Trustworthy Software Systems

Research Methods in Software Engineering

2006

Wilhelm Hasselbring
Simon Giesecke
(eds.)

Contents

1 Preface **7**

2 The Role of Experimentation in Software Engineering **11**

2.1 Introduction 11
2.2 Research and Experimentation in SE 13
2.2.1 Empirical Knowledge versus Theoretical Knowledge 13
2.2.2 Research Paradigms and Methods 14
2.2.3 Data Collection Methods 17
2.2.4 Research in Other Fields 19
2.2.5 The Difference of SE 20
2.3 Current State of Experimentation in SE 21
2.3.1 Study by Tichy et. al. (1994) 21
2.3.2 Study by Zelkowitz et. al. (1997) 22
2.3.3 Further Analyses 23
2.4 Common Fallacies on Experiments 25
2.5 Future Direction of Experimentation in SE 28
2.5.1 General Recommendations 28
2.5.2 Repeatability and Families of Studies 30
2.6 Critical Reflection 31
2.7 Conclusions 33

3 Example of Empirical Research: n-Version Programming **37**

3.1 Introduction 37
3.2 N-version programming 39
3.2.1 Functional redundancy in hardware 41
3.2.2 Software reliability 42
3.2.3 Development process 43
3.2.4 Effectivity of NVP 45
3.3 Selected empirical studies 47
3.3.1 Early studies (77-78,80-83) 48
3.3.2 The Knight Leveson Experiment (86) 49
3.3.3 The 'Six Language' Experiment (86-88) 50

3.3.4 The ‘Second Generation’ Study (85-88,91) 51
3.4 Analysis and discussion 53
3.5 Conclusions 55

4 Integration of Qualitative and Quantitative Methods 61
4.1 Introduction 61
4.2 Categories of Research Objects in Software Engineering 62
4.3 Qualitative and quantitative research approaches 64
4.3.1 The research process 64
4.3.2 Data Scale Levels in Quantitative Research . . . 65
4.3.3 Comparison of quantitative and qualitative approaches 66
4.3.4 Research Approaches 68
4.4 Research Methods 70
4.4.1 Data Collection Methods 70
4.4.2 Data Analysis Methods 74
4.5 Combining Qualitative and Quantitative Approaches . . 76
4.6 Exemplary Study Designs 77
4.6.1 Example 1: Analysis of Architectural Design Decisions 77
4.6.2 Example 2: Action Research in a Development Process 79
4.7 Conclusion 81

5 Patterns in Building Architecture and Software Engineering 87
5.1 Introduction 87
5.2 Designing a Building in Civil Architecture 88
5.3 Architectural patterns and pattern languages 92
5.4 Software patterns and pattern languages 98
5.4.1 Software patterns forms 100
5.4.2 Software pattern languages 103
5.5 Similarities and differences 105
5.6 Conclusion 109

6 Legal Methodology and Research 119
6.1 Introduction 119
6.2 The Technique of Subsumtion 120
6.3 The Construction of Law - Hermeneutics 121
6.3.1 Literal Construction 123
6.3.2 Systematic Construction 124

6.3.3 Teleological Construction 126
6.3.4 Historical Construction 126
6.3.5 Additional Construction 127
6.3.6 Example: Pistol for Blanks as Weapon 128
6.3.7 The Construction of Treaties 130
6.4 Standard Juristic Argumentations 130
6.4.1 Analogy and Reverse Argumentation 130
6.4.2 Teleological Reduction 132
6.4.3 Argumentum A Fortiori 132
6.5 Research Aims in Law 132
6.5.1 The Descriptive Approach 133
6.5.2 The New Approach on the Basis of Existing Law 133
6.5.3 Developing New Laws 134
6.6 Conclusion 134

1 Preface

This collection consists of selected contributions to a graduate seminar conducted within the Graduate School "TrustSoft" at the Carl von Ossietzky University of Oldenburg, Germany. The Graduate School is funded by the German Research Foundation (DFG). It was established in April 2005 and will—over a period of nine years—support three cohorts of 14 graduate students in Computer Scienceand Law by a scholarship for obtaining a PhD degree. The first cohort of scholarship holders as well as externally funded PhD students are currently working on their PhD theses. They are supervised by thirteen professors from the Department of Computing Science and the Institute of Law. All theses are related to the topic of the Graduate School—Trustworthy Software Systems—, but the students and professors in the school have diverse backgrounds. To increase the benefit of the graduate school for its members as well as the scientific community, the graduate school aspires close cooperation of its members. One means of this cooperation was the conduct of two seminars in the starting period of the Graduate School, which resulted in the publication of two volumes of selected papers, one of which is the present volume on Research Methods in Software Engineering. The other volume provides survey papers on Dependability Engineering and is available from the same publisher.

Topic of the Graduate School Software increasingly influences our daily life, as we depend on an raising number of technical systems controlled by software. Additionally, the ubiquity of Internet-based applications increases our dependency on the availability of those software systems. Exemplarily consider complex embedded software control systems in the automotive domain, or IT systems for eGovernment and eHealth.

Fortunately, the rise of the software industry creates jobs for academically trained professionals and generates an increasing proportion of the national creation of value. However, this increased dependency on software systems intensifies the consequences of software failures.

Therefore, the successful deployment of software systems depends on the extent we can trust these systems. This relevance of trust is gaining awareness in industry. Several software vendor consortia plan to develop so-called "Trusted Computing" platforms. These current initiatives primarily focus on security, while trust is a much border concept. In fact, trust is given by several properties, such as safety, correctness, reliability, availability, privacy, performance, and certification.

Therefore, the graduate school will contribute to this comprehensive view on trusted software systems by bundling the Oldenburg computing science competences with those of computer law.

From a technical point of view, the research programme of the graduate school builds on and advances the paradigm of component-based software engineering. Besides the industrial relevance of components, components also constitute a more general paradigm employed successfully in the areas of formal verification (compositional reasoning), the prediction of quality properties, and the certification of software systems. The scientific methods to be developed in the graduate school vary according to the aspects of trust under investigation. For example, correctness is demonstrated by mathematical proofs while quantifiable quality properties, such as availability, reliability, and performance require analytical prediction models, which need additional empirical studies for calibration and validation. Generally, benefits of software engineering methods must be demonstrated empirically by case studies and controlled experiments.

Topic of the Seminar There is no generally accepted approach to research in software engineering, let alone computer science as a whole. Possibly because computer science is still a young science, a clear separation into distinct fields of research with respective research methods has not emerged (yet). Nonetheless, several different approaches to software engineering can be identified: E.g., some scientists are concerned with formal and semi-formal development methods, others are concerned with empirical software engineering. Different approaches correspond with understandings of software engineering as a branch or variant of mathematics, natural science, engineering, architecture, psychology and others.

Overview of the Contributions There were ten contributions to the seminar, five of which were selected for inclusion in this volume.

The first three papers are concerned with research methods in software engineering in a strict sense. First, a general introduction to "The Role of Experimentation in Software Engineering" is given by Heiko Koziolek. Then, the empirical research into a specific topic—N-version programming—performed by several researchers is presented by Matthias Rohr, which makes the long-term research more tangible in an exemplary way. The last of the three papers by Simon Giesecke and Thorsten Giesecke-Kopp on "Integration of Qualitative and Quantitative Methods in Software Engineering Research" combines theoretical considerations on the application of integrative research from general social sciences to Software Engineering research with exemplary thoughts on conducting such integrative research.

The next paper on "Patterns in Building Architecture and Software Engineering" by Marko Boskovic is concerned with the relationship of two distinct disciplines, Building Architecture and Software Engineering. Software Engineering, and especially the emerging sub-field of Software Architecture, use many metaphors targeted at Building Architecture. The concept of "patterns" originates from the field of Building Architecture and Urban Planning and eventually gained popularity in Software Engineering. The contribution outlines similarities and differences between the fields, which supports the further development of Software Engineering as a research discipline.

The final paper on "Legal Methodology and Research" by Daniel Winteler reflects the interdisciplinary setting of the Graduate School by presenting research methods from the Legal Sciences.

Oldenburg, November 2005

Prof. Dr. Wilhelm Hasselbring
Chair of the Graduate School "TrustSoft"
`hasselbring@informatik.uni-oldenburg.de`
Dipl.-Inform. Simon Giesecke
PhD Student of the Graduate School "TrustSoft"
`giesecke@informatik.uni-oldenburg.de`

2 The Role of Experimentation in Software Engineering

Heiko Koziolek <heiko.koziolek@informatik.uni-oldenburg.de>

Abstract

Research proposals need to be validated either by formal proofs or by applying empirical methods (e.g. controlled experiments). Many authors have pointed out that the level of experimentation in software engineering is not satisfactory. The quantity of experimentation is too low as a lot of software engineering publications do not contain empirical validation at all. Furthermore, the quality of software engineering experiments conducted so far is often weak, because no proper methodological approach is applied and statistical methods are misused. This paper provides an overview of experimentation in software engineering. First, the role of experimentation among other types of research is clarified. Several research paradigms are introduced, a classification of different types of experiments in software engineering is provided, and a comparison with experimentation in other research disciplines is drawn. Afterwards, the current state of experimentation in software engineering is presented with more detail. Some discussion points from various researchers about the situation of experimentation are summed up.

2.1 Introduction

Any research proposal in computer science needs to be validated properly to check it claims, improvements and also its applicability in practice. To conduct such a validation, either the proposal has to be proven formally or empirical methods have to be used to gather evidence. The fact that formal proofs are only seldom possible in software engineering (SE), has lead researchers to emphasise the role of empirical methods, which are traditionally more common in disciplines like physics, social sciences, medicine, or psychology.

Several different empirical methods are known. Quantitative methods try to measure a certain effect, while qualitative methods search for the reasons of an observed effect. Examples for quantitative methods are experiments, case studies, field studies, surveys and meta-studies. Examples for qualitative methods are interviews and group discussions.

The *controlled experiment* is the method with the highest degree of confidence into the results. In such an experiment researchers try to control any variable influencing the outcome, except the variable they want to analyse. For example, this involves testing a product or method with a larger group of persons, so that the differences in the qualifications of the participants can be reduced by averaging and thus do not influence the result. A less strict method is the *case study*, in which a (possibly artificial) example is analysed and initial interpretations of the observed effects can be drawn, but the results are normally not generalisable to other examples. *Surveys* include searching the literature or passing out questionnaires to experts to gather evidence. *Meta-studies* analyse other studies and try to gain knowledge by comparing different approaches. A more detailed introduction into empirical methods in computer sciences can be found in several books, recent such as [WRH$^+$00, Pre01, JM01].

Many authors have pointed out that the level of experimentation in SE is not satisfactory. The quantity of experimentation is weak, as a lot of researchers are reluctant to validate their approaches empirically. The quality of experimentation is also often not sufficient, as statistical methods are used inappropriately or it is neglected to draw proper conclusions from experiments. Based on these observations this paper discusses the role of experimentation in SE in more detail.

This paper is organised as follows: The following section 2 contains an overview of general research approaches and describes how research is conducted in other disciplines. Afterwards, the focus shifts specifically to experimentation and its application in SE. Section 3 analyses the status quo of experimentation in SE, while section 4 sums up several common fallacies on this topic. Section 5 summarises future direction of experimentation and presents some ideas how to improve the conduction of empirical studies. Section 6 contains a critical reflection of some of the statements found in the literature analysed before. Finally, Section 7 concludes the paper.

2.2 Research and Experimentation in SE

Evolution in science disciplines is based on encapsulating experience into models and their verification and validation based on experimentation [Bas96]. The science process is a cycle of applying ideas, analysing results and collection feedback. SE requires the same cycle of model building, experimentation and learning to become a matured scientific discipline.

Scientific research in SE is conducted under the assumption that "if we look long enough and hard enough, we will find rational rules that show us the best ways to build the best software" [Pfl99].

In the following, first different levels of knowledge are presented before four general research methods and the experimentation/learning cycle are introduced. To conduct empirical research, multiple data collection methods can be applied, which are summed up in the third subsection. Older scientific disciplines like physics and psychology have developed their own experimental paradigms, which are shortly described in the fourth subsection. This section concludes with an argumentation why SE is different from other scientific disciplines and why an own experimental paradigm has to be developed.

2.2.1 Empirical Knowledge versus Theoretical Knowledge

Three levels of scientific investigations to gain knowledge can be identified [JM01]:

- **Survey inquiries** observe an object or process and try to find out, which variables affect other variables. The effect is neither quantified nor explained. Although other disciplines have identified the relationships between important variables, most of the influencing variables in SE are still not known.
- **Empirical inquiries** aim at quantifying how variables affect other variables. The goal of such inquiries is to construct an empirical model. According to Popper [Pop59] empirical inquiries cannot prove any theory, they can only fail to falsify it. From

this viewpoint scientific knowledge is nothing more than a system of untrue statements and claims that are provisionally true as long as they are not contradicted. This insight should always be kept in mind when reasoning about experimentation. Apart from the inability to prove a theory, empirical inquiries are also not able to create an understanding of the observed phenomena.

- **Mechanistic inquiries** represent the highest level of scientific investigation. The aim is to explain why variables affect other variables in the observed manner and to construct a theoretical model. Such a theoretical model contributes to the understanding of a phenomena and additionally provides a basis for extrapolation, which is not possible with an empirical model. Thus, predictions can be made about certain phenomena based on an theoretical model. Theoretical models also provide a stricter representation of the response function to a research question.

As survey inquiries have hardly been conducted in SE and not enough background knowledge has been collected, it is not yet possible to construct theoretical models in SE. Empirical models are a necessary step towards theoretical models for SE.

2.2.2 Research Paradigms and Methods

Basili distinguishes between two main research paradigms [Bas93]. The experimental paradigm includes the scientific method with the engineering method and the empirical method as subsets, while the analytical paradigm includes the mathematical method:

- **Scientific method:** ”Observe the world, propose a model or a theory of behaviour, measure and analyse, validate hypotheses of the model or theory, and if possible repeat the procedure.”
  - **Engineering method:** ”Observe existing solutions, propose better solutions, build or develop, measure and analyse, and repeat the process until no more improvements appear possible.”
  - **Empirical method:** ”Propose a model, develop statistical/qualitative methods, apply to case studies, measure and analyse, validate model and repeat the procedure.”

- **Mathematical method:** ”Propose a formal theory or set of axioms, develop a theory, derive results and if possible compare with empirical observations.”

The experimental paradigm is an *inductive* approach: it uses empirical evidence to formulate generalised knowledge from the observations made. On the other hand the analytical paradigm is *deductive*: it takes known theories and provides new evidence for special cases. In deductive reasoning evidence provided must be a set about which everything is known before the conclusion can be drawn. Since not all variables of SE (software artefacts, development processes, people, environment) are known formally, it is difficult to fully apply such an approach in this discipline. Thus, Basili concludes that ”the inductive paradigm might be best used when trying to understand the software process, product, people, environment”.

According to Glass [Gla94] the engineering, empirical and also the mathematical method are applicable in computer science, however the scientific method (although a superset of the first two methods) is problematic because of the phrase ”observe the world”. Software is intangible and has an invisible nature, so the phrase should be replaced by ”observe the problem space”.

To clarify the relationship between deduction and induction, Juristo et. al. [JM01] illustrate the iterative *experimentation/learning cycle* (Figure 2.1). Deduction proves something, induction shows that something is operational and abduction suggests that something could be. The cycle is composed as follows: a preliminary hypothesis is made about some phenomena. A process of deduction from the hypothesis leads to results, which are compared against real data. Discrepancies about the deduction results and reality can lead to a new hypothesis via induction. With the new hypothesis the cycle starts again.

The two variations of the experimental paradigm are rather contrary [Bas93]. The engineering method is *evolutionary*, it takes existing solutions and tries to improve them. For example the knowledge that a method is less cost-intensive than another or that a new tool preforms better than its predecessors might be the result of such an approach. Opposed to that, the empirical method can be a *revolutionary* approach not based on existing solutions. For example the proposal of a new method or model and the evaluation of its effect to the software development process or software products is a result of this approach. For both approaches careful analysis and measurements are critical for

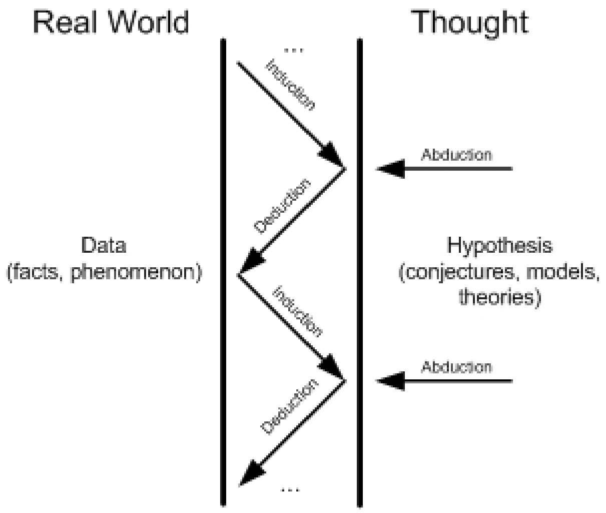

Figure 2.1: Iterative learning cycle [JM01]

their success.

With these paradigms research activities and development can be differentiated. If someone is neither applying the experimental, nor analytical paradigm, his work cannot be considered as research. For example, constructing a system or tool alone without validation is development and not research. Only if an understanding is gained how a system works, or why it is useful, and this is validated by experimentation, then a work qualifies as being research.

In Glass' [Gla94] critique about the state of the art of computer science research, the author sums up some observations about the research methods introduced above. Few studies have used the scientific method. Unfortunately, it seems uncommon for computer scientists to formulate and validate hypotheses and to do evaluation in an iterative cycle. The engineering method has been used more frequently, as sometimes proposals aim to improve existing solutions. Yet the cycle ”repeat, until no further improvement is possible” can hardly be found in most approaches. The empirical method is under-represented and only used by a few researchers. Glass criticises that most researchers have no interest in this approach, and if they do, they tend to use only student experiments, not industrial case studies. The mathematical method is still the most common in computer science because of the mathematical heritage of the discipline. Within this method it is extremely rare that researchers compare their results with empirical observations. Glass concludes to characterise the most common research method as seriously flawed in practice.

2.2.3 Data Collection Methods

Data collection from reality for the experimental/learning cycle can be done in multiple ways. A taxonomy by Zelkowitz et. al. [ZW98] lists 12 different data collection methods:

- **Observational Methods:** project monitoring, case study, assertion, field study
- **Historical Methods:** literature search, legacy data, lessons-learned, static analysis
- **Controlled Methods:** replicated experiment, synthetic environment experiment, dynamic analysis, simulation

An overview of the empirical methods most common in computer sciences and SE (controlled experiments, case studies, survey, meta-studies) can be found in [Pre01, WRH$^+$00].

Basili [Bas96] provides a classification for several kinds of experiments, which are identified by:

- Type of results
  - Descriptive: relationships among variables have not been examined
  - Correlational: variation of dependent variables is related to variation of independent variables
  - Cause-effect: independent variable is only possible cause for variation of dependent variables
- Type of participants
  - Novice: inexperienced students
  - Expert: practitioners experienced in the study domain
- Type of environment

  - In vivo: in the field (i.e. the software industry) under realistic conditions
  - In vitro: in the laboratory (i.e. the university) under controlled conditions

- Level of control
  - Controlled experiment: typically in vitro, mostly with students, strong statistically confidence in results, expensive, difficult to control
  - Quasi-experiment: typically in vivo, with practitioners, qualitative character
  - Observational study: no treatment or controlled variables, possibly no set of study variables defined beforehand

Furthermore Basili examines, that in general two patterns of experiments have been performed so far: human factor studies and project-based studies. *Human factor studies* are usually cause-effect studies and try to find out how software engineers perceive and solve problems. *Project-based studies* are often correlational or descriptive studies and aim to help practitioners building models of software product and processes.

Still, there is a disagreement about the right methodology in empirical SE, as discussed in a panel session on the ICSE 2003, which was summed up in a position paper by Walker [WBN$^+$03]. Tichy favours controlled, quantitative, and statistically-analysable experiments, but like Popper he also states that no "silver-bullet experiments" exists, which could provide a final answer to a research question. Kitchenham also favours quantitative studies, but highlights the value of field studies in an industrial environment. For Seaman, qualitative studies are a valuable addition to quantitative studies. On one hand they can be used to replicate a study with a different methodology, thereby improving the trust in the results. On the other hand qualitative studies can help reducing the complexity of SE experiment resulting from the human factor, because they were originally developed to analyse human behaviour. Murphy (also in [MWB99]) points out that different evaluation treatments have to be applied according the degree of maturity of a technology under analysis. Briand adds that the discipline of SE needs to develop its own body of experience and strategies like other disciplines. Additionally, he states that qualitative and quantitative methods actually complement each other. Furthermore, in his opinion controlled experiments (with often high internal, but low external validity) and field studies (with often low internal but high external validity) both are necessary to create a body of evidence. Notkin states that empirical evidence is not always needed to transfer research results into practice. Because of the uniquely high rate of change in SE, the application of research results must be arranged differently. Although not each minor solution to a problem needs to be evaluated empirically, for deeper solutions this is necessary ("heavy claims require heavy evaluation").

2.2.4 Research in Other Fields

The science approach of the early Greeks was to observe something and then create knowledge by a chain of logical thought. Experimentation was hardly used by the Greeks for scientific purposes. One of the first popular scientific experiment in science history was performed by Galileo Galilei, who allegedly dropped balls from the tower of Pisa to study the free fall. Over the course of history each research field has developed experimental paradigms. Basili [Bas96] argues that SE needs to follow this model of other physical sciences and analyses other disciplines:

- **Physics:** In physics researchers are usually either theorists or experimentalists. Models are built by theorists to explain the universe, while experimentalists observe and measure the environment with the goal of validating a theory or exploring new areas.

- **Medicine:** Medical professionals are divided into researchers and practitioners, where the researcher aims at understanding the human body and the practitioner aims at curing other people. In medicine, knowledge is often built by feedback from the practitioner to the researcher.

- **Manufacturing:** In manufacturing, the relationship between process and product characteristics is generally well understood. Progress is made by experimenting with varying processes, building models of what occurs and measuring the effects on an overworked product.

- **Psychology:** In psychology experiments have been used since the end of the 19th century, to base knowledge not on the personal experience of single human beings, but to check claims systematically and methodological controlled. Experiments are used to explain relationships of different variables and to predict certain events.

2.2.5 The Difference of SE

However, SE embodies *different* characteristics than other disciplines. Unlike in physics, but like in manufacturing, computer scientists are able to manipulate the essence of their product, because software and the corresponding development processes can easily be modified. Yet unlike in manufacturing, the problem in SE is not production (because software can easily be copied), but development, because each new software product is different from the last [Bas96]. This implies that the mechanisms for model building are different in SE than in other disciplines.

A lot of computer science methods and technologies are influenced by human behaviour. Unlike in physics, the same experiment can produce different results based on the people involved. The human factor in SE prevents researchers from directly applying the causal-deterministic model of for example physics, because psychological and social influences are not completely determined by preceding facts [JM01]. Pfleeger [Pfl99] notes that researchers still often view SE as a natural process like in physics with deterministic effects to causes. In fact software development is on large parts a social process, which leads to stochastic effects to causes. The occurrence of an effect can be described best by a probability function. For example, a software company with a high development certification level is not guaranteed to develop high quality software. Yet, the customers are assured, that they will actually get good software from this company with a high probability.

Another difference in SE is the amount of variables, which has to be considered on the outcome of an experiment. Software products, processes, the goals to be achieved and the context are all variable [Bas96]. A result of the high variability in SE is a lack of useful models for reasoning about software processes and products can still be observed. Because researchers have not been able to construct mathematical tractable models, like it is possible in physics and manufacturing, they have the tendency to not build models at all. Few attempts have been made to construct non-mathematical models, heuristics, and models representing simple variable relationships, which for example are common in medicine.

2.3 Current State of Experimentation in SE

After introducing research approaches in the preceding section, the status of experimentation in SE shall be analysed in the following section. Two meta-studies about experimentation in SE are often cited and will be described more detailed. Additionally, several authors have written articles about the problems and trends in experimentation and some of their remarks will be summed up afterwards.

2.3.1 Study by Tichy et. al. (1994)

Tichy et. al [TLPH95, LHPT94] conducted a quantitative study surveying over 400 research articles and looked for the amount of experimental validation. Included in this study for instance were articles from the ACM Transactions on Computer Systems (TOCS), ACM Transactions on Programming Languages (TOPLAS), IEEE Transactions on Software Engineering (TSE), and a random sample of articles published from the ACM in 1993. Also included for comparison with other disciplines were journals entitled Neural Computing (NC) and Optical Engineering (OE). The area NC was chosen because it overlaps with computer sciences (CS) and has a similar level of youth like CS. OE in contrast was chosen, because it has lots of immediate applications like CS, but also has a longer history.

The metric used for measuring the amount of experimentation in design and modelling articles was simply the physical space the authors reserved for experimental evaluation. This metric was chosen because the physical space correlates with the importance the authors attached to their experimental evaluation and also the quality of the experiments.

Over 40% of CS papers about design and modelling completely excluded experimentation, in software engineering papers the number was even higher at 50%. In contrast, only 14% of the NC and OE articles did not contain experimental evaluations. The CS papers contained a significantly lower amount of purely empirical papers than in NC and OE. Hypothesis testing articles were rare in all samples

(1%). The number of papers dedicating more than 20% of their space to experimental validation was significantly lower in CS (31%) than in NC and OE (67%).

