Using Academic Courses for Empirical Validation of Software Development Processes

Marcus Ciolkowski[1,2], Dirk Muthig[2], and Jörg Rech[1,2]

[1]*University of Kaiserslautern*
Kaiserslautern, Germany
+49 (0)631 205 3339
ciolkows@informatik.uni-kl.de

[2]*Fraunhofer IESE*
Kaiserslautern, Germany
+49 (0)6301 707 {161, 210}
{muthig, rech}@iese.fraunhofer.de

Abstract

Software Process Improvement needs sound empirical data gathered from a range of empirical studies such as controlled experiments or case studies. However, conducting empirical studies is often cost-intensive; in particular in industrial environments. Therefore, we need to find a way to balance cost and value of empirical findings. In this paper, we discuss a way of gathering required empirical data in Software Engineering education programs. Although an academic setting is not representative of industrial organizations, the information gathered there can help organizations to decide upon alternatives to improve current software engineering processes. We describe how a practical software engineering course for graduate students at the University of Kaiserslautern was used as a platform for validating software development approaches. As there are specific restrictions and constraints defined by this context, we discuss how they influence the design of empirical studies. In addition, we present a concrete example of comparing product line approaches.

1. Introduction

Software Process Improvement (SPI) faces the challenge to decide which Software Engineering technique or approach is appropriate under which circumstances. To be able to make such an informed decision, SPI needs sound empirical data. However, conducting empirical studies is costly; not only the cost for planning and analyzing the study is high, but also the subjects of studies sometimes have to be paid, at least if they are professionals.

Several attempts exist that aim at capturing experience and making it available (through internet portals) for industry; for example, the German ViSEK and VSEK projects (www.software-kompetenz.de). In particular, small and medium enterprises (SMEs) rely on such portals, as they often do not have the resources for conducting empirical studies and analyses. However, it is often hard to find relevant empirical evidence or experience. One reason is that the software industry does either not collect such data, or is unwilling to share it to maintain their technological advantage. One possibility to close that gap is to include studies conducted at universities. In this view, empirical studies at universities open an additional possibility to gain empirical knowledge.

In this paper, we present how Software Engineering education programs at universities can be used to conduct cost-effective studies. Although it can be argued that students are not representative of professional developers, the evidence gained there allows valuable conclusions, and under certain conditions, the difference between students and professionals is quite small [8]. A second advantage of such studies is that the participants can gather practical experience in applying the technique or approach under investigation; this can help to convince students as well as researchers and practitioners of the advantages of Software Engineering techniques and approaches.

To illustrate the discussion, we present an example of an evaluation of a software product line approach that was conducted in the context of a practical course at the University of Kaiserslautern.

The remainder of this paper is structured as follows. Section 2. generally discusses the design of empirical studies with respect to the restrictions of academic courses. In Section 3., we introduce our SE course, which covers the software development life-cycle. The description of a concrete empirical validation is given in Section 4. We end this paper with a conclusion and outlook in Section 5.

2. Using Experiments in Education

In the recent years, many empirical studies have been conducted in Software Engineering field. However, these studies have mostly been undertaken in academic environments, still rare in industrial settings.

In this section, we briefly introduce empirical studies and discuss their advantages and disadvantages. Then, we will elaborate how empirical studies for research can be employed in Software Engineering education.

2.1. Types of empirical studies

We distinguish between three types of studies, according to Fenton and Pfleeger [6]:

- **Controlled Experiments (Research in the small)**: A controlled experiment is a rigorous, controlled investigation of an activity. In a controlled experiment, subjects are randomly assigned to experimental conditions. The purpose is to study the effect of changing one or more factors. Therefore, the researcher compares at least two experimental conditions (i.e., two different treatments) against each other; for example, the situation under investigation and a control situation. Typically, controlled experiments are conducted off-line in laboratory environments, but on-line controlled experiments (i.e., experiments embedded in a "real" project) are also possible.
- **Case Studies (Research in the typical)**: A case study typically monitors a project or assignment. Case studies are normally aimed at tracking a specific attribute or establishing relationships between different attributes. Case studies usually monitor small but typical projects or assignments of the organization, rather than trying to capture information about all possible cases. Usually, case studies do not compare the effects of different treatments.
- **Surveys (Research in the large)**: Usually a retrospective study to describe relations and outcomes. This can be done by analyzing data from past projects, or by asking people about their past experiences (e.g., via interviews). We will not consider surveys in the following, as they are retrospective.

