Empirical Software Engineering, 6, 161–180, 2001.
© 2001 Kluwer Academic Publishers. Manufactured in The Netherlands.

A Preliminary Software Engineering Theory as Investigated by Published Experiments

ANDREAS ZENDLER zna@mgm-edv.de
University of Potsdam, Institute of Computer Sciences, 141415 Potsdam, Am Neuen Palais 10, Germany

Abstract. The investigation of software engineering techniques by using the experiment sets up the discipline of experimental software engineering. The effectiveness and efficiency of software engineering techniques have been studied in experiments and confronted with empirical data. The results of these experiments are published in a diversity of journals and proceeding papers. However, there is not an overview so far which represents the available results systematically. By taking the results from published experiments that deal with analysis, design, implementation, test, maintenance, quality assurance, and reuse techniques a preliminary software engineering theory is developed. From this theory, fruitful problems, suggestions for gathering new data, and entirely new lines of investigation are deduced.

Keywords: Experimental software engineering, theory of software engineering.

1. Introduction

Software engineering is concerned with techniques to develop, run, and maintain application systems. Empirical software engineering is the subdiscipline of software engineering that investigate software engineering techniques by using empirical methods: case study, inquiry, and experiment. Empirical software engineering has been pushed by internationally recognized research institutes such as Software Engineering Institute (SEI), Empirical Foundations of Computer Science (EFoCS), Empirical Software Engineering Laboratory (ESEL), Institute of Experimental Software Engineering (IESE), Centre for Advanced Empirical Software Research (CAESAR), Empirical Informatics Research (EIR), and NASA's Software Engineering Laboratory (SEL). An important role takes International Software Engineering Research Network (ISERN), which is a union of more than 20 research institutes.

Experimental software engineering is the subdiscipline of empirical software engineering that uses the experiment to validate, improve, and select software engineering techniques which permit an efficient application development. Experiments have been conducted to investigate analysis, design, implementation, testing, maintenance, quality assurance, and reuse techniques. These experiments have been published in different journals and proceeding papers. A comprehensive overview that presents the results of the software engineering experiments and results is missing. Moreover, a systematic context is not available that guides the selection of what to experiment next.

The purpose of this paper is to develop a systematic context—a preliminary software engineering theory—for the results of software engineering experiments that allow to ask for new software engineering problems, to collect further data to support software engineering hypotheses, and to deduce new software engineering investigation lines.

The rest of this paper is organized as follows: Section 2 introduces the necessary concepts—theory and hypotheses—to build a systematic context for the published software engineering experiments. Section 3 gives a brief history of experimental software engineering. Section 4 contains results from published software engineering experiments to build the preliminary software engineering theory. Section 5 presents the preliminary software engineering theory; Section 6 shows how it can be used. Section 7 contains conclusions.

2. Software Engineering Theory and Hypotheses

A theory can be understood as a system of hypotheses, as a system of assumptions to explain propositions of the real world: "A scientific theory is a system of hypotheses that is supposed to give a partial and approximate account of a bit of reality. At this point we shall emphasize that a theory is a system" (Bunge, 1967, p. 391—emphasis in the original). Bunge stresses that a theory is a system, i.e.:

1. it refers to a universe of discourse,
2. it consists of several hypotheses,
3. the hypotheses refer to the universe of discourse,
4. the hypotheses are connected,
5. no hypothesis is isolated.

2.1. Software Engineering Theory

By transferring the general statements made by Bunge to software engineering, a software engineering theory can be understood as a theory,

1. whose universe of discourse is software engineering,
2. whose hypotheses are software engineering hypotheses,
3. whose software engineering hypotheses refer to the universe of discourse that is software engineering,
4. whose software engineering hypotheses are connected,
5. none of the software engineering hypotheses are isolated.

2.2. Software Engineering Hypotheses

Software engineering hypotheses can be understood as assumptions to explain software engineering propositions. With the formulation of a software engineering hypothesis the intention is to explain a software engineering phenomenon for which so much information is available that the explanation withstands even critical objections. Software engineering hypotheses can be characterized by four characteristics: (1) they are not to be equated with fiction; (2) software engineering hypotheses have a greater content than the empirical propositions they cover: "A (software engineering) datum is not a hypothesis: every hypothesis goes beyond the [· · ·] data it purports to account for" (Bunge, 1967, p. 223); (3) by not referring to singular experiences, software engineering hypotheses cannot be established by any single experience; on the other hand, single software engineering phenomena can refute a software engineering hypothesis; (4) software engineering hypotheses are corrigible based on new experiences.

2.3. Formulation of Software Engineering Hypotheses

Three main requests for the formulation of software engineering hypotheses are imposed: (1) software engineering hypotheses must be syntactically correct and meaningful (semantically not empty); (2) software engineering hypotheses must be grounded to some extent on previous knowledge; entirely new software engineering hypotheses should be compatible at least with most of the available knowledge; (3) software engineering hypotheses must be empirically testable by the objective procedures of science (experimental methods), i.e., by confrontation with empirical data.

Examples of software engineering hypotheses (h) are:

- "object-oriented programming techniques have advantages against structured programming techniques";
- "so-called *Event-Controlled Process Chains* are better suitable for requirement analysis than so-called *Use Cases*";
- "faceted classification is superior to enumerated classification when retrieving reusable software components".

2.4. Types of Software Engineering Hypotheses

Software engineering hypotheses (h) can be divided according to whether they are based on available antecedent knowledge (A) and whether they are confronted with empirical data (e). We can distinguish between the following types of software engineering hypotheses (see Figure 1): (1) software engineering guesses, (2) empirical software engineering hypotheses, (3) plausible software engineering hypotheses, and (4) corroborated software engineering hypotheses.

2.4.1. Software Engineering Guesses

Software engineering guesses are hypotheses (h), which are based neither on background knowledge (A) nor on empirical tests (e). The predominance of guesses characterizes speculation and the early stage of theoretical work.

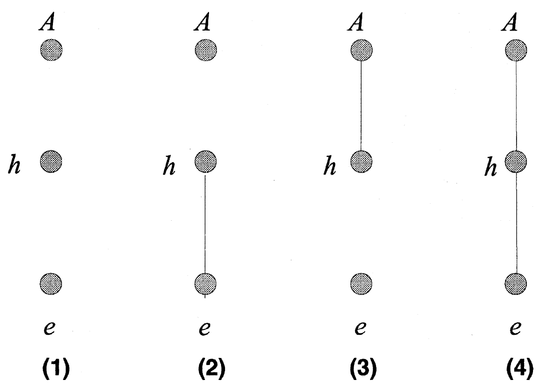

Figure 1. Characterization of software engineering hypotheses.