The authors of the study used these results to disprove a common perception in CS that the lack of experimentation is resulting from the youth of the discipline. During the conduction of the study the field of NC was only six years old, but had a much better level of experimentation established that was comparable to the much older discipline of OE. Although NC overlaps with CS, computer scientist are actually a minority among the NC community.

An explanation the authors offer for the lack of experimentation in CS, is the lack of well established measuring techniques. Another problem is the fact, that a lot of conference committees accept papers without experimental validations. The authors encourage conference committees to set higher standards for accepting papers and also viewing empirical work as first class science.

2.3.2 Study by Zelkowitz et. al. (1997)

Zelkowitz and Wallace [ZW97] conducted a similar study like Tichy and analysed 612 SE papers and 137 papers from other sciences (for example physics, management science and behaviour theory). SE papers were from three different years (1985, 1990, 1995) and were taken from IEEE Transactions on Software Engineering, IEEE Software and the Proceedings of the International Conference on Software Engineering (ICSE) of the corresponding year. The articles from other disciplines were taken from various journals of the respective sciences from the years 1991-1996.

Similar to the study by Tichy, here, about one-third of the SE articles had no experimental validation at all. But the percentage was decreasing over the years (1985: 36.4%, 1990: 29.2%, 1995: 19.4%). About one-third of the articles contained only assertions as experimental validation. In assertions the developer of a technolgy becomes the experimenter and the subject of the study and usually does not perform any kind of control. These are often preliminary forms of validation and are potentially biased. 5% of the articles used simulation techniques for result validations, while the other data collection methods (see section 2.2.3) were found in only 1-3% of the articles.

Concerning other sciences, the authors observed specific patterns of experimentation approaches in each fields. Physical publications commonly contained dynamic analyses and simulations. In psychological articles, mainly replicated and synthetic experiments could be found, while anthropological papers used passive techniques on historical data like legacy data and literature search.

Additionally to the confirmation of Tichy's result, that experimental validation is under-represented in CS, in this study the authors found out, that one-third of papers which actually contained experimental validation only did it in an insufficient and weak way (with assertions). This means that authors are starting to realise the need for experimental validation, yet still do not do it with strong methods. Encouraging is the fact, that the authors observed a development with fewer papers without experimentation comparing the numbers from 1985, 1990 and 1995.

Some qualitative observations of the study are mentioned in [ZW98]. Often, researcher do not succeed in stating their goals explicitly and clearly. Sections containing experimental validation were sometimes not labelled accordingly and the terms "case study", "controlled experiment" were used very loosely with different meanings.

2.3.3 Further Analyses

Fenton et. al. [FPG94] also discuss the current state of experimentation in SE. Although Basili et. al. [BSH86] had made several recommendation for experimentation in SE, most of the experiments documented later did not follow their recommendation. The authors make five observations:

- **Intuitive research:** Too many concepts in SE are still based on so-called "analytical advocacy research", meaning that concepts have been described, analysed for their benefits informally and recommended for practical use but that rigorous, quantitative experimentation of them is missing. For example, formal methods in SE, which use mathematical precise specifications of software (e.g. the language Z [Spi88]) to prove its correctness, have been used for decades. It has been claimed that these methods are cost saving and that they reduce the amount of product failures. However, quantified evidence to support these claims does not appear in the respective publications. Nevertheless, counter-examples, in which methods have become a standard in SE because of empirical analysis, also exist (for example the use of inspections to uncover code-defects).

- **Experimental design flaws:** Experimental designs in SE are still often flawed. For example, Shneiderman showed with an experiment that program flowcharts did not increase the comprehension of program behaviour better than pseudocode. Years later, Scanlan made a comparison of flowcharts and pseudocode and investigated the amount of time needed to understand a program and the amount of time needed to make appropriate changes to it. In both dimensions using flowcharts was superior to using pseudocode. Fenton et. al. claim that flaws in experimental designs are a result of the thin representation of experimentation, statistical analysis and measurement principles in the computer science curricula of universities.

- **Toy analysis:** Empirical investigations in SE too often only analyse "toy projects in toy situations". The costs of large-scale industrial experiments lead most researchers to only experiment with student groups on small artificial examples. It is often unknown how the results of such experiments scale up and whether the results can be generalised. Because of this, a better cooperation of research institutes and the software industry is recommended.

- **Inappropriate statistics:** Many experiments use inappropriate measures and misuse statistical methods. For example, several experiments can be found that use a nominal scale for their data and apply means and standard deviations to that data, although this data can only be analysed in terms of frequency and mode.

- **No long-term view:** Some experiments are conducted over a too short period of time and thus omit long-term effects. For example, the comparison of the benefits of the programming languages Ada and Fortran at first showed that Ada programmers were less productive and delivered programs with less quality than the Fortran programmers. However, the study did not take into account that Ada has a long learning curve and that the actual benefits of Ada can only be observed after the programmers had at least implemented three projects with this language.

Perry et. al. [PPV00] mention that the amount of empirical studies and also their quality is rising over the last 10-20 years. Empirical validation is still not a standard part of research papers, yet a powerful addition. Especially in the testing community empirical studies are quite common. US funding agencies such as the National Science Foundation (NSF) and the National Academy of Sciences are realising the importance of empirical studies and are sponsoring for example the "Experimental and Integrative Activities" program or a workshop on the topic of statistics and software engineering [Pre96]. Furthermore, Perry et. al. observe, that the awareness for empirical studies is growing, as can be seen in tutorials, panels and presentations at major conferences such as ICSE, FSE and ICSM.

The authors notice that the discipline is still suffering from several systemic problems. Most of them originate from misunderstandings about the reasons for experiments and their proper conduction. Many researchers only use empirical studies retrospectively for early validation of their research results, but do not remember, that studies can also be used pro-actively to direct research.

Additionally, it is often tried to make the perfect study without flaws, which in the authors' opinion is impossible. Instead, more interest should be directed to the conclusions drawn from studies, which is often a weak part of such studies. Many studies only observe the obvious, thereby encouraging an argument by intuition approach. More studies should investigate unintuitive relationships.

Data collection and analysis is sometimes done very extensive and precise in empirical studies today, but the use of data to answer questions is neglected. In these cases no conclusions are drawn making it difficult to learn anything from such a study. Often studies simply lack hypotheses, do not ask insightful questions and therefore contain no well-defined end. Because the drawing of conclusions is disregarded, researchers are reluctant to generalise their results. But without generalised results it is hard to make any progress in SE.

Concluding this section, it can be stated that experimentation is still under-represented in SE, but a positive trend can be observed. Although the awareness for the need of experimentation is starting to increase, lots of the experiments conducted today are still flawed in their designs and in the ability to draw proper conclusions.

2.4 Common Fallacies on Experiments

Researchers often try to justify their lack of experimentation. Tichy [Tic98] wrote an article refuting common comments given by computer scientist when they are asked about why they neglect the experimental validation of their research results:

- **"The Traditional scientific method isn't applicable":** A major difference of computer sciences to other disciplines is that information is neither energy nor matter. Several researchers conclude that traditional scientific methods are thus not applicable in computer science. This is a fallacy, because researchers can use the same methods as in traditional sciences, like observation of phenomena, formulation of explanation and theories and testing, to gain an understanding on the nature of information processes. The fact that the information itself is different from energy or matter does not alter the treatment of information processes from other sciences.

- **"The current level of experimentation is good enough":** This can be refuted with the studies mentioned earlier [TLPH95, ZW98], which analysed the amount of experimentation in SE papers. Experimentation has the purpose of reducing uncertainty about untested theories, may serve to start new areas of research and eliminates fruitless approaches thus directing the science process.

- **"Experiments costs too much":** Tichy admits, that experiments cost a lot of effort, but that this effort is justified, if the research question asked is of importance and the gained insights outweigh the costs. For example, Isaac Eddington undertook an expensive expedition to West Africa in 1919 to observe a total solar eclipse. Doing so he was able to check Einstein's theory that gravity bends light when passing large stars. In this case the experiment required a lot of effort, but the importance of the answer to the research question was tremendous.
  A way to overcome the high costs of experiments might be involving software industry. Companies may be able to get competitive advantages out of experiments and thus may be interested in sponsoring such research. Furthermore, Tichy mentions that a

possible cheaper substitute to experiments might be simulations techniques.

- **"Demonstrations will suffice":** Demonstrations only provide proof of concepts, but not hard evidence to research questions. Demos only illustrate a potential and are dependent on the authors to generalise the results. The need for replication of results is not stressed by demonstrations.

- **"There is too much noise in the way":** Often researchers complain, that too many variables have to be controlled in an experiment and that the results are hard to interpret because they are swamped by noise. Tichy advises researchers to use benchmarks to simplify repeated experiments. Noise created by human subjects may be reduced by using techniques from areas like psychology and sociology.

- **"Progress will slow":** As experiments require a lot of effort to be conducted, many argue that experimenting slows the flow of ideas down, if every idea must be extensively validated with experiments. Contrary, Tichy replies, that experimenting might in fact speed up the science progress, because fruitless approaches would be discarded earlier and science would focus more on the most promising approaches.

- **"Technology changes too fast":** Problematic in computer science is the fact that technology changes so fast that ideas might not be relevant anymore if experimental validations are finally available. If a research question falls into this category it is obviously formulated too narrow. Experimentation techniques should be applied to deeper, fundamental questions and not to the newest fashion of tools and methods.

- **"You will never get it published":** Theoretical computer scientists usually expect perfection and absolute certainty in results published. But because experiments are always flawed in some way [PPV00], it is difficult to get experimentation papers accepted at important conferences. Tichy replies that several journals exists, that would welcome more experimental papers, but that the supply is actually too low.

Tichy also states that intuition and personal experience is not sufficient to claim the applicability of a product or process in a matured engineering discipline. Several examples are known in computer science in which intuition falsely favoured an opinion (the need for meetings in code reviews, the much lesser failure probabilities of multi-version programs). It is also dangerous to simply trust well-known experts and not to demand hard evidence of their claims. A fundamental precondition of science is, that it is based on healthy scepticism.

2.5 Future Direction of Experimentation in SE

As the need for experimentation in SE has been emphasised before and problems of the discipline have been examined, the following section deals with the possible future of experimentation. Most of the authors, who have complained about the lack of experimentation in SE, have also made recommendations about what could be improved (next subsection). An approach still neglected in SE experiments is the repetition of empirical studies and the creation of families of studies to answer larger research questions (last subsection).

2.5.1 General Recommendations

Basili [Bas96] advocates the use of the Goal-Question-Metric (GQM) method (ref Klaus Krogmanns Ausarbeitung) for a better direction of experimentation. Especially the level of sophistication of the goals towards experiments are designed must improve. The results of experiments need to be shared more and the results by one group need to be used by other groups. An organisation called International Software Engineering Research Network (ISERN) has been established especially for this purpose. A forum for empirical researchers is provided by the International Journal of Empirical Software Engineering by Springer.

Perry et. al. [PPV00] state that in order to improve the quality of empirical studies in SE, more clarity about the goals of the studies is needed. Researchers must ask important and sophisticated questions and establish causal, actionable and general principles. Causality implies constructing a chain of factors influencing each other, while actionable principles require factors which can be controlled effectively.

Factors also have to be so much general, that the results are relevant to multiple persons in multiple contexts.

Credible studies have a high degree of confidence in the results. To create such studies the internal, external and construct validity have to be checked explicitly. Enough data has to be published to let other researcher recheck the validity of a study. Additionally, studies should always be conducted on the basis of hypotheses. If a study is not sufficient enough to create a causal relationship of factors, several alternative explanations may be proposed and data from other source might be used to discredit certain alternatives.

Glass [Gla94] emphasises the need for a close cooperation between practitioners and researchers to improve the state of research in computer sciences. For him, the Software Engineering Laboratory (SEL) [BCM+92] as an institution involving academic (University of Maryland), industrial (Computer Sciences Coporations) and governmental (NASA-Goddard, as sponsors) organisations is an exemplary model which should be replicated by other researchers and countries to stimulate the exchange between practice and research.

Recently, Kitchenham et. al. [KPP+02] have proposed a set of guidelines for empirical research in SE, which are based on a review of guidelines for medical researchers. The guidelines cover six topics:

- **Experimental context:** It is important for studies to include information about the industrial context they were conducted in. The research hypotheses have to be discussed and also how they have been derived. The main guidelines about the context are to state and discuss the goal of the study and to include sufficient details for researchers as well as practitioners.

- **Experimental design:** Studies must describe the population under analysis (e.g., students, practitioners) and the sampling technique used for it. It has to be documented which interventions have been conducted and which method has been used to reduce bias and to determine the sample size.

- **Conduct of the experiment and data collection:** Because the measures for the outcome of a study are not standardised they have to be documented in sufficient detail. The entity, attribute, unit and counting rules of measures must be defined. Quality control methods used on the data and data about subjects who dropped out of the study should be presented.

- **Analysis:** Procedures used to control multiple testing should be specified and a sensitivity analysis should be performed. Additionally, the data must not violate the assumptions of the tests used on them. Appropriate quality control methods should be applied for verification of the results.

- **Presentation of results:** A reference should be provided for all statistical procedures used and the statistical package used should be reported. The magnitude of effects and the confidence limits of quantitative results should be presented as well as confidence levels. If possible, the raw data should be provided with the study, so that independent researchers can draw their own interpretations from them. Descriptive statistics should be used in an appropriate way. Graphics can increase the degree of understandability of a study if used correctly.

- **Interpretation of results:** Researchers should differentiate between statistical significance and practical importance. The type of the study needs to be define and limitations of it should be discussed.

2.5.2 Repeatability and Families of Studies

Perry et. al. [PPV00] note, that it is often not possible to design studies for complex issues and difficult questions, especially considering the effort and costs needed for empirical studies. In this case it is necessary to focus on smaller problems and to create multiple studies, which results possibly can be combined to answer more deep questions. The credibility of empirical studies is improved drastically if other researcher are provided with enough information to reproduce the results.

The same is mentioned by Lewis et. al., who state: "The use of precise, repeatable experiments is the hallmark of a mature scientific or engineering discipline." [LHKS92]. In the history of science several examples for experiments can be found, which could not be repeated by other scientists as summarised by Juristo et. al. [JM01]. For example, the psychodynamic theories developed by Freud are criticised as being unscientific because they cannot be verified or disproved empirically. In 1989 two physicians claimed they had successfully conducted a cold fusion during an experiment, but after they published the design of their experiment, other scientist were not able to reproduce their result.

Apart from external replication run by independent researchers, also internal replications run by the original experimentator are necessary to improve the trust in an experiment. This might be hard to do in practice because a software project cannot be repeated precisely, yet it is not an excuse for not experimenting at all.

Basili [BSL99] also emphasises the importance of replicated experiments . Too many SE experiments stay isolated and do not lead to a larger body of knowledge. This can only be achieved by a set of unifying principles that allows the combination and generalisation of results. The authors propose a framework including the GQM method for a family of experiments.

Additionally they classify three major categories for replicated experiments:

- **Replication without a variation of the research hypothesis:** These include strict replications and replications that alter the way an experiment is run. Strict replications aim at a very accurate replication of the original experiment and ensure the repeatability of an observation. Replications that are run with a different experimental set-up than in the original case but with the same hypothesis might reveal flaws of the internal validity of an experiment.

- **Replication with a variation of the research hypothesis:** Included are variations of independent, dependent and context variables. Varying *independent* variables means changing variables intrinsic to the object under study, for example changing an attribute of the process or product under study. This form of replication is only possible if the experimental design has been made explicit by the original authors. *Dependent* variables are variables regarding the focus of the study. An example of a replication with a change of the dependent variables may be using other metrics or measurements for the effects that are be studied. Varying the *context* of an experiment might help in identifying influencing environmental factors. For example an experiment might be carried out with a group of professionals opposed to a group of students to analyse the influence of personal experience to the studied effects.

- **Replications that extend the theory:** Replications in this category change a large part of the process or product under analysis to determine its limit of effectiveness.

2.6 Critical Reflection

After reviewing the literature about the role of experimentation in SE, some critical remarks about this topic are summed up in this section.

Tichy tried to refute common comments by researchers, who neglect empirical evaluations of their work, and tried to encourage scientists to put more emphasis on experimentation. However, overcoming the organisational effort for proper controlled experiments is still a major problem. Lots of research is conducted by PhD-students, who simply do not have the means and time to conduct elaborate experiments. Including the software industry into experimentation as suggested by Tichy is also very difficult, because practitioners are hard to motivate to put resources and money on testing unproven new methods, if the direct value for their customers cannot be made evident. Also, as pointed out by Tichy, it is not necessary to evaluate every small research proposal with large controlled experiments. But criterias for necessity of experiments are still informal and hard to determine.

One problem, sometimes mentioned by the authors cited here, is the researcher's inappropriate knowledge of empirical methods in software engineering. This point is clearly underestimated by the empirical software engineering community. The researchers' knowledge is inappropriate because most universities do not teach empirical methods, and such courses are not part of the standard curriculums. If experiments are conducted, the methods of experimentation have often only been learned ad-hoc by the researchers, if they have laid emphasis on a methodological approach at all. Young researchers are seldomly taught the proper conduction of an experiment and have to collect experience about it on their own. The facts that books about methods like experiments or case studies for SE have appeared only recently (in the last 5 years) and that the methods are still not established well enough also contribute to this situation. Additionally the inappropriate application of statistical methods in experimentation can also be seen as a result of the missing courses in the computer science curriculum.

As seen in the literature analysed above, researchers often criticise experiments that are carried out with students as the subject under analysis, claiming that the results are not transferable to experienced practitioners. Apart from the organisational difficulties and high effort of including practitioners, there are other reason to counter this criticism. First, experienced under-graduate students often become practitioners just a short time later, so that their qualification and performance is not as different to practitioners as it seems. Second, the experience by practitioners might actually distort the results of an experiment, because specialists might have an unusual advantage over common developers. Thus, the results obtained with some experienced specialists might not be generalisable for the average developer.

Another reason why researchers are reluctant to experiment, which has been neglected in the literature reviewed, is simply the fact that some researchers do not like to experiment and consider it an inconvenient but necessary task. Most scientists rather want to create new ideas than spending time on validating old ones. Computer scientists in particular are more interested in technical problems and how to solve them. If their solutions are intuitively correct, most of them do not bother to conduct further evaluation on them. They even might be scared to prove their solutions empirically wrong. A stronger motivation for experimentation has to be created, possibly by documenting popular experiments in SE, which revealed unexpected results.

In the future experiments will be conducted with a higher quality and also more experiments will conducted. But whether a level of experimentation can and needs to be established in SE like it has been in other disciplines (physics, medicine, psychology) remains doubtful.

2.7 Conclusions

In this paper the situation of experimentation in SE in the past, present and future has been presented. "Experimentation is central to the scientific process" [Tic98]. This statement is especially true for SE, because most research proposals cannot be proven formally. It has been discussed that experimentation is vital for SE to become a matured scientific discipline. Multiple data collection methods are known for empirical SE, controlled experiments are the method with the highest degree of confidence in the results. When analysing experimentation in SE, the special characteristics of the discipline (like the human factor and the high variability) have to be kept in mind.

In the past, experimentation in SE has not been sufficient as two studies from the mid-nineties have shown, although a positive trend can be observed. The reluctance of researchers to experiment can be refuted with multiple arguments. Researchers in SE should understand that empirical validation as an essential part of their work and that their proposals are not valid unless empirical evidence has been provided. In the future, hopefully, not only the quantity but also the quality of experiments can be improved. It should be easier to conduct good empirical studies in SE because an increasing body of literature about the topic has been published (e.g. [WRH+00, Pre01, JM01]. Additionally, replicated experiments and families of studies should help to create larger bodies of knowledge.

Bibliography

[Bas93] Basili, V. R.: The Experimental Paradigm in Software Engineering. In: *Proceedings of the International Workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions*, London, UK: Springer-Verlag, 1993, ISBN 3-540-57092-6, pp. 3–12

[Bas96] —— The Role of Experimentation in Software Engineering: Past, Current, and Future. In: *ICSE*, 1996, pp. 442–449

[BCM+92] Basili, V.; Caldiera, G.; McGarry, F.; Pajerski, R.; Page, G.; Waligora, S.: The software engineering laboratory: an operational software experience factory. In: *ICSE '92: Proceedings of the 14th international conference on Software engineering*, New York, NY, USA: ACM Press, 1992, ISBN 0-89791-504-6, pp. 370–381, doi: 10.1145/143062.143154

[BSH86] Basili, V. R.; Selby, R. W.; Hutchens, D. H.: Experimentation in software engineering. In: *IEEE Trans. Softw. Eng.* 12 (1986), № 7, pp. 733–743, ISSN 0098-5589

[BSL99] Basili, V. R.; Shull, F.; Lanubile, F.: Building Knowledge through Families of Experiments. In: *IEEE Trans. Softw. Eng.* 25 (1999), № 4, pp. 456–473, ISSN 0098-5589, doi:10.1109/32.799939

[FPG94] Fenton, N.; Pfleeger, S. L.; Glass, R. L.: Science and Substance: A Challenge to Software Engineers. In:

IEEE Softw. 11 (1994), № 4, pp. 86–95, ISSN 0740-7459, doi:10.1109/52.300094

[Gla94] Glass, R. L.: The Software-Research Crisis. In: *IEEE Softw.* 11 (1994), № 6, pp. 42–47, ISSN 0740-7459, doi: 10.1109/52.329400

[JM01] Juristo, N.; Moreno, A. M.: *Basics of Software Engineering Experimentation.* Kluwer Academic Publishers, 2001

[JM03] —— *Lecture Notes on Empirical Software Engineering*, vol. 12 of *Series on Software Engineering and Knowledge Engineering.* World Scientific, 2003

[KPP+02] Kitchenham, B. A.; Pfleeger, S. L.; Pickard, L. M.; Jones, P. W.; Hoaglin, D. C.; Emam, K. E.; Rosenberg, J.: Preliminary guidelines for empirical research in software engineering. In: *IEEE Trans. Softw. Eng.* 28 (2002), № 8, pp. 721–734, ISSN 0098-5589, doi: 10.1109/TSE.2002.1027796

[LHKS92] Lewis, J. A.; Henry, S. M.; Kafura, D. G.; Schulman, R. S.: On the relationship between the object-oriented paradigm and software reuse: An empirical investigation. In: *Journal of Object-Oriented Programming* 5 (1992), pp. 35–41

[LHPT94] Lukowicz, P.; Heinz, E. A.; Prechelt, L.; Tichy, W. F.: *Experimental Evaluation in Computer Science: A Quantitative Study.* tech. rep., Fakultät für Informatik, Universität Karlsruhe, August 1994

[MWB99] Murphy, G. C.; Walker, R. J.; Baniassad, E. L. A.: Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming. In: *IEEE Trans. Softw. Eng.* 25 (1999), № 4, pp. 438–455, ISSN 0098-5589, doi:10.1109/32.799936

[Pfl99] Pfleeger, S. L.: Albert Einstein and Empirical Software Engineering. In: *Computer* 32 (1999), № 10, pp. 32–38, ISSN 0018-9162, doi:10.1109/2.796106

[PK04] Port, D.; Klappholz, D.: Empirical Research in the Software Engineering Classroom. In: *CSEET '04: Proceedings of the 17th Conference on Software Engineering Education and Training (CSEET'04)*, Washington, DC, USA: IEEE Computer Society, 2004, ISBN 0-7695-2099-5, pp. 132–137

[Pop59] Popper, K. R.: *The Logic of Scientific Discovery.* London: Hutchinson, 1959

[Pot93] Potts, C.: Software-Engineering Research Revisited. In: *IEEE Softw.* 10 (1993), № 5, pp. 19–28, ISSN 0740-7459, doi:10.1109/52.232392

[PPV00] Perry, D. E.; Porter, A. A.; Votta, L. G.: Empirical studies of software engineering: a roadmap. In: *ICSE - Future of SE Track*, 2000, pp. 345–355, doi:10.1145/336512.336586

[Pre96] Pregibon, D.: *Statistical Software Engineering.* National Academy of Sciences: Washington D.C., 1996

[Pre01] Prechelt, L.: *Kontrollierte Experimente in der Softwaretechnik.* Springer Verlag, 2001

[Spi88] Spivey, J. M.: *Understanding Z: a specification language and its formal semantics.* New York, NY, USA: Cambridge University Press, 1988, ISBN 0-521-33429-2

[Tic98] Tichy, W. F.: Should Computer Scientists Experiment More? In: *IEEE Computer* 31 (1998), № 5, pp. 32–40

[TLPH95] Tichy, W. F.; Lukowicz, P.; Prechelt, L.; Heinz, E. A.: Experimental evaluation in computer science: a quantitative study. In: *J. Syst. Softw.* 28 (1995), № 1, pp. 9–18, ISSN 0164-1212, doi:10.1016/0164-1212(94)00111-Y

[WBN$^+$03] Walker, R. J.; Briand, L. C.; Notkin, D.; Seaman, C. B.; Tichy, W. F.: Panel: empirical validation: what, why, when, and how. In: *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, Washington, DC, USA: IEEE Computer Society, 2003, ISBN 0-7695-1877-X, pp. 721–722

[WRH+00] Wohling, C.; Runeson, P.; Höst, M.; Ohlsson, M.; Regnell, B.; Wesslen, A.: *Experimentation in Software Engineering – An Introduction.* Kluwer Academic Publishers, 2000

[ZW97] Zelkowitz, M. V.; Wallace, D. R.: Experimental Validation in Software Engineering. In: *Information and Software Technology* 39 (1997), pp. 735–743

[ZW98] —— Experimental Models for Validating Technology. In: *Computer* 31 (1998), № 5, pp. 23–31, ISSN 0018-9162, doi:10.1109/2.675630

3 Example of Empirical Research: n-Version Programming

Matthias Rohr <matthias.rohr@informatik.uni-oldenburg.de>

Abstract

Software engineering aims to improve the process of software development and its resulting product. The human factor introduces complex behavior into the process. This is difficult to describe with analytical models. Therefore, empirical research methods often are a more effective way to validate research results.
In this paper empirical research methods are studied. We focus the analysis on examples from N-version programming research. N-version programming is a software development strategy to increase reliability through redundancy. This area is chosen because its nature suggests and supports the empirical examination very well, with still providing typical complexity and a large number of empirical studies available.

