Using Empirical Knowledge for Software Process Simulation: A Practical Example

Ove Armbrust

7. Mai 2003

Fraunhofer IESE
Sauerwiesen 6
67661 Kaiserslautern

Diplomarbeit

Using Empirical Knowledge for Software Process Simulation: A Practical Example

Ove Armbrust*

Mai 2003

Betreuer: Prof. Dr. H. Dieter Rombach†

Dr. Jürgen Münch‡

* Ove.Armbrust@web.de

† rombach@informatik.uni-kl.de

‡ muench@iese.fhg.de

Erklärung

Hiermit erkläre ich, Ove Armbrust, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kaiserslautern, den 7. Mai 2003

Ove Armbrust

ABSTRACT

This thesis describes the combination of real experiments with software process simulation. It introduces different types of real experiments and simulations and describes three main possibilities of combining them: using empirical knowledge for simulation, using simulation for real experiments, and online simulations. The first possibility is explored further by the systematical development and calibration of an executable discrete-event simulation model of a requirements inspection process. The model supports an arbitrary number of documents, each inspected by an arbitrary number of reviewers.

The model was developed systematically by following an existing method [RNM03]. The following steps were conducted: the creation of a static process model, the collection and analysis of empirical data, the subsequent creation of an influence diagram, and the final creation and calibration of the dynamic model. The model was calibrated with data from a requirements inspection experiment and two replications.

Furthermore, the thesis explains model limitations and lessons learned, and gives a detailed description of how to use the model. A first model validation with real-world data that was not used for the model development suggests that the model indeed reflects reality decently: The deviation between simulation results and real-world data averages 3.3 percent. Conclusions on possible further usage of the simulation model and modeling results complete this work.

Keywords: Experimental software engineering, empirical studies, software process modeling, software process simulation, software measurement

Table of Contents

1 Introduction

1.1 Motivation

Today's software industry faces various challenges. Systems grow more complex while at the same time, quality demands rise. Furthermore, development time as well as costs should be reduced. To cope with this, new technologies are constantly being developed. One approach to studying the discipline of software engineering is the empirical studies approach [RBS93]. It focuses on finding out about strengths and weaknesses of techniques, methods and tools. The goal is to provide software development projects with a specifically tailored set of techniques, methods and tools, so that an appropriate result can be achieved. Experimenting is one way to gather empirical findings.

Experiments, however, are expensive undertakings. Software engineering is a human-based profession, since all major development steps are performed by humans, using their creativity and knowledge. Thus, experimenting in this field naturally comprises numerous humans, which makes it expensive. Calculations of the costs of a software professional start at $500 per person per day [BGL+96]. To get statistically significant results, a high number of experimental subjects is preferred. In addition to the costs for these subjects, the costs for experimental setup, conduction and subsequent data analysis must be considered.

The enormous costs of experimentation have been noticed by other sciences years ago. Consequently, simulations were developed to save some of these costs. This was partially simplified by the fact that some problem areas do not rely on human performance as heavily as software engineering. In car manufacturing, for example, many real experiments have at least been partially replaced by simulations. Thanks to reducing the number of real experiments and increasing simulation usage, e.g., for parts strain, the German car manufacturer BMW could reduce the development time of a new engine from 72 months to about 36 months [Fli02], thus saving enormous costs.

Simulation is also becoming increasingly popular in the software engineering community due to the pressure of lowering costs while at the same time increasing quality and development speed. Using simulation technology lowers costs because it reduces the number of real experiments, and at the same time allows variations of the (virtual) experiment at almost no additional cost. Simulations can be used for technology evaluation, decision making in project management, as well as controlling and training,

among other fields. Hence, simulation might become an important factor in the future of software engineering.

One of the key problems of simulation is to create accurate models reflecting real world behavior for specific contexts. Correctly mapping the real world to the model, executing the simulation, and mapping the results back to the real world is crucial to the success of the modeling and simulation effort. Integrating empirical knowledge into simulation is an important approach to achieve this goal, because it allows for the creation of a better image of the real world and a closer connection of real and virtual world. Nevertheless, many existing simulation models are purely based on hypothetic assumptions that have not been proven in real practice.

This thesis presents an example for the development of a simulation model by using empirical knowledge, and describes the experience with the modeling process. The goal is to demonstrate how empirical knowledge can be effectively used for simulation modeling, and to highlight typical challenges, difficulties, and limitations. The simulation model created here is fully operational, and can be adapted easily to different contexts. A practical example is included.

1.2 Thesis Goals

This thesis pursues several goals:

1. It wants to give an introduction on empirical software engineering. Different types of real and virtual experiments are introduced and structured.

2. It will point out the main possibilities of combining empirical studies and software process modeling and simulation. This emphasizes on the diversity of the combination possibilities.

3. An exemplary model is developed for the case of using empirical knowledge for simulations. This includes all modeling steps such as deriving necessary data from real world data sources, determining influences, and creating and calibrating the model, with the purpose of pointing out typical problems and possible solutions.

4. Based on a methodology developed by Rus et al. [RNM03], a systematic guide to support the usage of empirical knowledge for software process modeling will be given. This should simplify creating other models.

5. Remaining open questions are described to point out possible fields of future work.

1.3 Structure of the Thesis

Chapter 2 describes the basic ideas of experimental software engineering, real and virtual experiments, and their relations. The basic ideas of experimental software engineering are described in Section 2.1. The most commonly used types of real experiments are introduced in Section 2.2. It is obvious that these experiments are expensive in real life, so new ways to reduce costs are currently being researched. One of them are simulations or virtual experiments as described in Section 2.3. Section 2.4 gives an introduction on the area of software process modeling.

Chapter 3 introduces the combination of real and virtual experiments. Section 3.1 gives an overview over this subject. Three basic variations can be distinguished: using empirical knowledge for simulation purposes as described in Section 3.2, using simulation for real experiments (Section 3.3), and online simulations (Section 3.4). Section 3.5 articulates typical problems.

Chapter 4 describes the simulation model development. This was done according to the methodology by Rus et al. [RNM03]. The steps taken towards the model are described in Sections 4.2 to 4.6. The model itself, model calibration and some extensions towards a more practically usable model are explained in Sections 4.7 to 4.10. Section 4.11 gives an overview of used empirical knowledge. Model limitations and lessons learned are presented in Sections 4.12 and 4.13.

Chapter 5 describes an exemplary simulation run with data from the real experiments, which illustrates the model. The scope of validity of the model, preparation steps and simulation results are explained in detail. Chapter 6 validates the model with data from another real-world experiment. Chapter 7 summarizes the findings, describes usage possibilities of the simulation results, and gives an outlook on future work.

2 Background

2.1 Experimental Software Engineering

The first software systems were mostly developed by hardware experts. This was the case because in the beginning of the software era, most software was hidden inside hardware, and fulfilled its duty there as primitive firmware or similar. The first software controlled industrial machines, for example sustaining a constant pressure in a reaction chamber or the like. Only very simple tasks were solved by software, consequentially software was not complex, but consisted only of a very limited number of LOC[1]. Ensuring the correctness and reliability of this code was relatively easy and was done by the corresponding hardware engineer.

Another application domain of software was solving mathematical problems. Algorithms were translated into executable programs, which in turn solved equations or the like. Proving the correctness of this code was also possible, since in most cases, a mathematician translated an algorithm into computer lingua. Well-known mathematical methods could be used for verification purposes.

As hardware systems grew more complicated, software also became more complex. Teams of several people were involved in software development, introducing more defects into the system. Soon, the software became the critical part of the system. Today, even a toaster contains several thousand LOC to get the toast exactly the right amount of heat. At the other end of the scale, software is controlling large parts of everyday life. Many of the amenities are only made possible by software, for example air conditioning, entertainment (TVs, radios, video games, shows), the ability to pick up the telephone receiver and call anyone anywhere on the globe, and other areas. Software also controls critical systems like air traffic control or military weapon systems. Demands for correctness and reliability are much higher here, since (software) failures can have extreme consequences.

In 1968, the software crisis was officially stated for the first time at the NATO conference in Garmisch-Partenkirchen. New ways other than the unsystematic development process used at that time had to be found, especially in the field of defect detection and prevention: Software engineering was born.

The new engineering approach proposed a very systematic way of developing software. Traditional engineering disciplines like construction or

[1] LOC = Lines of code

mechanical engineering demonstrated both the need for and the benefits of a systematic process, especially when larger projects were to be completed successfully: When building a hut in the back yard, no architect is needed for a decent result. When building a skyscraper, starting without specialists for materials, static and construction will inevitably result in a complete failure. The same is true for software projects. Writing a simple program, e.g., for the calculation of Fibonacci numbers, can be done by a single person rather quickly. When it comes to software for bank account management, specialists are needed, because failure is not an option, and the subject is a very complex one. The (in other areas) well-known engineering principles help immensely in achieving the time, quality and cost goals [Som87].

Since 1968, many efforts have been made in the field of software engineering. They all have in common the main goal: To develop software in a controlled manner so that quality, costs and time needed can be forecasted with high precision. Other goals include the overall reduction of development time and costs and a general increase in product quality. As of today, not one single best approach can be identified to achieve these goals. Following Rombach et al. [RBS93], three different approaches to study the discipline of software engineering can be identified:

- the mathematical or formal methods approach,
- the system building approach,
- and the empirical studies approach.

While the mathematical or formal methods approach aims at finding better formal methods and languages, and understands software development as a mathematical transformation process, the system building approach concentrates on finding better methods for structuring large systems. Here, software development is understood as a creative process, which is controlled through constraints on the resulting products. The empirical studies approach focuses on finding out about strengths and weaknesses of techniques, methods and tools. The goal is to provide every software development project with a specifically tailored set of techniques, methods and tools, so an appropriate result can be achieved. Experimenting is one way to gather empirical data.

Regarding the empirical approach, the word “empirical” requires special attention. Derived from the Greek “empeirikós“ (from “émpeiros“ = experienced, skilful), it means “drawn from experience”. This suggests that data from the real world is analyzed in order to understand the connections and interactions expressed in the measured data. This is an important point about the empirical approach: The data must be measured somehow. Measuring empirical data is not trivial, on the contrary: Many wrong decisions are based on incorrectly measured or interpreted data.

A good example is the infamous Lufthansa accident at the Warsaw airport in 1993. An Airbus A320 did not stop after landing, but raced over the end of the runway and collided with an earth bank, resulting in two dead people and a destroyed airplane: The flight computer refused to activate any brake system (air brakes, thrust reverse, wheel brakes) for nine seconds after touchdown. The primary reason for this was the incorrectly registered plane status. By measuring the revolutions of the wheels, the flight computer determined whether the plane was flying in the air or not. While being in the air, brakes (including thrust reverse!) can obviously not be activated. When the plane landed in Warsaw, there was heavy rain, resulting in a water film sitting on the runway. An effect known as "aquaplaning" prevented the wheels from spinning. Consequentially, the flight computer refused to activate the brake systems, which led to the catastrophe [Lad94].

What had happened? The software developers believed that by measuring whether the wheels spin or not, they could find out whether the plane does already have contact to the runway, so brake systems can be activated. Actually, they only measured whether the wheels were spinning or not. The assumed connection "wheels spinning ↔ plane on ground" was not correct. This makes clear that although measuring may be easy sometimes, interpreting the measured data may be significantly harder. Of course, measurement itself may also be hard: How can "usability" or "user friendliness" be measured?

The aim of the empirical approach consists of two parts: Measuring the real world correctly (with correct interpretation of the data measured) and coming to the right conclusions from the data measured. This concerns selection and usage of techniques, tools and methods for software development as well as activities like resource allocation, staffing and time planning.

At this point, the experimental part gets involved. In order to improve software development performance, experimenting with changes to the current situation needs to be done. Generally, this is achieved by determining the current situation, then changing some parameters and evaluating the results. The problem often is the context, which is hard to record and different with every project. Therefore, evaluating a new technique over several projects is difficult due to the different contexts. Another point is that one approach may work differently in different contexts, so whatever is proven by one experiment must not necessarily be true for other contexts [Jos88].

Experimenting, however, is a rather expensive way of gathering knowledge about a certain technique or tool. The cost can be divided into two main parts. First, the experiment itself must be conducted, that is, the experimental setup must be determined, the experiment needs to be prepared and carried out, the data must be analyzed and the results evaluated. These are the obvious costs, which can be measured easily.

The second part consists of the risk of delaying the delivery of the product. If a new technique is tested in a formal experiment before it is used in production, this ensures that no immature or unsuitable technique is used in real life. On the other hand, since software development is human-based, human subjects are needed for the experiment. During the experiment, they cannot carry out their normal tasks, which may result in delays, especially in time-critical projects. Competitors not carrying out experiments may be able to deliver earlier. These costs are not easily to be calculated. Still, using immature or unsuitable techniques is more expensive in most cases in terms of product quality, delivery time and person hours.

In other engineering sciences, these costs have been known for quite some time, and companies naturally wanted to minimize them. One approach is simulation. The engine development time at BMW, for example, has been reduced from 72 months to about 36 months [Fli02]. Engineers used to build five prototypes of a car engine before mass production started. Under the pressure of reducing costs, this has been reduced to two prototypes, where the second one is merely a proof that the concepts work. This could only be achieved by making extensive use of simulation. Most of the characteristics of a modern car engine are calculated and simulated on software systems and then only verified by the prototype.

A major difference between traditional engineering sciences and software engineering is that in software engineering, there is no production. After a car engine is developed, it is built several hundred thousand times, always the exact same way. Therefore, simulations were introduced in this area first. Production processes were simulated first, later development processes followed.

The software industry is under the same pressure right now as the producing industry has been 20 years ago. Customers demand higher-quality systems in less time for less money. Instead of time-consuming experiments, simulation can be used as in other engineering sciences. Using simulation, possible changes to the development process or utilized tools may be evaluated with comparatively low costs and without any risk for the current product. The results of the simulation need to be evaluated analog to the engine construction example, of course. Still, savings in time and costs are significant.

When looking at experimentation at a larger scale, other goals appear. Humanity has been experimenting ever since: interbreeding plants or animals, developing new technologies, or cooking new meals. All this can be seen as experiments. In the end, humankind learned from it. Today, nobody asks how much money the first Diesel engine meant for its inventor. It is just still being used, in a very refined form, and still being improved. The experiments of Rudolf Diesel led to a new type of engine. Over time, many engineers altered many things and observed the results, they learned from their experiments with the engine.

The important point is: Whichever experiment is conducted, people should learn something from it. This learning should be organized in some form, to enable efficient learning and to prevent that findings get lost over time and have to be re-discovered. One such approach specifically for software engineering is the Quality Improvement Paradigm (QIP) [BW84] developed in the 1980s by Victor Basili and Dieter Rombach at NASA SEL[2]. One QIP circle consists of six steps: *Characterize* the project (i.e., describe the context), *set goals* which should be achieved, *choose models* for measurement, *execute* the project, then *analyze* the data acquired, and finally *package* the information gathered during the analysis step, e.g., in the form of lessons learned, new/altered models or processes.

One learning scenario based on QIP is the following: First, the problem(s) must be identified. The actual state is measured and then an experiment is conducted with some parameters altered to identify where to change about the process. Once this is clear, the process is altered and again the actual state is measured. The experience learned from the experiment and the altered process is stored, and the (new) process is examined for further problems, thus starting a new cycle.

The storage of information gained in the execution phase is important: Information that cannot be retrieved and used is not worth anything. Therefore, a storage solution must be found which allows easy storing and finding of information. One such approach is the Experience Factory (EF) [BCR94a]. While QIP describes a methodological model for advanced learning in software organizations, EF provides an organizational model for the storage of knowledge. It consists mainly of one experience database and many project databases. The experience database contains process models, product models, project plans and other packaged information. These are gained from various projects, and stored with their context. Each project database contains "raw" products and measurement data. After project completion, this knowledge is analyzed, packaged and added to the experience database.