2.4.2. Empirical Software Engineering Hypotheses

Empirical software engineering hypotheses are hypotheses (*h*), which are not based on software engineering antecedent knowledge (*A*), but are empirically (*e*) tested. An empirical software engineering hypothesis is a rather isolated conjecture with no support other than the ambiguous one offered by the facts it covers: it lacks theoretical validation.

2.4.3. Plausible Software Engineering Hypotheses

Plausible software engineering hypotheses are hypotheses (*h*), which are based on software engineering antecedent knowledge (*A*), but are not empirically (*e*) tested: plausible software engineering hypotheses do not have any empirical justification.

2.4.4. Corroborated Software Engineering Hypotheses

Corroborated software engineering hypotheses are hypotheses (*h*), which are based on software engineering antecedent knowledge (*A*) and are empirically tested (*e*). The predominance of such hypotheses would characterize theoretical knowledge and is the mark of a mature science.

3. History of Experimental Software Engineering

Experimental software engineering goes back to the year 1967. Figure 2 illustrates that in the seventies and still in the eighties only few software engineering experiments were published. This situation changes in the nineties, in which many experimental investigations are published. The reasons for the enormous rise of the

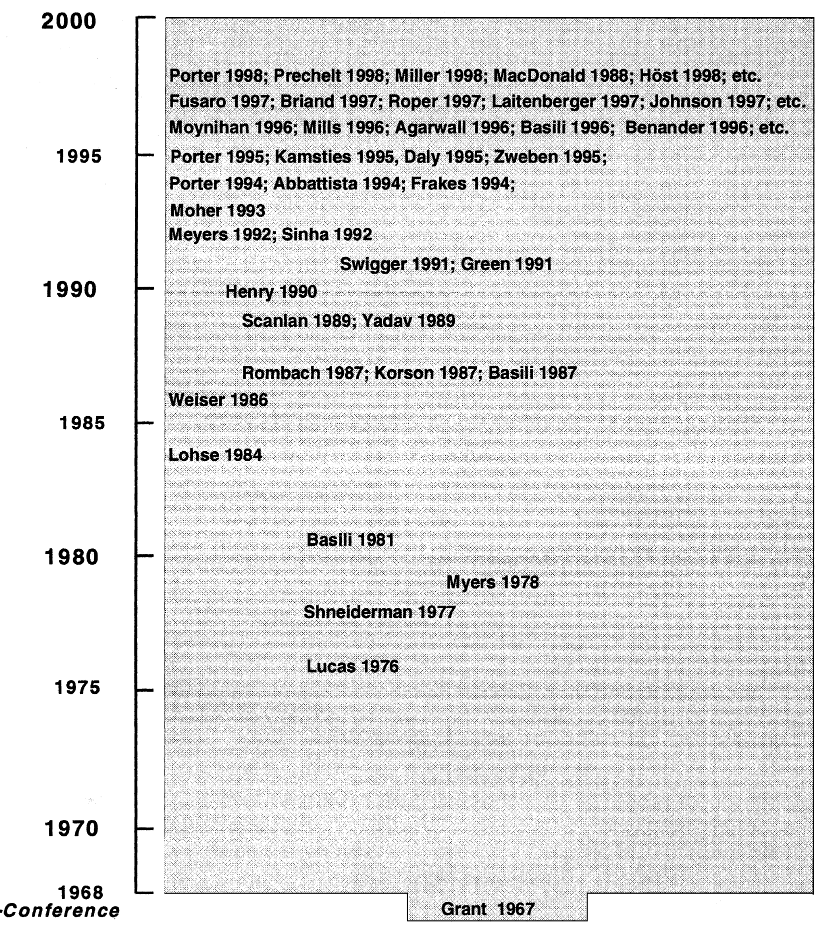

Figure 2. Software engineering experiments.[1]

publication rate in the nineties are due to the facts that (1) the necessity for experimental research in the discipline of software engineering was realized, (2) the availability of methodological papers to conduct software engineering experiments, and (3) research institutes have been founded whose research programs are empirically oriented.

3.1. First Experiment 1967

The first investigation that treats a software engineering problem experimentally is the experiment published by Grant and Sackman (1967). They evaluate—in an experiment that is interesting from a methodological point of view even today—the efficiency of programmers when testing under on- and off-line conditions.

3.2. Experiments 1970–1979

Software engineering experiments of the seventies strongly concentrate on software engineering problems that deal with implementation and testing techniques. Lucas and Kaplan (1976) investigate the advantages of structured programming, Shneiderman et al. (1977) analyze the use of flowcharting. Myers (1978) conducts an experiment that is concerned with software testing techniques.

3.3. Experiments 1980–1989

In the eighties, experiments are conducted that deal with analysis, design, implementation, test and maintenance techniques. Yadav et al. (1989) evaluate analysis techniques while Korson and Vaishnavi (1987), Lohse and Zweben (1984) and Scanlan (1989) investigate design techniques. Basili and Reiter (1981) and Rombach (1987) experiment with implementation techniques. Myers (1978) and Weiser and Lyle (1986) study test techniques. The experiment of Basili and Selby (1987) that evaluates three test techniques (code reading, functional testing and structural testing) represents a highlight of experimental software engineering in the eighties. It is considered as a very important reference on how to conduct experiments that study test techniques.

3.4. Experiments 1990–1999

The nineties are characterized by a multiplicity of laboratory and field experiments. Analysis, design, implementation, test and maintenance as well as quality assurance and reuse techniques have been studied experimentally during the last years.

Moynihan (1996) analyzes the appropriateness of object-oriented and structured techniques. Experimental investigations for the evaluation of particular design techniques are available from Agarwal et al. (1996), Mills (1996) as well as Briand et al. (1997). Experiments on implementation techniques are published by Benander et al. (1996), Cartwright (1998), Daly et al. (1996), Green et al. (1991), Henry et al. (1990), Kieburtz et al. (1996), Kiper et al. (1997), Meyers and Reiss (1992), Moher et al. (1993), Prechelt (1997), Prechelt and Tichy (1996, 1998) and Sinha and Vessey (1992). Experiments that deal with test techniques are available from Kamsties and Lott (1995) and Roper et al. (1997). Abbattista et al. (1994), Swigger and Brazile (1991) and Tryggeseth (1997) conducted experiments that analyze maintenance techniques.