3.1 Introduction

One goal of software engineering research is to improve the process of software development and its product. Many research results in this area are published as suggestions on how to develop good software efficiently. Some kind of evidence or a conclusion from accepted knowledge can convince other researchers or practitioners to apply new methods. Without it, new ideas might not be applied, even if they are better than alternatives. Many researchers complained about the low rate of empirical verification of research results in software engineering.

Prechelt observes two strategies to advance in software engineering [Pre01]:

- The engineering strategy develops methods or tools in order to reach selected goals. Support is given by the market or other researchers.

- The scientific strategy uses iterations of model creation and validation to built up lasting knowledge. It can be used to give support to concepts developed by the engineering approach.

Empirism is a scientific strategy. It derives theories from observed facts [Ebe99, p. 20]. In combination with generalization, this allows to support general statements from a limited number of observations, which can result from experiments. Empirical reseach methods are often the method of choice to support findings, because other methods, such as conclusion from proved knowledge, cannot be applied due to the complexity of the process of creating software from requirements. The complexity emerges "from technical issues, from the awkward intersection of machine and human capabilities, and from the central role of human behavior in software development"[Sea99]. Empirical research has the disadvantages, that different theories can be concluded from the same observations [Ebe99, p. 34] and that results based on single scientific observations are not able to make a final prove [Pre01]. A counter example can disprove a well established theory based on generalized observations anytime.

Controlled experiments are discussed in chapter 2 as one of its most important methods. This chapter studies the application of empirical research. We use N-version programming as example research domain, because as we will see later, it supports empirical reseach methods very well and many studies are available as research objects.

The research method of this paper is also empirical research. This is not a meta study on N-version programming as some might guess, because that one would evaluate the studies of N-version programming in order to study N-version programming. We investigate software engineering research methods based on a selection of few connected empirical studies. This chapter is a qualitative survey on the research method application in N-version programming. Our goal is to discuss consequences of the application of empirical research methods.

N-version programming (NVP) is a strategy to increase the reliability of software. Faults are tolerated through the parallel execution of redundant software modules. The key idea is to use functional redundancy in the software (the N-version software, NVS) and its development process (the N-version programming). By functional redundancy we mean here the additional integration of software functionality, which is already included in the software system. The hope is that the effects of faults of single parts can be masked by correct results of others.

The remaining parts of the paper are structured as follows: Section 2 introduces the basics of N-version programming to support the reader with a general knowledge that is required to understand the goals and settings in the empirical NVP studies. Section 3 gives an overview of some selected empirical studies. The research methods applied are analyzed and discussed in section 4, and the conclusion follows in section 5.

3.2 N-version programming

This sections provides an overview of N-version programming. It starts with a definition and related terms. Section 3.2.1 presents the underlying concept of reliability improvement through functional redundancy and its assumption of failure independence in hardware engineering. Section 3.2.2 explains how NVP applies the concept to software. A closer description of the NVP development process and a discussion about the effectivity in improving reliability follow in 3.2.3 and 3.2.4.

N-version programming is the *process* of developing *multi-version* software (also known as N-version software). $N \geq 2$ functionally equivalent software entities, called *versions*, are developed by independent single programmers or teams of programmers [Avi95]. It is the hope that the parallel execution of different versions will greatly reduce the probability of similar failures [AC77]. This implies that the N individuals or groups do not communicate with each other during the development to avoid that the versions contain similar faults.

The *initial specification* defines the functional and non-functional behavior of the versions. It is important that more effort than usual is spend to ensure its correctness, completeness, and unambiguity before the N-version development process starts [CA78]. Additionally, the initial specification should not limit the possible diversity between the

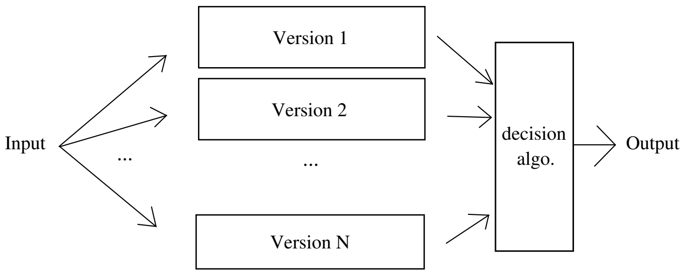

Figure 3.1: N-version architecture

versions by unnecessary restrictions or suggestions [AC77]. During runtime, the N versions are executed in parallel on the same input (see figure 3.1). The output of the versions is processed by some form of decision algorithm (comparison or voting) [LPS01]. Figure 3.1 shows the architecture of N-version software.

The term '*programming*' should not give the impression that only the implementation process is done separately. NVP usually requires independent design, coding, and testing to achieve sufficient levels of diversity between the versions.

We use the following definitions for correct service and (service) failure:

> "**Correct service** is delivered when the service implements the system function. A **service failure**, often abbreviated here to **failure**, is an event that occurs when the delivered service deviates from correct service. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function."[ALRL04]

A fault is the cause of a failure.

$N = 2$ is a special case ("matching" comparison [CA78]), because the decision algorithm has problems to decide, which result might be correct if both results are different. In this case of disagreement, at least it is sure that one or both outputs are not correct. Therefore, 2-version software supports fault detection. It can be used to execute a safe shutdown (e.g., turning traffic lights red), to start a recovery process, or simply to show an error message [KL86, Avi95].

Fault *tolerance* can be implemented by a voter in $N \geq 3$-version software (“voting” comparison [CA78]). Voting policies are a separate research area, and we will only discuss the common majority voting. Usually, effects of faults occur only in a minority of the versions at the same time. Therefore, faults are toleranted as long as the majority agrees on the correct result. If the majority of versions fails with different values, then the majority voter is not able to select an output value, but at least the fault is detected. In the worst case, the majority fails with the same result. In that case, the voter will deliver the wrong result, without detection of the failure.

3.2.1 Functional redundancy in hardware

The concept of functional redundancy is known to be a general concept to increase reliability in other engineering disciplines such as hardware development, or even human resource management. The first documented application was by Babbage in the 1830s [Avi95]. He suggested to use several people to solve the same, complicated task *independently* to ensure its correct fulfillment. Not every task can be solved independently at the same time. Babbage instructed the same mathematical problem to different people. Most mathematical tasks have only one single solution. If different results occur, it is obvious that not both results can be correct. It was experienced, that the chance that the majority fails or wrong results are not detected, is lower than the single ones. In this example, all additional people provide a redundant service - every one of them alone is usually able to complete the tasks successfully. In N-version programming the software development process is the task.

One should not mistake the independent solving of tasks for the principle of cooperation. Cooperation means that more than one person (or other entity) solve the same task *together*. It can be required if a single entity is not able to provide the service alone. We understand functional redundancy as a special form of cooperation, because functional redundancy only shows benefits in the case of failure of single units. In contrast, cooperation is also beneficial for extra-functional properties of the service. The human body is a good example for the difference between this terms, because it contains many elements more than once such as the hands, eyes, arms, and legs. Every eye provides the (functional redundant) critical service of vision (e.g., looking for food or dangers) alone, but only in cooperation they allow to percept the environment three-dimensionally. It seems not easy to discover pure functional redundancy in nature (the reader may try).

The concept of using functional redundancy to increase reliability is known for a long time to hardware engineering disciplines [CA78]. Services that could be provided by a single hardware unit, are distributed to functionally redundant, identical hardware units. For instance, safety-critical (hardware-) systems of airplanes are replicated several times. The pilot has still access to critical information (such as the artificial horizon) if one of the replicated sensor devices breaks down because of wearout. Another real world example are nuclear power plants that have at least two separate cooling systems. Functionally redundant hardware systems have been proven in practice to be dependable if simple guidelines are followed. The most important one is maximum physical separation of the redundant parts, so that a local physical effect (e.g., fire) cannot affect multiple components at the same time.

The highest[1] increase of reliability can be achieved, if the failure different modules fail statistically independent [KL86]:

> "Two events, A and B, are independent if the conditional probability of A occuring given that B has occurred is the same as the probability of A occurring, and vice versa. That is $pr(A|B) = pr(A)$ and $pr(B|A) = pr(B)$. Intuitively, A and B are independent if knowledge of the occurrence of A in no way influences the occurrence of B, and vice versa."

The assumption of independence has the advantage of simplifying the calculation of the failure probability to the product of the single module probabilities. Even adding modules with a relatively high probability of failure improves the reliability of the combined modules.

3.2.2 Software reliability

"Software reliability is the probability of failure-free operation of a computer program for a specified time in a specified environment" [MIO87].

The N-version programming approach tries to apply the concept of improving reliability with parallel functional redundancy from hardware to software.

[1] Theoretically it would be even better to aim for negative correlation of the failure process (independency means zero-correlation)[LPS01, p. 186].

Under the assumption of failure independence, the chance is very low that failures will occur for the same input. The assumption implies a second aspect for software: if two versions fail, there is no special probability that the wrong output values are identical (for non-boolean values). If the output value has a large range, then it is unlikely that two version that fail, will fail with the same value. In mathematical terms, it is expectable that the reliability increases exponentially with additional parallel versions under the assumption of failure behavior (occurrence and value) independence.

Unfortunately, the concept cannot be applied to software one-to-one. An important difference is that software usually fails because of design faults and in contrast, hardware usually fails because of detoriation or other physical effects. The reliability of hardware systems can be improved by the usage of identical parallel hardware channels. In contrast, identical parallel software versions (copies) will usually fail together for the same input - only hardware related faults such as bit-shifts in the memory can be tolerated by parallel copies.

Software failure independence of different versions cannot be understood as an realistic assumption. Moreover, it is important to be clear about that it is just the goal to reach as much as possible diversity in the failure behavior of single versions [Avi95, p.25].

The main strategy of the N-version approach to reach failure diversity is to use diverse versions that are the product of diverse software developments (see figure 3.2). It would be far beyond the scope of this paper to analyze the possible reachable level of diversity of programmers, their backgrounds, used tools and programming languages, but experience and empirical studies (e.g., [KL86]) let believe that even in 'independent' software developments, programmers have a tendency to create similar faults. Only some simple kinds of faults such as typos are relatively random, but anyway these are not that much of a problem, because they are in general simpler to find than those that are related to a misunderstanding of a complex abstract problem.

3.2.3 Development process

As noted in the last section, the reliability improvement strongly depends on the failure process diversity between the versions. The suggested way is to introduce diversity via separate diverse development processes for each version. Guidelines have been introduced to prevent some common causes of low diversity.

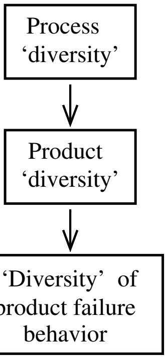

Figure 3.2: "Different types of diversity at different stages of the software design and development process" (from [LPS01, p. 186])

N-version programming requires special preparation during the early design stages. The first step in the application is to write a formal specification (the 'initial specification'). This requires more effort than in other programming projects to ensure that (1) little communication is required later, (2) the different versions are functionally equivalent, and (3) the specification must not be too restrictive by making suggestions that reduce the diversity [Avi95].

Every N-version development process contains three main tasks: (1) the development of an initial specification for every NVS unit (the set of versions with the same functionality), (2) the development of the different versions by separate developers, (3) and the combination and development of an environment for the NVS units, which organizes the parallel execution and evaluation of the outputs of the versions by a decision algorithm [Avi95].

The developer teams of the different versions have to follow a strict communication and documentation protocols and have to be isolated from each other to avoid 'fault leaks', which could introduce the same faults into different versions. For instance, communication between the developer groups is not allowed at all. A coordination team supervises the compliance with regulations and communicates to the developer groups only if it is not avoidable (e.g., changes in the initial specification) [Avi95, p. 30].

Avižienis [Avi95] lists the following dimensions to introduce diversity:

- training, experience and location of software developers
- algorithms, data structures (restricted or open)
- programming languages
- software development process
- development tools
- testing methods

There are two different ways to maximize diversity: (a) random diversity, and (b) forced diversity [LPS01, p. 185]. The versions of isolated developers are diverse, because software development depends on individual skills, knowledge, and background. The amount of diversity is random, because people vary in the amount of differences to each other.

A problem can be that people tend to decide too similarly. Students from the same university might make very similar decisions based on their identical educational background. Experienced programmers often chose methods that are regarded as best-practice. The strategy of 'forced diversity' intents to ensure some diversity by giving different initial specifications to the developer groups. Avižienis et al [ALS88] forced diversity by assigning different programming languages to each development team. Programming languages vary in concepts and styles. For example, C allows the use of pointers, which are effective but a possible source of memory access failures. Pointer failures could be masked by a version written in a pointer-less language, such as Java, which might itself have the risk of making simple binary operations such as bit-shifting complicated, and therefore another possible source for failures. The research of Littlewood et al [LPS01, p. 197] suggests that it is better to force some diversity.

3.2.4 Effectivity of NVP

.

NVP has lost attractiveness after the empirical work of Knight and Leveson [KL86] showed that failures are not independent in different versions, and therefore the reliability increase is much lower than expected before. As consequence, it seems not to be realistic and dangerous to base reliability prediction on the assumption of independence. However, as Littlewood et al [LPS01] point out, there is no reason to believe that design diversity (or NVP) is ineffective to reach high levels of reliability. However, the reliability of the resulting system has to be measured, because it is insecure to predict the reliability increase without knowledge of the correlation of the failure process of the versions.

NVP has additional benefits besides the reliability improvement through fault tolerance:

Cross-testing: N-version programming allows to do a cross-testing (also called back-to-back testing) between the different versions. This helps to detect a large number of (especially simple) faults in a short time. Anyhow, cross-testing can be a risk if the developers are not aware that equal results are not necessarily correct. It is unlikely that the total testing effort is reduced, because especially complex faults, such as common mode failures, are not detected [KL86].

High quality specification: NVP requires high quality from the initial specification. The additional effort can discover mistakes early in the development process, this usually reduces the risks and the costs of the software development.

Intense specification checking: More people are involved in the development and the required separation between system developers and version developers leads to a double check of the specification. This can also improve the software quality.

Field experience: NVP was applied in several industrial projects such as the Airbus A320/330/340, Boeing 737-300 and railway signaling and control systems [Avi95].

The major disadvantages of NVP are:

Development costs: Multi-version software is costly because parts of the software are developed multiple times (design, implementation, testing) and additional organizational work ([LPS01, p. 188]) is required to combine the different versions, and manage the parallel developments.

Unknown effectiveness: It is difficult to conduct a cost-benefit-analysis, because it is not known how effective NVP is. On of the later studies by Eckhardt et al [ECK$^+$91] observed a factor two to five, but the result is not generalizable.

No guarantee for reliability increase: There is no guarantee for a reliability increase of the multi-version software.

Higher complexity: The introduction of diversity to improve reliability always requires some additional logic such as decision algorithms (i.e. voter) and the parallel execution environment. Additional parts can introduce new reliability risks.

Reduced maintainability and readability: The architecture of NVS is more complex [LPS01, p. 188]. This can reduce the maintainability and reusability of the system [Avi95, p. 41].

Higher resource requirements: The resource requirements of multi-version software are up to (but usually less than) N-times higher than of a single version software.

Reduced performance: It can be required to wait for the slowest parallel version [Avi95, p. 40].

In conclusion, the efficiency to improve reliability of NVP is unsure, while newer results are encouraging [Hat97, TXM01]. Efficiency is determind by effectiveness and the costs. The costs of failure are increasing with further integration of software in the environment. This tendency could make NVP a more interesting alternative in the future.

3.3 Selected empirical studies

In this section, a selection of empirical studies on NVP is summarized. The scope of this paper does not allow to include all investigations. The selection represents the major empirical evaluations of NVP-research from the beginnings in the late 70s to the early 90s. This series of early studies allows to follow the maturing of both research methods and NVP, and to learn from weaknesses, which support todays quality requirements of empirical work from a practical point of view.

It is not our intention to assess the empirical studies. The guidelines of empirical software engineering were not even invented at the time of publication. An assessment would require a very detailed study of the research methods and NVP, and would probably lead to hypothetical discussions. Moreover, we want to focus on highlighting parts of the evolution of the studies and connect it to the research domain. Our goal is to present the application of empirical research at different stages of research, discuss some of the research methods applied, and to derive some suggestions.

Several criteria were suggested to analyze empirical research methods in software engineering [Bas96, Bro97, Tic00, DBO$^+$03, KPP$^+$02, Pre01, HWT05]. Most criteria address controlled experiments (see chapter 2) or are related to the Goal-Question-Metric paradigm.

3.3.1 Early studies (77-78,80-83)

The first set of empirical studies [AC77, CA78] was published together with the introduction of N-version programming. We have to note that [AC77] was not available to us during the writing of this paper. There is reason to believe in large similarities between [AC77] and [CA78] from citations, the facts that it is written by the same authors, and the close time frame between the publication dates.

In [CA78] two experiments were set up to test the feasibility of NVP. The method of empirical research is chosen because of the absence of formal theories. Besides the test of feasibility, a large scope of additional qualitative and quantitative objectives is derived from several general research questions to objectives.

Graduate seminars of students had to develop two different applications independently. Seven of 16 programs are analyzed closer. It should be mentioned that three of these were written by the designers of the experiment.

The major conclusion is that NVP is simple enough to be applied. Some failures observed are related to missing logic elements in the programming, and are contained in different versions. Correct results might have been outvoted. The assessment of effectiveness discovered to be more complicated than expected. Therefore, the authors decided to make no conclusion on it, although the results of the study were classified as encouraging. Chen and Avižienis [CA78] note that their results only have small general meaning because of the newness of N-version programming.

A following set of experiments was introducted by related authors [Kel82, KA83] and more independent investigations by [DL80, RMB$^+$81]. Unfortunately the papers could not be analyzed in this work because they were neither available online, nor in the library. The study of later survey articles and empirical studies suggests that their character is supportive without significant discoveries. Knight and Leveson [KL86] add that these studies more or less assume failure independence and do not assess this critical aspect.

3.3.2 The Knight Leveson Experiment (86)

The empirical investigation of Knight and Leveson [KL86] marks a breakpoint in NVP-research. Their discovery was that the general expectations about the reliability growth achieved through NVP is much lower than implicitly expected.

The authors explain the derivation of a null hypothesis up from the intuition that people *tend* to create faults that can lead to similar failures. Such failures were observed in earlier experiments (i.e. [CA78]), but the extend of this critical aspect had never been directly assessed before. As in earlier studies, students are the subjects in an in vitro (lab) study. The participating students have different educational backgrounds (mixture of graduates and undergraduates from two universities and different faculties). Some mathematical skills are required to solve the developing task. The application scenario was reused from former software engineering experiments and was published as well. In total, 20 pairs of programmers required in average about 50 hours to study the specification, and design, study, and debug the software.

After the development and the debugging process (with small given test data sets and outputs), the single versions had to pass a given acceptance test. The versions are evaluated in a long term (several month) test against a so-called gold version, which was carefully developed and tested by the authors. The communication with supervisors was limited to email.

The versions produced several correlated failures. Especially 'high-level' failures related to mathematics were observed for more than one version. Some of them are related to complex semantics, others to flaws in the specification. Six of the 27 versions did not show any failure at all during the long term testing against the gold version.

The formal definition of the null hypothesis allows a statistical evaluation of the experimental results and therefore a systematic conclusion.

This reveals that the assumption of independent failure behavior is wrong with a high confidence level. Some problems, which seem not to weaken the conclusions, are reported. Knight and Leveson [KL86] conclude that the reliability of an NVP system might be lower than expected under the assumption of failure process independence. This does not automatically mean that NVP is ineffective to improve reliability.

The authors explain that the complexity of the problem is low compared to real world application sizes, so that it is not sure if the results can be generalized. Additionally, they note that the observed results are application dependent, and that no general conclusion can be drawn just on basis of this single evaluation.

The study was criticized by researchers who had presented optimistic NVP results before (a discussion of the criticism can be found in [KL90]).

A detailed analysis of the causes of similar failures can be found in [BKL90]. The two major findings are that the common mode failures observed were not necessarily caused by similar faults, and that the faults were not in the areas that have been expected by the programmers. Fundamental flaws in the algorithms were a major cause for the similar failures. The authors expect that it is not possible to significantly reduce failure correlation by simple methodical changes (Note: This is in contrast to [ALS88]).

3.3.3 The 'Six Language' Experiment (86-88)

A six-language study of design diversity is given by Avižienis et al [ALS88] in cooperation with an industrial partner. Reasons for the investigation are the research interests: (1) development of design guidelines to remove the causes of related design faults; (2) search for potential causes of common mode failures; (3) assessment of the potential for diversity of the specification; (4) development of methods to assess the diversity between a set of versions. "It was hypothesized that different programming languages will lead people to think differently about the application problem and the program design, which could lead to significant diversity of programming efforts"[ALS88]. The authors claim to design an 'industrial' study. However, it is controlled and organized by university researchers. Six teams of two "programmers" had to develop software defined by 64 pages initial specification. At least some of them were PhD students at that time. Their educational background is diverse with programming experience between two and six years. Avižienis et al [ALS88] describe the experiment design clearly and required the subjects to report faults after the first successful compilation to allow a closer analysis of faults.

The six versions are measured by several metrics (e.g., lines of code, number of statements, number of procedures). This data is only presented without making conclusions based on them. About two of one hundred identified faults (discovered during three different testing phases) are identical. The order how information is presented in the specification has an influence to the order in development, and therefore reduces diversity. The use of different programming languages required additionally efforts for the decision algorithm.

The authors say that the two identical faults are not very important because they are "rare" and "could have been avoided"[ALS88]. The existing design guideline are judged to be sufficient complete. The initial specification is discovered to be overspecified, which limits the diversity potential. Furthermore, it is concluded that different programming languages are "very effective" for team separation. The conclusions are descriptive and it is not explained how they can be derived systematically from the results. Avižienis et al [ALS88] argue that the fundamental problem of identical failures is avoidable through rigorous application of the NVP paradigm, in contrast to [KL86].

3.3.4 The 'Second Generation' Study (85-88,91)

[KEV$^+$88] and [ECK$^+$91] discuss the same experiment differently. The first one is an 'early result'-presentation.

Kelly et al [KEV$^+$88] aim to take NVP research a step further by conducting a "large-scale" empirical study. They want to get more comprehensive results to evaluate the effectiveness of NVP and to "define and develop a multi-version programming methodology" by using a more realistic application scenario. The primary objective is to evaluate the reliability improvement, with a closer focus on the identification and definition of key factors and analysis of the resulting product. Many secondary general goals are also mentioned (e.g., analysis of failure causes, reliability modeling, role of recovery).

The study design increases the potential for diversity and confidence by a high level of physical separation of the development teams (eight involved research labs), the employment of programmers (instead of students), and the larger the application scale (compared to former experiments). The average size of the program was 2500 lines of code.

After an acceptance test 17 versions were tested in detail with only low sensor noise levels. 14 of these 17 versions showed no failures at all compared to a gold program [KEV^{+}88]. Back-to-back testing discovered more failures. The initial specification was published to support repeatability.

A high number of flaws was discovered in the initial specification during the study. The reliability of the developed N-version software (using consensus decision by voting) is stated higher than the reliability of the gold program.

Kelly et al [KEV^{+}88] conclude that the benefits of the redundant development process "clearly extend beyond than those anticipated for the final MVS system alone" (MVS = multi-version software) from the high number of early discoved faults in the specificaltion and the effectiveness of back-to-back testing to find a large number of faults. Fault analysis "proved" that the specification is "the primary source" of common mode failures.

Eckhardt et al [ECK^{+}91] published a closer description of the same experiment and a more sophisticated analysis of the results is given. The large differences to the presentation of early results in [KEV^{+}88] are not explained.

The focus is set to assess the effectiveness of NVP for critical systems. It was decided to redefine failures to abstract from voting. No distinction is made between identical failures and dissimilar failures. In other words, some possible benefits of voting strategies are ignored intentionally (no assessment of decision algorithm). The reliability growth is measured under the baseline assumption of independent programming teams. Diversity in form of common mode failures is assumed as key variable for this evaluation. The authors are aware that the number of common mode failures is not a very good metric for diversity, but it was not possible to find better candidates for metrics.

The choice of 40 programmers (in development, additional 20 during validation) from 4 universities is explained.Flaws in the specification lead to insufficient reliability and common faults during the development. This problem is recognized early enough and the authors decide to remove these failures. This can be done because the specification quality is not the key variable of the analysis and related faults should not be part of the conclusion.