Usually, empirical studies take the form of either small, controlled experiments at universities using students, or case studies in industry (e.g., accompanied by introduction of measurement programs). The first case is popular because students are easily available for academics (as they have to attend courses anyway), don't cost much, and because controlled experiments can be done in a short time period; for example, as part of a lecture. The second case is used often because industrial organizations can afford to introduce and test new ideas in small, low-risk projects (and extend them later to other projects).

Both approaches carry advantages and risks. The main advantage of controlled experiments is that they offer higher significance of results (because more variables can be controlled) compared to case studies. For example, it is possible to control that the participants do what they are supposed to do. For that reason, it is possible to establish cause-effect relationships, as it can be excluded (to a certain degree) that an observed effect is caused by influence factors that are out of control.

One main critique on controlled experiments is that they only allow to study small, isolated effects of technologies, not the 'big picture'; that is, the interaction of a technology with the whole development process. Further, the typical participants in controlled experiments are students. Although it can be argued that students are not representative of professional developers, many useful lessons can be learned through university experiments [3][4]. Moreover, a recent study by Höst et al. [8] showed that the difference between students and professionals are sometimes only minor. They conclude that, under certain conditions, students are even representative for professional developers.

Case studies, in contrast, offer the advantage that they allow to examine approaches that cover large parts of the software lifecycle, as well as side-effects

of technologies (both of which can usually not be examined in controlled experiments).

The weakness of case studies is that the control over the study is lower: They are usually "real" projects, so the experimenter may not be able to influence project goals as much as s/he would like. Further, they cover a longer time period than controlled experiments, which makes it impossible to control some aspects (such as developer fluctuation) that may have an influence on the result. Thus, it is usually not possible to establish a cause-effect relationship with a case study alone, as the observed effect could be caused by other factors than the examined one. Moreover, the generalizability of results is lower for case studies than for controlled experiments, as the results are usually only valid for the specific situation the study was conducted in. That means that results of case studies are only applicable within the context in which they were conducted, and it is hard to predict the results in a different organization, even in a different project.

However, although case studies in industrial environments offer several advantages, they are usually cost-intensive. Gathering reliable empirical data to compare Software Engineering approaches is effort-intensive and time-consuming. The reason is that, to allow comparisons, several teams are needed that concurrently and independently apply the same or an alternative approach. In industry, this can take the form of a *sister project* (i.e., a project that has the same objective but follows a different approach) [6]. As a sister project is not productive (it duplicates work of the reference project), it is associated with high cost. This is why usually, case studies in industry are conducted in real project, and their results are—at best—compared to a company baseline. In such a comparison, the cost is lower than with a sister project, but it is not clear how representative the project is; that is, how close it is to the company baseline.

2.2. Empirical Studies and Education

In this section, we describe how empirical studies can be conducted in Software Engineering education.

We distinguish between industrial and academic studies; the former are conducted with practitioners in industry, the latter usually with students in a university setting. The problem of an *industrial study* is that it is often very expensive. There are also several critical threats to the validity of industrial studies. Contextual independence is hard to realize in the same organizations, long duration is expensive, people communicate with each other, people object to process changes, and competition among teams and mangers are common. Thus industrial studies are conducted with experienced people, have low statistical significance and high personnel cost, but the results are representative of the target environment; i.e., industry.

Academic studies usually use students as subjects who participate in a university course; the main objective of the study is therefore that the students gather practical experience in applying Software Engineering techniques and approaches. Academic studies are conducted with inexperienced students, have medium to high statistical significance (as potentially many students participate) and low personnel cost; however, the results are not necessarily representative of industry.

At the Software Engineering Research Group at the University of Kaiserslautern, we differentiate between short-term *controlled experiments* in a laboratory setting that are conducted as part of a lecture and long-term *case studies* spanning a whole practical course over approximately 14 weeks. The *controlled experiments* are carried out as part of a lecture, using at most four lecture slots of 90 minutes each of with approximately 30 to 70 students. Software development processes that influence many parts of a software development project over a long period of time are not suitable to be evaluated in these kinds of studies. A *long-term case study* is conducted within a practical course with about 15 students over the time span of a whole semester. These practical courses cover the whole lifecycle of a product; therefore, it is possible to evaluate software engineering approaches and processes that span or influence the whole project.