Experimental studies with quality assurance are available from: Basili et al. (1996), Fusaro et al. (1997), Johnson and Tjahjono (1998), Laitenberger and DeBaud (1997), MacDonald and Miller (1998), Miller et al. (1998), Porter et al. (1995, 1998) and Porter and Votta (1994). In particular, the software engineering experiments of Porter and Votta (1994) and Porter et al. (1995, 1998) should be emphasized. Their results are interesting as regards where and when quality assurance seems to be done:

"Anywhere, anytime code inspections: using the Web to remove inspection bottlenecks in large-scale software development" (Perpich et al., 1997, p. 14). Moreover, experiments are available that deal with reuse techniques: Frakes and Pole (1994) evaluate representation techniques for retrieving software components; Zweben et al. (1995) analyze the effects of black box and white box reuse on software quality and cost.

4. Published Software Engineering Experiments

To develop a software engineering theory published up-to-date experiments are considered that respect the following three criteria:

1. at least two software engineering techniques are studied,
2. subjects (software developers) apply software engineering techniques and produce artifacts,
3. the artifacts are measured by software metrics.

The criteria stress: (1) the object of software engineering experiments is the comparative investigation of techniques; (2) by using subjects software engineering experiments are delimited against computer-based experiments such as compiler tests, benchmark tests, etc; (3) by emphasizing software metrics software engineering experiments are delimited against experiments with psychological variables.

Some experiments that have been referenced in the above section will not be considered to build the preliminary software engineering theory, because their results are obsolete to modern software engineering (e.g. Grant and Sackman (1967)). To describe the experiments, a pattern should have been offered that considers the essential parts of an experiment: goal, experimental design, results and conclusions. Due to lack of space the experiments could not be described according to this pattern; nevertheless, detailed representations of all experiments can be found in Zendler (1999) based on the so-called *goal template* (Basili et al., 1986; Lott and Rombach, 1996).

4.1. Experiments on Analysis Techniques

Classical analysis techniques are structured analysis and design technique (SADT), structured analysis, data flow diagrams, problem statement language/problem statement analyzer (PSL/PSA). Enhancements of the classical techniques are process oriented, applicative, interpretable specification language (PAISLey), and requirement modeling language (RML). Object-oriented analysis techniques are general object-oriented design (GOOD), hierarchical object-oriented design (HOOD), responsibility driven design (RDD), object-oriented Jackson structured design (OOJSD), object-oriented design language (OODLE). For developing the software engineering theory the experiments by Yadav et al. (1989) and Moynihan (1996) will be used.

Yadav et al. (1989) compare two analysis techniques, the data flow diagram (DFD) and SADT; they are interested in syntactic and semantic criteria: "Criteria that evaluate syntactical rules for the modeling process as well as for the final outcome of a technique (· · ·) the semantic dimension provides criteria to evaluate the meaning of the modeling process as well as the meaning of the final outcome" (Yadav et al., 1989, p. 1092). Besides, they compare the two analysis techniques with respect to the communication ability and the usability in future projects. The main results are: (1) with respect to the communication ability the two analysis techniques differ significantly. DFD are superior to SADT regarding the learning process of syntactic rules and the ease of drawing diagrams; (2) concerning the usability in future projects DFD is also advantageous.

Moynihan (1996) studies the effects of the object-oriented and functional paradigm on the client/developer communication in the early stages of the system development process. The object-oriented paradigm is applied according to object modeling technique (OMT) (Rumbaugh et al., 1991), the functional paradigm according to Martin (1992). The main results are: (1) regarding "subject's task performance" they do not differ; (2) "overall (· · ·) compared with the object model, the functional decomposition was easier to understand, provoked them to ask more questions and to make more comments; gave them a more holistic understanding of the business; better helped them to evaluate likely implementation benefits and priorities" (Moynihan, 1996, p. 167).

4.2. Experiments on Design Techniques

Classical design techniques are structured design (SD) and SADT. Classical design methods (with a process model that controls the application of several techniques) are Merise (Tardieu et al., 1989) and structured systems analysis and design method (SSADM) (Ashworth and Slater, 1993). An object-oriented design technique is object-oriented system analysis (OSA); unified modeling language (UML) as a notation can be used to deduce object-oriented techniques. Object-oriented design methods are OMT, object-oriented software engineering (OOSE), methodology for object-oriented software engineering of system (MOSES), and OPEN modeling language (OML). For developing the software engineering theory the experiments by Agarwal et al. (1996), Briand et al. (1997), Korson and Vaishnavi (1987), Lohse and Zweben (1984), Mills (1996) and Scanlan (1989) are considered.

Lohse and Zweben (1984) evaluate the effect of module coupling on system modifiability. The results of Lohse and Zweben can be summarized as follows: (1) module coupling (coupling with global variables versus coupling with parameter transfer) do not have any effect on software modifiability; (2) however, there was an interaction effect between module coupling and the type of modification.

Korson and Vaishnavi (1987) investigate the effects of the modularity of Pascal on adaptive program maintenance. The main result is that modular software can be modified more efficiently than monolithic software.

Scanlan (1989) analyzes the effects of specification techniques on the understandability of algorithms. The main results are: (1) when using flowchart in comparison with pseudocode specifications more correct responses are obtained for the understanding of the algorithms; (2) if flowchart specifications are given, the time to understand algorithms is shorter than with pseudocode specifications; (3) flowchart specifications are particularly more efficient than pseudocode specifications when the complexity of the algorithms rises.

Agarwal et al. (1996) investigate the role of prior experience and task characteristics in object-oriented modeling. The experimental results show: (1) the quality of object-oriented design documents of system analysts with prior experience in process oriented analysis is significantly better than the quality of the documents of system analysts of the control group; (2) the type of task has significantly more effect on the quality of process oriented and of object-oriented documents when developed by system analysts with prior experience in contrast to the control group.

