanding, but what happens when our understanding is expanded, when our prejudices are tested and revised? We speak then of a fusion of horizons. Our horizon does not become completely replaced by the horizon of the other, whom we are trying to understand; nor does the meaning of the other suddenly become understandable fully within the prejudice of our original horizon. Rather, our horizon becomes nearer to that of the other, and at the same time the other's horizon comes to be interpreted in a new light that connects it to our original horizon. A partial fusion of the horizons takes place.

In order to understand the other we have to be able to set ourselves in the other's place, at least for a short while. Real understanding cannot happen if we deny the validity of the other's premises

[146]"Wir waren davon ausgegangen, daß eine hermeneutische Situation durch die Vorurteile bestimmt wird, die wir mitbringen. Insofern bilden sie den Horizont einer Gegenwart, denn sie stellen das dar, über das hinaus man nicht zu sehen vermag. Nun gilt es aber, den Irrtum fernzuhalten, als wäre es ein fester Bestand von Meinungen und Wertungen, die den Horizont der Gegenwart bestimmen und begrenzen, und als höbe sich die Andersheit der Vergangenheit dagegen wie gegen einen festen Grund ab." Ibid. II.II.1.d, p. 289.

[147]"Alle endliche Gegenwart hat ihre Schranken. Wir bestimmen den Begriff der Situation eben dadurch, daß sie einen Standort darstellt, der die Möglichkeit des Sehens beschränkt. Zum Begriff der Situation gehört daher wesenhaft der Begriff des *Horizontes*. Horizont ist der Gesichtskreis, der all das umfaßt und umschließt, was von einem Punkt aus sichtbar ist. In der Anwendung auf das denkende Bewußtsein reden wir dann von Enge des Horizontes, von möglicher Erwiterung des Horizontes, von Erschließung neuer Horizonte usw." Ibid. II.II.1.d, p. 286.

[148]"In Wahrheit ist der Horizont der Gegenwart in steter Bildung begriffen, sofern wir alle unsere Vorurteile ständig erproben müssen." Ibid. II.II.1.d, p. 289.

without understanding the consequences of those premises. Thus we necessarily have to suspend our own perspective temporarily. Otherwise, the perspective of the other becomes unintelligible to us. On the other hand, there lies also a danger in being too eager to adapt to the foreign perspective. This leads to an unrealistic perception of our own reality, and to a romanticization of the other's – for example the romantic fictions of "the noble savage" and of historical "Golden Ages".[149] A horizon of understanding is a frame of reference of higher generality that suspends the particularity of both ourselves and the other before understanding can take place.[150]

We see then that our horizon of understanding is a fundamental limitation of what we can understand. But horizons are not static. All true speech serves to bring horizons together, resulting in a fusion of horizons. Moreover, horizons are not isolated. No culture's horizon of understanding is completely separated from others'.[151] There is always some overlap of horizon between cultures, at the very least the basic experience of being human.

Horizons are always shifting, because life itself is not static. In this regard, we may perceive human existence itself as fusions of horizons,[152] at least so long as we are talking about understanding as

[149] "Das historische Bewußtsein tut offenbar Ähnliches, wenn es sich in die situation der Vergangenheit versetzt und dadurch den richtigen historischen Horizont zu haben beansprucht. ... Der Text, der historisch verstanden wird, wird aus dem Anspruch, Wahres zu sagen, förmlich herausgedrängt. Indem man die Überlieferung vom historischen Standpunkt aus sieht, d.h. sich in die historische Situation versetz und den historischen Horizont zu rekonstruieren sucht, meint man zu verstehen. In Wahrheit hat man den Anspruch grundsätzlich aufgegeben, in der Überlieferung für einen selber gültige und verständliche Wahrheit zu finden. Solche Anerkennung der Andersheit des Anderen, die dieselbe zum Gegenstande objektiver Erkenntnis macht, ist insofern eine grundsätzliche Suspension seines Anspruchs." Ibid. II.II.1.d, p. 287.

[150] "Solches Sichversetzen ist weder Einfühlung einer Individualität in eine andere, noch auch Unterwerfung des anderen unter die eigenen Maßstäbe, sondern bedeutet immer die Erhebung zu einer höheren Allgemeinheit, die nicht nur die eigene Partikularität, sondern auch die des anderen überwundet." Ibid. II.II.1.d, p. 288.

[151] "Wie der Einzelne nie ein Einzelner ist, weil er sich immer schon mit anderen versteht, so ist auch der geschlossene Horizont, der eine Kultur einschließen soll, eine Abstraktion." Ibid. II.II.1.d, p. 288.

[152] "Es macht die geschichtliche Bewegtheit des menschlichen Daseins aus, daß es keine schlechthinnige Standortgebundenheit besitzt und daher auch niemals einen wahr-

being fundamental to human existence; and human existence is certainly not possible without understanding.

Historical consciousness and horizons of understanding are not exactly the same things, but they are related. At the most fundamental level, they are tied together by human existence – an existence that is determined by heritage and received tradition.[153]

Thus, all understanding is a fusion of horizons.[154] One's own horizon of understanding and that of tradition are two horizons that are as a minimum always involved in the fusion. It is imperative for a scientifically conscious hermeneutics to be aware of the distinction between one's own horizon and that of tradition. We see now why the concept of effective history is so important to philosophical hermeneutics, for effective history expresses nothing other than a conscious awareness of the relationship between tradition and understanding. Effective history is a way of letting the horizons fuse in a controlled manner.[155]

haft geschlossenen Horizont. Der Horizont ist vielmehr etwas, in das wir hineinwandern und das mit uns mitwandert. Dem Beweglichen verschieben sich die Horizonte. So ist aus der Vergangenheitshorizont, aus dem alles menschliche Leben lebt und der in der Weise der Überlieferung da ist, immer schon in Bewegung." Ibid. II.II.1.d, p. 288.

[153] "Die eigene und fremde Vergangenheit, der unser historisches Bewußtsein zugewendet ist, bildet mit an diesem beweglichen Horizont, aus dem menschliches Leben immer lebt und der es als Herkunft und Überlieferung bestimmt." Ibid. II.II.1.d, p. 288.

[154] "*Vielmehr ist Verstehen immer der Vorgang der Verschmelzung solcher vermeintlich für sich seiender Horizonte.*" Ibid. II.II.1.d, p. 289.

[155] "Im Vollzug des Verstehens geschieht eine wirkliche Horizontverschmelzung, die mit dem Entwurf des historischen Horizontes zugleich dessen Aufhebung vollbringt. Wir bezeichneten den kontrollierten Vollzug solcher Verschmelzung als die Aufgabe des wirkungsgeschichtlichen Bewußtseins." Ibid. II.II.1.d, p. 290.

3.3.8 *Application*

The central problem in hermeneutics is that of application.[156] Application is a part of all understanding, and cannot be separated from it.[157] There are two aspects of this problem. One is that the phenomenon we are trying to understand originally had some intention behind it: an application of knowledge. The other is our own reason for trying to understand: the application to which we put our understanding. Understanding is action in that it requires effort: that is, an application of will.

To give an example of the importance of application, consider that performance and interpretation cannot be completely separated in the arts of poetry, music and theater.[158] Our interpretation of a work is necessarily influenced by the circumstance that it was meant to be read aloud, played or shown. Even if we do not have an actual performance in mind, nevertheless the potential of the artwork to be performed will determine the meaning that we ascribe to it.

For example, if a soldier were to refuse to carry out an order, he would first have to understand it correctly. If he did not understand the order, then his failure to carry it out would not be refusal but

[156] "Während von dem ästhetisch-historischen Positivismus im Gefolge der romantischen Hermeneutik diese Aufgabe verdeckt worden war, liegt hier in Wahrheit das zentrale Problem der Hermeneutik überhaupt. Es ist das Problem der *Anwendung*, die in allem Verstehen gelegen ist." Ibid. II.II.1.d, p. 290.

[157] "Das heißt aber negativ, daß ein Wissen im allgemeinen, das sich nicht der konkrete Situation zu applizieren weiß, sinnlos bleibt, ja die konkrete Forderungen, die von der Situation ausgehen, zu verdunkeln droht." Ibid. II.II.2.b, p. 296.

[158] "Niemand wird ein Drama inszenieren, eine Dichtung vorlesen oder eine Komposition zur Aufführung bringen können, ohne den ursprünglichen Sinn des Textes zu verstehen und in seiner Reproduktion und Auslegung zu meinen. ... Wenn wir vollends daran denken, wie die Übersetzung fremdsprachlicher Texte oder gar ihre dichterische Nachbildung, aber auch das richtige Vorlesen von Texten mitunder die gleiche Erklärungsleistung von sich aus übernehmen wie die philologische Auslegung, so daß beides ineinander übergeht, dann läßt sich dem Schluß nicht ausweichen, daß die sich aufdrängende Unterscheidung kognitiver, normativer und reproduktiver Auslegung keine grundsätzliche Geltung hat, sondern ein einheitliches Phänomen umschreibt." Ibid. II.II.2.a, p. 294.

incompetence. On the other hand, if he never considered the meaning of the order but refused to comply out of hand, it would not be a refusal of that particular order but a case of blind rebellion against authority. The soldier who denies an order must first understand it; that is, evaluate its meaning and consequence – which is to understand its application.[159]

Understanding is an aspect of effect and shows itself as effect.[160] For, as we have already seen, understanding without consequence is not true understanding.

Older hermeneutics is divided into three parts: understanding, explaining[161] and application.[162] Together, these parts make up the interpretation. It is important to understand that application is not somehow subordinate to the others – it is fully as relevant to interpretation as are understanding and explaining.[163]

In judicial hermeneutics, the application takes the form of judgment. This is the essential use of judicial hermeneutics: to help the judge understand the law in order to pass judgment.[164] In theological hermeneutics, the application is preaching. Preaching is the act by which the priest conveys an explanation of his understanding of the Bible to the congregation.[165]

The branches of hermeneutics differ both in application and subject matter, but that does not mean that they are without relation to

[159] "... Es ist eine Schelmenmotiv, Befehle so auszuführen, daß man ihren Wortlaut, aber nicht ihren Sinn befolgt. Es ist also kein Zweifel, daß der Empfänger eines Befehls eine bestimmte produktive Leistung des Sinnverständnisses vollbringen muß." Ibid. II.II.2.c, p. 317.

[160] "Das Verstehen erweist sich als eine Weise von Wirkung und weiß sich als eine solche Wirkung." Gadamer 1960 [1965] II.II.2.c, p. 323.

[161] "Auslegung".

[162] Gadamer 1960 [1965] II.II.2.a, p. 290 f.

[163] "Auslegung ist nicht ein zum Verstehen nachträglich und gelegentlich hinzukommender Akt, sondern Verstehen ist immer Auslegung, und Auslegung ist daher die explizite Form des Verstehens. ... Wir werden also gleichsam einen Schritt über die romantische Hermeneutik hinaus genötigt, indem wir nicht nur Verstehen und Auslegen, sondern dazu auch Anwenden als in einem einheitlichen Vorgang begriffen denken." Ibid. II.II.2.a, p. 291.

[164] And in order to explain his judgment in the motivation for the decision.

[165] Gadamer 1960 [1965] II.II.2.a, p. 292.

each other. On the contrary: Gadamer takes judicial hermeneutics and its interrelation between understanding and application as an example of this interrelation in the other branches of hermeneutics. In that way, it can serve to restore the old unity between judicial, theological and philological hermeneutics.[166]

Modernistic science splits understanding into three functions: cognitive, reproductive and normative. This is problematic for several reasons. Firstly, this perception of understanding ignores the significance of explaining and application. Secondly, the introduced opposition between normative and cognitive is at odds with the experience from judicial and theological hermeneutics, which shows that normative and cognitive function cannot be conceptually separated.[167]

The romanticist perception of history is grounded in a psychological explanation that is based on a false opposition between subject and object.[168] In this perception, the main problem of interpretation becomes the unification of distinct subjective perceptions. This has the consequence that understanding becomes a matter of congeniality, and ultimately that interpretation comes to be dependent on some kind of mystical meeting of souls.

Modernistic science goes in the opposite direction, and commits the fallacy of disregarding anything but the objective perspective of history. This results in an inability to reconcile judgements and facts, and leaves science always hunting for a pure objective reality that turns out to be an illusion, for the simple fact that while subject and object can be distinguished, they cannot be separated.

In opposition to both romanticism and modernistic science, hermeneutic thinking maintains that understanding is based on meaning and intention found in the received tradition, and understanding is

[166]Ibid. II.II.2.c, p. 311. Also Ibid. II.II.2.a, p. 292: "Die enge Zusammengehörigkeit, die ursprünglich die *philologische* Hermeneutik mit der *juristichen* und *theologischen* verband, beruhte aber auf der Anerkennung der Applikation als eines integrierenden Momentes alles Verstehens."

[167]Ibid. II.II.2.a, p. 293.

[168]Ibid. II.II.2.a, p. 294. The concepts "subjectivity" and "objectivity" are discussed further in section 3.4 (Cultural form theory).

immediately accessible to us, neither dependant on congeniality nor on reducing all reality to objects.[169]

In hermeneutics, explaining is bound to the text in a way similar to that in which perspective is bound to a picture. The picture can be interpreted in many ways, but we are not free to choose the perspective, it is given beforehand.[170] Each text must be understood on its own premises, without the interpreter adding premises of his own. That is the demand of science.[171]

The hermeneutic insight is that understanding always requires application of the understanding mind. This is denied by modernistic science, but the denial lands science in trouble when it demands of the scientist a distanced mind, because the distance itself hinders understanding.[172] Hermeneutically, the scientist must be personally engaged in the text in order to understand. This engagement is necessarily subjective, but it can nonetheless be scientific.

3.3.8.1 *The application of history*

To return to the example of judicial hermeneutics, its application, judgment, is a model of the relationship between past and present. Judicial hermeneutics presupposes a community under law.[173] This

[169] "Unsere Überlegungen verwehren uns, die hermeneutische Problemstellung auf die Subjektivität des Interpreten und die Objektivität des zu verstehenden Sinnes aufzuteilen. Ein solches Verfahren ginge von einem falschen Gegenüber aus, ... Das Wunder des Verstehens besteht vielmehr darin, daß es keiner Kongenialität bedarf, um das wahrhaft Bedeutsame und das ursprünglich Sinnhafte in der Überlieferung zu erkennen. Wir vermögen uns vielmehr dem überlegenen Anspruch des Textes zu öffnen und der Bedeutung zu entsprechen, in der er zu uns spricht." Ibid. II.II.2.a, p. 294 f.

[170] "Die Zugehörigkeit des Auslegen zu seinem Text ist wie die Zugehörigkeit des Augenpunktes zu der in einem Bilde gegebenen Perspektive. Es handelt sich nicht darum, daß man diesen Augenpunkte wie einem Standort suchen und einnehmen sollte, sondern daß der, der versteht, nicht beliebig seinem Blickpunkt wählt, sondern seinen Platz vorgegeben findet." Ibid. II.II.2.c, p. 312.

[171] "Das aber besagt, daß die historische Wissenschaft jeden Text zunächst in sich zu verstehen sucht und die inhaltliche Meinung desselben nicht selber vollzieht, sondern in ihrer Wahrheit dahingestellt sein läßt. ... Nur der versteht, der sich selber aus dem Spiele zu lassen versteht." Ibid. II.II.2.c, p. 317.

[172] Ibid. II.II.2.c, p. 316.

[173] "Rechtsgemeinschaft". Ibid. II.II.2.c, p. 312.

community consist of both received tradition and of living practice. Where there is no community under law, judicial hermeneutics is not possible, because the declarations of a tyrannic ruler can immediately and unforeseeably annihilate any given rule of law. Understanding the declarations becomes then not a task of understanding the application of law, but instead of understanding the self-serving interests of a tyrant.

Where judicial hermeneutics is possible, the task of a legal professional is to determine the normative content of the law so that he can anticipate what the court will do. To do this, he will look at the text of the law, and in order to understand exactly what the text means he must look at how the law has been used in the past. The law speaks, for example, of "intent", but what exactly does that mean? What are the requirements to proving in court that there was intent? To answer this, the legal professional must look at the history of the use of the law, and discern the intention behind the wording.[174]

An historian of law that is interested not in the current application of the law, but in how it was used in the past, must do the same deliberation as the present legal professional in trying to discern how to apply the law.[175] That is, to understand how the law was used in the past he must understand the consequences of applying the law in this way or that. To do that he must have knowledge of how to apply the law, and that knowledge must ultimately have its foundation in the present practicing of the law, for the present use sets the perspective in which all interpretations of the past are made.[176]

An interpretation of received tradition is, by its very nature always seeking some application, though this does not necessarily have to be a concrete task.[177] The application of history is to see each single text

[174]Ibid. II.II.2.c, p. 308 f.

[175]"Ein unmittelbares Zugehen auf den historische Gegenstand, das seinen Stellenwert objektiv ermittelte, kann es nicht geben. Der Historiker muß die gleiche Reflexion leisten, die auch den Juristen leitet." Ibid. II.II.2.c, p. 310.

[176]Ibid. II.II.2.c, p. 311.

[177]"Der Interpret, der es mit einer Überlieferung zu tun hat, sucht sich dieselbe zu applizieren. Aber auch hier heißt das nicht, daß der überlieferte Text für ihn als ein Allgemeines gegeben und verstanden und danach erst für besondere Anwendungen

as a source of received tradition.[178] A person who understands a text is always personally connected to the text and future generations must necessarily understand the text in a different way.[179]

The application of history is made complete by the historical critique of received tradition. This is what it means to be conscious of effective history.[180]

3.3.9 *Ethical and technical knowledge*

As mentioned above, philosophical hermeneutics is intended as a general theory of true understanding, which means that the hermeneutical insights apply in all cases without exception. However, following the rise of modernistic science it has become the norm in epistemological philosophy to draw a distinction between the natural sciences, on one hand, and the humanities – or as J.S. Mill called them, the moral sciences – on the other.[181] It is therefore appropriate here to discuss how the objectifying natural sciences are regarded in hermeneutical theory.

Gadamer's discussion of the different kinds of knowledge is based on Aristotelian ethics. The central problem of Aristotelian ethics is to

in Gebrauch genommen würde. Der Interpret will vielmehr gar nichts anderes, als dies Allgemeine – den Text – verstehen, d.h. verstehen, was die Überlieferung sagt, was Sinn und Bedeutung des Textes ausmacht. Um das zu verstehen, darf er aber nicht von sich selbst und der konkreten hermeneutische Situation, in der er sich befindet, absehen wollen. Er muß den Text auf diese Situation beziehen, wenn er überhaupt verstehen will." Ibid. II.II.2.b, p. 307.

[178]"Für den Historiker tritt jedoch der einelne Text mit anderen Quellen und Zeugnissen zur Einheit des Überlieferungsganzen zusammen. Die Einheit dieses Ganzen der Überlieferung ist sein wahrer hermeneutischer Gegenstand." Ibid. II.II.2.c, p. 322.

[179]"In allem Lesen geschieht vielmehr eine Applikation, so daß, wer einen Text liest, selber noch in dem vernommenen Sinn darin ist. Er gehört mit zum zu dem Text, den er versteht. . . . Er kann sich, ja er muß sich eingestehen, daß kommende Geschlechter das, was er in dem Texte gelesen hat, anders verstehen werden." Ibid. II.II.2.c, p. 323.

[180]Ibid. II.II.2.c, p. 323.

[181]Gadamer speaks about "Geisteswissenschaften".

examine what role reason plays in ethical behaviour.[182] Ethical knowledge[183] cannot be exact in the same way as for example mathematics can. Furthermore, ethical knowledge cannot be reduced to formality: the person that would act ethically must himself know and understand the situation adequately.[184]

Ethical knowledge is contrasted with technical knowledge.[185] The two are similar in that they are both forms of practice. The fundamental difference between them is that in ethical knowledge, we are not masters of the object of knowledge, whereas in technical knowledge we are.[186] The central question is thus one of mastery.[187]

The rules that a craftsman use to guide his work are a form of technical knowledge. These rules aim at perfection. By necessity, the rules cannot in practice be followed to perfection, but the craftsman would rather be without this imperfection; deviation is a sort of loss that is in a philosophical sense painful. Contrary to this, the law, which is a form of ethical knowledge, is by its essence imperfect. Softening the law to apply to the situation at hand, and showing mercy, is not a loss – this deviation does not result in a lesser law, but in a better one. Attempting to create a perfect law, a law that anticipates all possible situations, would result in a totalitarian law: the opposite of the desired. All ethical traditions, like the law, are tied to a specific time in history and to a specific nation. They are neither mere conventions, nor are they written in the stars.[188]

Ethical knowledge is to take counsel with oneself – technical knowledge is not. Ethical knowledge does not possess a prescient aspect in the same way technical knowledge does.[189] Technical knowledge can be taught to others, whereas ethical knowledge has to be lived; one

[182]Gadamer 1960 [1965] II.II.2.b, p. 295.
[183]"Sittliche Wissen", *φρόνησις*.
[184]Gadamer 1960 [1965] II.II.2.b, p. 296.
[185]*τέχνη*.
[186]In Aristotle's terms, ethical knowledge is "self-knowledge" ("Sich-Wissen") whereas technical knowledge is "for-itself-knowledge" ("Für-sich-Wissen").
[187]Gadamer 1960 [1965] II.II.2.b, p. 299.
[188]Ibid. II.II.2.b.1, p. 303 f.
[189]Ibid. II.II.2.b.2, p. 304.

cannot be taught how to live life, it has to be done. In the same way, it is easy to resolve to live a virtuous life, but to carry out the resolve is not as easy as that.[190]

To consider which is best out of a range of equally appropriate means is technical, but to consider which means are appropriate at all is ethical. Ethical knowledge includes knowledge of both ends and means, and is the fundamental form of experience.[191]

Experience can only be understanding if it is related to someone else, and if, through this relation, it is an expression of the will to do the right thing. Technical ability without an ethical goal or excellence without moral restraint is *δεινός*: horrible.[192]

The fields of science cannot simply be separated into those that are concerned with ethical knowledge and those that are concerned with technical knowledge. Because the humanities are concerned with studying the conditions of human existence, there is a strong affinity between ethical knowledge and the humanities, also known as the moral sciences. However, ethical knowledge is not identical to moral science. In addition to being contrasted with technical knowledge, ethical knowledge is contrasted also with theoretical knowledge,[193] of which mathematics is a prime example.

However, any good science partakes in ethical knowledge, regardless of the field. The different fields of science are demarcated by their objects of study, and they are of course influenced by the character of

[190]Carl von Clausewitz illustrates this aspect of ethical knowledge in his writings on the virtues of a military commander. The good commander needs a certain amount of experience that can only be accumulated in actual war, because "War is the realm of uncertainty; three quarters of the factors on which actions in war is based are wrapped in a fog of greater or lesser uncertainty." von Clausewitz 1832-34 [2007] book 1, chp. 3.

[191]"Das sittliche Wissen ist wirklich ein Wissen eigener Art. Es umgreift in einer eigentümlichen Weise Mittel und Zweck und unterscheidet sich damit vom technischen Wissen. ... Denn das sittliche Wissen enthält selbst eine Art der Erfahrung in sich, ja, wir werden noch sehen, daß dies vielleicht die grundlegende Form der Erfahrung ist, der gegenüber all andere Erfahrung schon eine Denaturierung, um nicht zu sagen Naturalisierung, darstellt." Gadamer 1960 [1965] II.II.2.b.2, p. 305.

[192]Ibid. II.II.2.b.3, p. 306 f. "Nichts ist so schrecklich, so unheimlich, ja so furchtbar wie die Ausübung genialer Fähigkeiten zum Üblen." Ibid. p. 307.

[193]Theoretical, or learned, knowledge: *ἐπιστήμη*. Ibid. II.II.2.b, p. 297.

that object, such that the natural sciences are strongly connected with technical knowledge and the moral sciences are strongly connected with ethical knowledge. But the scientific fields do not follow the distinctions of knowledge, and in any science worth its name, technical, theoretical, and ethical knowledge is found.

3.4 *Cultural form theory*

Applying cultural theory to the study of programming is like applying mathematical theory to a problem in electrical engineering. This is fruitful because mathematical theory is well suited to describe relationships between quantities, and those quantities can be defined and measured in terms of electrical circuits. When it comes to the study of why people do what they do, that is, the study of culture, there are rarely any measurable quantities or fixed relationship that can be described with something like a mathematical theory. Cultural theory, therefore, is concerned not with things that are certain but with things that are uncertain: how meaning arises, and the relationships between people. Just as mathematical theory is an abstract way of describing necessary relationships, and is useful for understanding quantities, so cultural theory is an abstract way of describing how meaning is made, and is useful for understanding human activity.

It is common to distinguish between culture and nature, or between human factors and physical factors, and even common to view them in opposition. While the distinction is useful, the notion of opposition is not congruent with how technological development actually takes place.[194] Human action always has a cultural as well as a natural, or physical, side: in this regard, culture and nature are different aspects of the same activity. The relationship between the two is an important concept at the base of cultural theory of technology.

[194]See Latour 1987 chp. 2.C, p. 94ff.

In order to conduct a cultural analysis of the work processes of programmers, it is necessary first to look at the concepts that will be used in the analysis: cultural practice and cultural form. The concepts as they are presented here come from the ethnological research community, particularly the Danish professor of ethnology Thomas Højrup and his followers. The specific presentation, however, is my own responsibility.

There are two fundamentally different ways of explaining what happens in the world. One perspective is teleological: I get in my car and drive because I want to get home. Getting home is my goal, my *télos* (τέλος), and driving in the car is the means of reaching my goal. The other perspective is causal: I press the gas pedal with my foot, which causes more fuel to be let into the combustion engine, which causes the car to move forward. The action is explained as a matter of cause and effect. If teleology and causality are viewed as unrelated or even contradictory principles, as is often the case, it becomes difficult to explain the world around us. For example, what is the explanation if I lose control of the car and drive into the ditch on my way home? In the teleological perspective, my strong desire to get home resulted in the means of getting there – speed – getting out of hand. In the causal perspective, the cause of the accident was the slippery road, which had the effect that the car lost traction, further causing me to lose control of the car.

The concept of practice[195] is a way of resolving the contradictions of teleology and causality. Teleology and causality are simply regarded as two opposite but equally valid aspects of practice. Furthermore, the concepts that make up teleology and causality are put into relation with each other: the goal of teleology is identified with the effect of causality; the means of teleology is identified with the cause of causality.

As a consequence of this, the concept of practice establishes a correspondence between the concepts of subjectivity and objectivity. The identification of means and cause shows that causal relations do

[195] Højrup 1995 p. 67.

not exist independently of ourselves.[196] It is the presence of a self-conscious subject that, through regarding the world as consisting of goals and means, identifies what is cause and effect. Thus subjectivity and objectivity are two perspectives on the world that proceed from the practice concept; again, they are opposite and both valid. Subjectivity is to take a teleological perspective on the world, and objectivity is to regard it as made up of cause and effect. The objective perspective is set by the subjective, and the subjective perspective is likewise set by the objective.[197]

The concept of practice is connected to hermeneutics in the way that a given practice is an expression of tradition – nothing comes from nothing; a practice must either be a repetition of an existing practice, or a break away from a former practice. Another point of connection is the purposefulness of practice: the teleological aspect of practice has a goal, and that goal is akin to the concept of application in hermeneutical theory.[198]

The realization of a practice will result in a plethora of related practices. What is a means in one practice is a goal in another practice: My goal is to get home, so I get in the car, which is the means of getting there. But while I am driving, the immediate goal becomes to keep the car safely on the road. Thus, the means to my original goal becomes a sub-goal in itself, in service of a higher goal.[199]

The concept of practice is closely connected to another concept: that of form. In cultural theory, the concept of form has a specific content that goes back to Aristotle and Plato.[200] A form is not simply

[196]Or more precisely, it is the designation of something as “cause” and something as “effect” that does not exist independently of ourselves. In a system that altogether lacks a teleological perspective, such as formal physics, everything affects everything else, so “cause” and “effect” become merely shorthand referrals to whatever interests us. Between two masses there will be a gravitational force at work, so we can say that the gravitational force causes the masses to attract each other. But we can equally say that the gravitational force is caused by the two masses being present.

[197]Højrup 1995 p. 69.

[198]See 3.3 (Hermeneutical theory) for an explanation of the hermeneutical concepts in this section.

[199]Højrup 1995 p. 74 ff.

[200]Højrup 2002 p. 375 ff.

the outer physical form of a thing, its *morphé* (μορφέ), as the word is often used in daily speech.[201] Rather, a form, *eidos* (εἶδος), is the idea of a thing, a structural concept that subjugates the matter of which it is made up. As such, the form of a fire, for example, can be made up of many different kinds of matter: wood, oil, tallow, *et cetera.* Each particular kind of matter will have different properties, which means that the resultant form – the fire – will also have many properties that are not essential, but accidental.

A form (*eidos*) is a solution to a problem, or a way of reaching a goal, that consists of an idea or structure that expresses the function of the form, plus the matter in which the idea is expressed and which is the means of the form:

$$\text{form} = \text{function (idea, structure)} + \text{matter (means)}.$$

The problem that a form solves comes from the goals of a practice. This means that it is ultimately practice that determines the forms, but constrained by the properties of matter. A form is a solution to the demands of practice, and demands are contradictory.[202] For example, a fire has to provide warmth and light, to be safe, and to conserve fuel. Not all of the demands can always be met efficiently at the same time. Thus, a bonfire and a oil lamp do the same thing, but they are well suited to different situations. On one level they are the same form – fire – and on another level they are different: warm-fire-burning-wood versus illuminating-fire-burning-oil.

For the purpose of cultural form theory, it does not matter whether a form is material, social, or theoretical. A fishing vessel, bureaucracy, and the rational numbers are all examples of forms.[203] However, we cannot hope intellectually to comprehend the existing forms fully. We are fundamentally limited by our experience with the practical world, and any concept of a specific form is always only a temporary and incomplete understanding of reality.[204] In this way, the concept

[201]Ibid. p. 378.
[202]Ibid. p. 380.
[203]Ibid. p. 384.
[204]Ibid. p. 376.

of form corresponds to the hermeneutical notion of understanding, which is fundamentally limited by the horizon of understanding.

3.5 *Rhetorical theory*

The primary focus of this dissertation is on hermeneutical and cultural form theory than on rhetorical theory. For this reason, a thorough explanation of rhetorical theory will not be given here. The rhetorical concepts that are used will be explained along with the analysis itself in chapter 8.

Fields other than programming have examples of rhetorical analysis of a practical bent. Of interest is McCloskey's excellent analysis of scientific writing in economics.[205] Also noteworthy is Kennedy's analysis of New Testament texts.[206]

A further illustrative example is Bruno Latour and Steve Woolgar's 1979 study of the work of neuroendocrinologists, a branch of the natural, or "hard", sciences. The study shows how scientific facts consist not only of observations from the laboratory, but also of rhetorical efforts to establish the explanations of observations as facts.[207] The rhetorical aspect of the scientists' work has the important function of persuading their colleagues, in order to strengthen the credibility of the scientists.[208]

It should be noted that hermeneutics and rhetorics are intimately connected; it is a central insight of both that doing and understanding are inseparable. Hermeneutics, then, looks at doing and understanding from the perspective of understanding, while rhetorics looks at the same thing from the perspective of doing.

[205]McCloskey 1985.
[206]Kennedy 1984.
[207]Latour & Woolgar 1979 [1986] p. 240.
[208]Ibid. p. 200.

Chapter 4

Method

The data on which this dissertation is based consist of a larger case study in game programming and a number of smaller case studies and interviews in safety critical programming. There is not a generally agreed upon definition in the literature of what constitutes a case study.[1] This thesis uses the definition put forth by Yin and Easterbrook et al.: “an empirical enquiry that investigates a contemporary phenomenon within its real-life context”.[2] For the purpose of this thesis, an inquiry can be based on data obtained either from participant observation or from interviews.

The case study in game programming is presented in section 5.1 followed by analyses. Two case studies in safety critical programming are presented in section 6.2 followed by analysis. Section 7.1 presents a comparative analysis of the cases in sections 5.1 and 6.2. Section 7.2 presents a number of other case studies in safety critical programming along with their comparative analysis. Chapter 8 presents some source code from the case study in game programming along with its analysis.

[1]Easterbrook et al. 2008 p. 296.
[2]Ibid.

4.1 *Choice of cases*

The choice of a small startup computer game company as one of the case studies presented advantages of a practical nature. Since the company is small and newly founded it is possible via participant observation to get an overview of the totality of the work process that is difficult to obtain in the case of larger and better established companies.

The primary reason for choosing a computer game company for study, however, is that it suits the argument in this dissertation well. It is argued in section 3.2 (Mainstream programming theories) that mainstream programming theories have some shortcomings in describing programming processes in general. This is easier to observe in some programming processes than in other. The game programming process is one in which it is easy to observe because the process is not derived from one of the mainstream theories and because the goals of game programming are relatively far from the administrative and computational origins of programming (see sections 3.2.1 and 3.2.3).

The second part of the empirical data consists of case studies of companies in the safety critical programming industries. Safety critical programming was chosen for two reasons. First, to provide a contrast to the work process of computer game programming. According to cultural form theory (section 3.4) it is important to study different forms of a practice (i.e. programming) in order to get a clear understanding of the practice. As safety critical programming is in many aspects different from game programming is serves this purpose well. From an ethnological point of view, making comparisons between contrasting empirical practices is a central method of increasing understanding of the practices.[3]

The other reason to study safety critical programming is that it is closely connected to software engineering and that its formal work processes often derive directly from this tradition. The shortcomings

[3]Christensen 1987 p. 14 ff.

of mainstream theories that are pointed out in section 3.2 form an argument for pursuing other theoretical perspectives on programming, in the case of this dissertation: hermeneutics. But in order for hermeneutics to be more than a mere contender to the mainstream theories it is necessary to show not only that hermeneutics is better suited to describing actual programming practices, but also that it is able to explain what function the mainstream theories have in programming practice. Since safety critical programming is ideologically derived from software engineering, it serves well as an example for this purpose. It would be possible, in future research, to do similar studies regarding Agile thinking and computer science theory.

4.2 *Participant observation*

Participant observation is an established method in software engineering research, cf. Seaman.[4] Participant observation was used in this dissertation to gather data for a case study in software engineering.

The author spent a total of 17 workdays in the period 15th August – 9th September 2011 observing the work at the small game company Tribeflame in Turku, Southwestern Finland. The observation normally took place in the developers' office whenever they were present. In addition to strict observation, conversations, regular interviews and collection of some documents took place.

Seaman does point out the risk of obtaining inaccurate data due to the participant observer's bias. However, since hermeneutical analysis is an interpretive approach participant observation is deemed to be the best alternative for data collection:

> "Qualitative data is richer than quantitative data, so using qualitative methods increases the amount of information contained in the data collected. It also increases the diversity of the data and thus increases confidence in the re-

[4]Seaman 2008 chp. 2.1, p. 37.

> sults through triangulation, multiple analyses, and greater interpretive ability."[5]

4.2.1 *Interdisciplinary concerns*

A number of researchers within social science and the humanities perceive the use of ethnographic methods, of which participant observation is an example, within the technical fields and natural science to be a contentious issue.[6] The concern is that ethnographic method has a subjective and holistic character that is violated when subjected to the demands of objective measurement of the technical and natural scientific fields. From the point of view of hermeneutical theory this implicit opposition between subjective and objective observations is misguided, as explained in section 3.3.5. The stance taken in this dissertation is that the ethnographic method used does indeed aspire to arrive at objective truth, but notably objective truth *as measured by the objectives of programming practice*, and not as measured by the objective criteria of either social/humanities research or technical/natural science.

4.3 *Interviews*

Interviews were used in this dissertation as part of the participant observation of game programming, and as the primary means of gathering data on safety critical programming. All the interviews were conducted as semi-structured interviews.[7]

[5]Ibid. p. 60f.
[6]Jensen 2008.
[7]Seaman 2008 p. 44.