The operational evaluation of the versions is conducted by an external organization. Eckhardt et al [ECK^{+}91] give reasons for choosing a simulation-based evaluation instead of using a goal program. The whole evaluation procedure and the results are discussed and presented in detail.

Some coincident failures are connected to a special point in the specification. However, the specification is sufficiently well. This is supported by the fact that 14 groups did not make the related mistakes. Therefore, Eckhardt at. al conclude that some aspects of the application problem are the cause of similar failures (and not the specification as its written representation). A second class of coincident failures observed is caused by dissimilar ('logical unrelated') faults. This means that diversity of the versions (and especially the faults) does not guarantee diversity in the failure behavior.

It is possible to create poor multiversion systems in terms of reliability increasement based on the versions developed. The failure probability under the experiment conditions is only two to five times smaller than the average failure probability [ECK$^+$91].

The results of Knight and Leveson [KL86] are supported with the conclusion that the assumption of independence is clearly not justified for predicting reliability increase [ECK$^+$91]. Furthermore, the results indicate that independent development (as the primary NVP paradigm) is no guarantee for sufficient reliability improvement, because some coincident failures are caused by 'input domain related' (problem related) faults. It is stated clearly that the study cannot be used to decide if NVP is more efficient than the development of one good version under conditions with equal resources.

3.4 Analysis and discussion of observations in the selected empirical studies

From the beginning of NVP-research, empirical methods were applied in order to support intuitions or explanations. Avižienis and Chen [CA78] used it because of the lack of alternatives. Many computer science domains missed the chance to support findings with empirical evidence [Tic98, Pre01]. This is not the case in the NVP domain - many later publication about NVP followed the example. Empirical investigations such as [KL86, ECK$^+$91] verified important hypotheses, theories and modes, and discovered results of importance beyond the scope of NVP.

The 'early studies' were basically testing the feasibility. Not every scientist would classify these papers as empirical research, because most of the general guidelines of empirical research in software engineering were not applied (because they were published later). Abstract goals and missing systematic derivation of questions or metrics reduce confidence and conclusion strength. However, feasibility tests can be very helpful to identify relevant system variables or relationships from scientific observations. Chen and Avižienis [CA78] explain their choice of research style with the lack of theory. We agree in this - a more informal experiment might be suitable and effective at the first period of research in a new domain. Good empirical studies are expensive and risky if theory and former experience are missing.

The later studies such as [ECK$^+$91, KL86] defined goals, purposed and metrics, described the experiments carefully, and try to achieve as much as possible control over the relevant system variables. This seems to be a evolution from 'demonstrations' towards controlled experiments. Controlled experiments are attractive because relatively strong conclusions can be drawn based on observation. This is a possible explanation for the fact that high levels of control were not possible at the beginning of NVP research, because system variables were not known and theory was missing to develop metrics.

The chapter about controlled experiments (chapter 2) recommends the rigorous application of research methods. The studies analyzed give examples to support these claims:

- Chen and Avižienis [CA78] wanted to evaluate the efficiency of NVP, but did define neither metrics nor objectives for the assessment of effectiveness of NVP in detail. In the end of the paper they were not able to conclude on the effectiveness. Besides, so far it has not been possible to draw strong conclusions on the effectiveness of NVP compared to the development of single versions (but many of the studies mentioned it as one of the goals).

- Knight and Leveson [KL86] derived a null hypothesis and metrics from a single research question. The experiment design is described in detail and sufficient repeatability is ensured. This allowed them to draw a strong conclusion with high confidence derived by statistical methods. An additional benefit appeared later as their surprising results had to withstand criticism. Their empirical investigation became a "famous"[Tic98] example for the benefits of empirical research.

The GQM-paradigm allows the study of more than one goal [Bas96, BR88] in a single study. Studies in software engineering are expensive. This can force researchers to address different goals in the same *empirical investigation.* This does not mean that it is required to address multiple goals in a single *paper.* Often the scope of a paper is limited to a few pages and a precise description of an empirical study with many goals is not possible. We believe to focus on one goal or one research question for each paper supports readability, the line of argumentation, repeatability and precision. One possible explanation to write about multiple goals at the same time might be that the writer believes the 'larger' contribution makes it easier to publish. This should not be the case. Tichy [Tic98] suggests to reviewers that not the amount of contribution is a primary criteria for the assessment - also empirical proves for 'obvious' theories are important. We think that [KL86, ECK$^+$91, BKL90] demonstrate that a limitation to single goals or a small number supports the quality without reducing the attractiveness.

In contrast, [CA78, KEV$^+$88, ALS88] use a broad scope of goals in a single paper. We regard this did not benefit to quality aspects such as clearness, repeatability, and strength of conclusion.

Many different goals or research questions can be a serious risk to a single empirical *study.* For instance [ALS88] aims to analyse the effects of some known variables (specification, design guidelines), to develop new metrics (diversity), to identify more relevant variables, and to assess the effects of an additional concept (different programming languages). These objectives are not independent if the metrics are not chosen very carefully (if possible). A controlled experiment would suggest that only a single variable is modified at the same time. If more than one aspect changes during the experiment, it might be not possible to identify and evaluate effect-cause relationships and compare results to other studies. The conclusions of [ALS88] might reflect that problem. For instance, we see no empirical support in the paper for the claims that the application of different languages are "very effective".

It is an important choice how to distribute the limited budget resources for a study in an experiment. As mentioned above, Avižienis et al [ALS88] address many different objectives. Only one team of programmers was used for each programming language. This seems far too few to make statements independently from the skills of the single teams. Avižienis et al [ALS88] are not doing this mistake (directly), because they just assess the feasibility of the application of different languages in multi-version software. They cannot make conclusions about the influence of the application of different programming languages to the reliability.

3.5 Conclusions

In this seminar paper we addressed the application of empirical research in N-version programming. A general introduction to N-version programming was used to analyse the influence of empirical research in the context of the research domain.

We draw two main conclusions:

1. The usage of empirical research was able to advance NVP research. The studies convinced practitioner of the industrial application of NVP in safety critical systems. Contrary research results could be supported by strong empirical evidence to withstand criticism.

2. The level of conformance to the guidelines of empirical research and GQM are different between the studies. We showed studies of strict research method application that drew strong conclusions. Additionally, we presented studies that had problems in drawing the conclusion, which seems to be related to a lack of strict empirical research methodology. For instance, we discussed that some goals of the studies were missed, or did not allow to conclude general statements because of missing focus, lack of control, or unknown key variables. We want to add that the level of theory available and experience seems to constrain the design possibilities and efficiency of empirical research methods.

Both results are not surprising, but support the general knowledge. It was not our intend to assess the studies, so our analysis gives only an idea of possible relationships between some design decisions of empirical studies and its results. The scale and the nature of this study limits the possible confidence in the two conclusions to a low level. Nevertheless, N-version programming is similar to many other subareas of software engineering research, and we believe that analogous results are likely in other domains of software engineering with similar complexity.

Bibliography

[AC77] Avižienis, A.; Chen, L.: On the implementation of N-version programming for software fault tolerance during program execution. In: *Proceedings of International Conference on Computer Software and Applications (COMPSAC 77)*, IEEE, November 1977, pp. 149–155

[ALRL04] Avizienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. In: *Transactions on Dependable and Secure Computing* 1 (2004), № 1, pp. 11–33, ISSN 1545-5971, doi:10.1109/TDSC.2004.2

[ALS88] Avižienis, A.; Lyu, M.; Schütz, W.: In search of effective diversity: a six-language study of fault-tolerant flight control software. In: *Digest of Papers of the Eighteenth International Symposium on Fault-Tolerant Computing (FTCS)*, IEEE, June 1988, pp. 15–22, doi:10.1109/FTCS.1988.5291

[Avi95] Avižienis, A.: The Methodology of N-Version Programming. In: *Software Fault Tolerance*, New York, NY, USA: John Wiley & Sons, Inc., section 2, 1995, ISBN 0471950688, pp. 23–46

[Bas96] Basili, V. R.: The role of experimentation in software engineering: past, current, and future. In: *ICSE '96: Proceedings of the 18th international conference on Software engineering*, Washington, DC, USA: IEEE, 1996, ISBN 0-8186-7246-3, pp. 442–449

[BKL90] Brilliant, S.; Knight, J.; Leveson, N.: Analysis of faults in an N-version software experiment. In: *Transactions on Software Engineering* 16 (1990), № 2, pp. 238–247, ISSN 0098-5589, doi:10.1109/32.44387

[BR88] Basili, V. R.; Rombach, H. D.: The TAME Project: Towards Improvement-Oriented Software Environments. In: *Transaction on Software Engineering* 14 (1988), № 6, pp. 758–773, doi:10.1109/32.6156

[Bro97] Brooks, A.: Meta Analysis-A Silver Bullet-for Meta-Analysts. In: *Empirical Software Engineering* 2 (1997), № 4, pp. 333–338, doi:10.1023/A:1009793700999

[CA78] Chen, L.; Avižienis, A.: N-Version Programming: A Fault-Tolerance Approach To Reliability Of Software Operation. In: *Digest of Papers of the 8th International Symposium on Fault-Tolerant Computing (FTCS) (reprint in Proc. FTCS-25)*, IEEE, June 1978, pp. 3–9

[DBO$^+$03] Dawson, R.; Bones, P.; Oates, B. J.; Brereton, P.; Azuma, M.; Jackson, M. L.: Empirical Methodologies in Software Engineering. In: *Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice (STEP'03)*, Washington, DC, USA: IEEE, 2003, ISBN 0-7695-2218-1, pp. 52–58

[DL80] Dahll, G.; Lahti, J.: An investigation of methods for production and verification of highly reliable software. In: *Proceedings of the Conference on Safety of Computer Control Systems (SAFECOMP 79)*, Pergamon Press, 1980, pp. 89–94

[Ebe99] Eberhard, K.: *Einführung in die Erkentnis- und Wissenschaftstheorie*. Stuttgard, Germany: Kohlhammer, 1999, ISBN 3-17-015486

[ECK$^+$91] Eckhardt, D.; Caglayan, A.; Knight, J.; Lee, L.; McAllister, D.; Vouk, M.; Kelly, J.: An experimental evaluation of software redundancy as a strategy for improving reliability. In: *Transactions on Software Engineering* 17 (1991), № 7, pp. 692–702, ISSN 0098-5589, doi:10.1109/32.83905

[Hat97] Hatton, L.: N-version design versus one good version. In: *Software* 14 (1997), № 6, pp. 71–76, doi:10.1109/52.636672

[HWT05] Höst, M.; Wohlin, C.; Thelin, T.: Experimental context classification: incentives and experience of subjects. In: *Proceedings of the 27th international conference on Software engineering (ICSE '05)*, New York, NY, USA: ACM Press, 2005, ISBN 1-59593-963-2, pp. 470–478, doi: 10.1145/1062455.1062539

[KA83] Kelly, J.; Avižienis, A.: A Specification-Oriented Multi-Version Software Experiment. In: *Proceedings of the 13th International Symposium on Fault-Tolerant Computing (FTCS 13)*, Pergamon Press, 1983, p. 120

[Kel82] Kelly, J.: *Specification of Fault-Tolerant Multi-Version Software: Experimental Studies of a Design Diversity Approach*. Ph.D. dissertation, University of California, Los Angeles, 1982

[KEV⁺88] Kelly, J.; Eckhardt, J., D.E.; Vouk, M.; McAllister, D.; Caglayan, A.: A large scale second generation experiment in multi-version software: description and early results. In: *Digest of Papers of the Eighteenth International Symposium on Fault-Tolerant Computing (FTCS)*, IEEE, June 1988, pp. 9–14, doi:10.1109/FTCS.1988.5290

[KL86] Knight, J.; Leveson, N.: An Experimental Evaluation of the Assumption of Independence in Multi-version Programming. In: *Transactions on SoftwareEngineering* 12 (1986), № 1, pp. 96–109

[KL90] —— A reply to the criticisms of the Knight & Leveson experiment. In: *SIGSOFT Softw. Eng. Notes* 15 (1990), № 1, pp. 24–35, ISSN 0163-5948, doi:10.1145/382294.382710

[KPP⁺02] Kitchenham, B.; Pfleeger, S.; Pickard, L.; Jones, P.; Hoaglin, D.; El Emam, K.; Rosenberg, J.: Preliminary guidelines for empirical research in software engineering. In: *Transactions on Software Engineering* 28 (2002), № 8, pp. 721–734, doi:10.1109/TSE.2002.1027796

[LPS01] Littlewood, B.; Popov, P.; Strigini, L.: Modeling software design diversity: a review. In: *ACM Comput. Surv.* 33 (2001), № 2, pp. 177–208, ISSN 0360-0300, doi:10.1145/384192.384195

[MIO87] Musa, J. D.; Iannino, A.; Okumoto, K.: *Software reliability: measurement, prediction, application*. New York, NY, USA: McGraw-Hill, Inc., 1987, ISBN 0-07-044093-X

[Pre01] Prechelt, L.: *Kontrollierte Experimente in der Softwaretechnik*. Berlin: Springer Verlag, 2001, ISBN 3-540-41257-3

[RMB$^+$81] RAMAMOORTHY, C.; MOK, Y.; BASTANI, E.; CHIN, G.; SUZUKI, K.: Application of a methodology for the development and validation of reliable process control software. In: *Transactions on Software Engineering* 7 (1981), № 6, pp. 537–555

[Sea99] SEAMAN, C.: Qualitative methods in empirical studies of software engineering. In: *Transactions on Software Engineering* 25 (1999), № 4, pp. 557–572

[Tic98] TICHY, W. F.: Should Computer Scientists Experiment More? In: *IEEE Computer* 31 (1998), № 5, pp. 32–40, ISSN 0018-9162

[Tic00] —— Hints for Reviewing Empirical Work in Software Engineering. In: *Empirical Software Engineering* 5 (2000), № 4, pp. 309–312

[TXM01] TOWNEND, P.; XU, J.; MUNRO, M.: Building dependable software for critical applications: multi-version software versus one good version. In: *Proceedings of the Sixth International Workshop on Object-Oriented Real-Time Dependable Systems*, IEEE, January 2001, pp. 103–110, doi: 10.1109/FTCS.1988.5291

4 Integration of Qualitative and Quantitative Methods in Software Engineering Research

Simon Giesecke <giesecke@informatik.uni-oldenburg.de>
Thorsten Giesecke-Kopp <kopp.thorsten@web.de>

Abstract

Empirical studies in Software Engineering do not only deal with software artifacts, but also with the people involved in software development. Therefore, both quantitative and qualitative methods should be integrated in Software Engineering research, in order to benefit from their specific advantages. We illustrate the integration of quantitative and qualitative methods using an example research design following the action research approach.

4.1 Introduction

To a large extent, empirical research in Software Engineering (SE) has been based on a quantitative approach. In the related field of Information Systems, which initially relied on quantitative research as well, the use of qualitative methods has been established for some time now (cf. [KD88]), but still remains in a minority position today [CH04]. The trend towards qualitative methods was based on the broad and controversial discussion in social sciences on the use of quantitative and qualitative methods. In more recent SE research which focuses on the people involved in software development qualitative methods are used as well [Sea99]. In this case, research methods originating from social sciences or psychology, are needed to deal with the complexity of human behaviour.

In more recent social research, opportunities for combining of qualitative and quantitative methods are discussed [May01]. We propose to apply these concepts to SE research. In an exemplary study design, we follow the Action Research approach, which has been successfully applied in IS research [BWH96].

Overview The rest of the paper is structured as follows: First, we discuss the different categories of objects that are considered by Software Engineering research (Section 4.2). Afterwards, we elaborate on the differences of qualitative and quantitative research methods (Section 4.3), and present several research methods, with a focus on research methods for qualitative research (Section 4.4). Then, we discuss how both kinds of methods can be combined in general (Section 4.5). In Section 4.6, we discuss exemplarily how this combination can be applied using the Action Research approach to resolve research questions which arose in a real academic/industrial collaboration project. Section 4.7 concludes the paper.

4.2 Categories of Research Objects in Software Engineering

The choice of an appropriate research method depends on the regarded object and the research question [Fli98, p. 4ff]. Empirical SE research is concerned with different categories of objects:

Software artefacts Software artefacts may be distinguished into those used for automated processing, e.g., transformation into another artefact or execution, and those used for reading and communication. An artefact may serve both purposes, which is the case for source code, for example; any commentary/documentation contained in source code primarily addresses human readers. Byte-code, on the other hand, is (usually) neither written nor read by humans.

Research in this category constitutes the majority of published SE research [GRV04]. For his "Preliminary Software Engineering

Theory" [Zen01], Zendler only considers research that reasons about measurements that are applied to software artefacts which have been produced by applying Software Engineering techniques.

Developers of software Research concerning developers of software (and other stakeholders involved in the development of software systems) is the second large field of SE research. The design and improvement of software development methods belongs to this type of research as well as the fields of Psychology of Programming [PPInd] and Program Comprehension. For example, it might concern team work aspects, the impact of developer incentives of the development of software, or the subjective theories [GS01] developers possess of the software architecture in some software project. Research on Requirements Engineering involves other stakeholders in addition.

Users of software Users of software *in general* or, to be more precise, users of arbitrary software-intensive systems are not in the main focus of SE. They are in the focus of the disciplines of Computer-Human Interaction [ACMnd], which takes a more individual point of view, and Information Systems, which takes a more organisational point of view.

However, research into developers of software can also be research into developers as users of software, in particular concerning the use of CASE tools.

Learning how to use and develop software The aspect of learning to use software is addressed by the fields of Information Systems, at least concerning the use of business information systems, and Computer-Human Interaction.

Since not only the user interface is affected, this aspect is also of relevance to Software Engineering: the underlying conceptualisation of the real world are affected as well; and the development of software, most importantly its decomposition, heavily depends on these conceptualisations.

The latter aspect is the focus of research on SE Education. Neither of these two aspects is in the primary focus of this paper.

The objects falling into these categories are closely related to each other; thus, concrete research cannot always be ascribed a single one of these categories. By distinguishing immediate from ultimate research objects, the categorisation can be supported: The ultimate research object is the object about which the researcher wants to increase knowledge. The immediate research object is the object that he regards for this purpose, and may in this sense considered his data. This distinction also brings about the distinction between data collection, which retrieves data from the ultimate research objects, and data analysis, which operates on the retrieved data (see Section 4.4). It is important to note that the distinction cannot be derived from the character of the objects, but depends on the subjective decision of the researcher on the purpose of his research. For example, research in source code documentation can be viewed as research into software artefacts (software artefacts as research objects), but also as research into people who write and read source code (software artefacts as data). More generally, research which evaluates the development process indirectly by assessing its products, regards the development process as the research object and the products of the development process as data.

Within each of these categories, several *levels of scope* can be distinguished. In [GRV04], levels were identified for software artefacts and users/developers of software: These are three technical levels (computing element, system, abstract concept) resp. six behavioural levels (individual, group/team, project, organisational context, external business context, profession, society). A similar distinction may be made for the learning categories, e.g., individual, learning group, course, cohort, university, national/international curricula and university/educational policies.

4.3 Qualitative and quantitative research approaches

4.3.1 The research process

In the following, we will refer to research processes, i.e., research activities at a level of scope that considerably transcends the daily planning of work. We define the important terms of research project, step and phase.

Definition 4.1 (Research project)**.** A research project is more or less arbitrary in scope with respect to the content of the research but ought to make it administratively handleable. It is often the basic unit for funding, either by public funding agencies (BMBF, DFG in Germany) or by industrial partners. However, research projects are also modelled at other levels of scope: First, at a coarser level, a long-term research project may consist of several funding phases. Second, at a finer level, a (doctoral or Master's) thesis project may also be considered a research project. A research project consists of several research steps.

Definition 4.2 (Research step, research phase)**.** A research step is confined by methodological considerations. One *research step* corresponds to the application of one research method. It is the unit of generation of *research results*, the character of which depends on the chosen method. If a research method is itself decomposed, the resulting elements will be referred to as *research (step) phases*.

In cyclic/iterative research methods, the distinction between step and phase is not possible in a definite way.

4.3.2 Data Scale Levels in Quantitative Research

Data in quantitative research can be classified into several scale levels (levels of measurement), which have an effect on the applicable statistical methods. The levels can be ordered hierarchically from nominal to ratio level, i.e., data on the higher levels have all features of that on the lower levels, and all statistical methods applicable at the lower levels can also be applied to data at a higher level.

We first define the notions of metric and measurement:

Definition 4.3 (metric, measurement)**.** Let U be the set of research objects, and S a set.

Then, a function $m : U \to S$ is a *metric* for U with the scale S.

For $x \in U$, the act of determining $m(x)$ is called a *measurement* of x in S.

The family $(m(x))_{x \in U}$ is called *data* (for U with respect to m).

Depending on the type of S (its structure or characteristics), different types of scales are distinguished (and thereby different types of measurement):

Nominal Scale The data is textual or numeric. No order is defined on S, i.e., no "less than" or "greater than" relation is available. Examples of nominal scales are names, addresses, and telephone numbers.

Ordinal Scale An order relation is defined on S, the data family is said to consist of ordinal values. Arithmetical operations such as addition and multiplication are not defined for this scale.

As a statistical method on the data family, the *median* can be calculated. Examples of ordinal scales are rankings, school grades, revision numbers.

Interval Scale The scale S consists of numeric values which are separated by the same interval, which is attached importance to. An addition operation is defined on S.

As a statistical method, the *arithmetic mean* can be calculated. However, the zero point of the scale is chosen deliberately, and no multiplication or division operations are defined on S.

Examples are temperatures in the Celsius and Fahrenheit scales, and years in the Georgian calender.

Ratio Scale For a ratio scale, in addition, a meaningful zero point is defined. A multiplication (and division) operation are defined on S. All statistical methods can be applied. Examples are distances measured in metres, temperatures measured in Kelvin, age of people measured in years.

While the mathematical definition allows any function to be called a metric, a metric is only useful if it has a natural interpretation. This also restricts the type of scales to be used, since it is possible to map any scale into the set of real numbers, of course, which would always lead to a ratio scale.

4.3.3 Comparison of quantitative and qualitative approaches

The intuitive understanding of the term “qualitative”—in contrast to “quantitative”—may be worded, for example, as in [Sea99]: “Qualitative data are data presented as words and pictures, not numbers.” This explanation is quite straightforward, but does not capture the two concepts completely, as explained in the following.

Quantitative Research

Quantitative research does indeed, but not exclusively, deal with data presented in numbers, which are analysed using statistical methods. As a borderline case, also data at the nominal scale level can be analysed quantitatively. The data are based on a large sample, optimally a *representative sample.*

Quality criteria in quantitative research are *objectivity* (is the measure independent from the researcher?), *reliability* (is the measure reproducible?), and internal and external *validity* (did the researcher measure what they intended to measure, and can the result be applied to every other case?).

The aim of quantitative research is to identify independent and dependent variables, eliminate disturbing variables and thus reduce the complexity of the regarded objects. A research step is based on a hypothesis, which is to be falsified or confirmed. Progress in the quantitative (or hypothetico-deductive) paradigm primarily emerges from testing hypotheses.

Qualitative Research

Qualitative research, as noted above, analyses—among others—textual and graphical data. The sample is restricted to a few instances or even a single instance. Thus, it is possible to deal with and interpret the full complexity of the regarded research objects. Yet, the results are not supposed to be objective (but intersubjective), or to be generalised to all other cases.

Characteristics of qualitative research are *openness* for unexpected results and an *interpretive* approach to data analysis. When people are the object of qualitative research, research is based on their ability to communicate—they are thus seen as 'experts' for their subjective point of view—, and research is preferably performed in their every-day environment rather than in a lab ('naturalism') [Lam93, pp. 17ff].

As a qualitative research design is based on open-ended questions rather than a hypothesis, the goal is to generate new propositions or to systematically improve existing theory.

Discussion

In contrast to the impression conveyed by these two abstract, prototypical descriptions, one cannot always discriminate definitely between concrete qualitative and quantitative research.

In SE, we also find several opportunities to combine both approaches: As noted in Section 4.2, different objects of research in SE require different methods. For most objects, both qualitative and quantitative methods can be applied, for example software engineers may participate either in a qualitative interview, or a quantitative multiple-choice questionnaire survey.

4.3.4 Research Approaches

In this section, several research approaches are presented, ranging from (Quasi-)Experimental Research (Section 4.3.4), over Action Research (Section 4.3.4) and Grounded Theory (Section 4.3.4) to Evaluation Research (Section 4.3.4).

In experimentation, in vivo setups may be distinguished from in vitro setups [Bas96]. The former are conducted in the field under normal conditions, while the latter are conducted in a laboratory under controlled conditions. Formal experiments require an in vitro setup.

The remaining research approaches are all virtually always applied in an in vivo setting.

(Quasi-)Experimental Research

The aim of experimental research is to create a Controlled Environment in a laboratory, in which "complete control of all variables is possible" [Kar02, p. 31]. This aim corresponds to the quantitative research setting, where the isolation of variables is crucial. Experimental research is the basis for most natural sciences and is also performed intensively in the psychological school of behaviourism.

However, when research on complex organisations is performed, experimental research settings encounter severe problems: "In software process research, where the practise is so tightly connected to organisational and human factors, it is difficult to recreate environments an situations in the laboratory that are realistic." [Kar02, p. 33]

Action Research

Action Research was originally conceived by Lewin [Lew48] and uses scientific methods to solve important social or organisational issues together with those who experience these issues unintermediately. Thus, Action Research has two goals: to solve a problem for a client and to contribute to scientific knowledge [CB01, p. 4–6].