Mills (1996) studies the effects of functional testing given different specification techniques (natural language, graphical real-time structured analysis, executable specification). The results show that no significant difference in performance was found neither in the degree of statement coverage nor in the development time.

Briand et al. (1997) compare object-oriented and structured design techniques with respect to the maintenance of design documents. The main results are: (1) maintainers with little experience gain not a great benefit maintaining object-oriented designs over structured designs; (2) "... adherence to 'good' object-oriented design principles will provide ease of understanding and modification for the resulting design when compared to an object-oriented design to which to the principles have not been adhered to"; (3) "an object-oriented design which did not adhere to quality design principles is likely to cause more understanding and modification difficulties than an appropriate structured design" (Briand et al., 1997, p. 308).

4.3. Experiments on Implementation Techniques

Implementation techniques can be divided according to whether software is either generated or implemented. Moreover, it is distinguished between fine-granular techniques (e.g. iterative, recursive programming), techniques for support (e.g. type checking), visual programming, etc. For developing the software engineering theory the experiments by Basili and Reiter (1981), Benander et al. (1996), Cartwright (1998), Daly et al. (1996), Henry et al. (1990), Kieburtz et al. (1996), Kiper et al. (1997), Lucas and Kaplan (1976), Prechelt (1997), Prechelt and Tichy (1998) and Rombach (1987) have been selected.

Lucas and Kaplan (1976) study the effects of structured programming. The main result is: when structured programming is used the time spent modifying the program, program compile time and program storage requirements are significantly lower than those for the control group.

Basili and Reiter (1981) compare the effects of a so-called disciplined method on different software metrics such as efficiency, quality, costs: "the disciplined methodology consisted of programming teams employing certain techniques and organizations commonly defined more under the umbrella term structured programming" (Basili and Reiter, 1981, p. 299). The results of Basili and Reiter can be summarized as follows: (1) with "structured programming" software can be developed significantly more efficiently than with ad hoc programming; (2) with "structured programming" software can be produced significantly more economically than with ad hoc programming.

Rombach (1987) analyzes the effects of structuring concepts in the computer language LAnguage for Distributed sYstems (LADY) on the maintenance of programs. LADY offers three so-called refinement levels (units, processes, modules) and two communication concepts (messages, common memory) to structure distributed systems. The main results are: (1) the maintenance of LADY programs in comparison with C-TIP programs (the control group) is easier if it concerns large programs; (2) the comprehensibility of LADY programs is better than of C-TIP programs; (3) error localizing in LADY programs is better than in C-TIP programs; (4) with respect to reusability LADY programs are superior to C-TIP programs.

Henry et al. (1990) compare the effect of the programming paradigm on the maintenance of programs. To study the programming paradigm the object-oriented programming language objective C and the procedural programming language C are used. The investigation shows that the program code which was prepared by objective C is significantly more maintainable than the code which was developed with C.

Daly et al. (1996) study the effects of inheritance in object-oriented programs on maintainability. The main results are: (1) maintenance costs for programs with inheritance depth of 3 are lower than costs for programs without inheritance; (2) in contrast, subjects maintaining object-oriented programs with inheritance depth of 5 took longer than the subjects maintaining the equivalent object-based programs (no inheritance).

Benander et al. (1996) compare the effects of recursive and iterative programming constructs on the understandability of Pascal source code. The main results are: (1) Pascal source code with recursive constructs is significantly more correctly understood than Pascal source code with iterative constructs, if the type of task is sorting; (2) it is shown that there is a tendency towards an interaction effect between programming constructs and type of task.

Prechelt (1997) studies the effects of the documentation with design patterns on the maintenance of object-oriented programs. The main results are: (1) maintenance tasks are done faster with pattern documentation than without pattern documentation; (2) in programs with pattern documentation the maintenance tasks are done with fewer errors than in programs without pattern documentation.

4.4. Experiments on Testing Techniques

Testing techniques cover black and white box test, structural tests, functional tests, regression tests. Moreover, there are code reading, walkthroughs and error detection techniques that belong to this category (Perry, 1995). For developing the software engineering theory the experiment by Basili and Selby (1987), Kamsties and Lott (1995) and Roper et al. (1997) will be used.

Basili and Selby (1987) study three different software test techniques (code reading by stepwise abstraction, functional testing, structural testing). The work of Basili and Selby is considered as a very important reference to the experimental investigation of test techniques. The main results are: (1) advanced reviewer obtain best results with the code reading technique; (2) intermediate reviewer obtain similar results with the code reading technique and the technique of functional testing; the results have been better than those with the technique of structural testing; (3) error detection depends on the software which is tested; (4) especially when using code reading interface errors are found more easily; (5) using the technique of functional testing more so-called control faults are found than by the two other techniques.

Kamsties and Lott (1995) study three defect detection techniques (code reading by stepwise abstraction, functional testing, structural testing) with respect to error location and cost-efficiency. The experiment is partly a replication of the experiment of Basili and Selby (1987). The experiment was conducted as a laboratory experiment at the University of Kaiserslautern. The main results are: the defect detection techniques do not differ significantly according to the number of found errors. However, it is shown that functional testing is superior to the other techniques, if one considers the time necessary for error location. The results of Kamsties and Lott support the results of Basili and Selby (1987). "The results of our experiments suggest that under the condition of testers who are inexperienced with both the implementation language and these three techniques, if time is not a factor, any of the techniques may be used effectively to detect defects" (Kamsties and Lott, 1995, p. 382).

The experiment of Roper et al. (1997) is a replication of the experiment of Kamsties and Lott (1995) and supports their results.

4.5. Experiments on Maintenance Techniques

Maintenance techniques allow the modification and enhancement of software. Basili (1990) distinguishes three techniques: quick fix, iterative enhancement and full reuse. Chikofsky and Cross (1990) add reengineering and reverse engineering techniques, such as redocumentation and design recovery. To develop the software engineering theory the experiment by Swigger and Brazile (1991) will be considered.

Swigger and Brazile (1991) study the effect of documentation techniques on the maintainability of an expert system. The results of Swigger and Brazile can be summarized as follows: the type of documentation (Petri net, entity relationship diagram) has an effect on the maintainability of the expert system: when documenting the software using Petri nets, data-oriented and procedural modifications can be processed significantly faster.