From the point of view of education, empirical studies provide several advantages for students. First, they can provide a practical training in state-of-the-art techniques (e.g., as part of an inspection experiment). Thus, it is possible to teach techniques better through practical application. Second, the data gained in the empirical study can be used to convince students of advantages of techniques. That is, because they are able to experience effects of their actions.

From the scientific point of view, empirical studies in education provide valuable contributions in comparing and evaluating techniques. Thereby, some technologies can be examined in experiments (such as inspections), while others (e.g., processes such as XP) can be examined only in case studies.

Although university students are not always representative of developers, successful university experiments can serve as indicator for industry that a technique or method is worthwhile examining in more detail. Hence a university environment is able to provide valuable insight into the benefits and risks of techniques and methods. This is especially true because there (and only there) it is possible (at low cost) to repeat experiments in subsequent courses and thus add significance to the observed effects. As the participating students change every year, results in such empirical studies can be a good indicator of the potential of a technique or method. Additionally, insight gained into techniques during university experiments can be used to improve them.

3. Educational context

In this section, we describe the environment in which we conduct the long-term case studies mentioned in the previous section. At the University of Kaiserslautern computer science students have to participate in practical courses during their study. We used this course to establish a laboratory as a way to teach our students the basics of Software Engineering (SE). Since its start in 1996 it is simultaneously used as a platform to evaluate SE methods and tools of the local research groups AGSE[1], SFB 501[2], and IESE[3] [13].

The goal of the laboratory is that the students gather practical experience in applying basic SE methods from analysis, design, implementation, and integration as well as verification and validation [9]. To train these methods the laboratory uses a building automation system, which was created in the laboratory and evolved from year to year. As part of the course, the students shall experience object-oriented software development in a team and learn about the difference between programming-in-the-large and programming-in-the-small [12]. Students have to attend at least one practical course after their preliminary diploma (the first 4 semesters of a 9 semester diploma study).

3.1. Participants

The course is supervised by an academic instructor, a student assistant, and (potentially) a scientist who wants to conduct an experiment. For backup purposes the supervisors are supported by various experts for the applied methods, tools and experimental issues from the local research groups.

One goal of the course is to simulate a small project and to teach the students to work in teams. In the last five courses the number of participants ranged from 10 to 25 students grouped into teams of three or four. The students are not restricted to work in a special course room at a given time. An exception from this is the course meeting once a week.

Teams

Typically, most of the teams do not have to be assigned by the supervisors. Often a group of friends registers for the course and wants to work together. The advantage of these natural teams is that they know each other, which helps them to coordinate their meetings and tasks in the course more efficiently. This is particularly important because they are allowed to work when and wherever they want (e.g. at home late at night). Therefore, to observe the work of more realistic teams, it is typically recommended to stick with the natural groups instead of setting up new teams, for example, to get more equally skilled groups of students.

Subjects

Consequently, as we usually form natural teams, we use a questionnaire to evaluate and compare the skills of the individual participants and the different teams. The essential conclusions drawn from the analysis of the questionnaires are the same for every course: Typically, students have basic knowledge about Software Engineering, but only little practical experience with the skills needed for the development

[1] AGSE: Software Engineering Research Group at the University of Kaiserslautern.

[2] SFB 501: Special Research Project for the Development of Large Generic Systems.

[3] IESE: Fraunhofer Institute Experimental Software Engineering.

of a software system. With respect to the tools, the application domain, and the concepts validated within the course, they typically have almost no experience at all. Only in the programming language (C++) they are more experienced.

3.2. System and System Documentation

The software used in the practical course is a reactive system for building automation that was created for and evolved within the course. Additionally, a simulator and testing environment with appropriate hardware are available for testing the system.

The building that is controlled by the software consists of an arbitrary number of floors and rooms that have various sensors and actors. Figure 1 shows a typical office room with the existing sensors, actors, and the control panel. This panel is used by a person (e.g. a clerk) to set variables of the software system.