Similarly to participant observation, interviews is an established data collection method in software engineering. It fits well with many types of approaches and theoretical frameworks.[8]

4.4 *Ethnological method*

From an ethnological point of view, the methods used in this dissertation are not unusual. Both interviews and participant observation are well-established ethnological methods.[9] The subject of the dissertation is not commonplace within ethnology, but it is not unheard of either (see section 2.2.1). The main theory of this dissertation, hermeneutical theory, is a well-established theoretical perspective within ethnological research.[10] The secondary theory of this dissertation, cultural form theory, is used in a variant, life mode theory, in numerous recent ethnological studies, primarily by Danish ethnologists.[11] The theory used least in this dissertation is rhetorical theory. Rhetorical theory is not much used within ethnology though examples of its use do exist.[12]

The fieldwork (participant observation and interviews) done for the sake of this dissertation gave rise to some interesting choices and strategies regarding data collection. These are discussed further in a published paper by the author, which for the sake of convenience is reprinted in this dissertation.[13]

[8]Singer et al. 2008 chp. 3.1.2, p. 14.
[9]Fägerborg 1999. Öhlander 1999.
[10]Borda 1989 p. 20. Pedersen 2005 p. 8.
[11]See e.g. Højrup 2002. Højrup 2003. Nielsen 2004. Suenson 2005.
[12]See e.g. Suenson 2008 p. 91.
[13]Suenson 2013. See appendix A.

4.5 *Analysis methods*

4.5.1 *Mainstream analysis*

It is asserted in this dissertation that a cultural approach to programming research is needed to supplement the mainstream approaches. To back up this assertation, the most prominent mainstream theories in programming research are used to analyze a case study in game programming, and the shortcomings of the mainstream theories are pointed out.

The mainstream theories in programming are presented in section 3.2. This presentation is in itself a conceptual analysis that identifies the basic concepts on which the mainstream theories are based. The mainstream perspectives are then applied to the case study in section 5.2.1.

4.5.2 *Hermeneutical analysis*

The hermeneutical analyses use the theory presented in section 3.3. Hermeneutical analysis is an interpretive approach. The facts of the case studies are seen through the perspective of hermeneutical concepts. The result is a hermenutically structured understanding of the case studies. The analysis is applied to the game programming case study in section 5.2.2 and to two safety critical case studies in section 6.3.

The analysis method itself is not so much a procedure as it is a process of understanding, similar to the constant comparison method described by Seaman.[14] This means that the data are interpreted repeatedly in a hermeneutical perspective until they make sense within the framework. The final interpretation is the result of the analysis.

[14]Seaman 2008 chp. 3.1.1, p. 49.

4.5.3 *Cultural form analysis*

4.5.3.1 *Comparative analysis*

The hermeneutical analyses of sections 5.2.2 and 6.3 provide insights into game programming and safety critical programming but they do not explain the differences between the two types of programming. Therefore, a comparative cultural form analysis is applied to the case studies in game programming and safety critical programming and to the hermeneutical analyses of these in section 7.1. Applying a cultural form analysis is similar to applying a hermeneutical analysis except that the concepts used are different.

The game programming data come from a single relatively detailed case study while the safety critical programming data come from two less detailed case studies. This is, however, not an obstacle for the analysis, since cultural form theory perceives all practices as more or less generalized forms. An uneven data material can easily be compared because what is being compared are not the concrete practices but the forms of practices. See section 3.4 (Cultural form theory).

4.5.3.2 *Safety critical case studies*

Looking at the safety critical programming process as a cultural form is an abstraction that disregards the variety found within the safety critical industries. In section 7.2 a number of case studies in safety critical programming are compared to each other using cultural form theory, similar to the comparison between game programming and safety critical programming, but within an industrial paradigm instead of between two different paradigms. The case studies used are based on interviews, similar to the two case studies that were subjected to hermeneutical analysis. However, they are treated in much less detail compared to the ones subjected to hermeneutical analysis.

4.5.4 *Rhetorical analysis*

This dissertation focuses on hermeneutical theory, not rhetorical. Yet, a rhetorical analysis is included for two purposes: to give a demonstration of how to carry out a rhetorical analysis, and to show the intimate connection that exists between rhetorics and hermeneutics.

The analysis is based on source code obtained from the case study in game programming as well as two interviews with programmers in which they explain the company's source code. Only a small part of one particular game's program code is analyzed, and not in depth.

Though the rhetorical analysis makes up only a small part of this dissertation it is important that it is included. The dissertation is directed towards programming practice and as such it is necessary that the results can be used in said practice.[15] Hermeneutical theory in itself has the potential to be a valuable conceptual tool in the programming process but in the inevitable interplay between understanding and application (in the form of creation) that characterizes a hermeneutical process such as software development, hermeneutics focuses more on the understanding part of the process while rhetorics focuses more on the creative part. For this reason rhetorics has the potential to become an important practical conceptual tool for programmers, as demonstrated in chapter 8 (Rhetorical case study and analysis); not only in the creation of source code but also in the communication between developers. This is the reason why it has been crucial to include an example of rhetorical analysis in this dissertation and it points towards a fruitful area of analysis for future research.

4.6 *Theory of science*

Runeson & Höst divide case study research into four categories according to purpose: exploratory, descriptive, explanatory, and improv-

[15]See sections 3.3.5 (Understanding) and 3.3.8 (Application).

ing. According to this classification, this dissertation is exploratory: it aims at “finding out what is happening, seeking new insights and generating ideas and hypotheses for new research.”[16]

They also identify three types of research perspective: positivist, critical, and interpretive. According to this classification, this dissertation is interpretive. Overall, hermenutics is a very general theory of interpretation and it is used not only as a method of analysis but also as the underlying theory of science of the dissertation. The use of hermeneutical theory in science is discussed further in the paper “Method and Fieldwork in a Hermeneutical Perspective.”[17]

4.7 *Data*

The game programming data were collected specifically for this dissertation and not used in any other setting. The safety critical programming data were also used as survey data in work product D4.2a.2 of the RECOMP joint research project.

4.7.1 *Game programming*

Observation by the author in the company Tribeflame in Turku, Finland during the period 15th-19th August, 22nd-24th August, 30th-31st August, 1st-2nd September, and 5th-9th September 2011. Observation time was normally around 9 AM to 5 PM, depending each day on the working hours of the company.

[16]Runeson & Höst 2009 p. 135.
[17]Suenson 2013. Also reprinted in appendix A.

Field diary — A handwritten field diary was kept wherein observations were systematically noted throughout the day. The owners and programmers of Tribeflame spoke Swedish amongst each other, and the graphic artists spoke Finnish, meaning that company meetings were also in Finnish. The diary is kept mostly in Danish mixed with Swedish and Finnish phrases. 233 pages

Interview 1 — Swedish. 1 hour 34 minutes. 19th August 2011. Björn (name changed). 33 year old male, born in Helsingfors. Founder and owner of Tribeflame.

Interview 2 — Swedish. 1 hour 54 minutes. 23rd August 2011. Mickie (name changed). 38 year old male, born in Helsingfors. Programmer in Tribeflame.

Interview 3 — English. 1 hour 27 minutes. 24th August 2011. Kati (name changed). 30 year old female, born in Kotka. Temporary graphic artist in Tribeflame.

Interview 4 — English. 2 hours 1 minute. 30th August 2011. Matti (name changed). 31 year old male, born in Parainen. Graphic artist in Tribeflame.

Interview 5 — Swedish. 1 hour 57 minutes. 1st September 2011. Andreas (name changed). 34 year old male, born in Åbo. Founder and owner of Tribeflame.

Interview 6 — Swedish. 1 hour 34 minutes. 1st September 2011. Interview with Mickie where he explains the source code for the game with the working title *Flower*.

Interview 7 — Swedish. 1 hour 11 minutes. 6th September 2011. Interview with Andreas where he explains the internally developed library code at Tribeflame and the company's technological strategy.

Interview 8 — Swedish. 1 hour 33 minutes. 22nd September 2011. Fredrik (name changed). 25 year old male, born in Korsnäs. Part time programmer in Tribeflame.

Presentation — English. Presentation of Tribeflame by Björn at the Department of Information Technologies, Åbo Akademi, 4th December 2012 at 9 AM – 10:10 AM. Notes. 5 pages.

Documents — Concept sketch and various documents used in meetings by Tribeflame. Printed and hand written. 20 pages.

Source code — Printed source code from the game with the working title *Flower*. From the files "GameScene.hpp", "GameScene.hpp", "Obstacle.hpp", and "Obstacle.cpp". 28 pages.

Diagrams — Hand drawn diagrams explaining the structure of Tribeflame's library code. 2 flip-chart sheets.

News article — "Akademisk spelhåla på mässan". *Meddelanden från Åbo Akademi* no. 13/2012, p. 28.

Photographs — Photographs of Tribeflame's office taken 2nd September 2011.

4.7.2 *Safety critical programming*

"RECOMP" stands for Reduced Certification Costs Using Trusted Multi-core Platforms and is a European Union-funded project from ARTEMIS (Advanced Research & Technology for Embedded Intelligence and Systems) Joint Undertaking (JU). The project started 1st April 2010 and had a duration of 36 months. The aim was to establish methods, tools and platforms for enabling cost-efficient (re)certification of safety-critical and mixed-criticality systems. Applications addressed were automotive, aerospace, industrial control systems, and lifts and transportation systems.[18] The Software Engineering Laboratory at Åbo Akademi participated in the project through professor Iván Porres, Jeanette Heidenberg, and the author. The interviews

[18] Information from the official RECOMP website.

listed here were conducted and transcribed by the author except in one case, noted below.

Interview 9 — Swedish. Åbo Akademi. 50 minutes. 21st January 2011. 38 year old female, born in Mariehamn. M.Sc. in computer science. Software design architect in a large telecommunications company, Finland.

Interview 10 — English. Telephone interview. 46 minutes. 2nd March 2011. Jensen, Martin Faurschou. 34 year old male, born in København. M.Sc. in engineering. Part of functional safety team at Danfoss Power Electronics, Graasten. Jeanette Heidenberg conducted the interview.

Interview 11 — English. Skype telephone interview. 1 hour. 7th March 2011. Ambrosio, Gustavo. 26 year old male, born in Madrid. Electrical engineer, masters degree in aerospace engineering. Software engineer. Integrasys, Madrid.

Interview 12 — English. Telephone interview. 1 hour 1 minute. 9th March 2011. Two persons interviewed. 33 year old male, born in Brno. Degree in electrical engineering and computer science. Responsible for quality and ISO standard, project manager. 45 year old male, born in Brno. Degree in electrical engineering and computer science. Co-founder. Company in image and signal processing, industrial and traffic management, Brno.

Interview 13 — English. Telephone interview. 1 hour 3 minutes. 10th March 2011. 40 year old male, born in Italy. Aerospace engineer. Director of critical real time software in a space industry company, Finland.

Interview 14 — Danish. Telephone interview. 1 hour 8 minutes. 16th March 2011. Jessen, Poul. Male. Electrical engineer. Director and owner of PAJ Systemteknik, Sønderborg.

Interview 15 — English. Telephone interview. 59 minutes. 17th March 2011. Loock, Detlef. 51 year old male. Electrical engineer. Group leader for quality assurance in functional safety in Delphi Automotive, Wiehl.

Interview 16 — English. Telephone interview. 1 hour. 24th March 2011. Philipps, Jan. 42 year old male, born in Saarland. Degree in computer science and executive MBA in innovation and business creation. Co-founder and management board member of Validas, München.

Interview 17 — English. Telephone interview. 30 minutes. 25th March 2011. Slotosch, Oscar. 45 year old male, born in München. PhD in computer science. CEO, co-founder, and management board member of Validas, München.

Interview 18 — English. Telephone interview. 1 hour 2 minutes. 25th March 2011. 44 year old male, born in Germany. PhD in computer science. Quality manager, previously team manager for electronic control units business unit in a automotive company, Germany.

Interview 19 — English. Telephone interview. 58 minutes. 29th March 2011. 38 year old male, born in Germany. PhD in electrical engineering. CTO in a company that makes software development tools for the automotive industry, Germany.

Interview 20 — Danish. Telephone interview. 1 hour 2 minutes. 30th March 2011. Riisgaard-Jensen, Martin. 49 year old male, born in København. Master in electrical engineering. Software project cooperation coordinator in Skov, Glyngøre.

Interview 21 — English. Telephone interview. 1 hour 1 minute. 4th April 2011. Delebarre, Vẽronique. 53 year old female, born in France. PhD in computer science. CEO and founder of Safe River, Paris.

Interview 22 — English. Åbo Akademi. 1 hour 7 minutes. 15th April 2011. Two persons interviewed. Tolvanen, Markku. 41 year old male, born in Lappeenranta. Computer science engineer. Principal designer in embedded systems. Hakulinen, Sami. Male. R&D Manager. Metso Automation, Finland.

Interview 23 — English. Telephone interview. 1 hour 6 minutes. 2nd May 2011. Two persons interviewed. Tchefouney, Wazoba. 32 year old male. Diploma engineer from Brest. Networks specialist, electronic architecture, research and innovation department. Graniou, Marc. 37 year old male. Diploma engineer from Brest. Specialist in safety domain, electronic architecture, research and innovation department. PSA Peugeot Citroên, Paris.

Interview 24 — English. Telephone interview. 1 hour 12 minutes. 4th May 2011. Two persons interviewed. Honold, Michael. 49 year old male, born in Germany. Electronics engineer. Hardware certification expert. Bitzer, Holger. 39 year old male, born in Germany. Electronics engineer. Project responsible for subsystems engineering. Cassidian Electronics, EADS, Ulm.

Interview 25 — English. Pasila. 1 hour 5 minutes. 5th May 2011. Two persons interviewed. Longhurst, Andrew. 40 year old male, born in Kent. Masters in robotics and automation. Engineering manager and quality manager. Davy, William. 25 year old male, born in Johannesburg. Masters in engineering. Senior engineer. Wittenstein Aerospace and Simulation, Bristol.

Interview 26 — English. Pasila. 1 hour 3 minutes. 5th May 2011. 48 year old male, born in Stanford. Computer science degree. Engineer in industrial research company, England.

Interview 27 — English. Telephone interview. 57 minutes. 17th May 2011. Two persons interviewed. 35 year old female, born in Zaarbrücken. Industrial engineer and quality assurance manager. 37 year old male, born near Hannover. Computer scientist and project manager. The company develops a real-time operating system. Germany.

Interview 28 — English. Telephone interview. 56 minutes. 23rd May 2011. Marino, Javier Romero. 43 year old male, born in Madrid. Aeronautic engineer. Project manager in telecommunications and control systems department in FCC, Madrid.

Interview 29 — English. Telephone interview. 1 hour. 26th May 2011. Suihkonen, Kari. 43 year old male, born in Parainen. Masters in physics. R&D division director in Kone, Chennai.

Interview 30 — English. Telephone interview. 31 minutes. 5th July 2011. Male. COO and R&D Director in a hardware company working with video surveillance and in the space industry, Spain.

Interview 31 — English. Telephone interview. 1 hour. 14th July 2011. Brewerton, Simon. 41 year old male, born in London. BSc in cybernetics and control systems. Senior principal for microcontroller division in Infineon, Bristol.

Interview 32 — English. Telephone interview. 1 hour 9 minutes. 28th May 2013. 32 year old male, born in Germany. Diploma engineer in communication systems. Certifier and group leader of generic safety systems in TÜV Süd, Germany.

Presentation — English. William Davy, Wittenstein Aerospace and Simulation. Presentation on Free, Open and Safe RTOS to the Embedded Systems Laboratory at Åbo Akademi 23rd May 2012, 2 PM to 3 PM. Notes. 4 pages.

Web sites — Official web sites of companies participating in RECOMP.

Documents — Presentation slides and various internal information documents provided by companies participating in RECOMP.

Deliverables — Official deliverables and intermediate work products of RECOMP, especially WP4.2a.

Chapter 5

Game programming

5.1 *Case study*

5.1.1 *Tribeflame*

In order to learn something about programming, we need to take a look at a programming process. In section 3.2 (Mainstream programming theories), we studied the models of programming processes that have already been made and discussed in the literature. But any model of programming processes must necessarily be an idealisation, to some degree; and while we are interested in finding out the ways in which these idealisations are useful, but to do that we cannot begin with the models. We have to start from the concrete, real work processes that the models represent.

Therefore in this chapter, we will study the work process of the company Tribeflame in Turku, Southwestern Finland, during four weeks in August and September 2011. Tribeflame is a small company that makes computer games for tablet computers, primarily Apple's iPad. At the time of the study period, Tribeflame had completed six

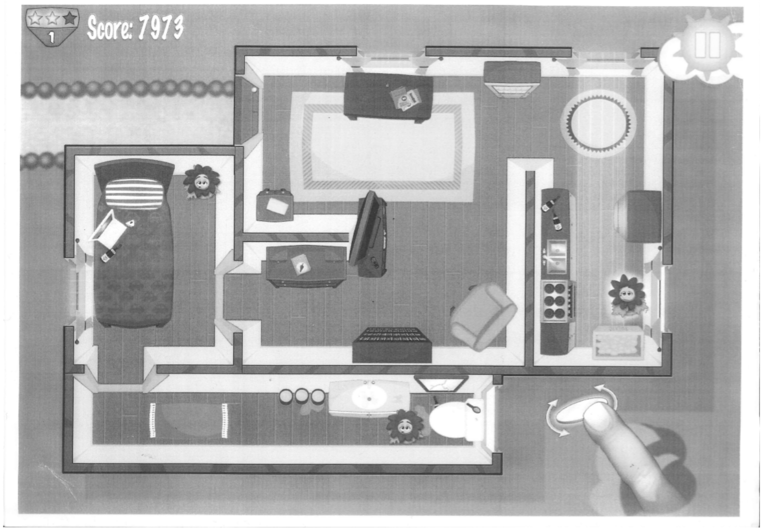


Figure 5.1: Concept sketch of the tablet computer game with the working title Flower, *which the Tribeflame developers worked on during the observation period.*

games. Over the course of the study, the company worked on developing three different games, though most of the time was spent on one of these (see Figure 5.1). At the time, Tribeflame consisted of the two founders, two programmers, and two graphic artists, though out of the six one worked only part time, and one was only employed for the summer.

Is the development process used by Tribeflame typical? There is no reason to doubt that it is a typical small game company.[1] However, we are not really interested in the question of whether Tribeflame's process is typical of programming in general: any real process will have its own peculiarities and in this sense be atypical. We are

[1]In 2010, 4473 out of 4981 Finnish software companies had 9 or fewer people working in them. In 2011, 3% of Finnish software companies (around 150) were in the game industry. Rönkkö & Peltonen 2012.

looking for some cultural traits and constraints on the possible forms computer programming can take, and for this purpose Tribeflame serves very well as an example. Later on in chapter 6 (Safety critical programming), we will take a look at some very different development processes.

5.1.2 *The development team*

Björn and Andreas[2] are the founders and owners of Tribeflame. They became friends in university and started Tribeflame together after having worked as employees for other companies for five or six years after graduation. Björn acts as the company's Chief Executive Officer (CEO), and takes care of most of the administrative side of running the business. The largest part of his work, however, is to do the level design for Tribeflame's games. That means thinking up the puzzles that the player has to solve when playing the games. Björn has the final say regarding how the games look and work. Andreas acts as the Chief Technological Officer (CTO): he works mainly as a programmer, and has the final say in technical decisions.

Mickie is employed full time as a programmer. Before Tribeflame, he worked as an employee elsewhere in the IT industry. Björn, Andreas, and Mickie all have technical IT degrees from the same Finnish university.[3]

Matti is employed as a full time graphic artist in Tribeflame, responsible for producing almost all the graphics for the games. Fredrik, a programmer, is finishing his education at the university while working part time; he is about to start his own company in an unrelated business, which means that he is slowly ending his time with Tribeflame. Kati is a graphic artist who was employed for the summer to replace Matti while he was on vacation.

[2]The names of the owners and employees of Tribeflame have been changed in this treatise to protect their privacy. The name of the company itself is the real name.

[3]Åbo Akademi, the Swedish language university in Finland.

Tribeflame mostly works as a team of individuals with specialised competences. The graphic artists Matti and Kati do not have the skills to do what the programmers, Andreas, Mickie, and Fredrik, are doing, and vice versa. Decisions about running the business are taken jointly by Björn and Andreas and do not involve the employees. Besides the administrative decisions, Björn and Andreas share the formal decision competence, with Björn being in charge of product decisions and Andreas in charge of strategic technical decisions. However, most of the decisions regarding the products (the games) are arrived at in a common process of discussion and decision that involves everyone in the company.

5.1.3 *The rhythm of work*

The work at Tribeflame has a certain rhythm. Most of the work takes place in the company's office, a single room in an office building next to the university's IT facility in Turku. At times, everyone is focused on their own tasks: the room is silent except for the clicking of mice and the tapping of keyboards, and the concentration is almost palpable.[4] At other times the room is alive with laughter and jokes – people constantly interrupt each other and show each other games and graphics on their screens. On frequent occasions, everyone leaves their desks and gathers together around the table in the center of the room in order to have a meeting.

In figure 5.2 we see the three main activities that are usually carried out alone, and how often they happen. Coding refers to the part of programming that is actually entering the programming code on

[4]Robert Willim describes a similar atmosphere in the Swedish IT company Framfab: "As mentioned, a quiet sense of being busy prevailed in the Ideon office. The mood of relative ease was partly due to much of the awareness being focused on the interface to the technology. The employees' concentration and focus was on what Steven Johnson (1997) calls the data rhythm. From this arises a contemplative concentration, trained inwards and at the same time connected to technology." Willim 2002 p. 84.

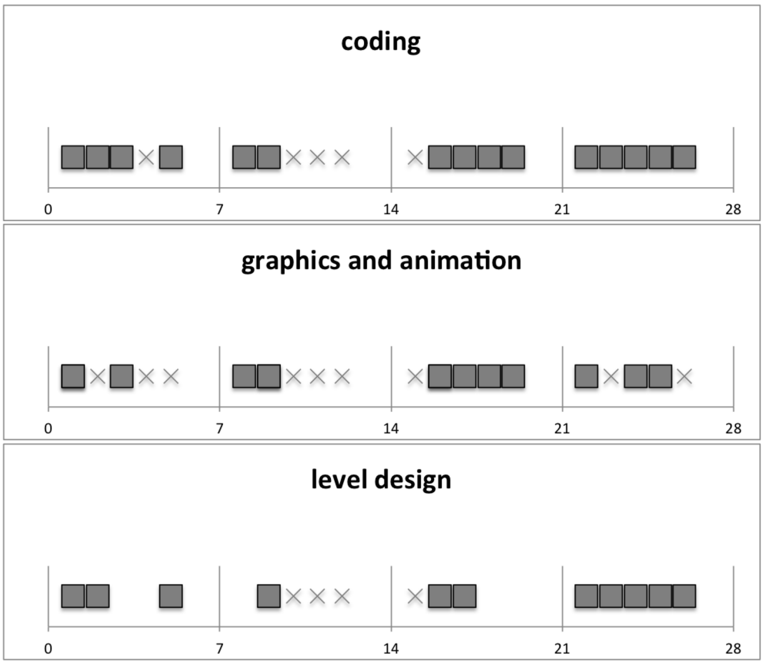


Figure 5.2: Frequency of tasks at Tribeflame during the observation period. The axis represents days. The boxes indicate that the activity occurred on that day, the crosses indicate missing observation data.

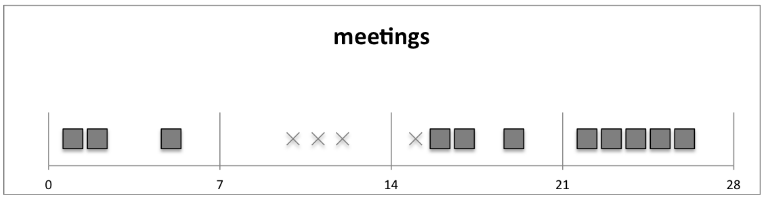


Figure 5.3: Frequency of meetings within Tribeflame. The axis represents days. The boxes indicate that the activity occurred on that day; the crosses indicate missing observation data.

the computer, as opposed to planning it and talking about it. This task is done by the programmers. Graphics and animation refers to drawing the pictures and animations used in the games, and this is done by the graphic artists. Level design refers to thinking up the puzzles the player has to solve and entering them into the game in a suitable form. This task is done by Björn, the CEO.

We can see that these tasks are very frequent. Coding and graphics work happens every day; level design happens whenever Björn is free from more important duties, though still quite frequently. These are the tasks that directly contribute to the games, so it is not surprising that they occur so often.

However, equally important as the time spent working alone is the interaction within the company. Figure 5.3 shows the occurrence of meetings, which happen almost as frequently as the activities that contribute directly to the games. Occasionally the meetings are short but commonly last for an hour or two. On Fridays the meetings sometimes take up most of the day. During the final two weeks of the observation period, Björn instituted a daily meeting. Note that Figure 5.3 shows only internal meetings in the company – Björn and Andreas also have meetings with external contacts.

This means that the work within Tribeflame is forever rhythmically shifting between working more or less individually on tasks, and coming together to discuss the tasks. If viewed from above, the work in Tribeflame looks like the schematic drawing in Figure 5.4: the de-

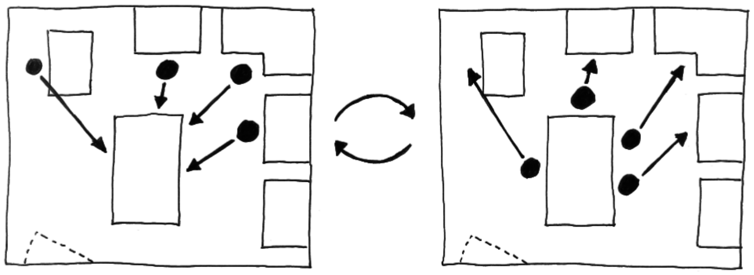


Figure 5.4: The work at Tribeflame shifts from the individual desks in the periphery of the room to the meeting table in the center (left); then it shifts back again to the periphery (right). This process goes on continually.

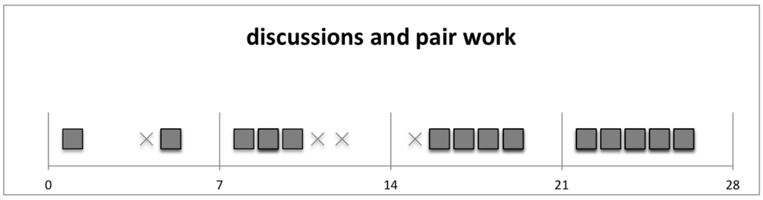


Figure 5.5: Frequency of discussions, work-related chats, and working together on tasks. The axis represents days. The boxes indicate that the activity occurred on that day; the crosses indicate missing observation data.

velopers move from their desks at the periphery of the room to the meeting table in the center; then they move back away from the center to the periphery, and so on and so forth.

The meetings at the central table are not the only form of exchange between the developers. As shown in Figure 5.5, interaction is just as frequent in the form of discussions, informal chat, banter, evaluation of each others' work, or working together on solving tasks. Schematically, the drawings of Figure 5.4 have to be complemented by the one in Figure 5.6, which shows the interaction that takes place between the developers when they are not in meetings.

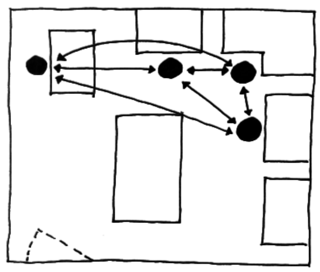


Figure 5.6: The developers also interact frequently outside of meetings.

The time spent on interaction is significant. As an example, Figure 5.7 shows the work interaction of Andreas on an ordinary Wednesday.[5] On this day, everyone spent more than half their time working with others. Over a third of the time was spent with everyone in a meeting around the central table (four people were at work during this week). Around a fifth of the time was spent in interactions in smaller groups, discussing and chatting.

As we see from Figure 5.7, there are two main blocks of time spent alone: one in the morning and one after lunch, beginning at around 220 minutes. Each of these blocks are followed by a meeting period involving everyone. Other types of interaction are spread throughout the day. Thus we see both the rhythmic movement between center and periphery, which is illustrated in Figure 5.4, and the individual interactions shown in Figure 5.6.

The lesson of this is that the Tribeflame work process is not only a coordinated effort between specialised individuals – it is also an intensely social and communicative process in which common discussions are central. The function of the discussions is to build understanding in the company. Decisions are then made on the basis of this common understanding, which is constantly evolving and corrected by the developers. Though the developers have specialised roles, they all contribute to the common direction of the company's games, and

[5]This particular day was chosen as an example in an attempt to find a typical work day: although it should be noted that all the observed days had some atypical features.

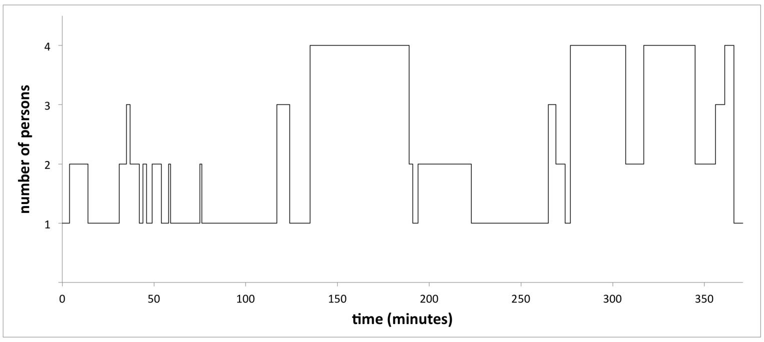


Figure 5.7: Work interaction for Andreas, Wednesday 7th September 2011, 9:00-15:00. When the number of persons is one it means that he is working alone; when it is greater than one that he is working together with others. The number of persons at work this week was four, so the high points in the graph indicate the whole company working together.

it is important that they are all heard. For example, though Kati is a temporary employee she is expected, even on her last work day, to participate in the development meeting and contribute her opinions,[6] regardless of the fact that she will not be involved in the development process anymore.

The focus on common understanding and consensus-building discussions means that an authoritative decision is rarely made. For example, after a lengthy discussion between the developers about where on the screen the game menu should be placed, Björn finally takes a decision because agreement cannot be reached: "In the end I decide that it will be moved to the right. Sorry guys."[7] Though Björn clearly has the authority to make the decision, and it is socially acceptable for him to do so, the fact that he feels the need to give an apology shows that this is not the normal way of reaching a decision in the company.

[6]Field diary, Friday September 2nd 2011, 11:00-12:24.

[7]"Til slut bestemmer jeg det bliver flyttet til højre. Sorry guys." Field diary, Thursday 8th September 2011, 16:15.

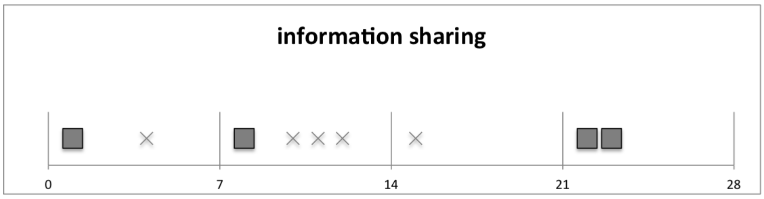


Figure 5.8: The relative infrequency of information sharing. Information sharing is when one person provides the information and the other merely listens and can ask clarifying questions. The axis represents days. The boxes indicate that the activity occurred on that day, the crosses indicate missing observation data.

The focus on understanding also means that the kind of communication that could be called "information sharing" is relatively infrequent: if we understand information sharing to be a process whereby one person gives some information to another person, who mainly listens and can ask clarifying questions. As can be seen in Figure 5.8, information sharing is much less frequent than the more collaborative meetings and discussions shown in Figure 5.3 and 5.5. That the communication patterns in Tribeflame are characterised more by discussions than by information sharing is somewhat paralleled by the decision process, which can better be described as "decision reaching" than as "decision making". The term "decision reaching" emphasizes that agreement and exchange plays a much larger role than in a process where decisions are simply made on the basis of the best available information.

5.1.4 *Playing and planning*

Playing the games that are being developed is an important part of the work process. For this, I use the term "playtest". The developers continually run the game to see how some detail has turned out, but

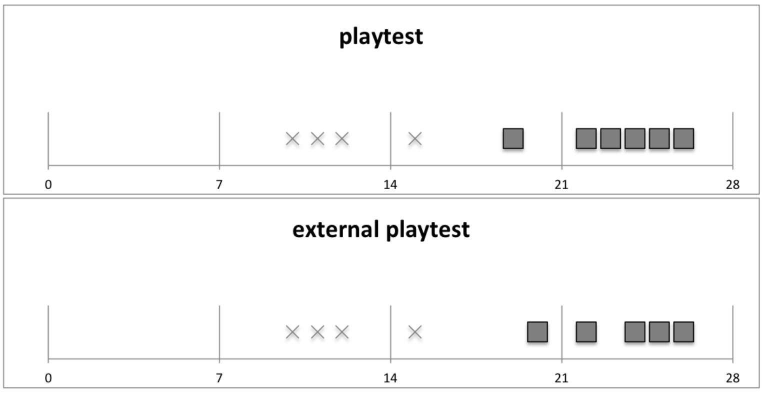


Figure 5.9: Frequency of playtest by the developers themselves (top), and by persons outside Tribeflame (bottom). The axis represents days. The boxes indicate that the activity occurred on that day; the crosses indicate missing observation data.

this is not what is meant by playtest. Rather, it is playing the game for a longer period to see how it works as a whole and how much fun it is. In this sense, there are two kinds of playtest: one when the developers themselves play their game, and another ("external playtest") when they get someone from outside the company to play it and talk about their experience.

Figure 5.9 shows the frequency of internal and external playtest. It indicates that the developers do not start to perform playtests until near to the final week of the observation period. At this time, however, playtests become a nearly daily occurrence, and the tests have a large influence on the discussion topics and meetings in the company. For the external tests, the developers ask colleagues in the game industry, colleagues in other branches of industry, relatives, spouses, and even me – almost everyone with whom they come into contact and can persuade to try the game.

Besides playing their own games, the developers also frequently play other companies' games and talk about them. Figure 5.10 shows

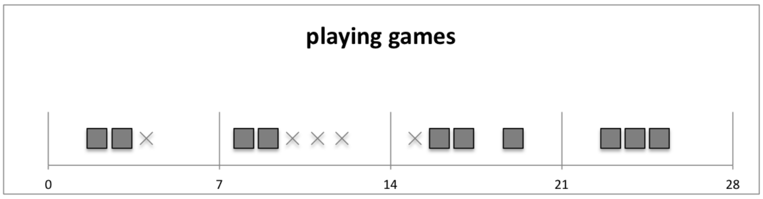


Figure 5.10: Frequency of playing other companies' games during work hours. The axis represents days. The boxes indicate that the activity occurred on that day; the crosses indicate missing observation data.

the frequency of playing other games during working hours. Since the company was only observed during work hours, there is no data on how often other companies' games are played outside work hours; but this clearly happens as it frequently features as a topic of conversation.

Planning in Tribeflame is neither very systematic nor very long term. There is a general idea about where the game is headed, but this is rarely written down or captured in a form other than concept sketches or similar. Tasks are usually planned one week ahead at the Friday meeting; they are written on a flip-chart or whiteboard and crossed out during the week as they are completed. New tasks that are discovered during the week are added to the whiteboard immediately. The tasks are frequently discussed during the course of the work. On an individual level, the developers sometimes write "to-do lists" for themselves on a piece of paper.

A humorous illustration of the *ad hoc* approach to planning in Tribeflame comes when Björn has difficulty wiping the whiteboard clean of old scribblings before a meeting. Instead of going in search of some spirit meant for cleaning he uses a small bottle of Minttu (Finnish mint liqueur) that happens to be standing around in the office, exclaiming: "whatever".[8] Though the gesture does not strictly have anything to do with the planning process, it illustrates very well the mindset in Tribeflame towards planning: do whatever works.