4.6. Experiments on Quality Assurance Techniques

Quality assurance techniques are tools to avoid errors in different software development stages, especially inspection techniques, walkthroughs, and review techniques. For developing the software engineering theory the experiments by Basili et al. (1996), Laitenberger and DeBaud (1997), MacDonald and Miller (1998), Miller et al. (1998), Porter and Votta (1994) and Porter et al. (1995) will be used.

Porter and Votta (1994) study the effects of inspection techniques for defect detection in requirements specifications. The main results are: (1) the inspection techniques differ with respect to the team defect detection rate significantly; (2) the best results are obtained for the scenario technique, the worst for the check list technique (which is industry standard); (3) the type of requirements specification has an effect on the team defection rate.

The experiment of Porter et al. (1995) is a replication of the experiment of Porter and Votta (1994) and supports their results.

Basili et al. (1996) study the effects of reading techniques on error detection in requirements documents. They are interested in the evaluation of the so-called perspective based reading: "Perspective-based reading (PBR) focuses on the point of view or needs of the customers or consumers of a document. For example, one reader may read from the point of view of the tester, another from the point of view of the developer, and yet another from the point of view of the user of the system" (Basili et al., 1996, p. 135). The main results are: (1) the PBR technique allows to identify a larger number of errors than with the 'usual' technique; (2) the PBR technique allows to discover a larger class of errors than those without PBR reading.

Laitenberger and DeBaud (1997) investigate the effects of reading techniques on error detection in code documents (C SOURCE documents). As Basili et al. (1996) they are interested in the evaluation of the so-called perspective-based reading: "PBR (···) provides instructions for an inspector in the form of operational scenarios (in short: scenario). A scenario is an algorithmic guideline on how inspectors ought to proceed while reading a software document" (Laitenberger and DeBaud, 1997, p. 781). Contrary to Basili et al., Laitenberger and DeBaud concentrate on the evaluation of the individual perspectives (analysis, module test, and integration test), which can be taken by "inspectors". The results are: (1) under individual perspectives different errors in the source code documents are found; (2) the relative number of found errors does not differ significantly under individual perspectives; (3) in additional review meetings only 6% more errors are found than under the individual perspectives.

MacDonald and Miller (1998) study the effect of tool-based versus paper-based software inspection on the error detection rate. The main results are: with respect to the number of defects found and as well as with respect to false positives tool-based and paper-based software inspection do not differ.

4.7. Experiments on Reuse Techniques

Techniques of planned reuse can be divided in two basic categories: The development of reusable software building blocks (components, frameworks, architectures) and the generation of software (Sametinger, 1997). A prerequisite for the reuse of software building blocks is to retrieve them. For this, knowledge-, hypertext- and index-based retrieval techniques exist (Frakes and Gandel, 1990). For developing the software engineering theory the experiments by Frakes and Pole (1994) and Zweben et al. (1995) can be used.

Frakes and Pole (1994) analyze representation techniques (attribute-value, enumerated, faceted, keyword) to retrieve reusable software components. The main results are: (1) representation techniques do not differ with respect to precision and recall; (2) concerning search time the representation techniques differ significantly: search time is longest for keywords, shortest for enumerated classification; (3) none of the representation techniques has advantages regarding component understanding.

Zweben et al. (1995) study the effects of layering and encapsulation on software development cost and quality in three experiments. In experiment I software developers have to extend the functionality of a simple component; in experiment II they have to extend the functionality of a complex component; in experiment III, they have to modify the functionality of a component. The results of experiment I are: (1) development costs with a black box layering approach are significantly lower than those with a white box approach; (2) there were no differences with respect to quality. The results of experiment II are similar to experiment I, but the quality of complex software components is significantly higher when the black box layering approach is used. The results of experiment III are: the development costs with the black box layering approach are significantly lower than the development costs with the white box layering box approach.

5. The Preliminary Software Engineering Theory

Figure 3 presents the preliminary software engineering theory. The theory distinguishes fundamental, central, and elementary software engineering hypotheses depending upon degree of abstraction. The glue that keeps the hypotheses together is entailment "$A \blacktriangleright B$", which is read 'A entails B'—not to be confused with 'A implies B', i.e. '$A \rightarrow B$'. "Entailment, the glue that keeps the formulas of a theory together, is stronger than implication": if $A \blacktriangleright B$, then $A \rightarrow B$ but not conversely (see Bunge, 1967, p. 403). Entailment in the context of our theory depicted in Figure 3 means that fundamental hypotheses cover central hypotheses, and central hypotheses cover elementary hypotheses.

5.1. Fundamental Hypotheses

In most theories a small subset of hypotheses can be regarded as fundamental. The hypotheses *F*1, *F*2, and *F*3 play this role of fundamental hypotheses in the preliminary software engineering theory.

*F*1 When applying software engineering techniques team composition (e.g. team size) has an effect on software development.

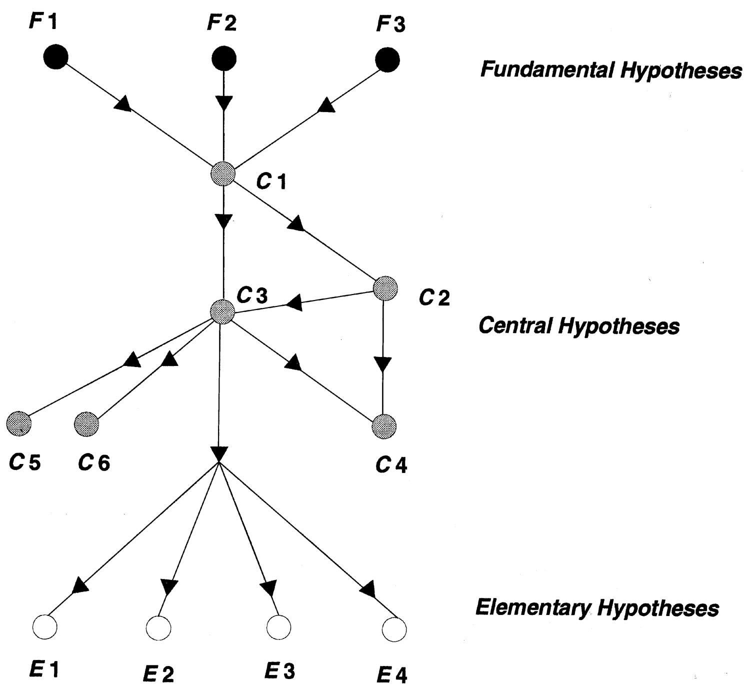

Figure 3. Preliminary software engineering theory.