To use the knowledge earned in former projects, the experience database can be queried during project setup. It provides appropriate models, cost/time/staff estimates and so on, which can be used for project planning. When the project is running, new measurement data is acquired and stored in the project database. In the end, this contributes to the knowledge already in the experience database. So every single project can be seen as an experiment and an opportunity to learn.

Of course, a lot of research is also being done at universities or research institutes. New technologies are developed, examined and refined. There is, however, a significant gap between state-of-the-art as in universities and

[2] Software Engineering Lab

state-of-the-practice in companies. This is partially so because companies refrain from using new and (in a productive environment) untested technologies, but there is also a lack of communication. Research in universities is often very academic with no immediate goal of practical application. Companies, however, prefer specific help with certain problems. This gap is not easy to cross. The Fraunhofer Gesellschaft tries to build a bridge between academic research and industrial needs.

This thesis focuses on the use of simulation in software development processes, the combination with empirical studies performed in the real world, and demonstrates essential benefits of such a combination using an example.

2.2 Real Experiments

2.2.1 Overview

One of the key elements of empirical software engineering is the proposal of new models, e.g., for costs or product quality. For example, implementing a new reading technique in the software development process will most probably influence the time needed for defect detection and possibly correction, as well as the costs for these activities. To reflect this in an appropriate model, three possible ways to come to a new model can be distinguished. Either an existing model can be adapted, or a new model is introduced based on theoretical consideration, or a new model may be derived from observation. There are also combinations of two or all three ways.

When a new model is introduced, adequate measures must be formulated, so that the behavior of the model (compared to reality) can be evaluated. In the example, this would typically be efficiency (e.g., the number of defects found per hour) and effectiveness (e.g., the total number of defects found). In a first approach, this measurement is done mostly through experiments.

The question is usually whether the model corresponds to reality, and if not, how and at which points to adapt it. This can be done with experiments. They are called “real” experiments (as opposed to “virtual”) because they feature real people spending real time for solving (possibly constructed) problems in a real-world environment. This costs time and money, because the people taking part in the experiment cannot do their normal work. This makes clear one characteristic feature of software engineering: Human involvement. Unlike other sciences like chemistry or physics, software engineering experiments heavily involve humans, and are in turn heavily influenced by human behavior. The psychological aspect is similarly important in software engineering as in typical social sciences like anthropology, political science, psychology and sociology.

The purpose of an experiment is usually the proof of a hypothesis. A hypothesis is a supposed connection between a number of (controlled) input variables and a certain result. The experiment should either prove the correctness of the hypothesis or show that it is wrong. A null hypothesis symbolizes the opposite state to that suggested in a hypothesis, postulated in the hope of rejecting its form and therefore proving the hypothesis. For example, if the hypothesis states "Reading technique A is more efficient than reading technique B", then the null hypothesis would be "Reading technique B is more efficient or equal to reading technique A".

One commonly used classification distinguishes the number of experimental treatments and teams [RBS93]. Table 2.1 gives an overview over the possible combinations.

number of teams per treatment	number of treatments	
	1	**m**
1	1:1 (Case study)	1:m
n	n:1 (Replicated experiment)	n:m (Lab experiment)

Table 2.1: Classification by number of treatments and teams

A 1:1 situation does not provide reliable information for comparisons. The results may completely depend on the team members that were assigned to the experiment (experimental subjects), or on the task the team was supposed to deal with (experimental object). There is no way to know how much each of the two extremes influenced the result.

For example, if a team performs very badly in a defect detection experiment, this may be because team competence in this specific area is low, or because the reading technique used was not appropriate, or a combination of both. Without further information, there is no way to tell. 1:1 settings may be used as a proof of concept, though, or to check in which areas current processes may be improved. Case studies usually are conducted as a 1:1 experiment type.

A 1:m situation already enables the experimenter to have one team solve several problems. This enables him to evaluate whether team performance is caused by a certain method (the team performs poorly using one method, but well using another) or by the team itself (the team performs poorly using any method). Still, the methods may all be bad, thus making the team perform badly although it is not a bad team per se. Additionally, it is difficult to exclude learning effects throughout the treatments.

Using more than one team makes experiments more expensive. Having n teams replicate a single experiment (n:1 setting) enables the experimenter to assess team performance as well as an average for the method examined

in the treatment. Still, this does not allow for comparing different approaches. This approach is often chosen to prove the validity of findings or to examine the benefits of a new method or technique in greater detail. The experiment is set up and conducted by one team, and the replicated by other teams to validate the results.

The only approach for comparing different solutions for a problem in one experiment is the laboratory experiment. Using a n:m setting, several teams perform several tasks. This provides information about how the methods compare (relatively) and how the respective team influenced the results. Unfortunately, this type of experiment is the most expensive one.

A n:m setting requires an amount of control that can only be reached in laboratory environments. Conducting a n:m experiment in an industrial setting would increase project costs extremely, so most controlled experiments can be found in facilities like universities or research institutes. Because of the high costs and the number of variables monitored, controlled experiments are mostly focused very narrowly. Table 2.2 gives an overview over the real experiments described in this chapter and some of their characteristic factors.

Three of the most commonly used experiment types will be introduced in short: controlled experiments, case studies and surveys. Controlled experiments and case studies may be used to verify a newly introduced model very well, whereas surveys usually help in determining the current state-of-practice and major causes of undesired effects [Rom01].

The three experiment types differ in the types of variables that are considered. An overview is given in Figure 2.1. Dependent variables are influenced by independent variables and therefore not considered separately. There are two types of independent variables: controlled ones and not controlled ones. Typically, the experimenter aims at lowering the number of controlled variables to save effort, and to choose the “right” ones for the respective context. The not controlled variables are to be kept as constant as possible, and also as representative for the respective context as possible. This prevents uneven influence on the experiment and describes the context well.

Factor	Survey	Case study	Experiment
Execution control	None	Low	High
Investigation cost	Low	Medium	High
Ease of replication	High	Low	High

Table 2.2: Characteristic factors for real experiments after [WSH+00]

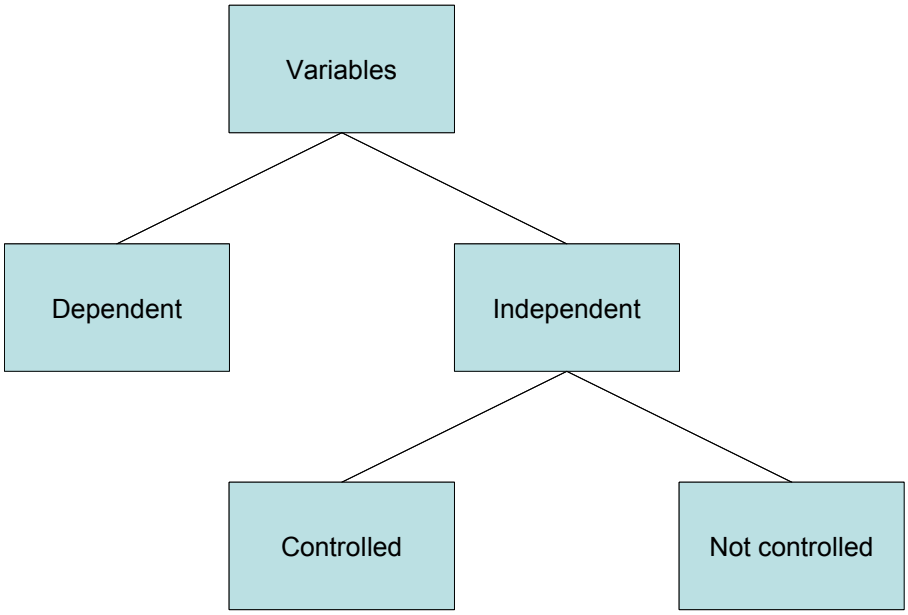


Figure 2.1: Variable types in experiments

2.2.2 Controlled Experiments – Research in the Small

A controlled experiment aims at completely controlling the experimental environment. In the field of software engineering, this includes the number and expertise of the test persons, the tasks they are given, but also variables like at which time of the day the experiments takes place, or room temperatures. Because of this high level of control, controlled experiments are also called laboratory experiments. The tasks examined in the experiment vary from real problems from earlier software projects to completely imagined situations.

If a new defect detection method were checked for efficiency, for example, it would be a good idea to use real documents from an older project. By doing so, two benefits can be achieved: First, an average real world problem is present. This ensures that the results are not only some academic insights, but do help with everyday work. Second, comparing the new method against the currently used one is easy. The number and severity of errors remaining in the documents after defect detection with the new method may be weighed against the time consumed, and so the method may be compared against the "traditional" one. This of course only works if the test persons do not know the test data because they were part of the project, because otherwise, they might not really detect the defects, but remember them.

If test subject knowledge or project confidentiality denies the usage of real data as experimental material, then artificial data must be used. This could

be specially created documents or a complete training system, which is developed and maintained exclusively for experimentation purposes. Still, help with a problem that really exists is the purpose of the experiment, so there should at least be some relation to actual development, to ensure that the results are not purely academic.

Made-up problems may also be used for comparing several new methods. Here, the problem may be specially tailored to the methods, focusing on areas with special interest. In our example, if project experience has shown a great density of interface problems, documents with a complex interface structure could be artificially created to test the new methods very intensively in this area. Creating a new scenario which has never happened before, but could happen someday and possibly have very severe consequences could be a third area where to use this kind of experiment.

Conducting a controlled experiment, however, requires significant amounts of both time and money. In most cases, a n:m setting as described in Section 2.2 is used. Due to this costly approach and the amount and complexity of the data acquired, only small portions of the software development process are analyzed this way. Examples might be probing different reading techniques in a defect detection process or different programming languages for solving specific problems.

2.2.3 Case Studies – Research in the Typical

A case study does not try to control all variables, but still aims at identifying the significant ones before the project starts. Once this is done, their status is recorded and monitored throughout the study, in order to identify their influence on the results. This makes case studies suitable for industrial use, because projects are only influenced on a minor scale: The number of values measured and project documentation (other than product documentation) may increase slightly, but this is only a minor “add-on” and does not require enormous amounts of extra time and money.

Most commonly recorded variables are inputs, constraints and resources (technical and human). Most of the time, a single actual software project is examined. Usually, only one team per task is scheduled, so the 1:1 setting is typical for a case study. The same applies for the experiment data classification: Only real-life data can be used.

The case study approach can be found far more often in industry than the controlled experiment. This is because case studies can be conducted accompanying the real project without enormous extra expenses. A sensible approach for process changes would be to alter a little part of the development process, small enough to prevent great problems for the overall process, but big enough to notice possible time and money savings, and then conduct a case study for a project with the new process. There is still the

danger of confusing team influence and process influence, but in software organizations, it is possible to estimate team performance from other projects, thus extracting process influence fairly accurate.

Because case studies do only pay special attention to factors typical for the current situation, instead of controlling the complete environment, they are quite popular. Plenty of knowledge about project progression may be achieved at relatively low costs in a realistic setting.

2.2.4 Surveys – Research in the Large

In contrast to case studies where monitored variables are defined before the project starts, surveys are retrospective studies: The occurrence that is to be examined has already taken place [WSH+00]. This means that typically the survey is planned and conducted after the project is finished. The focus of surveys lies on larger-scale relations and not so much on details. The experimenter tries to collect as much data as he can find, evaluate it and determine connections between the discovered factors and the results. An analogy would be an assessment in a company: There is no influence of the survey on the project and the data available. The experimenter can only use whatever data were recorded throughout the project. Examples for surveys can be found in [Lur02] and [Mey02].

This makes clear one great danger of surveys: If a survey states (from comparisons of past projects) that a certain factor combination has a significant effect on the project outcome, this may be an incorrect assumption. The observed effect may not be caused by the formulated factor combination, but by another reason that was not discovered because it was not recorded during the project.

Of course, this danger does not only exist in surveys, but also in laboratory experiments and case studies, where the researcher has the possibility to set the monitored variables. The difference is that in laboratory experiments and case studies, he can use his experience to minimize the risks of mistakes such as not measuring important variables. A normal project usually is not explicitly set up to later support a survey, but to achieve the desired goals at minimum costs. Hence, the risk that important variable data are missing is greater in surveys than in laboratory experiments and case studies.

The results of a survey are usually not as detailed and “hard” as results from a case study or a controlled experiment. If a significant cause for a certain undesired effect was (supposedly) detected, a more thorough investigation may be undertaken to refine the findings. This may be done in form of a case study or a formal experiment, depending on the situation. A case study may be conducted if a strong indication exists that a certain factor influences the result significantly, and there is a proposal to improve

that factor with only a limited risk of failure. If there are several improvement proposals, or if it is not even sure which factor(s) are to be altered, a formal experiment may help reducing the risk for everyday business, at higher experimentation costs.

2.2.5 Experiment Sequences

One of the biggest problems with experiments is comparing and evaluating the results. This is easy to understand when looking at surveys or case studies, because here, the context can only be described, not changed. Nevertheless, why is this also difficult with controlled experiments? Wasn't one of their advantages the complete control of all variables, including the context?

Controlling all variables makes the results very valid – in the respective context. Example: Fagan argues that most defects would be detected during team meetings [Fag76], whereas McCarthy et al. [MPSV96] and Votta [Vot93] come to the contrary result. The context is very different in all three cases. So which study is "correct" and which one not? This question cannot be answered, because in their respective contexts, all three are correct.

Recording the context with every study is very important to evaluate the outcome. However, to come to a more comprehensive view of the software development process, all the different studies must be combined somehow. Their "essence" must be extracted and combined somehow. This can be done by regarding all experiments in one area (requirements specification, defect detection,…) as a sequence of experiments. Common variables and phenomena may then be detected and included in the global model.

Experiment sequences may be planned or unplanned. A planned experiment sequence usually consists of a number of teams conducting a number of experiments. An unplanned sequence collects experimental data from many experiments and extracts commonalities and differences. In most cases, the majority of the teams will take part only in one experiment, and the experiments usually examine similar phenomena.

An example for an unplanned experiment sequence can be found in [Arm02]. Here, the author examined a large number of experiments (surveys, case studies and laboratory experiments) and extracted factors that influence the progression and results of software inspections. The factors form a complex network of interrelations. Only very few factors were found in every experiment that was reviewed. The combination of the experiments showed new correlations that would not have been obvious when regarding every experiment individually.

The question of the context, however, is still not answered satisfactory. How can factors be included that significantly influence the process only sometimes? Here, for example a model of the software development process would need to be tailored to the specific situation. Context information is mandatory and must be individually determined for each situation. Depending on the context, some factors may be left out at some point, while others are added. More research is needed here to determine when to include what.

A planned experiment sequence, on the other hand, might consist of several teams carrying out several tasks. For the purpose of evaluating a word processor, these tasks might be writing a 50-page thesis with few figures, writing a 5-page report with lots of formulae, and writing a 1-page application for a scholarship. One possible experimental setup is depicted in Table 2.3. By shuffling teams and tasks, both team performance and tool capability can be evaluated.

Unplanned experiment sequences may be used when findings from isolated experiments are to be generalized and supported by a broader foundation. Analyzing many experiments concerning similar phenomena can lead to new conclusions and explanations, rather than a single experiment. In addition, case studies and surveys are often too hard to coordinate with each other to form a planned experiment sequence.

Planned experiment sequences may be used to break up tasks too complex to be investigated in a single n:m laboratory approach. By conducting an experiment sequence, the researcher may examine a complex phenomenon over several years while not losing his focus.

Team	Task		
	Thesis	Report	Application
1	X	X	
2	X		X
3		X	X

Table 2.3: Exemplary setup for a planned experiment sequence

2.2.6 Benefits

Following Basili et al. [BSH86], benefits in the following areas can be expected from experiments in software engineering:

- *Basis.* Experiments provide a basis for the needed advancement in knowledge and understanding. Through experiments, the nature of the software development process can be assessed and evaluated.