[8]Field diary, Friday 2nd September 2011, 11:11.

From a business perspective, the goal of Tribeflame is to produce games that can be sold and generate enough profit to sustain the development. This, however, does not explain what it is that makes it possible to sell a game, or how to make one. Mickie explains that "the code is just a means to reach a goal".[9] The game customers do not care about Tribeflame's business goal: they have their own goal, which is to have fun.

Consequently, many of the discussions at Tribeflame revolve around whether the games are fun and what it means for a game to be fun – both their own games and those developed by other companies. The developers are conscious that what they are doing is a form of entertainment. When presenting the company, Björn says that "entertainment is the constant in the company."[10] Andreas justifies charging money for the games by comparing them to other forms of entertainment: "it's a form of entertainment, it's fair to pay for games. People pay ten euro for two beers at a café."[11] – meaning that the entertainment value justifies charging more than the production costs.

5.2 *Analysis*

5.2.1 *Analysis with mainstream theories*

In section 5.1 (Case study), we discussed the work process of Tribeflame, a small computer game company of six people in Turku, Southwestern Finland. In section 3.2.1 (Software engineering), 3.2.2 (Agile software development), and 3.2.3 (Computer science), the most important mainstream theories of programming were explained. In this

[9]"koden er jo bare et middel til at nå et mål". Field diary, Thursday 1st September 2011, 13:00.

[10] Presentation of Tribeflame at Åbo Akademi, Tuesday 4th December 2011.

[11]"det er en form for underholdning, det er fair at betale for spil. Folk betaler 10 euro for to øl på café." Field diary, Wednesday 7th September 2011, 13:50.

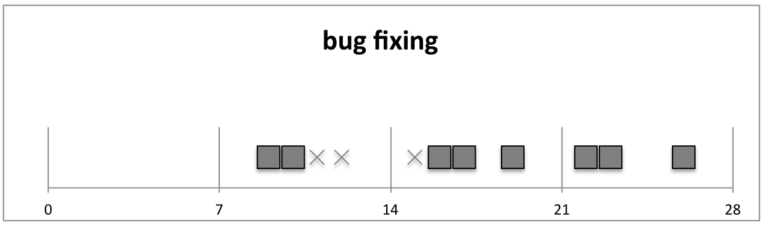


Figure 5.11: Frequency of bug fixing, a part of testing. The axis represents days. The boxes indicate that the activity occurred on that day; the crosses indicate missing observation data.

section we will attempt to apply the mainstream theories of programming to Tribeflame's process in order to see how well they are suited to analysis of a concrete programming process. The purpose of this is not to indicate where the mainstream theories are wrong, but to investigate whether they work as general programming theories. If they are truly general programming theories, they should be able to explain a viable commercial process as provided by the Tribeflame example. If they are not, science demands that we explain what situations the theories are suited to and what their limitations are.

5.2.1.1 *Software engineering*

Looking at the Tribeflame process, we see a distinct lack of the clearly separated phases that are the basis of software engineering theories. The phases that are common to nearly all theories are specification, architecture (or high level design), design, coding, and test. Of these, the specification, architecture, and design activities at Tribeflame are carried out during the meetings, shown in Figure 5.3 (page 144), and during the constant discussions and interactions outside meetings, shown in Figure 5.5 (page 145). As indicated, the activities are spread out over the duration of the process.

The same is the case with coding, as can be seen in Figure 5.2 (page 143). Testing is composed of several activities. According to traditional software engineering, bug fixing (error fixing) is part of testing. Bug fixing at Tribeflame is shown in Figure 5.11. Playtests and external playtests, shown in Figure 5.9 (page 149), also function as forms of testing. As we can see, testing occurs throughout the observation period except for the first week, and testing is concurrent with the specification, design and coding activities.

Even though the observation period covers but a small fraction of the development of a complete product, we find the activities of all of the phases of traditional software engineering represented within it. Moreover, the activities do not occur in an orderly, separated fashion, so we cannot interpret the development process as consisting of smaller iterations, each containing a complete set of phases, as advocated by some software engineering theories. The conclusion is that the activities described by the software engineering phases all occur in Tribeflame's process, but the idea of separating the activities into distinct phases does not help to explain what is happening in the process.

Corresponding to the lack of separate phases is a lack of transition criteria. In software engineering, transition criteria between phases are normally the completion of a document, work product, or artifact. In Tribeflame's work, there are remarkably few documents, and they do not play a central role. A few concept sketches and similar are important, and these are lying around in the office. But they are not kept up to date, and they are seldom referred to. Most of the knowledge is transmitted orally or in the game itself. The source code of the game can conceivably be thought of as a work product, but it does not function as a transition point since it is constantly evolving and never reaches a finished state that is signed off.

As we saw in section 3.2.1 (Software engineering), the main concerns of software engineering are planning, documenting, and evaluating. We can recall from section 5.1 (Case study) that in Tribeflame planning is done in an impromptu way on temporary flip-charts and whiteboards – far from the systematic planning and milestone-oriented methods promoted by software engineering, such as PERT and Gantt charts. Regarding documentation, Tribeflame produces almost none. Knowledge is kept in the developers' memories, in the

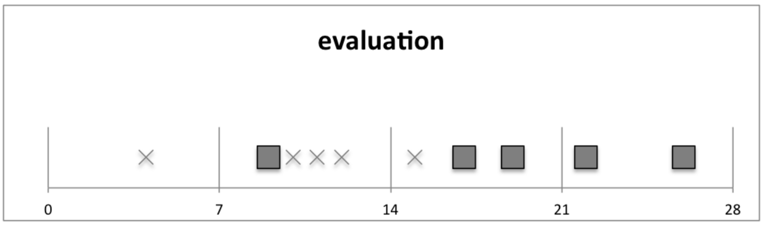


Figure 5.12: Frequency of evaluation sessions. The axis represents days. The boxes indicate that the activity occurred on that day; the crosses indicate missing observation data.

evolving source code, and occasionally concretized in a few concept sketches.

Whereas planning (in a software engineering sense) and documentation are not major aspects of Tribeflame's work, evaluation is prominent. This is often a significant feature of meetings at Tribeflame, as indicated in Figure 5.3 (page 144). In addition, distinct evaluation sessions sometimes take place, shown in Figure 5.12. Thus, of the three major concerns of software engineering, evaluating seems to be the concept that is most useful for describing what is going on during the process observed, but it only applies to a part of Tribeflame's work.

As we saw in Figure 5.10 (page 150), playing other companies' games during work hours is a frequently-occurring activity. It is hard to characterize this important activity within the conceptual framework of software engineering, as it clearly does not fit into any of the categories of architecture, design, coding, or testing. It could perhaps be perceived as a kind of research necessary for requirements specification. However, specification is supposed to be as complete as possible before starting the design work, and this does not fit well with the ongoing activity of game playing.

Some might say that it is unfair to try to analyze Tribeflame's process with software engineering terms, as software engineering is specifically oriented toward large projects, and not small teams. Software engineering theory is useful in the right circumstances. In chapter 6 (Safety critical programming) we shall see examples of software engineering theory being put to good use in practice. Nevertheless, as we have seen above, traditional software engineering theory is inadequate for explaining this example of a small team development process. Since 90 percent of Finnish software companies are of a similar small size, this is a real challenge to the explanatory power of software engineering.[12]

From a research perspective, it is not necessarily a problem that software engineering theory might only be applicable to a minority of companies within the software industry.[13] From a programming perspective, however, this can have serious consequences. Most programmers learn about software engineering during their education, but not so much about the limitations of software engineering. When presented with a real software process, many will unconsciously perceive it in terms of the software engineering theories they have learned, even when the process is not suited to software engineering analysis, as is the case in Tribeflame's example. Not being aware of the limitations of software engineering theory then has the unfortunate effect of making the programmer apply the wrong mental tool to his task.

5.2.1.2 *Agile development*

The diversity of Agile approaches makes it rather challenging to use Agile concepts for analyzing a concrete process, for it is not given which of the Agile approaches should be used. It is immediately clear, however, that on a practical level, Tribeflame's process corresponds only sporadically to Agile concepts.

To take one example: Tribeflame's work corresponds to the Extreme Programming practice of placing all developers together in the same room, simply because Tribeflame has only one office room available. But there is no correspondence to the practice that all program code should be tested every day, as Tribeflame does not have an auto-

[12]In 2010, 90% of Finnish software companies had 9 people or fewer. Rönkkö & Peltonen 2012. See note 1 on page 140.

[13]Though it is of course a serious problem if the limitations of the theory are not clearly understood and presented.

mated testing system in place, much less test cases for all of the code. This, of course, stems from the nature of their product – it is nearly impossible to design test cases that determine whether an interactive computer game is "correct". Likewise, the practice of programmers coding together in pairs is not followed, as Tribeflame's developers spend much more time working alone than in pairs. There are more practices in Extreme Programming, but it is clear that Tribeflame's process does not resemble Extreme Programming to a significant extent.[14]

In another example, we can look at how well Tribeflame's work corresponds to the Scrum rules. The three important roles in Scrum are the Product Owner, the ScrumMaster, and the Team. Tribeflame has neither Product Owner nor ScrumMaster. The Team, understood as everyone working in Tribeflame, is of course very important in Tribeflame; but it is not exactly the same as the Scrum conception of a Team. In Scrum, the Team has to be without internal hierarchy; in Tribeflame, Björn and Andreas, as the owners, have the formal decision making authority, even if they seldom exercise their authority directly.

Those working at Tribeflame do not use the Scrum version of iterations, Sprints; they seldom explicitly prioritize their tasks, and they do not commit to a specific workload each month. Sprint reviews do not have an equivalent either, as there is rather little evaluation of the work process itself. Only once during the observation period did Tribeflame have a stand-up meeting that could be interpreted as something akin to the daily Scrum meeting.

Whereas the concrete rules and practices of Agile development do not correspond to Tribeflame's process very well, the more abstract concepts of Agile fit much better. As an example, let us compare the process to the Scrum values: transparency, inspection and adaption.

Scrum demands that the process is so completely transparent that it is visible at all times who is working on what, and that this is repeated verbally every day at the Scrum meeting. Tribeflame's process is not quite as transparent as that. However, since the developers sit so

[14]There are at least 12 Extreme Programming practices.

close to each other and interact so frequently, they are generally well informed about what the others are doing, and they do not hesitate to ask each other what they are working on. Perhaps more importantly, the game they are developing is tested and played constantly, so that everyone knows the current state of the product. This creates a high level of transparency, since there are only a few aspects of the product whose state cannot be assessed by playing the game, namely integration with the distribution channel (Apple's Game Center), and cross-platform support.

Inspection is closely linked to transparency. Inspection in Tribeflame is also performed by playing the game, both by the developers themselves and by external persons. In addition, there are evaluations (Figure 5.12) and frequent discussions (Figure 5.5, page 145) which often include aspects of inspection. That the company instituted daily meetings in the middle of the observation period gives evidence that inspection of the work process itself also happens.

Adaption is very prominent in the work process. There are no detailed, long-term plans; rather, the planning is done as the game is progressing from week to week. Features are constantly taken up for discussion at meetings, and whenever a feature in the game is complete it is promptly evaluated in discussions and has an effect on further development. As mentioned above, daily meetings are added halfway through the observation; this shows that adaption is present at the process level, as well as the product level.

Thus, the abstract values of Scrum correspond well to Tribeflame's process. The common values of Agile development, as expressed in the Agile manifesto, correspond partly to the process. We recall the Agile values, shown in the manifesto, Figure 3.8 (page 65):

1. *Individuals and interactions* over processes and tools.

2. *Working software* over comprehensive documentation.

3. *Customer collaboration* over contract negotiation.

4. *Responding to change* over following a plan.

Regarding the first value, at Tribeflame, individuals and interactions are clearly much more important than processes and tools, as processes are seldom referred to, and tools are only referred to when necessary. Regarding the second value, Tribeflame's product is always in the form of working software, and the company produces very little documentation in general, and no documentation that can be called comprehensive. Regarding the fourth value, the decision process at Tribeflame is dynamic and adaptive rather than planned in advance.

The third value is not applicable to Tribeflame. The company produces for a mass market of anonymous players, so they do not collaborate with customers or with representatives of customers in the sense that is meant in Agile development. Nor do they enter into contracts or negotiate them with customers: they have investors, but their role is not quite the same as that of a customer. This value therefore seems to reflect an aspect of Agile development that is absent in Tribeflame.

As we can see, many, but not all, of the more abstract values of Agile development correspond well to Tribeflame's process. With one exception, the Agile concepts are well suited, in contrast to the concepts used by software engineering which, as we have seen, are in fundamental conflict with Tribeflame's way of working.

However, though Agile concepts are generally compatible with Tribeflame's process when used descriptively, they nonetheless have serious deficiencies as analytical concepts. The reason for this is that even though the Agile concepts are based on a philosophical ethical system, as described in section 3.2.2.5 (Work ethics), the concepts themselves are primarily part of a prescriptive system of software development. Agile methodologies are possible approaches to productive software development, but they are not meant to or well suited to analyze processes that are not Agile and are not intended to become Agile.

Thus, the Agile concepts offer us no help in explaining the differences we see between Agile processes and Tribeflame's process. For example, Agile concepts cannot explain why the playing of other companies' games is so important to Tribeflame. Agile thinking certainly leaves room for this activity, but it offers no guidance in determining what its function is.

Another example is the absence of a customer or customer representative in Tribeflame's process. The customer (represented in Scrum by the Product Owner) is such a central part of Agile thinking that the fact that Tribeflame's process lacks it is something of a mystery from an Agile perspective.

5.2.1.3 *Computer science*

As we saw in section 3.2.3 (Computer science), mainstream computer science perceives itself as a mix of applied mathematics and formal methods. During the observation period at Tribeflame, mathematics was applied in only two cases, both of which had to do with how the graphics should be moved around on the screen in the game, and used mathematics of a high school level. There were no cases where formal methods were applied, or could appropriately have been applied.

A common view of programming within computer science is that a program consists of algorithms and data structures. The task of the programmer is to divide the program into modules, then use the best algorithms he can find. Tribeflame, like any other software company, needs to select algorithms for its programs. However, it is not a top priority that the algorithms are the best possible – they merely have to be good enough.

As an example: before developing one of their games, the developers were worried whether the main algorithm was too inefficient to run on an iPad. They quickly developed a simple prototype of the algorithm and let it run with a few hundred objects, far more than would typically used in a real computer game. When this test proved successful, they proceeded with development without caring further about the efficiency of the algorithm. No mathematical analysis or advanced methods were employed apart from this simple but effective practical test.

Another common computer science view of programs is to regard them as computable functions. This means that there is some input and a mathematical rule that transforms it into output. This view does not correspond very well to the practice of Tribeflame either, for the reason that it is unknown what form the input and output should take for a computer game. If those working at Tribeflame spent their time figuring out how a game could be represented mathematically, they would not have time actually to make the game.

As we have seen, in computer science there is a pronounced orientation towards the machine, and not towards the uses to which it is put. This perspective is not consistent with the practice of Tribeflame. On one hand, the machine is central to the work, because without it there could be no computer games at all, and the limitations of what the people can do are dictated by the limitations of the machine. On the other hand, though the limitations of the machine are certainly present in everything they do, they are seldom discussed or addressed directly. The discourse at Tribeflame normally focuses not on the machine itself, but on the experience of using the machine. This is not a technical but a human discourse and it takes place mostly in terms of human imagination and emotions.

If we look at the computer science perception of the work process involved in programming, we see that the correspondence with Tribeflame's process is mixed. There is within computer science an emphasis on mathematical intellectual elegance in the work process. Since mathematics plays no big role in Tribeflame's work, this emphasis is obviously lacking in their process. Computer science also emphasizes the individual in the process, almost to the point of becoming personal. Neither does this emphasis resonate with Tribeflame's process, which is intensively social and cooperative.

On the other hand, computer science emphasises that the details of a problem only becomes understood through actual programming. This corresponds well to Tribeflame's process, which is essentially a continuous cycle of discussing ideas and then trying them out in practice: that is, programming them. There is also Naur's view, that programming is essentially a process of acheiving a certain kind of insight. This corresponds well with the ongoing knowledge building in Tribeflame, which we will examine in more detail in the next section.

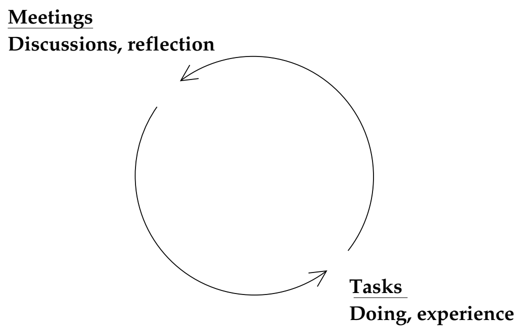


Figure 5.13: The hermeneutical circle depicts the relationship between tasks and meetings. Tasks are usually carried out alone and result in experience. In meetings the experience from the tasks is discussed and reflected upon. Meetings and tasks are mutually dependent.

5.2.2 *Hermeneutical analysis*

5.2.2.1 *The work process*

When we looked at Tribeflame's process we noticed the rhythm inherent in the work. There is a continuous rhythmical shift between working at the individual desks in the periphery of the room, and coming together at the central table to have discussions. At the same time, the rhythmical shift is between working with concentration and alone, on one hand, and on the other hand talking, bantering, joking, and interacting with others. The work that is mostly done alone at individual desks is specific tasks. Through the tasks the game is slowly built. Each task amounts to doing something, trying something out, and getting experience when it is seen if and how it works. In the meetings the experience is discussed and reflected upon. The meetings provide afterthought, which again leads to ideas for new tasks that can be tried out.

The relationship between meetings and tasks is shown schematically in Figure 5.13 as a circle that leads from meetings to tasks, and back again from tasks to meetings. Meetings and tasks are dependent on each other. The meetings generate the ideas and the plans for tasks that should be carried out. The tasks provide experience of what works and what does not, and serve as input to the meetings to make sure that the discussions are grounded in reality. It would be extremely inefficient to discuss and plan the whole game in detail before attempting to carry it out in practice; likewise, it would be unwise to make a whole game without stopping along the way to reflect on the progress.

The sort of mutual dependence between two parts of a process that is shown in Figure 5.13 is very common in hermeneutics – so common that is has been given a name: the hermeneutical circle.[15] The hermeneutical circle expresses that experience and reflection cannot be separated. Neither can be subordinate to the other, and they need to happen concurrently.

Because the hermeneutical circle is such a common phenomenon, it is often also recognized outside of hermeneutic theory, under another name. Donald Schön has analyzed the practice of a range of modern professions: engineers, architects, managers, etc. Schön writes that the professional oscillates between involvement and detachment, from a tentative adoption of strategy to eventual commitment.[16] "The unique and uncertain situation comes to be understood through the attempt to change it, and changed through the attempt to understand it."[17] We recognize here the structure of the hermeneutical circle as it is observed in Tribeflame.

In theory, this rhythmical, circular process goes on indefinitely. In practice, the game is deemed finished at some point. The meetings and tasks do not stay the same but lead to an ever-better understanding of the game being developed. Schematically, we can picture the relationship between tasks and meetings in the hermeneutical circle

[15]The hermeneutical circle depicted here shows the relationship between experience and reflection. The hermeneutical circle also appears between other concepts. The most common use of the hermeneutical circle is to describe the relationship between the part and the whole in a process of understanding: see section 3.3.5 (Understanding).

[16]Schön 1983 p. 102.

[17]Ibid. p. 132.

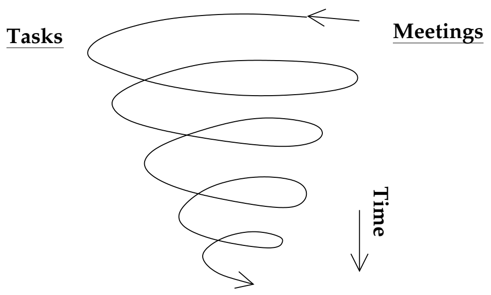


Figure 5.14: Over time, the hermeneutical circle between tasks and meetings results in better understanding. Whereas the process is very open in the beginning and decisions are relatively easy to undo, with time it becomes more and more tight and resistant to radical change.

as time goes by as the spiral shown in Figure 5.14. The hermeneutical circle is still seen shifting from tasks to meetings and back again, but we also see that the process becomes more focused with time.

Initially, the process is very open. The available alternatives are big and visionary, and the chosen decisions are not clung to with great commitment. But as time passes and understanding deepens, decisions made in the past become increasingly costly to revise because time and resources have already been committed to them. Decisions are still up for discussion – the process is not rigid – but it is no longer as easy to undo them. The process has become more tight, and focused. Schön recognises the same gradual tightening of the professional process. He writes that the professional can always break open his chosen view, but that "This becomes more difficult to do as the process continues. His choices become more committing; his moves, more nearly irreversible."[18]

[18]Ibid. p. 165.

5.2.2.2 *Horizons of understanding*

Tribeflame's process is not simply a matter of planning, of optimizing, or of construction. It is primarily a process of understanding. At the outset, the developers do not fully understand the myriad details that together make up a successful game. As the work progresses, they slowly and gradually come to a fuller understanding of the central question of their work: what makes a game fun, and specifically, what makes the game we are working on right now fun?

The process of understanding does not start from scratch. Tribeflame's team is composed of individuals with different and highly specialized competences. Matti and Kati are competent in the area of graphic art; Andreas, Mickie, and Fredrik in the area of programming. Björn is competent in the areas of business economy and programming. They all have a general understanding and personal experience of computer games. All this foundational knowledge and competence forms the pre-understanding that is the basis of the process of understanding. If Tribeflame's developers did not have this pre-understanding, they would first have to acquire it before the process could begin of understanding the game they are developing.

The understanding process is a social process that takes place in meetings, discussions, and chat between the developers. As we have seen in Figure 5.8 (page 148), "information sharing" is quite rare in the process, where information sharing means that some information is simply delivered from one person to another. The process of understanding is much more interactive. Nor is the process primarily a decision process. Decisions are usually the outcome of discussions, but the discussions contain much more than the information and deliberations needed to make decisions. Sometimes, the decisions that are the result of a discussion are even left more or less unstated. As soon as the developers reach a common understanding, the consequences often become so clear to them that they do not really need explicit decisions; everyone agrees on the common direction.

So how can we characterize the majority of the content of the discussions if it is neither information sharing nor explicit decision making? The developers seek to expand their understanding in the process. At the onset of a discussion, they have a certain knowledge and range of ideas that they are prepared to understand: their horizon of understanding. Through discussions they exchange facts, opinions, and viewpoints, and in that way expand their horizon. The process of understanding is a process of horizon fusion. When the developers' horizons of understanding fuse, their understanding becomes larger – not by streamlining everyone's opinion, but by expanding each individual's horizon.

5.2.2.3 *Authority and tradition*

Though the process of understanding is complex and difficult to describe in simple categories such as for example the software engineering phases, this does not mean that it is unstructured or haphazard. The process is strongly guided by authority in various forms. The final authority of decisions rests with the owners, Björn and Andreas. They share the responsibility for business decisions, while Björn has the overall product responsibility, and Andreas has the overall technical responsibility. The business decisions rarely impact upon the development process directly, and in this area they exercise their authority freely. Regarding the development, they rarely exercise their authority directly, as we saw in section 5.1 (Case study), with Björn's reluctance to impose a decision outside of the consensus process. It does happen, however. For example, Andreas has insisted that the programmers use shared pointers, a form of memory management.[19]

In their daily operation and management of business decisions, there is hardly any conflict between the two. The basis of the unproblematic sharing of responsibility and authority between them is their good relationship – the personal reverence they have toward each other. The authority they have over their employees is based partly on the formal economic relationship between employer and employee, and partly on the personal reverence the employees have for Björn and Andreas in Tribeflame's small, tight-knit team.

[19]The class shared_ptr of the Interprocess library from the Boost collection of libraries for C++.

Since Tribeflame's process is primarily a consensus process, the dominating form of authority in discussions is not the authority of Björn and Andreas as employers. The work is a teamwork of understanding. Each individual has a specialized role and authority within his field of competence, but the work is not just a collaboration between experts where each has the final say in his or her area: rather, each individual can influence every area of the game and contributes to the overall direction of the development. The primary form of authority between the developers comes from the ability to argue convincingly.

Convincing arguments also rest on authority and, as mentioned above, not merely the authority of expertise. The most frequently invoked authority in Tribeflame's discussions is that of other games. All the developers have a notion of what a good game should be like, as well as examples of games that they consider successful. An argument about how their own game should be is very often supported by a reference to another game that contains the feature in question, and thus the other game lends its success to the argument.

A few games carry so much authority that a reference to them can almost settle an argument in itself. The most authoritative game is *Angry Birds*, the most successful tablet game in the world, which is also made by a Finnish company. Another game that is often mentioned is *Cut the Rope*, perhaps because its gameplay is somewhat similar to Tribeflame's own game. Older games are also mentioned occasionally, for example *The Incredible Machine* from the 1990s.

The collected knowledge of the developers about how a game is supposed to be forms a tradition with which they became acquainted before they became involved with Tribeflame, and which acquaintance they continue to develop. Tribeflame's games are built upon, and are a continuation of, this tradition of computer games.

The tradition of computer games is an important source of authority and this is one reason why the Tribeflame developers spend so much time playing games and talking about them. Another is that the computer game tradition provides the yardstick with which the success of Tribeflame's games is measured. As long as the developers do not have hard data on their game's success in the form of sales figures or number of downloads, they have to compare their half-finished game to the existing tradition. For this reason, the fusion of horizons of understanding that goes on in the hermeneutical circle is not only a fusion of the horizons of the individual developers on the team; it is at the same time a fusion of the developers' horizons and the horizon of tradition.

Thus there are at least two levels of fusion going on in the hermeneutical process of Tribeflame: a fusion of the different expertises that is needed to make a game, and a fusion with the tradition that sets the standard for what a good game is. With this notion, we can explain some aspects of Tribeflame's process that were difficult to explain satisfactorily with mainstream theories of programming. Tribeflame's developers spend a significant amount of time playing games made by other companies (see Figure 5.10, page 150), which has the effect both of providing them with arguments and of making them more knowledgeable of the computer game tradition.

They also spend a significant amount of time playing their own game (see Figure 5.9, page 149). This has not only the function of finding errors in the programming and graphics, but also of giving the developers a sense of whether their game is fun to play: that is, whether they are reaching their goal. Playing their own game is not only a technical activity that corrects the code: it is as much or more a way for the developers to get a sense of the practical application of their game. They are trying to put themselves in the place of the eventual player, and experience the game as a player would.

To experience the game as another person would requires that the developer can suspend his horizon of understanding temporarily, or else his own knowledge will prevent him from understanding the other person's experience. For example, the developer must try to forget his knowledge of how a certain puzzle is meant to be solved, and approach the puzzle as if he saw the game for the first time, as an eventual player. This is a difficult task, and to make it easier the developers also have other people from outside the company play their game (see Figure 5.9).

The developers experience their game on the basis of prejudice. In hermeneutic theory, prejudice does not have the negative association that it does in everyday language. Rather, prejudice is a requirement for understanding – understanding simply cannot happen without prejudice, whether the prejudice is consciously known or not. Prejudice is simply the fundamental assumptions humans make in order to make sense of things.

However, prejudice can be either true or false. True prejudice will make understanding easier, while false prejudice will hinder understanding. It is a central part of the process of understanding to examine prejudice and correct it. The developers do this by playing their game themselves, but correcting one's own prejudice is difficult precisely because it is the basis of understanding. This is an area where the developers cannot trust their own competence completely.

Therefore it is of great importance to them to have external people play their game. It provides them with an interpretation of the game that they have not made themselves, and therefore they can use it to correct their prejudice and thus move along in the process of understanding.

5.2.2.4 *To have fun*

All of the activities at Tribeflame, and the process of understanding that is analyzed above, is directed by the application of their product: to play the game and have fun. From the use of shared pointers in the code to the choice of colour scheme for the graphics, and even to business decisions such as an eventual merger with another company – every decision has to be evaluated in the light of the overall goal of creating a game that is fun to play. The developers are conscious of this when they say that the code is just a means to reach a goal.[20] This has an important consequence for programming research, in that it does not make sense to study programming in isolation from the context in which it appears. Computer game code cannot be understood without some knowledge of computer games.

[20]See note 9, page 151.

The application, in the form of "having fun", is what drives the activities, but the application in itself is not quite enough to explain the process of understanding fully. After all, if "fun" were completely defined by an existing computer game, all Tribeflame would have to do was to copy that game. All understanding is in essence driven by a question, and for Tribeflame the question is what "fun" actually means. The beginning of Tribeflame's answer is that "fun" means entertainment.[21] This might seem self-evident, but it is not. Consider, for example, that for Björn and Andreas themselves, "fun" means to work, among other things.[22]

Thus, in essence, Tribeflame's business is a process of trying to answer the question of what it means to have fun, for the kind of person that would be inclined to play its game; and all of the company's activities are somehow related to this question. Of course, those working at Tribeflame have to do many activities that do not directly contribute to answering this question, such as administration of the payroll. But the supporting activities are only necessary for success, not sufficient. They are not the critical factor that decide whether Triblame will succeed or not in competition with other competent companies.

The question of what is "fun" is interesting and it would be quite instructive to analyze more in depth both what Tribeflame's developers' concept of fun is and how they arrive at it. Unfortunately, such an analysis lies beyond the scope of this dissertation. While the concept of fun is central to this kind of programming it is not relevant to other kinds of programming practice, as we shall see in chapter 6. Thus, in the compound concept "game programming" fun is intrinsic to games but not to programming, the subject of the dissertation. The reader that is interested in further understanding fun and computer games is advised to consult the works of Turkle and of Crawford.[23]

In the analysis of Tribeflame with hermeneutical theory, we have seen the use of all the essential hermeneutical concepts apart from

[21]See note 10, page 151.

[22]Andreas. "is it already time for lunch?" Björn. "yes, time passes quickly when you're having fun." (Andreas. "er det allerede tid til lunch?" Björn. "ja tiden går fort når man har det roligt.") Field diary, Monday 5th September 2011, 11:47.

[23]Turkle 1984. Crawford 2002.

one: the concept of effective history. Effective history means to be conscious of the role of the knowledge one produces in a historical perspective. During the observation period, Andreas touches upon this question a single time when he muses on the fairness of charging money for computer games.[24] His comment brings to mind the question of the moral justification for the game, and of the place of their product – entertainment – in the grander scheme of things. But his is only an offhand comment; an explicit consciousness of effective history is absent in Tribeflame's process.

5.2.2.5 *Summary*

The process of Tribeflame can be explained as a hermeneutical process, which is shown in schematic form in Figure 5.15. The alternating activities of doing tasks and discussing them makes up a hermeneutical circle in which understanding is built through repetition. The increase in understanding takes the form of a fusion of horizons of understanding, both between the individual developers, and between the developers as a team and the outside world: their customers and competitors.

The hermeneutical process rests on a foundation of pre-understanding and prejudice. The pre-understanding is made up of the developers' preexisting skills and competences. The prejudice is made up of their ideas of what a computer game is supposed to be like, and what it means for it to be fun.

The process is guided by authority and tradition. Authority draws from various sources; both from the expertise of the developers and from the tradition of computer games. The tradition is not an ancient and static tradition; rather it is living and constantly evolving. Thus, the newest games on the market are also part of tradition.

The process is directed by the goal of the computer game, in other words by its application. This is the reason for the process, and as such it influences all parts of the process. The reason, entertainment,

[24]See note 11, page 151.

Hermeneutical concept	**Results**
Prejudice	The developers' ideas of what the game players will like. Corrected by external playtests.
Authority	Business decision authority rests with the owners. Other authority comes from convincing arguments, drawing from expertise and references to exemplary games.
Received tradition	An informal knowledge of computer games from life experience. Rests on a collection of authorative games.
Personal reverence	The developers respect each others' competence and personal authority.
Pre-understanding	Skills and expertise in programming and graphic design.
Understanding	A gradual process that slowly deepens over time, and that involves the whole company.
Hermeneutical circle	Manifest in the mutual dependency between tasks that bring experience, and meetings that reflect upon the experience.
Effective history	Largely absent.
Question	The question of what "fun" acually means. All activities relate to this.
Horizon of understanding	Expanded during discussions in the company. Developers' horizons fuse with each other, and with computer game tradition.
Application	To make a game that is fun to play, in order to provide entertainment, so that the company can stay in business.

Figure 5.15: A schematic summary of the principal results of analyzing Tribeflame's process with hermeneutical theory.

is an encompassing perspective that must be kept in mind in order to understand the process and the product correctly.

Chapter 6

Safety critical programming

6.1 *Introduction to safety critical development*

In chapter 5.2 (Analysis) we examined the working process of a small computer game company. Using concepts from hermeneutical theory, we found that the programming work in this case was easily understood as being a process of understanding. By contrast, the mainstream theories – software engineering, Agile development, and computer science – proved to be less useful. The conclusion drawn by the analysis was that the mainstream theories are not well suited as general theories of programming.

In this chapter we will take a look at some programming processes where *software engineering theory* is prominent and used in practice: namely, safety critical programming. We will do this for two reasons. First, investigating programming practices that take place in very different circumstances to a small game company will deepen our understanding of how programming works.

The second reason is that, from a cultural perspective, it is not enough to criticize the mainstream theories on the basis of one example. We must also be able to explain what the theories are actually useful for and why they look the way they do. In this respect, the software engineering theories are the most interesting examine. As explained in sections 3.2.2 (Agile software development) and 5.2.1.2 (Agile development), Agile development is very close to ordinary programming practice, and computer science theory is not primarily concerned with work practices. Software engineering is oriented toward work processes and, at the same time, quite different from the way programmers organize themselves if left to their own devices.

The term "safety critical" applies to products and areas in which malfunctions can result in people being injured or killed, or in which errors are extremely expensive.[1] Safety critical products are products such as cars, aeroplanes, and hospital equipment; and safety critical industries include nuclear power plants and chemical processing plants. The software involved in safety critical applications is often control software for machines or products, such as the software that controls the brakes in a modern car, or the software that controls the valves in an industrial plant. This kind of software is often embedded software, which is situated in the machine or product itself during execution.

The safety critical area is regulated by a large number of standards, depending on the particular industry or geographical area. The most important standard in Europe currently is called IEC 61508 "Functional safety of electrical/electronic/programmable electronic safety-related systems".[2] The IEC 61508 is a general standard, with a number of derived standards that apply to specific industries. The standards are often backed by legal requirements in national courts. On other occasions, the only requirements that apply are the ones that industry associations agree upon among themselves. In any case, the standards are most often verified by separate verification agencies,

[1]These last are also called "mission critical" areas.

[2]International Electrotechnical Commission 2009.

such as the German TÜV, or the German-American Exida, which perform the verifications for a fee for their industry customers.

6.2 *Case studies*

6.2.1 *A large avionics company*

Before we discuss safety critical programming processes in general, we will take a look at a couple of specific examples of processes. First is the way work is done at a software department in the company Cassidian. The following description is based on an interview with two engineers who work there and thus respresent the employees' own opinion of their work.[3] The company makes components for military aircraft such as the Airbus A400M. Cassidian is a part of the holding company EADS, which is an international European aerospace and defence company with around 110,000 employees – more than a small city in size. Cassidian itself has around 31,000 employees. The most well known EADS products are the Airbus aircraft.

A typical software department in Cassidian will be a sub-department of an equipment level department, a department that works on coordinating the efforts of creating some piece of equipment for the aircraft. The equipment level department might have, for example, around 30 system engineers and four sub-departments. Two of the sub-departments are hardware departments, with around 25 engineers each. The two others are software departments with around 20 software engineers each – one department for resident software (embedded software) and one for application software.