*F*2 Software engineering techniques of different paradigms (e.g. functional, object-oriented paradigm) differ in their efficiency and effectiveness.

*F*3 When using software engineering techniques the experience of software developers has an effect on the efficiency of software development.

*F*1 is empirically tested by the experimental findings of Basili and Reiter (1981), Basili and Selby (1987) and Porter et al. (1995). *F*2 is grounded by the experimental work of Briand et al. (1997), Henry et al. (1990) and Moynihan (1996). By the experimental results of Agarwal et al. (1996), Basili and Selby (1987) and Porter and Votta (1994) *F*3 is empirically confirmed.

5.2. Central Hypotheses

*C*1, *C*2, *C*3, *C*4, *C*5, and *C*6 are the central hypotheses in the preliminary software engineering theory. Central hypotheses are more concrete than the fundamental hypotheses but abstract over concrete experimental findings.

*C*1 Software engineering techniques of one paradigm (e.g. object-oriented paradigm) differ in their efficiency and effectiveness.

*C*2 When using software engineering techniques the type of application system has an effect on software development.

*C*3 Software engineering techniques for one activity differ in their efficiency and effectiveness.

*C*4 When applying software engineering techniques the task to be solved has an effect on the efficiency of software development.

*C*5 When using software engineering techniques information given by the client has an effect on software development.

*C*6 When using the same concept of a software technique (e.g. inheritance) with respect to its concrete application (inheritance depth) this concept may have different effects on efficiency and effectiveness during software development.

*C*1 is experimentally supported by the experimental results of Lohse and Zweben (1984), Mills (1996), Yadav et al. (1989) and others. Empirically *C*2 is grounded by the experimental work of Agarwal et al. (1996), Rombach (1987) and Scanlan (1989). *C*3 is empirically supported by the experimental findings of Frakes and Pole (1994), Korson and Vaishnavi (1987) and MacDonald and Miller (1998). *C*4 is experimentally confirmed by the investigations of Benander et al. (1996), Prechelt and Tichy (1996) and Swigger and Brazile (1991). *C*5 is supported by the experimental results of Moynihan (1996), *C*6 is confirmed by the experimental findings of Daly et al. (1996).

5.3. Elementary Hypotheses

Elementary hypotheses are the most concrete hypotheses in our preliminary software engineering theory. *E*1, *E*2, *E*3, and *E*4 concentrate on similar results of several software engineering experiments.

*E*1 Design techniques with coarse-grained structuring concepts are superior to those without coarse-grained structuring concepts when developing complex application systems with respect to costs, quality, and maintenance.

*E*2 When applying implementation techniques structured programming has advantages against unstructured programming regarding efficiency, quality and costs.

*E*3 With professional programmers code reading is superior to functional and structural testing regarding effectiveness at isolating faults.

*E*4 The scenario technique (PBR technique) for error detection is superior to check lists and ad hoc testing during software development with respect to the error location rate in requirements and source code documents.

*E*1 is empirically confirmed by the experiments of Korson and Vaishnavi (1987), Rombach (1987) and Zweben et al. (1995). *E*2 is supported by the experimental results of Lucas and Kaplan (1976) as well as Basili and Reiter (1981). *E*3 is empirically grounded by the experiments of Basili and Selby (1987), Kamsties and Lott (1995) as well as Roper et al. (1997). *E*4 is very well supported by the experimental results of Basili et al. (1996), Porter et al. (1995), Porter and Votta (1994) as well as Laitenberger and DeBaud (1997).

6. Usage of the Preliminary Software Engineering Theory

The preliminary software engineering theory presented in Section 5 comply with the following goals: (1) posing and reformulating fruitful problems; (2) suggesting the gathering of new data; (3) suggesting entire new lines of investigation.

6.1. Posing and Reformulating Fruitful Problems

The preliminary software engineering theory contains the central hypotheses *C*2, *C*4 and *C*5 that deal with software engineering techniques with respect to type of application system, type of task to be solved, and information given by the client. For further confirming the central hypotheses further experiments should be conducted.

There should be investigations into:

- "the effects of software engineering techniques with respect to different types of application systems (real time systems, interactive systems, agent-based systems) on the efficiency and effectiveness during software development";
- "the effects of software engineering techniques with respect to different types of task (new development, restructuring, maintenance of an application) on the efficiency and effectiveness during software development";
- "the effects of software engineering techniques with respect to different information given by the client (range, frequency) on the efficiency and effectiveness during software development".

6.2. Suggesting the Gathering of New Data

The preliminary software engineering theory covers the elementary hypotheses *E*1, *E*2, *E*3 and *E*4, which permit statements concerning concrete design, implementation, test, and quality assurance techniques. These statements should be confronted with further empirical data. Besides, further experiments should be conducted that allow statements concerning analysis, maintenance, and reuse techniques.

There should be investigations into:

- "the effects of particular analysis techniques on the efficiency and effectiveness during software development";
- "the effects of particular maintenance techniques on the efficiency and effectiveness during software development";
- "the effects of particular reuse techniques on the efficiency and effectiveness during software development".

6.3. Suggesting Entirely New Lines of Investigation

The preliminary software engineering theory does not contain any software engineering hypotheses that deal with the evaluation of formal methods such as VDM

(Jones, 1990), Z (Diller, 1994), Larch (Guttag and Horning, 1993), etc. Moreover, no experimental software engineering experiments have been conducted so far.

There should be investigations into:

- "the effects of different formal methods on the quality of application systems";
- "the effects of formal versus semi and non-formal methods on the quality of application systems";
- "the effects of formal versus semi and non-formal methods on the efficiency and effectiveness when developing application systems".

The suggested problems represent a selection of possible new investigation lines. They serve to illustrate how problems can be derived on the basis of a software engineering theory.

7. Conclusions

In this article a preliminary software engineering theory is developed. Starting point for it were the results of published software engineering experiments that investigate analysis, design, implementation, test and maintenance techniques as well as quality assurance and reuse techniques. For developing the preliminary software engineering theory the concept of 'theory' was used that stems from science of science. New problems and new investigation lines were derived from the developed preliminary software engineering theory.

