# 04051 Abstracts Collection
# Perspectives Workshop: Empirical Theory and the Science of Software Engineering
## — Dagstuhl Seminar —

James D. Herbsleb[1] and Walter F. Tichy[2]

[1] CMU - Pittsburgh, US
jherbsleb@acm.org
[2] Univ. Karlsruhe, DE
tichy@ira.uka.de

**Abstract.** From 25.01.04 to 29.01.04, the Dagstuhl Seminar 04051 "Perspectives Workshop: Empirical Theory and the Science of Software Engineering" was held in the International Conference and Research Center (IBFI), Schloss Dagstuhl. During the seminar, several participants presented their current research, and ongoing work and open problems were discussed. Abstracts of the presentations given during the seminar as well as abstracts of seminar results and ideas are put together in this paper. The first section describes the seminar topics and goals in general. Links to extended abstracts or full papers are provided, if available.

## Theory for studying effective work practices for Open Source Software development

*Kevin Crowston (Syracuse University, USA)*

In my talk, I will discuss an on-going project examining the general research question: what practices make some Free/Libre Open Source Software (FLOSS) development teams more effective than others? In the talk, I'll first talk a bit about theory in general, then discuss the general framework for the study, the role of theory and the proposed design, and conclude by reviewing some recent results.

As a conceptual basis for our study, we draw on Hackman's model of effectiveness of work teams. Following on work by Crowston and Kammerer, we also use coordination theory and collective mind to extend Hackman's model by further elaborating team practices relevant to software development. The literature on shared mental models, collective mind theory in particular, focuses our attention on actions that develop and exhibit shared understandings. Coordination theory suggests identifying tasks, interdependences among tasks and resources and the coordination mechanisms that are adopted.

The recent results are from a social network analysis of interaction in FLOSS teams for bug fixing.We found teams exhibited a wide range of centralizations,

contrary to our expectation from FLOSS publications that they would be generally centralized.

## Let's get Metaphysical! or Validating requirements and validating scientific theories

*Steve Easterbrook (University of Toronto, CDN)*

In this talk I address the question of validation. My starting point was a consideration of how we validate requirements. By drawing an analagy between requirements and scientific theories, I conclude that the empirical difficulties that affect the scientific method are also relevant to requirements engineering. I briefly survey key ideas in the philosophy of science and the sociology of science, to show that empirical validation (of scientific theories) is neither as precise nor as complete as is generally supposed. I then contrast logical positivism with postmodernism to gain vastly different perspectives on the role of validation.

Finally, if we wish to apply these ideas to the question of theory building in software engineering, we need to consider whether software engineering really can be considered to be science-like, so I explore the nature of engineering, the nature of design, and the reasons why software design is radically different from industrial design. I conclude that any attempt at theory building in SE must be with respect to a given Weltanschauung, so I attempt to articulate my own Weltanschauung, to illustrate how it influences my own theory- building. As a postscript to the talk, I offer a brief survey of common research idioms in software engineering.

## Toward a social psychology of software engineering

*Thomas Finholt (University of Michigan, USA)*

The social nature of software development and use suggests the applicability of theory from social psychology to understanding aspects of software engineering. Specifically, the concept of common ground can be used to explore communication dysfunction in software development teams. Common ground is a term developed by Clark to describe the extent of mutual understanding that exists among two or more people engaged in cognitively complex work. In particular, following from Nisbett's "geography of thought" this talk argues that different cultural orientations, either defined by national or professional culture, make it difficult to achieve common ground. For example, within the NEESgrid project, the collaboratory element of the NSF's George E.Brown, Jr. Network for Earthquake Engineering Simulation, software and earthquake engineers used common terms - but with different meanings. Therefore, early in the project the two groups worked at cross-purposes, resulting in poorly calibrated expectations by

the system users (earthquake engineers) and incomplete understanding of requirements by the system builders (software engineers).

There are two approaches that may help avoid breakdowns in the creation and maintenance of common ground. First, because of the difficulty associated with collecting communication data to test hypotheses about common ground, there is a potentially important role for computational simulation in understanding communication and collaboration in software teams. An aim of this simulation work should be models that allow researchers and practitioners to easily explore competing explanations. Second, improved understanding of the impact of common ground on group process and performance requires new methods and measures. These new methods and measures must be easy to opreationalize and use, such that feedback becomes more proximate to actions. For instance, dynamic social network visualizations of communication patterns provide a means to quickly analyze information flow within a team, such as from email traffic. To conclude, there are well-understood theories from psychology, such as common ground, that can be used to understand important processes in software engineering - given that methods and measures are developed that allow quick and efficient analysis.

## Modeling Software Changes

*Audris Mockus (Avaya - Basking Ridge, USA)*

Software systems are changed constantly throughout their lifetime.