- *Process and product effects.* Experiments help to evaluate, predict, understand, control and improve both software development process and product.

- *Model and hypothesis development and refinement.* In iterations, models are built and hypotheses about the models postulated. These hypotheses can be tested by experiments and then be refined, or new models can be constructed.

- *Systematic learning.* In a rather young profession such as software engineering, where there is still very much to learn, it is important to follow a systematic approach, rather than intuition.

2.3 Virtual Experiments

2.3.1 Overview

To overcome the limitations and problems with real experiments, many professions turned to virtual experiments. While in the 1960s or 70s, nuclear weapon tests were a rather usual phenomenon, this has ceased (almost) completely in the new millennium. In part, this reduction has been caused by the increased knowledge about the harmful side effects of nuclear weapons (namely radiation and contamination of vast areas for thousands of years), but not completely. For more information about human radiation experiments, please refer to [oE94].

Since the military has priorities other than preventing ecological damage, but wide parts of human society was not willing to accept that, other ways for testing these weapons had to be found. Today, (at least some rich western) nations have the ability to simulate nuclear explosions. This saves enormous expenses, because real testing is always destructive and nuclear weapons are not exactly cheap. In addition, simulations are not getting society upset. Another great advantage is that simulations can be repeated as often as wished with any set of parameters, which is impossible with real tests.

Other areas where simulation is used amply are mechanical engineering or construction. Here, simulation helps saving cost and time. As a rather young profession, software engineering has only recently started to discover and use the benefits of simulation. Empirical software engineering yet is not very common in industry. Many decisions are still based on intuition rather than measured data. This trial-and-error approach is very expensive and delivers lower-quality products than more systematic approaches would do. When implemented, experiments already yield good results, for example in choosing a reading technique for inspections.

Still, the required experiments cost a lot of money, which companies naturally dislike, despite the benefits. But as in other professions, simulation or virtual experimentation can also help with that in software engineering. A virtual experiment is conducted in two basic parts: modeling the real-world situation, and afterwards simulating it on a computer. The model tries to reproduce some aspects of the real world as accurately as needed, in a representation that can be used in the simulation. The optimal model would contain exactly the entities and relations needed for the simulation, nothing more and nothing less.

An entity represents an element of the real world, e.g., a person or a document. Like their real counterparts, entities interact with each other; this is mapped to relations in the model. One problem in modeling is that it is usually not clear which entities and relations of the real world need to be modeled. This depends on the scope of the model: If the goal is the prediction of product quality, other factors must be included than when cost optimization is aimed at. In any case, including too many factors increases model complexity unnecessarily and may even influence results, whereas considering too few factors may render the results unreliable.

Working with simulations can be seen as research in a virtual lab (Figure 2.2). The conventional software engineering laboratories explore the research object by using various empirical studies, e.g., controlled experiments or case studies. The virtual laboratory examines its research object with simulations. Transferring information from one to the other can help improve either research results, as shown in Chapter 3.

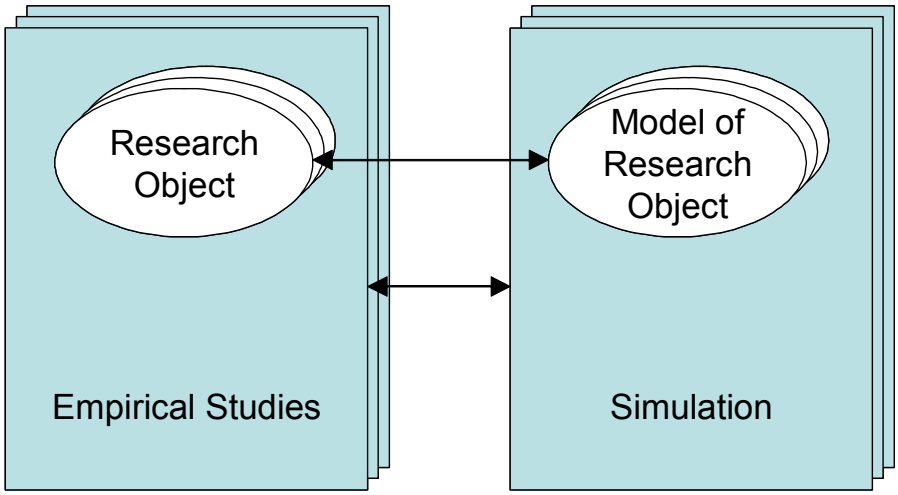


Figure 2.2: Real and virtual laboratory approach based on [Hag03]

Simulations come in two main variations: discrete and continuous. In discrete event simulations, variables can change their value only at discrete moments of time, as opposed to continuous simulations, where their values change continuously with time. A hybrid approach tries to combine the two, to overcome the specific disadvantages. Any simulation helps decreasing experiment costs by reducing the amount of people needed to conduct the experiment and by speeding up the process faster than real-time.

Simulation supports any of the three types of experiment regarded earlier, each in a different way. Once the simulation model has been built and verified, laboratory experiments can at least partially be replaced, for example to determine the consequences of (gentle) context changes. Case studies focus on monitoring real-life variables, so completely replacing them by simulations would not make sense. Simulations may still be useful for identifying variables that should be monitored. A simulation of an altered process can reveal variables that are likely to change significantly, so the case study can pay special attention to them. Regarding the other direction of information flow, case studies can deliver valuable data for calibrating the simulation model. The authentic nature of the data (they are taken directly from a real project) also helps refining the model.

Simulation does not seem very useful replacing surveys, since surveys are optimized for finding relations in real-life processes. Replacing surveys of real processes with surveys of modeled processes would not save time or money. Actually, surveys can supply information for simulation models, like case studies: In the retrospective, problems often become obvious that were invisible during the project. A survey reveals them in many cases.

Let’s say that a certain product was delivered significantly later than planned, and that this phenomenon did not occur for the first time. A sur-

vey reveals that the customer changed the initial specifications of the software several times in cooperation with customer service. However, it took customer service several weeks to communicate these changes to the development team, thus already delaying the necessary rework and, in addition, making it more complex by forcing the developers to change a more mature piece of software.

The survey reveals this lack of communication within the software development process, and modeling that part of the process could help understanding and solving the problem. Once the model has been built, the data from the survey can be used to calibrate it. Process changes can then be simulated before realizing them in the actual process. This indicates that surveys and simulation should be used complementary, not alternatively.

2.3.2 Simulation in Software Engineering

Simulation is increasingly being used in software development processes. After ignoring it for a long time, software engineers are beginning to realize the benefits of simulation. These benefits may be found in the areas of strategic management of software development, operational process improvement or training situations [KMR99]. According to the respective purpose, the simulation focus ranges from lifecycle modeling (for long-term strategic simulation) to small parts of development, for example a code inspection (for training purposes).

Due to the ever-increasing demands to software (better, faster, cheaper), software development has changed a lot during the past 20 years. Small programs written by single persons have evolved into gigantic “code monsters” developed and maintained by hundreds or thousands of people. In addition, software fulfils more functions every day, thus increasing complexity even more. To cope with this, new techniques, methods and tools are being developed constantly. New programming paradigms and languages are introduced to improve the quality of software development and the resulting products.

There is, however, no silver bullet for software development. Each tool or programming paradigm has its own benefits and disadvantages. The hugely popular object-oriented programming, for example, makes constructing large systems possible at all and improves product quality in most cases. Nevertheless, engine control software at Bosch is still developed in a functional manner, because using object-oriented technology would only make the system more complex and slower.

This example makes it clear that there must be some kind of evaluation for new technologies. No engineer would build a bridge with new and untested material. The same applies to software engineers: Using a new pro-

gramming paradigm without testing it properly may result in a disaster, namely low product quality and time/cost overruns.

The first steps in investigating the possibilities and limitations of new technologies are usually taken by experimenting. With respect to this, software engineering is not any different from other engineering sciences. Once the properties of the examined technique/method/tool are clear, most engineering sciences take the next step: simulation. Simulation enables the experimenter to examine the test object in an environment different from the usual one with relatively little extra effort.

Software engineering has only recently started to discover the possibilities of simulation. In this section, the benefits of simulations will be discussed, as well as the question what to simulate. Two different simulation approaches will be introduced later in this chapter, as well as their combination.

Three main benefit classes of software development process simulation can be identified: cost, time and knowledge improvements. Cost improvements originate from the fact that conventional experiments are very costly. The people needed as experimental subjects are usually employees. This makes every experimentation hour expensive, since the subjects get paid while not immediately contributing to the company's earnings. Conducting a simulation instead of a real experiment saves the cost for experimental subjects.

Time benefits can be expected from the fact that simulations can be run at (almost) any desired speed. While an experiment with a new project management technique may take months, the simulation may be sped up almost arbitrarily by simply having simulation time pass faster than real time. On the other hand, simulation time may be slowed down arbitrarily. This is done in biological processes simulation, for example. It might be useful in a software engineering context when too much data is accumulated in too little time and therefore cannot be analyzed properly. While in the real world, decisions would have to be based on partial information, the simulation can be stopped and the data analyzed. When the analysis is complete, then the simulation can be continued with a decision based on the completely analyzed data.

Knowledge improvements stem mainly from two areas: Simulation can be used for training purposes and experiments can be replicated in different contexts. Training software engineers in project management, for example, requires a lot of time. The trainee needs to make mistakes to learn from, which in turn cost time and money in the project, and has to be instructed to prevent him from making real bad mistakes which would cost even more time and money. Training them in a laboratory setting is even worse. Using a simulation environment such as the one introduced in [Nav02] enables the trainee to experience immediate feedback to his decisions. The

consequences of choosing a certain reading technique, for example, do not occur months after the decision, but minutes. This way, the complex feedback loops can be better understood and mistakes be made without endangering everyday business.

Simulations can also be used to replicate an experiment in a different context, for example with less experienced subjects. If the properties of the experimental objects are sufficiently explored, the consequences of such changes can be examined in simulations instead of costly experiment replications. Learning from simulations can save both time and money, compared to real experiments.

Another useful application of simulation are high-risk process modifications. This may be (yet) uncommon processes, or catastrophe simulations. An example for the first is Extreme Programming, which seemed to contradict all software engineering principles of documentation and planning at first, but proved beneficial in certain situations after all. When high-risk process changes are to be examined, simulations can provide a sandbox in which the changes can be tried out without any consequences. If the results show the proposed changes to be a complete letdown, at least no money was spent on expensive experiments.

A catastrophe simulation can investigate extreme process changes, for example the loss of key persons in a project. While it is clear that this will influence the process and its outcome noticeably, determining quantitative effects can only be done in an experiment. Will it be 20% delayed, or 200%? What about product quality? Since this situation rarely occurs, this kind of real experiment is not conducted because of the high costs. A simulation does not have such high costs. It probably also does not forecast the exact numbers, but nevertheless, it shows their magnitude. It may also show key problem areas, which may then be addressed in real life, to absorb the worst consequences of the hypothetic catastrophe.

The following classification of simulations follows Kellner et al. [KMR99]. The authors have determined relationships between model purpose and what has to be modeled, as well as classified approaches how to simulate.

When a simulation is to be set up, the first question that is to be answered is for the purpose. What is the simulation to be used for? Is it for strategic planning, or for training, or operational process improvement? After having answered this question, it is possible to determine what to include in the simulation, and what to leave out. To structure this decision, the model scope, result variables, process abstraction and input parameters can be distinguished.

The model scope must be adapted to the model purpose. Let’s say a software company wants to test a new reading technique for defect detection.

Clearly, the inspection process must be modeled, but changes to the reading technique will most likely have other effects later in the development process. The defect density might be higher, therefore lowering product quality and increasing rework time. This must be considered in the model because otherwise, the impact of the process change is not reflected correctly in the model.

According to Kellner et al. [KMR99], the model scope usually reaches from a part of the lifecycle of a single product to long-term organization considerations. They introduced two sub-categories: time span and organizational breadth. Time span is proposed to be divided triply: Less than 12 months, 12–24 months, and more than 24 months. Organizational breadth considers the number of product/project teams: less than one, exactly one, or multiple teams involved.

The result variables are mandatory for answering the questions posed when determining the model purpose. Variables can be thought of as arbitrary information sources in an abstract sense here, however, most models include variables such as costs, effort, time consummation, staffing needs or throughputs. The choice of variables depends once again on the purpose of the model. If the purpose is to predict overall end-of-project effort, variables other than the ones needed for predictions at the end of every major development step must be measured.

Questions like this also influence the level of abstraction at which the model is settled. If effort at every major development step is to be determined, the model probably needs a finer granularity than if only overall end-of the project effort is the focus. In any case, it is important to identify key activities and objects (documents) as well as their relationships and feedback loops. In addition, other resources such as staff and hardware must be considered. Depending on the level of abstraction, this list gets more or less detailed.

Finally, input parameters must be determined. They depend largely on the desired output variables and the process abstraction. In general, many parameters are needed for software process simulation; the model by Abdel-Hamid and Madnick [AHM91] requires several hundred. Kellner et al. [KMR99] provide some examples: effort for design and code rework, defect detection efficiency, personnel capabilities,… Figure 2.3 illustrates the relationships among the aspects described above.

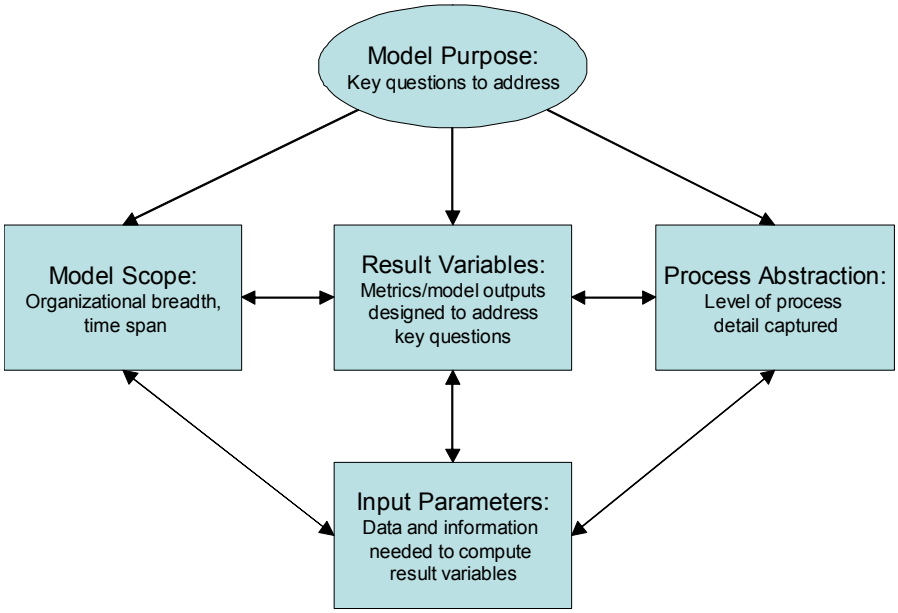


Figure 2.3: Relationships among model aspects after [KMR99]

After determining the purpose of the model and what to include, the choice of a simulation technique comes up. The continuous and the discrete simulation approach will be introduced in sections 2.3.3 and 2.3.4, as well as their combination (Section 2.3.5). In addition to that, there are several state- and rule-based approaches, as well as queuing models [1299].

Simulation techniques may be distinguished into visual and textual models. Most models support some kind of visual modeling today, to ease development of the models. Some modeling tools support semi-automatic validation functionality, e.g., Fraunhofer IESE's Spearmint. Appropriate possibilities to specify interrelationships among entities of the model enable accurate modeling of the real world. Continuous output of result variables as opposed to only presenting the results allows monitoring the simulation run and early intervention. Interactive simulations allow for altering input variables during execution.