Besides the software departments that do the actual software work, and the equipment level department that gives the software department its requirements, there are some more departments involved in the work. The quality department contains some sub-departments

[3]Interview 24.

that observe the software development process and modify it based on feedback from lower-level departments. The safety department is involved in making sure that the work fulfils the safety standards. Both the quality department and the safety department are independent departments within Cassidian. The quality department is independent because this is required by the safety standards; the safety department is not required to be independent, but it has been found very helpful to have it that way.

The general working process at Cassidian is an engineering process. The first phase is requirements engineering, in which it is decided what the piece of equipment is going to do. After that follows conceptual design and detailed design. Then comes implementation – the actual making of the equipment – followed by the verification phase, which checks whether the equipment actually fulfils the requirements from the first phase.

The work process is standardized internally in Cassidian, but it is tailored to each project. Before a project is started, a project team gets together and defines the tailoring of the process that will be used. This team consists of high-ranking managers that represent different interests: the project's financial officer, the quality manager, the configuration manager, the project responsible for engineering, and people representing logistics support, tests, and production. Sometimes even the head of Cassidian is involved in the project team's decisions. The reason that so many important people are involved is that the projects are big and costly and therefore carry a lot of responsibility, with budgets of between 5 and 10 millions Euros. In addition to determining the process tailoring of a project, the project team also carries out risk assessment, risk analysis, feasibility studies, assesses financial feasibility, and predicts marketing consequences.

When the project has been approved by the project team the actual engineering process can begin. The higher level system requirements are broken down to unit level requirements, and separated into basic hardware and software requirements. This makes up the equipment architecture. After the architecture has been defined the unit activities start. Each unit prepares its own requirements paperwork and works out a unit architecture. After this the unit requirements are validated, if it is required in that particular project. This is followed by the actual implementation of the unit, after which comes informal testing to weed out errors. Once the unit is ready, the various hardware and software units are integrated and formally tested. When all units have been integrated and tested, the final product is complete.

When, in the engineering process, the system level architecture has been defined, it has at the same time been decided which parts of the system are to be created in hardware and which in software. The software departments can then start their work, which follows the software development process. This process starts with writing a software requirements specification, which document is then reviewed by the system department, the hardware department, and members of the software department. The review includes validation of the software requirements to make sure that they are in accordance with the system requirements and hardware requirements. Sometimes, if the piece of equipment is meant to stand alone, the validation is omitted to save on the workload and cost, but for systems that go into aircraft the full program is always required.

After the software requirements specification has been written and reviewed, the conceptual design phase takes place. In this phase, the designers tend to arrive at a conceptual design they think will work, and try it out in some way. The feedback they get from this reveals problems in the conceptual design, which can then be corrected. The correction happens in the form of derived requirements, which the designers impose in addition to the system requirements from the requirements specification phase. The system department then verifies the derived requirements, because the derived requirements are not a part of the high-level system requirements that have been agreed upon.

After the conceptual design comes the detailed design phase, which consists mainly of generating the C code of the programs, compiling it to machine code, and finally implementation. But here, implementation does not mean coding, as is usual in programming. Rather, it means transfering the programs to run on a piece of hardware, which will often be some kind of test hardware, and then conducting some trials and tests on the newly-transferred software. After the detailed design phase comes integration of software and hardware, this time not with test hardware but with the hardware that has been developed by the hardware departments at the same time as the software departments wrote the programs. Finally, there is verification of the whole integrated system.

Because of the derived requirements, there is close cooperation throughout the development between the software, hardware, and systems departments. The derived requirements are always assessed by the independent safety department.

Equipment units can be qualified and certified according to standards. All units are qualified, but not all are certified. Each software or hardware unit does its own verification and qualification, meaning that it prepares its own qualification specification and qualification procedures. For a unit to be certified, it is necessary to convince a body of authority outside Cassidian that the unit and the development procedures fulfil the relevant standard. In order to do that, it is necessary to record evidence of each step in the development process so that it can be shown later that the step has been carried out correctly. This can, for example, be evidence connected with phase transition criteria, such as the outcome of a phase; it can also be evidence that certain rules have beenfollowed, for example that certain forbidden structures are not found in the C code. The verification of requirements is a part of the evidence gathering process. There are three main ways of showing that requirements are covered: tests, analysis, and reviews. Tests provide the strongest evidence, but it can be difficult to get to cover a high enough percentage of the requirements with tests.

A number of software tools are used in Cassidian to help with the development process. IBM Rational DOORS is a requirements management program that is used to keep track of the large number of requirements in a project. The derived requirements of a subsystem need to be linked to the higher level system requirements from which they come. All the requirements also need to be linked to the tests that verify that they are satisfied, and all links need to be updated at all times. Aside from the requirements management software, a code checker is run regularly on the C code to check that it conforms to the style guides. This is required by the standards.

At Cassidian, experienced colleagues act as coaches for new employees and less experienced colleagues. The coaches help their colleagues with using the software tools, and with following the work processes. In addition, there are internal training programs where necessary; for example, if an engineer is promoted to systems engineer he will participate in a training program explaining his new responsibilities. Regardless, the day-to-day work is the most important way of educating the engineers, as at least 80 percent of the understanding needed to work with the avionics standards depends on everyday experience.[4]

The work processes at Cassidian are defined in the internal company standard, FlyXT. This standard is a synthesis of a number of other standards. On the system level, it originally builds on the V-model, a government model that was taken over from the military administration department Bundesamt für Wehrtechnik und Beschaffung. FlyXT completely covers two avionics standards: DO178B and DO254. DO178B applies only to the software parts of aircraft and DO254 applies only to the hardware. Together, they are the standards that are most usually applied to aircraft production. The software development process at Cassidian follows the DO178B standard as it is expressed in the FlyXT standard. At the beginning of each project, the process steps are then copied to the project's handbook or referenced from it, according to the wish of the project leader.

After each project is completed, an effort is made to record the lessons learned and incorporate them in the company standard. However, it is a big company, and whether this actually happens depends on the workload and on the people involved in the process. The top level management wants to make the company as a whole comply to the process maturity model CMMI level 3. A process maturity model is a standard that is not particular to safety critical develop-

[4]See quote on page 191.

ment: it sets some requirements to a company's paperwork processes, especially that they be well documented and consistent. CMMI compliance is an attempt by management to record systematically and standardize all processes in the company. The goal of this is to get rid of all tailoring of the process, so that the projects always follow the company standard without modifications. This goal is linked to a wish in Cassidian to make FlyXT the only standard in use by the company. Since Cassidian works as a subcontractor, the customers sometimes impose their own choice of standard on the work process. The ambition is to be able to say that FlyXT covers every avionics standard so that there is never any need to work with other standards. As part of this goal, the company strives to have reproducible process steps for all projects.

6.2.2 *A small farming systems company*

The process at Cassidian is in many ways typical of how software is developed in large companies in safety critical industries. However, not all safety critical software is developed by large companies or in typical settings. For a different example, we will take a look at how work is done in a small, Danish company named Skov. The description is based on an interview with the engineer who is in charge of software methods and tools.[5]

Skov makes automated equipment for animal production, primarily for swine and chicken. One area of business is ventilation and heating systems for animal stables. Another is production control systems, for controlling feeding, water dispensation, and lights. Minor areas of business include air purification systems and sensors for use in animal stables.

Skov has around 300 employees, of which 45 are employed in the development department. Of these, around 20 are software develop-

[5]Interview 20.

ers. Ten software developers work with ventilation systems, 10 with production control systems, and one with sensors. The majority of the software made is control software for the systems that Skov sells, meaning that it is embedded software.

Skov considers its software development process to be nothing out of the ordinary compared to other small Danish companies. Most of the process is very informal, and it is not written down. The developers follow an Agile process where the work is planned in relatively short periods of a month or so, called Sprints.[6] Much of the planning is also done from day to day. Every morning there is a short standing Scrum meeting where the day's work is discussed, and before each project starts, developers, project managers and department managers have an informal discussion about the course of the project. The Agile process was adopted in the company three years ago after a trial Agile project.

As a part of the Agile process the developers make an effort to include the customer throughout the development process. The software customers are typically internal Skov customers, though occasionally there is an external customer. The close collaboration with customers during software development is something the company has always practiced. The basis for the software development is a written requirement specification, and occasionally a design specification is written too, though neither document is at all formalized.

The software for Skov's controllers is coded in UML, a graphical notation for software models. The models are then automatically translated to C code, which can run on the hardware. The company uses the commercial software tool Rhapsody for translating the UML models: this is used with almost no custom modification. The UML-based way of working is well suited to Skov's products, which from a software perspective consist of a few main products that are sold in many slightly different variants under different trade names.

Because of the safety critical nature of Skov's products, the testing process is very well developed. The tests are fully automated and take place in Skov's test center, where the software is tested against its

[6]See section 3.2.2.3 (Scrum as an example).

specifications and in a variety of application situations. The testing procedures are thorough and formalized. In addition to ordinary tests, the company also performs so-called service tests, where customers test the company's products; and there are stable tests, where the products are tested in real environments.

The safety critical aspects of Skov's systems are mostly connected to ventilation. If the ventilation fails in a modern animal stable, the animals can suffocate in a few minutes. A stable filled with hundreds or thousands of dead animals would be catastrophic and reflect very poorly on the company's reputation.

There are no formal requirements to certification of processes or products in the animal production industry. Likewise, there are no standards within the industry. Skov is very careful with its testing, not because of any external requirements, but in order to avoid bad publicity and a reputation for unsafe products. It strives to attain a high degree of safety while at the same time, their people "do not produce documentation for the sake of documentation".[7]

Skov considers itself to be a developer of total systems, not of components, and that identity also influences the perception of software development in the company. From the company's point of view it is an advantage that there are no formal standards in the industry, because it makes it possible to come up with innovative solutions and products without risking conflict with existing standards. The only standards that exist are the traditions in the markets. Each market has its own distinct cultural tradition, and there is a big difference between what the United States' markets expect of a product and what the European markets expect.

Besides their own Agile software development process, Skov has an ambition to conform to the process maturity model CMMI level 2 (level 2 is a lower stage of compliance than level 3). This ambition held by Skov's management and, as such, the introduction of CMMI has been a top-down process. By contrast, the Agile development method has been introduced as a bottom-up process, in that it

[7]" . . . altså vi producerer ikke dokumenter for dokumenters skyld." Interview 20 ∼00:15:39.

was the software developers themselves that desired and introduced the method with the consent of management. CMMI and the Agile method have never been in conflict in Skov: they appear to coexist peacefully within the company without much conscious effort.

The introduction of Agile methods did not in any way revolutionize the development process at Skov. When the company became curious with regard to Agile method and decided to try it out, they discovered that they were already very agile in their approach, so the introduction did not add or change a whole lot of things. In Skov's opinion, constant testing is part of what makes a process Agile. This corresponds well with Skov's process, which emphasizes the need for automatic, reproducible tests during the whole process.

Until a few years ago, Skov did not even use the phrase "safety critical" to describe itself. The company just said that it had to ensure that its products did not kill animals. However, in the past few years the company has become interested in finding out of how much it has in common with the more formalized safety critical industries, as it has grown in size. This has been an organic growth without drastic changes in process, meaning that the company has always been able to retain the experience that have been accumulated among the employees. The company's processes in general are primarily based on the experience of the participating persons. This is also true of the critical test process, which is not based on formal certification but on a slow growth of experience and adjustments.

6.2.3 *The safety critical standard*

Having looked at the actual safety critical development processes of two different companies, we will now focus on one of the safety critical standards that governs most of safety critical development, in order to get an impression of what they are like and what it requires to work with the standards. It is necessary to go into some detail regarding the standard, but the reader should not be discouraged if its technicalities

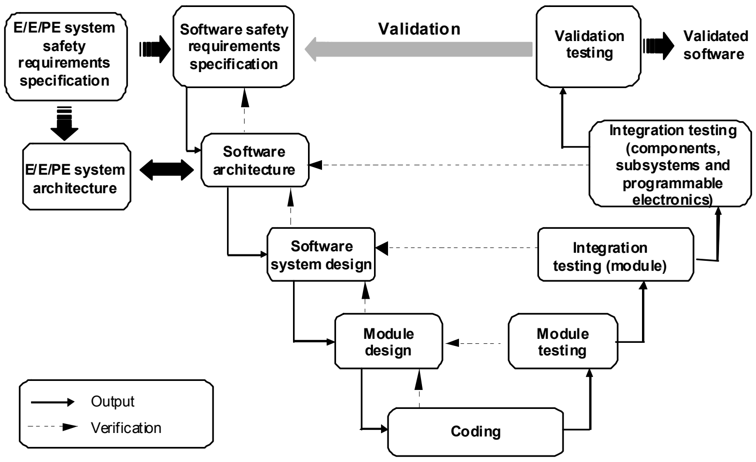

E/E/PE system safety requirements specification
Software safety requirements specification
Validation
Validation testing
Validated software
E/E/PE system architecture
Software architecture
Integration testing (components, subsystems and programmable electronics)
Software system design
Integration testing (module)
Module design
Module testing
Coding
Output
Verification

Figure 6.1: "Software systematic capability and the development lifecycle (the V-model)". IEC 61508-3. International Electrotechnical Commission 2009.

seem overwhelming – the aim is solely to give a taste of what the standard is like. Bear in mind that even for experts in the field, it will take years to become thoroughly familiar with a standard such as IEC 61508.

As mentioned, the IEC 61508 is the most important safety critical standard, at least in Europe. It is a document numbering 597 pages, in seven parts, which governs all of the safety critical development process in general terms not specific to any particular industry.[8] Part 3 – "Software requirements" – chiefly governs software development.

The standard specifies how software development should be done. Figure 6.1 shows the model for software development taken from IEC 61508-3 Figure 6, the V-model. This model was also mentioned in section 3.2.1.3 (Process thinking); it takes its name from its shape.

[8]International Electrotechnical Commission: IEC 61508-(1–7) Ed. 2.0. 65A/548/ FDIS Final Draft International Standard, distributed on 2009-12-18.

In principle, the choice of development model is free, since the standard states:

> "7.1.2.2 Any software lifecycle model may be used provided all the objectives and requirements of this clause are met."

However, in practice, the choice of model is severely limited, since it essentially has to perform the same function as the V-model. The next paragraph of the standard specifies that the chosen model has to conform to the document-oriented software engineering models described in section 3.2.1.3:

> "7.1.2.3 Each phase of the software safety lifecycle shall be divided into elementary activities with the scope, inputs and outputs specified for each phase."

Let us take a closer look at a typical paragraph of the standard:

> "7.4.4.4 All off-line support tools in classes T2 and T3 shall have a specification or product manual which clearly defines the behaviour of the tool and any instructions or constraints on its use. See 7.1.2 for software development lifecycle requirements, and 3.2.11 of IEC 61508-4 for categories of software off-line support tool.
>
> NOTE This 'specification or product manual' is not a compliant item safety manual (see Annex D of 61508-2 and also of this standard) for the tool itself. The concept of compliant item safety manual relates only to a pre-existing element that is incorporated into the executable safety related system. Where a pre-existing element has been generated by a T3 tool and then incorporated into the executable safety related system, then any relevant information from the tool's 'specification or product manual' should be included in the compliant item safety manual that makes possible an assessment of the integrity of a specific safety function that depends wholly or partly on the incorporated element."

Off-line support tools refers to the programs the developers use to make the safety critical software, such as editors, compilers, and analysis software. Categories T2 and T3 are those programs which can directly or indirectly influence the safety critical software; the categories are defined in sub-clause 3.2.11 of part 4 of the standard, as stated. Thus, paragraph 7.4.4.4 states that all programs that are used to make safety critical software, and which can directly or indirectly influence the software, have to have some documentation of the programs' function and use.

The reference to sub-clause 7.1.2 is probably intended to indicate where among the overall safety development phases the activities connected with paragraph 7.4.4.4 have their place. The software phases and their corresponding sub-clauses and paragraphs are listed in table 1 of IEC 61508-3, which is referenced from sub-clause 7.1.2.

The note to paragraph 7.4.4.4 makes it clear that the documentation required for programs used to make safety critical software is not the same as that required for the safety critical software itself. It also points out that if a program is used to make something that ends up in the safety critical software, then the relevant bits of the program's documentation should be copied over into the safety critical documentation.

A sizeable portion of the standard is made up of tables. In part 3 alone there are 43 tables, some spanning multiple pages. An example of such a table is shown in Figure 6.2. The table shows recommendations for design and coding standards, that is, guidelines for how to write the computer code. From top to bottom, the table lists some different techniques that can be used during coding. They are mainly prohibitions on using certain forms of code that are considered unsafe, such as, for example, automatic type conversion.[9] To the left of each technique are listed the safety levels from SIL (Safety Integrity Level) 1 to SIL 4. SIL 1 is the lowest level of safety. For each level and each technique, it is noted whether that particular technique is recommended, "R", or highly recommended, "HR", for the safety level

[9]A programming language technique which some programmers find more convenient, but which also carries the risk of allowing serious errors to go undetected.

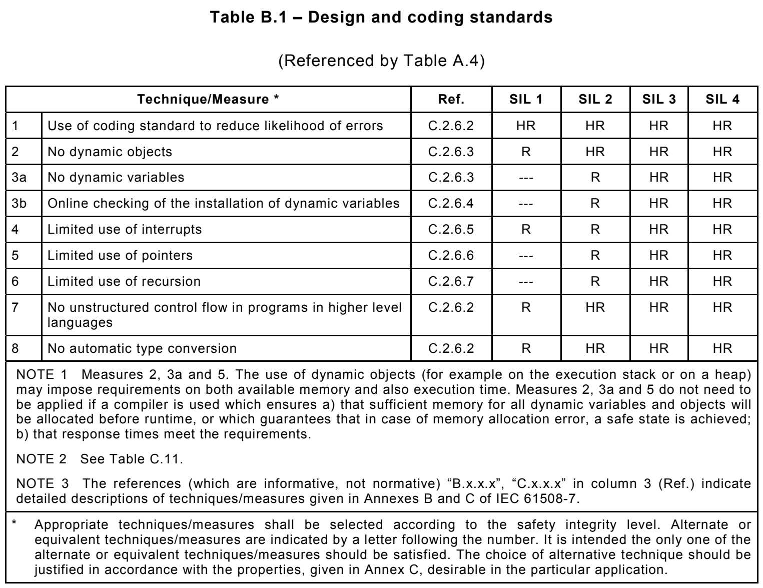


Table B.1 – Design and coding standards

(Referenced by Table A.4)

	Technique/Measure *	Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Use of coding standard to reduce likelihood of errors	C.2.6.2	HR	HR	HR	HR
2	No dynamic objects	C.2.6.3	R	HR	HR	HR
3a	No dynamic variables	C.2.6.3	---	R	HR	HR
3b	Online checking of the installation of dynamic variables	C.2.6.4	---	R	HR	HR
4	Limited use of interrupts	C.2.6.5	R	R	HR	HR
5	Limited use of pointers	C.2.6.6	---	R	HR	HR
6	Limited use of recursion	C.2.6.7	---	R	HR	HR
7	No unstructured control flow in programs in higher level languages	C.2.6.2	R	HR	HR	HR
8	No automatic type conversion	C.2.6.2	R	HR	HR	HR

NOTE 1 Measures 2, 3a and 5. The use of dynamic objects (for example on the execution stack or on a heap) may impose requirements on both available memory and also execution time. Measures 2, 3a and 5 do not need to be applied if a compiler is used which ensures a) that sufficient memory for all dynamic variables and objects will be allocated before runtime, or which guarantees that in case of memory allocation error, a safe state is achieved; b) that response times meet the requirements.

NOTE 2 See Table C.11.

NOTE 3 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. It is intended the only one of the alternate or equivalent techniques/measures should be satisfied. The choice of alternative technique should be justified in accordance with the properties, given in Annex C, desirable in the particular application.

Figure 6.2: Table B.1 "Design and coding standards" from IEC 61508-3. International Electrotechnical Commission 2009.

in question. For safety levels 3 and 4, all the techniques are highly recommended.

Since the table lists recommendations, the safety critical software is not absolutely required to use the suggested techniques. However, it must still fulfil the standard; so if it does not use a particular recommended technique, a good explanation is expected for why , and how the standard's requirements are nevertheless still fulfilled. This goes especially for the highly recommended techniques. Thus, while a company is in principle free to develop its safety critical software as it likes, in practice it must take into account that the more it deviates from the standard's recommendations, the more it must be prepared to argue, prove, and provide evidence for its own solution. As such, in terms of paperwork the easiest thing might well be to simply follow the standard's recommendations without question.

6.3 *Hermeneutical analysis*

After discussing the realities of safety critical programming in section 6.2.1 (A large avionics company), 6.2.2 (A small farming systems company) and 6.2.3 (The safety critical standard), we will turn to a hermeneutical analysis of safety critical programming, in parallel with the hermeneutical analysis of game programming at Tribeflame in section 5.2.2 (Hermeneutical analysis). The purpose is to show how hermeneutical theory can be applied in a concrete case. In the case of safety critical programming, the analysis is conceptually more difficult than in chapter 5.2 because we are not analyzing a single company's concrete work process, but rather the role that safety critical standards play in the work processes of different kinds of companies. To make matters even more complicated we are not even analyzing a concrete safety standard as such, but rather the idea of using a standard in safety critical programming. In other words, we will analyze safety critical standards as a cultural form.[10] The analysis is primarily based on 24 interviews made in connection with a pan-European research project about reducing the costs of certification, mostly according to IEC 61508. The project was sponsored by the European Union, and universities and companies from different safety critical industries participated. The interviews are summarized in the list of source material, page 133. For a short introduction to hermeneutical analysis, see section 3.3.1 (Introduction). For a more thorough explanation of hermeneutics, see section 3.3 (Hermeneutical theory).

The first step in understanding safety critical standards is pre-understanding; that is, those fundamental skills such as reading and writing that are a prerequisite even to beginning the process of under-

[10]See section 3.4 (Cultural form theory).

standing. In the field of safety critical standards, those fundamentals are: to understand engineering terms; to understand software engineering terms; and to have some knowledge of programming techniques, which is necessary in order to understand, for example, the table in Figure 6.2. In addition, it is necessary to be able to understand the kind of documents that IEC 61508 represents, and which is exemplified above by paragraphs 7.1.2.2, 7.1.2.3, and 7.4.4.4 of this standard. It is a lawyer-like language of numbered clauses, sub-clauses and paragraphs, full of references and abbreviations, which uses awkward sentence structures and many uncommon words with precise definitions. It is perhaps not a daunting task for a trained engineer to learn this, but many other people would find it challenging.

The prejudice inherent in safety critical standards are those assumptions that are necessary in order to make sense of them. The certification company Exida has published a book about the IEC 61508 that contains the following characterization of the standard:

> "Many designers consider most of the requirements of IEC 61508 to be classical, common sense practices that come directly from prior quality standards and general software engineering practices."[11]

It is undoubtedly the case that most of the IEC 61508 is common sense in safety engineering, but what is interesting here is that the safety critical notion of "common sense" is a very specific one, limited to a narrow range of situations. For example, the table in Figure 6.2 essentially recommends that the programming language technique of dynamic memory allocation should not be used at all (entries 2 and 3a). This is a good idea in a safety critical system, but in most other forms of programming, including game programming, it would not be common sense to program without dynamic memory allocation: arguably, it would be madness.[12]

[11]Medoff & Faller 2010 p. 2.

[12]Some common programming languages, most notably C, lack automatic memory management, but it is only the most hardware-centric programming languages that entirely lack the possibility of allocating memory dynamically. An example is the original definition of Occam, the Transputer programming language.

What, then, are the assumptions needed to make sense of safety critical standards? First of all that safety is an engineering problem that can be approached with engineering methods and solved satisfactorily in that way. Closely connected with this assumption is the notion that difficult problems can be broken down into simpler problems that can be solved separately, and that the solutions can be combined to form a solution to the original problem. A third, and also closely connected, assumption is that both process and problem can be kept under control, using the right approaches. All three assumptions are also found in the software engineering models described in section 3.2.1.3 (Process thinking).

Personal reverence is a foundation of understanding. A person interpreting a text or a practice must have some kind of personal attachment to the thing being interpreted. In Tribeflame, we saw that the personal reverence mainly took the form of personal relations between actual people; but personal reverence can also take the form of, for example reverence, for the author of a text. The safety critical standards do not depend on personal relationships; they are impersonal. Nor is there any particular reverence for the authors of the standards; they are anonymous or hidden among the other members of committees. To where, then, is the personal reverence directed in safety critical programming? The programmers revere neither colleagues nor authors, but instead they seem to revere the system of safety critical standards itself, as an institution.

The reverence associated with safety critical standards is probably a case of reverence towards a professional identity and code of conduct. What programmers revere in the standards is their professionalism, rather than the standards themselves. This is a consequence of their bureaucratic character of. According to Weber, modern bureaucracies replace personal loyalty with loyalty to an impersonal, professional cause.[13] The prerequisite for this shift in loyalty is that professionals have a long and intensive education,[14] which provides them with the necessary training and self-discipline to behave profes-

[13]Weber 1922 [2003] p. 66.
[14]Ibid. p. 65.

sionally. Accordingly, most of the people working with safety critical standards have long educations at university level or equivalent. Out of 26 people whom I interviewed, and who work with safety critical software development, 20 are engineers, five are educated in computer science, and one in physics.

All knowledge builds upon tradition of some kind. In the case of safety critical standards, the tradition goes way beyond that which is written in the standards themselves. In the words of an experienced engineer at Cassidian:

> "In order to understand the avionic standard it's really not sufficient to have a training course of only two or three days. I think a training course is helpful, but it covers only let's say 10 or 20 percent – if you are very intelligent, let's say 20 percent – of your knowledge, but the remaining 80 percent, it's really experience. It's your everyday job."[15]

The point is that it is not enough to read and study the standard alone; it is also necessary to acquire the needed experience in order to understand it. I have already mentioned the lengthy education of people who work with safety critical standards and which prepare them to understand the standards to some degree. The additional experience they require is what we can call exposure to the tradition of safety critical standards.

A specific safety critical standard does not exist in isolation. There are previous versions of the standard, and the standard itself is based on an older standard, and yet other standards have influenced it. The tradition of which a standard is part can be followed back in time through a trail of standards. The main components of the tradition are nicely summed up in the Exida quote above:

> "Many designers consider most of the requirements of IEC 61508 to be classical, common sense practices that come directly from *prior quality standards* and *general software engineering practices.*"[16] (My emphasis.)

[15]Interview 24 ~01:00:07.

[16]Medoff & Faller 2010 p. 2.

The "general software engineering practices" embody a tradition of engineering of which the safety critical standards are also part. The education of people working with the standards familiarizes them with this part of the tradition. The other part of tradition, the "prior quality standards", points to a long judicial tradition for regulating commerce and manufacture. The standards all have a more or less clear legal authority for deciding the validity of practices, and this authority is, per tradition, delegated to experts in the field. The history of engineering is so long and complicated that it is probably futile to try to point out exactly when engineering tradition was institutionalized for the first time.[17] Likewise with the judicial tradition, but it is at least safe to say that engineering practice was regulated via patent legislation as early as 1624 in England.[18]

The question of authority is important because it plays a central role in the hermeneutical notion of understanding. The texts of the safety critical standards carry great authority in themselves; but they can only do so because they have been assigned authority by outside parties. The ultimate source of authority is often the law, where national or international (European Union) legislation has adapted some standard as a requirement of professional liability. Where standards are not backed by courts, but rely solely on the agreement of industry associations, they will often be backed indirectly by courts, since there is often a legal formulation that the "state of the art" in engineering will be taken as the measure of liability, and the "state of the art" will then in practice often be taken to be the existing standards of the industry association.[19]

The words of the standards, however, are rarely taken to court in lawsuits, and therefore the authority of the courts are in practice delegated to other institutions, the most important being the independent assessment companies such as the German TÜV. These private

[17]Though it was probably in the military, since most early engineering applications were military – in contrast to the later "civil engineering".

[18]Cardwell 1994 p. 106. An example of a more recent institution that is closer in function to the standards (though distinct from them) is the Danish special court of commerce and maritime law, Sø- og handelsretten, which was established in 1862.

[19]Interview 32.

institutions have authority over the interpretation of the standards in practice, as they judge whether a product satisfies the standard or not. Since there are several institutions that act in competition with each other, they do not have absolute authority individually. Another important source of authority lies with the committees of experts that define the standards. Their authority, which represents a compromise position taken by the competing companies in a particular industry, is the guarantee that the standards do actually represent "general engineering practice", or in other words the "state of the art". A final, local form of authority is the management of the individual company which decides what standards the company are to follow, and as such acts as a source of authority over the individual developer.

The central question in understanding how safety critical standards are used is that of application – what the standards are used for. The obvious answer is that the standards are used to ensure that products are safe, but this answer is too simplistic. As we have seen in the case of the farming systems developer Skov in section 6.2.2, it is not strictly necessary to follow standards in order to develop safety critical products. This has to do with the fact that although part of the standards' function is to impose a certain "safe" engineering form on the product, this is not their primary function. The primary function is as a bureaucratic mechanism to generate documentation, by making it compulsory. The engineering manager for Wittenstein Aerospace and Simulation, a small company, describes safety critical software development thus:

> "The [program] code's a byproduct of the safety critical development, the code is sort of always insignificant, compared to the documentation we generate and the design. You can spend five percent of your time doing the coding in safety critical development, and the rest of the time on documentation, requirements, design, architecture, safety planning, evaluation, analysis, verification, and verification documents."[20]

[20]Interview 25 ~00:47:49.

It is of course crucial to notice that the engineering manager speaks not merely of requirements, design, and so on, but of documented requirements, documented design, documented architecture, and so on. It is clear that while documentation plays a central role in safety critical development, documentation of the development is not the same thing as safe products. To answer the question of what the application of safety standards is, we have to split the question in two: How do the standards contribute to safety? And what exactly is meant by "safety"?

The standards contribute to safety primarily by acting as a system for generating documentation, and the documentation serves as evidence for what has happened during development. This evidence serves to link the engineers' application of standards to the courts of law from which their authority derive by means of assigning responsibility for every part of the development. The legal system cannot carry out its authority without being able to clearly determine responsibility in each case. On the other hand, the standards are linked to the engineering tradition through the standards' insistence on repeatability. Responsibility cannot be assigned meaningfully to the engineers unless the engineers are in control of what they are responsible for, and the engineering tradition's way of enacting control is through repeatability. In the safety standards' assignment of responsibility, the role of the engineers is to provide the required evidence and to convince the standards' authorities that it is indeed sufficient. The role of the independent assessors, such as TÜV, is to provide the verdict on whether the engineers' evidence is convincing enough.

The question of what "safety" means is the central question in understanding safety critical standards. In parallel with the process at Tribeflame, which could be described as a single effort to determine what "fun" means for computer game players, the whole practice of safety critical development is essentially an attempt to find out what safety means. Being pragmatic, engineers would like to define unambiguously what safety means: that is, to operationalize the concept completely. However, this is not possible. An illuminating definition from an article about integrating unsafe software into safety critical systems reads:

> " 'Unsafe software' in the context of this article means: Software that does not fulfil the normative requirements for safety critical software (especially those of IEC 61508)."[21]

This definition is both pragmatic and operational, but it does not say much about the nature of safety. Defining safety as that which fulfils the standards does not help to explain why the standards are defined as they are.

We saw that repeatability is central to the standards. Repeatability is expressed in the safety standards through underlying probability models. These models are present in terms such as "failure rate" and "safe failure fraction", which combine to form the measure for safety – SIL (Safety Integrity Level). Thus, the safety critical standards' notion of safety is a probabilistic one.[22] What is probabilistic, however, is never certain, and so safety becomes not a question of whether accidents can be avoided altogether, but rather a question of how much one is prepared to pay in order to try to avoid an unlikely event. In that way, safety becomes a question of cost-effectiveness. In the words of a senior principal engineer for the microcontroller manufacturer Infineon:

> "The whole key in automotive is being the most cost effective solution to meet the standards. Not to exceed it but only to meet it. So you can put all the great things you can think of inside ... [but] if it means more silicon area

[21]" 'Nicht-sichere Software' – das bedeutet im Kontext dieses Artikels: Software, die nicht die normativen Anforderungen (insbesondere der IEC 61508) an sicherheitsgerichtete Software erfüllt." Lange 2012 p. 1.

[22]Interestingly, it is difficult to apply a probabilistic failure model to software, because in principle software either works or not. There is no such thing as "mean time to failure" in software, nor a notion of structural failure due to stress on materials, for example. Presentation by William Davy at Embedded Systems Laboratory, Åbo Akademi, May 23rd 2012.

> in the manufacturing, and it's not required to meet the standard, then you're not cost competitive anymore."[23]

In a hermeneutical perspective, understanding always happens as a result of a fusion of horizons of understanding. So in order to find areas where a lot is happening in terms of understanding, one has to identify the places where horizons of understanding changes a lot. In the world of safety critical standards, horizons change relatively little.[24] Once a horizon of understanding of the standards is established, it serves as a common language or a shared set of basic assumptions between the involved parties, making it easier to understand, and communicate with, each other. A quote from the software director of the elevator company Kone illustrates how the development models of the standards serve as a common language between developer and certifier:

> "Waterfall [i.e. the V-model] is definitely easier [to follow] for the notified body, for the people who are giving the certificate. Waterfall is easier for them because there is a very easy process, because you have steps, you have the outputs of each phase, and then you have the inputs for the next phase, so it's very easy to analyze."[25]

This shared horizon of understanding is a practical reality, but it is also an ideal; the ideal essentially being that a well-written standard should not need to be interpreted. However, this ideal is contradicted by reality, in which the shared understanding is not, after all, universal. This can be seen by the fact that it is not uncommon that the companies in safety critical industries have to educate their customers: to teach them the proper understanding of the standards, and

[23]Interview 31 ∼00:10:00.

[24]This is because practitioners have very similar backgrounds. Contrast with the practice of Tribeflame in which horizons change relatively more as the developers seek to expand their horizon to merge with those of their future customers, a mass of largely unknown players.

[25]Interview 29 ∼00:55:04.

to teach them "to see the intent behind the standard and not just the words on the page."[26]

This education of customers works to bring about a shared understanding if the people who need to be educated are prepared to accept the assumptions and prejudices of the safety standards. However, if their values are in conflict with the assumptions of the standards, a mismatch will occur. The principal engineer from Infineon sees a conflict between creative engineers and the standards:

> "Engineers like to create, they like to be inventive, they like to experiment ... Consequently engineers don't like the standard. They don't wanna adopt it, it makes their lives more painful, it makes them do more jobs they don't want to do. So they don't see any value in it because they think they're perfect engineers already."[27]

The conflict in this case is primarily with the implicit assumption of the standards, that documentation is the way to engineer a safe system. When, as the Wittenstein engineering manager says, 95 percent of the safety critical engineering job consists of documentation; this can be a real point of contest with other traditions who do not value documentation as highly. An example of other traditions are the Agile development philosophies, illustrated by the company Skov, which does "not produce documentation for the sake of documentation".[28]

In order truly to understand the safety critical standards, it is necessary to understand why they are written the way they are, and this requires understanding the two sides of their functioning: both the

[26]Interview 31 ∼00:50:02.

[27]Interview 31 ∼00:50:02. The conflict between creativity and adherence to process can be resolved by regarding, not the product, but the process, as a target for creativity. This seems to be appropriate for safety critical companies that are highly process oriented, such as Wittenstein: "We like to think we don't have a quality system. We have a business manual quality, and the way we run our business is running the same thing – so, we operate a business. And these processes are how we operate our business." Interview 25 ∼00:17:10. The same point is made about the development of the Space Shuttle software in a journalistic essay by Charles Fishman from 1997: "People have to channel their creativity into changing the process ... not changing the software."

[28]See note 7, page 182.