Action Research takes place in an iterative process, which is shown in figure 4.1. In a pre-step, *Context and purpose* of the Action Research design are determined. Then, four steps follow iteratively: (1) *Diagnosing*, i.e., identifying a problem and relating it to relevant scientific theories, (2) *Planning Action*, i.e., planning concrete steps to solve the problem based on the diagnosis, (3) *Taking Action*, (4) *Evaluating Action*, which may result in refining the problem definition or planning new action, then, continue at (1) [CB01, p. 16–18].

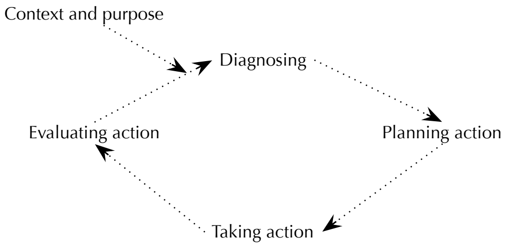

Figure 4.1: Action Research Cycle (from: [CB01, p. 17])

Grounded Theory

The notion of Grounded Theory goes back to the work of Glaser and Strauss [GS75]. The main idea of Grounded Theory is to inductively generate theories, or propositions, that are grounded in data. The generation of Grounded Theory is a circular process, during which the data sample is expanded until *theoretical saturation* is reached.

In detail: At the beginning, a group of cases is chosen according to their *theoretical relevance* [GS75, p. 49] for the development of new knowledge concerning the research question. Then, research is carried out in order to generate new propositions, according to the constant comparison method (see section 4.4.2). This method basically includes taking, reviewing and categorising field notes. Afterwards, a new group of cases is chosen to collect data from, or additional data from an previously used group is collected. This process ends, when no additional data is found, from which the researcher can develop new properties for a category: The researcher "sees similar instances over and over again" [GS75, p. 61]. At this point, *theoretical saturation* is reached, and the research cycle for this category ends.

Evaluation Research

The notion of ‘evaluation research’ covers several different research approaches and methods (both qualitative and quantitative), which serve different goals and have different assumptions. Its origins are the assessment of political programs, as defined by Leeuw [Lee00, p. 57]:

Definition 4.4 (Evaluation Research (Leeuw))**.** Evaluation is [. . .] the application of social science theory and methodology in order to assess both ex-ante and ex-post the implementation, the impact and the side-effects of programs, policies, strategies and other ‘tools of governments’ [. . .] on society, including the explanation of those impacts/side-effects.

Although evaluation research is also employed in Software Engineering research, the notions and taxonomies from social research must be modified slightly to be applicable. An exemplary approach to this adaption can be found in [FF01].

4.4 Research Methods

Research methods must be chosen depending on the research question and research object. We distinguish two types of research phases, which require different methods: Data collection and data analysis. In many cases, there is a clear distinction between these two phases; however, in some settings data collection and analysis overlap and at least some analysis is performed already during data collection (e.g., reconstruction of subjective theories, see Section 4.4.1).

4.4.1 Data Collection Methods

Although we focus on qualitative data collection methods in this chapter, many of the methods presented here have quantitative counterparts. In addition, it is possible to combine qualitative data collection with quantitative data analysis (see Section 4.4.2).

Observation

Observation is used frequently both in qualitative and quantitative research. Observational methods may be classified in several dimensions [Fli98, p. 137], of which we consider the following most important for our issue:

Covert vs. overt observation The individuals being observed may be aware of the observation (overt) or unaware of the observation. In the first case, they may know the research question to varying degrees of detail.

Non-participant vs. participant observation In some settings, the observer becomes an active part of the observed field. In the continuum between non-participation and participation, at least four different roles may be distinguished: The complete participant, the participant-as-observer, the observer-as-participant, the complete observer.

Degree of standardisation In quantitative approaches, observation is usually standardised: For example, in a development team meeting, the observer may count the number of contributions to the discussion made by each team member. In qualitative research the observers stay more flexible, as the setting is less standardised; yet the observation is also focused on one or more research questions.

Interview

Oral interviews are very widespread in both qualitative and quantitative research. Interviews can be classified in (at least) two dimensions: open-endedness of individual question and standardisation of the question set.

Interviews containing open-ended questions enable the interviewee to express his or her ideas very freely, while closed-ended questions have the interviewee make a choice in a pre-determined value space. Common are questions where two or more options are listed and the interviewee has to choose from them ('multiple-choice questions'). Semi-open-ended questions are similar to open-ended questions, but contain the possibility to add own answers in the case the all values in the predetermined value space seem inappropriate to the interviewee.

Standardised interviews consist of a predetermined set of questions, that the interviewer must follow exactly. Non-standardised interviews in contrast consist only of one question at the beginning, framing the subject of the interview. In the following, the interviewer only listens and asks questions spontaneously to ensure he or she has understood the interviewee correctly. Also, a form between these two extremes exists: In semi-standardised interviews, a set of questions is given as a basis for the interview, but the interviewer may alter them during the interview and is free to ask ad-hoc questions in addition [Lam05].

Today, the usual practise is to record interviews on some audio media, and to transcribe them afterwards into a written text, which is the object of the following analysis. This way, the interviewer can fully concentrate on the interviewing and does not have to take notes during the interview. Yet, it is regarded useful to take field notes after the interview, covering the setting, the impressions and ideas during the interview, and any extraordinary events during the interview.

Questionnaire Survey

Questionnaire surveys are the paper-and-pencil (or online) counterpart to (oral) interviews. In contrast to the latter, surveys are always standardised, as the questions are written down. Yet, they may be more or less open-ended, as the examples in table 4.1 illustrate.

Closed-ended question	Your gender: a) Male b) Female
Semi-open-ended question	Which is the most important reason for choosing a programming language? a) availability of tools as open-source software b) personal experience c) theoretical considerations d) other: ....
Open-ended question	What measures during the software engineering process are used to find and deal with errors? ....

Table 4.1: Example survey questions

Group discussion

Instead of interviewing single individuals, in many studies group interviews and group discussions are used. Following this approach, persons can be interviewed more efficiently, saving time and money.

Moreover, group discussions also reveal group dynamics which would not be perceived conducting single interviews. Thus, they “may reveal how opinions are created and above all changed, asserted or suppressed in social exchange” [Fli98, p. 121].

In Software Engineering research, group discussions are suitable primarily when the researchers focuses on team work, stakeholder cooperation, roles of stakeholders in the development, or decision making.

Documents

Sometimes, data which can be used in research is already available and does not need to be generated by the researcher. Such documents of interest can be diaries, memos, pictures and video documents, as well as help-desk logs, purchasing records and official publications [Oat03, p. 61].

Besides these documents, in Software Engineering research there are also some more technical data sources that can be analysed using qualitative or quantitative methods:

- source code comments
- architectural design decisions
- other software artefacts in general

Approaches exploiting such data are known as mining software repositories [MSRnd] and Software Archaeology [HT02].

Reconstruction of Subjective Theories

Mainly in German social research, research on so-called ‘Subjective Theories’ has been proposed. The basis of this approach is the image of ‘man-the-scientist’ (Kelly [Kel63]), which means that all people use subjective theories in everyday life that are similar in structure with objective theories in science [GS01].

These Subjective Theories are reconstructed in a two-phase approach. Firstly, a special case of semi-standardised, open-ended interview is performed. Secondly, a graphical structure consisting of the key notions from the interview transcript is created by both the interviewer and the interviewee. This structure is used to explain the causal relationships between the key notions.

Special about this approach is the validity criterion of ‘dialogue consensus’. This means that not the interviewer alone interprets the interview text and defines the results of the research, but both the interviewer and the interviewee construct the results in a dialogue. Thus, in this approach there is no clear border line between data collection and data analysis.

In the field of Software Engineering, for example it were possible to reconstruct a developer's Subjective Theories on the architecture of a software project.

4.4.2 Data Analysis Methods

Statistical Methods

In order to analyse data quantitatively, a large set of statistical methods is available, especially the measures of central tendency (e.g., modus, median and mean), of variance, and of statistical connection (e.g., correlation, contingency or co-variance). Yet, these statistical methods are outside the scope of this text.

These methods can, for example, be applied to data from a quantitative survey, a standardised observation, as well as software-artefacts (software metrics).

Qualitative Content Analysis

The main idea of qualitative content analysis as proposed by Mayring is to "preserve the advantages of quantitative content analysis as developed within communication science and to transfer and further develop them to qualitative-interpretative steps of analysis" [May00, § 2].

Qualitative content analysis consists of three analysis phases [May03, p. 58]:

Summarising The analysis reduces the available material, while keeping its essential contents. By abstraction, a manageable amount of data is obtained.

Explication The analysis aims at explaining doubtful parts of the text, by collecting additional context information.

Structuring A system of categories is applied to the text material, in order to extract structured information from the text. The categories can either be *deduced* from theoretical positions, or be developed *inductively* from the text material [May00, § 8].

Hermeneutics

Hermeneutics is “the study of the methodological principles of interpretation” [Mis03], i.e., an hermeneutic approach of data analysis implies the interpretation of textual data.

In SE research, different kinds of textual data can be distinguished: Firstly, software artefacts may be available in some textual representation, i.e., source code. Secondly, the text of interviews, protocols of team meetings, documents and other can be interpreted. In the first case, methods specific for SE are needed, while in the latter case, methods from sociology may (in some cases) be applicable.

Objective Hermeneutics Objective Hermeneutics is a qualitative method proposed in sociology by Oevermann. The aim of this method is to reconstruct the objective (i.e., generalisable) structures of sense that lay behind a specific case. Such sense may be among others of psychological, cognitive, or sociological nature. The objective sense may possibly but does not necessarily fit with the subjective sense articulated in the text.

The analysis of these structures starts with an experiment in mind: All imaginable senses of a text segment are constructed and written down by a team of researchers. In a second phase, the common ground of these constructions is filtered out, in order to reconstruct the common structure behind them [May93, p. 88].

Constant Comparison Method

The Constant Comparison Method is used to generate *grounded theory* (see section 4.3.4). In the first phase, labels (codes) are attached to text passages, which are important for a particular question under research. These codes are then used to group text passages into patterns, which are examined for underlying explanations of phenomena. As a result, the researcher constructs a new proposition and writes it down as a field memo [Sea99, p. 566].

Grounded theory is based on a cyclic process, as noted earlier. Thus, the field memos, being the output of one research phase, are also part of the input of the next step, are labelled, grouped and interpreted. This includes that the feasibility of the new proposition is checked in the next research phase [Sea99, p. 566].

Coding

Seaman [Sea99] presents an approach called "Coding", which combines qualitative data collection with quantitative data analysis. She describes a data transformation from qualitative data (like "Tom, Shirley, and Fred were the only participants in the meeting") into quantitative data ("num_participants = 3"). In a further phase, these data can be analysed using statistical methods. As Seaman points out, in many cases coding is not as trivial as in this example; then, more interpretive phases are necessary. These phases resemble the qualitative content analysis presented by Mayring (see Section 4.4.2).

4.5 Combining Qualitative and Quantitative Approaches

In this section, we first present several *setups* of quantitative and qualitative research steps, which are then related to SE research. We only consider combinations involving both types of research; of course, the combination of research steps of the same type (see, e.g., [RT03]) is important as well, but outside the scope of this section.

Preliminary Study When research is performed on a subject for which no hypothesis can be deduced from the existing theory, a preliminary qualitative study can be used to generate new hypotheses "from scratch". In a second step, these hypotheses are tested using a (representative) quantitative study [May01, §22].

Generalisation In the generalisation setup, qualitative methods have a greater weight than in the preliminary study setup noted above. Here, a complete qualitative study on a given question is performed, which produces rich knowledge on a single case. This knowledge is generalised in a second step using a quantitative approach [May01, §23].

Elaboration The elaboration setup is in some sense the counterpart of the generalisation setup. Here, the first step in research is a (representative) quantitative study. Afterwards, the results are refined using qualitative methods [May01, §24].

The motivation for elaboration of a quantitative study may be of several kinds. Firstly, qualitative research can be used to identify some *causality relationship* behind some statistical correlation found in the quantitative study. Secondly, qualitative research can illustrate an exemplary case which is a *typical representative* of the quantitative data, e.g., one that is close to the arithmetical mean or median. Thirdly, qualitative research can be used to explain *extreme cases*: While in quantitative research, outliers are eliminated using statistical methods, in qualitative research they are of special interest, as they are thought to enrich the theory by raising new questions.

Triangulation The most complex combination of qualitative and quantitative methods is called triangulation. In this setup, the researchers enquire into a common question from different points of view using different research methods, both qualitative and quantitative. Neither the qualitative nor the quantitative methods dominate necessarily. Furthermore, the results of all used methods are used to create a more complete picture of the research object, than it could be achieved by using only either qualitative or quantitative methods.

Combination setups in SE research The *preliminary study* setup is often applied implicitly before conducting quantitative SE research, since no comprehensive theory of SE has emerged yet. Examples for the *generalisation* and *triangulation* setups are given in Section 4.6.2. The *elaboration* setup is applicable to the wealth of quantitative studies that have been conducted in SE.

4.6 Exemplary Study Designs

In this section, two exemplary study designs are proposed. Firstly, opportunities to analyse architectural design decisions (in the sense of description decision theory) are discussed (Section 4.6.1). Then, based on a research report describing experiences in an academic/industrial collaboration project, an action research setup for a software development process is proposed (Section 4.6.2).

4.6.1 Example 1: Analysis of Architectural Design Decisions

The purpose of the research discussed here is to understand how decisions concerning software architecture (high-level design decisions) are actually made in software development. The findings may be used to discover deficiencies in decision processes, and to design new decision processes and supporting tools for future software development projects.

We will first present possible research setups (Section 4.6.1), and then discuss the opportunities for conducting quantitative and qualitative research within these setups (Section 4.6.1).

Possible setups

Three distinct basic settings concerning the research object are of interest: Either a dedicated project may be conducted, which most probably will use students as developers, a real (single location) industrial project may be observed, or a real open source project can be analysed. The first case qualifies as in vitro research, the latter as in vivo research. The first case might also be considered in vivo concerning student development, but this is not considered here, since we set the focus on normal productive development. In this respect, a student project may merely act as an indicator for real industrial projects, which certainly impacts the validity of the research results considerably. We emphasise the "single location" property of an industrial development project to additionally distinguish it from an open-source development project, which is in case of a larger project virtually always global.

Concerning an industrial project, different activities of the researcher may be distinguished, which determine his role in the project. He may either only observe regular activities of the developers, conduct individual interviews or group discussions specifically conducted for the research project, or act as a regular member of the development team (here again, different development roles are possible). Of course, combinations of these roles are possible within one research project.

Concerning analysis of an open source project, we again distinguish three interesting settings: Firstly, an active participation in development and mailing-list discussions is possible, similar to the industrial project setting above. Since due to the character of the development projects, most discussions take place within mailing lists or newsgroups and are often archived, relevant data can be expected to be available publicly and at virtually no cost. Research may be conducted entirely ex post. So, as a second possibility, mailing lists can be analysed for the preparation of decisions and actual collaborative decision procedures, which may or may not be be formalised. Since not only mailing lists, but also source code repositories and other artefacts are archived, a third possibility is to conduct research on recovering design decisions from source code and other design documents. The opportunities discussed in this paragraph may also apply to certain industrial projects with respect to the necessary technical preconditions, but it is often difficult in practise for researchers to get access to the relevant data (and even more difficult to get permission to publish results).

Quantitative and qualitative methods

Quantitative analysis of some kind of research object is only meaningful if many similar research objects are available, or if many similar operations are applied to some research object. Since neither the structure of architectural design decisions nor the relevant types of such decisions are well-understood yet, it is not sound to regard any two architectural design decisions as similar in this respect a priori.

Research in all of the presented settings will therefore need to be chiefly qualitative. Quantitative data may be collected if at all from mining open source software repositories or mailing-list archives.

Thus, any quantitative data will need to be substantiated by qualitative research. A preliminary study may appear as the most natural setup in this context: First, a qualitative study is conducted, which must provide a categorisation of design decisions that may be restricted in applicability to the regarded development project. This study proposes some hypothesis on design decisions, which may be tested using a further quantitative study.

A generalisation setup is also very interesting in this context: By qualitatively analysing one or a few projects, a categorisation of design decisions may be inductively derived. Then, its applicability to a greater variety of projects is tested using some quantitative study.

The elaboration setting and the triangulation setting are not readily applicable since no sufficient theory of design decisions currently exists as discussed above.

4.6.2 Example 2: Action Research in a Development Process

We sketch an exemplary study design based on a research report [PG05]. The authors describe problems that small, informal software development organisations encounter when they approach programming-in-the-large, and summarise experiences made in an academic-industrial cooperation project. Although this was not explicitly considered in [PG05], the project can be interpreted as Action Research. The research methods used are participant observation [Fli98, p. 141–148] and discussions with the project participants.

As a result, the authors find several problems in the development process which can be categorised as follows: (1) implementation technology selection, (2) technology-specific deficiencies, (3) inefficient build process and version control, (4) selective use of programming language features, (5) requirements elicitation, (6) code documentation, (7) missing automated testing mechanisms [PG05].

Using this study as a basis, we now extend the work and draft a multi-dimensional research design. Table 4.2 shows finer-grained research topics and relates them to a primary category of research objects. It also shows possible research methods, which are presented on a relatively generic level and must be fine-tuned to fit the requirements of the research topics.

Object Category	**Research Topic**	**Research Method**
Software Artefacts	Code documentation	Content analysis (of existing documentation)
	Reconstruction of (missing) software architecture	Interviewing, source code analysis
Developers of Software	Selective use of programming language features	Source code metrics, Group discussion, Questionnaire survey
	Incentives for code documentation	Interviewing
	Version control misuse	Group discussion
	Implementation technology selection	Interviewing, group discussion, participant observation
Developers as Users of Software	Coping with tool deficiencies	Statistical analysis of tool usage, Interviewing, Observation
Learning to Develop Software	Knowledge transfer into commercial software development projects	Evaluation questionnaire survey

Table 4.2: Multi-dimensional Research Design

Exemplarily, we focus on the selective use of programming language features. The authors of [PG05] found that only a subset of C++ language features was used in the project *(quantitative method: source code metrics).* In discussions *(qualitative method: group discussions)* they found technical, organisational and personal-skill-related reasons why the developers did not use specific language constructs.

At this point, possible further research steps would be to transfer academic knowledge into the project team, and to evaluate the success afterwards *(quantitative method: evaluation questionnaire survey and analysis of new source code).*

In each new research step, research methods must be chosen that are appropriate to the respective research topic. The results of the different steps must be combined and integrated into a larger theoretical context. In this way, they refine the problem description and bring up new questions. This iterative process ends, when no new aspects to the research question are found, and the problem leading to the academic-industrial cooperation can be regarded as sufficiently solved. This setup can be characterised as *triangulation*.

Furthermore, the question of generalising the results of Action Research may arise. This question may be addressed using quantitative methods after Action Research is completed, following the *generalisation* setup.

4.7 Conclusion

In this paper, we discussed the motivation for integrating both quantitative and qualitative research methods in Software Engineering research projects. Furthermore, we exemplarily discussed opportunities of applying such integration to Software Engineering research questions.

The general problems of implementing qualitative research in Software Engineering apply to the integration of quantitative and qualitative research approaches as well: Time pressure and confidentiality issues in software projects affect qualitative research more than it does quantitative research.

On the other hand, the approach of Action Research which involves qualitative elements results in more immediate benefits for the participating organisation than mere quantitative research does. Quantitative research aims at produce general results, which necessarily reduces the significance of results for specific cases. Therefore, Action Research could provide new incentives to software developing organisations for participation in research projects.

Bibliography

[ACMnd] ACM: ACM Computer-Human Interaction Special Interest Group. n.d., URL `http://www.sigchi.org/`

[Bas96] BASILI, V. R.: The role of experimentation in software engineering: past, current, and future. In: *Proceedings of the 18th international conference on Software engineering*, IEEE Computer Society, 1996, ISBN 0-8186-7246-3, pp. 442–449

[BWH96] BASKERVILLE, R.; WOOD-HARPER, A.: A critical perspective on action research as a method for information systems research. In: *Journal of Information Technology* 11 (1996), pp. 235–246

[CB01] COGHLAN, D.; BRANNICK, T.: *Doing Action Research in Your Own Organization.* London, UK: SAGE Publications, 2001

[CH04] CHEN, W.; HIRSCHHEIM, R.: A paradigmatic and methodological examination of information systems research from 1991 to 2001. In: *Information Systems Journal* 14 (2004), pp. 197–235

[FF01] FARBEY, B.; FINKELSTEIN, A.: Evaluation in Software Engineering: ROI, but more than ROI. In: *Proc. of the 3rd International Workshop on Economics-Driven Software Engineering Research (EDSER-3 2001)*, 2001

[Fli98] FLICK, U.: *An Introduction to Qualitative Research.* London, UK: SAGE Publications, 1998

[GRV04] GLASS, R. L.; RAMESH, V.; VESSEY, I.: An analysis of research in computing disciplines. In: *Commun. ACM* 47 (2004), № 6, pp. 89–94, ISSN 0001-0782, doi:10.1145/990680.990686

[GS75] GLASER, B. G.; STRAUSS, A. L.: *The discovery of grounded theory : strategies for qualitative research.* New York, USA: Aldine, 1975

[GS01] Groeben, N.; Scheele, B.: Dialogue-Hermeneutic Method and the "Research Program Subjective Theories". In: *Forum Qualitative Social Research (on-line journal)* 2 (2001), № 1, URL `http://www.qualitative-research.net/fqs-texte/2-00/2-00groebenscheele-e.htm`

[HT02] Hunt, A.; Thomas, D.: Software Archaeology. In: *IEEE Softw.* 19 (2002), № 2, pp. 20–22, ISSN 0740-7459, doi: 10.1109/52.991327

[Kar02] Karlström, D.: *Increasing Involvement in Software Process Inprovement.* Licentiate thesis, Lund University, Lund, Sweden, 2002

[KD88] Kaplan, B.; Duchon, D.: Combining Qualitative and Quantitative Methods in Information Systems Research: A Case Study. In: *MIS Quarterly* 12 (1988), № 4, pp. 571–586

[Kel63] Kelly, G. A.: *A Theory of Personality. The Psychology of Personal Constructs.* New York, USA: W. W. Norton and Company, 1963

[Lam93] Lamnek, S.: *Qualitative Sozialforschung. Band 2: Methoden und Techniken.* Weinheim, Germany: BeltzPVU, 2. edn., 1993

[Lam05] —— *Qualitative Sozialforschung.* Weinheim, Germany: BeltzPVU, 2005, ISBN 3-621-27544-4

[Lee00] Leeuw, F. L.: Evaluation in Europe. Opladen, Germany: Leske+Budrich, 2000, pp. 57–76

[Lew48] Lewin, K.: Action Research and Minority Problems. In: Lewin, K., ed., *Resolving social conflicts : selected papers on group dynamics*, New York, USA: Harper, 1948

[May93] Mayring, P.: *Einführung in die qualitative Sozialforschung. Eine Anleitung zu qualitativem Denken.* Weinheim, Germany: BeltzPVU, 2. edn., 1993

[May00] —— Qualitative Content Analysis. In: *Forum: Qualitative Social Research (on-line journal)* 1 (2000), № 2, URL `http://www.qualitative-research.net/fqs-texte/2-00/2-00mayring-e.htm`

[May01] —— Combination and Integration of Qualitative and Quantitative Analysis. In: *Forum Qualitative Social Research (on-line journal)* 2 (2001), № 1, URL `http://www.qualitative-research.net/fqs-texte/1-01/1-01mayring-e.htm`

[May03] —— *Qualitative Inhaltsanalyse : Grundlagen und Techniken.* № 8229 in UTB für Wissenschaft, Weinheim: Beltz, 8. edn., 2003, ISBN 3-8252-8229-5

[Mis03] Mish, F. C., ed.: *Merriam-Webster's collegiate dictionary.* Springfield, Mass., USA: Merriam-Webster, 11. edn., 2003, ISBN 0-87779-807-9

[MSRnd] Workshop Series on Mining Software Repositories. n.d., URL `http://msr.uwaterloo.ca/`

[Oat03] Oates, B. J.: Widening the Scope of Evidence Gathering in Software Engineering. In: *STEP '03: Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice (STEP'03)*, Washington, DC, USA: IEEE Computer Society, 2003, ISBN 0-7695-2218-1, pp. 59–64

[PG05] Ploski, J.; Giesecke, S.: When Small Outgrows Beautiful. Experiences From a Development Project. In: *Proc. of ECOOP Workshop on Practical Problems of Programming in the Large (PPPL2005)*, 2005. Accepted for publication

[PPInd] Psychology of Programming Interest Group. n.d., URL `http://www.ppig.org`

[RT03] Runeson, P.; Thelin, T.: Prospects and Limitations for Cross-Study Analyses. In: Jedlitschka, A.; Ciolkowski, M., eds., *WSESE'03 – 2nd Workshop in Workshop Series on Empirical Software Engineering*, Fraunhofer IRB Verlag, Stuttgart, Germany, 2003, pp. 133–142

[Sea99] Seaman, C. B.: Qualitative Methods in Empirical Studies of Software Engineering. In: *IEEE Transactions on Software Engineering* 25 (1999), № 4, pp. 557–572, ISSN 0098-5589. Special Section: Empirical Software Engineering

[Zen01] ZENDLER, A.: A Preliminary Software Engineering Theory as Investigated by Published Experiments. In: *Empirical Software Engineering* 6 (2001), № 2, pp. 161–180

5 Patterns in Building Architecture and Software Engineering

Marko Boskovic <marko.boskovic@informatik.uni-oldenburg.de>

Abstract

This paper outlines similarities and differences between patterns of civil architecture documented by Christopher Alexander and software patterns. Software patterns became an important tool in software development during the last ten years. They present a structure of solutions for common problems in software engineering. Usually experienced developers discover patterns by identifying solutions to common problems in their work. They reuse these solutions every time when they face similar problems, because they have already been proven in practice. Software patterns were inspired by Christopher Alexander, who established patterns in civil architecture. Although the idea of software patterns and patterns in civil architecture is the same, they are essentially very different.