Simulations can be entirely deterministic, entirely stochastic or a mix of both. Entirely deterministic models use input parameters as single values without variation. The simulation needs to be run only once to determine its results. A stochastic simulation assumes random numbers out of a given probability distribution as input parameters. This recognizes the inherent uncertainty in software development, especially when evaluating human performance. Consequentially, several runs are needed to achieve a stable result. Batch runs of simulations with changed input parameters simplify this approach significantly.

A sensitivity analysis explores the effects of input variable variations on the result variables. This has two advantages: First, the researcher knows how much variation in the results has to be expected due to variations in input parameters, and second, he can identify the parameters with the biggest influence on the result. These should be handled and examined with special care.

Finally, to obtain valid results from the simulation, calibrating the model against the real world is necessary. This should be done by integrating actually measured data into the model, and comparing the results of simulations of real processes with the respective real-world values. Accurate measuring of input parameters and result variables is mandatory here. In many cases, however, there is no suitable real-world data available. In this case, Kellner et al. [KMR99] suggest trying to construct the data needed from data available (effort → costs by assuming an average hourly rate) or retrieve it from original documents, rather than final reports, or obtain estimates from the personnel or published values.

2.3.3 Continuous Approach (System Dynamics)

In the early 1970s, the Club of Rome started an initiative to study the future of human activity on our planet. The initiative focused on five physical and easily measurable quantities: population, food production, industrial capital, production and non-renewable natural resources [Pfa01]. A research group at the Massachusetts Institute of Technology (MIT) developed a societal model of the world. This was the first continuous simulation model: The World Model.

Today, most continuous models are based on differential equations and/or iterations, which take several input variables for calculation and in turn supply output variables. The model itself consists of nodes connected through variables. The nodes may be instantaneous or non-instantaneous functions. Instantaneous functions present their output at the same time the input is available. Non-instantaneous functions, also called memory functions, take some time for their output to change.

This approach of a network of functions allows simulating real processes continuously. An analogy would be the construction of mathematical functions (add, subtract, multiply, integrate) with analog components like resistors, spools or condensers. Even complex functions containing partial differential equations that would be difficult or impossible to solve analytically or numerically may be modeled using the three basic components mentioned above. Before computers became as powerful as they are today, the analogue approach was the only way to solve this kind of equations in a reasonable time. Due to the continuous nature of the “solver”, the result could be measured instantly.

Of course, simulating this continuous system on a computer is not possible due to the digital technology used. To cope with this, the state of the system is computed at very short intervals, thereby forming a sufficiently correct illusion of continuity. This iterative re-calculating makes continuous models simulated on digital systems grow complex very fast.

In software engineering contexts, continuous simulation is used primarily for large-scale views of processes, like management of a complete development project, or strategic company management. Dynamic modeling enables us to model feedback loops, which are very numerous and complex in software projects.

2.3.4 Discrete Approach

The discrete approach shows parallels to clocked operations like car manufacturers use in their production. The basic assumption is that the modeled system changes its state only at discrete moments of time, as opposed to the continuous model. So, every discrete state of the model is characterized by a vector containing all variables, and each step corresponds to a change in the vector.

Example. Let's consider a production line at a car manufacturer. Simplified, there are parts going in on one side and cars coming out at the other side. The production itself is clocked: Each work unit has to be completed in a certain amount of time. When that time is over, the car-to-be is moved to the next position, where another work unit is applied. (In reality, the work objects move constantly at a very low speed. This is for commodity reasons and to realize a minimal time buffer. Logically, it is a clocked sequence.) This way, the car moves through the complete factory in discrete steps.

Simulating this behavior is easy with the discrete approach. Each time a move is completed, a snapshot is taken of all production units. In this snapshot, the state of all work units and products (cars) is recorded. At the next snapshot, all cars have moved to the next position. The real time that passes between two snapshots or simulation steps can be arbitrary. Usually the next snapshot of all variables is calculated and then the simulation assigns the respective values. Since the time needed in the factory for completion of a production step is known, the model describes reality appropriately.

A finer time grid is certainly possible: Instead of viewing every clock step as one simulation step, the arrival at a work position and the departure can be used, thereby capturing work and transport time independently.

The discrete approach is used in software engineering as well. One important area is experimental software engineering, e.g., concerning inspec-

tions. Here, a discrete simulation can be used to describe the process flow. Possible simulation steps might be start and completion of activities and lags, together with special events like (late) design changes. This enables discrete models to represent queues.

2.3.5 Hybrid Approach

Continuous simulation models describe the interaction between project factors well, but lack in representing discrete system steps. Discrete simulation models perform well in the case of discrete system steps, but make it difficult to describe feedback loops in the system.

To overcome the disadvantages of the two approaches, a hybrid approach as described by Martin and Raffo [MR00] can be used. In a software development project, information about available and used manpower is very important. While a continuous model can show how manpower usage levels vary over time, a discrete model can point out bottlenecks, for example inspections. Because of insufficient manpower, documents are not inspected near-time, so consumers of those documents have to wait idly, therefore wasting their manpower.

In a purely discrete approach, the number of inspection steps might be decreased to speed up the inspection process. While possibly eliminating the bottleneck, this probably introduces more defects. The discrete model would not notice this until late in the project, because continuously changing numbers are not supported. The hybrid approach, however, would instantly notice the increase, and – depending on how the model is used for project steering – more time would be allocated for rework, possibly to the extent that the savings in inspection are overcompensated. Thus, the hybrid approach helps in simulating the consequences of decisions more accurately than each of the two single approaches does.

2.3.6 Benefits

Following Kellner et al. [KMR99], benefits in the following areas can be expected from simulation in software engineering:

- *Strategic management issues.* These can be questions like whether to distribute work or to concentrate it in one spot, or whether development should be done in-house or be out-sourced, or whether to follow a product-line approach or not.

- *Planning.* Planning includes forecasting schedule, costs and product quality and staffing needs, as well as considering resource constraints and risk analysis.

- *Control and operational management.* Operational management comprises project tracking, oversight of key project parameters such as project status and resource consumption, comparing actual to planned values, as well as operational decisions such as whether to commence the next major phase (e.g., coding, integration testing).

- *Process improvement and technology adoption.* This includes evaluating and prioritizing suggested improvements before they are implemented as well as *ex post* comparisons of process changes against simulations of the unchanged process with actually observed data.

- *Understanding.* Simulation models can help process members to better understand process flow and the complex feedback loops usually found in software development processes. Also, properties pervasive through many processes can be identified.

- *Training and learning.* Similar to pilots training in flight simulators, trainees can learn project management with the help of simulations. The consequences of mistakes can be explored in a safe environment and experience can be collected that is equivalent to years of real-world experience.

2.4 Software Process Modeling

An area closely related to simulation is software process modeling. Modeling the software process is the logical step that has to be taken before a simulation can commence. The process model is converted into an executable form for the simulation. Thus, without a process model correctly describing the software process, no sensible simulation can be achieved.

To understand software process modeling, one must first understand how software is developed in the real world. On a very abstract level, an organization uses inputs and resources to produce output. The organization can be a manufacturing company, a knowledge company, or a research institute. Every organization owns certain resources, e.g., real estate, buildings, people (or their knowledge), or financial assets. Inputs are transformed into outputs using the available resources. Inputs may be raw material, or a customer's problems, and outputs assembled machinery or a solution for the customer's problems.

Generally, the transformation process adds value to the inputs, so that the outputs can be sold in some form. This way, the organization earns money to replenish resources and to grow. Not all organizations want to grow in size, though. Many research institutes grow to a "reasonable" size, and then concentrate on expanding knowledge, not personnel. Here, the surplus from the outputs is used for resource improvement only.

A traditional example for such an organization would again be a car manufacturer. It uses raw material such as steel or aluminum as inputs and produces cars as output. The transformation process is fulfilled using the manufacturer's resources: buildings, machines, the workers, and engineers constructing the cars. Earnings are used to improve the transformation process, in order to produce the same output at lower input and resource consumption or to produce more or higher-quality output at the same input and resource consumption.

A software organization works similarly and differently at the same time. Similarly, because it also uses inputs and resources to produce outputs, and differently because most of its inputs and outputs are not physical objects, but immaterial. This results in the absence of any sort of production process. Using the car manufacturer example, the process would be over when the car is fully developed – it would never be built. This clearly distinguishes software development processes from traditional engineering processes. However, software organizations share with traditional organizations the urge to improve their processes.

The more complex the software products became, the more obvious became the need for process improvements. A variety of techniques, methods and tools were developed to cope with this, such as different programming paradigms, higher-level programming languages, and tools to administer the complexity of developing software products, the so-called IDEs[3]. While this was sufficient to reach a certain quality standard for smaller projects, it was not enough to support the organization-wide process improvements needed.

Software process modeling aims not at helping the development process with a set of individual tools, but at understanding the software organization as a whole. Figure 2.4 shows a simplified model of the software development process. In this model, products and activities alternate. Even in this primitive form, it gives a clue what must be considered in a software project. Coding cannot start before the requirements are fixed, for example.

[3] IDE = Integrated Development Environment

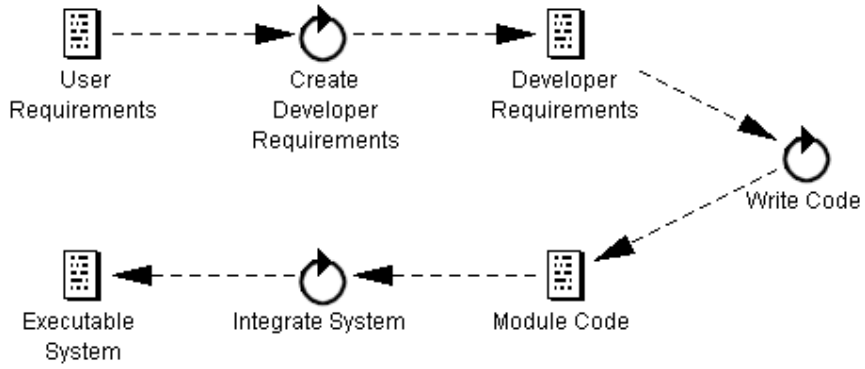


Figure 2.4: Simplified software development process

Of course, real process models are way more complex than the example presented here. They usually operate on different levels of abstraction, because system complexity is too high to be sensibly modeled in one model. A two-level approach might model the complete process on a higher abstraction level, considering major process steps, and model each of the steps individually on a lower abstraction level, thus in more detail. The two levels need to be connected in some way, though, so changes in a subprocess are reflected in the global model and vice versa.

This approach of modeling the complete process has been pursued by manufacturing companies for a long time. Modeling a software company is driven by the same goals (time, costs, quality), but typically models a totally different environment. In a manufacturing company, mostly technical processes are modeled (with the focus on controlling machinery), whereas a software company consists mainly of people. Controlling machinery (hardware) should only form a small part of the development process, the real work is done by human minds. This makes software processes difficult to model per se, because modeling creative processes is not understood very well yet. Additionally, multiple feedback loops make the situation more complicated.

One consequence of those feedback loops are changes to the development process. These changes are not as severe and do not happen as often in manufacturing processes because auf two reasons. First, the production processes are much better understood than software development processes, and second, the intention of (automated) manufacturing is to have only very few changes, for this would change the whole production line. Software development processes are rather young; the software community is still struggling to evolve from the art to the profession. Consequentially, there is no engineering handbook yet, which says what to do in certain situations.

So, modeling software development processes faces two great challenges. First, the processes to be modeled are only partially researched, and second, managing changes in the processes is difficult. Models need to be updated constantly to match new research results and real-world evolutions. In general, process modeling goals comprise but are not limited to [Mün02]:

- enable human understanding and communication,
- improve software development activities,
- support project management,
- guide project team members,
- trace project history, and
- automated execution of processes.

If the challenges can be addressed adequately, process modeling holds several benefits. By modeling the process, it is understood better, sometimes even understood at all. Often problems within the process, which have gone unnoticed for years, become obvious when it is modeled. The modeled process can then be optimized on paper, potential changes can be drafted and the consequences predicted. Finally, if the process model is enriched with empirical data, the process can be simulated and traced, thus enabling better project and resource planning and better directing of process changes, which lowers time, costs and quality risks overall.

There exist two main approaches to construct a model. The first approach could be described as theoretical approach, the second one as empirical. The theoretical approach builds the “ideal” process by capturing general principles of a methodology. This makes instantiation and refinement necessary. Most probably, the real-world process will differ from the constructed one. An example for such a model is the waterfall process model, which does not work out exactly as planned in the real world. Such theoretical models are often used prescriptively.

The empirical approach tries to transfer the real-world process into a model by acquiring its characteristics as accurately as possible. The resulting process models are then revised using experience from other processes. These models are mainly descriptive, for example, IDEF0 models.

Many tools and languages exist for process modeling: Appl/A, Funsoft Nets, Marvel Strategy Language, Statemate, MVP/L or IDEF0, just to name a few. They have in common that they want to help formalize the software development process, but differ in many other areas, e.g., the level of abstraction or the level of integration. Some regard a rather abstract process, some a more concrete one. Some only present a workbench with an (unconnected) set of tools, some an integrated process engine. A typical structure for process modeling tools of the latter kind is depicted in Figure 2.5 [Mün02].

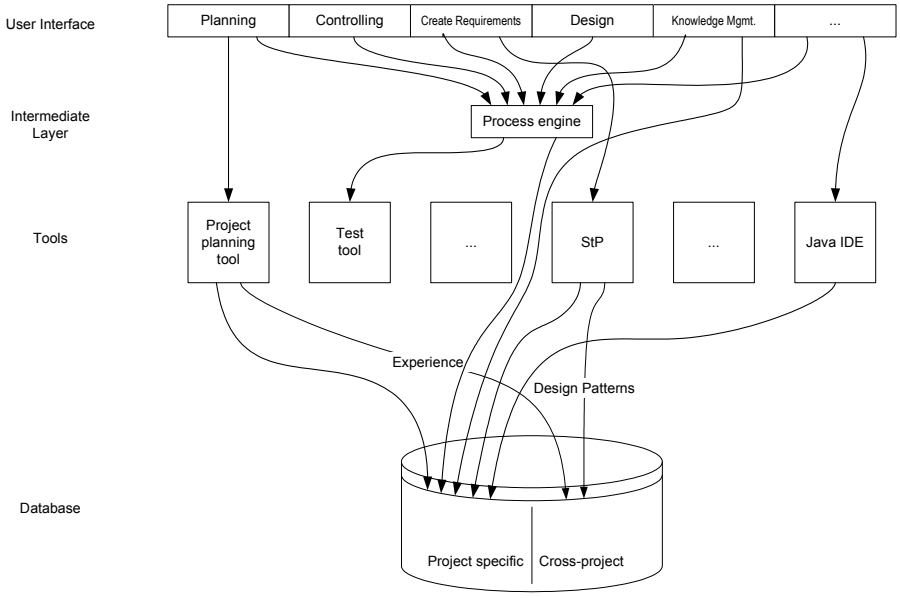


Figure 2.5: Typical structure for process modeling tools

Different process models and modeling tools support process improvement in a different way. Depending on notation and understandability of the model, it might be easy to identify process weaknesses ad-hoc and integrate respective changes. Other notations may support measurement-based improvement, for example the GQM approach [BCR94b].

In this work, a software process model is enriched with empirical data to improve model quality and to enable more accurate predictions for the real process. The resulting model may then be executed in a process engine to compare the model with real processes.

3 Combining Real and Virtual Experiments

3.1 Overview

If there are already that many experiments, real and virtual, why is there the need for combining them, and with this creating an even more complicated experiment universe? The answer lies in the industrial demands for faster and cheaper creation of higher-quality software. Technology alone cannot deliver the massive improvements longed for in these three key areas. Despite new programming paradigms, more sophisticated tools and modern integrated development environments, most software projects suffer from significant time and financial schedule overruns and/or poor product quality.