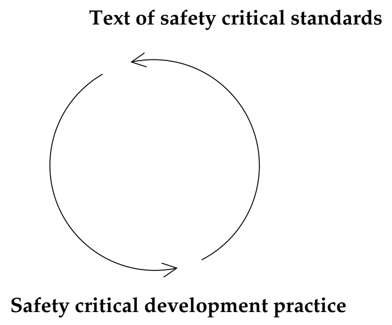

Text of safety critical standards
Safety critical development practice

Figure 6.3: Illustration of the hermeneutical circle between the safety critical standards themselves and the development practices that follow them. One cannot be understood without the other, and vice versa.

engineering side and the judicial side. In both the process by which the standards are created and in the work practices of safety critical development, we see a remarkable degree of influence between on the one hand the text of the standards, that resemble law texts, and on the other hand the practical engineering experience that comes from working with the standards. The influence constitutes a hermeneutical circle between the regulatory text of the standards and the engineering practice being regulated (see Figure 6.3). To understand one, it is necessary to understand the other. Indeed, it is practically impossible to understand the standard just by reading it, without prior knowledge of the engineering practice to which it refers. Likewise, the engineering practice of safety critical development is directed by the standards to such a degree that it would be hard to make sense of the developers' work practices without knowledge of the standards.

To consider the effective history of safety critical standards means acknowledging that they have grown out of certain traditions and that the problems they solve are the problems of those traditions: first and foremost problems of engineering, and of assignment of responsibility. It is to acknowledge that there are limitations to the problems that can be solved with safety critical standards, and that the notion of

Hermeneutical concept	**Results**
Prejudice	Engineering problem solving. The assumptions that problems can be reduced to simple sub-problems, and that development can be controlled.
Authority	Ultimately the courts of national and international law. More immediately industry associations and official certifying bodies.
Received tradition	A trail of standards. Institutions that preserve engineering knowledge and legal oversight of industrial production.
Personal reverence	Largely replaced by reverence towards institutions and a professional identity. A consequence of the bureaucratic character of standards.
Pre-understanding	Knowledge of engineering and software engineering terms, and programming techniques. Familiarity with the legalistic language of safety standards.
Understanding	Safety critical programming rests on both engineering and judicial tradition, and it is necessary to understand both.
Hermeneutical circle	The regulatory texts of the standards and the experiences of engineering practice presuppose one another.
Effective history	Largely absent.
Question	What "safety" means. Predominantly a probabilistic concept, which as a consequence leads to a search for cost-effective solutions.
Horizon of understanding	Horizons change relatively little compared to e.g. game programming. The safety standards and process models function as a common language between practitioners.
Application	To exert control over development processes through the production of documentation: a bureaucratic mechanism. Control is a prerequisite for placing responsibility, which is demanded by courts of law.

Figure 6.4: A schematic summary of the principal results of analysing safety critical programming with hermeneutical theory.

safety implied by the standards is a specific one – safety in this sense means to fulfil the requirements of the standards in a cost-effective way, and to avoid certain classes of engineering malfunctions with a certain probability. This notion of safety is not necessarily the same notion that the eventual users of safety critical products have. As was the case with game development at Tribeflame, there is very little explicit reflection upon effective history within the safety critical industries. Safety critical developers are of course aware that their practices and actions have important consequences in society, but they tend to regard their own actions wholly from within the thought world of safety critical development.

To sum up this hermeneutical analysis of safety critical programming, some of the results are presented in schematic form in Figure 6.4.

Chapter 7

Cultural form analysis

7.1 *Comparative analysis*

The examples of companies we have looked at so far – Tribeflame, Cassidian and Skov – have in common that their businesses depend critically on software development. However, what we have discovered with hermeneutical analysis is that their work processes are not dominated by the concept of programming but by concepts that stem from the products they make: in one case "fun", in the other cases "safety".

The hermeneutical analysis of Tribeflame was straightforward in that it is an analysis of a concrete, individual work process at a given time. The analysis of safety critical programming was more complex in that it is an analysis of the cultural form of safety critical work processes. It focuses on the essential features that make a programming work process safety critical in contrast to something else. In this section we will compare the cultural practice of safety programming with the cultural practice of game programming that is represented by Tribeflame's work process.

The standards and processes that are used in safety critical programming is a form of cultural practice. When they are used in practice, they take the form of hierarchical organization and procedures to be followed, with the goal of assigning responsibility to individual steps in the programming process and controlling it. It is clear from the examples we have seen so far that this particular cultural form is not absolutely necessary to programming, but that it can be useful in some cases, depending on circumstances.

Cassidian is constrained by its customers in that it is required to fulfil the safety standards. The company makes the most of this by adopting the thinking behind the standards and organizing the whole company's work around a cultural practice that follows the standards quite closely. Moreover, the company is so large that it believes it can influence its circumstances; not passively being subject to the dictates of the standard, but attempting to impose its own version of the standard on its customers. For Skov, there is no external requirement to follow standards and processes, and the company is consequently free to adopt a development practice that it likes and sacrifice rigid control of the programming process. The company only considers it worth having a control- and document-oriented approach in a limited area of the development process: namely, the testing process. For Tribeflame, there is no external constraint and the company clearly sees no benefit in adopting a rigid process. Of course, this does not prove that it is impossible to run a successful game company following the principles of safety critical standards – but the example of Tribeflame does show us that the programming principles inherent in safety critical standards are not strictly necessary to game programming.

An important feature that these examples have in common is that though their businesses depend crucially on programming, the goal of their business is not programming. What they are really striving for is to make games that are fun to play, or products that are safe to use. Programming is, in this respect, a means for the business to achieve its goals; and the specific form given to the programming process – whether the V-model, Agile, or something else – is merely a sub-means to the programming means. To put this another way:

the goal of the programming process is to make a program. The program is a sub-goal that itself acts as the means to the real goal of the companies: to make their products useful.

The safety critical programming practice that follows standards is traditional: it builds on other, older traditions. These are primarily the engineering tradition, legal tradition, bureaucratic tradition, and software engineering tradition. Although the software engineering tradition is arguably a descendant of the engineering tradition, the two are not identical. Even among its practitioners, software engineering is frequently regarded as something that is not "real engineering".[1] Building on these diverse traditions, the safety critical programming practice has existed for so long that it has itself become a distinct tradition, expressed in the standards' chapters on software development and in the non-written knowledge about safety critical programming that is preserved among the professionals who work with it. As we saw in the hermeneutical analysis, the safety critical tradition is largely impersonal, based on extensive education of its practitioners and commitment to a professional identity.

The game development practice at Tribeflame is also based on tradition, but this is a tradition of a different character. The developers here related to a tradition that is primarily personal and based on direct experience rather than formal education. Their knowledge of the tradition comes from the games they play themselves, from games they have heard about through friends, and from exposure to computer games and other games since their youth. It is a tradition that is shared with many people who are not themselves developers of games, but merely play them. For most people, the computer game tradition is more immediately accessible than the safety critical tradition.

[1]See for example Jacobson et al. 2012 p. 9: "Software engineering is gravely hampered today by immature practices. Specific problems include: The prevalence of fads more typical of fashion industry than of an engineering discipline. . . . " Also Bryant 2000 p. 78: "Whatever the basis for understanding the term *software engineering* itself, software developers continue to be faced with the dilemma that they seem to wish to mimic engineers, and lay a claim for the status of an engineering discipline; but commercial demands and consumer desires do not support this."

In safety critical development, the processes and standards function as a common language that makes it easier to communicate across companies and departments – both for engineering and bureaucratic purposes. It works well because most communication is with other engineers who have similar training and experience. As a cultural form, the common language of safety standards is a response to the circumstances of safety critical development. The products are so complex to make that a vast number of people are involved and they need to talk to each other about their work in order to coordinate it.

In Tribeflame, investors and computer game players are the outsiders to whom the developers mostly need to talk about their work. They need to talk to investors in order to convince them that they know what they are doing, and they need to talk to players, who represent the actual customers of the company, in order to get feedback on their games. They have a need for talking about their games, but they do not have a great need for talking about exactly how they make them. An investor might of course take an interest in how the company carries out its development in order to be reassured that the invested money is spent well; but it is essentially not the investor's job to inspect and control the development process in the way that an assessment agency like TÜV inspects and controls the safety critical development process.

In safety critical programming, the developers' need to discuss their work has, over time, led to the development of models, most notably the V-model, which serve as references during arguments and discussions about the work. In this sense, the models are argumentative: an aspect of a work process can be explained by pointing out the place in the model where it belongs.

In the computer game industry, the need for discussing work processes is not so great. Consequently, it would not be an efficient use of time and energy to develop and maintain consensus about models of the working process. Whenever game developers do need to discuss their work process, it is likely to be more efficient for them to come up with an argumentative model of their process spontaneously, in a face to face meeting.

It is widely known that the process models of software engineering rarely describe what actually happens in development practice. A textbook on software requirements states:

> "The waterfall model represents an ideal. Real projects don't work like that: developers don't always complete one phase before starting on the next. . . . All this means that the phase idea of the waterfall model is wrong. The activities are not carried out sequentially and it is unrealistic to insist that they should be. On the other hand, developers do analyze, design, and program. The activities exist, but they are carried out iteratively and simultaneously."[2]

One of the founders of a consulting company in the safety critical industry characterizes the development processes of automotive companies in this way:

> "Usually they work. The car companies manage to put out car after car precisely on the scheduled time. They have these start of production dates that they can never shift, they are fixed a few years in advance and they always keep them and I think this is rather impressive. If I were to form an opinion I would say that they are really impressive and they work really really well.
>
> That's the big view of it, of course if you look at the smaller view then you see that for instance in software development, things do happen to go wrong, so you always have a last minute panic when the start of production comes closer and the control units don't seem to work the way they were intended to work, and then you see some kind of panic mode: that people work overtime and it is stressful and everybody has to examine where the

[2]Lauesen 2002 p. 3.

> problems come from, to change the software, to make additional tests. This is something where you think that obviously this is no straight development path that is being taken here, it's a little bit chaos in the end, and ... this means that the development processes for software are not working as nicely as they should work."[3]

Another experienced consultant, a founder of three consulting companies, explains that important parts of the safety critical development process necessarily must take place outside the framework of the V-model:

> "The V-model is a nice framework for reasoning but I never saw someone adding all the complexity of a problem at the specification stage. I mean, in fact you have plenty of implicit assumptions on the design. Also, you need to go further in the development in order to, for instance, evaluate some solution, and then go back to the specification. You know, the process is not so straight, so I think it's better that you have some sort of prototypes and some trials before you go back to the specification stage. And only when you have a precise idea of the system and the system behaviour, in the form of a technical solution, only then is it useful to do verification of the specification and the design. ... You must sort of go and return between specifications and proof of concepts, go back to the specifications and so on. And once it has been more or less validated you can enter the V-model and synchronize verification and development."[4]

In the software engineering literature, the usual explanation for the fact that process models rarely describe what people actually do is that the models are not actually meant to describe what people do: they are prescriptive, not descriptive, serving as an ideal for what

[3]Interview 16 ∼00:30:18.

[4]Interview 21 ∼00:50:01.

people *should* be doing. That the ideals do not work out in practice is blamed on human weakness and organizational inadequacies. The hermeneutical analysis of safety critical programming provides a much simpler explanation: that the process models are not essentially models of how the work is done, but models of how the work is understood by those involved in it. This makes sense when the models are understood as part of a common language through which the work processes can be talked about. In the words of the safety consultant, the models are a framework for reasoning; not only reasoning with oneself but also reasoning with others, as part of communication.

In the game programming practice of Tribeflame, the lack of bureaucracy means that there is no direct counterpart to the process models of safety critical programming. Of course the developers have some kind of mental model of how they do their work, but since they rarely need to talk about it, it is not explicit or schematic. Moreover, it is far more flexible than formal process models. For example, Tribeflame came up with the idea of having a short meeting for the whole company every day and instituted it in the course of a few days.[5]

All forms of cultural practice are dependent on history, and therefore it is important to understand their history in order to understand them. Safety critical programming has a long history of institutions that enforce standards: courts of commerce, regulatory bodies, assessment companies, and industry associations. So far game programming has a lot less institutional history – the existing game programming institutions are mostly the game companies themselves. The historical aspect of game development often comes through the developers' personal history. Game development is shaped by the history of tabletop games and the history of computer games. When a new game company is formed by inexperienced programmers, the work practices they bring to game programming are the traditions of working at a desk job, and in many cases the work practices of conducting study projects in university or other educational institutions. When inexperienced engineers join a safety critical company, they are faced with a long-standing tradition of safety critical work practice

[5]Field diary, 30th August 2011, 11:25-12:15.

with which they have to familiarize themselves before they can hope to have an impact on the industry.

Forms of practice are dependent on history but they do not merely repeat old traditions. Cultural practice is shaped by the goal of the practice, no matter what the historical origin of the practice is. For this reason, the goal of a cultural practice will be able to explain much about why the practice looks the way it does. In safety critical programming, the goal is to make safe products through the mechanisms of control and assignment of responsibility. In the cultural context of safety programming, control and assignment of responsibility therefore appear as sub-goals, meaning that much of the cultural practice of safety programming has to be understood through these concepts. In game programming, the goal is to provide entertainment through the sub-goal of making the products fun to play. Since fun is a somewhat ambiguous concept, much of the cultural practice of game programmers revolves around trying to find out what fun is, in the eyes of potential players.

Software engineering theory is created for, and suited to, specific cultural contexts that are shaped by external hermeneutical demands, primarily the demands of bureaucracy and legal systems. This is particularly true for safety critical development but it also applies to ordinary software engineering, which in the main has its origin in projects for the U.S. Department of Defense and similar official bodies.[6]

In game programming, those particular bureaucratic and judicial demands are missing. To follow a process model like the V-model would be a waste of resources, unless, of course, the process model served some other specific purpose for the company. To insist in game programming on a professionalism of the same kind as that in safety programming would be pointless, since professionalism is a mechanism rather thana goal in itself, and software engineering professionalism has been created to serve goals other than those of game programming.

[6]See section 3.2.1 (Software engineering).

The purpose of software engineering practice – and safety critical practice in particular – is not essentially to create entirely new things, but largely to preserve the existing state of affairs, the *status quo.* Especially in safety critical practice, this is easily understandable because it is dangerous to make new, untested products.[7] It is better to keep to the traditional way of doing things until it is absolutely certain that a new way of doing things is equally safe. For this reason, new inventions are only allowed if they are kept within strictly defined parameters, which are defined beforehand and permitted to be adjusted only within limits. Inventions that cannot be fitted within the predefined parameters will usually be rejected in order to err on the side of safety.

This practice of the conservation of the existing state of affairs is enforced conceptually by a hierarchical categorization, which is in turn the source of the parameters within which variation is allowed. An example of hierarchical categorization is the V-model: the overall category is a sequential process of phases; within each phase are inputs, outputs, and workflows; each input or output can again be divided into its constituent parts and so forth. In practice, conservation is enforced by rigorous procedures that ensure that things are done in a certain way according to the hierarchical categorization. The procedures are, for example, the step-by-step workflow descriptions for carrying out the V-model in practice, or the list of documents that must be written, verified, and approved in a certain order. The bureaucracies connected with software engineering exist, not for their own sake, but to uphold the tradition of engineering practice.[8]

Game programming is different in this respect, because tradition is experimented with quite freely. If a game oversteps the boundaries that players find acceptable, this will quickly become apparent in the market as the game will sell badly. Computer games are a form of entertainment, and in entertainment businesses in general it

[7]Bear in mind that much ordinary software engineering has its origins in a military context, and the military can indeed also be a dangerous work environment.

[8]Should the bureaucracy become an end in itself a dysfunctional work practice will be the result.

is not particularly valuable to preserve the existing state of affairs: customers demand constant innovation and change, which is why creativity is so highly valued within these industries. Of course, computer games do have to preserve some amount of tradition – games are expected to stay within broad genres. A platform shooter, for example, has to conform to the expectations to the platform shooter genre; but within these broad limits the more innovative games tend to be more successful.

We see that the hermeneutical interpretation of game programming and safety critical programming in chapters 5.2 (Analysis) and 6 (Safety critical programming) support the cultural practice analysis of the current chapter: safety critical programming practice is very traditionally bound. The hermeneutical circle of interpretation works very slowly in a safety critical context: the time span from a standard being interpreted to its consequences being carried out in practice, and from there to the consequences affecting the original interpretation, can be many years. The safety critical processes are thus slow to carry out in practice and slow to change. The strong emphasis on tradition and bureaucracy means that safety critical practice offers a precise and easy form of communication that is also inflexible, because it cannot accommodate thoughts outside the hierarchical categorizations, and requires years of highly specialized training.

Game programming practice is, by comparison, fluid, because the timescale of the hermeneutical circle operating in practice is short – consequences of interpretation can often be seen in mere days or weeks. This means that the practice can change more quickly. The practice is seldom explicit, but lives in the actions and thoughts of the game makers: it is simply inefficient to write everything down. Consequently, the practice is seldom presented to outsiders other than in the form of the product – the game. The tradition that underlies game practice is more personal and informal, and the ability to contribute constructively and get involved is more important than formal qualifications. In this way the work is similar to that in other creative jobs, such as writing and design.

In safety critical programming as well as in game programming, we see that in the work practices, the concept of programming is subordinate to other concepts that more directly express the goals of the companies. In the case of safety critical development, programming is subordinate to machine production and its regulation. As we see in section 6.2.3 (The safety critical standard), much of the vocabulary that is used in safety development comes from factory settings or from the legal sphere, and the terms used in safety development are usually object terms – they describe how to reach given goals, without questioning the interpretation of the goals to any great extent. In game development, programming is subordinate to entertainment, and entertainment is usually measured by how much fun a game is. The observations from Tribeflame indicate that the discussion about game development centers around terms that express the players' experience of fun, and as such they are subject terms.

However, though programming in both practices is subordinate to the goal of concepts safety and fun, it is important to keep in mind that programming, in essence, is neither of these concepts. Though directed by the dominating goal concepts, programming has its own cultural characteristics that are not fully explained by the goal concepts. We will return to this point in the conclusion, chapter 9.

7.2 *Safety critical case studies and analysis*

So far we have looked at safety critical programming as a single cultural form, which means that we have been looking at safety critical programming as a more or less undifferentiated, homogenous practice. This was the case both in the hermeneutical analysis of safety critical programming in chapter 6 (Safety critical programming) and in the section above, which contrasted safety critical programming with game programming. Of course, safety programming practice is not a single, undifferentiated activity: in reality, it displays as much variety as any other cultural activity. In this section, we will look at how the concept of cultural form can be used to analyze safety critical programming in order to categorize and explain some of the differences that are visible in the practices as they are expressed in empirical data. Thus, this section is based on interviews with employees in 17 companies in the safety critical industry which provided data that was suitable for form analysis (see the list of source material, page 133). The companies are briefly summarized in Figure 7.1.

At this point it is probably prudent to repeat the warning that, to the reader unfamiliar with cultural research, it might seem strange to say something about the practices of a whole industry on the basis of only 17 cases, and sometimes on the basis of just a single case. The explanation for this is that we are not really trying to say anything about the "average" company, whatever an "average" company might be. Rather, we are trying to say something about the *possible forms of practices*, and to that end a single case is more than sufficient, because it demonstrates what is possible in practice. This of course means that we are not guaranteed to end up with a complete picture of all possible forms of practice; however, when the reader encounters a new, hitherto unknown form of practice, he should be able to analyze the practice himself, with the help of the cultural concepts of form and practice and the examples provided by this chapter.

The majority of the companies that we are observing – 10 out of 17 – are required to follow the safety critical standards, and they have done so in the most obvious way, by building a bureaucratic organization that revolves around the standards and which has internalized the development models that the standards prescribe, both explicitly and implicitly. Below, we will take a closer look at the variations in the way in which these companies work; first, we examine the remaining group of companies, which for one reason or another are not required to follow safety critical standards development procedures. Although these companies are not subject to the dictates of the standards they do not fall entirely outside the safety critical industries – each company is in its own way connected to safety critical development. Looking at these forms of development practice can, by contrast, tell us some-

Company	**Size**	**Dept. size**	**Office location**	**Product or industry**	**Process**
Skov	300	45	Glyngøre	Farming	Agile
(anonymous)	25		Germany	Automotive	Agile
Integrasys	20		Madrid	Satellites	Software engineering
FCC	700	60	Madrid	Simulation, planning	Software engineering
(anonymous)	60000	100	England	Research	Eclectic
Metso Automation	700	5	Finland	Valves	Software engineering
Wittenstein	1800	15	Bristol	Real-time OS	Safety critical
(anonymous)	90		Germany	Real-time OS	Safety critical
Cassidian	31000	70	Ulm	Aircraft	Safety critical
PAJ Systemteknik	20		Sønderborg	Medico, railway, sensors	Safety critical
(anonymous)	28		Finland	Satellites	Safety critical
PSA Peugeot Citroên	200000	700	Paris	Automotive	Safety critical
Delphi Automotive	146000	300	Wiehl	Automotive	Software engineering
Danfoss	22000	2000	Graasten	Electrical motors	Software engineering
(anonymous)	800	250	Germany	Automotive	Tool supported
Safe River	10		Paris	Consulting	Formal methods
Kone	33000	85	Chennai	Elevators	Software engineering / Agile

Figure 7.1: Brief summary of the companies mentioned in this section, in the order in which they are encountered in the text.

thing important about the practices of companies that do actually follow the standards.

The first thing to notice is that, because of the costs, no company follows safety critical standards unless it is forced to. The farming systems company Skov is one example of a company that is unaffected by standard regulation. In section 6.2.2 (A small farming systems company), we saw how Skov rejects traditional software engineering thinking and has adopted a less formal, Agile way of working. Testing is the only part of the process where Skov finds it worthwhile to maintain a formal work process. The circumstances that allow Skov's form of work practice to flourish are unique among the companies studied here, in that Skov is a company that produces safety critical systems but operates in an industry that is not regulated by standards.

Another example of a company that escapes standard regulation is a small company of 25 employees that makes timing analysis software tools for the automotive industry.[9] This company has built its development process around Scrum, an Agile methodology.[10] Like Skov, the testing part of the company's process is given high importance. But unlike Skov, the automotive industry is not free of regulation; on the contrary, it is heavily regulated by standards. The reason that the timing analysis company can escape regulation is that none of its software ends up in the finished cars: it is used by other companies as an aid in their own safety regulated processes.

This is a conscious strategy taken by the company, and allows it to occupy a niche in the automotive industry without following safety critical standard procedures. Thus both Skov and the timing analysis company find themselves in circumstances where they are free to follow an informal, Agile work process, but where the absence of regulation in Skov's case largely depends on factors outside Skov's control, in the timing analysis company it is a result of a deliberate choice. Of course, this choice has consequences: the company is limited in the range of software it can offer, since it cannot be used directly in safety critical hardware. Thus while the form of the company's work process

[9]Interview 19.

[10]See section 3.2.2 (Agile software development).

has advantages in some repects, such as the freedom to use an Agile process, it has disadvantages in others.

Integrasys is a small company of 20 employees that makes signal monitoring software for satellites and the aerospace industry.[11] The company does not follow any standards: it is preparing itself to use IEC 61508, but has no experience so far. The company's work process is a traditional software engineering phase-based process. The company does most of its work in large projects involving as many as 20 larger companies. Coordination with these companies dictates the working processes, and the planning and communication takes place mostly via software engineering Gantt charts.[12]

What makes Integrasys special is that, while it externally appears to follow formal, bureaucratic phase-based processes, the internal day-to-day planning is informal. Requirements are written in Microsoft Excel spreadsheets, and there is no formal evaluation – any experience that is built up is personal experience. Integrasys is unique among the studied companies in having a process that externally appears bureaucratic, but is informal internally.

The reason that this form is possible for the company is probably twofold. First, the company is quite small; a larger company would presumably need a more formal bureaucracy. Secondly, the company is subject only to the relatively lax requirements of ordinary software engineering processes, rather than the much more strict safety critical standard processes. In combination, these circumstances allow the company to have a software engineering process with very little bureaucracy. Of course, the fundamental principles in the company's work processes are still those of software engineering, in contrast with those companies discussed above, which follow Agile principles.

FCC is a medium-sized company with 700 employees, which produces simulation and mission planning systems for military and civil authorities.[13] Most of the software is made in the systems and telecommunications department, which employs 60 people. The company is

[11]Interview 11.

[12]See Figure 3.7 in section 3.2.1.3 (Process thinking).

[13]Interview 28.

not normally subject to safety standards; it has recently begun its first safety critical project. Safety critical development has been found to be more work than expected, although not particularly difficult. Normally, FCC's customers dictate the work processes. The processes are all software engineering processes and the core of the company's form of work are military software engineering standards.

This company is different in one important repect from all the companies in this study that follow safety critical standards. Companies that follow safety critical standards are usually very conscious about their working processes and frequently evaluate and revise them, but FCC has not changed or proposed improvements to its methodology in 11 years. Rather, if there are problems, the company postpones its deadlines and milestones. This is in sharp contrast with companies following safety critical standards, which are dedicated to meeting their deadlines meticulously.

Other companies do not have the option to postpone their deadlines – so how can this be possible for FCC? Part of the answer is undoubtedly that the company follows ordinary software engineering standards. These standards, though strict, are relatively lax compared to the even stricter safety critical standards. The other part of the answer might be that FCC primarily delivers to the military and civil authorities, and these kinds of public institutions are well known for suffering delays and deadline postponements. Indeed, though delays are generally viewed as the worst kind of problem within software engineering, the origins of software engineering in part lies in attempts to bring the delays in public institutions' projects under control.[14]

The next example is a research company made up of fewer than 100 people, which is part of a larger engineering group that employs 60,000 people.[15] The company makes proofs-of-concept, mashups and demonstrators in order to investigate the feasibility of proposed new engineering solutions. Some projects are as short as four weeks; some are longer. The work processes are continually adapted to whatever the particular customer or project demands, and so the company

[14]See section 3.2.1.2 (Brooks).

[15]Interview 26.

works with a number of different software standards, some of which are safety critical standards. According to the employee whom I interviewed, the best way of working is that the employees themselves decide how to do their work. That is both more efficient and more motivating than safety critical work processes, in which people only become experienced and efficient in working with the standards after a long period of time. This employee also expressed that all software processes are exactly the same, whether they are Agile or safety critical.

The research company demonstrates a way of thinking that departs both from common software engineering thinking and from Agile thinking. Instead, it is much more in line with the thinking that follows the traditions of computer science, described in section 3.2.3 (Computer science), where there is a marked emphasis on the creativity and insight of the individual. A work process is not seen as something shared but more as something private. This style of thinking fits the form of the research company exactly, because it is not overly concerned with long term efficiency and the cost competitiveness of production. It has much more in common with the kind of scientific and experimental mentality that lies behind the computer science tradition. The company is partially subject to safety critical standards and is, in many ways, in the same circumstances as companies who follow the safety critical standard tradition. However, the company pursues research, not cost-effective safety – and this difference in goals makes a difference in the form of its work processes.

Our final example of a company that does not conform to safety critical standards is Metso Automation, a company of about 700 people that makes automatic valves for the chemical process industries.[16] Only four or five people make the software to control the valves.[17] Because the company has non-software solutions for ensuring the safety of the valves, the certification agency TÜV ranks their software as not safety critical and thus Metso does not need to follow those standards.

[16]Interview 22.

[17]The valve control software is technically firmware, and the company refers to it as such.

The software people are trying to follow a traditional software engineering V-model process, with requirements, reviews of requirements, phases, inputs, and outputs. However, they are struggling to do so. The process is not written down in its entirety anywhere, and only about 10 percent of the software is covered by tests. Furthermore, the software architecture does not support module testing, which leads to problems in the work process. The company is reluctant to spend time and resources on improving the software because it is not seen as crucial to product quality. Hence the software process is allowed to continue to flounder.

What we see in Metso is a software department that tries to use a form of programming that is not really suited to its circumstances. The department lacks the necessary company support to build the bureaucracy that is needed in order to work the way it wants to, but it is either unwilling or unable to take the consequence and abolish the V-model way of working altogether. This is perhaps because those working in the software department do not know of any other tradition for software development, or perhaps because there are too few of them to build a new work tradition in the company. The underlying problem, however, is that the software department is just a tiny part of a much larger company that is entirely dominated by hardware engineering. Since the company views itself as a hardware company, software is simply not seen as something that can threaten the well-being of the company, and therefore the process problems are allowed to persist. For the time being, the company seems to be correct in this assessment.

Next, we will look at a range of companies that are subject to safety critical standards and conform to them both in deed and in thought. We have already discussed Wittenstein Aerospace and Simulation in section 6.3, page 193, a small company of about 15 employees.[18] The company's main product is an operating system for use in flight software. The company is strictly bureaucratic, conforming to the safety critical standards. All work is organized around the documents and phases required by the standards. Another slightly larger company

[18]Interview 25.

of 90 employees makes an operating system for embedded software in general.[19] Its working process is similarly bureaucratic and strictly conforming to the standards.

Wittenstein and the larger company have in common that they produce operating systems, which are pieces of software that cannot in themselves be certified according to the standards because they do not constitute a complete safety application in themselves. Accordingly, the certification is actually done by the companies' customers. This means that it is important to the companies that their customers understand the products and how the work processes conform to the standards. An engineer and project manager from the larger operating system company says that:

> " ... they need a certain understanding, the customer, because they cannot work with it if they do not understand; and if we deliver some artefact – some document or things like that – the customer needs to understand it because he has to take these documents and work with them inside his [own] company ... "[20]

This emphasis on the need for customers properly to understand the products is particular to the operating systems companies, because they cannot themselves complete the certification process. This phenomenon is not only found in software companies, the hardware company Infineon that produces integrated circuits is faced with the same challenge.[21]

In section 6.2.1 (A large avionics company), we saw a detailed account of the work process of Cassidian, a large aerospace company. Cassidian works with a large number of standards, and in an attempt to cut down the ensuing confusion it has developed an internal standard that combines the elements of all the standards with which it works. The internal standard is also part of an attempt to streamline the processes inside the company. Cassidian's size makes it powerful

[19]Interview 27.
[20]Interview 27 ~00:30:33.
[21]Interview 31.

enough to try to influence its customers, and its desire is to impose its own standard on its customers, in place of the various standards that the customers demand.

PAJ Systemteknik is a small company of 20 people that works as a subcontractor and assembles equipment for major companies such as Siemens, MAN, and Honeywell.[22] The company works in the areas of medical industries, railway, and safety of machinery. PAJ also deals with a large number of different standards that are dictated by its customers. Unlike Cassidian, however, PAJ does not have the size to influence its customers. The company therefore follows another strategy for trying to reduce confusion: its ambition is to develop a "self-certifying" platform for its products. That means a set of procedures that, if they are followed, will guarantee that the resulting product can be certified. Whether the strategies employed by Cassidian and PAJ will work remains to be seen, but it is interesting to note that differences in circumstances cause the companies to react in different, yet similar, ways to the same challenge: the proliferation of standards.

A further example is a small company of 28 people that makes control software for satellites.[23] Like PAJ, this company works as a subcontractor and has its work processes imposed on it by customers. The standards in the industry are dictated by the European Space Agency. The company has growing business in other industries, such as nuclear, railway, production and medical industries. This part of the business is becoming more important, and consequently the company is working with an increasing number of standards in addition to the space standards. These non-space standards are perceived by the company as using the same concepts as the space standards, but applied in a slightly different way. An engineer from the company explains:

> "Our processes are mostly derived from the European space standards. When we are to work in [non-space] industrial

[22]Interview 14.
[23]Interview 13.

> applications, well, it is a variation of that. So, it's not a completely different story, it's more or less the same concepts applied in a slightly different way. The names of some things are different; maybe you do some things before, some things after; some things more or some things less – but it's a variation of what we already know."[24]

Like PAJ, this company is trying to control the challenge of working with a number of standards; but unlike PAJ, it is not trying to create a single procedure that fits all kinds of standards. Rather, it identifies what is common in the standards, and thinks of the different standards as variations on what they already know: a strategy that presumably makes it easier to deal with the differences.

PSA Peugeot Citroên is a very large European car manufacturer. It employs 200,000 people, half of them in France.[25] In many respects, the company operates in circumstances similar to those of Cassidian. PSA Peugeot Citroên is a large, highly bureaucratic and tightly controlled organisation. The planning that goes into producing a new model of vehicle is comprehensive and very strict; milestones and deadlines absolutely have to be obeyed. There are also some interesting differences – where Cassidian tries to streamline and centralize its working processes by developing an internal standard, PSA instead allows different parts of the company to have their own processes and traditions, or, as an engineer from the company puts it, their "historical reasons to work in a certain way".[26] The reason for this is that "if they work in a certain manner they also have some good reason."[27] The company has a department of innovations that makes suggestions about changes in the work processes of the different departments, in close cooperation with the departments in question. This process can take several years and again shows that PSA's approach is much less centralized than Cassidian's.

[24]Interview 13 ~00:25:01.

[25]Interview 23.

[26]Interview 23 ~00:30:01.

[27]Interview 23 ~00:35:26.

Cassidian tries to affect the standards to which it is subject by making its customers accept its own internal standard. PSA also affects the standards, but in a different way. The company is of such size that it has representatives in the ISO committee that authors the standards, and PSA can thus influence the standard to make it accord better with the company's wishes. The company also uses some strategies for reducing the complexity of working with the standards. The main software component of a car[28] is consciously kept at a low level of safety criticality.[29] This means that there are important safety components that must instead be taken care of in other parts of the car, but it simplifies the work on the software component. Another strategy is to allow subcontractors to take care of the safety critical aspects of components. PSA could, in principle, safety engineer the components itself, but it simplifies the work process to have trusted subcontractors do this.

The following two examples differ from the others used in this study in that their work processes do not spring directly from the safety critical standards, but instead have their origins in general software engineering theory that is adapted to fit safety critical purposes. Delphi Automotive is a global company with around 146,000 employees.[30] It makes, among other things, embedded control systems for cars. The company's software processes derive from the SPICE standard,[31] which is a software process standard that is not concerned with safety: it has been chosen because of customers' requirements. The company has a globally defined standard process that is tailored locally in the individual departments and to each project. The form of the work process thus mixes a centralized and decentralized approach. Local departments deal with as many as 30 different safety critical standards and other legal requirements. The company is large enough that it is able to influence the standards to which it is subject;

[28]The Electronic Control Unit (ECU).

[29]Called ASIL B, the second lowest safety level, excluding components classified as not safety critical.

[30]Interview 15.

[31]Software Process Improvement and Capability Determination, ISO / IEC 15504.

the German part of the company is a member of the German working group for the ISO standards committee.

Danfoss Power Electronics is a daughter company of Danfoss, a company of 22,000 people that makes pumps and other hardware.[32] Danfoss Power Electronics makes electrical motors and the software to control them, and has around 100 software developers. The company follows a work process of its own devising that is an elaborated-upon version of an ordinary software engineering iterative waterfall model. The company has a slightly different version of its software process for each of its three software product lines, because standardizing the process to have "one-size-fits-all" is not deemed to be worth the effort it would take. Since the processes have not been made with safety in mind the company needs to interpret the process steps from the IEC 61508 standard to match its own process steps whenever a product needs to be certified.

An interesting detail is the way the company keeps track of its software requirements. Currently, the requirements are linked directly to the software source code simply by putting comments in the code. But they are considering adding an intermediate layer of "features", such that requirements would be linked to features and features in turn would be linked to the source code. In that way it is easer to keep track of functionality that is spread out in the source code. The programmers would then arguably have to be more aware of exactly which feature they are working on. This is an example of how demands can shape the programming work process; in this case, bureaucratic demands rather than demands arising from the programming itself.

The following two examples illustrate the inherent conflict between creativity and the safety standards' requirements for documentation and control, which was discussed on page 197 in section 6.3. The two companies simply solve this conflict by keeping creative innovation apart from the safety critical process. The first company employs 800 people and makes software for car controllers[33] based

[32] Interview 10.