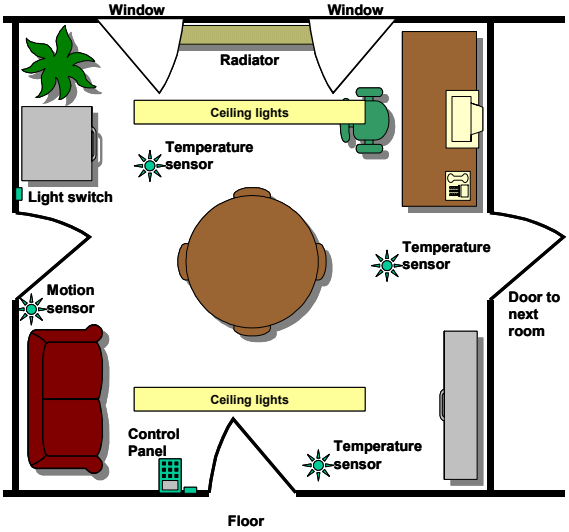

Figure 1. Layout of a room including sensors

Software Documentation

To document this building automation system we created a software documentation that is written in natural language (German) and modeled in UML using StP™ ("Software through Pictures" by Aonix). As a comprehensive documentation it currently comprises of about 1500 pages of natural text and UML diagrams. The sourcecode was written right from the start in C++ and had grown to more than 80 classes with about 22 kLoC. In 2000 it was refactored [7] and currently consists of about 10 kLoC of C++ Code in 65 classes.

As shown in Figure 2 several products are used in the course.

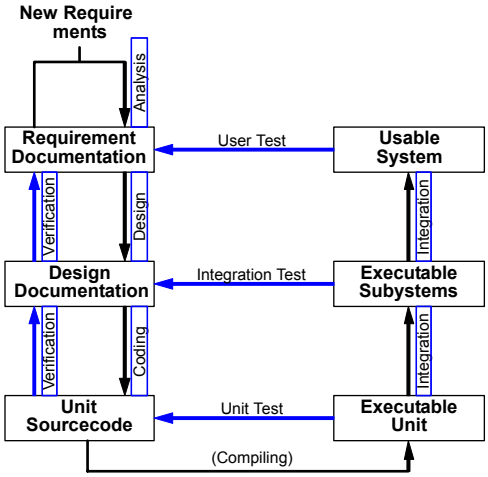

Figure 2. Product model of the course

3.3. Laboratory and Development Process

The task for the students in our practical course is to maintain and extend the existing software system. Typical changes are corrective, adaptive or perfective maintenance. Therefore, each document, from requirements to code, has to be modified.

As an example, the schedule for the practical course in 2001 is listed in Table 1, during which we evaluated a product line approach. After each phase, the students meet with the academic instructor. The goal of the meeting is to present both the tasks for the new phase and the results of the previous phase, including the modified documents and the inspection results.

Table 1: Timeplan for the course

Phase	Task	Time
Introduction	Training with the tools.	1 Week
Analysis	Analysis of the system and new requirements as well as the modification of the documentation.	2 Weeks
Change Verification	Verification of the changes.	1 Week
Design	Design of a solution and modification of	2 Weeks

	the design document.	
Design Verification	Verification of the changes as well as a consistency check with the requirement document.	1 Week
Coding	Implementation and modification of the unit sourcecode.	1 Week
Code Verification	Verification of the changes as well as a consistency check with the design document.	1 Week
Unit Test	Construction of test programs and execution of unit test cases.	1 Week
Integration Test	Construction of test programs (e.g. stubs and drivers) and execution of integration test cases.	1 Week
User Test	Execution of the user test cases and final presentation	1 Week

Students receive each abstraction of the documentation (i.e. requirements, design, source) right before the respective phase starts. This assures that no knowledge that is encoded in a more detailed document is available at an earlier phase; that is, while changing the system design, they have no knowledge how the current design is implemented in the code.

4. Validation of the Product Line Approach

In this section, we report on a validation performed in the setting described above. After providing some background information, we describe the particular design of the course and analyze the achieved results.

4.1. Background

In 1996, the Fraunhofer Institute Experimental Software Engineering (IESE) started to build up competence in the area of software product line engineering. Since then, these technologies were applied in numerous industrial transfer projects. With respect to the idea of experimental software engineering, however, there has been a lack of empirical data on experience gained with product line approaches. The reason for that is that product line engineering represents a totally different development approach than single system development, the approach mostly used in industry. Therefore, product line engineering changes nearly everything: processes and products. Product line engineering splits the overall life-cycle into two main phases: domain engineering and application engineering. During domain engineering, assets are constructed to be reused by applications in the particular domain. During application engineering these assets are reused to construct specific applications. One of the most interesting questions with a product line approach is: How big is the required initial investment to construct a useful reuse infrastructure, and which improvements (effort, time-to-market, quality, etc.) can be achieved with it? Especially for small and medium-sized companies such an investment is typically not acceptable. Hence, Fraunhofer IESE developed a lightweight product line approach that facilitates an evolutionary transition to software product lines, that is, distributing the required investment across several projects.