5.1 Introduction

During the last ten years, software patterns started to attract great attention of the software engineering community. The book *Design Patterns: Elements of Reusable Object Oriented Software* [GHJV95] made a breakthrough, and introduced software patterns to the wide audience. This book is still one of the most selling books on software engineering. The question which arises is, what is the reason for such success? Why are they so popular? What kind of problems do they solve?

Software patterns are attempts do describe successful solutions to common problems in software development. They help people to reuse successful practices [SFJ96]. Using them, people better communicate and share ideas, and what is most important, reason about success of particular solutions.

Software patterns are not an original idea of software developers. It is rather an idea taken from civil architecture. In fact, the inspiration for software patterns were patterns of civil architecture. Patterns in civil architecture are established by Christopher Alexander. In his work he defines patterns as a three-part rule that relates some particular context, the problem, and the solution to this problem. He also calls it a thing that at the same time happens in the world, and the rule that tells us how to create that thing [Ale79].

In software development as it is in civil architecture, patterns are relation between the context, the problem, and the solution. This led to many miss conclusions in early days of software patterns. Software engineers were looking for analogy with patterns in architecture under every rock. During the time it became clear that both different nature of artifacts and development process are breaking analogies with Alexander's patterns [Cop96]. Although analogies break, vocabulary of software patterns community stayed as it is in his work. Pattern, pattern language, forces are terms that Alexander established. However, the greatest legacy to the pattern community is his vision and value system which is not common to most software practice [Cop96].

The paper starts with the description of the current process of planning the buildings and apartments as well as some shortcomings of that process. Third section explains architectural patterns and pattern languages. After it, comes section where software patterns and pattern languages are explained. Fifth section outlines similarities and differences between software patterns and languages and architectural patterns and languages. The last section is conclusion. At the end comes appendix with descriptions of patterns used as examples.

5.2 Designing a Building in Civil Architecture

Patterns in software architecture were inspiration for applying the same methodology in building software. Architectural patterns were discovered by Christopher Alexander [AIS77] Building is a well organized whole which should satisfy needs of humans that live in it [Dan05]. Needs of residents of buildings should be satisfied in apartments and common rooms and areas. Common rooms and areas are elevators, stairs, rest rooms, garages, playgrounds etc. Number and types of common rooms depend on decision of owner of a building.

Apartments consist of several rooms which can be divided into two categories, rooms for day living and rooms for night rest. Rooms for day living are:

- Living room, in which family gathers after some activities, lunch, business, school etc.
- Kitchen with dining room
- Working room
- Hole for communications among the other rooms with toilet.

Rooms for night rest:

- Rooms for grownups
- Rooms for parents
- Rooms for children

Beside number of rooms very important is their connectivity which depends on functional needs of humans. For a description of functional requirements we will start from the entrance in the apartment. The first room is the entrance hole. The entrance is connected to central hall, and central hall functionally connects most of rooms in apartment and usually is room that does not have natural light. The toilet should be near the apartment's entrance due to human needs to use it after doing some activities outside. In case of smaller apartments, the toilet is a bathroom at the same time. In larger apartments the bathroom is in a part of the apartment which is used for the night rest in order to provide better intimacy.

The kitchen should be also reachable from the entrance. The kitchen is meant for preparing food, so it should be easy to bring foodstuff. Carrying foodstuff and other things needed in the kitchen all over the apartment until you reach the kitchen would be dysfunctional and of

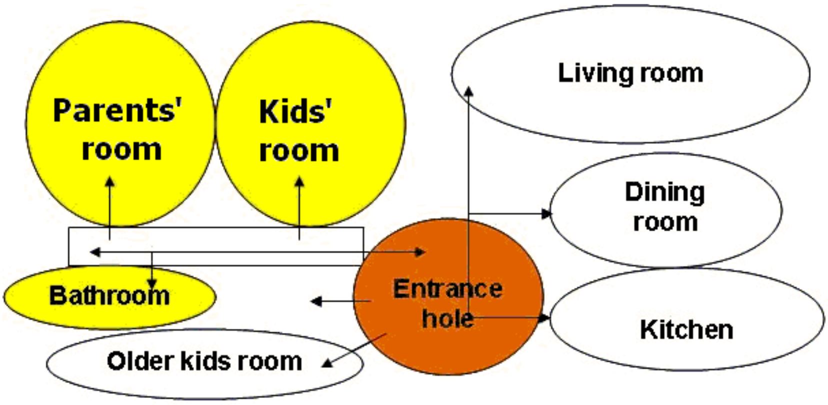

Figure 5.1: An example of a basic schema of an apartment. Yellow presents rooms for night rest and white are rooms for day living [Dan05]

course it could leave some unpleasant smell in rooms in which should day activities be made.

The dining room should be connected with the kitchen as well as with the central hole. It is mostly located at the sequel of the kitchen, or should be a part of it. The dining room should be naturally enlightened. Together with living room play a vital role in both the design and the composition of other rooms. The dinning room has to be directly connected with the living room physically and visually, in order to allow a visual contact with children that can be playing in living room.

The living room is a gathering place of a family and should have a visual contact with the central hole and the entrance to enable to monitor who enters the apartment. An important fact is that the main room for the day life should be oriented to south or southwest, as it is in case of parents and kids room. Both parents and kids' rooms are in the part of the apartment that is meant for night rest.

Taking into account all aforementioned facts the principal example of an apartment design can be seen in Figure 5.1.

To decide which dimensions rooms should have, we use *anthropometry*, a science for measuring dimensions of human body in respect to parts of apartments. This science standardizes the smallest dimensions of parts of apartment, while the biggest ones are not standardized. For instance, the smallest width of a hallway is 90 cm. That is enough for one person to walk through and if other person has traversed to wait aside. Furthermore, it defines *module*, and that is an increment of dimensions. Adopted modules are 30, 60 and 90, but common used modules are 30 and 60. In case of the hallway, module is 30, so 120 is hallway which allows that two persons pass by each other without waiting, 150 is enough for two persons to pass by each other with carrying some baggage, etc. It is not only places where people pass by each other that are standardized. Dimensions of other immobile parts of the apartment are standardized. As and example we can use a window. The window is not a place where people pass against each other. According to the average dimensions of human body, it is standardized that height of window sill should be at least 90 cm, but in most of cases is 120 cm , because of possibility of dropping out.

As we have already mentioned, dimensions and look of apartments in one building depends on the owner of the building. If the owner of the building plans to live in it, then it will be more adjusted to his way of living and personality. However, the owner plans to sell apartments or building in most cases, so that the owner can actually make some profit.

When persons who will live in apartments are not known at the time of designing apartment then it is left to the imagination to architects to find a way to fulfill needs that are common to each person a potential apartment tenant.

Unfortunately, although much of the dimensions are made to fit human needs, most of the people were not enjoying their life in apartments developed in this way, regardless whether they took a part in the development process as the future owner or not. Most of simple needs of their life were not satisfied. People were not satisfied with their apartments. Although everything may look fine in the project, it can have shortcomings that are important for living and not visible from the projects' blueprints. Christopher Alexander, whose work is inspiration for software patterns, in his early work 'Notes on the Synthesis of Form" [Ale64] mentions that current methods fail to fulfill real needs of real people because process of designing is artificially separated to model, process, context and artifact. Instead, they are intertwined aspects of the same system. This separation makes real micro-adaptation to real human needs hardly possible. Problems he identified in *Notes* are [Lea94]:

- Inability to balance individual, group, societal, and ecological needs
- Lack of purpose, order and scale
- Aesthetic and functional failure in adapting to local physical and social environments
- Development of materials and standardized components that are ill suited for the use in any specific application
- Creation of artifacts that people do not like

After identifying these problems, Alexander started to look at traditional architecture to try to find solutions which could solve these problems. As part of cultural heritage, people were inheriting good practices of building houses. Using those practices people were making artifacts highly adapted to its particularities. This was possible because building houses was in the hands of people that would live in it [Ale99]. People that were using those practices do not have to be aware why those practices work. He believed that by documenting this kind of practices, he would help architects to shape the artifacts, so that they better fit to human needs. Practices that were parts of ones community cultural inheritance were actually patterns pattern languages.

5.3 Architectural patterns and pattern languages

Patterns actually represent experience of generations that were building architectural artifacts for their own use. Accordingly, patterns are solutions to common problems in development of buildings for living. The solutions are general and the problems can be found in some particular context. At the same time, they are the thing that happens in the world, the rule which tells us how to create that thing, and when we must create it. It is both the thing and the process for creating that thing [Alexander, 1979].

Alexander discovered patterns by analyzing building and town structures of several cultures. Those structures are solutions that evolved during the ages, so that they could fulfill cultural, personal and social needs. As an example, let us consider one pattern from his book "A Pattern Language" [AIS77]:

Low Sill:

...this pattern helps to complete NATURAL DOORS AND WINDOWS (221), and the special love for the view, and for the earth outside, which ZEN VIEW (134), WINDOW PLACE (180) and DOWS OVERLOOKING LIFE (192) are need.

***

One of a window's most important functions is to put you in touch with outdoors. If the sill is too high, it cuts you off.

[. . .] People are drawn to windows because of the light and the view outside-they are natural paces to sit by when reading, talking, sewing and so on, yet most windows have sill height about 30 inches or so, so that when you sit down by them you cannot see the ground right near the window. This is unusually frustrating-you almost have to stand up to get a complete view.

[. . .] On the other hand, glass all the way down to the floor is undesirable. It is disturbing because it seems contradictory and even dangerous. It feels more like a door than a window; you have the feeling that you ought to be able to walk through it. If the sill is 12 to 14 inches high, you can comfortably see the ground, even if you are a foot or two away from the window, and it still feels like a window rather than a door.

Therefore:

When determining exact location of windows also decide which windows should gave low sills. On the first floor, make the sills of windows which you plan to sit by between 12 and 14 inches. On the upper stories, make them higher, around 20 inches (see Figure 5.2).

***

Make the sill part of the frame, and make it wide enough to put things on-WAIST-HIGH SHELF (201), FRAMES AS THICKENED EDGES (225), WINDOWS WHICH OPEN WIDE (236). Make the window open outward, so that you

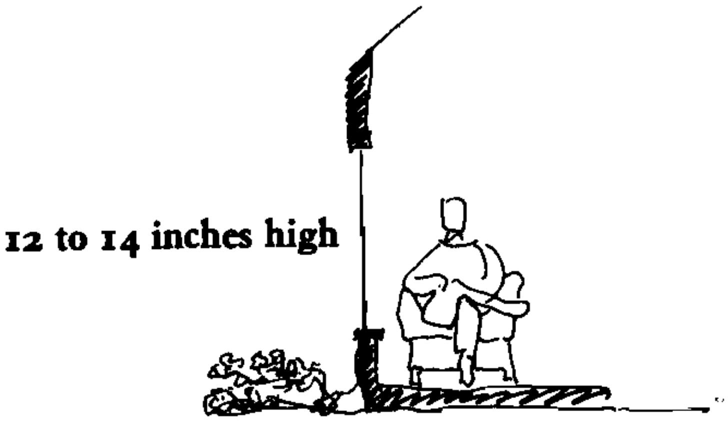

Figure 5.2: Sketch of the pattern Low Sill

> can use the sill as a shelf, and so that you can lean out and tend the flowers. If you can, put flowers right outside the window on the ground or raised a little, too, so that you can always see the flowers from inside the room-RAISED FLOWERS (245)...

As we can see the pattern is generative solution to a problem. It does not make a clear statement how the problem should be implemented, but only the principal structure that solves the problem. Accordingly, patterns can be applied million times without being the same twice [AIS77]. Other dimension of the pattern is that it cares about people, and their comfort in architectural artifacts. In the pattern we can see that the pattern care about what do people like and according to that makes proposal for solution. It is different from anthropometry, and we can see that on an example of height of sill standardized by anthropometry is at least 90 cm. The reason for this height is the possibility of dropping out. The conclusion is that the standardized height of the window sill should protect life, but this protection should not be part of a size of window because it constraints life.

The documentation of this structure is a pattern as well. The documentation that describes the patterns is a literary form. The literary form should be used as a tool for: introducing a reader to a problem, to describe where the context might arise, to analyze the problem, and to present and explain solution [Cop96]. The literary form in which patterns of civil architecture are explained is a text that is divided, into several groups of three diamonds and with a word "Therefore". The first one there is a picture of an example of the pattern. The paragraph that introduces the context in which pattern can be applied comes after the picture. The context is usually a larger pattern that this pattern helps to complete. After this paragraph come three diamonds which mark the beginning of the problem. The description of problem starts with a bold statement that gives the essence of the problem. After comes the body of the problem which explains the empirical background of the pattern, the range of different ways the pattern can be manifested in a building, and so on. Then, after the word "Therefore", in the bold type comes the solution. The solution describes both social and physical relationships that are required to solve the stated problem, in a form of an instruction. After the solution, comes a diagram that shows the structure of the pattern with labeled important parts of it. Then, separated by three diamonds come a part of the text that connects this pattern to smaller patterns in the pattern language they are actually parts of it [AIS77].

From the structure of the form, we can clearly see that we can not talk about a pattern in isolation, but it is always connected with other patterns. A collection of patterns that work together under a rule of composition, and that build on each other to generate a system is called *pattern language* [Cop96]. The name pattern language should describe that patterns and rules for their composition represent one whole, the way it is in natural language. In natural language words do not make a language without connection rules. Accordingly, each pattern depends on the patterns that are in relations to it. Related patterns can be either smaller patterns which are part of the pattern or a larger pattern the given pattern is contained by. Patterns documented in *A Pattern Language* [AIS77] go from very large patterns that represent regions and towns, through smaller and smaller which represent neighborhoods, clusters of buildings, buildings and at the end they finish with details of constriction. In Figure 5.3 we can see an example of composition of patterns [Cop04].

From Figure 5.3 we can see that one pattern consist of several smaller patterns, and it also is a part of a larger one. Larger pattern consists on smaller ones, and relies on functionality they provide. However, there are also smaller patterns that are parts of two larger patterns. This

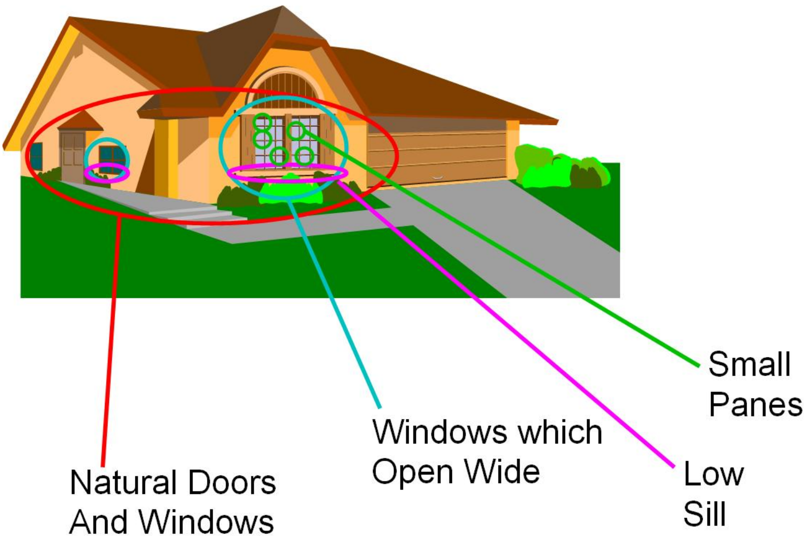

Figure 5.3: An example of composition of patterns [Cop04]

kind of relation is result of interweaving these two patterns. In this terms pattern language is directed acyclic graph formed by patterns and their relationships. An example of a pattern language we can see in Figure 5.4.

An important property of pattern languages is that they create one whole. The whole is defined by a set of properties it should have. Examples of wholes are a house, a software system, a software framework, etc. A pattern language is complete when the system of patterns it defines is fully capable of allowing all its inner forces to resolve themselves [Cop04]. However, that does not mean that the whole is isolated from another architectural artifact. They can be a part of another larger whole. Accordingly, we can clearly see that one pattern language can be also a part of another larger pattern language which defines that larger whole.

A whole is a thing that is discovered and fulfills requirements that serve human needs. Pattern languages help develop this thing, so that it has all needed properties. There is no a pattern language that helps create only one property of the whole. As an example we can use a

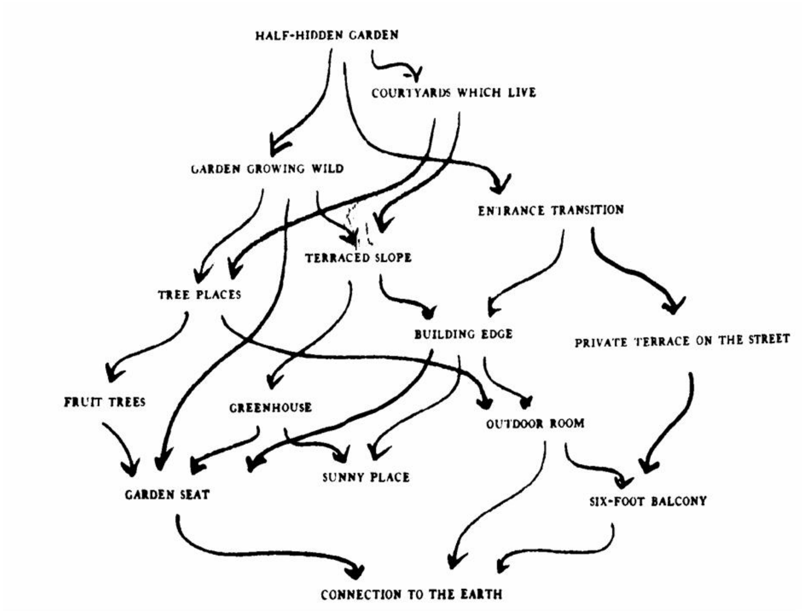

Figure 5.4: Half-Hidden Garden pattern language. This pattern language is a part of a larger pattern language for building a house. [Cop04]

house. Important properties of the house are that it should be beautiful, safe, economical etc. Therefore, patterns help generate a structure that has all these properties, but the thing they are building is not "safety", "beauty", or something else. Trying to discover a pattern language for one property is impossible. The whole the pattern language help develop are wholes that community has discovered and that has all the properties humans need from this artifact. The pattern language creates system that has many desirable attributes [Cop04].

Patterns and pattern languages are not only peace of literature that is used for documenting good proven practices in architectural design. Upon the basis of patterns and pattern languages lie deeper ides. First of all, patterns and pattern languages have a moral capacity of producing a living structure, the structure that is in constant interaction with its environment, and which makes human life better. "The most successful designs are not those that try to fully model the domain in which they operate, but those that are 'in alignment' with the fundamental structure of that domain, and that allow for modification and evolution to generate new structural coupling" [WF86]. The second idea is that they should be generative things with the capability of producing coherent wholes. Pattern languages are not precise plans for building the entity. They are rather principles and principle structures. On one hand, an implementation depends on architects and civil engineers. One the other hand, it depends on the micro structure of the area where artifact should be created.

Christopher Alexander's work influenced software designers, over all because of the idea to exchange proven ideas and basing the future work on something that has already worked in the past. In software design there is a constant idea to reuse implemented peaces of one software product in the implementation of a new software artifact. The paradigm taken from patterns of civil architecture seemed like one step forward to making this idea real.

5.4 Software patterns and pattern languages

During the years of software development, software developers had both success and failures in their projects. In the 1960s "software crisis" caused the birth of software engineering. Software engineering is defined as a set of formalisms, methods, and practices for producing reliable, economical, efficient software systems that meet their specifications -

"reesstms" [DD94]. During ages of software engineering there were many inventions like structured programming, data abstractions, information hiding, program verification and many more. The failure of one was followed by invention of another. However, it seems that software engineering could not fulfill expectations that it should.

Starting from 1970s, as an answer to the software engineering inability to produce "reesstms", a software design received a great deal of the attention. Software design is a set of practices and implementation techniques that allow for construction of software which is provided for satisfying the users. Software designers claim that software design is a craft and it can be learned only through apprenticeship and the process of designing. Through apprenticeship one can learn techniques that are discovered by experienced developers and which were already working in the past. Software patterns make another step forward. They try to capture the experience of many, document it and make it available to the software community.

Software patterns are structures that solve common problems in software engineering. They are discovered by experienced developers. Experienced software developers do not solve each problem from bottom line. They rather use some solutions that worked for them in the past [GHJV95]. Since the solution that is proved for them in the past, is probably going to work for them in this new situation. Knowledge of those solutions is actually the thing that makes them good designers.

Software patterns started to be documented at the end of the 1980's when Ward Cunningham, Kent Beck, James O. Coplien and Erich Gamma started to collect and document recurring solutions in different areas of software development [BCC$^+$96], all of them inspired by the work of Christopher Alexander. Soon they together with some other people intensified their research in pattern area what led to forming "Hillside Generative Patterns Group" (www.hillside.net).

Beside the moment of gathering as the Hillside Group, an important event was the publication of still one of bestseller books on software design *Design Patterns: Elements of Reusable Object Oriented Software* [GHJV95] popularly known as the Gang of four (GoF) book. It is "a book that helped people conceptualize beyond individual design relationships, grasp important structures of micro-architectures, and value proven solution strategies over raw innovation" [Ale99]. The book consists of 23 documented patterns that names, abstracts and identifies the key aspects of design recurring structures discovered in developing of graphical user interface [Hel95]. Success of this book encouraged people to document software patterns in their area of development. This way they could exchange knowledge, and make software better.

It is important to distinguish patterns from paradigms, idioms, principles, heuristics, architectures, frameworks and role-models [Vil95, Cop96]. *Paradigm* is a style of work that is followed through the design of a whole system. *Idiom* is a language specific typical way of using and combining elementary building blocks. It is a language specific pattern. *Principle* is a design role that a designer follows “always” when designing software. For example, important principles of reusable object oriented programming are [GHJV95]: *Program to an interface and not implementation* and *Favor object composition over class inheritance. Heuristics* is experience which does not absolutely claim that actions taken will actually work. *Architecture* is a total structure of an application, possibly described by the multiple patterns involved, which are then often called “micro-architectures”. *Framework* is a collection of classes that work together to accomplish a parameterizable task. Often collaboration between classes in the framework is organized using design patterns. *Role-models* describe a single coordinated collaboration among multiple participants and role models can be something closest to formalize patterns [Vil95].

As an example for software patterns the *Composite* design pattern from [GHJV95] is described in the Appendix.

5.4.1 Software patterns forms

From the Composite design pattern example, it can clearly be seen that software patterns are described in another literary form then patterns of civil architectures are. Despite the fact that forms are different, content of these forms clearly identifies same important parts of pattern as they are identified in the form of Alexander’s work. Most popular forms for software patterns are The GoF (Gang of Four) Form, The Coplien Form and The Portland form.

The GoF form is established in [GHJV95] consists of following 13 sections:

1. *Name and Classification* - The name is very important for a pattern. Since it will become part of a vocabulary, it should describe the essence of the pattern in a short way. The classification places the pattern according to scheme presented in the book. Patterns can be placed according to their purpose to one of three groups:

Creational, Structural or Functional, and according to scope of application they can be *Class* or *Object*.

2. *Intent* - is a short statement that answers to questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

3. *Also Known As* - other well-known names of the same pattern if there are any.

4. *Motivation* - a scenario that illustrates the design problem as well as both class and object structures in the pattern that solve the problem. Although this is not place where the pattern is strictly defined we can clearly see the structure of the pattern from an example that is presented.

5. *Applicability* - In the form of bulleted answers to the following questions: What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

6. *Structure* - represents the structure of the pattern with both class and interaction diagrams.

7. *Participants* - responsibilities of classes in the structures.

8. *Collaboration* - presents how the participants in patterns collaborate to carry out the responsibilities.

9. *Consequences* - How does the pattern support its objectives? What are the tradeoffs and results of using the pattern? What aspect of the system structure does it let you vary independently?

10. *Implementation* - What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

11. *Source Code* - Code fragments that illustrate how the pattern might be implemented in C++ or Smalltalk.

12. *Known Uses* - Examples of the pattern found in real systems.