[33] Electronic Control Units (ECUs).

on a software platform for the car industry called AUTOSAR,[34] which is jointly developed by car manufacturers and others.[35] Most of the requirements for the products come from the specification for AUTOSAR, which changes regularly. The work process is heavily supported by software tools. The company only does certification when demanded by customers, and only on well-defined components – if new features are needed, a technical solution is found before safety is addressed, as one manager explains:

> "If we are discussing a *new* feature which requires a new technical solution, maybe as an additional product component, then at a first stage we are trying to solve the technical matters, and then we are going to assure that this will fulfil all the [safety] requirements ... we are not happy with doing it the other way around, which means definition of a big process, then breakdown of requirements, and in the end doing a technical solution."[36]

Safe River is a consulting firm of 10 employees in the aeronautics and railway industries.[37] The consultants work and participate in Safe River's customers' work processes where they help the customers to use formal methods, a collection of very demanding and costly methods that are only used for the highest levels of safey critical categorization. The founder of Safe River explains that the innovative part of doing a new product should be kept apart from the work process that conforms to safety standards:

> "Suppose the system is completely new and you don't have any experience and so on – you must study and do the proof of concept *before* you enter the [safety] process itself."[38]

[34]Automotive Open System Architecture.
[35]Interview 18.
[36]Interview 18 ~00:55:23.
[37]Interview 21.
[38]Interview 21 ~00:55:11.

However, she emphasizes that even if the creative part takes place before the safety process is engaged, it is necessary to at all times be aware of the safety requirements that the product must eventually fulfil:

> "You have some phases which are more experimental when you must do the proof of concept, but some people do not take into account safety constraints at this stage and afterwards, when they go to the real development phase, there are conflicts between the constraints, which have not been taken into account, and the proof of concept itself, and in this case it can be very very expensive to go back to the first phase."[39]

This last comment shows that although it is in principle a feasible form of practice to separate innovation and fulfilment of the safety standards, it is not always so easy to do in practice.

The final example given here is an interesting hybrid between the companies that conform to the safety critical standards and the companies that avoid or work around them in some way or another. Kone is a company of 33,000 employees that makes and installs elevators worldwide.[40] The 85 employees in the software department makes the controllers for the elevators. Kone uses an Agile methodology, Scrum, for projects in which it develops something completely new. For projects that add features to existing software, and for safety critical projects, Kone uses a traditional iterative waterfall approach. The desire to use an Agile process came from the software developers rather than managers, which is unusual: the company's software developers normally do not initiate process changes themselves.

Interestingly, Kone combines two forms of safety programming that we have otherwise seen used in a mutually exclusive way in separate companies: an Agile form that does not conform to safety critical standards, and a software engineering form that does. In Kone these forms exist side by side, not only within the same company, but within

[39] Interview 21 ~00:55:11.

[40] Interview 29.

the same department. This example illustrates a point made by the economist R.H. Coase in his article "The Nature of the Firm": that the exact delineation of which tasks belong within one and the same company is, in essence, an economic and therefore a cultural question.[41] That is: what constitutes "a company" cannot be taken for granted; it is always possible that some task within the company is better left to a subcontractor, and conversely it is always possible that two smaller companies could better be combined into a single entity.

When, in the previous section, we looked at safety critical programming as a form in contrast with the game programming within Tribeflame, we perceived safety critical programming as a fairly homogenous form of culture with distinct features. In this section we have taken a closer look at safety critical programming forms and seen that, even within this specific form of programming, there is ample diversity in approaches. This reveals that while it is possible to identify some general traits of programming, it is equally important to be aware of the context, because it is not possible to identify the form of an example of programming without taking the context into account.

We have also seen how the same form can appear in vastly different circumstances, such as in Skov and in the small German company that makes timing analysis software tools; both employ an Agile form of programming, but while Skov operates in an industry without safety critical standards, the other company operates in the automotive industry, which is heavily regulated by standards. Also, we have seen examples of companies that operate in similar circumstances but choose different forms to survive in those circumstances: Cassidian, which has a very centralized process form in which one standard is made to fit all processes, and Danfoss Power Electronics, which considers it inefficient to make one process fit all software product lines.

[41]Coase 1937.

Chapter 8

Rhetorical case study and analysis

8.1 *Rhetorics and programming*

In the previous chapters, we have been primarily examining programmers' working processes: that is, how they do their job. In this chapter, which concludes the analysis of programming presented by this treatise, we take a closer look at the programmers' primary work product: program code.

In chapters 5.2 (Analysis) and 6 (Safety critical programming) we saw how hermeneutical theory can be used to understand programming processes. In chapter 7 (Cultural form analysis) we used cultural form theory to look at the differences between programming in different situations. As we saw, these forms of theory are well suited to study programming in order to understand better what happens in the programming process. However, the ultimate goal of understanding programming better is that we might be able to program better; and in order to do that, our understanding of programming needs to be applied in practice to programming tasks.

For that reason, we will in this chapter apply *rhetorical theory* to an analysis of program code. Program code is the medium in which programmers work, and is therefore of interest to a practical approach to programming tasks. Rhetorical theory is the classical body of learning about how to formulate and deliver a speech in practice. The practical element of rhetorics means that it is a more direct route to practical application than hermeneutics, which is more concerned with understanding texts than with creating them.

A rhetorical analysis of program code should be seen in contrast to the conventional forms of analysis that are associated with the thinking described in section 3.2 (Mainstream programming theories). We did not, in that chapter, go into detail about the ways program code is analyzed, but it is in line with the overall priorities of mainstream programming theory and usually focuses on mathematical properties, quantification, efficiency, program organization and modularization, and business value. As we shall see, rhetorical analysis provides a perspective on source code that is quite different from all these, yet is closer to the practical everyday experience that a programmer has when working with code.

The relationship between rhetorics and hermeneutics is close. Essentially, rhetorics is the practical side of hermeneutics, as the philosopher of hermeneutics Hans-Georg Gadamer has explained in a 1976 lecture about rhetorics and hermeneutics. It is a central principle of hermeneutics that understanding depends on application – that is, a cultural phenomenon can only be understood correctly through understanding its purpose. Corresponding to this principle is, in rhetorics, the notion that communication should be purposeful. An act of communication must be judged according to whether it achieves its intended purpose. The theories of hermeneutics and rhetorics together constitute a philosophy that is symmetrical around the concepts of application and purpose; whereas hermeneutics is primarily concerned with understanding, rhetorics is primarily concerned with applying knowledge.

Applying a rhetoric perspective to programming is a rarity in the literature. In his 2002 book *The Art of Interactive Design*, computer game designer Chris Crawford regards programming as essentially a matter of interactivity, which he defines as: "a cyclic process in which two actors alternately listen, think, and speak".[1] He employs additional perspectives on programming, but his main metaphor is that of a conversation. The use of conversation as an image of programming, and its division into three species – listening, thinking, and speaking – places his work in the domain of rhetorics even though his starting point is not classical rhetorical theory but rather a non-academic and very practical approach.

Popular handbooks in practical programming, such as *Code Complete*, provide much advice that can easily be understood in a rhetorical way. Examples in *Code Complete* are an explanation of the importance of metaphors in understanding programming, advice on how to structure code so that the meaning is clear to a reader, and advice on how to use layout and indenting in order to communicate the intent of the code better.[2] None of this is presented in connection with rhetorical theory, but rather as things that are self-evidently good to do. Nevertheless, these topics fall within the domain of rhetorics.

Bryant speaks for an approach that, while not exactly rhetorical, can easily be understood in rhetorical terms: he argues that metaphors for understanding are a central part of software engineering.[3] Metaphor is a central concept not only to literature but also to rhetorics, and speaking of metaphors as playing a part in how we perceive things, as opposed to merely a way of making speeches more colourful, fits well with the approach of Perelman and Olbechts-Tyteca, called "new rhetorics".

[1]Crawford 2002 p. 5.
[2]McConnell 1993 chp. 2, chp. 13, chp. 18.
[3]Bryant 2000.

8.2 *Audience*

A fundamental concern of rhetorics is the audience for an utterance. This raises the question of who the audience is for a piece of program code. Ultimately, the audience for a computer program is of course the intended user. However, much as a playwright does not manipulate a theater play directly but rather by instructing the play's actors through a manuscript, the programmer does not manipulate the program's behaviour directly but rather through instructions to the machine that will be executed later. The theater manuscript and the program code are both indirect representations of acts of communication that will be realized later. The theater manuscript is to be read by the actors, not the theater audience, and therefore it has to be written for the actors rather than the audience.

Thus although the audience for a program is the eventual user, the audience for program code is other programmers. Who, then, are these other programmers? First of all, the programmer needs to write his code so that it is understandable by his colleagues. It is a very real possibility that a programmer other than the author will have to make modifications to the code at some point, perhaps at a time when the author has left the company and is not available to answer questions; the recipient colleagues may even be future colleagues, not presently known to the author. This means that the audience of programmer colleagues, for whom the author is writing code, is a case of what Perelman and Olbrechts-Tyteca call "the universal audience";[4] and in the case of program code we can call the audience "the universal programmer". The universal programmer is the author's idea of what a reasonable and intelligent programmer would be like – writing code to be read by the universal programmer means writing code that is orderly and proper, "the way it should be".

One more programmer is going to read and understand the program code and that is the author himself. Consequently, when the author writes the program code he thinks not only of how poten-

[4]Perelman & Olbrechts-Tyteca 1958 §7.

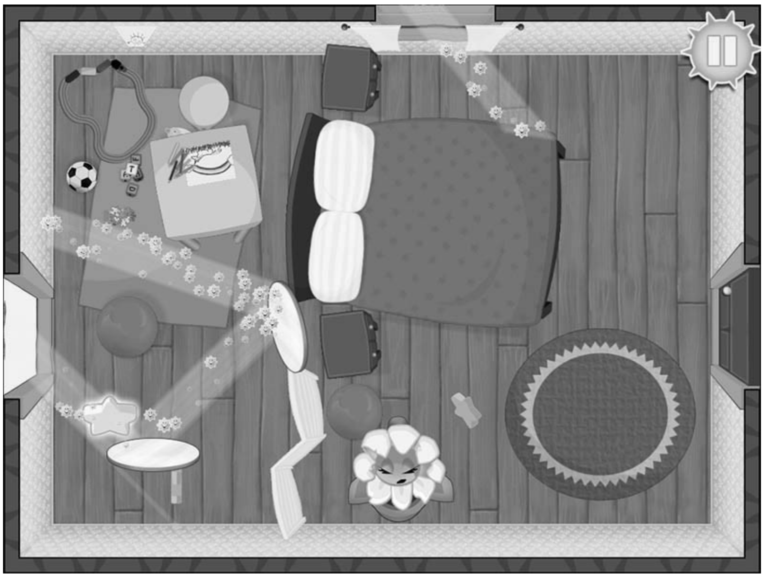


Figure 8.1: Tablet computer game with the working title Flower. *Tribeflame 2014.*

tial future colleagues or the universal programmer might perceive his code, he also deliberates with himself on how to best understand the code he is writing. According to Perelman and Olbrechts-Tyteca, the private deliberations one silently has with oneself is a special case of general argumentation, and can be understood as yet another address to the universal audience.[5]

We notice that the members of the audience for the program code – the author himself, his colleagues, and the universal program-

[5] "L'individualisme des auteurs qui accordent une nette prééminence à la façon de conduire nos propres pensées et la considèrent comme seule digne de l'intérêt du philosophe – le discours adressé à autrui n'étant qu'apparance et tromperie – a été pour beaucoup dans le discrédit non seulement de la rhétorique, mais, en général, de toute théorie de l'argumentation. Il nous semble, par contre, qu'il y a tout intérêt à considérer la délibération intime comme une espèce particulière d'argumentation." Perelman & Olbrechts-Tyteca 1958 [2008] §9, p. 54.

mer – all are included because they might modify the program code in the future. Recalling the analysis in chapter 5.2 (Analysis), this is because program code is seldom written in one go, never to be modified again. If this were the case, program code could be written in any way accepted by the machine, and it would not matter if the code were neat or understandable. But program code is written to be modified; and this is because of the hermeneutical process of writing the code, where adjustments are carried out and errors corrected in a slowly evolving hermeneutical circle that moves in cycles between program creation and experience with the program's behaviour – see section 5.2.2 (Hermeneutical analysis).

8.3 *Programming conventions*

Figures 8.2, 8.3, and 8.4 show parts of the program code for a computer game for tablet computers made by the company Tribeflame. This game, which they worked on during my time observing the company, had the working title *Flower.* The game shows a living room in a house seen from above, as shown in Figure 8.1. The living room is dark, and in the corner, amidst chairs and tables, stands a neglected, sad-looking flower. A ray of sunshine falls through the living room window but does not reach the unfortunate flower. The player's task is to place one or more mirrors in the living room and make sure that they reflect the ray of light in such a way that the sunlight hits the flower and restores its health and happiness.

The program code file shown in figure 8.2 is named "Game-Scene.hpp". It is a so-called header file. A header file does not itself contain any functional program code; it merely lists the contents of another file. Technically, a header file is not strictly necessary for the program to function, but it serves some practical purposes and has come to be expected in any reasonably-sized program. This means that the use of a header file has become a rhetorical convention – much like a list of contents in a book. The mere presence of a header

```
#ifndef GAME_SCENE2_HPP
#define GAME_SCENE2_HPP

TF_DECLARE_CLASS( Light );
TF_DECLARE_CLASS( Mirror );
TF_DECLARE_CLASS( Room );
TF_DECLARE_CLASS( Flower );

#include "UI/Scene.hpp"
#include "Util/Geometry.hpp"
#include "Levels/LevelInfo.hpp"
#include "Game/Updater.hpp"
#include "Game/RoomDefinitions.hpp"

TF_CLASS(GameScene) : public Scene, public tf::TouchClientMixin {

public:

    /**
     * Creates an instance of the scene.
     *
     * @return a new instance.
     **/
    static sGameScene create (sRoom room, sLevelInfo level_info);

    GameScene ();

    ~GameScene ();

    void init ();

    /**
     * Signal used to tell that the game scene is done, i.e. the
         game has finished through the user quitting it manually
     **/
    boost::signals2::signal<void ()> quitSignal;
    boost::signals2::signal<void ()> nextLevelSignal;

protected:
```

Figure 8.2: Part of the C++ header file "GameScene.hpp" from the tablet computer game with the working title Flower. *Only the start of the file is shown.*

file, therefore, does not tell us a lot about the game; but the specific way it is written, its style and arrangement, can tell us something about the priorities of the game's creation.

First of all, the lines:

```
TF_DECLARE_CLASS( Light );
TF_DECLARE_CLASS( Mirror );
TF_DECLARE_CLASS( Room );
TF_DECLARE_CLASS( Flower );
```

and

```
TF_CLASS(GameScene)
```

are not written in standard C++ code. They are macros written by Tribeflame's programmers, as the letters "TF", standing for "Tribeflame", show. They indicate that not all standard conventions of writing C++ code are preferred by the programmers.

The line

```
TF_CLASS(GameScene) : public Scene, public tf::TouchClientMixin {
```

tells us that the program module GameScene is related to the more general module Scene, meaning that there is more to the computer game than just the game itself – there are modules, for example, for selecting a level, showing the high score, and changing the settings of the game. The bit

```
tf::TouchClientMixin
```

tells us that the game works on a tablet computer ("`TouchClient`"). It also tells us that Tribeflame has made its own collection of modules that can be used for developing tablet computer games – again "`tf`" stands for "Tribeflame". It would be possible to make a game without such a collection of modules. This is a more direct technique, but it would result in more disorderly program code. Tribeflame has spent quite a lot of energy on making orderly program modules that makes it possible for them to run games on different brands of tablet computers, in addition to the most common, Apple's iPad. The line

```
public:
```

marks the part of the code that is available to other program modules. This means that the following lines show the most important parts of this program module in terms of interaction with other parts of the code. This particular program module is not meant to do more than to be started and then run the computer game. The lines

```
GameScene ();

~GameScene ();
```

are the conventional ways in the programming language C++ of starting and ending a program module. The line

```
static sGameScene create (sRoom room, sLevelInfo level_info);
```

is an alternative way of starting a module that Tribeflame has chosen instead of the C++ convention. This shows us that the design principles upon which C++ is built, so-called Object-Oriented Programming, is not in this case attractive to Tribeflame, though the company uses the principles in other places in the program code. It does not matter to the machine where in the file this line is placed.[6] By placing the line first in the program code, the programmer emphasizes, for the benefit of the human reader, that the program deviates from the C++ conventions in this way.

The only parts of the programming module that are accessible to other modules and do not have to do with starting and stopping the module are the lines

```
boost::signals2::signal<void ()> quitSignal;
boost::signals2::signal<void ()> nextLevelSignal;
```

These lines both have to do with the player controlling the game by quitting it, or moving on to another game level. The prominence of these lines show that the player's ability to control the game is an important part of the design of the whole game. A part of the entertainment value of the game is that the player should be able to enjoy it in a way he chooses himself.

[6]Within some limits.

8.4 *Tone and purpose*

When doing a rhetorical analysis of a text it is often helpful to determine the *exigence* of the text. In rhetorics, exigence means a situation that demands an answer.[7] One comparable example of exigence is a judge's accusation of a defendant, which demands that he answer the charge in order to defend himself. The exigence for the program code as a whole is the company's decision to develop a game that can succeed commercially. However, each program module has an exigence of its own. The exigence for the module GameScene is to organize the code that has to do with displaying and interacting with the game, and to provide a starting point for running the game.

Even smaller parts of code inside a module have their own exigence. Frequently, each single line of code has an exigence of its own and sometimes the programmer records the exigence in comments in the code. At other times, the programmer records the exigence in a version control system, a program that keeps track of all changes made to the code. Examples of comments entered in a version control system are "bonus items",[8] "linux compat[ibility], do not log if NDEBUG is defined",[9] and "added some debug code for a single sun showing up on the first ray every time we update".[10] When using a version control system, the exigence for a given line of code and for all changes made to it can be found. This is an important tool for programmers whenever they try to understand some code, which is not surprising since exigence is often essential to understanding the meaning of a text.

In a rhetorical analysis, the matter of concern is how purposeful program code is – which is determined not by technical criteria, but by *its meaning.* Figure 8.3 shows the start of the program file "GameScene.cpp", the file that containes the functional code which is listed in the file "GameScene.hpp", which we examined above. This file,

[7] Kennedy 1984 p. 35.

[8] Field diary, 30th August 2011, 15:47.

[9] Field diary, 31st August 2011, 16:08.

[10] Field diary, 31st August 2011, 16:22.

```
#include <Flower.hpp>

#include "Game/GameScene.hpp"
#include "Game/Mirror.hpp"
#include "Game/Room.hpp"
#include "Game/Light.hpp"
#include "Game/Flowers/Flower.hpp"
#include "Game/Sensor.hpp"
#include "Game/LevelCompletedNode.hpp"
#include "Game/Obstacles/Obstacle.hpp"
#include "Game/Obstacles/Animal.hpp"
#include "Game/Obstacles/Firefly.hpp"
#include "UI/MenuButton.hpp"
#include "Textures/Textures.hpp"
#include "Util/Audio.hpp"
```

Figure 8.3: Part of the C++ program file GameScene.cpp from the tablet computer game with the working title Flower. *Only the start of the file is shown.*

like the last one and indeed almost all C++ files, starts with a list of other files to be included ("`#include`"). Technically, this means that the contents of the other files are "copied" to this file, so that their contents are accessible to the program code. We could go more in detail with how this is accomplished and what technical consequences it has. However, for the purpose of rhetorical analysis it is more interesting to see what the inclusions can tell us about the program's meaning.

The first inclusion of the module "`Flower`" signifies that the file is part of the *Flower* game. The rest of the inclusions indicate which modules and which parts of the game are conceptually most important. To analyze them, we can use the concepts called the "three species of rhetorical expression". These are simple but effective categories for classifying the "tone" of an expression.[11] The historical

[11]The rhetorical species are seldom found in pure form.

names for the species are *judicial* speech, *epideictic* speech, and *deliberative* speech.[12]

Judicial speech is that which has to do with the past. It is often concerned with recording and stating what has happened, and with passing judgment on past actions. It is so named because the archetype for judicial speech is the speech of accusation or of defence offered in a court of law. Epideictic speech has to do with the present, and is often concerned with setting the mood and stating values. Epideictic speech has been a little overlooked in modern times: its archetype is the speech of praise delivered at a public ceremony, such as, for example, a graduation speech. Deliberative speech has to do with the future. It tries to convince the audience to make a choice or to take a certain action, and its archetype is the political speech aimed at convincing voters.

Looking at the inclusions, most of them are primarily in the epideictic species. "`Mirror`", "`Room`", "`Light`", "`Flowers/Flower`", "`Sensor`", "`Obstacle`", "`Animal`", "`Firefly`", "`Textures`", and "`Audio`" are all modules that have to do with establishing the "present" of the game, with setting the right mood for the player and presenting him with a game that looks interesting. That so many inclusions are in the deliberative species indicate that constructing an interesting "present", a game ambience, is a task that demands much work.

Only one inclusion seems to be squarely in the judicial species: "`LevelCompletedNode`". Though the game records certain things – the "history" of the game in the form of a high score list – the game is not a program that is primarily oriented toward recording and judging the past, as for example a database or an accounting system would be. The primary use of records and judgement in this game is to restrict what levels the players can access.

The deliberative species is represented by "`Mirror`" and "`MenuButton`". These are the things in the game that the player can manipulate. If the sheer amount of code is counted, these elements do not seem to be as important as the epideictic elements. However, while they might not require huge amounts of code, it is important

[12]See e.g. Kennedy 1984 p. 19, Perelman & Olbrechts-Tyteca 1958 [2008] §4, p. 28.

that the few lines required are just right. This is because the whole point of the game is to present the player with a situation that can be manipulated, an interesting choice. If this dimension is missing in a game, it becomes less of a game and more of an interactive movie or animation.

The program code shown in figure 8.4 shows the part of "Game-Scene.cpp" that is run whenever a new game is started – "`init`" stands for "initialization". The line

```
Scene::init();
```

again is a sort of rhetorical convention in C++. The next statement

```
game_state = State::Playing;
```

simply records that the game is now playing. The first thing that happens that is noticeable to the player is that the game music is started with the line

```
audio->setMusicType( Audio::GameMusic );
```

This shows that the music is very important to set the ambience of the game. Music creation is one of the only creative functions in Tribeflame that is bought from an external provider. The task is not big enough to warrant a regular employee, but it is still of such importance to get music of high quality that it is necessary to hire a professional.

The next line

```
//tf::Flurry::get_flurry_instance()->start_timed_event( game_\
    mode->getLeaderBoardId() );
```

is supposed to assist with keeping a record of the player's activities. However, we notice that the line starts with the characters "//" which means that it is a comment – from the computer's perspective it is as if the line had been deleted. So why has the programmer not simply deleted the line?

There are several reasons the programmer might like to keep some code as a comment. A primarily technical reason would be to make the line a comment for a short time, try out the program, and then

```
void GameScene::init () {
   // superclass initialization
   Scene::init();

   // now we're again playing
   game_state = State::Playing;

   // play game music
   audio->setMusicType( Audio::GameMusic );

   // report it as a statistic too
   //tf::Flurry::get_flurry_instance()->start_timed_event( game_\
       mode->getLeaderBoardId() );

   // now we want touch events, this allows the player to immedi\
       ately "tap to continue"
   set_enable_touch_dispatch( true, 10 );

   // first add the room
   add_child( m_room );

   sGameScene self = TF_GET_SHARED_THIS( GameScene );
   tf::signal_weak_connect( m_room->obstaclesMoving, boost::bind(
       &GameScene::obstaclesMoving, self.get(), _1 ), self );

   // create a button for an in game menu
   sMenuButton quit_button = TF_VAL( MenuButton, getMenu(),
       tp_game_quit_button, tp_game_quit_button_pressed,
       tf::Point2F( 0, 0 ) );
   quit->button->init();

   // left corner
   tf::Size2F size = tf::get_screen_native_bounds();
   quit_button->set_position( -1024 / 2.0 + 25, 768 / 2.0 - 25 );
   tf::signal_weak_connect( quit_button->signal_activate,
       boost::bind( &GameScene::quit, self.get()), self);

   // initial ray
   updateLights();
}
```

Figure 8.4: Part of the C++ program file "GameScene.cpp" from the tablet computer game with the working title Flower. *Only part of the file is shown.*

uncomment the code again. There are also more rhetorical uses of code as comments. One is that the programmer has begun to create some functionality in the code but decided to postpone it to a later time, when the commented code will serve as a reminder of what the programmer originally thought up or, if another programmer has taken over the code, it indicates what the programmer thinks is a good way to solve the problem. Another use is essentially the opposite: the code might be kept around as a comment to show that a particular point has been considered, tried out, and then rejected. This might save the programmer or his future colleagues some trouble later on. In both cases, the commented code serves a purely rhetorical purpose since it has no effect on the program's behaviour.

The next line

```
set_enable_touch_dispatch( true, 10 );
```

makes the game responsive to the player's gestures. That this happens before the game is even fully prepared for playing suggests that letting the player manipulate the game has high priority, as wmentioned above.

The line

```
add_child( m_room );
```

establishes the "room", meaning a game level containing a puzzle for the player to solve.

The lines

```
sGameScene self = TF_GET_SHARED_THIS( GameScene );
tf::signal_weak_connect( m_room->obstaclesMoving, boost::bind(
    &GameScene::obstaclesMoving, self.get(), _1 ), self );
```

establish a connection between the game and the room. In the first line, a variable called "`self`" is declared with the help of a macro of the company's own called "`TF_GET_SHARED_THIS`". With this line the company again circumvents the rhetorical programming conventions of C++. The programming language has a built-in mechanism called "this", but the programmers have chosen not to use it and instead make their own, similar mechanism called "self". "Self" does not do exactly the same thing as "this", but the name seems to be chosen in order to emphasize the similarities between the two mechanisms.[13]

The lines

```
sMenuButton quit_button = TF_VAL( MenuButton, getMenu(),
   tp_game_quit_button, tp_game_quit_button_pressed,
   tf::Point2F( 0, 0 ) );
quit->button->init();
```

and

```
tf::signal_weak_connect( quit_button->signal_activate,
    boost::bind( &GameScene::quit, self.get()), self);
```

create a button on the screen that the player can press in order to control the game. The lines

```
tf::Size2F size = tf::get_screen_native_bounds();
quit_button->set_position( -1024 / 2.0 + 25, 768 / 2.0 - 25 );
```

determine where on the screen the button is placed. Interestingly, in the first line a variable called "`size`" is declared, which is never used. Instead the calculation "-1024 / 2.0 + 25, 768 / 2.0 - 25" is used to determine where the button is placed. Apparently, the programmer first thought of using "`size`" in this calculation but then for some reason abandoned this approach. That "`size`" is left in place in the code might either be deliberate, as in the case of the commented code above, or it might be an oversight. In either case the rhetorical effect on the reader is to indicate that this small piece of code is probably not completely finished.

The final line of this part of the code is

```
updateLights();
```

which has the effect of letting a ray of sunshine shine through the window in order to begin the game.

[13]In the programming language Smalltalk, the mechanism that is called "this" in C++ is indeed called "self". Stroustrup 1985 [1997] p. 231.

8.5 *Rhetorical understanding*

This chapter provides an example of how to approach a rhetorical analysis of program code. Several rhetorical concepts have been used in the analysis and more could be added: The rhetorical concept of audience.[14] The concept of rhetorical convention.[15] The importance of choosing the right unit as a starting point for the analysis: whole program, program file, program module, or a single line of code.[16] The concept of *exigence*.[17] The three rhetorical species: judicial, epideictic, and deliberative rhetorics.[18] The rhetorical concepts of style and arrangement.[19]

An experienced programmer reading this chapter will have noticed that, apart from using some specific rhetorical terms, what is going on in the rhetorical analysis above is very much like what is going one whenever a programmer reads and tries to understand a unfamiliar piece of program code. This is no coincidence. Classical rhetorics is, properly understood, the study of what people already and always do when they express themselves well – whether or not they have a theoretical understanding of what they are doing. Gadamer writes:

> "Now, rhetorics and hermeneutics are in a certain point closely connected: the ability to speak and the ability to understand are natural human abilities that can be fully developed also without conscious use of learned rules, as long as a natural talent and the proper care and use of it coincide."[20]

[14]Perelman & Olbrechts-Tyteca 1958 §4.

[15]See e.g. the discussion of rhetorical figures in Perelman & Olbrechts-Tyteca 1958 §41.

[16]Kennedy 1984 p. 33.

[17]Ibid. p. 35.

[18]Ibid. p. 19.

[19]Kennedy 1984 p. 13. Perelman & Olbrechts-Tyteca 1958 §29.

[20]"Nun sind in einem Punkte Rhetorik und Hermeneutik zutiefst verwandt: Redenkönnen und Verstehenkönnen sind natürliche menschliche Fähigkeiten, die auch ohne

The point of learning rhetorical theory is, as a matter of course, to become more aware of what we are doing and thereby better at it. The most important lesson from rhetorics is that everything that is written, in program code or otherwise, must be purposeful – and not only this, but it furthermore has to serve the *right* purpose. Deliberations over what purpose a program has and how best to express it is what distinguishes an excellent programmer from a merely competent one.

A lesson to take away from the rhetorical analysis above is the importance of context when analyzing program code. Much of the discussion above depends on information that is not found in the code itself. Because I was present during the development of the code, or at least some of it, it is much easier for me to understand the code than it would be for someone whose only source of information is the code. This observation shows the folly of trying to study program code in isolation, disregarding the context in which it is developed.

The great benefit of using rhetorical concepts in analysis of program code is that they are general, so that the analysis can easily be related also to phenomena outside the code. The advice for program code layout in a handbook such as *Code Complete*,[21] for example, is very specific to just program code, which means that it is hard to relate to bigger themes in the program creation. In contrast, rhetorical concepts like exigence and the rhetorical species are not limited to program code. They can be applied to any form of communication, and this means that all the parts of the software development effort, including meetings, discussions, and the like, can be made to relate to each other within a rhetorical analysis.

bewußte Anwendung von Kunstregeln zu voller Ausbildung zu gelangen vermögen, wenn natürliche Begabung und die rechte Pflege und Anwendung derselben zusammenkommen." Gadamer 1976 p. 8.

[21]McConnell 1993.

Chapter 9

Conclusion

9.1 *Results*

This treatise is directed by three research goals, as stated in section 1.4 (Research problem and results). The first is whether the software development process can be adequately explained by hermeneutical and rhetorical theory. In section 5.1 (Case study) we saw a real-life example of a development process at the small computer game company Tribeflame. In section 5.2.1 (Analysis with mainstream theories), Tribeflame's process was analyzed using the theories and concepts of mainstream programming, and we saw that mainstream programming theories were of limited usefulness. The failure of mainstream theories to explain a real programming process is easily explained by remarking that, although the mainstream theories often are taken to be universally valid, they are in fact of limited perspective and do not apply without modification outside the range of practices for which they were developed.

With this in mind, section 5.2.2 (Hermeneutical analysis) presents a hermeneutical analysis of Tribeflame's process. We see here that hermeneutical concepts are able to explain some features of the process that were difficult to understand using only mainstream theory. First, the process, which initially seemed to be strangely unstructured, turned out to have a definite structure consisting of a continuous oscillation between working alone and working together; between trying out things and reflecting on experience – and we saw that this structure could be explained by the concept of the hermeneutical circle. We also saw how hermeneutical concepts such as authority, tradition, and prejudice could explain the presence of, and relationship between, different parts of the process: leadership, the expertise of the developers, experiences of outsiders trying the game, meetings, and so on. Finally, we saw that the process is first and foremost driven by its application – to make a game that is fun to play – and not so much by any abstract idea of what the process should be like.

To provide a contrast with the particular example of a small computer game company, chapter 6 (Safety critical programming) describes the processes used by some companies in safety critical industries, which make heavy use of mainstream software engineering theory. The hermeneutical analysis of safety critical development processes revealed them to be primarily bureaucratic processes that are as much about producing documentation as about programming. Again, hermeneutical concepts such as authority, tradition, and prejudice proved to be helpful in understanding the processes. The hermeneutical circle is useful for explaining the mutual influence that industry standards and development practice have on each other. Just as game programming is driven by the concept of fun, safety critical programming was seen to be driven by the concept of safety, although a very distinct notion of safety that is deeply connected with the bureaucracy of safety critical standards.

Chapter 8 (Rhetorical case study and analysis) returns to Tribeflame and presents an analysis of a small part of the source code of a computer game. It is shown that the code is not only instructions to the machine but is also meant to be read by people and is therefore amenable to rhetorical analysis. The analysis shows how rhetorical concepts can be used to understand how the code is constructed and how it should be read. Examples of rhetorical aspects of program code that are beyond purely technical analysis include the use of comments, conventions for code, and the purpose behind code lines.

In conclusion, chapters 5, 6, and 8 demonstrate that software development practices can indeed be explained adequately with hermeneutics and rhetorics, and that these forms of analysis can tell us things that escape the narrower viewpoints of mainstream programming theories such as software engineering, Agile development, and computer science.

The second goal of this treatise is to examine whether the differences between different kinds of programming practices can be explained by cultural theory. Chapter 7 (Cultural form analysis) addresses this question. In section 3.4 (Cultural form theory) the concept of cultural form is explained. This is a cultural analytical concept that enables us to carry out a comparative investigation of different forms of programming practice.

In section 7.1 (Comparative analysis), Tribeflame's form of game programming is compared with forms of safety critical programming. We saw that game programming is a cultural form that is primarily subject to market constraints. It is a form that satisfies a demand for creativity and does not demand much communication with outsiders about the details of the work process. Safety critical programming is a form that, in its most common manifestations, is also subject to market constraints, but it is even more pressingly subject to the constraints of a highly regulated bureaucracy of standards. It is a form that conserves tradition to a high degree, and creative expression is carefully regulated by rules. Communication with outsiders plays an important role and is made easier through standardization of the process.

Section 7.2 (Safety critical case studies and analysis) compares different forms of safety critical programming to each other. We saw here that companies in the safety critical industries are far from homogenous. Companies can play many different roles within their respective industries, and there are variations in the strategies employed to deal with the demands of the bureaucracy of standards. This shows that the context of a programming practice is essential to understanding the practice, because different forms can appear in similar circumstances and similar forms can appear in different circumstances.

Taken together, chapter 7 shows that differences between different kinds of programming practice can indeed be explained by cultural theory, and that cultural form is a conceptual tool that can help us understand differences and similarities between practices.

The third goal of this treatise is to examine what the practical implications are of a hermeneutical and rhetorical understanding of programming. This is addresed in the discussion in section 9.2

9.2 *Discussion*

First, it is important to note that philosophical hermeneutics claims to be true of all understanding, which means that it is both epistemological and ontological. This means that hermeneutics is not a method or a theory that will produce a certain result; it is a way of describing what people always do when they understand something, whether they are conscious of it or not. Thus, we can look hermeneutically at what programmers do regardless of whether the programmers themselves know about hermeneutics.

Programming has many different aspects, and we have touched upon some of them in this treatise: economic aspects, technical aspects, management aspects, mathematical aspects, and others. Consequently, there are just as many different ways to approach the study of programming. These aspects are all cultural, insofar they involve human perception and human use of computers. Cultural theory therefore provides a starting point for the study of programming that can encompass all the other approaches. Culture itself can, in turn, be regarded with hermeneutical and rhetorical perspectives – these are ways of looking at cultural expression, not replacements for cultural theory.

Throughout this treatise we have looked at programming phenomena from a hermeneutical perspective. These phenomena include the mainstream theories of programming, and as such these theories have implicitly been judged according to hermeneutical criteria. This means that hermeneutics is used as a kind of "metatheory": a theory that provides a certain view of the world, according to which other theories are judged.