New technologies are being developed continuously to overcome these issues. The problem is, however, that no known technology is optimal for every situation. Therefore, new technologies must be tested in the target environment for their suitability. While this has been understood quite early, it is still not very popular among management due to the costs involved in setting up experiments. Consequentially, simulations have been tried to establish to reduce costs while keeping the advantages of experiments.

The two preceding chapters made clear the specific advantages and drawbacks of real and virtual experiments. To overcome the respective limitations, the recent research approach of combining real and virtual experiments has proven promising. This combination allows reducing cost and time needed through simulating experiments while at the same time almost fully preserving the quality of the knowledge gained through the experiments. It can be utilized in decision support, software project management and training as well.

One view on virtual and real experiments is the laboratory view depicted in Figure 3.1. The virtual laboratory is responsible for simulations; the real laboratory conducts conventional experiments, namely case studies and controlled experiments. Between the two, data is exchanged. Each of the labs has its own advantages and drawbacks. By combining the two, drawbacks of one lab may be compensated with advantages from the other.

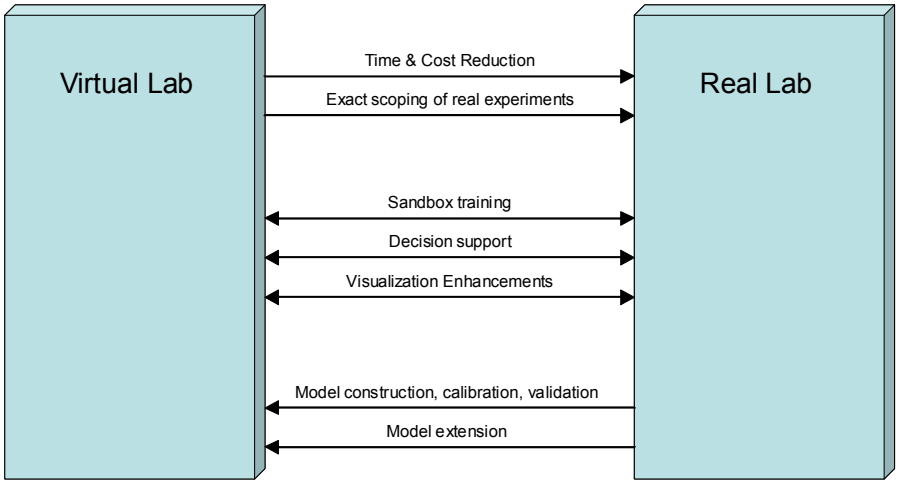


Figure 3.1: Virtual and real laboratories

Real and virtual experiments may take place in three main timely fashions (Figure 3.2). The real experiment may precede the virtual experiment, the virtual experiment may precede the real experiment, or both experiments may be conducted simultaneously. The first option enables using empirical knowledge in designing, calibrating and analyzing the simulation. The second option does the same vice versa, and the third option enables to capture effects on a "big" process while only conducting a "small" experiment. This is called online simulations.

Apart from direct connections to empirical experiments, simulations can also be used for training purposes. Software project issues and most popular mistakes can be taught to employees in a sandbox environment. Everybody knows the famous words "Adding people to a late project makes it later"[4], but still many project managers do exactly this. Letting them crash a simulated project is a cheap and efficient way of education.

Many project managers complain about insufficient and confusing project status information. But even if they have all information they need in an easy to understand form, it is still challenging to forecast the project progress. Simulation can also help here. When entered into a simulation system, information gained in a real experiment could be used in a faster than real-time simulation of the whole project. This may reveal potential bottlenecks before they appear, so countermeasures can be taken early. This decision support may help project managers founding their decisions more on facts than on intuition.

[4] Also known as "Brooks' Law" [Bro95]

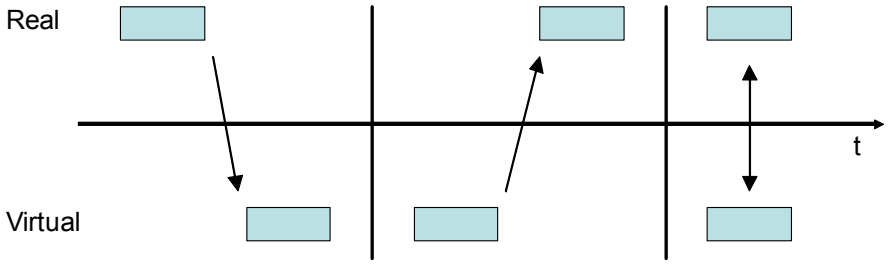


Figure 3.2: Possible timely arrangements of real and virtual experiments

Simulation can also help communicating with the customer. A seemingly little change in requirements may actually delay the project by months. This sometimes is hard to explain to non-technical personnel, thus growing anger on both sides. A simulation could scale up an appropriate real experiment and thus visualize the consequences of a change request. This helps both understanding and evaluating the costs, thereby enhancing customer satisfaction. The same applies to the own management.

3.2 Using Empirical Knowledge for Simulation

In order to take advantage of a simulation model at all, it must somehow describe reality. Any model only reflects a part of the real world, and is therefore easier to understand and process, but also never tells the whole truth, for some information from reality is left out. Whenever working with models, this should be kept in mind. The closer the model relates to the real world, the better the behavior of reality can be understood and also forecasted.

3.2.1 Determine Simulation Behavior

In order to describe the real world as exactly as needed and possible, it is necessary to include measures from the real world in the model. This ascertains and increases model validity. Including this data can be done virtually anywhere in the model. It can be used as an input, to give the model a defined starting point to begin with, to evaluate the consequences of this combination of input data and process modeled. During the simulation run, real-world data can be used to determine model behavior. This can be done completely deterministic, for example “If effort for activity A is lower than 150 hours, assume 200 hours for activity B, otherwise assume 300 hours”, but it can also be done partially nondeterministic, e.g., on the base of knowledge-based systems. Such a system could analyze the situation (that is, the model state as determined by its variables) and try to find similar situations with their consequences. After completion of the simulation, the simulation results can be compared to real-world data to determine the quality of the simulation model and which parts need improvement.

Most of these uses of empirical data require the real experiment to have taken place before the virtual experiment. For input data, this is obvious. Since model behavior needs to be clear when the simulation starts, the data used at this point also needs to be available at simulation start. Only output data can be made available after the simulation ends. This delays evaluation of the simulation model, but does not affect the simulation itself.

3.2.2 Enhance Simulation Model

Once the real experiment is conducted and the data analyzed, it can be used to enhance the simulation model, thus the virtual experiment. The model can be calibrated by simulating the real experiment and comparing the outcomes. By doing this, the quality of the simulation model can be evaluated, and model parameters can be adjusted to better reflect reality. If discrepancies between the results of real and virtual experiments are much greater than expected, revising the concept of the virtual experiment might be sensible: Did it really simulate what was expected, or were maybe some important factors left out?

If the results of the simulation and the real experiment sufficiently correspond, the existing simulation model can be expanded using empirical data. For example, if the simulation model fairly accurately describes inspection efficiency and effectiveness, a new module might be introduced considering time and costs of the inspection in respect to the quality of defect detection. For this, the real-world connections need to be determined in the real experiment, to enable building that part of the simulation model. An initial calibration can be done with the data available from the real experiment, followed by fine-tuning with data from other experiments, if their context is known sufficiently.

3.2.3 Prepare Simulation Model for Data Integration

If the virtual experiment takes place before the real experiment, only little empirical knowledge can be used for the simulation. Since measured data is not yet available, only the simulation design can be prepared to be enhanced with empirical data in the future. This includes a modular design to add additional influence factors and an easy to change configuration of the simulation model in terms of altering model behavior. If model parameters are hard-coded into the model, it will be more difficult to include discoveries made in experiments than when text-based configuration files are used, for example.

3.2.4 Determine Model Quality

Simulating past real experiments enables the experimenter to get a feeling for the simulation model, to determine its accuracy and mean and maximum deviations from reality. This is important if the simulation model is going to be used to answer specific questions: While it is vital to know the answer, it is even more important to know how much it can be trusted.

3.2.5 Benefits

Constructing, Calibrating and Validating Simulation Models

Using empirical knowledge, a simulation model can be brought "to life" at all. It may be easy to capture the actual process, that is, the sequence of process steps. This does help understanding the process, but not the interdependencies and the dynamic behavior. A good analogy might be a black-and-white photograph compared to a color video. The photograph shows what is present, while the video gives more detailed information about everything and at the same time shows the dynamic behavior. So, including empirical data during the construction of a simulation model yields a much more accurate model.

Validation of existing models is also vital to simulations. The processes modeled in the first place may change over time. If this is not reflected in the simulation model, the model produces incorrect output data and eventually becomes unreliable. So, constantly validating a simulation model against the real world is beneficial to simulation results. The same is true for calibration: Even if the modeled process *per se* did not change, employee skills may change over time. Reflecting this in the simulation model, e.g., by altering model parameters results in more accurate forecasts.

Extending Existing Models to new Areas

Most models are created in increments. The initial core model is extended with more modules to describe more aspects of reality. Again, empirical knowledge helps constructing, validating and calibrating these add-on modules. By integrating empirical data from not yet modeled areas, the overall simulation model can be extended to cover these areas. This enhances model quality and may result in better forecast capabilities due to the more exact image of reality.

3.3 Using Simulation for Real Experiments

The economically more interesting knowledge transfer is from simulation to real experiments. Here, more monetary benefits are expected by saving experimentation time and thus reducing experimentation costs. This section explains which information gained through simulations can be used in real experiments, or which real experiments might be replaced by simulations.

3.3.1 Scoping of Real Experiments

When a simulation is run prior to a real experiment, this means that the simulation results may be used to set the real experiment's parameters more accurately than without that information. In the first place, it can help determining which real experiments are needed most. Let's look at a situation where several process changes are proposed in order to improve product quality while not increasing delivery time or costs. Since real experiments are expensive, usually not every proposal is actually examined. There is usually only a limited budget for experiments, which leads to someone making a decision which experiment to conduct and which not. Whether it is a single human or a group like a steering committee, the decision is always based on a mixture of facts, experience and intuition.

To better support these decisions, simulations can be used to determine which of the possible process changes are likely to have the desired effect. If the simulation model adequately reflects reality and has proven to provide trustworthy results, only the process changes that prevailed in this first evaluation stage may be worth an extensive examination. In any case, the simulation model gives advice why or why not a certain proposal worked out, and where the problem was. Using this information, the proposed process change can be reworked or abandoned. Examining only the promising proposals optimizes the ratio of money spent and results yielded.

The same prioritization may be made at single experiment level. Input and output variables may not change smoothly over time, but rather little changes to an input variable may result in great changes in output variables. An example is given in Figure 3.3. Assuming that variable x changes constantly, variable y shows noticeable behavior: During a rather little change of x denoted as a, y changes the amount of b. When a simulation yields this result, special attention should be paid to the "a" part in a real experiment. The other parts can rather safely be neglected because no great changes to y are to be expected from changes of x there.

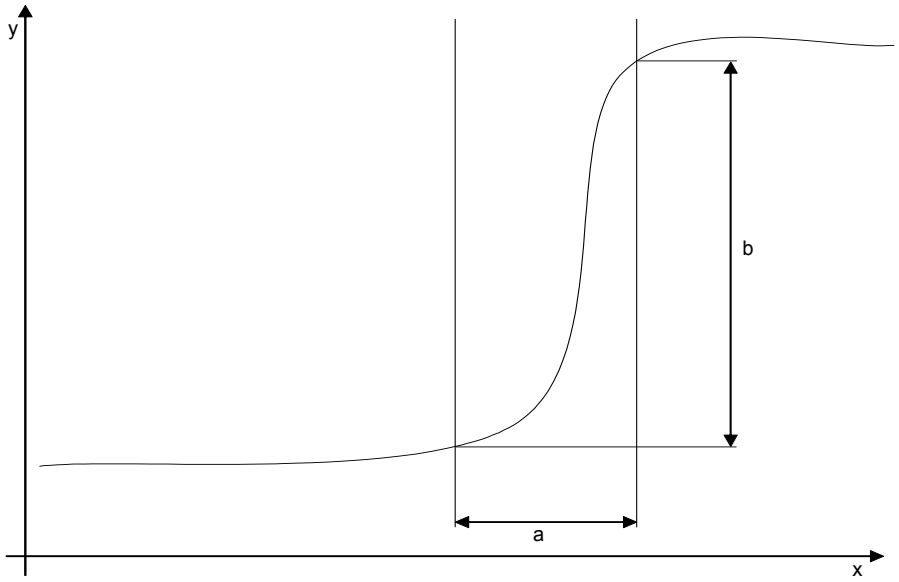


Figure 3.3: Experimental point of interest

3.3.2 Determine Data Needs

There is only little simulation knowledge that can be used in the experiment when the real experiment is conducted before the simulation. If the two experiments are planned in advance as a unit, there are, however, at least limited possibilities. Depending on the purpose and scope of the simulation, the experimental setup may be designed to suit the specific needs of the simulation. For example, if there is no simulation model at all yet, basic data about the process is needed. The real process must be analyzed and the experiment set up accordingly, with basic measures to be recorded. If there already is a simulation model, maybe even in a decent degree of maturity, the focus might be on fine-tuning the model by including more input factors that were neglected so far. The real experiment must reflect that in the data measured. This way, at least some meta data from the simulation can be used in the experiment.

3.3.3 Data Validation/Sensitivity Analysis

Supposing there is a decent simulation model available, it can be used to validate data obtained in real experiments. When the simulation model fits the real situation, the simulation parameters can be adjusted to the ones in the real experiment and the simulation of the real experiment can then be run. This way, not the simulation model is validated, but the real experiment. The simulation results should vary from the real results only about the same as in simulation validation runs. If variations are too great, this

might indicate problems with the real experiment's results. This way, a quick sensitivity analysis of the experimental results can be conducted.

3.3.4 Benefits

Reduction of Experimentation Time and Cost

Generally, wherever a real experiment can be replaced by a virtual experiment, this saves cost. Any real experiment needs test subjects, which in turn cost money, whether they are hired externally or chosen from company staff. The real experiment must be set up every time it is to be conducted. A virtual experiment also needs to be set up, but execution is almost free then. Replicating a real experiment requires almost the same effort as the first time, whereas replicating a virtual experiment merely consists of altering input and model parameters. So, there may be significant cost benefits from using simulations for real experiments.

Since simulations may be run faster than real-time, a virtual experiment can be conducted in a shorter time than the corresponding real experiment. This saves time, which results in faster integration of new technology into the development process. Additionally, time-to-market of the product can be reduced when using simulation tools for project steering and decision support. Simulation can help making better decisions concerning which real experiment to conduct and which not to.

Simulations can also replace replications and (simple) variations, assuming a decent simulation model. Multiply replicating a real experiment with the same input parameters using an entirely deterministic simulation model is useless of course, since it will achieve the same results every run. Stochastically scattering model and input parameters will give an impression of the range the results will be in.

Simple variations may also be evaluated by a simulation. Research in the simulation field is far from being complete and therefore, only simple variations may be applied to the simulated process without falsifying the results too much. However, these simple variations need not be evaluated through real experiments, thus saving time and cost.

3.4 Online Simulations

When both the real and the virtual experiment are conducted simultaneously, this is called an online simulation. Two basic kinds of online simulation can be distinguished. Both experiments can investigate the same matter, or they may examine different matters. In the first case, the results can be compared directly. This is useful for model calibration purposes.

Since this case is basically the same as the one already described in Section 3.2, it is not discussed any further here. For the rest of this work, "online simulation" will refer only to the case when different matters are examined simultaneously.