13. *Related Patterns* - Section that answers to questions like: What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

The Composite pattern is described using this form, although not all sections are presented in the paper due to the size of the paper. The Coplien form consists of seven sections and mostly is based on the form that Alexander uses [Cop96, Cop04]:

1. *Pattern name* - it is common to give the name using a noun or a short verb phrase.
2. *Problem* - The problem is often presented with a question or a design challenge.
3. *Context* - The description of the context in which the problem can be found and applied. It is not a part of the pattern, but it is a placeholder of the pattern.
4. *Forces* - Describe design trade-off; what pulls problem in different directions, toward different solutions? Forces are not mechanical, but rather social, economic, psychological etc.
5. *Solution* - Solution explains how to solve the problem.
6. *Rationale* - Explains why the pattern work, and what history behind the pattern is. Emphasis the importance of principles behind the pattern.
7. *Resulting Context* - Explains which forces the pattern resolves and which forces are not resolved. It also contains a list of patterns that might be considered next.

The Portland form is used for online pattern repository that Ward Cunningham maintains (http://c2.com.ppr/). The Portland form is more narrative, not separated by outlined sections of text. It is an emulation of Alexander's form with some simplifications in typesetting. Each document in Portland repository contains a pattern language. Patterns are represented as paragraphs in the Portland form. Each pattern makes a statement that goes something like: "such and so forces create this or that problem, therefore, build a thing-a-ma-jig to deal with them." The pattern takes its name from the thing-a-ma-jig, the solution. Each pattern in the Portland Form also places itself and the forces that create it within the context of other forces, both stronger and weaker. They also place them selves in the places of language according to solutions they require. Patterns in the Portland form capture ordering of good designers, which resolve stronger forces first, by citing stronger and weaker patterns in opening and closing paragraphs. At the end, the structure looks like [Cun94]:

- Having done so and so you now face this problem...
- Here is why the problem exists and what forces must be resolved...

Therefore:

- Make something along the following lines. I'll give you the help I can...
- Now you are ready to move on to one of the following problems...

Each document has summary section as well. Since pattern languages can be very long sometimes, patterns which work around similar ideas can be found in them. These patterns are introduced in the summary section. The summary section also introduces the problem which the patterns try to solve. An example of a pattern language written in the Portland form is the CHECKS pattern language on information integrity (http://c2.com/ppr/checks.html).

5.4.2 Software pattern languages

Following idea of *A Pattern Language* [AIS77], software developers tried to identify pattern languages in domains of their work. There was attempt to gather all the patterns identified for software development and to organize them in pattern language for object oriented software [Noble, 1998]. However, pattern languages are still not so common in software patterns community, and pattern community is far a way from discovering one large pattern language for software development.

At the moment, pattern catalogues are most common sources of the contemporary use, especially the GoF book on design patterns [GHJV95], series of *Pattern Oriented Software Architecture* catalogues [BMR$^+$96, SSRB00, KJ04], and books based on papers presented at *Pattern Languages of Programs - PLoP* conferences [CS95, VCK95,

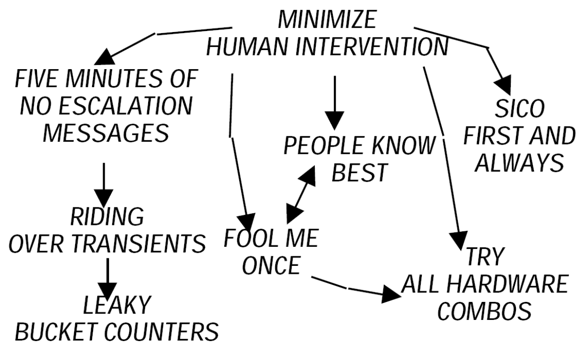

Figure 5.5: The pattern language identified in AT&T telecommunication systems [ACG+96]

MRB97, HFR99]. However, despite the fact that the books contain a huge number of patterns, they are rather catalogues of patterns then they are a pattern languages. Pattern languages create a whole in a particular domain. Software patterns, e.g. [GHJV95], are not completely enough to make guidelines for making all the programs of object oriented programming, which is their domain. Therefore, they do not present pattern language. Although pattern catalogues are the most common source in the contemporary use, there are also pattern languages identified by practitioners of a particular domain. Such a kind of a pattern language is presented in Figure 5.5.

Pattern language presented in Figure 5.6 is actually a part of a larger catalogue collection of patterns adopted in AT&T laboratories. These patterns are used in some other domains as well, but they are originally identified in the telecommunication community, here they are addressed as telecommunication patterns. To see relations between these patterns we will consider two patterns at the right side of the pattern language, patterns *Minimize Human Intervention* and *SICO First and Always*. Description of these patterns can be found in the Appendix.

From description of these two patterns, it can be seen that larger patterns are actually employing smaller ones for implementation. The thing that makes them a pattern language is that they represent a system of patterns that are relaying on each other and working together to solve some broader problems [Cop96]. The problem this pattern language solves is to improve reliability by minimizing human intervention in the system.

From the previous examples it can clearly be seen that idea of software patterns is to improve human comfort. That is one of properties of patterns in civil architecture, which was the inspiration for software practitioners of the pattern community. These two pattern areas have much in common. Many early members of pattern community tried to find analogies under every rock, nature of produced artifact and pragmatics of development process seem to break these analogies. Next section will try to outline similarities and differences between these two groups of patterns.

5.5 Similarities and differences between software patterns and civil architecture patterns

Christopher Alexander's work brought a completely different view on all the artifacts produced in industry. After the period of industrialization, it became clear that products are becoming more and more separated from humans. Products are being produced and assembled by standards which did not care about real human needs and the real nature of products. Christopher Alexander tried to get building back to the people that live in them, by fitting houses to their needs. Software developers which are member sof the pattern community are trying to do the same. However, although idea is very similar, patterns of different communities differ significantly. Similarities and differences are following:

1. ***Value system of Christopher Alexander and the members of pattern community are the same and include*** [Cop96]:

    - *The Quality Without a Name.* This quality is in Alexander's patterns quality to serve human needs, to help them feel better. It can not be explained in any other way, so it is without a name. In the software patterns community, it is a module of a system that just feels good. Many practitioners had this pleasure when they build part of a system that is satisfying.

- *Real Stuff.* Patterns are about some real stuff, about systems that are really implemented, so that they capture the proven practice, but not some theories, postulates or techniques that might work. They rather offer design principles built on the repeated success in applications. At this point, the pattern community quote addressed by Edsgar Dijkstra “Premature abstraction is root of all evil”.
- *Constraints Are Liberating.* Pattern form constraints a pattern writer. However, this constraint helps him/her to more focus on the real nature of the problem the pattern solves. It helps him/her not to go into the explanation of unimportant design elements.
- *Participative Architecture.* In patterns of software architecture it means that a user should participate in designing of the dwellings. They should not leave it only to a professional architect. In the software community this spirit drove to employing patterns for human interface design. It led to form a completely new part of the pattern community which deals with HCI (Human Computer Interaction) patterns. Members of this community force the involvement of users in designing user interfaces.
- *Dignity for Programmers.* Patterns are about the real stuff, so that they are a part of daily concerns of the programmer. Patterns are dedicated to techniques that enable to make a product that is deliverable to potential customers. That approach is totally different form learning them general theories of programming. They are making them skilled workers with knowledge of little tricks of the domain in which they are developing software. Architects have also an important role in this process, because they should supervise them all the time and try to tackle with them some important issues of the project, before they become the problem [Fow03].
- *Aggressive Disregard for Originality.* Patterns try not to make same mistake as all new computer science techniques and technologies do. Usually they get larger attention then they deserve because of removing shortcomings of their predecessors. Patterns try to focus to long proven practices.
- *The Human Element.* As all software should serve human needs at some level, patterns do not deal only with technical

  aspects of patterns, they also take care about human issues. The pattern language of telecommunication systems mentioned earlier in the paper is an example of such a language.

- *Aesthetics.* Some well-written code means that it is easy to understand. When the code is formatted to be easily readable, that improves the system's maintainability and improves human comfort of programmers. However aesthetics is not only about some nicely written code. It can be found in the nicely and easily understood overall structure of the system.
- *Interdisciplinary scope.* Software patterns try to understand the domain in which software patterns are applied. They try to combine organizations, processes, and other important parts of the domain in which are applied, so that they can connect human and technological concerns. The most powerful patterns integrate human concerns with technology.

2. ***Alexander's patterns are for development tangible artifacts and software patterns are not.*** Beside the system of values, both Alexander's patterns and patterns in software engineering talk about the structure. Alexander's patterns talk about relationships between real artifacts that can be seen, real tangible artifacts. Software patterns are more about functional structures of the system, since software is not tangible artifact.

3. ***Alexander pattern help make systems, so that they fit to human needs, while software patterns adopting hardware, so that it fits to human needs.*** Beside the fact that software is not a tangible artifact, there is another difference between software and architectural artifacts, namely software needs some hardware to be executed. While patterns in architecture try to build real tangible artifacts that can serve human needs, software patterns try to improve hardware as tangible artifacts, so that they can better serve human needs, taking human issues as guidelines for the software structure.

4. ***In architecture and in software, the same pattern can be found in different contexts.*** Software patterns can be used in other context, not only in the context in which they were discovered. Design Patterns [GHJV95] are discovered in development of graphic user interface, but they are also used across

various domains like operating system kernels, telecommunication switching systems etc [Hel95]. In case of patterns in architecture ENTRANCE TRANSITION is a part of the pattern language of garden, and the language for the house [Cop96].

5. ***Patterns in architecture are applied in progressive order, while patterns in software are applied in iterative and incremental development.*** Alexander's way of applying patterns is in progressive order, from the top of the language to the bottom, while development of software is in iterative an incremental fashion.

6. ***Alexander's patterns are the product of people that were making architectural artifacts for their own use, while software patterns are the product of people that are developing systems for both their own use and the use of people not involved in computer science.*** The environment of patterns in civil architecture is nature, and they rely on natural laws. It is a place where people live since their very beginnings. Architectural artifacts are things that are made to improve the living area, and changing living area is one important stage in human evolution. During the history, people were adjusting nature in order to fit to their needs, and an important part was architectural artifacts. Environment of software patterns is invented. Software is an artifact which is based on rules that computer engineers invented, so it is not common to most of the people. Most of the people do not understand these rules, and they can not adopt it to personal needs. Instead, software developers do it.

7. ***Alexander's pattern improve human comfort only of users of architectural artifacts, software patterns improve the comfort of persons that use software systems as well as people that maintain them.*** Software patterns should improve human comfort as patterns in civil architecture should do. However, there is a significant difference. Alexander's patterns improve human comfort of users of the artifacts. These patterns should help produce artifacts in which only people, which are living in it, should enjoy them. Software patterns try to do it too. They identify solutions to problems in designing software users were satisfied with. However, there is another dimension of soft-

ware patterns. They improve the maintenance of software, and the comfort of developers of systems. Software patterns localize some functional parts of the system, so it is easy to look for causes of failures in the system. They also capture knowledge that is implicitly understood [SS95], so that they help less experienced developers to rapidly produce effective design [Hel95]. Furthermore, they provide a shared vocabulary and a common design resource for teams and organizations that improve communication between developers.

5.6 Conclusion

Software patterns became one of important parts of the software engineering community during the last ten years. Origins of software patterns can be found in civil architecture that is in the work of Christopher Alexander. This paper tries to compare Alexander's patterns and patterns in software. A review of the process of making a project of a house describes how plans of buildings are made. This kind of process of planning does not fulfill all the needs that human need. Although it may look perfect on the plans, people were not happy when they live in it. Christopher Alexander in his work points out that the buildings should have "A Quality Without a Name", quality that makes people enjoy their apartments. People should be really comfortable on places where they are living. He searches for this quality in several traditional cultures, cultures of the time when people were making their own houses. He believes that at that time people were enjoying more in places they lived, then they are enjoying now. His believes were proven with series of experiments [Ale99]. During his work he identified "patterns", solutions to common problems in particular contexts in architectures. Further in his work he identified "pattern languages", patterns that are calling on each other to make a whole. Pattern languages are in a form of directed acyclic graphs. This kind of building process helps architects make buildings in which people will really live and enjoy.

Software engineers were inspired with Alexander's work and annoyed with the inability of that time used methodologies and techniques that were not able to produce software that will be used to improve human comfort. Software that was produced was not reliable, economical, efficient software that meet their specification [DD94]. As Alexander, members of software pattern community started to write down solutions of experienced developers. Soon they realized that there are similar solutions which different developers use without knowing each other. As a result, new community was created which was dedicated to discovering patterns and pattern languages in software development. At the beginning software patterns community tried to find analogies with Alexander's patterns everywhere in patterns and pattern languages they were discovering. Soon, they realized that nature of artifact and development processes are different and that the software pattern community has to discover real nature of software patterns on their own.

This paper represents a comparison of software and patterns of civil architecture. In the paper a comprehensive introduction to both, patterns of civil architecture and software patterns are presented, and similarities and differences are outlined. Similarities are value system and different contexts in which patterns can be found. Differences are nature of artifact, and the kind of improvement of this different kind of artifacts. In case of software patterns it is not only customer that benefit, it is developer as well. In the paper only patterns concerning software were analyzed. Pattern community did grow very large last years, and patterns for software engineering went far beyond software. Discovery of the patterns nowadays goes in to different areas of software development even in process of software development and organization of software development as well [Cop04].

Appendix

Name: Composite

Intent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Motivation: Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to the form still larger components. A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives...

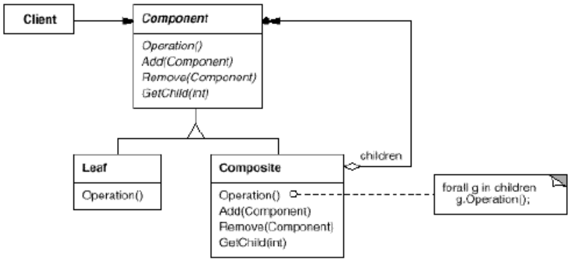

Figure 5.6: Structure (Sketch) of Composite design patterns [GHJV95]

Applicability: Use Composite pattern when:

- you want to represent part-whole hierarchies of objects.
- you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Structure: Figure 5.6 shows the structure (Sketch) of the Composite design patterns

Consequences: Composite design pattern

- defines class hierarchies consisting of primitive objects and composite objects...
- makes the client simple...
- makes it easier to add new kinds of components...
- can make your design overly general...

...

Known Uses: Examples of the Composite pattern can be found in almost all object-oriented systems. The original View class of Smalltalk Model/View/Controller was Composite, and nearly every user interface toolkit or framework has followed in its steps...

Related Patterns: Often the component-parent link is used for a Chain of Responsibility (223).

Decorator (175) is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. That means, decorators will have to support the Component interface with operations like Add, Remove, and GetChild.

Flyweight (195) lets you share components, but they can no longer refer to their parents.

Iterator (257) can be used to traverse composites.

Visitor (331) localizes operations and behavior that would otherwise be distributed across Composite and Leaf classes.

Pattern: Minimize Human Intervention

Problem: History has shown that people cause the majority of problems in continuously running systems (wrong actions, wrong systems, wrong button).

Context: High-reliability continuous-running digital systems, where downtime, human-induced of otherwise, must be minimized.

Forces: Humans are truly intelligent; machines are not. Humans are better at detecting patterns of system behavior, especially among seemingly random occurrences separated by time (People Know Best).

Machines are good at orchestrating a well thought-out, global strategy, and humans are not.

Humans are fallible; computers are often less fallible.

Humans feel a need to intervene if they can not see that the system is making serious attempts at restoration. Human reaction and decision times are very slow (by orders of magnitude) compared to computer processors. Quiet system is a dead system.

Human operators get bored with ongoing surveillance and may ignore or miss critical events.

Events, normal processing or failures, are happening so quickly that inclusion of the human operator is infeasible.

Solution: Let the machine try to do everything itself, deferring to the human only as an act of desperation and last resort.

Resulting Context: A system is less susceptible to a human error. This will make the systems customers happier. In many administrations, the system operator's compensation is based on the system's availability, so that this strategy actually improves the lot of the operator.

An application of this pattern leads to a system where patterns such as Riding Over Transients, SICO First and Always and Try All Hardware Compos apply to provide the system with the ability to proceed automatically.

Rationale: Empirically, a disproportionate fraction of high-availability system failures are operator errors, not primary system errors. By minimizing human intervention, the overall system availability can be improved. Human intervention can be reduced by building strategies that counter human tendencies to act rashly; see patterns like Fool Me Once, Leaky Bucket Counters and Five Minutes of No Escalation Messages.

Notice the tension between the pattern and People Know Best.

Author: Robert Hanmer, Mike Adams, 23.03.1995

Pattern: SICO First and Always

Problem: Making a system highly available and resilient in the face of hardware and software faults and transient errors.

Context: Systems where the ability to do some meaningful work is of utmost importance, but rare periods of partial application functionality can be tolerated. For example, the 1A/1B processor-based 4ESS switch from AT&T.

Forces: Bootstrapping is initialization. A highly-available system might require (re)initialization at any time to ensure the system sanity.

The System Integrity Control Program (SICO) coordinates the system integrity.

The system integrity must be in control during bootstrap.

The focus of operational control changes from bootstrap to the executive control during normal call processing.

The application functioning is very important.

The system integrity takes processor time, but that is acceptable in this context.

The system is composed of proprietary elements, for which design criteria may be required of all developers.

Hardware designed for fault tolerance which reduces the occurrence of hardware errors.

Solution: Give the system integrity the ability and the power to reinitialize the system whenever system sanity is threatened by error conditions. The same system integrity should oversee both the initialization process and the normal application functionality, so that initialization can be restarted if it runs into errors.

Resulting Context: In short, System Integrity Control has the major role during bootstrapping, after which it hands control over to the executive scheduler, which in turn lets the System Integrity Control regain the control for short periods of time on a periodic basis.

See also Audit-Derivable Constants After Recovery.

Rationale: During a recovery event (phase of bootstrap), SICO calls the processor initialization software first, the peripheral initialization software second, then the application initialization software, and finally transfers to the executive control. Unlike a classic computer program where initialization takes place first, and “normal execution” second, the SICO architecture does not place the software initialization as the highest level function. System integrity is at an even higher level than system initialization.

The architecture is based on a base level cycle in the executive control. After bootstrapping, the first item in the base cycle is SICO (though this is different code from that run during bootstrapping). After the SICO part of bootstrapping is done, the base level part of SICO is entered each base level cycle to the monitor of the system on a periodic basis.

System integrity must be alert to watch for failures during both the bootstrap and the normal base level operation. There is a system integrity monitor in the base level to watch timers as well as both overload control and audit control (not to run audits, but to ask audits if there are error conditions). These are checking in with SICO to report software and hardware failures and potentially request initialization.

During bootstrap and initialization, system integrity employs a number of similar mechanisms to monitor the system. For example, Analog Timers, Boot Timers, Try All Hardware Combos and others.

Much of the rationale comes from AUTOVON, Safeguard, missile guidance systems, and other high-reliability real-time projects from early AT&T stored program control experience. See the Bell System Technical Journal Vol. 56 No.7, Sept. 1977, pp. 1145-7, 1163-7.

Author: Robert Hanmer

Bibliography

[ACG+96] Adams, M.; Coplien, J.; Gamoke, R.; Hanmer, R.; Keeve, F.; Nicodemus, K.: Fault-tolerant telecommunication system patterns. In: (1996), pp. 549–562

[AIS77] Alexander, C.; Ishakawa, S.; Silverstein, M.: *A Pattern Language*. New York: Oxford University Press, 1977

[Ale64] Alexander, C.: *Notes on the Synthesis of Form*. Harvard University Press, December 1964

[Ale79] —— *The Timeless Way of Building*. Oxford University Press, 1979

[Ale99] —— The Origins of Pattern Theory: The Future of the Theory and the Generation of a Living World. In: *IEEE Software* (1999), pp. 71–82. Keynote speech at OOPSLA'96

[BCC+96] Beck, K.; Coplien, J. O.; Crocker, R.; Dominick, L.; Meszaros, G.; Paulisch, F.; Vlissides, J.: Industrial Experience with Design Patterns. In: *Proc. International Conference on Software Engineering, ICSE, Berlin*, IEEE CS Press, March 1996, pp. 103–114. Reprinted in Rising98

[BMR+96] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M.: *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, New York, 1996

[Cop96] Coplien, J.: *Software Patterns.* New York: Sigs Books, 1996, ISBN 1-884842-50-X

[Cop04] Coplien, J. O.: Practice and Theory of Patterns: Towards a General Design Theory, Lectures at FON-School of Business Administration, 20-24.9.2004, Belgrade., 2004

[CS95] Coplien, J.; Schmidt, D., eds.: *Pattern Languages of Program Design.* Addison-Wesley, 1995

[Cun94] Cunningham, W.: "About the Portland From", The Portland Patterns Online Repository. Available online [accessed 2005/09/09]. 1994, URL `http://c2.com/ppr/about/portland.html`

[Dan05] Danicic, D.: Personal communication, April 2005

[DD94] Denning, P. J.; Dargan, P. A.: A discipline of software architecture. In: *interactions* 1 (1994), № 1, pp. 55–65, ISSN 1072-5520, doi:10.1145/174800.174807

[Fow03] Fowler, M.: Who Needs an Architect? In: *IEEE Software* 20 (2003), № 5, pp. 11–13, ISSN 0740-7459, URL `http://csdl.computer.org/dl/mags/so/2003/05/s5011.pdf`

[GHJV95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design patterns: elements of reusable object-oriented software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN 0-201-63361-2

[Hel95] Helm, R.: Patterns in Practice. In: *ACM SIGPLAN Notices* 30 (1995), № 10, pp. 337–341, ISSN 0362-1340

[HFR99] Harrison, N.; Foote, B.; Rohnert, H.: *Pattern Languages of Program Design 4.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999, ISBN 0201433044

[KJ04] Kircher, M.; Jain, P.: *Pattern-Oriented Software Architecture: Patterns for Resource Management*. John Wiley & Sons, 2004, ISBN 0470845252

[Lea94] Lea, D.: Christopher Alexander: an introduction for object-oriented designers. In: *SIGSOFT Softw. Eng. Notes* 19 (1994), № 1, pp. 39–46, ISSN 0163-5948, doi:10.1145/181610.181617

[MRB97] Martin, R. C.; Riehle, D.; Buschmann, F., eds.: *Pattern languages of program design 3*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, ISBN 0-201-31011-2

[SFJ96] Schmidt, D. C.; Fayad, M.; Johnson, R. E.: Software patterns: introductions. In: *Communications of the ACM* 39 (1996), № 10, pp. 36–39, ISSN 0001-0782

[SS95] Schmidt, D.; Stephenson, P.: Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms. In: Olthoff, W. G., ed., *ECOOP '95—Object-Oriented Programming*, Springer-Verlag, 1995, vol. 952 of *Lecture Notes in Computer Science*, ISBN 3-540-60160-0, pp. 399–423

[SSRB00] Schmidt, D. C.; Stal, M.; Rohnert, H.; Buschmann, F.: *Pattern-Oriented Software Architecture Volume 2 – Networked and Concurrent Objects*. John Wiley and Sons, 2000

[VCK95] Vlissides, J.; Coplien, J. O.; Kerth, N. L.: *Patterns Languages of Program Design 2*. Reading, Mass.: Addison Wesley, 1995, ISBN 0-201-89527-7

[Vil95] Viljamaa, P.: The patterns business: Impressions from PLoP-94. In: *ACM SIGSOFT Software Engineering Notes* 20 (1995), № 1, pp. 74–78

[WF86] Winograd, T.; Flores, F.: *Understanding Computers and Cognition: a New Foundation for Design*. Ablex, 1986

6 Legal Methodology and Research

Daniel Winteler <daniel.winteler@uni-oldenburg.de>

Abstract

This paper is written for juristic laymen and describes some basic juristic methods when applying law or researching into law. Within the area of juristic methodology it focuses on the construction of law as the most important hand tool of every jurist. Thus, the paper wants to intensify the juristic laymens' comprehension of the daily work of jurists. Another focal point is laid on standard juristic argumentations that are often used to justify a certain result. To explain these topics a lot of examples are given to bring the theoretical comments into live. Finally, some general aims of juristic research are described.

6.1 Introduction

The paper focuses on some of the basic hand tools of jurists when applying law or doing research into law. The main focal points are:

(a) The Technique of Subsumtion

(b) The Construction of Law - Hermeneutics

(c) Standard Juristic Argumentations

(d) Research Aims in Law

(e) Conclusion

The motivation of this paper is to give juristic laymen an idea of what studying law and doing research into law is all about. Every scientific piece of work in law handles with the following items, understanding them is fundamental to understand the work of jurists. A good overview about the essential basics of jurisprudential work is also given by [Sch03]. For other topics of juristic methodology the books [KHN05] and [LC95] shall be recommended.

6.2 The Technique of Subsumtion

The technique of subsumtion means examining whether the facts of a case fit among the concrete requirements of a certain law. Thus, applying law is a lot of doing subsumtions. Subsumtion is carried out in the following way: First you have to split the requirements of a rule in single requirements. Then you define each requirement in an abstract way and examine whether the facts of the case fulfill that abstract definition. If each requirement is fulfilled, the legal consequences can be applied. Thus, subsumtion takes place in 3 steps [Rue05, p. 440]: The abstract definition of the legal requirements composes the first step (premise 1). The decisive parts of the matters of the case compose the second step (premise 2). As a logical consequence the third step shows whether the legal requirements are fulfilled or not (conclusion).

To give a first example:

Section 303 para. 1 German Criminal Code states the following about damaging property[1]: "Whoever unlawfully damages or destroys the property of another shall be punished with imprisonment for not more than two years or a fine." C throws a stone into his neighbours window and smashes it because he is upset about his neighbour´s loud music. That facts of that short case match each requirement of sec. 303 para. 1 Criminal Code:

(a) Whoever=C;

(b) Property of another=the neighbour´s window;

(c) Destroying the property=C smashed the window with the stone;

(d) Unlawfully=He had no right to do that.