However, it is important to understand that the points made by hermeneutical theory can to the untrained eye seem obvious. This is because many hermeneutical insights align with common sense. Hermeneutics is not some kind of exotic worldview that is incompatible with analytical thinking. On the contrary – it is a call to return to the essentials of science: to observe a phenomenon, then seek to understand it, and finally to test the results of that understanding, which may lead to observing the phenomenon in a new light, and so on. Science is thus a circular process of experiencing and understanding.

It is essential to hermeneutic theory that understanding and practice are inescapably linked and conceptually constitute each other. Likewise, subject and object are inextricably linked and constitute each other; the subjective perspective cannot be defined without referring to the objective one, and *vice versa.* According to hermeneutics, from the time of Descartes philosophy has been mistaken in believing that subject and object can be separated and exist independently of each other. In modernist philosophical thought, subjectivity is typically regarded as epistemology, and the domain of the subject is said to be the domain of action, language, and thought. Conversely, objectivity is seen as ontology, and the domain of the object is said to be the domain of occurrences, observations, and things.

From a hermeneutical point of view, both computer science and software engineering replicate this mistake by declaring a gap between subjectivity and objectivity, and focusing on objectivity to the exclusion of subjectivity.[1] In addition, there is a separation between understanding and practice, because it is thought that true under-

[1]The converse, focusing on subjectivity to the exclusion of objectivity, is exemplified by the romanticist movement.

standing can come about by theoretical speculation that may be informed by practical observations but is not itself intimately connected to practice.

Latour has warned against this belief, noting that theory cannot be properly understood in isolation: "Speaking about theories and then gaping at their 'application' has no more sense than talking about clamps without ever saying what they fasten together, or separating the knots from the meshes of a net."[2]

To be clear: it is not uncommon within computer science to have exaggerated ideas about how much theoretical results can accomplish in industry.[3] On the other hand, it is not uncommon within software engineering to believe that it is possible to control every aspect of the software development process as long as the right engineering principles and theories are applied.

From a hermeneutical point of view, both of these ideas are mistaken and stem from the schism between subject and object that computer science and software engineering have inherited from modernist philosophy. The consequence is that computer science and software engineering thinking has a slightly unreal quality to it, as exemplified in the software paradox described in section 1.1 (Motivation), in which

[2]Latour 1987 p. 242. For a hermeneutical treatment of Latour's insight see section 3.3.8 (Application). Latour's insight is of course equally applicable to hermeneutic and ethnological theory as it is to mainstream programming theory. To assess whether the theories used in this dissertation are suitable to the purpose to which they have been put it is advisable to look at the practices from which they arise.

Gadamer's philosophical hermeneutics stems from philological critique. Its purpose is on the basis of historical studies of academic theology, philology, and history to determine how it is possible to arrive at true insight – truth not in the sense of truth about the physical world, but ethical truth.

Ethnology, in the sense it is used in this thesis, is a part of history. The purpose of ethnology is to preserve knowledge about our forebears' ways of life in order that the purpose of their lives and actions may be understood and not reduced to a symbol of quaint old customs. Ethnological studies of contemporary phenomena have arosen out of the need to understand our own way of life in order to interpret history, see section 3.3.6 (Effective history).

[3]Compare with Naur's viewpoint, discussed in section 3.2.3.5 (Premises of argumentation).

academic software research, despite being hugely successful, is never quite satisfied with its own results.

Agile thinking is different because it preserves a close connection between understanding and practice, and between subjectivity and objectivity. For agents in an Agile project, it is only possible to measure their progress objectively because they have the right subjective approach.[4] Thus, Agile thinking is much more in line with hermeneutical thinking and seems to be an approach whose philosophical foundations are not, in principle, far from the foundations of hermeneutics. The main shortcomings of Agile thinking from a programming research point of view is that, with a few exceptions, it is primarily concerned only with the management aspects of programming[5] and that the movement lacks literature about its philosophical foundation.[6]

The critique that research is not intimately connected to programming practice is not limited to computer science and software engineering. In section 2.2.2 several researchers were criticized on account of their research being speculative: Ihde, Coyne, Manovich, Kittler, and Haraway. There is of course nothing wrong with taking a reflective approach to research, indeed it is a prerequisite for true insight. However, when speculation is not connected to any other form of practice than perhaps the academic practice of reflection, the result can tend to become disconnected from the real world. There is a danger that the technology that is nominally the topic of the research becomes nothing more than a backdrop for musings about the novelty of technology.

[4]Hence, if they do not have the right subjective approach, they might use the Agile rules "wrong": see section 3.2.2.3 (Scrum as an example) .

[5]See e.g. Schwaber 2003: *Agile Project Management with Scrum*, and Coplien & Harrison 2004: *Organizational Patterns of Agile Software Development.*

[6]The so-called "patterns community" is in some way trying to establish a philosophical foundation for Agile thinking, but patterns literature is poorly connected to mainstream philosophical thinking, taking its main inspiration from the architect Christopher Alexander, and the concept of patterns has philosophical problems of its own. Examples of patterns literature are Coplien & Harrison 2004, and Gamma et al. 1994.

This is related to the discussion of ethical knowledge in section 3.3.9. A possible pitfall is that while researchers' speculations make sense in their own practice they neglect to think through what value the act of publication will have to others and others' practices. Thus a necessary ethical deliberation regarding research is omitted. To take the most extreme of these examples: Haraway's goal is ostensibly the promotion of socialist-feminist ideology.[7] To this end, she writes about technology but her approach appears to be cavalier with regard to the practices in which technology is used. Had she instead applied socialist-feminist theory to empirical observations she would most likely have been confronted both with the inherent contradictions in her ideology and its inability to explain concrete practices. Hence, to the extent that her writings actually promote socialist-feminist theory they are in a rhetorical sense technically sound, while being in an ethical sense unsound.[8]

So what is the consequence of these theoretical distinctions for programming practice? The answer is that theory is important for practice. Good habits follow from good principles, and bad habits follow from theoretically unsound principles, such as maintaining an untenable opposition between subjectivity and objectivity. Hermeneutics tells us that theory is unavoidable, in the form of prejudice. If one has "no theory", this does not mean that one has no prejudice – it means that one has a non-conscious, implicit theory that might be unsound, which is hard to discover as long as it remains unarticulated.

The argument made in this treatise is essentially an argument against reductionism. Reductionism is the belief that in every situation, the important aspects can be reduced to a small set of logical principles and rules for how to proceed. Hermeneutics tells us that reductionism is not tenable as a first principle. Reduction is a choice that must be carefully weighted against other choices in the particular situation. This does not mean that thinking cannot be stringent: a true hermeneutic analysis, for example, can never be reductionist,

[7]See page 41 in section 2.2.2 (Sociology, anthropology, and other).
[8]See section 3.3.9 for the distinction between ethical and technical knowledge.

but it must be stringent. It does mean, however, that practice should always include some form of deliberation over goals and application.

The critical contribution of hermeneutics in this respect is that it points out how very important tradition is in evaluating the goals of a practice. Looking at tradition is a way of discovering the original intentions of the practice, and these original intentions will heavily influence the theories and assumptions of a practice – as demonstrated, for example, by the heavy reliance of safety critical programming on assumptions that stem from the engineering tradition of regulating practice by standards.

When we look at the hermeneutical analysis of game programming and safety critical programming in chapters 5.2 (Analysis) and 6 (Safety critical programming), we notice that the concept of effective history is the hermeneutical concept that plays the smallest part in any of the programming processes. This is no coincidence. Effective history is a concept that expresses awareness that knowledge is historical and that the motives for knowledge have consequences. Since mainstream programming theory is to a large degree based on modernist philosophy, it is also somewhat ahistorical, which makes it difficult for its adherents to assess its own history. Software engineering and computer science in particular are ahistorical to the degree that they embrace universal principles rather than historical context.

Historical consciousness is important because it underpins the values that are embodied in a company's culture. Leslie Perlow's sociological study of corporate culture among software engineers shows how short-sighted values can be detrimental both to productivity and to the employees' satisfaction with their work.[9] In an experiment with trying to change software engineers' use of their work time, she has also shown how changes in routines and behaviour have no lasting effect unless the underlying values are changed.[10]

What, then, do programmers need to do in order to be able to take advantage of the insights of hermeneutical theory? They need to learn to see themselves as cultural agents: that is, as part of, and

[9]Perlow 1997.

[10]Perlow 1999.

carriers of, historical tradition. They also need to understand that a large part of their work consists of interpretation – interpretation of code, requirements, situations, and what it means to be a good programmer.

The comparative cultural analysis of chapter 7 (Cultural form analysis) shows us that all understanding of programming has to take the context into account. This means that there are no universal principles for programming that are independent of the goals of the work. Programming involves making trade-offs and choosing what is best in the current situation; and what is best in a situation cannot be determined beforehand with the knowledge of only a limited set of parameters. It follows from this that part of the work as a programmer is to figure out what the work actually consists of, what the real goals of it are, and how these are effectively pursued. The programmer's ability to do this can be divided into three aspects:

- Technical ability – programming skills.
- Hermenutical ability – ability to interpret and understand.
- Rhetorical ability – ability to express oneself properly.

This begs the question: if programming always has to be seen in a cultural context and therefore has no universal principles, is it even possible to say something general about programming, as this treatise tries to do? It is. The key to this is recognizing that programming is rhetorical in its essence. Like a speech, a program is a form of expression, and it is impossible to give a universal method for how to write a good speech or a good program – it depends on the audience.

However, that does not mean that we cannot learn something about the general principles that underpin programming expression – by studying good programs, for example. Hermeneutics and rhetorics give us some conceptual tools to understand programming practice and help us discriminate and find out what we can learn from a given programming example.

A consequence of the conclusions drawn above is that that there are limits to what research and theory can do for practitioners. A scientist who is far removed from actual practice cannot solve every practical problem – practitioners understand their problems better than do academics and researchers. With this insight, we have come full circle to the story of the Tower of Babel that Brooks use as an explanation in his essays on software engineering.[11]

Brooks views the Tower of Babel as an engineering project that failed because of lack of communication. Hermeneutic theory shows that Brooks is wrong in his interpretation, and that better communication or engineering principles would not have allowed the "engineering project" to succeed: the tower fell because mankind is fundamentally limited in our knowledge and abilities, and this is the ontological aspect of hermeneutics. God's destruction of the Tower was not revenge but a simple acknowledgment of this limitation: man is not God and therefore cannot hope to have perfect knowledge of anything. We have only imperfect knowledge.

Imperfect knowledge is the domain of interpretation and hermeneutics. The goal of this treatise has been to demonstrate the cultural, hermeneutical, and rhetorical nature of programming. I have made that demonstration by example: through applying cultural, hermeneutical, and rhetorical analysis to a range of programming practices. The idea has been to show how it is done, and in order to do so it is necessary to know the theory.

9.2.1 *Ethnological significance*

From an ethnological perspective, there is nothing particularly innovative in the methodological approach of this dissertation. As has been pointed out in section 4.4 (Ethnological method), both the methods used and the bulk of the theory are well-established within ethnology and used within contemporary research. As stated in section 2.2.1 (Ethnology), the dissertation can be seen as a continuation of a long

[11]See section 3.2.1.2 (Brooks).

tradition of ethnological work studies. This thesis is thus essentially a piece of classical ethnology that brings no innovation in terms of method or theory.

The ethnological significance of the dissertation, then, is the light it sheds on the work of computer programmers. It is obvious that increased knowledge of programming is useful to programmers themselves. However, it is also useful to ethnologists. As explained in section 1.1 (Motivation), computer programming is an important part of contemporary society. Ethnology seeks to understand people's lives and consequently it is important to understand the professions of contemporary society.

Another and more theoretical contribution of this dissertation to ethnology is the perspective that is applied to programming. As explained in section 2.2 (Cultural research), a large amount of cultural research on programming focuses more on the end users perspective and on perceptions of programming than on the work itself. Even when the work is addressed, it is often in terms of economical or organizational significance rather than the practice of work. This dissertation provides a model for studying work practice that has the potential to be useful in general in cultural studies of new technology.

9.3 *Validity and generalization*

It is not the aim to quantify or study what is typical within programming. The aim has been for a certain amount of cultural variation in the situations studied, but it has not been attempted to cover the range of possibilities in a systematic way. Every cultural situation is unique in the sense that what happens depends on the free will of the persons involved. This means that it is not possible to make general rules that can say with certainty what people will do in a given situation – not even probabilistic rules are very useful. In other words, cultural studies are historical studies, even if they are only concerned with the present.

Another reason to avoid quantification at this stage is that quantification presupposes a notion of what counts as a significant difference. It is impossible to count situations without an idea of what makes a situation different from the next. The trouble is that every cultural situation is different from the next in myriad ways, and the question of which ways are significant depends on the observer as well as his purpose with the observation.

Construct validity is the extent to which "the operational measures that are studied really represent what the researcher have in mind and what is investigated according to the research questions."[12] In other words: do the researcher's concepts actually reflect reality as represented by the data? The point of interpretive research such as this dissertation is to apply concepts as an interpretation in order to arrive at an perspective that brings new insight. The construct validity is therefore predicated on two questions: is the insight new, and does it agree with the data? That the interpretation agrees with the data is demonstrated in the various analyses of the dissertation in sections 5.2.2 and 6.3, and in chapters 7 and 8. That it is new is demonstrated by comparison with the analysis carried out with mainstream theories in section 5.2.1. The comparison with the mainstream theory analyses further supports that the concepts agree with the data and thus the construct validity of the dissertation.

Internal validity is the question of whether causal relations and confounding factors are understood correctly in a study.[13] This dimension of validity is not applicable since this dissertation is concerned not with causal relations but with hermeneutical and cultural relations.

The external validity is the extent to which it is possible to generalize the findings to situations outside the study.[14] The insights into specific traits of the studies processes, e.g. the exact forms of the processes of Tribeflame, Cassidian, and Skov, are of course not generalizable. The insights into the workings and forms of the game and

[12]Runeson & Höst 2009 p. 153.
[13]Ibid. p. 154.
[14]Ibid.

safety critical industries are generalizable to the respective industries, however, with research into a larger number of relevant case studies the findings could be made more precise. The insight into programming in general is presented in the from of application of the general concepts of hermeneutics, cultural form theory, and rhetorics. These concepts and their application are generalizable by design. Thus, by following the example of this dissertation it is possible to apply the theories to other case studies in programming. This of course necessitates knowing the theories in question as well as having the ability to accomodate unforeseen phenomena and findings – in short, the ability to perform independent, critical scienctific studies.

Reliability is the extent to which a study can be repeated by other researchers and still give the same result.[15] This kind of validity applies to controlled experiments, but it is not applicable to explorative and interpretive research since the latter by its nature generates new knowledge. A hypothetical study replicating that knowledge would therefore not achieve construct validity according to the criteria of bringing new insight (see above).

Triangulation has been employed in various ways to increase the validity of the research. Runeson and Höst list four types of triangulation: data triangulation, observer triangulation, methodological triangulation, and theory triangulation.[16] Of these, three has been employed. Data triangulation has been employed by collecting data from both game programming and safety critical programming; also by collecting data from a range of companies in the safety critical sector. Methodological triangulation has been acheived by utilizing both participant observation, semi-structured interviews, and also collection of work documents, primarily source code. Theory triangulation has been employed extensively by using multiple theories for analysis: hermeneutics, cultural form theory, and rhetorics, as well as analysis with mainstream programming theories.

[15]Ibid.

[16]Ibid. p. 136.

Bibliography

Abrahamsson, Pekka, Outi Salo, Jussi Ronkainen, and Juhani Warsta 2002. *Agile Software Development Methods: Review and Analysis.* VTT Publications 478. VTT Technical Research Centre of Finland. Espoo. ISBN 951-38-6010-8.

Agile Alliance 2001. *Manifesto for Agile Software Development.* http://agilemanifesto.org.

Aiken, Howard H. 1964 [1975]. "Proposed Automatic Calculating Machine." Previously unpublished memorandum. *IEEE Spectrum* 62-69. In: Brian Randell (editor) 1975. *The Origins of Digital Computers: Selected Papers.* Second edition. Springer-Verlag Berlin Heidelberg New York. ISBN 3-540-07114-8.

Alexander, Christopher 1964. *Notes on the Synthesis of Form.* Harvard University Press. Cambridge, Massachusetts.

Aranda, Jorge 2010. *A Theory of Shared Understanding for Software Organizations.* PhD thesis. Graduate Department of Computer Science, University of Toronto.

Auvinen, Jussi, Rasmus Back, Jeanette Heidenberg, Piia Hirkman, and Luka Milovanov 2006. "Software Process Improvement with Agile Practices in a Large Telecom Company." In: Jürgen Münch, Matias Vierimaa (editors). *Product-Focused Software Process Improvement.* Proceedings of the 7th International Conference on Product Focused Software Process Improvement. Lecture Notes in Computer Science vol. 4034. Springer-Verlag Berlin Heidelberg. ISBN 978-3-540-34683-8.

Beck, Kent 1999. “Embracing Change with Extreme Programming.” *Computer* vol. 32, issue 10. IEEE computer Society Press. Los Alamitos, California.

Begel, Andrew, and Beth Simon 2008. “Struggles of New College Graduates in Their First Software Development Job.” *SIGCSE Bulletin* 40.

Bergquist, Magnus 2003. “Open Source Software Development as Gift Culture: Work and Identity Formation in an Internet Community.” In: Garsten & Wulff.

Boehm, Barry W. 1988. “A Spiral Model of Software Development and Enhancement.” *Computer* vol. 21, issue 5. IEEE Computer Society.

Boehm, Barry W. 2006. “A View of 20th and 21st Century Software Engineering.” In: Proceedings of the 28th International Conference on Software Engineering (ICSE ’06). ACM. New York. ISBN 1-59593-375-1.

Boehm, Barry W., and Richard Turner 2003. “Using Risk to Balance Agile and Plan-Driven Methods.” *Computer* vol. 36, issue 6. IEEE Computer Society Press. Los Alamitos, California.

Bolter, Jay David, and Richard Grusin 1998 [2000]. *Remediation: Understanding New Media.* First MIT Press paperback edition 2000. The MIT Press. Cambridge, Massachusetts, and London. ISBN 978-0-262-52279-3.

Borda, Beatriz 1989. “Mellan medvetande och text.” *Nord Nytt* no. 37. Nordisk Etnologisk Folkloristisk Arbejdsgruppe.

Brooks, Frederick P., Jr. 1975 [1995]. *The Mythical Man-Month: Essays on Software Engineering.* Anniversary edition 1995. Addison-Wesley. ISBN 0-201-83595-9.

Brooks, Frederick P., Jr. 1986 [1995]. “No Silver Bullet—Essence and Accidents of Software Engineering.” *Information Processing, IFIP.* Elsevier. Reprinted 1987 in IEEE *Computer* magazine. In: Brooks 1975 [1995].

Bruegge, Bernd, and Allen H. Dutoit 2000 [2010]. *Object-Oriented Software Engineering: Using UML, Patterns, and Java.* Third edition, International edition 2010. Pearson Education publishing as Prentice Hall. ISBN 978-0-13-815221-5.

Bryant, Antony 2000. “It’s Engineering Jim … But Not as We Know It: Software Engineering – Solution To the Software Crisis, Or Part of the Problem?” In: Proceedings of the 22nd International conference on Software engineering (ICSE ’00). ACM. New York. ISBN 1-58113-206-9.

Buxton, J.N., and B. Randell (editors) 1970. *Software Engineering Techniques.* Report on a conference sponsored by the NATO Science Committee Rome, Italy, 27th to 31st October 1969. NATO Science Committee.

Cardwell, Donald 1994. *The Fontana History of Technology.* FontanaPress. London. ISBN 0-00-686176-8.

Christensen, Lone Rahbek 1987. *Hver vore veje: Livsformer, familietyper & kvindeliv.* Museum Tusculanums Forlag. Københavns Universitet. ISBN 87-7289-243-9.

von Clausewitz, Carl 1832-1834 [2007]. *Vom Kriege.* Translated by Michael Howard and Peter Paret. *On War.* Princeton University Press 1976. Abridged by Beatrice Heuser. Oxford University Press 2007. ISBN 978-0-19-954002-0.

Coase, Ronald H. 1937. “The Nature of the Firm.” *Economica* 4 (November): 386-405.

Cockburn, Alistair 2001. *Agile Software Development.* Addison-Wesley Professional. ISBN 0-201-69969-9. Draft version: 3b.

Cole, Melissa, and Davis Avison 2007. “The Potential of Hermeneutics in Information Systems Research.” *European Journal of Information Systems*, vol 16: 820-833.

Coleman, E. Gabriella 2005. *The Social Construction of Freedom in Free and Open Source Software: Hackers, Ethics, and the Liberal Tradition.* PhD thesis. Department of Anthropology, University of Chicago.

Coplien, James O., and Neil B. Harrison 2004. *Organizational Patterns of Agile Software Development.* Pearson Prentice Hall. Upper Saddle River, New Jersey. ISBN 0-13-146740-9.

Coyne, Richard 1995. *Designing Information Technology in the Postmodern Age: From Method to Metaphor.* The MIT Press. Cambridge, Massachusetts, and London. ISBN 0-262-03228-7.

Crawford, Chris 2002. *The Art of Interactive Design: A Euphonious and Illuminating Guide to Building Successful Software.* No Starch Press. San Fransisco. ISBN 1-886411-84-0.

Curtis, Bill, and Diane Walz 1990. "The Psychology of Programming in the Large: Team and Organizational Behaviour." In: Jean-Michel Hoc, T. Green, R. Samurçay, and D.J. Gilmore (editors). *Psychology of Programming.* Academic Press. ISBN 0-12-350772-3.

Czarnecki, Krysztof, and Ulrich W. Eisenecker 2000. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley Professional. ISBN: 978-0-2013-0977-5.

DeMarco, Tom, and Timothy Lister 1987 [1999]. *Peopleware: Productive Projects and Teams.* Second edition 1999. Dorset House Publishing. ISBN 978-0-932633-43-9.

Dijkstra, Edsger W. 1975. *Homo Cogitans: A Small Study of the Art of Thinking.* Unpublished manuscript no. EWD533. The Center for American History, The University of Texas at Austin.

Dybå, Tore, and Torgeir Dingsøyr 2008. "Empirical Studies of Agile Software Development: A Systematic Review." *Information and Software Technology* vol. 50, issues 9-10. Elsevier.

Easterbrook, Steve, Janice Singer, Margaret-Anne Storey, and Daniela Damian 2008. "Selecting Empirical Methods for Software Engineering Research." In: Shull et al.

Ehn, Billy 1981. *Arbetets flytande gränser: En fabriksstudie.* Bokförlaget Prisma. Stockholm. ISBN 91-518-1434-X.

Ekman, Susanne 2010. *Authority and Autonomy: Paradoxes of Modern Knowledge Work.* PhD thesis. Doctoral School of Organisation and Management Studies, Handelshøjskolen i København. ISBN 87-593-8435-0.

Fein, Louis 1959. "The Role of the University in Computers, Data Processing, and Related Fields." *Communications of the ACM* 2(9): 7-14.

Filinski, Andrzej, Robert Glück, and Neil Jones (editors) 2005. *Noter i Datalogi V – Programmeringssprog.* HCØ Tryk. Datalogisk Institut, Københavns Universitet. ISBN 87-7834-663-0. In English.

Fishman, Charles 1997. “They Write the Right Stuff.” *Fast Company* issue 6, Dec 1996 / Jan 1997. Fast Company, Inc.

Fowler, Martin, and Jim Highsmith 2001. “The Agile Manifesto.” *Dr. Dobb’s Journal.* http://www.drdobbs.com/open-source/the-agile-manifesto/184414755.

Fägerborg, Eva 1999. “Intervjuer.” In: Kaijser & Öhlander.

Gadamer, Hans-Georg 1960 [1965]. *Wahrheit und Methode: Grundzüge einer philosophischen Hermeneutik.* J.C.B. Mohr (Paul Siebeck). Tübingen. Second edition 1965.

Gadamer, Hans-Georg 1976. *Rhetorik und Hermeneutik: Als öffentlicher Vortrag der Jungius-Gesellschaft der Wissenschaften gehalten am 22. 6. 1976 in Hamburg.* Joachim Jungius-Gesellschaft der Wissenschaften. Vandenhoeck & Ruprecht. Göttingen. ISBN 3-525-85553-2.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley. ISBN 0-201-63361-2.

Garsten, Christina, and Helena Wulff (editors) 2003. *New Technologies at Work: People, Screens and Social Virtuality.* Berg. Oxford, New York. ISBN 1-85973-649-1.

Glass, Robert L. 2002. *Facts and Fallacies of Software Engineering.* Addison-Wesley. ISBN 0-321-11742-5.

Gormsen, Gudrun 1982. “Hedebonden: Studier i gårdmand Peder Knudsens dagbog 1829–1857.” *Folk og Kultur.* Offprint in the series *IEF småskrifter.* With an English summary.

Grady, Robert B. 1997. *Successful Software Process Improvement.* Hewlett-Packard Professional Books. Prentice Hall PTR. Upper Saddle River, New Jersey. ISBN 0-13-626623-1.

Haraway, Donna J. 1991. *Simians, Cyborgs and Women: The Reinvention of Nature.* Routledge. New York. ISBN 978-0-415-90387-5.

Heidenberg, Jeanette 2011. *Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches.* PhD Thesis. Department of Information Technologies, Åbo Akademi. ISBN 978-952-12-2533-8.

Henderson, Kathryn 1998. *On Line and On Paper: Visual Representations, Visual Culture, and Computer Graphics in Design Engineering.* The MIT Press. Cambridge, Massachusetts, and London. ISBN 0-262-08269-1.

Herbsleb, James D., Daniel J. Paulish, and Matthew Bass 2005. "Global Software Development at Siemens: Experience from Nine Projects." In: Proceedings of the 27th International Conference on Software Engineering. IEEE Computer Society.

Highsmith, Jim 2002. "What is Agile Software Development?" *CrossTalk, The Journal of Defense Software Engineering.*

Hoare, C.A.R. 1980 [1981]. "ACM Turing Award Lecture." *Communications of the ACM* 1981.

Hunt, Andrew, and David Thomas 1999. *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley. ISBN 0-201-61622-X.

Huth, Michael, and Mark Ryan 2000 [2004]. *Logic in Computer Science: Modelling and Reasoning About Systems.* Second edition 2004. Cambridge University Press. ISBN 0-521-54310-X.

Højrup, Ole 1967. *Landbokvinden: Rok og kærne. Grovbrød og vadmel.* Nationalmuseet. København.

Højrup, Thomas 1995. *Omkring livsformsanalysens udvikling.* Museum Tusculanums Forlag. Københavns Universitet. ISBN 87-7289-337-0. With an English summary.

Højrup, Thomas 2002. *Dannelsens dialektik: Etnologiske udfordringer til det glemte folk.* Museum Tusculanums Forlag. Københavns Universitet. ISBN 87-7289-785-6.

Højrup, Thomas 2003. *Livsformer og velfærdsstat ved en korsvej? Introduktion til et kulturteoretisk og kulturhistorisk bidrag.* Stats- og livsformer vol. 5. Museum Tusculanums Forlag. Københavns Universitet. ISBN 87-7289-858-5.

Ihde, Don 1979. *Technics and Praxis.* Boston Studies in the Philosophy of Science vol. XXIV. Pallas edition. D. Reidel Publishing Company. Dordrecht, Boston, and London. ISBN 90-277-0954-8.

Institute of Electrical and Electronics Engineers 1985. *Computer.* Theme: "Visual Programming." Vol. 18, no. 8. IEEE Computer Society.

Institute of Electrical and Electronics Engineers 1990. *IEEE Standard Glossary of Software Engineering Terms.* IEEE Std 610-12.1990. ISBN 1-55937-067-X.

International Electrotechnical Commission 2009. *Functional safety of electrical/electronic/programmable electronic safety-related systems.* IEC 61508-(1–7) Ed. 2.0. 65A/548/FDIS Final Draft International Standard, distributed on 2009-12-18.

Jacobson, Ivar, Shihong Huang, Mira Kajko-Matsson, Paul McMahon, and Ed Seymour 2012. "Semat – Three Year Vision." *Programming and Computer Software* vol. 38, no. 1. Pleiades Publishing, Ltd.

Jensen, Charlotte 2008. "Gevinster og udfordringer i tværvidenskabelig forskning." *Nord Nytt* no. 103. Nordisk Etnologisk Folkloristisk Arbejdsgruppe. Syddansk Universitetsforlag. ISBN 987-87-91714-10-8.

Jones, Neil D., and David A. Schmidt 1980. "Compiler Generation from Denotational Semantics." In: Neil D. Jones (editor). *Semantics-Directed Compiler Generation, Proceedings of a Workshop.* Springer-Verlag London. ISBN 3-540-10250-7.

Kaijser, Lars, and Magnus Öhlander (editors) 1999. *Etnologiskt fältarbete.* Studentlitteratur. ISBN 91-44-00944-5.

Kennedy, George A. 1984. *New Testament Interpretation through Rhetorical Criticism.* The University pf North Carolina Press. Chapel Hill and London. ISBN 978-0-8078-4120-4.

Kernighan, Brian W., and Dennis M. Ritchie 1978 [1988]. *The C Programming Language.* Second edition 1988. Prentice Hall PTR. Englewood Cliffs, New Jersey. ISBN 0-13-110362-8.

Kittler, Friedrich 2009. "Towards an Ontology of Media." *Theory, Culture & Society* vol. 26. Sage.

Knuth, Donald E. 1968-2011. *The Art of Computer Programming.* Vol. 1, 1968 [1997]. *Fundamental Algorithms.* Vol. 2, 1969 [1997]. *Seminumerical Algorithms.* Vol. 3, 1973 [1998]. *Sorting and Searching.* Vol. 4A, 2011. *Combinatorial Algorithms.* Addison-Wesley. Reading, Massachusetts, and Upper Saddle River, New Jersey. ISBN 0-201-89683-4; 0-201-89684-2; 0-201-89685-0; 0-201-03804-8.

Knuth, Donald E., 1989 [1990]. “The Errors of TEX.” *Journal of Software: Practice & Experience* vol. 19, no. 7. John Wiley & Sons, Ltd. In: Tom DeMarco and Timothy Lister (editors) 1990. *Software State-Of-The-Art: Selected Papers.* Dorset House Publishing. New York. ISBN 0-932633-14-5.

Ko, Andrew J., Robert DeLine, and Gina Venolia 2007. “Information Needs in Collocated Software Development Teams.” In: Proceedings of the 29th International Conference on Software Engineering. IEEE Computer Society.

Kraft, Philip 1979. “The Industrialization of Computer Programming: From Programming to ‘Software Production’.” In: Andrew Zimbalist (editor). *Case Studies on the Labor Process.* Monthly Review Press. New York and London. ISBN 0-85345-518-X.

Kuhn, Sarah 1989 [1992]. “The Limits to Industrialization: Computer Software Development in a Large Commercial Bank.” In: Stephen Wood (editor). *The Transformation of Work? Skill, Flexibility and the Labour Process.* Unwin Hyman Ltd. Second impression 1992. Routledge. London and New York. ISBN 0-415-07869-5.

Kunda, Gideon 1992. *Engineering Culture: Control and Commitment in a High-Tech Corporation.* Temple University Press. Philadelphia. ISBN 0-87722-845-0.

Lammers, Susan 1986. *Programmers at Work.* Microsoft Press. Redmond, Washington. ISBN 0-914845-71-3.

Lange, Martin 2012. “Integration nicht-sicherer Software in sicherheitsgerichtete Systeme.” 10th International TÜV Rheinland Symposium, May 15-16. Cologne.

Larsson, Marianne 2008. “Numeral Symbols on the Uniform Collar: Post Office Constitution of Subordinated Masculinity.” *Ethnologia Scandinavica* vol. 38.

Latour, Bruno 1987. *Science in Action: How to Follow Scientists and Engineers Through Society.* Harvard University Press. Cambridge, Massachusetts. ISBN 0-674-79291-2.

Latour, Bruno, and Steve Woolgar 1979 [1986]. *Laboratory Life: The Construction of Scientific Facts.* Sage Publications, Inc. Second edition 1986. Princeton University Press. Princeton, New Jersey. ISBN 0-691-02832-X.

Lauesen, Søren (Soren) 2002. *Software Requirements: Styles and Techniques.* Addison-Wesley. ISBN 978-0-201-74570-2.

Liddell, Henry George, and Robert Scott 1940. *A Greek-English Lexicon.* Revised and augmented throughout by Sir Henry Stuart Jones with the assistance of Roderick McKenzie. Clarendon Press. Oxford.

Mahoney, Michael S. 1990. "The Roots of Software Engineering." *CWI Quarterly* vol. 3, no. 4: 325-334.

Mallery, John C., Roger Hurwitz, and Gavan Duffy 1987. "Hermeneutics: From Textual Explication to Computer Understanding?" Massachusetts Institute of Technology Artificial Intelligence Laboratory. In: Stuart C. Shapiro (editor). *The Encyclopedia of Artificial Intelligence.* John Wiley & Sons. New York.

Mancy, Rebecca, and Norman Reid 2004. "Aspects of Cognitive Style and Programming." In: E. Dunican and T.R.G. Green (editors). Proceedings of the 16th Workshop of the Psychology of Programming Interest Group, pp. 1-9. Carlow, Ireland.

Manovich, Lev 2013. *Software Takes Command.* Bloomsbury Academic. New York and London. ISBN 978-1-6235-6672-2.

McCloskey, Donald (Deirdre) N. 1985 [1998]. *The Rhetorics of Economics.* Second edition 1998. The University of Wisconsin Press. Madison, Wisconsin, and London. ISBN 0-299-15814-4.

McConnell, Steven C. 1993. *Code Complete: A Practical Handbook of Software Construction.* Microsoft Press. Redmond, Washington. ISBN 1-55615-484-4.

Medoff, Michael D., and Rainer I. Faller 2010. *Functional Safety: An IEC 61508 SIL 3 Compliant Development Process.* Exida.com L.L.C. Sellersville, Pennsylvania. ISBN 978-0-9727234-8-0.

Miller, Daniel, and Don Slater 2000. *The Internet: An Ethnographic Approach.* Berg. Oxford, New York. ISBN 1-85973-389-1.

Naur, Peter 1985 [2001]. "Programming as Theory Building." Reprinted in *Computing: A Human Activity.* In: Cockburn 2001.

Naur, Peter 1995. *Datalogi som videnskab.* DIKU Rapport no. 95/4. DIKU Tryk. Datalogisk Institut, Københavns Universitet. ISSN 0107-8283.

Naur, Peter, and Brian Randell (editors) 1969. *Software Engineering.* Report on a conference sponsored by the NATO Science Committee. Garmisch, Germany, 7th to 11th October 1968. NATO Science Committee.

Nielsen, Niels Jul 2004. *Mellem storpolitik og værkstedsgulv. Den danske arbejder – før, under og efter Den kolde krig.* Stats- og livsformer vol. 6. Museum Tusculanums Forlag. Københavns Universitet. ISBN 87-7289-862-3.

Ó Riain, Seán 1997. "An Offshore Silicon Valley? The Emerging Irish Software Industry." *The Journal of Global Business and Political Economy* 2:175-212.

Ó Riain, Seán 2000. "Net-Working for a Living: Irish Software Developers in the Global Workplace." In: M. Burawoy et al. (editors) *Global Ethnography.* University of California Press.

Paaskoski, Leena 2008. "Brothers and Sisters of the Forest: Gender in the Forester Profession." *Ethnologia Scandinavica* vol. 38.

Pane, John F., and Brad A. Myers 2000. "The Influence of the Psychology of Programming on a Language Design: Project Status Report." In: A. F. Blackwell and E. Bilotta (editors). Proceedings of the 12th Annual Meeting of the Psychology of Programmers Interest Group. Corigliano Calabro, Italy: Edizioni Memoria. pp. 193-205.