3.4.1 Scale-up

Real experiments usually only cover a sub-process of a larger process, for example, the inspection sub-process of a complete development process. Due to the immense costs, it is virtually impossible to conduct a real experiment examining a complete software development process. It might be done with a small (<10 people) team, but this does not help in industrial software development where development teams consist of hundreds of people. So, only a simulation can take care of the complete process.

Nevertheless, parts can be examined in real experiments. To capture the effect on the complete process, real and virtual experiments can be combined into online simulations. Typically, the simulation covers the large-scale process with real experiments examining some selected points of interest. This way, changes to sub-processes can be evaluated using empirical data, while at the same time the big picture is not neglected. Altering some sub-process, for example in duration, may result in altered feedback loops which cannot be foreseen by only examining the sub-process in an experiment. The simulation keeps track of that. This situation is exemplarily depicted in Figure 3.4. The simulation comprises the complete development process, while the real experiment examines only the design review sub-process. In this example, the often-investigated question whether to use perspective-based reading or checklist-based reading is focused on.

The simulation starts first. Since it can be run many times faster than real-time, the real experiment can start shortly after. It uses the output from the simulation, namely the system design. Of course a computer system cannot do a system design on its own. In the example, a finished project could be used. Assuming a decent simulation model and sufficient measured data, the simulation should come to similar results concerning effort and defects in the design as the real project. Having replayed the old project so far, the real experiment can be started then. The simulation is halted meanwhile, since it is waiting for input data from the real experiment.

Once the real experiment is over, its results need to be analyzed to gather input parameters to continue the simulation. In the example, this would be mainly the effort spent for the inspection sub-process and the number of defects found. The simulated rework would then commence, and after that, the rest of the development process is simulated. When the simulation is completed, output parameters may be compared to the measured data from the real project. How did the changed sub-process affect the whole project? Project duration (and linked with this, project costs) is interesting in

this context, but also the resulting product quality. Especially remarkable effects on other sub-processes can be determined, e.g., great integration problems due to poor design inspection performance.

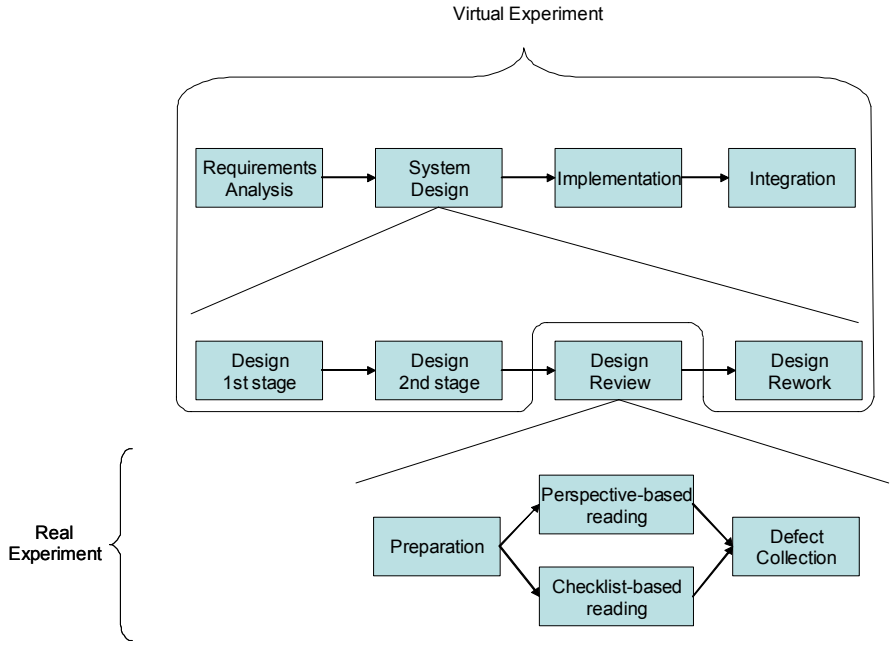


Figure 3.4: Online simulation principle

3.4.2 Training

Another field of interest is training. Any theory learned in university or other classes can only partially be applied to reality. It must be altered, adapted and evaluated. In this learning process, mistakes are made. While this is dangerous in real projects, using a simulation tool is safe. Employees can practice project management in a sandbox. Here, no real experiments are held during a simulation run, but the trainee acts as a project manager, making decisions and having the simulation model execute them. In this case, the real experiment is the trainee himself. For example, if the simulated inspection results in many defects found, is a re-inspection after rework sensible? In the real world, this would be a major decision, influencing every following project step. In a simulation, making the wrong decision does not endanger any real projects, but shows the trainee the consequences clearly, yet in a safe environment.

Thus, employees can be trained in a safe environment using simulation technology. Whether new employees need training to understand the company processes or there are regular trainings to improve the knowledge of employees, all mistakes can safely be made in a simulation. They do not harm the real-world processes, but are a good means of learning how to

handle specific situations. Another benefit is that the consequences of the trainee's decision appear almost immediately, whereas in the real world, the connection between decision and its consequences is not always obvious, due to the potentially long time between decision and consequence. Simulations reduce this lag to at worst hours, at best only seconds, depending on the complexity of the simulation model.

3.4.3 Benefits

Inexpensive Scale-up

Online simulations combine the better of the two worlds of simulation and real experiments. Using online simulations, specific issues can be examined while at the same time the big picture is not neglected. This adds value to the real experiment, because the effects of the examined issue can be evaluated regarding the complete process.

Immediate Feedback

Another benefit of online simulations concerns the psychological factor. If an experiment for example comparing different reading techniques is conducted, no immediate feedback is supplied to the test subjects as well as the experimenters. This may result in poor performance due to lack of motivation and thus distorted results. A typical question in this context is "What is this good for, after all?" Using an overlying simulation that supplies data on the effects on the rest of the process, this question – among others – can be answered.

Visualization Enhancements

It is especially difficult to explain the consequences of software development process changes to non-technical persons, e.g., management. Decision-makers in the management are often not from the software engineering domain. A seemingly small requirements change (for the customer) may require repeating large parts of the development process, which delays delivery time significantly. Any manager is much more likely to understand and approve this fact when there is a simulation that visualizes the necessary process changes.

An analogy from the automobile industry would be a car order. The late requirements change could be the demand for a car that is 50 centimeters longer. This is obviously a major change; however, a person without knowledge about car production might wonder why it is possible to get another color, but not a longer car. Once this person sees how production processes need to be changed for both requests, this becomes clear.

This visualization is not only helpful with customers; it also helps the software company. It can help determining how many changes a customer request involves to the process. A preliminary schedule update may be extracted, so the cost for the request can be forecasted correctly. This cost forecast can then be used for a contract with the customer. The exact determination of the cost is vital to any company: If the costs are higher than the revenue too often, it will eventually doom the company.

Project monitoring may also benefit from the combination of real and virtual experiments. While in conventional project monitoring and steering, the focus is on the past with some static forecasts for major project phases, using combined empirical data and simulations may enhance project management significantly. Processing empirical data collected in the already passed project phases gives the oversight that project managers are used to. When a simulation model is added to project steering, dynamic forecasts can be made at every point in the project, thus providing a more precise project control.

3.5 Problems

The usefulness of every simulation model clearly depends on how exactly it describes reality. In order to improve the model in a way that this description gets more precise, the usage of empirical data from the real world is very important [RKP+99].The authors state that often the empirical data is the only available and directly traceable link between model and reality.

Acquiring this data, however, often represents a non-trivial problem. First, there is the question of what data is needed for the model. This depends mainly on what is being modeled, and to what degree. Models for product quality need data about defects, strategic enterprise planning models need data about cycle time, and support planning models need both. When it is clear what data is needed, it becomes necessary to evaluate how detailed the data must be. This depends on the desired granularity of the model, whether it is going to be a high-level model or shall simulate on a very detailed level.

Data Collection

Once the model focus and scope are determined, collecting data begins. This is easiest if empirical experiments can be influenced, or at least the data collection. Data specifically needed for the simulation model can be acquired, so there is a perfect fit of data and model. Unfortunately, this rarely ever happens.

A more common case is the one where the model is built with an initial data set, maybe even from a tailored experiment, but validation and cali-

bration is done using other data. “Other data” mostly means older data which has been sitting in a closet anyway. This way, modeling costs can be reduced and the once expensively acquired data can be reused. There are problems with this approach.

Older data may not describe the modeled aspects accurately. This is most often found when only data “similar” to the data really wanted is available. Sometimes, this can be compensated by deriving the desired data from the data available. For example, if defect severity and quantity data is needed as model input, but only data on rework is available, the defect data can be concluded from the rework data: Longer rework suggests more and/or more severe defects. However, this can only be done if more information about the context is available. In the example, this would be data about the technology used, personnel knowledge and experience and of course the type of system that was developed.

Data Interpretation

Data on effort and defects has a great advantage: It can be quantified rather easily. This is not the case with other data such as readability, usability and understandability. Much data is acquired through forms where the test subject is to evaluate something on a scale, e.g., from one to ten. Extracting this introduces noise into the data. A nine for one person maybe is more like a five for another. Furthermore, answers to questions for readability or usability depend largely on the person answering, much less than “objective” facts like defects.

It gets even worse when there isn’t even a scale, but the questions are answered in natural language. The answers are interpreted by humans and then transformed into the input data needed for the simulation model. This introduces additional noise. In any case, it is important to know who answered the questions. Was it a group of senior software engineering professionals, or rather junior persons just evaluating their first project? If this is not known, evaluating the answers for the current simulation model is very hard and not likely to yield very reliable results.

Unknown Context

All the problems mentioned so far have one thing in common: It is more or less obvious if there are any problems with the context. When deriving data from other sources, the dangers are clearly visible, the same applies to unknown test subjects or interpreted natural language answers. These dangers can be discovered and accounted for, for example by comparing them with other data, so one bad data set does not spoil the model.

Nevertheless, inherent to using older data is always the danger that certain assumptions were made at the time the data was acquired, but that this has

not been recorded. In this case, the context is different, but this may go unnoticed. This is maybe the greatest danger, since it might not be obvious that there is a different context at all. If data taken from one context is mixed with data from another in the simulation model, the simulation may show some weird results without any indication that this is not because of the model, but because of the data used.

Only very careful choosing of data can help with these problems. Experience can help detecting potential problems within the data, but not completely eliminate all dangers. Choosing data and integrating it into the model is critical to model quality and should be accomplished with great caution.

3.6 Summary

In addition to isolated real or virtual experiments, combining both approaches holds additional advantages in several fields of application. When using empirical knowledge in simulations, the simulation model can be refined in an iterative way by integrating the experiences from real experiments held over time. Thus, more accurate predictions can be made with the simulation model. The model itself can already be based on real data acquired through controlled experiments or case studies, as opposed to purely artificial models in a simulation-only approach.

The simulation model can also be expanded by using experimental results. Aspects of the real world not yet considered can be included in a new version of the simulation model. This way, a model can grow to a more comprehensive model over time. The simulation model can also be prepared for later data integration, if data from real experiments is not yet available. By simulating appropriate real experiments, model quality can be checked.

Using simulations for real experiments holds some more synergy effects. First of all, real experiments can be prioritized and scoped. This means that before (expensive) real experiments are held, simulations are used to determine what to examine in which detail. Simulations can also help determine which kind of data is needed, so real experiments can be planned according to actual needs, rather than intuition. Finally, simulation results can be used as a quick sensitivity analysis of real experiments' results.

As can be seen, combining real and virtual experiments holds many benefits. It may not always be easy to express these benefits in numbers. However, this should not keep the software engineering community from exploiting them. Table 3.1 gives an overview over the expected benefits.

Knowledge Usage	Benefits
Empirical → Simulation	Constructing, calibrating and validating simulation models Extending existing models to new areas Evaluating simulation models
Simulation → Empirical	Reduction of experimentation time and cost
Online simulations (↔)	Inexpensive scale-up Employee motivation through immediate feedback Visualization enhancements Training

Table 3.1: Benefit overview

4 Simulation Model Development

4.1 Overview

As an example for the combination of simulation and real experiments, a simulation model of an inspection process has been developed. The inspection process has been chosen because most of the research being done is situated in this area. This supplies a good knowledge basis for modeling as well as sufficient real experiments to calibrate and test the model.

More precisely, the simulation model used in this work represents an experiment conducted by Basili et al. in 1994/95 [BGL+96] and two replications by Ciolkowski et al. in 1995/96 [CDLM97]. The initial experiment and its replications provide three independent data sets to calibrate the model with.

The simulation model described in this section implements the combination of real and virtual experiments described in Section 3.2. Using empirical knowledge, the simulation model is developed and calibrated. To reduce complexity, the model concentrates on two key factors influencing the inspection process. These two are the document that is to be inspected, and the set of reviewers available. The document is characterized through the attributes *number of defects, size* and *complexity;* each reviewer is characterized through the attributes *experience, expertise,* and his *individual defect detection rate (iDDR)* and *document coverage.* An overview is given in Table 4.1.

Factor	Attribute	Description
Document	Number of Defects	Absolute number of defects in the document
	Size	Document size
	Complexity	Document complexity
Reviewer	Experience	Reviewer experience with reading technique, document type…
	Expertise	Reviewer domain expertise
	iDDR	Individual defect detection rate of reviewer
	Document Coverage	Percent of document covered by the reviewer

Table 4.1: Overview over model attributes

Biffl [Bif00] stated that experience and expertise influence the individual defect detection rate of the reviewer. Furthermore, document properties such as size and complexity also have an effect [BLW97]. The defect de-

tection ratios used for model calibration must be seen in combination with the documents that were inspected and the set of reviewers that were available—these factors are not independent.

The development of the simulation model followed a method that is currently being developed [RNM03]. First, the goals pursued with the model were defined. Then, a static process model was created that reflects the inspection process. This static model was used as the basis for the dynamic simulation model. An influence diagram captured dynamic behavior and interrelations. During this phase, empirical data from the experiments was integrated iteratively. Then, the dynamic simulation model was built using the simulation tool Extend and calibrated with data from the experiments. To enable the model for multi-document runs, some extensions to input parameter and simulation handling were made. Finally, model limitations are pointed out and lessons learned presented.

4.2 Modeling Goals

The primary goal is to develop an empirically-based simulation model that allows for determining the effects of PBR and ad hoc requirements inspections in specific contexts. The intended usage of the model is decision support for planning software projects and tailoring the requirements inspection process to individual environments. This can be combined with other decision support techniques as for example described in [RBH02]. Furthermore, the model should help to reduce risks in introducing or changing requirements inspection processes by better forecasting their impacts. Several simulation models for inspections processes exist (e.g., [NHM+02], [NHM+03]) that are mainly based on hypothetical assumptions or expert knowledge.

The scientific focus was to better understand the use of quantitative empirical knowledge for simulation model development. Therefore, the model abstracts from several details and highlights the use of empirical data and further results from empirical studies. Secondary goals comprise modeling itself as well as the practical demonstration of the methodology presented by Rus et al. [RNM03].

4.3 Analysis and Creation of the Static Process Model

Before modeling dynamic behavior, a static process model that reflects the examined real-world process was developed. This can be done, for instance, by descriptive software process modeling [BK01]. The static model of the inspection process used is depicted in Figure 4.1. It is adopted from the model of Münch et al. for requirements inspections instead of code inspections. The activities, products and roles are modeled according to [LD00]. It is a well-known inspection process with individual preparation

phases and a subsequent phase where individual defect lists are merged into one. Software engineering literature provides various experiments and studies where real team meetings are held and others where only nominal teams are examined. An overview can be found in [LD00]. A nominal team is a team that does not work together as a team, but individual defect lists are simply merged. This means that no real team meetings take place. There is no conclusive answer on whether team meetings enhance inspection results or not. The two replications of the original experiment that were used in this work compared real team meetings against nominal teams and found no increase in defect detection effectiveness. That is the reason why only nominal teams were considered in the simulation model.