[1]The translations within this paper are made by the author. All laws translated and referred to are German laws.

The mentioned three steps (premise 1, premise 2, conclusion) shall not be used here for each legal requirement. But to take the legal requirement “destroying” property as an example:

- Destroying something can be defined as damaging something so substantially that the thing gets absolutely unusual for its purpose (premise 1).
- By smashing the window it can not be used as window any more, it is damaged in a way that it got absolute unusual for its purpose (premise 2).
- Thus, the window was “destroyed” (conclusion).

The fact that C did this because he is upset about the neighbour’s loud music does not affect the subsumtion since it can not be attached to one certain requirement. That does not mean that C’s motives do not matter in any way, they will be considered in the context of the sentence.

The example becomes not as clear if the facts of the case are changed: If C does not smash the window but takes an aerosol can and soils the neighbour´s house, is that a “damaging” or “destroying” of the house when the neighbour just has to clean the walls again and they will be like before? Does it matter how much time it would take the neighbour to clean the walls up again?

This leads to another point: For deciding whether the requirements of a law are fulfilled in doubtful cases, you have to examine the meaning of the requirements, that is to interpret or to construe the law. Thus, the accepted ways of construction shall be briefly introduced in the following.

6.3 The Construction of Law - Hermeneutics

When people hear that someone studies law, a sentence often heard is: “Oh you poor, you have to learn so many laws by heart!” Then it is always quite hard to explain that the problem is not to memorize law since you always keep the law with you (even in the exams) but is actually more that of understanding law. An example often given to clarify that point is sec. 164 para. 2 Civil Code (right of representation): “If the intention to act in the name of another is not apparent, the agent's absence of intention to act in his own name is not taken into consideration." To understand a sentence like that, it is elementary to construe it. Construction of law is therefore finding the sense and application area of rules.

When law shall be interpreted (Grecian: hermeneutics), there are four common ways of construction-methods [Hor04, p. 176]: The grammatical construction, the teleological construction, the systematical construction and the historical construction. They are flanked by additional construction-methods like the constitutional-conform construction and the european-law-conform construction. These methods of construction are part of the basic hand tools jurists need. Although some aspects are even today still in dispute, they are generally accepted as "state-of-the-art."

The need for construction of parts of the law results from the fact that laws are per definition general and abstract. Legislative can not consider each individual case, thus many requirements are not absolutely precise defined. Although many laws are formulated in an "if-then" style that must not be confused with "if-then" relations in computer science. The main difference is that computer scientists can exactly define what "if" and "then" are. Either it is "true" or "false" and if you ask 100 computer scientists they will all give you the same answer (if they calculate correctly) when asking them whether a certain event is "true" or "false."

Main problems occur in law due to the fact that in the first instance "if" is usually not exactly defined in law, perhaps it is never exactly defined. In general legislator can not define requirements so precise since it is not possible to anticipate all supposable developments. So legislator often uses general terms. But the more general terms are, the more interpretations are fungible. Besides, it is an attribute of language in general that it has never a definition everybody would agree to. The meaning of language or of a certain word depends on many circumstances, thus, many questions of law are in dispute among jurists.

To give an easy example:

That thievery, robbery and comparable crimes are punished is obvious. Besides law states that someone who commits such a crime within a gang (German: "Bande") will be punished more severe since in general danger for other people and their property is bigger when several people act together than if one person acts alone. But the term "gang" is not defined elsewhere in law. So the questions arises how many people must act together so that you can speak of a “gang.” Are two enough or are three persons needed? Perhaps even four or five? That question can decide whether a criminal offender is released on licence or is imprisoned for one year. The only obvious wrong interpretation - mostly everybody would agree on—is that a “gang” can be formed by one person alone, all other interpretations mentioned are in principle fungible[2].

The following methods of construction are—in general—always applicable. They do not depend on the question whether the constitution, a “normal” law or a charter of a public corporation is construed. Besides it does not matter whether the law to be construed is from the area of civil, public or criminal law.

6.3.1 Literal Construction

In general an interpretation must be in the wording of a law. It is obvious that in the example just mentioned, one person alone will never form a “gang.” That is because the literal construction of “gang” does not allow that wording. Hence, the literal interpretation is sometimes called “the last line of interpretation.” On the other hand, the German Federal Constitutional Court[3] formulated that when the literal interpretation comes to a clear interpretation, the usage of other methods of interpretation is not allowed[4]. Thus, the literal construction serves primarily two functions: it is the starting point for each interpretation and at the same time restricts the other methods of construction.

The literal interpretation examines the “usual” or “natural” meaning of a word. But from the point of view of whom? That is again disputed. Some say it has to be interpreted from a juristic layman´s point of view and a good argument for that point of view is the following: Law should be constructed in a way that every citizen can understand it. He only can comply with the rules when he knows what law demands of him. This is best warranted when the literal construction of laws is done from a layman´s point of view. On the other hand legislative often

[2]The Grand Senate for Criminal Matters decided that at least three persons can form a gang, ref. BGH 3/22/2001 – file ref.: GSSt 1/00, published in NJW 2001, p. 2266-2270.

[3]Cf. for the methods used by the Federal Constitutional Court when construeing law: [Ble02].

[4]E.g. BVerfGE vol. 19, p. 147; vol. 21, p. 305,; vol. 55, p. 170.

uses technical terms in a way jurists define them. But if the legislator wants certain expressions to be understood in a way jurists define them you would foil his explicit will if you interpreted the requirements from a different point - the juristic layman´s - of view. The German Federal Constitutional Court tends to the first opinion[5].

The literal construction often gets easier if the legislator himself defines certain legal requirements. For example sec. 276 para. 2 of the Civil Code defines negligence like that: "A person acts negligently if he fails to observe the relevant accepted standards of care." If law defines certain term itself that has to be accepted and that definition has to be applied. But that example also shows that the definition of a certain term by law itself often is only of little assistance. The logically next question would be: "But in what case does someone not observe the relevant accepted standards of care?" and: "What are accepted standards of care, who defines them?" But nevertheless it has to be stated that as far law defines a certain term that definition has to be applied in every case.

Other examples of juristic definitions are:

- Sec. 13 Civil Code defines a consumer as "any natural person that concludes a legal transaction for a purpose that can neither be assigned to its commercial nor to its self-employed type of work."
- Sec. 90 Civil Code defines a thing (German "Sache") as "physical items only."
- Sec. 121 para. 1 defines promptly (German "unverzüglich") as "without culpably delay."

But those definitions are often only applicable to the law in which the definition can be found and sometimes a certain term is defined in different ways in different laws. An example for that will be given below.

6.3.2 Systematic Construction

The systematic construction examines the function of a specific rule of law in the total system of law. This way of interpretation handles with logic and tries to avoid discrepancies in valuation. The German Federal

[5]E.g. BVerfGE vol. 73, p. 235; vol. 85, p. 73; vol. 92, p. 20.

Constitutional Court outlines the systematic interpretation as follows[6]: "Within the systematic construction it has to applied that single rules that were put in an objective context have to be interpreted in a way that they are logically comformable. Since it has to be assumed that the whole regulation has a continous, obliging sense."

An example:

Since Jan. 1, 2002, prostitution is not regarded as immoral (in the sense of law) any more: Treaties prostitutes place with their suitors are effective. On the other hand sec. 4 para. 1 Licencing Act states that a licence to operate a restaurant shall be refused if the applicant is unreliable. The Licencing Act concretises that a person is for example then unreliable if he promotes immorality. Both regulations mentioned can only be regarded as continous and obliging if a publican who employs or tolerates prostitutes in his rooms can not be regarded as unreliable. Only this way of interpretation of the word "unreliable" makes sure that the unity of the legal system is assured [Cas02].

One aspect of the systematic construction is for example that it can be assumed that the same word has the same meaning in different rules. But that must not be so. For example the word "nighttime" appears in different contexts (example taken from [Rue05]). Sec. 292 para. 2 Criminal Code states that poaching in nighttime is to be penalized more seriously than poaching at daytime. Judgement defines nighttime as the time from dusk untill dawn. In sec. 19 para. 4 Federal Game Law there is a ban on hunting regulated for nighttime. But nighttime is defined in the same paragraph as the time one and a half hour after dusk to one and a half hour before dawn. In sec. 104 Code of Criminal Procedure the requirements of a house search during nighttime are defined. Here again law itself defines nighttime, but this time depending on the date of the house search: From April, 1 to September, 30 nighttime is from 9pm to 4am in the time of October, 1 to March, 31 nighttime is from 9pm to 6am. So the definition of the word "nighttime" depends on the context it is used in.

Besides it can for example be referred to the title of a rule or a section in which the rule is settled.

Part of the systematic construction are the general rules that the newer regulation replaces the older (Latin: "lex posterior derogat legi priori") and that the specific regulation replaces the more general one (Latin: "Lex specialis derogat legi generali").

[6]Ref. BVerfGE vol. 48, p. 257.

6.3.3 Teleological Construction

The teleological construction is generally appreciated the most important construction method, even though that is often critizised (e.g. by [Her05]). It has to be examined what the spirit and purpose of the law is. Deciding is not the subjective will of the legislator but the purpose that the law can reasonably have. But nevertheless it is often reverted to the official reasons of a law in order to determine the spirit and the purpose of a law, although that seems more to be the following, i. e. the historical construction method.

Again an example:

The Law on Political Parties stated that a donation to a party above 10.000 Euro has to be published. The donator now does not contribute 45.000 Euro at once but gives 5 persons of his trust each 9.000 Euro, advices them to donate the money and these persons contribute in their name. Now the formal contributor are the friends, but is should be clear that the spirit and the purpose of the law wants that kind of arrangement also to be put under its scope.

6.3.4 Historical Construction

The historical construction is fed by two sources: The history of origins and the recourse to its previous history. The historical construction is generally regarded to be the weakest construction method. It is often only used to fortify a conclusion already found by other methods of construction.

The historical construction by referring to the history of origins means recoursing to the opinion of the different legislative organs and their members. It is often referred to the official reasons for a law.

The recourse on the previous history of a law is done by examining the development of a law through the years. When formulations have been retained unchanged in a succeeding law, it can be assumed that the legislative did not want to change the existing prevailing view - otherwise the legislator would have changed the law.

An interesting example is the following:

Sec. 69 Criminal Code states the following:

> “If someone is convicted of an unlawful act which he committed in connection with the driving of a motor vehicle or in violation of the duties of a driver of a motor vehicle, or is not convicted only because his lack of capacity to be

> adjudged guilty has been proved or may not be excluded, then the court shall withdraw his permission to drive if the act reveals that he is unfit to drive a motor vehicle."

In dispute was for a long time how to interpret "if the act preveals that he is unfit to drive a motor vehicle". Obvious someone is unfit to drive a motor vehicle when he often drives drunken or often causes accidents driving in gross violation of traffic regulations or recklessly. But is someone unfit in the mentioned sense if he is convicted because of robbery and he only used the car to get to the site of crime and away again, driving the car in accordance with the relevant road traffic regulations? What about someone who sneaks out of an restaurant without paying and then drives away with a car (again in accordance with the relevant road traffic regulations)?

The law that introduced sec. 69 into the Criminal Code in the year 1952 had as official reasons the abatement of traffic accidents. The aim of the law was - according to the official reasons - to take the actions that are necessary to improve road safety and to reduce the number of road accidents. So sec. 69 has to be interpreted in the way that the unlawful act that reveals the unfitness of the driver must be an act that shows that he endangers the road saftey, that the general public has to fear that he will commit road accidents[7]. But that is not the case as long as the offender complies with traffic regulations. Thus, recoursing to the opinion of the legislator what the purpose of a law is helps interpretating laws.

6.3.5 Additional Construction

The constitutional-conform construction is an effect of the hierarchy of Law. Each federal law has to comply with the Constitution, otherwise it is not effective. To avoid that ineffectiveness in certain cases, that construction should be applied that complies with the Constitution instead of other ones that do not conform with it. The reason for that way of construction may be the principle of separation of powers [Lue04]. A part of that principle can be seen as the respect the judiciary must have for the legislative. That respect demands to maintain a law effective as long as possible.

[7]Ref. BGH 8/26/2004 - file ref.: 4 StR 85/03, published in NJW 2004, p. 3497-3504.

The constitutional-conform construction is also often referred to as a part of the systematic construction.

The object lesson students are taught to explain the historical construction is the following:

Sec. 14 of the law concerning assemblies and processions states that a “planned” public open-air assembly has to be registered 48 hours before its start. If the assembly is not registered in time or is not registered at all, it can be prohibited by the public authorities (sec. 15). But what happens if there is an unforeseen event, like the begin of a war or the death of a prominent person and people come together spontaneously and without appointment. How should that assembly have been registered 48 hours before? So the question arises whether a spontaneous assembly that reacts on unforeseen events also can be prohibited due to the lack of registration, that is whether sec. 14 of the law must also be applied to such assemblies. With respect to sec. 8 Constitutional Law, which guarantees the freedom of assemblies, a duty for registration has to be denied in such cases. Another construction would not be conform with the constitution, the result would be that sec. 14 is ineffective in whole. The wording of sec. 14 of the law concerning assemblies and processions is therefore construed that it only is applicable if the assembly is pre-”planned” a certain time before the event.

Since Germany is part of the European Union and has to comply with the European Law there is also the rule that national law has to be consistent with European Law. How far that rule has to be applied, that is whether really every law in Germany, especially the whole German Constitution has to conform with all European law is strongly disputed even between the German Federal Court and the European Court of Justice. That problem shall not be deepened here. But in the scope of the European law conform construction, national law has to be understood so that it is conform to European Law as far as the literal construction allows that.

6.3.6 Example: Pistol for Blanks as Weapon

Here is an example of a judgement of the Federal Court of Justice[8] in which most of the presented construction methods are applied [Wan05, p. 108-110]:

[8]Ref. BGH 2/4/2004, file ref.: GSSt 2/02, published in NJW 2003, p. 1677-1679.

In the following the circumstances of the case: A gets into a bank, aims with a pistol for blanks (German: "Schreckschusspistole") at the bank assistant B from a distance of 2 meters and shouts: "Give me the money or I will shoot you." B gives him some thousand euros and A flees.

It is clear that A has committed a robbery. But a robbery is even stronger punished if the robber acts in a certain way. Sec. 250 para. 1 Criminal Code states the following: "Imprisonment for not less than three years shall be imposed, if: 1. the perpetrator or another participant in the robbery: a). carries a weapon or other dangerous tool (. . .)." The question now arises, whether even a pistol for blanks is a weapon as mentioned in sec. 250 para. 1 lit. a Criminal Code. In the following the construction of that rule as carried out by the Federal Court of Justice is outlined.

Literal Construction

The literal construction of a "weapon" is not clear. In any case it allows the interpretation that a pistol for blanks can be understood as a weapon.

Systematic Construction

Sec. 1 firearms act defines the pistol for blanks as a weapon in the sense of that act. As mentioned above from that fact it can be assumed that this definition can be also used within the Criminal Code. But that is not mandatory since sec. 1 firearms act defines the word "weapon" only for that act and not for others like the Criminal Code.

Historical Construction

According to the official reasons of the 6th penal law amendment that introduced sec. 250 para. 1 Criminal Code a pistol for blanks shall not be defined as a weapon in the sense of that section.

Teleological Construction

The reason for the aggreviation of sentence by sec. 250 para. 1 lit a Criminal Code is that in these cases the victim´s health is endangered by the usage of certain means. Actual research findings of the forensic science show that even a pistol for blanks is suitable to evoke serious injuries. When applied at the head, eyes or throat a shot can also have lethal effect.

Mainly referring to the systematic and the teleological construction, the Federal Court of Justice ruled that also a pistol for blanks has to be considered a “weapon” in terms of sec. 250 para. 1 lit. a Criminal Code.

6.3.7 The Construction of Treaties

It should be underlined that the methods of construction mentioned above are only applied when interpretating law “made” by government. Legal transactions between persons also have to be construed but that has little to do with the methods applied when construing law. The construction of treaties follows other rules that should not be explained here.

6.4 Standard Juristic Argumentations

Some standard forms of argumentation are generally accepted today. One or more of them generally can be found in every detailed juristic script. Thus, the analogy and the reverse argumentation, the teleological reduction and the argumentation a fortiori are also part of the jurist´s basic methods. But it shall also be mentioned that even these methods of juristic argumentation are in dispute, too. A theoretical background to juristic argumentation gives [KHN05, p. 333-346].

6.4.1 Analogy and Reverse Argumentation

Analogy

As already mentioned, the legislator can not predict all eventual cases, so he tries to formulate laws as general as necessary and as exact as possible. But sometimes he formulates a law too specific or forgets to regulate some case at all. Then it is possible - under certain requirements - to transfer the legal consequence of a regulated case on that not regulated case. But that is only possible, if there is a not planned legal loophole and the interests are comparable. Thus, in general it has to be asked whether the legislator would have put that case under the same legal consequences if he had seen that case. If that can be approved jurists talk of an analogy or in Latin “argumentum a simili.”

Here is a demonstration of an analogy:

Sec. 823 para. 1 Civil Code states that whoever willfully or negligently violates the live, the body, the health, the freedom, the property or another right of someone is obliged to make good the damage. Sec. 1004 para 1 sentence 1 Civil Code states that whoever affects or intends to affect the proiperty of another can be filed for injunctive relief. It is obvious that someone must have the possibility to file for injunctive relief if someone intends to hurt him. But sec. 823 para. 1 Civil Code only states that after a violation of the body of someone he has to make good the damage and sec. 1004 para 1 Civil Codes states that only the imminent danger of the violation of property allows to apply for an injunction. With respect to the unity of law sec. 1004 para. 1 Civil Code is applied analogously in all the cases in which sec. 823 para. 1 Civil Code gives the possibility of claiming damages.

The only area in which an analogy to the disadvantage of an offender is not allowed is the area of substantive criminal law. That is because sec. 103 para 2 of the Constitution states: "An act may be punished only if it was defined by a law as a criminal offense before the act was committed."

Reverse Argumentation

The counterpart to the analogy is the reverse argumentation, in Latin: "argumentum e contrario." The regulated case is understood as an exception that can not be generalized. So sometimes it can be concluded from a regulated case that other similar but not equal cases shall not be subsumed. The reverse argumentation is also referred to as "eloquent silence."

An example for the method of reverse argumentation could be the following:

Sec. 1601 para. 1 Civil Code states that directly related relatives have in general a maintenance obligation for each other. When the law was created the legislative knew that there were also other kind of relatives but he only stated that direct related relatives schould have that support obligation. Thus, you can draw the reverse argumentation that collateral relatives shall not have such a support obligation.

Analogy and reverse argumentation do not have a mandatory relationship but exclude each other.

6.4.2 Teleological Reduction

Teleological reduction means not applying a rule to the facts of a case that seem to match the requirements of the law when concidering the literal construction. But the spirit and purpose of the law do not match the facts. Thus, the scope of the rule is too broad and has to be reduced.

An example (ref. [Wra03, p. 987]):

Sec. 306 para. 1 Criminal Code states the following: "Whoever sets fire to or, as a result of setting a fire, destroys in whole or in part: (...) 4. motor vehicles, rail vehicles, aircraft or watercraft (...) 6. agricultural, nutritional or forestry facilities or products, shall be punished with imprisonment from one year to ten years." As the reader may recognize, the minimum sentence is one year for these crimes. But is it really right to bring someone behind bars for one year if he burns and destroys an aged canoe that could hardly swim anymore or if he burns some apples? The reason for the hard punishment of those crimes were not cases like that, so jurists try to construe these objects in a restrictive way. In general they add an "unwritten legal requirement": the property of another of significant value must be endangered by the fire. This requirement can also be found in sec. 315c Criminal Code.

6.4.3 Argumentum A Fortiori

The argumentation a fortiori exists in two versions: the "argumentum a maiore ad minus" and the "argumentum a minore ad maius."

The "argumentum a maiore ad minus" describes an argumentaion from the bigger to the smaller. An example could be the following: If someone has the right to cancel a treaty without respite he may a fortiori cancel the treaty with recipe. The argumentation a minore ad maius takes place the other way round, e. g.: If the citizen gets a compensation in the case of legal expropriation he will get a compensation a fortiori in the case of illegal expropriation.

6.5 Research Aims in Law

Research in law can have different aims. In general three aims can be distinguished. These aims are not strictly separated but can be mixed in different proportions in one scientific work. Of course it always depends on the specific area law research is done in. Research in sociology of law or history of law for instance may have a lot of other aims, then mainly due to the fact that research is done in an interdisciplinary area, so that the research methods of the area also affected have to be integrated.

The research aims are:

(a) a descriptive approach,

(b) a new approach on the basis of existing law,

(c) an approach for a new law.

6.5.1 The Descriptive Approach

Research in law is basically of descriptive nature. Many scientific works “just” represent the state-of-the-art regarding a specific question. This is not as easy as it may sound. A great effort has to be made to understand the arguments delivered by other jurists. These arguments often base on other axioms so they have also to be examined. But these axioms may also be disputed and so one thing may lead to the other.

In general the different positions are discussed and the author of the research work tries to decide who´s arguments are better and why they are. Maybe by new arguments of the researcher himself or by disproving the arguments of the contra-opinion. For the argumentation the techniques and construction methods mentioned above are applied.

6.5.2 The New Approach on the Basis of Existing Law

The core of a jurist´s research work may also be a new approach to a certain question on the basis of existing law. The author of this paper for example examines in his doctoral thesis whether there are new ways to consider the particularities of software within the given system of law. Basis could be for example that the warranty rights within sales law are only directly applicable if “software” is a thing (German: “Sache”). If software is not a thing in the sense of sec. 90 Civil Code (a “physical thing”) but some kind of intangible thing, the rules of sales law are only applicable mutatis mutandis. But an application mutatis mutandis is nothing else than an application in consideration of the specialties of the area the law shall be applied. So the specialties of software in contrast to other objects of purchase have to be examined.

The two kinds of descriptive approaches are the aims mostly researched in. That is because the function of jurisprudence is generally seen in helping the one who applies law (mainly judges and public officials) to find a correct decision; correct not in the sense of “true” since jurisprudence consists of a lot of value judgements which are not provable as “true” or “false,” but correct in the sense of best fungible.

6.5.3 Developing New Laws

More rarely juristic laymen may think the issue of a doctoral thesis in law is to develop new laws. The main focus of such a work will be to highlight the unsactisfactory parts of law and examine which problems occur when applying the law in question. For example the Law of Obligations of the Civil Code was “updated” in large parts in 2002. This was due to several reasons but it was mainly because this part of the Civil Code was over 100 years old and therefore had some sections that had become obsolete and a lot of judge-made law had to be included.

Such an advancement of law is quite rare as jurisprudence has not the power to make laws. The predominant function of jurisprudence is examing that what is not that what could be. As already mentioned: The classical challenge of jurisprudence is supporting the judge in his decision-making.

6.6 Conclusion

The reader of this paper should now have an idea of what jurists do when handling with law, in particular when interpreting it. Hopefully he parted with the misconception that studying law is about memorizing section by section and paragraph by paragraph of laws. Being introduced into the technique of subsumtion and the methods of construction helps the reader to understand juristic papers or judgements. Perhaps this knowledge motivates juristic laymen to read some other juristic contributions.

Bibliography

[Ble02] Bleckmann, A.: Zu den Methoden der Gesetzesauslegung in der Rechtsprechung des BVerfG. In: *Juristische Schulung*

(2002), pp. 942–947

[Cas02] Caspar, J.: Prostitution im Gaststättengewerbe? Zur Auslegung des Begriffs der Unsittlichkeit im Gaststättengesetz. In: *Neue Zeitschrift für Verwaltungsrecht* (2002), pp. 1322–1328

[Her05] Herzberg, R. D.: Die ratio legis als Schlüssel zum Gesetzesverständnis? - Eine Skizze und Kritik der überkommenen Auslegungsmethodik. In: *Juristische Schulung* (2005), pp. 1–8

[Hor04] Horn, N.: *Einführung in die Rechtswissenschaft und Rechtsphilosophie.* Heidelberg, Germany: C. F. Müller Verlag, 2004

[KHN05] Kaufmann, A.; Hassemer, W.; Neumann, U.: *Einführung in die Rechtsphilosophie und Rechtstheorie der Gegenwart.* Heidelberg, Germany: C. F. Müller Verlag, 2005

[LC95] Larenz, K.; Canaris, C.-W.: *Methodenlehre der Rechtswissenschaft.* Berlin, Germany: Springer Verlag, 1995

[Lue04] Luedemann, J.: Die verfassungskonforme Auslegung von Gesetzen. In: *Juristische Schulung* (2004), pp. 27–30

[Rue05] Ruethers, B.: *Rechtstheorie.* Munich, Germany: Verlag C. H. Beck, 2005

[Sch03] Schmidt, T. I.: Grundlagen rechtswissenschaftlichen Arbeitens. In: *Juristische Schulung* (2003), pp. 551–556 and 649–654

[Wan05] Wank, R.: *Die Auslegung von Gesetzen.* Cologne, Germany: Carl Heymanns Verlag, 2005

[Wra03] Wrage, N.: Typische Probleme einer Brandstiftungsklausur. In: *Juristische Schulung* (2003), pp. 985–991