Understanding relationships between different types of changes and the effects of these changes on the success of software projects is essential to make progress in Software Engineering. By using novel methods and tools to retrieve, process, and model data from ubiquitous change management databases at the granularity of Modification Requests (individual changes to software) we have gained insights regarding the relationships between process/product factors and key outcomes, such as, quality, effort, and interval.

We exemplify this approach by describing models based on a premise that each modification to software will cause changes later and investigate their theoretical properties and applications to several software projects. The models present a unified framework to investigate and predict effort, schedule, and defects of a software project. The results of applying these models confirm a fundamental relationship between the new feature and defect repair changes and demonstrate model's predictive capabilities in large software projects.

## Non-linear Modelling in Software Engineering

*Frank Padberg (Universität Karlsruhe, D)*

Here are a few - possibly controversial - statements about theory building in software engineering:

• In software engineering, we encounter non-linear phenomena all the time. The reason are feedback loops in the underlying processes.

•There are techniques in statistics, operations research, and machine learning for detecting and handling non-linearity.

•We have an abundance of models at various levels of abstraction. Most models - in particular, formal models with explanatory power - have an underlying theory. Hence, we do know how to build theories; weŠre just not explicit enough about this.

•We should start to classify existing models in software engineering along several dimensions, including model scope, explanatory power, generalization ability, model purpose, and applicability exceptions.

•We already can perform tradeoff analysis (and even optimization) based on empirical observations and regression models without necessarily being able to explain the mechanisms underlying a phenomenon, that is, without having an underlying theory for the process. An example is our tradeoff analysis for pair programming.

## Software Process Simulation: A Potential Platform for Enriching Empirical Studies

*David M. Raffo (Portland State University, USA)*

Software Process Simulation is a tool that has shown some success in addressing issues of strategic management, project management planning and control, technology adoption, training and understanding. Some of the unique advantages over other types of model formulations include:
1. Ability to incorporate relationships from multiple modeling paradigms,
2. Relaxed restrictions compared to closed form analytic paradigms,
3. Ability to include empirically observed distributions, ability to capture rich causality models, and
4. Freedom to explore risks and changes in scenarios to address questions.Capturing of dynamic system behavior

This paper presents an overview of currently utilized simulation paradigms along with their advantages and limitations. We then discuss the usefulness of

simulation as a tool to support software engineering theory formulation and testing. Moreover, we present how simulation can be used to enrich software engineering theory through extension, condition contingency and boundary exploration. Examples from previous work are shown.

## Goodness Criteria for Empirical Theory

*Susan Elliott Sim (Univ. California - Irvine, USA)*

Literature from philosophy, sociology, psychology, political science, and others were reviewed for criteria for evaluating empirical theory. Three types of goodness criteria were identified: empirical, analytical, and pragmatic. The empirical criteria are postdictive power, predictive power, testability, and relevance. The analytic criteria are logical soundness, generality of explanans, hypothetical yield, progressive research program, breadth of policy implications, and parsimony. The pragmatic criteria are increase in understanding and persuasiveness to scientific peers. Use of these criteria were illustrated using a theory of benchmarking, which I developed to account for the impact of benchmarking on a scientific community.

## Defining the scope of empirical theories in software engineering

*Dag Sjøberg (Simula Reseach Laboratory - Lysaker, N)*

The need for empirical evaluation of software engineering technologies (processes, methods, techniques, languages or tools) has been more widely recognized in the last decade. In mature sciences, building theories is the way to gain and cumulate general language. Initial attempts have been made to formulate and test theories based on empirical evidence in software engineering, but there are few guidelines for how to develop theories that are useful to the software engineering community. An important part of a theory is its universe of discourse Ũ its scope, that is, for what populations and settings are the propositions or hypotheses of a theory expected to hold? I suggest that empirical theories in software engineering should represent knowledge about which technology is useful for whom to conduct which (software engineering) tasks in which contexts. Hence, a theory should be considered in terms of (1) the subjects that are supposed to benefit from the technology, (2) the tasks that it should support (including the system on which the tasks should be applied), and (3) the contexts in which the technology is supposed to be useful. An example of a series of experiments that extend the scope along these three dimensions will presented.

## The Limits of Empiricism

*Walter F. Tichy (Universität Karlsruhe, D)*

Software research has come to depend on empirical studies to validate contributions to the field. However, empirical research alone does not accumulate a coherent body of knowledge. Validating, for example, that method A is superior to method B does not necessarily tell us anything about other, existing or yet-to-be-developed methods. Nor do experiments provide reasons why a method is better than others or which hypotheses are most in need of testing. In the mature sciences, these roles are played by scientific theory and models.

We illustrate the power of models with several examples. First, we suggest that chunking theory explains why design patterns have the empirically observed, positive effects on programmer performance.

Next, we summarize the theory about software inspections by Sauer et al. A model for pair programming (a core practice of Extreme Programming) demonstrates the explanatory and predictive power of a simple model. Finally, we introduce a stochastic model for software cost estimation and suggest that stochastic models that provide probabilty distributions for the variables of interest should be sought.