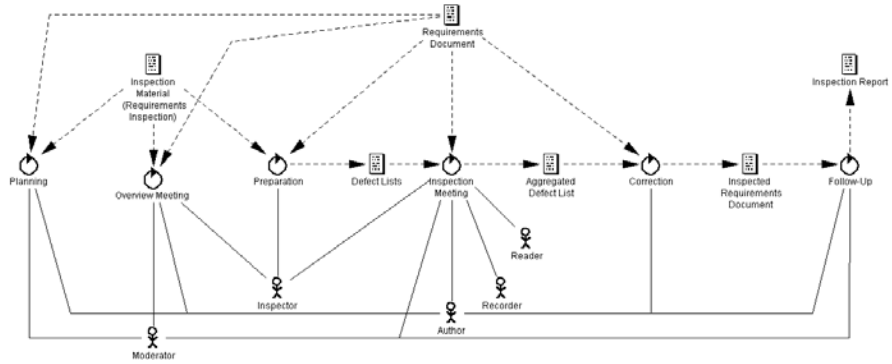


Figure 4.1: The inspection process model after [MBH+02]

4.4 Collection and Analysis of Empirical Data

The empirical knowledge used for developing the simulation model was mainly gathered from the experiments described in [BGL+96] and [CDLM97]. Throughout the rest of this thesis, the "original experiment" refers to [BGL+96], and the "replications" refer to [CDLM97]. The object of all experiments is a requirements document inspection process consisting of an individual preparation phase and a phase where the individually detected defects are pooled together. The initial experiment used professional software developers from the NASA/GSFC[5] Software Engineering Laboratory (SEL) as test subjects. They were tasked to review different types of documents, one from the NASA domain and one generic in nature, using their usual reading technique and the perspective-based reading (PBR) technique. Due to time and costs constraints (the experimenters estimated at least $500 per subject and day), no real teams were formed. After individual defect detection, the data gathered was pooled together according to the nominal team approach. The original experiment was conducted in two runs. After a pilot run, several changes to the documents and

[5] National Aeronautics and Space Administration/Goddard Space Flight Center

the experimental settings were made. This is why only results from the second run are considered here.

The replications took place one year later at the University of Kaiserslautern. Since the subjects were all undergraduate students, only the generic documents were used. These were identical to the ones used in the second run of the original experiment. This time, PBR was compared to an ad-hoc approach. Furthermore, the experimenters examined whether real teams discovered more defects than nominal teams.

In both experiments, PBR proved to be superior to the other technique used in almost all cases. Although no statistically significant effects could be proven, the experimental data from the replication suggested that meetings do not provide synergy effects: Although usually some new defects were detected, there were a huge number of meeting losses (i.e., detected defects that were not reported in real meetings).

Although the experiments were conducted to compare different reading techniques, and data collection was driven by this purpose (as opposed to developing and calibrating a simulation model), data like the average individual defect detection rate or team defect detection rates could be used for the development of the simulation model.

Data used for modeling included information about the individual defect detection rate *(iDDR)* of each reviewer, document defect detection rate *(dDDR), defect detection overlap* (how many defects are also detected by other reviewers?) between reviewers, and *document size* and *complexity*. To ease adoption of the model to different context, the relative importance of reviewer *experience* and *expertise* can also be adjusted. Finally, *document coverage* allows for the consideration of time pressure. Not all the data that would have been helpful was collected during the real experiments. They were designed to examine different reading techniques, but not to support simulation model development.

When the real experiment is planned and conducted to (also) suit simulation model development needs, all data considered necessary for the simulation model can also be made available. This way, a good symbiosis of real experiments and simulation can be achieved. This represents the preferred case, however, probably not the most likely. It can be expected that most data stems from older experiments which were not designed to suit simulation needs. Thus, the data that is to be included in the model must be extracted from available data. While this may not be possible for all data, this work proved that this approach works for a real context. There was, however, no methodological support for deriving the data needed from the data available.

4.5 Creation of the Influence Diagram

An analysis of the data provided by the experiments revealed a relationship among individual defect detection rate (iDDR), team defect detection rate (DDR) and reviewer know-how. The defect detection rate is defined as the percentage of the defects found with respect to all defects (Equation 1).

$$defect\ detection\ rate = \frac{number\ of\ defects\ found}{total\ number\ of\ defects} \quad (1)$$

iDDR indicates the individual performance of every reviewer, DDR specifies team performance for one document with doubles sorted out. The experiments confirmed that DDR rises with iDDR, while iDDR was higher with professional software developers than with undergraduate students.

The description of the real experiments provided further information: The number of defects in the inspection documents was known (27/29) as well as the document size in pages (16/17). This results in an average of about 1.7 defects per page. Additionally, document complexity can be determined, since the documents are publicly available. A possible measure could be McCabe's cyclomatic complexity measure [McC76]. On the reviewer side, some attributes are also known. The original experiment used professional software developers as opposed to undergraduate students in the replication. While the number of years of experience did not correlate to inspection performance, the professionals yielded significantly higher average iDDR scores than the undergraduate students (28.39% versus 21.08%, which is a 34.68% increase).

The model is based on average defect detection ratios; all other context information is normalized. This corresponds to the experiments, which also concentrate on average DDR scores. As can be seen in the influence diagram (Figure 4.2), iDDR values are derived directly or indirectly from the context variables. Thus, by focusing on defect detection ratios, simulation complexity could be decreased while at the same time the context was not neglected.

Figure 4.2 includes literature references for positive or negative correlations in the influence diagram. References named as [A] describe relations I introduced to structure the model. This concerns primarily the intermediate attributes *Document Difficulty, EE Level* (i.e., experience and expertise level) and *Inspector Capability*. Basili et al. [BGL+96] found that the document reviewed significantly influences individual defect detection ratios in the generic problem domain. Biffl [Bif00] found in his experiment that inspector's experience as well as expertise as defined in Section 4.7.2 significantly influence inspection performance. Rising document difficulty lowers the inspector's defect detection ability for the specific document

[BGL+96], while rising experience and expertise also rises inspector capability [Bif00]: The better qualified an inspector is for the document to be inspected, the higher his individual defect detection rate can be expected.

Another correctional factor *(iDDR Correction Factor)* needed to be introduced for the calculation of the document DDR. This factor symbolizes the overlap of iDDR values combined for one document. Depending on the reading technique, the defects found by each individual reviewer may overlap more or less due to restrictions resulting from the chosen reading technique. A PBR technique focusing on certain defect classes for each reviewer is more likely to yield a lower overlap than an ad-hoc technique, where every reviewer scans the complete document for any defects.

For the immediate model purpose, the variables describing the context (i.e., inspector experience and document complexity) were not mandatory, since the context is already reflected in the iDDR values. An adaptation of the model to new situations could then be done by adjusting the individual defect detection ratio, of course, but this value can then again only be determined in costly real experiments, thereby contradicting part of the modeling goal. This is why the context variables were included in the model, but with neutral behavior. Their potential influence is not impaired by this; they are just set to values that do not influence input iDDR values. To adapt the model to different contexts, the context variable values can be altered accordingly, with this model and its experiments as a reference value.

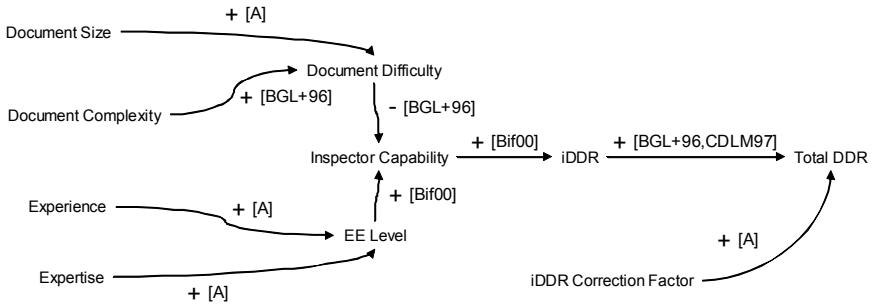


Figure 4.2: The influence diagram for the simulation model

4.6 The Dynamic Model

As a first step towards modeling the influence diagram, variables that were used in the real experiments to capture real-world behavior were modeled (see Figure 4.2). *Document Difficulty, Inspector Capability* and *EE Level* are additional intermediate attributes defined in order to understand and describe dynamic behavior. They are not reflected in the final model as discrete blocks, but combined into some (more comprehensive) equations. Their influence remained the same, but the combination reduced the num-

ber of blocks and connections in the model implementation significantly, thereby enhancing readability.

First, the attributes included in the final model will be introduced. These are attributes found to have a major influence on inspection results. Scales to measure these attributes are provided, in order to alter their values and thereby adapt the model to new contexts. There are other scales that can be used, of course. In any case, it is possible to adapt the model to various concrete measures.

4.7 Model Attributes

A key element of modeling is choosing attributes to be included in the model. It should be examined very carefully which attributes of the real world need to be modeled to adequately represent reality, and which can be left out to reduce complexity without altering model behavior too much. The attributes chosen for this model are depicted in Table 4.1 and will be explained further here.

4.7.1 Document Attributes

Document attributes characterize the document that is to be inspected. Thus, their values may differ with every document.

Number of Defects (1..n)

The number of defects in the document influences the number of detected defects. In a defect-free document, no real defects can be detected, if any, only false positives can be expected. A false positive is a reported defect in an actually correct part of the document. With an increasing number of defects, the number of actually detected defects can also be assumed to rise. A natural limit can be expected at the point where the inspected document contains so many errors that no sensible defect detection can be done any more due to the fact that it gets difficult to identify defects at all. If this is the case, a document should be rejected, since it is not yet ready to be sensibly inspected. In this model, the number of defects is needed to model the effect of the inspection in terms of defect reduction. As mentioned earlier, the original experiments used documents with a defect density of about 1.7 defects per page.

Document Size (0..2)

The document size influences the inspection in at least two ways. First, the bigger the document gets, the more time is needed to read it and perform the defect detection. Second, with growing document size it becomes in-

creasingly difficult to understand it completely, thus impeding defect detection [BLW97]. Since in different domains very different document sizes occur, it did not seem sensible to use an absolute number here. Instead, a relative scale is used, with 1 symbolizing an average document size (average for the domain, company and review team). Numbers below 1 mean that the document is smaller than usual, while numbers above 1 mean an unusually big document. The 1 in our model represents our reference document with 17 pages. Document size cannot only be measured by the number of pages, of course. While this might make sense for requirements documents, the size of code documents could be measured in LOC, for example.

Document Complexity (0..2)

Document complexity influences defect detection in that it may be very difficult to understand the document itself and therefore to find any defects at all [BLW97]—on the other hand, a complex document may lead to more false positives than a very simple one. Thus, document complexity is a key attribute in the model. For quantifying this influence, the same considerations as for the document size apply: In different domains, very different document complexities may appear. Therefore, the same relative scale as for document size has been chosen, with 1 symbolizing average complexity and numbers below and above symbolizing lower and higher complexity than usual. In order to use the simulation model in different contexts, comparing document complexities, e.g., using McCabe's cyclomatic complexity measure [McC76] seems plausible.

4.7.2 Reviewer Attributes

Reviewer attributes characterize the reviewers. Each reviewer has different capabilities, which must be taken into account. Reviewer attributes typically change only slowly over time, e.g., with growing domain knowledge. The model parameters, however, need to be adjusted for each different set of reviewers.

Experience (0..2)

Experience symbolizes the reviewer's experience with inspections, the document type, and the reading technique chosen. It is assumed that this also influences inspection performance. A reviewer using a new reading technique may need more time for administrative tasks than a more experienced reviewer. An uncommon document type may lead to confusion, lowering the defect detection ratio. Finally, the reading technique also influences inspection effects. For simplicity, experience is measured on a relative scale with 1 symbolizing average experience (in the respective

context), and values below and above symbolizing lower and higher experience.

Expertise (0..2)

Expertise takes into account the domain knowledge of the reviewer. While this might not be very important for some defect types, it may become absolutely necessary when regarding core system functions. When there are only reviewers foreign to the application domain, critical defects may go unnoticed until late in the development process. Thus, it is expected that domain knowledge also influences defect detection. Since the same scale problems as with experience apply to expertise as well, the same relative scale is used here.

Document Coverage (0..1)

This symbolizes how much of the document can be covered by the reviewer in the allocated inspection time. Time pressure, for instance, can be expressed through this value. The values can be seen as percentage values. Normally, document coverage should be 1, but when time pressure inhibits a thorough inspection, the value can be lowered accordingly.

Individual Defect Detection Rate (0..1)

This depicts the defect detection rate of every individual reviewer. Mostly, more than one reviewer will participate in an inspection. Thus, different individual defect detection rates must be considered. If the reviewer's capabilities are well-known, this value can be set accordingly. If only group performance has been monitored so far, mean and standard deviation values may be introduced at this point. The model presented in this work was calibrated using mean and standard deviation values for individual defect detection rates.

4.7.3 Additional Adjustment Variables

These additional adjustment variables are deemed necessary to better describe the context of the inspection. When using a specific review technique, for example, experience with this specific technique may be more important than domain knowledge [Bif00]. This is reflected in the simulation model through adjustment variables. If no special focus lies on one or more variables, all adjustment variables except for the review correction factor are set to 1. Lowering or raising a value increases or decreases the weighing of the corresponding factor relative to the other factors. This enables the simulation model user to put special emphasis on single factors. Defect detection overlap ranges from 0 to 1.

Experience Importance

This describes the importance of experience as defined in the section "Experience". Experience may be especially important in special inspection processes with tailored reading techniques or special document types.

Expertise Importance

If the software focuses on very special application domains such as aviation or weapons systems, expertise may become relatively more important than the other factors. This situation can be reflected through this value.

Review Correction Factor (0..1)

The review correction factor symbolizes the overlap of detected defects of the individual reviewers. Thus, it can change with the reading technique. An ad-hoc approach would have a high overlap since no special focus is given on where or how to detect defects. A perspective-based approach would have a lower overlap. Altering this value could become necessary, for instance, if individual defect detection performance for a certain reading technique is known, but the search fields are redefined. If the new search fields supposedly overlap less, this value could be increased to reflect this context change in the model.

4.8 The Simulation Model

The simulation model is described in two steps. First, the model basics are explained. That includes the realization of the attributes described above, the calculation of various values and general document and reviewer flow. In this stage, the inspection of only one document with one group of reviewers is modeled. In a second step, the model is extended to allow for the simulation of an arbitrary number of documents, each with a different set of reviewers, in one simulation run. This is described in Section 4.10.

The model is realized as a discrete event model using the modeling and simulation environment Extend [Kra00]. The document is generated with its properties, and a set of reviewers is generated with their properties. Then their individual defect detection rates (iDDR) are combined into the total defect detection rate of the (nominal) team for the respective document (dDDR). To illustrate the effects, the number of defects remaining in the document is calculated. An overview over the complete model is given in Figure 4.3.