4.2. Design

In order to initially validate the lightweight approach for product line engineering, we decided to perform a quasi-experiment in the context of the one-semester practical course on software engineering described above. Thereby, a quasi-experiment is an experiment with lower control; in particular, the researcher is not able to assign participants randomly to experimental groups [2]. Fourteen graduate students from the computer science department at the University of Kaiserslautern were enrolled in this course. Previous to the practical course, most of the students had attended a one-semester class on the basic principles of software engineering and object-oriented modeling. The four teams, each consisting of three to four students, concurrently developed a product line of air-conditioning systems to compare the performance of the lightweight product line approach (Evolutionary Process), and the traditional, domain-analysis-based approach,. Full details of the case study are provided in [10]. Here we provide a brief presentation of the case study and its results.

The duration of the course was 14 weeks and the effort typically spent by students on such a course is ten hours per week. Hence, it is a challenge to investigate product line engineering within the given time and effort limitations. Consequently, we limited the usage of product line technology to the requirement

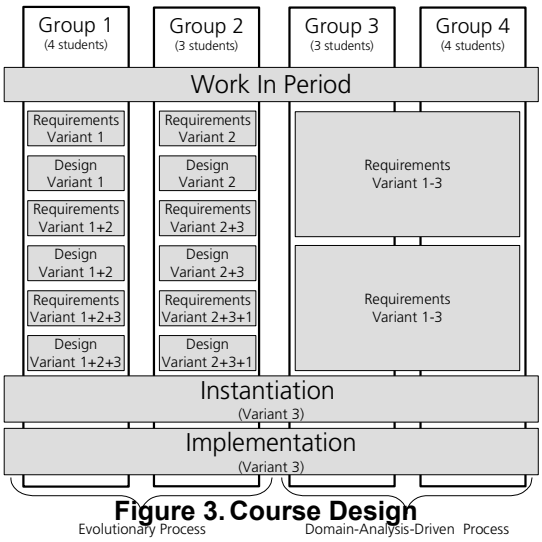

Figure 3. Course Design

and design phases of development and required only one of the variants be implemented by the students. The design of the case study is depicted in Figure 3. Two groups followed an evolutionary product line strategy: that is, they first modeled the requirements of a single system, designed it, successively integrated the second and the third variant, instantiated their generic models, and implemented one variant. The other two teams directly analyzed the requirements of all three systems, created a generic design covering all three variants, instantiated their generic models, and implemented the same variant as the first two groups.

Table 2: Feature definitions for the air-conditioning domain

Feature	Choice	Definition
Ventilation Slots	automatic	The ventilation slots are opened and closed automatically and autonomously by the system to let or stop cool air streaming into the room.
	manual	The user in a room can overrule the system by open or close the ventilation slots manually via an external control device in the room.
Window-based Cooling	isolated	The air-conditioning is always responsible for cooling the air down when the temperature is too high.
	coupled	The air-conditioning system is more integrated with overall building automation system and thus when the temperature outside is lower than inside, the windows are opened to cool the room down instead of using the air-conditioning.
Unit Power	constant	The power of the air-conditioning system is constant except when the system is completely turned off (e.g. in winter time)
	intelligent	The air-conditioning system automatically turns off its power unit if none of the rooms must be cooled down by the system and vice versa.

4.3. Analysis

A general result of the case study is that both approaches enabled all teams to model a software product line after a single day of training. Independently of the concrete effort data, the students developed a family of air-conditioning systems in 14 weeks. The students were convinced that the implementation of the other two variants would not consume major effort. Hence, the students experienced the benefits of a product line approach directly - they developed three similar systems in the same time than usually a single system was developed in the years before.

The measured effort data does not only show the high up-front investment required for the analysis-driven approach but also that this approach offers the lowest effort for future systems (compare with data published in [14]). In summary, the results from experiments back-up the theoretical models for the cost-effectiveness of the alternatives product-line strategies, and confirm the predicted benefits of incremental and lightweight migration strategies.