Patterson, David A., and John L. Hennessy 1997. *Computer Organization and Design: The Hardware / Software Interface.* Second edition. Morgan Kaufmann Publishers, Inc. San Fransisco. ISBN 1-55860-491-X.

Paulson, Lawrence C. 1991 [1996]. *ML for the Working Programmer.* Second edition 1996. Cambridge University Press. ISBN 0-521-56543-X.

Pedersen, Mikkel Venborg 2005. *Hertuger: At synes og at være i Augustenborg 1700-1850.* Etnologiske Studier vol. 12. Museum Tusculanums Forlag. København. ISBN 978-87-635-0191-0.

Peirce, Charles Sanders 1905 [1931-1958]. *Collected Papers of Charles Sanders Peirce.* Vols. 1-6 edited by Charles Hartshorne and Paul Weiss. Vols. 7-8 edited by Arthur W. Burks. 1931-1958. Harvard University Press. Cambridge, Massachusetts.

Perelman, Chaïm, and Lucie Olbrechts-Tyteca 1958 [2008]. *Traité de l'argumentation: La nouvelle rhétorique.* Sixth edition 2008. Editions de l'Université de Bruxelles. ISBN 978-2-8004-1398-3.

Perlow, Leslie A. 1997. *Finding Time: How Corporations, Individuals, and Families Can Benefit from New Work Practices.* Cornell University Press. Ithaca and London. ISBN 978-0-8014-8445-2.

Perlow, Leslie A. 1999. "The Time Famine: Toward a Sociology of Work Time." *Administrative Science Quarterly* vol. 44, no. 1. Sage Journals.

Peterson, Tina 2009. "The Zapper and the Zapped: Microwave Ovens and the People Who Use Them." In: Phillip Vannini (editor). *Material Culture and Technology in Everyday Life.* Intersections in Communications and Culture vol. 25. Peter Lang Publishing Inc. New York.

Pierce, Benjamin C. 2002. *Types and Programming Languages.* The MIT Press. Cambridge, Massachusetts, and London. ISBN 0-262-16209-1.

Reynolds, Carl H. 1970. "What's Wrong with Computer Programming Management?" In: Weinwurm.

Robinson, Hugh M., and Helen Sharp 2003. "XP Culture: Why The Twelve Practices Both Are and Are Not The Most Significant Thing." In: Proceedings of the Conference on Agile Development. IEEE Computer Society.

Robinson, Hugh M., and Helen Sharp 2005 (I). "The Social Side of Technical Practices." In: Proceedings of the Sixth International Conference on Extreme Programming and Agile Processes in Software Engineering. Springer Verlag.

Robinson, Hugh M., and Helen Sharp 2005 (II). "Organisational Culture and XP: Three Case Studies." In: Proceedings of the Agile Development Conference. IEEE Computer Society.

Rorty, Amelie Oksenberg 1983. "Experiments in Philosophical Genre: Descartes' Meditations." *Critical Inquiry* 9: 545-65.

Royce, Winston W. 1970. "Managing the Development of Large Software Systems." *Proc. IEEE WESCON.* IEEE Press.

Runeson, Per, and Martin Höst 2009. "Guidelines for Conducting and Reporting Case Study Research in Software Engineering." *Empirical Software Engineering* vol. 14, issue 2: 131-164. Springer.

Rönkkö, Mikko, and Juhana Peltonen 2012. *Software Industry Survey 2012.* School of Science, Aalto University. Espoo. ISBN 978-952-60-3614-4.

Sach, Rien, Helen Sharp, and Marian Petre 2011. “What makes software engineers go that extra mile?” In: Proceedings of the 23rd Annual Psychology of Programming Interest Group. York, UK.

Scanlon, Leo J. 1981. *The 68000: Principles and Programming.* Howard W. Sams & Co., Inc. Indianapolis, Indiana. ISBN 0-672-21853-4.

Schein, Edgar H. 1985. *Organizational Culture and Leadership: A Dynamic View.* Jossey-Bass Publishers. San Fransisco, Washington, and London. ISBN 0-87589-639-1.

Schön, Donald A. 1983. *The Reflective Practitioner: How Professionals Think in Action.* Basic Books. United States of America. ISBN 0-465-06878-2.

Schwaber, Ken 2003. *Agile Project Management with Scrum.* Microsoft Press. Redmond, Washington. ISBN 0-7356-1993-X.

Schwaber, Ken, and Jeff Sutherland 2010. *Scrum Guide.* http://www.scrum.org.

Seaman, Carolyn B. 2008. “Qualitative Methods.” In: Shull et al. Based on: “Qualitative Methods in Empirical Studies of Software Engineering.” *IEEE Transactions on Software Engineering* 25(4):557-572, 1999.

Shull, Forrest, Janice Singer, and Dag I.K. Sjøberg (editors) 2008. *Guide to Advanced Empirical Software Engineering.* Springer-Verlag London Limited. ISBN 978-1-84800-043-8.

Singer, Janice, Susan E. Sim, and Timothy C. Lethbridge 2008. “Software Engineering Data Collection for Field Studies.” In: Shull et al. Based on: Lethbridge, T., S. Sim, and J. Singer 2005. “Studying Software Engineers: Data Collection Techniques for Software Field Studies.” *Empirical Software Engineering,* 10(3), 311-341.

Solin, Ulla 1992. *Animation of Parallel Algorithms.* PhD thesis. Department of Computer Science, Åbo Akademi. Acta Academia Aboensis, Ser. B, Mathematica et physica, vol. 52, no. 2. Åbo Akademis förlag. ISBN 952-9616-03-1.

Spolsky, Joel 2004. *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. ISBN 978-1-59-059-389-9.

Star, Susan Leigh, and James R. Griesemer 1989. “Institutional Ecology, ‘Translations’ and Boundary Objects: Amateurs and Professionals in Berkeley’s Museum of Vertebrate Zoology, 1907-39.” *Social Studies of Science* vol. 19, issue 3. Sage Publications, Ltd.

Star, Susan Leigh, and Karen Ruhleder 1996. “Steps Toward an Ecology of Infrastructure: Design and Access for Large Information Spaces.” *Information Systems Research* vol. 7, issue 1.

Stroustrup, Bjarne 1985 [1997]. *The C++ Programming Language.* Third edition 1997. Addison-Wesley. ISBN 0-201-88954-4.

Suchman, Lucy A. 1987 [2007]. *Human-Machine Reconfigurations: Plans and Situated Actions.* Second edition 2007. Cambridge University Press. ISBN 978-0-521-67588-8.

Suenson, Espen 2005. “Kritik af den enkle vareproduktion – en begrebslogisk analyse.” *Nord Nytt* no. 96. Nordisk Etnologisk Folkloristisk Arbejdsgruppe. ISBN 87-91714-03-6.

Suenson, Espen 2013. “Method and Fieldwork in a Hermeneutical Perspective.” In: Frog and Pauliina Latvala (editors).*Approaching Methodology.* Humaniora 368. Finnish Academy of Science and Letters. ISBN 978-951-41-1085-6.

Suenson, Thomas 2008. “Begrebet FOLK i bibelsk-kristen forståelse.” *Tidehverv* vol. 82, no. 5/6.

Sundt, Eilert 1862 [1976]. *Om Bygnings-Skikken paa Landet i Norge.* Christiania. Reprinted by Gyldendals Norske Forlag 1976.

Szymanski, Margaret H., and Jack Whalen (editors) 2011. *Making Work Visible: Ethnographically Grounded Case Studies of Work Practice.* Cambridge University Press. ISBN 978-0-521-17665-1.

Turkle, Sherry 1984 [2005]. *The Second Self: Computers and the Human Spirit.* Simon & Schuster, Inc. New York. Twentieth Anniversary edition 2005. MIT Press. Cambridge, Massachusetts, and London. ISBN 978-0-262-70111-2.

U.S. Defense Science Board 1987. *Report of the Defense Science Board Task Force on Military Software.* Office of the Under Secretary of Defense for Acquisition. Washington, D.C.

U.S. Department of Commerce 2011. GDP by Industry. Tables of Value Added by Industry. Bureau of Economic Analysis.

U.S. Department of Defense 1985. *Military Standard: Defense System Software Development.* DOD-STD-2167. Washington, D.C.

Vincenti, Walter Guido 1990 [1993]. *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History.* Johns Hopkins Paperbacks edition 1993. The Johns Hopkins University Press. Baltimore and London. ISBN 0-8018-4588-2.

van Vliet, Hans 2008. *Software Engineering: Principles and Practice.* Third edition. John Wiley & Sons, Ltd. ISBN 978-0-470-03146-9.

Weber, Max 1922 [2003]. "Kapitel IX. Soziologie der Herrschaft. 2. Abschnitt. Wesen, Voraussetzungen und Entfaltung der büreaukratischen Herrschaft." First published in: *Wirtschaft und Gesellschaft. Dritter Teil. Typen der Herrschaft.* J.C.B. Mohr (Paul Siebeck). Tübingen. Translated from: Marianne Weber and Johannes Winckelmann (editors) 1990. *Wirtschaft und Gesellschaft. Grundriss der verstehenden Soziologie. Zweiter Teil.* J.C.B. Mohr (Paul Siebeck). Tübingen. In: Heine Andersen, Hans Henrik Bruun, and Lars Bo Kaspersen (editors) 2003. *Max Weber: Udvalgte tekster. Bind 2.* Translated by Ole Bjerg. Hans Reitzels Forlag. København. ISBN 87-412-2554-6.

Weinwurm, George F. (editor) 1970. *On the Management of Computer Programming.* Auerbach Publishers Inc. United States of America. Standard Book Number 87769-044-8.

Willim, Robert 2002. *Framtid.nu: Flyt och friktion i ett snabbt företag.* Brutus Östlings Bokförlag Symposion. Stockholm and Stehag. ISBN 91-7139-549-0. With an English summary.

Willim, Robert 2003. "Claiming the Future: Speed, Business Rhetoric and Computer Practice." In: Garsten & Wulff.

Wilson, Kax 1979 [1982]. *A History of Textiles.* Paberback edition 1982. Westview Press. Boulder, Colorado. ISBN 0-86531-368-7.

Winskel, Glynn 1993. *The Formal Semantics of Programming Languages: An Introduction.* Foundation of Computing Series. The MIT Press. Cambridge, Massachusetts, and London. ISBN 0-262-73103-7.

Öhlander, Magnus 1999. "Deltagande observation." In: Kaijser & Öhlander.

Appendix A

Method and Fieldwork in a Hermeneutical Perspective

Espen Suenson, Åbo Akademi

Published in: Frog and Pauliina Latvala (editors) 2013. Approaching Methodology. *Humaniora 368. Finnish Academy of Science and Letters.*

The background of the present text is my ongoing work on a doctoral dissertation at Åbo Akademi, a dissertation that is jointly in ethnology and computer engineering. My academic background is similarly partly ethnology and partly computer science. My professional experience as a programmer, along with my interest in ethnology, prompted me to begin an ethnological study of computer programming.

This text is a reflection on the fieldwork I have done to collect data for my dissertation. The fieldwork consists of interviews with and observations of computer programmers collected during the spring and autumn of 2011. I discuss my method along with an example of an ethnological historical study and I put it all in perspective by arguing for a hermeneutical understanding of scientific method.

The purpose of this text is to show how hermeneutics can help in understanding what happens during the scientific process. Hermeneutics is the classical study of what requisites there are to understanding. It has been particularly developed within Bible Studies – biblical exegesis – but has also been applied to other fields such as law and, increasingly since the 19th century, to texts in general. Ethnology is the study of folk culture and as a discipline has always been informed and inspired by other traditions, not least by the hermeneutical tradition and by anthropology.

The hermeneutical influence can be found in the works of ethnological figures such as Troels-Lund and H.F. Feilberg in Denmark, and Helmer Tegengren in Finland. The anthropological influence in ethnology can be felt especially in the discussions on fieldwork, and is connected with authors such as, for example, Bronislaw Malinowski, Franz Boas and Clifford Geertz. The discussion of the influence of anthropology on fieldwork will in this text be limited to the work of Bruno Latour and Steve Woolgar.

Science as Persuasion

Science is, at its heart, a persuasive activity. Any given research result will at some point be presented either in written form, as a book, article or report, or in oral form, as a talk at a conference or even as a remark during an informal chat between colleagues. The purpose of presenting scientific results is of course to convince the audience of the scientific truth of said result. The ideal of scientific practice is that through free and frank discussion and exchange of arguments between scholars, scientific truth will eventually prevail. The real test of scientific validity lies not in citation count but in the ability to convince educated and informed colleagues of the truth of the matter on the basis of the given scientific evidence. Since argument is the form of all persuasion, this means that scientific activity is a form of argumentative activity. Certainly, a scientific insight may be ever so true, but, if it cannot be presented convincingly, that is, if it cannot be argued, then it will have no impact on science.

We might ask of ourselves now whether argumentation is really an essential part of the scientific process as such. After all, it is possible to imagine that the scientist first reaches his scientific conclusions without giving any thought at all to how they are to be presented and only later constructs the arguments with which to present them. According to this way of thinking, argumentation is added to scientific results almost as an afterthought – as something that is certainly necessary to the spread of scientific knowledge but which is not an intimate part of how the scientist comes to the knowledge. Argumentation is seen as something external to science. This view, however, is not defendable in light of 20th century philosophical knowledge of argumentation and of science.

Chaïm Perelman and Lucie Olbrechts-Tyteca published in 1958 their *Traité de l'argumentation*, which was the result of ten years of intensive studies of argumentation. In their work, they present what is called "the new rhetorics", a modern theory of argumentation that rehabilitates Aristotle's classical thinking on rhetoric and connected it with present day thinking on argumentation. They compare the way a person addresses an audience with the way he considers a matter in the privacy of his own mind:

> L'individualisme des auteurs qui accordent une nette prééminence à la façon de conduire nos propres pensées et la considèrent comme seule digne de l'intérêt du philosophe – le discours adressé à autrui n'étant qu'apparance et tromperie – a été pour beaucoup dans le discrédit non seulement de la rhétorique, mais, en général, de toute théorie de l'argumentation. Il nous semble, par contre, qu'il y a tout intérêt à considérer la délibération intime comme une espèce particulière d'argumentation.
> (Perelman & Olbrechts-Tyteca 1958 : §9, p. 54.)

That is to say that to consider a person's deliberation with himself and his private convictions to be the primary object of philosophical and scientific thought, and to consider that arguments directed to an audience are but an afterthought, is both wrong and harmful to the theory of argumentation. Instead, private convictions are a special case of argumentation in general. This view is clearly at odds with the idea that scientific discovery should be independent of subsequent presentation. Accordingly:

> Aussi, de notre point de vue, c'est l'analyse de l'argumentation adressée à autrui qui nous fera comprendre mieux la délibération avec soi-même, et non l'inverse.
> (Perelman & Olbrechts-Tyteca 1958 : §9, p. 54.)

That is, the analysis of arguments directed to others informs the study of private conviction and not the other way around. Perelman and Olbrechts-Tyteca point out that this way of understanding argumentation allows an explanation of how a person can be convinced of something and yet not be able to express his conviction in a way that can persuade others. This is because the argumentation that suffices to convince himself can be based on arguments that are valid to him alone. But, such arguments, though they may be true and valid as far as the individual is concerned, are not scientific arguments, since they are not held by the general scientific community to be valid. The practice of science requires the uncovering of arguments that are more generally accepted than personal conviction or opinion. We see thus that, in the light of argumentation theory, we cannot completely separate scientific discovery from the way it is to be presented to a scholarly audience.

Such is the judgment of argumentation theory on the matter at hand. We turn now to philosophical thought on the subject. Hans-Georg Gadamer published in 1960 his magnum opus *Wahrheit und Methode* in which he practically founded the field of philosophical hermeneutics and summed up the preceding centuries' thoughts on the essence of scientific interpretation and scientific understanding. Gadamer points out that understanding is inescapably linked to application. Application is not something that comes after understanding, but is given in advance and determines the whole of understanding. An interpreter of history seeks to apply his interpretation, and the use of it is not something that comes strictly after a general understanding of the text:

> Auch wir hatten uns davon überzeugt, daß die Anwendung nicht ein nachträglicher und gelegentlicher Teil des Verstehens-phänomens ist, sondern es von vornherein und im ganzen mitbestimmt. . . . Der Interpret, der es mit einer Überlieferung zu tun hat, sucht sich dieselbe zu applizieren. Aber auch hier heißt das nicht, daß der überlieferte Text für ihn als ein Allgemeines gegeben und verstanden und danach erst für besondere Anwendung in Gebrauch genommen würde.
> (Gadamer 1960: II.II.2.b, p. 307.)

Gadamer gives an example of what this means in the practice of judicial hermeneutics. In judicial hermeneutics, the application of understanding is the action of passing judgment. In order to understand the original intent of a law, the interpreter must understand how the law is used for passing judgment. This means that he must undergo the same process of mental reasoning, of thinking through the consequences of the law, as the judge who is actually passing judgment according to the law. On the other hand, a judge passing judgment in the present situation must understand the intent of the law. That means setting aside the matter at hand for a moment, in order to understand what the original circumstances were in which the law was to be used. Since circumstances always change over time, the letter of the law alone is not enough in passing just judgment. The concept of application of the law is what links the judge of the present with the lawgiver of the past. (Gadamer 1960: II.II.2.c.)

In law, the application of a text is obvious. Regarding history, it seems less immediate. In history, the essential application is to interpret texts and other sources in order to obtain a coherent and meaningful understanding of the past:

> Für den Historiker tritt jedoch der einzelne Text mit anderen Quellen und Zeugnissen zur Einheit des Überlieferungsganzen zusammen. Die Einheit dieses Ganzen der Überlieferung ist sein wahrer hermeneutische Gegenstand. (Gadamer 1960: II.II.2.c, p. 322.)

That is, for the historian, each single text that he studies joins with other texts and sources and forms a whole that expresses the understanding of our past. The unity of this whole is the true hermeneutical purpose of history.

What is of special interest to us in this is that, accordingly, scientific understanding must be understood in terms of scientific application. For a scholar, the immediate application of research is not the eventual practical usefulness of the results, but rather the necessity of persuading other scholars and, as we understand from the above, oneself. An example of this that should be familiar to many is what we experience when we teach a difficult subject for the first time. Even though we feel that we have mastered the subject ourselves, we find that the fullest understanding comes to us only when we try to teach it to others.

We have argued that, both from a communicative and a philosophical perspective, science is best understood as a persuasive activity. However, though Gadamer's thoughts apply to all understanding in general, he is first and foremost concerned with the phenomenon of understanding within *Geisteswissenschaft*, a term that can be somewhat imprecisely translated as "the humanities", but one that really means something like "the sciences concerned with free human thought". Nevertheless, this does not mean that the persuasive aspect can somehow be avoided in certain fields of science.

The exact sciences are argumentative in exactly the same way as all other sciences. Indeed, Perelman and Olbrechts-Tyteca (1958: §6, p. 37f.) point out that there is no such thing as pure objectivity. This is not to say that objectivity does not exist. Rather, objectivity must always be understood in terms of a subject that regards the object. Without subject there is no object. It is because of this that application has such a central place in Gadamer's explanation of understanding, for it is precisely application that establishes the relationship between subject and object, in that the subject performs some action on the object in order to reach a goal. (Højrup 1995: 65–69.)

In 1979, Bruno Latour and Steve Woolgar published the book *Laboratory Life*, an anthropological study of how science is done in a neuroendocrinological laboratory based on two years of observation. Neuro-endocrinology as a field is at the very heart of exact sciences and the book has since become a modern classic in the field of science and technology studies. Latour and Woolgar show how science is indeed a highly rhetorical, persuasive activity. Facts and findings are constantly being argued for, questioned and recast in new formulations, with the scientists' credibility and rhetorical skills being important factors in the eventual acceptance or dismissal of their ideas. The rhetorical persuasion is so effective that in the end, the scientists are not even aware that they have been persuaded, but come to regard the accepted arguments as objective, immutable facts. (Latour & Woolgar 1979: 240.) As Latour and Woolgar show conclusively, not even in the exact sciences are the bare facts in themselves enough to make up a scientific finding.[1]

The Scientific Argument

As shown above, science is an argumentative activity. In other words, science is persuasion – though not "mere" persuasion, but a special form of persuasion that is especially convincing. It is therefore of interest to examine what a scientific argument consists of in more detail. In the classical theory of rhetoric, Aristotle divides the means of demonstration that can be used in an argument into two classes: the non-technical and the technical, where "technical" is to be understood as rhetorical.[2] (Aristotle: 1355b, A.II.2.) Non-technical means

[1]Compare with the quotation from Gadamer in the end of the next section.

[2]Since Aristotle considers rhetorics to be a technique, *τέχνη*, which means something like an art or a craft – something that can be taught. (Aristotle: 1354a, A.I.2.)

are here to be understood as the evidence that is given and available to the argument in the form of documents, witness explanations and the like. It is non-technical (not rhetorical) because it is not common to argumentation in general as such, but is particular to the matter being debated. Put another way, when we argue scientifically, we need both something to speak about, which is the scientific evidence, and a way of forming our speech. Scientific evidence is not the same thing as proof. Rather, evidence is the means of proof. A piece of evidence can be interpreted in different ways, yielding different conclusions.

The problem of obtaining the scientific evidence, the data, is the subject of much scientific method. Sometimes the evidence is more or less given, as in an archive of collected material that is just waiting to be analysed. However, in most cases there are some specific questions that we want to answer and our first problem is how to get any evidence at all. At first glance, it would seem that the situations are very different for historical and contemporary research. In historical research, the material available is that which is preserved. We can never hope to get more, short of an unexpected discovery of previously unknown sources. In contemporary research, on the other hand, our informants are still available; the life we are studying is unfurling around us. We can generate as much data as we want to.

A closer examination, however, reveals that this depiction is not entirely accurate. True, the past is the past and in that sense more historical evidence cannot be produced; it is limited to what has been preserved. However, the decision of how much of the preserved evidence should be included in a scientific argument is left to the scholar's discretion.

To take an example: When studying a Danish peasant doing construction works on his fields in the poor moorlands of Vestjylland in 1834, it is evidently useful to know something about which fields were considered of high quality at that time and in that area. (Gormsen 1982: 13.) Perhaps it would also be relevant to know about the general economic conditions in Vestjylland at the time. Perhaps in all of Denmark. Maybe it would be informative to know about the earlier history of farming techniques, to find out from where the peasant got his knowledge of construction works. The construction works were not particularly successful, so perhaps it would also be useful to have some knowledge of farming techniques in later times in order to interpret the lack of success – not to speak of comparing similar construction works in the area at the time. Also, the construction works were just a small aspect of the peasant's activities.

As we see, the limited availability of historical evidence is only apparent, since much more historical evidence has been preserved than a single person can possibly process in its entirety. The real limit on the availability of evidence is that the evidence does not always speak about the things that we want to know about. The peasant's diary speaks mostly of farming tasks, of construction works and money loans, when what we are really interested in is the farmer's perception of his existence, a classic ethnological subject. Any historical research involves a selection of the relevant historical evidence. This selection is a limitation that the historian imposes on herself in order to be able to make an interpretation; see for example Jill Bradley's discussion of how to select material for image research in this volume of *RMN Newsletter*. Thus, the fundamental limits on the availability of historical evidence is in essence a problem of interpretation rather than quantity.

Let us now examine the case of contemporary research. My current research involves conducting interviews by phone with engineers in other countries, transcribing those interviews and finally analysing what the engineers tell me. It is often quite difficult to make out what the engineers say over a bad phone connection and in a language that is foreign to both of us. Even if I can understand what they are saying, it does not always make sense to me. Of course, since the research is contemporary, I can always collect more evidence, either by talking to the engineers again or by finding some other engineers to ask. There is, though, a limit to how much evidence I can process – I cannot talk to every single engineer in the world. And even if I could, the problems of understanding the engineers are still there. If there is something I do not understand, I can ask the engineers again, but it is perfectly possible that I will still not understand the answer.

The essential problem of the availability of contemporary scientific evidence is, as in the case of historical research, one of interpretation. This is, of course, assuming that the people I am studying want to let me interview them in the first place. People have their reasons for wanting to talk to me or not, and that is a factor outside my control. The access to the field of study is a fundamental limitation in contemporary research. This is akin to historical research in that, for some reason or other, the people of the past chose to write some things down and not others, as in the diary mentioned above where the peasant chose to write about his work, not his emotions. That cannot be changed. This limitation evidently does not preclude contemporary studies of a field that is difficult to access or historical studies of a sparsely documented subject, but the available evidence will be more indirect and the task of interpretation accordingly more difficult.

This discussion of the availability of evidence reveals that it is of crucial importance when talking about scientific method to know what it is that we want to know something about – the research goal. We mentioned that the scientific argument has to have something to speak about and a way of saying it, and a final requirement is of course that there is something we want to say. This something, which is the research goal, is determining for the interpretation of evidence, and this is the reason that Gadamer devotes so much effort to the relationship between interpretation and application in *Wahrheit und Methode*. Gadamer puts it this way:

> Der Historiker verhält sich zu seinen Texten wie der Untersuchungsrichter beim Verhör von Zeugen. Indessen macht die bloße Feststellung von Tatsachen, die er etwa der Voreingenommenheit der Zeugen ablistet, noch nicht wirklich den Historiker, sondern erst das Verständnis der Bedeutung, die er in seinen Feststellungen findet.
> (Gadamer 1960: II.II.2.c, p. 321.)

That is, the historian's relationship to the historical document is like that of a judge to a witness being interrogated. The raw facts in themselves, stripped of the bias of the witness, are not interesting but for the understanding of meaning that the historian finds during the discovery of facts.

Examples of Method in Fieldwork

As argued above, availability of evidence and research goals are factors that are important in forming scientific method. I will now give some examples from my ongoing research of how scientific method is influenced by these factors and how it in turn influences them.

My research is concerned with the work practices of computer programmers. The goal is to present a characterization of programming work based on my observations and on an ethnological perspective on culture, and to compare this characterization with the programmers' own understanding of their work practice. The focus on work practice and its connection to cultural context makes my research comparable to studies such as *Arbetets flytande gränser* by Billy Ehn from 1981, in which Ehn presents the results of the seven months he spent as a factory worker in the medical industry. Gudrun Gormsen's 1982 study of the diary of a moorland peasant in the years 1829-1857 is also an inspiration for my research, since Gormsen's work can be perceived as a historical work study.

The data I have collected for my research falls in two parts. The first part consists of interviews conducted by telephone with software engineers from about twenty companies from all over Europe. The companies all work with safety-critical systems, that is, they make automobiles, airplanes, medical equipment and so forth. The second part consists of notes from four weeks I spent as an observer in a small company that makes computer games. I was present during work hours: ordinary office hours, usually nine to five. The time was spent predominantly in observation and taking notes, without interacting with the people concerned. This is supplemented by interviews with the employees and a collection of some photographs and written material.

The collection of the first part of the data is a prime example of how the availability of evidence can influence method. I was offered, as part of another research project, to participate in making the interview series. The interviews were to be focused on how software engineers describe their work, as that was the focus of the other research project. My original intent was to perform observations on site in companies. However, it is time consuming to find informants who are willing to be studied. Moreover, from my contacts in academia, I knew that it could be difficult to get access to companies in this particular branch of the software industry because they are sometimes secretive about their detailed operations. Thus, when it became possible to gain access to informants from all these companies with whom it might otherwise have been difficult to establish contact, I chose to collect data with the prescribed method of the other research project – telephone interviews – instead of my original preference, observation on site.

This, on the other hand, also offers an example of how method can influence research goals. The telephone interview method and the focus on the informants' descriptions of their work practices was not as well suited as the observation method for my prime research interest at the time, the concrete day to day work practice. With the telephone interview material, I have to infer the work practices from the conversations with the engineers instead of observing it directly. This could be seen as a deviation from my original intent; however, I realized that the material offers other possibilities. Specifically, the telephone interview material shows in a much more direct way than observations of practice how the programmers describe their work and thus how they understand their work. The programmers' understanding of their work and the relation it has to their work practice thus became a much more important aspect of my research goals than previously. This also goes to illustrate the point of the preceding section, that availability of evidence is more a question of interpretation than of quantity.

The influence of research goals on method is in many cases immediately obvious: a method is chosen for its ability to generate evidence that can reveal something about that which we want to investigate. This influence also applies to the collection of the second part of my data. To observe work practice as directly as possible, I chose to use immediate, direct observation. This choice may perhaps seem obvious, but it is not the only option available. I could have chosen to rely exclusively on interviews, to do a pure academic literature study or to collect written evidence from the internet. All of these methods have their merit. However, as I seek to investigate programming not only as it is understood but also at is it concretely practiced, I chose the method that has the most immediate connection to concrete practice, namely to be present during the work. Or rather, there exists an even more immediate method – which is to actually do the work, as Ehn did in his factory study. I decided not to do the latter, partly because it would take longer than I was prepared to spend on the study and partly because I already have years of practice as a programmer and thus judge myself capable of understanding the practice that I observe without carrying out the practice myself.

The influence of method on the availability of evidence is also exemplified by the second part of my data collection. Choosing on-site observations as my method limited the availability of companies to study. Having an observer present affects the workplace and this can be seen as an unnecessary burden on the company. I was thus turned down by one company on this ground. Even within the observation situation, the choice of method can be felt. Because I was more interested in the programmers' interaction with each other than with me, I sought to minimize my interaction with them. This meant that explanatory comments and casual remarks directed to me, evidence in their own right, became much scarcer. The relative availability of two kinds of evidence that to a degree exclude each other was affected by my choice of method.

The Role of Scientific Theory

Let us now take a look at how we can understand the role of scientific theory in the scientific argument. At a very general level, a theory explains what is relevant about the subject matter and how the relevant parts relate to each other. It is a point of departure for our understanding. Thus, theory ideally tells us how we expect things to be before we start an investigation into the matter.

The question of prerequisites to understanding is treated in depth by Gadamer. What he arrives at is that there can be no understanding without prejudice (*Vorurteil*). (Gadamer 1960: II.II.1.a.α.) Prejudices are perspectives and opinions, and we all always hold some prejudices. No mind is a blank slate. Without prejudice we cannot even begin to comprehend. For example, if I try to read a Greek play without knowing Classical Greek, the text will just appear to me as incomprehensible scribblings. A first prerequisite is to have a basic understanding of facts, e.g. to know the letters and the words. This basic understanding (*Vorverständnis*) is a part of prejudice. (Gadamer 1960: II.II.1.c, p. 278.) When this is present, the actual process of understanding can begin. Here prejudice is crucial. Prior to reading the text, I will have formed an idea, accurate or not, of whether the author is to be trusted to tell the truth or whether he for some reason lies. If I read Aristophanes' plays as a literal description of ancient Greek society, my understanding will falter. To make sense of the plays, I need to have the proper prejudicial view that they do not literally tell the truth – that they exaggerate and distort it in order to amuse, and to criticize society. The task of hermeneutics is to distinguish between true and false prejudice. (Gadamer 1960: II.II.1.c, p. 282f.)

We can thus understand scientific theory as a part of our prejudices in the sense of Gadamer. We always have prejudices, whether we acknowledge them or not. Scientific theory is a form of prejudice that we are conscious of, have made explicit and have written down. What makes it prejudice – as opposed to simply judgment – is that we take the theory as a starting point whenever we encounter new evidence. Exactly because this explicit prejudice is not unconscious and taken for granted, we are able to have a scientific discussion about it. We need to keep in mind, though, that understanding is a continuous process. (Gadamer 1960: II.II.1.d.) In good scientific practice, theory is constantly confronted with evidence and revised. As understanding deepens, theory changes.

Science as Dialogue

Choosing good metaphors is an essential part of science. A metaphor for scientific understanding itself is that it is a dialogue with the evidence, the field. The scientist poses a question by looking at the evidence in a certain way. The 'answer' is the new understanding that the scientist gains, in turn leading to more questions, and more answers. The process of understanding is described in this way as an ongoing dialogue between scientist and evidence.

Is this metaphor justified? Gadamer himself points out that questions play a central role in understanding (Gadamer 1960: II.II.1.c, p. 283) and the entire last third of *Wahrheit und Methode* is devoted to examining the relationship between language and understanding. As we have seen earlier in this article, Perelman and Olbrechts-Tyteca consider private deliberation to be a special case of argumentation, which means that it can also be considered a special kind of dialogue.

s McCloskey writes in a treatise on the scientific rhetoric of the field of economics, science is not a privileged way of knowing, it is a way of speaking about things (McCloskey 1985: ch. 4, p. 67). This fits well with our characterization of science as a persuasive activity and as dialogue. We can then ask what characterizes scientific speech, what is the prototypical form of scientific argument. Here we can find a model in the classic rhetorical concept of epicheireme. Ordinarily, an argument[3] does not state fully and completely all of its

[3]In rhetorical terminology: enthymeme.

premises; something is left out and meant to be tacitly understood. The epicheireme is the fully elaborated argument where the major premises, minor premises and conclusion are stated in their entirety. (Kennedy 1984: ch. 1, p. 17.) This, then, is the ideal model for the scientific argument where everything is laid bare for other scholars to examine. Of course, in practice, most scientific writing is not composed of epicheiremes and most scientific investigations are not even epicheiremes in themselves; instead, they build upon each other. As an ideal though, the epicheireme is the rhetorical concept that best characterizes science.

If we view scientific understanding as a dialogue with the field, then method becomes the way of engaging in the dialogue, of posing questions and listening to answers. Good method, then, is to let the dialogue guide the method in such a way that we always engage in the dialogue in the most fruitful manner. Bad method is to choose once and for all to fix a method and let it impose arbitrary and unwarranted restrictions on the dialogue with no regard to how the said dialogue is evolving. In other words, both the subject of scientific research and the increasing scientific understanding need to be both the determinant for and to be above method. "Wie man sieht, ist das Problem der Methode ganz von dem Gegenstand bestimmt … " (Gadamer 1960: II.II.2.b, p. 297.)

Acknowledgements: This research is graciously funded by a grant from Svenska Kulturfonden. Thanks to my reviewers Frog and Sonja Peterson-Lewis for their insightful comments.

Works Cited

Aristotle. *Rhetoric.* Here cited according to: *Retorik.* Trans. Thure Hastrup. København: Museum Tusculanums Forlag, 2002.

Ehn, Billy 1981. *Arbetets flytande gränser: En fabriksstudie.* Stockholm: Prisma.

Gadamer, Hans-Georg 1960 [1965]. *Wahrheit und Methode.* 2nd edn. Tübingen: J.C.B. Mohr (Paul Siebeck).

Gormsen, Gudrun 1982. "Hedebonden: Studier i gårdmand Peder Knudsens dagbog 1829–1857". *Folk og Kultur* 1982: 58–101. Here cited according to: Offprint in the series *Ief småskrifter.*†

Højrup, Thomas 1995. *Omkring livsformsanalysens udvikling.* København: Museum Tusculanums Forlag.†

Kennedy, George A. 1984. *New Testament Interpretation through Rhetorical Criticism.* Chapel Hill: University of North Carolina Press.

Latour, Bruno, & Steve Woolgar 1979 [1986]. *Laboratory Life: The Construction of Scientific Facts.* 2nd edn. Princeton: Princeton University Press (1st edn.: Sage Publications).

McCloskey, Donald (Deirdre) N. 1985 [1998]. *The Rhetoric of Economics.* 2nd edn. Madison: University of Wisconsin Press.

Perelman, Chaïm, & Lucie Olbrechts-Tyteca 1958 [2008]. *Traité de l'argumentation: La nouvelle rhétorique.* 6th edn. Bruxelles: Editions de l'Université de Bruxelles.

† With an English summary.

ISBN 978-952-12-3226-8
ISSN 1239-1883

Espen Suenson How Computer Programmers Work

Espen Suenson How Computer Programmers Work

Espen Suenson How Computer Programmers Work