The submitted preliminary software engineering theory is a first attempt to systematically summarize the present results of software engineering experiments. A next step should be—as soon as further experimental software engineering data are available—the formalization of the software engineering theory by algebraic or set-theoretical concepts. By this step precision, possibilities for deduction and testability are gained.

Acknowledgment

I would like to thank Mr Klaus Kobler for final readings.

Notes

1. Due to lack of space only the first author is referenced.

References

Abbattista, F., Lanubile, F., Mastelloni, G., and Visaggio, G. 1994. An Experiment on the Effect of Design Recording on Impact Analysis. In: *Proceedings of the International Conference on Software*

Maintenance (September 19–23, Victoria, British Columbia, Canada). Washington: IEEE Computer Society Press, pp. 253–259.

Agarwal, R., Sinha, A. P., and Tanniru, M. 1996. The Role of Prior Experience and Task Characteristics in Object-Oriented Modelling: An Empirical Study. *International Journal of Human-Computer Studies* 45: 639–667.

Ashworth, C., and Slater, L. 1993. *An Introduction to SSADM Version 4*. London: McGraw-Hill.

Basili, V. R. 1990. Viewing Maintenance as Reuse-Oriented Software Development. *IEEE Software* 7(1): 19–25.

Basili, V. R., Selby, R., and Hutchens, D. 1986. Experimentation in Software Engineering. *IEEE Transactions on Software Engineering* SE-12(7): 733–743.

Basili, V. R., and Reiter, R. 1981. A Controlled Experiment Quantitatively Comparing Software Development Approaches. *IEEE Transactions on Software Engineering* SE-7(3): 299–311.

Basili, V. R., and Selby, R. W. 1987. Comparing the Effectiveness of Software Testing Techniques. *IEEE Transactions on Software Engineering* SE-13(12): 1278–1296.

Basili, V. R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgård, S., and Zelkowitz, M. V. 1996. The Empirical Investigation of Perspective-Based Reading. *Empirical Software Engineering* 1(2): 133–164.

Benander, A. C., Benander, B. A., and Pu, H. 1996. Recursion vs. Iteration: An Empirical Study of Comprehension. *Journal of Systems and Software* 32: 73–82.

Briand, L. C., Bunse, C., Daly, J. W., and Differding, C. 1997. An Experimental Comparison of the Maintainability of Object-Oriented and Structured Design Documents. *Empirical Software Engineering* 2(3): 291–312.

Bunge, M. 1967. *Scientific Research I: The Search for System*. Berlin: Springer.

Cartwright, M. 1998. An Empirical View of Inheritance. *Information and Software Technology* 40(4): 795–799.

Chikofsky, E., and Cross, J. H. 1990. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software* 7(1): 13–17

Daly, J., Brooks, A., Miller, J., Roper, M., and Wood M. 1996. Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software. *Empirical Software Engineering* 1(2): 109–132.

Diller, A. 1994. *Z (An Introduction to Formal Methods)*. New York: Wiley.

Frakes, W. B., and Gandel, P. B. 1990. Representing Reusable Software. *Information and Software Technology* 32(10): 653–664.

Frakes, W. B., and Pole, T. P. 1994. An Empirical Study of Representation Methods for Reusable Software Components. *IEEE Transactions on Software Engineering* SE-20(8): 617–630.

Fusaro, P., Lanubile, F., and Visaggio, G. 1997. A Replicated Experiment to Assess Requirements Inspection Techniques. *Empirical Software Engineering* 2(1): 39–57.

Grant, E. E., and Sackman, H. 1967. An Exploratory Investigation of Programmer Performance Under On-Line and Off-Line Conditions. *IEEE Transactions on Human Factors in Evolution* HFE-8: 33–48.

Green, T. R. G., Petre, M., and Bellamy, R. K. E. 1991. An Empirical Study of the Effects of Design/Documentation Formats on Expert System Modifiability. In: J. Koenemann-Belliveau, T. G. Moorhen and SP. Robertson (eds.): *Empirical Studies of Programmers: Fourth Workshop*. Norway: Able Publishing, pp. 121–146.

Guttag, J. H., and Horning, J. J. 1993. *Larch: Languages and Tools for Formal Specification*. New York: Springer.

Henry, S., Humphrey, M., and Lewis, J. 1990. Evaluation of the Maintainability of Object-Oriented Software. In: *IEEE Region 10 Conference on Computer and Communication Systems*. New York: IEEE Computer Society Press, pp. 404–409.

Johnson, P. M., and Tjahjono, D. 1998. Does Every Inspection Really Need a Meeting? *Empirical Software Engineering* 3(1): 9–35.

Jones, C. B. 1990. *Systematic Software Development using VDM*. 2nd Ed. New York: Prentice Hall.

Kamsties, E., and Lott, C. M. 1995. An Empirical Evaluation of Three Defect-Detection Techniques. In: W. Schäfer and P. Botella (eds.): *Proceedings of the Fifth European Software Engineering Conference,* Berlin: Springer, pp. 362–383.

Kieburtz, R. B., McKinney, L., Bell, J. M., Hook, J., Kotov, A., Lewis, J., Oliva, D. P., Sheard, T., Smith, I., and Walton, L. 1996. A Software Engineering Experiment in Software Component Generation. In: *Proceedings of the 18th International Conference on Software Engineering*. New York: IEEE Computer Society Press, pp. 542–553.

Kiper, J. D., Auernheimer, B., and Ames, C. K. 1997. Visual Depiction of Decision Statements: What is Best for Programmers and Non-Programmers? *Empirical Software Engineering* 2: 361–379.

Korson, T. D., and Vaishnavi, V. K. 1987. An Empirical Study of the Effects of Modularity on Program Modifiability. In: E. Soloway and S. Iyengar (eds.): *Empirical Studies of Programmers: First Workshop*. Norway: Able Publishing, pp. 168–186.

Laitenberger, O., and DeBaud, J.-M. 1997. Perspective-Based Reading of Code Documents at Robert Bosch GmbH. Evaluation and Assessment in Software Engineering. *Information and Software Technology* 39(11): 781–791.

Lohse, J. B., and Zweben, S. H. 1984. Experimental Evaluation of Software Design Principles. *Journal of Systems and Software* 4: 301–308.