5. Conclusions

In this paper, we have shown how Software Engineering education programs can be used to evaluate Software Engineering approaches. We demonstrated this with an example evaluation of a software product line approach.

The validation of the product line approach showed two things: on one hand, the described practical course was a successful vehicle for educating students in the product line approach. The students experienced the benefits of product line engineering by developing three variants in the same time span in which other student teams developed single systems only. On the other hand, the effort data collected within the course closely matches published data from industrial contexts. Hence, the practical course provides results that, to a certain extend, can be transferred to industrial contexts.

Due to the gained results in validating a whole software development approach rather than single techniques only, we will continue the work in validating development approaches within the described context. In the future, we will utilize more systematically a series of subsequent courses to get more significant or complimentary results. For instance, the same product line development is repeated to add more significance to the initial results, or other aspects of the product line approach are validated to get a more complete picture of the advantages and risks involved in the product line approach.

ACKNOWLEDGEMENTS

This work was supported by the Project VSEK (www.software-kompetenz.de), funded by the German Federal Ministry of Education and Research (BMBF) under grant 01ISA02, and by the special research project SFB 501 of the DFG ("Deutsche Forschungsgemeinschaft", German Research Council) for the construction of large software systems based on generic methods. Many thanks go to Prof. Dieter Rombach for his feedback and his work for creating a comprehensive environment for experimental software education. We thank Thomas Schneider and our students Wolfgang Wagenbichler and Mathias Grund for supporting us, as well as our forerunners Dr. Antje von Knethen and Dr. Christiane Differding for their work in the building and elaboration of the course and the software system.

REFERENCES

[1] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. PuLSE: A Methodology to Develop Software Product Lines, in Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99), (Los Angeles, CA, USA), pp. 122-131, ACM, May 1999

[2] D. Campbell, J. Stanley, "Experimental and quasi-experimental designs for research", Houghton Mifflin Company, Boston, 1963

[3] J. Carver, L. Jaccheri, S. Morasca, and F. Shull, "Issues in using students in empirical studies in software engineering education", In Proceedings of 2003 International Symposium on Software Metrics (METRICS 2003), Sydney, Australia, September 2003, pp. 239-249.

[4] J. Carver, F. Shull, and V. Basili, "Observational studies to accelerate process experience in classroom studies: an evaluation," In Proceedings 2003 International Symposium on Empirical Software Engineering. ISESE 2003, Los Alamitos, CA, USA, 2003.

[5] M. Ciolkowski, O. Laitenberger, D. Rombach, F. Shull, and D. Perry, "Software inspections, reviews and walkthroughs", In Proceedings of the 24th International Conference on Software Engineering. ICSE 2002. pages 641-642, 2002.

[6] Fenton, N., and Pfleeger, S.: Software Metrics—A Rigorous and Practical Approach, Thompson Computer Press, 1996

[7] Grund, Mathias: Integration von Entwurfsmustern in ein existierendes Gebäudeautomationssystem. (in German) Project Thesis, Department of Computer Science, University of Kaiserslautern, 2000.

[8] Höst, M.; Regnell, B.; and Wohlin, C.: Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment, Journal of Empirical Software Engineering, November 2000.

[9] Jalote, P.: An Integrated Approach for Software Engineering. 2nd Edition, Springer Verlag, 1997

[10] D. Muthig. A Light-weight Approach Facilitating an Evolutionary Transition towards Software Product Lines, Ph.D. thesis, Ph.D. Theses Series in Experimental Software Engineering, Volume 11, Fraunhofer Verlag, 2002

[11] Muthig, D. and Bayer, J. Helping Small and Medium-Sized Enterprizes Moving Towards Software Product Lines, Workshop on Software Product Lines: Economics, Architectures, and Implications, ICSE, Limerick, 2000.

[12] DeRemer, F.; Kron, H.H.: Programming-in-the-large versus Programming-in-the-small. IEEE TSE, 1976

[13] Rombach, D. Fraunhofer: The German Model for Applied Research and Technology Transfer, in Proceedings of the 22nd International Conference on Software Engineering, Limerick, 2000.

[14] D. Weiss and C. Lai. Software Product Line Engineering - A Family-Based Software Development Process, Addison-Wesley, 1999