Lott, C. M. and Rombach, H. D. 1996. Repeatable Software Engineering Experiments for Comparing Defect-Detection Techniques. *Empirical Software Engineering* 1(3): 241–277.

Lucas, H. C., and Kaplan, R. B. 1976. A Structured Programming Experiment. *The Computer Journal* 19(2): 136–138.

MacDonald, F., and Miller, J. 1998. A Comparison of Tool-Based and Paper-Based Software Inspection. *Empirical Software Engineering* 3(3): 233–253.

Martin, J. 1992. *Strategic Data Modelling*. New Jersey: Prentice Hall.

Meyers, S., and Reiss, S. P. 1992. An Empirical Study of Multiple-View Software Development. *ACM SIGSOFT Software Engineering Notes* 17(5): 47–57.

Miller, J., Wood, W., and Roper, M. 1998. Further Experiences with Scenarios and Checklists. *Empirical Software Engineering* 3(1): 37–64.

Mills, K. L. 1996. An Experimental Evaluation of Specification Techniques for Improving Functional Testing. *Journal of Systems and Software* 32: 83–95.

Moher, T. G., Mal, D. C., Blumenthal, B., and Leventhal, L. M. 1993. Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets. In: C. R. Cook, J. C. Scholtz and J. C. Spohrer (eds.): *Empirical Studies of Programmers: Fifth Workshop*. Norwood, Ablex Publishing, pp. 137–186.

Moynihan, T. 1996. An Experimental Comparison of Object-Orientation and Functional Decomposition as Paradigms for Communicating System Functionality to Users. *Journal of Systems and Software* 33(2): 163–169.

Myers, G. J. 1978. A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections. *Communications of the ACM* 21(9): 760–768.

Perpich, J. M., Perry, D. E., Porter, A. A., Votta, L. G., and Wade, M. W. 1997. Anywhere, Anytime Code Inspections: Using the Web to Remove Inspection Bottlenecks in Large-Scale Software Development. In *Proceedings of the 19th International Conference on Software Engineering*. New York: IEEE Computer Society Press, pp. 14–21.

Perry, W. 1995. *Effective Methods for Software Testing*. New York: Wiley.

Porter, A. A., and Votta, L. G. 1994. An Experiment to Assess Different Defect Detection Methods for Software Requirements Inspections. In: *Proceedings of the 16th International Conference on Software Engineering*. Washington: IEEE Press, pp. 103–112.

Porter, A. A., Siy, H. P., Mockus, A., and Votta, L. 1998. Understanding the Sources of Variation in Software Inspections. *ACM Transactions on Software Engineering and Methodology* 7(1): 41–79.

Porter, A. A., Votta, L. G., and Basili, V. R. 1995. Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment. *IEEE Transactions on Software Engineering* SE-21(6): 563–575.

Prechelt, L. 1997. An Experiment on the Usefulness of Design Patterns: Detailed Description and Evaluation. Technical Report 9/1997. Fakultät für Informatik, Universität Karlsruhe.

Prechelt, L., and Tichy, W. F. 1996. An Experiment to Assess the Benefits of Inter-Module Type Checking. In: *METRICS Symposium Berlin*. New York: IEEE, pp. 112–119.

Prechelt, L., and Tichy, W. F. 1998. A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking. *IEEE Transactions on Software Engineering* 24(4): 302–312.

Rombach, H. D. 1987. A Controlled Experiment on the Impact of Software Structure on Maintainability. *IEEE Transactions on Software Engineering* SE-13(3): 344–354.

Roper, M., Wood, M., and Miller, J. 1997. An Empirical Evaluation of Defect Detection Techniques. *Information and Software Technology* 39(11): 763–775.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenson, W. 1991. *Object-Oriented Modeling and Design*. Englewood Cliffs: Prentice Hall.

Sametinger, J. 1997. *Software Engineering with Reusable Components*. Berlin: Springer.

Scanlan, D. A. 1989. Structured Flowcharts Outperform Pseudocode: An Experimental Comparison. *IEEE Software* 6(5): 28–36.

Shneiderman, B., Mayer, R., McKay, D., and Heller, P. 1977. Experimental Investigations of the Utility of Detailed Flowcharts. *Communications of the ACM* 20(6): 373–381.

Sinha, A. P., and Vessey, I. 1992. Cognitive Fit: An Empirical Study of Recursion and Iteration. *IEEE Transactions on Software Engineering* SE-18(5): 368–379.

Swigger, K. M., and Brazile, R. P. 1991. An Empirical Study of the Effects of Design/Documentation Formats on Expert System Modifiability. In: C. R. Cook, J. C. Scholtz and J. C. Spohrer (eds.): *Empirical Studies of Programmers: Fifth Workshop*. Norway: Able Publishing, pp. 210–226.

Tardieu, H., Rochfeld, A., and Colletti, R. 1989. *La Méthode Merise*. Paris: Les Editions d'Organisation.

Tryggeseth, E. 1997. Report from an Experiment: Impact of Documentation on Maintenance. *Empirical Software Engineering* 2(3): 201–207.

Weiser, M., and Lyle, J. 1986. Experiments on Slicing-Based Debugging aids. In: E. Soloway and S. Iyengar (eds.): *Empirical Studies of Programmers: First Workshop*. Norway: Able Publishing, pp. 187–197.

Yadav, S. B., Bravoco, R. R., Chatfield, A. T., and Rajkumar, T. M. 1989. Comparison of Analysis Techniques for Information Processing. *Communications of the* ACM 31: 1090–1097.

Zendler, A. 1999. *Elements of experimental software engineering*. Habilitationsschrift, Universität Potsdam.

Zweben, S. H., Edwards, S. H., Weide, B. C., and Hollingsworth, J. E. 1995. The Effects of Layering and Encapsulation on Software Development Cost and Quality. *Transactions on Software Engineering* SE-21(3): 200–208.

Andreas Zendler received the Ph.D. degree (Dr.rer.nat.) in Computer Sciences from the University of Postdam, Germany in 1997 and also the Ph.D. degree (Dr.phil.) in Experimental Psychology from the University of Regensburg, Germany in 1988. He is currently head of a software development department at MGM EDV-Beratung GmbH, Germany. His current research interests include software architecture, component technology, empirical software engineering, and especially bioinformatics. He is member of the German Society of Informatics (GI), IEEE and ACM.