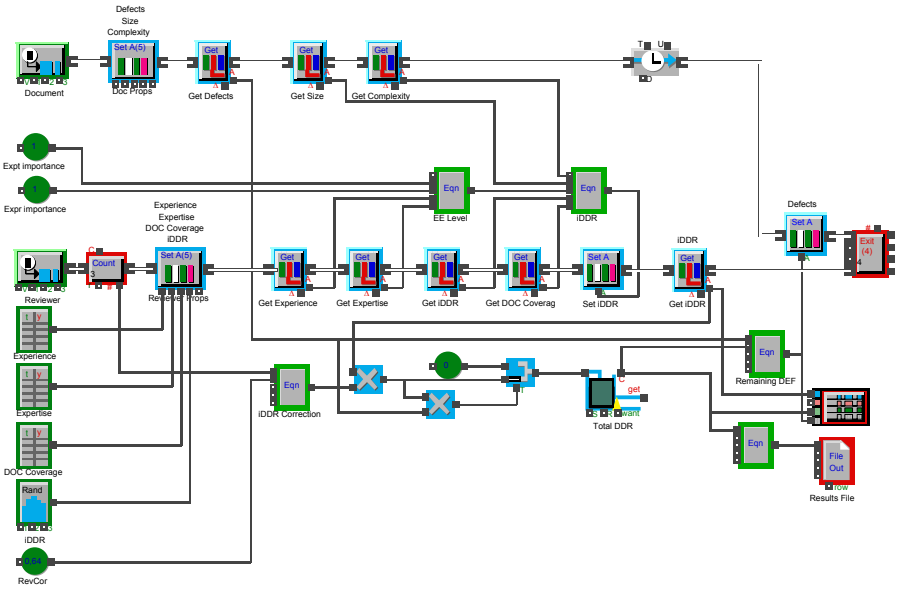


Figure 4.3: The Extend simulation model

The equation block labeled *iDDR* outputs the individual defect detection rate of the reviewer adjusted to the actual process context. It uses the attributes depicted in Figure 4.2 and the reviewer iDDR as input parameters. In this instantiation of the model, input iDDR equals the block output, because the context is already reflected in the input iDDR. In order to adapt the model to different contexts, experience and expertise can be adjusted relative to each other. If an inspection requires intimate domain knowledge, the importance of that factor can be raised compared to reviewer experience with the reading technique, for instance. Furthermore, individual experience, expertise and document coverage levels can be altered. This can be useful if defect detection rate data is available for groups of people, but not for individuals. Also, individual improvements can be reflected through these values.

The equation block labeled *iDDR Correction* takes care of the fact that not all defects detected by an individual are really new defects. Some have already been found by others, some not. Since the model is based on average defect detection rates, the individual detection rates are decreased with an increasing number of reviewers. This yields a potential problem if the group of reviewers is very inhomogeneous. If all reviewers perform about the same, it does not matter which one comes first and which one last. If there are major differences in defect detection ratio, their starting in the model determines the simulation results. To eliminate this threat, several simulation runs with random starting numbers of the reviewers should be conducted and the results then averaged.

At a certain point, adding more reviewers does not lead to any more defects discovered. Although the number of reviewers needed to reach that point has not been provided by the real experiments, a constraint is included in the model that discards individual defect detection success if the (calculated) number of detected defects is smaller than one. After this check, the final adjusted defect detection rates are totaled up into the document defect detection rate (dDDR). For illustration purposes, the remaining defects are calculated last.

A simulation run consists of the following steps: At simulation time 0, the document is created and its properties are set. In this work, defects were set to the number of defects in the reference document, while size and complexity were set to 1 in order not to spoil simulation results. The *Get* modules read out their respective values and provide them as an input to the defect detection rate calculation modules. Then the document is halted for the time the reviewers need to pass through the simulation.

At simulation time 1, the first reviewer is created. The counter is needed to decrease the number of newly detected defects for later reviewers. Reviewer properties are set during reviewer creation, analog to document creation. Experience and expertise are set to 1, again not to spoil simulation results. Throughout this work, a document coverage of 100% was assumed, so the corresponding value remains 1 throughout the simulation. Finally, iDDR gets set to values from a random number generator initialized with mean values from the reference experiments and a modest standard deviation of five percentage points of iDDR. This ensures that the simulation outputs a range of results in consecutive runs, so that mean results and typical variation can be identified. Experience and expertise importance were both set to 1, thus not preferring any of the two factors.

Following the setting of the reviewer properties, they are read again and used as inputs to various equations. The *EE Level* equation block adjusts the relative value of expertise and importance (Equation 2).

$$EE\ Level = \frac{Ec \cdot EcIm + Et \cdot EtIm}{EcIm + EtIm} \qquad (2)$$

with

Ec = Experience

$EcIm$ = Experience Importance

Et = Expertise

$EtIm$ = Expertise Importance

This result is combined with document size, complexity and coverage to an adjusted *iDDR,* which is then written back to the reviewer (Equation 3).

$$adjusted\ iDDR = \frac{EELevel \cdot iDDR \cdot Coverage}{Size \cdot Complexity} \quad (3)$$

These equations represent the corresponding parts of the influence diagram. The reviewer then leaves the simulation. At simulation time 2, the next reviewer enters the simulation and passes through the explained stages. This is repeated until all reviewers have passed through the simulation. Finally, the document is released, the number of remaining defects is written back into the document, and the document exits the simulation, which stops the simulation run. All results are written to a file to simplify multiple runs and the analysis of the results.

Simply adding up the iDDR values calculated in this manner would, of course, lead to extremely high document defect detection rates, which could easily exceed 100%. Therefore, individual defect detection rates for later reviewers are adjusted. This adjustment is calculated in the *iDDR Correction* equation. Determining this equation proved to be the hardest part of the modeling process. The original experiments did not provide detailed data about individual defect detection overlap. The only available information said that about 60% of the defects were found by more than one reviewer when using PBR, with a tendency to higher values when using ad-hoc techniques. This is not surprising, since PBR techniques try to overlap search fields only partially. While this did not give clear advice on how to decrease iDDR ratios for consecutive reviewers, it at least provided a general direction for defining a correctional factor. The equation was finally determined during model calibration by using data from the real experiments.

4.9 Model Calibration

In order to determine model behavior and to calibrate the model, empirical data from the original experiments [BGL+96, CDLM97] was integrated. This is described in the following. The experiments were not designed to support simulation models. They were conducted to compare different reading techniques. Therefore, some points of special interest will be dwelled on in this section.

The two main data sources were the average individual defect detection rate (iDDR) and the average nominal team defect detection rate for the complete document (dDDR). From the data published, standard deviations from the mean values could also be extracted. The original standard deviation values were not used for model calibration, though. They were up to

twice as high as the absolute iDDR values measured. Using these standard deviations in our model created reviewer iDDR values that were too unstable for sensibly calibrating the model. Therefore, a standard deviation of five percent points of iDDR was used for calibration. In a real-world application of the model, however, realistic values should be used whenever available. This could become especially important when worst case/best case inspection scenarios are to be evaluated. The value for the standard deviation used directly influences lowest and highest iDDR values. The values gathered in the real experiments are displayed in Table 4.2.

Value	Pilot usual	Pilot PBR	1995 usual	1995 PBR	Rep95 ad-hoc	Rep95 PBR	Rep96 ad-hoc	Rep96 PBR
avg. dDDR (%)	44,3	62,9	48,2	62,4	41,9	52,75	32,2	47,7
avg. iDDR (%)	20,58	24,92	24,64	32,14	21,33	25,93	14,59	22,46
avg. standard deviation (defects)	8,01	8,01	4,5	4,5	8,98	8,98	7,98	7,98
avg. dDDR - 2*iDDR (%)	3,14	13,06	-1,08	-1,88	-0,76	0,89	3,02	2,78

Table 4.2: Experiment values

Data from the 1995 run of the initial experiment and the two replications only was integrated into the model. After the pilot run of the initial experiment, the experimenters changed the context of the experiment. For example, they changed the time granted for defect detection and the place where the experiment took place (normal workplace → classroom setting). Because of this, the results of the initial and the other runs cannot be compared directly and, therefore, cannot be used for model calibration. Nevertheless, there were three sets of empirical data available to calibrate the simulation model. Throughout the rest of this work, “experimental data” refers to these three data sets.

When examining the data, it can be noticed that no matter which reading technique was used, the average dDDR was always about twice the average iDDR (Table 4.2). The highest deviation was 3.02 percentage points, with an average of 1.74. I do not believe this to be coincidence, so I used this fact for model calibration.

No isolated defects were modeled. The model relied on average values for defect detection. This is because of two factors: First, it reduced model complexity. Second, there was no complete data available from all three experiment runs concerning individual defects. Model calibration would have to be done with fewer data, which seemed worse than using average values.

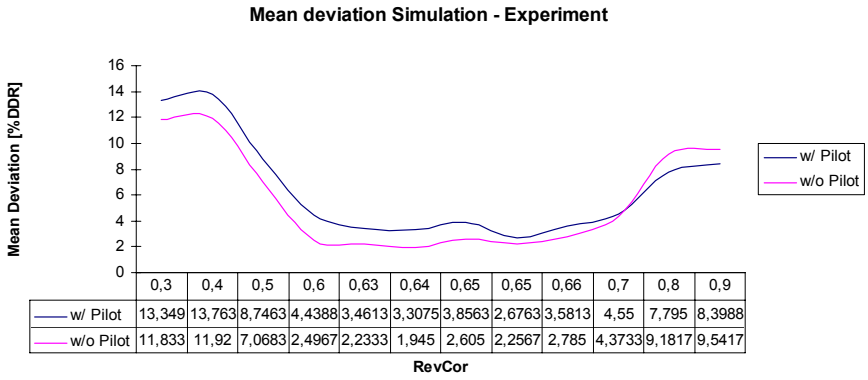
The drawback was that relying on average values made it more difficult to determine overlap in defect detection. “Overlap” means defects that were detected by more than one reviewer. They increase individual DDR, but not overall document DDR. When not examining single defects, there is no possibility to determine how many of the defects one team member de-

tected were already detected by others. There was, however, general information about how many of the total defects detected were found by how many persons. This at least indicated where to look for a correctional factor. A value was then determined for the data sets through systematic simulation runs with different correctional factors.

An exponential decrease in the defect detection ratio was used. In this scenario, the first reviewer's iDDR is added fully to the total DDR, while the following iDDRs are decreased by a factor determined by the number of the respective reviewer and a review correction factor (RevCor). The actual iDDR that is added to the dDDR is calculated according to Equation 4.

$$final\ iDDR = iDDR \cdot RevCor^{Count} \qquad (4)$$

where Count equals the number of the respective reviewer minus one. When systematically testing the model with different RevCor values, a value around 0.64 showed the lowest average deviation between the real experiments and the simulation results (see Figure 4.4).



	0,3	0,4	0,5	0,6	0,63	0,64	0,65	0,65	0,66	0,7	0,8	0,9
w/ Pilot	13,349	13,763	8,7463	4,4388	3,4613	3,3075	3,8563	2,6763	3,5813	4,55	7,795	8,3988
w/o Pilot	11,833	11,92	7,0683	2,4967	2,2333	1,945	2,605	2,2567	2,785	4,3733	9,1817	9,5417

Figure 4.4: Deviation Experiments – Simulation

This can only be a first calibration, of course. More data from other experiments is needed to determine the validity of the equations used and the RevCor factor.

4.10 Model Extensions

After the initial model was developed and calibrated, it was extended to comprise a less abstract inspection process: Only few inspections consist only of one document examined by one group of reviewers. Usually, more than one document is inspected. While this could be emulated with multiple runs of the simulation, it seemed not very comfortable. All input vari-

ables would have had to be set according to the new document/reviewers combination for every run, thus not allowing first setting all parameters, and then running the simulation unattended.

Because of this, an extended version of the model was developed. It completely contained the initial version in that it uses the same document and reviewer flow and calculates the values of all variables similarly. The changes allow for the simulation of multi-document inspections in one simulation run. Also, the reviewers' iDDR values are read from a file. This way, each reviewer's iDDR can be set individually. As in the basic version, all results are written to a file. The complete model is depicted in Figure 4.5.

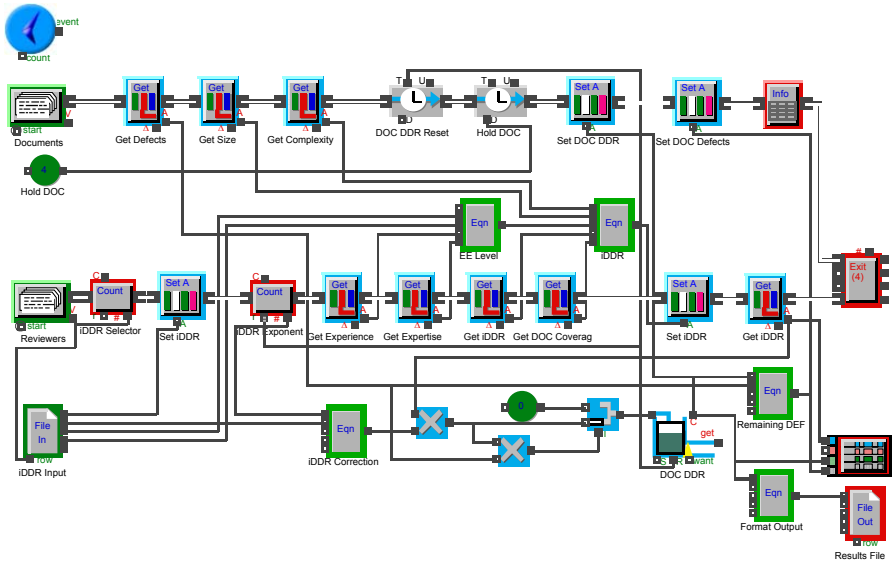


Figure 4.5: The complete simulation model

4.10.1 Multiple Documents

Document generation now is realized as a *Documents* program block generating document items at predefined points in time. In order to achieve a valid (i.e., each document is inspected by the correct number of reviewers) model output, document items should be created in intervals depending on the number of reviewers. To get an easily readable output value plot, an offset of *maximum number of reviewers per document + 2* is recommended. For example, if the maximum number of reviewers per document is 3 and the first document is created at simulation time 0, the subsequent documents would be created at simulation times 5, 10, 15,…

The *Program* block also provides means to immediately set item attributes. This was used to set the values of *Defects, Size* and *Complexity* right at

document creation. This allows for pre-determining all documents with their parameters and then importing the data in one step into the model, instead of setting various parameters in various places.

4.10.2 Multiple Reviewer Groups

Each document can be inspected by an arbitrary number of reviewers. This was modeled using a *Reviewers* program block. The reviewers start one time unit after their respective document. The iDDR values are read from a file. Due to the backward integration function of the *DOC DDR* summation block, a dummy reviewer having an iDDR of zero must run through the simulation after the last real reviewer. This must be considered when creating the reviewer and iDDR input data. *Experience, Expertise* and *Document Coverage* are set in the program block. It is recommended to set these values to zero for the dummy reviewer, to better visualize the disjunction of the reviewer groups. However, other values do not influence the simulation results.

The *iDDR Input* file needs to be composed the following way. Each reviewer's *iDDR, RevCor, EcIm* and *EtIm* values stand tab-separated in a row. Reviewer groups (=documents) are separated by a row containing four tab-separated zeros. Because the *RevCor* factor depends on the inspection technique, it is likely to remain constant within reviewer groups, but possibly changes between documents. This would be the case when groups of reviewers use different reading techniques on their respective documents. *Experience* and *Expertise Importance* values probably also change only between documents. Nevertheless, the simulation model allows for per-reviewer settings.

The *iDDR Selector* counter selects the row that is to be used from the input file. The zero values are assigned to the dummy reviewers. This way, they do not influence simulation results while the final document DDR is summed and then written back to the document.

4.10.3 Miscellaneous Changes

Some minor changes were required by the multi-document, multi-reviewer inspections. A *DOC DDR Reset* block is waiting for a document to pass. When a document passes this block, the *DOC DDR* tank and the *iDDR Exponent* blocks are reset to zero. This clears the way for a new document.

The document that is being inspected needs to be held back for the time the reviewers are running through the simulation. This is done by the *Hold DOC* block. Unfortunately, the holding time cannot be calculated automatically. This would require knowledge about the number of reviewers planned for each document. To simplify model usage, a single holding

time that is valid for all documents is defined. Corresponding with the document item offset, a value of *maximum number of reviewers per document + 1* is recommended. This ensures that documents and corresponding reviewers are correctly paired.

After the document DDR is calculated, the document is released. The values for document DDR and remaining defects are written into the document item, which exits the simulation after that.

4.11 Overview of Used Empirical Knowledge

This section gives an overview of the empirical knowledge used for simulation model development, calibration and validation. It is intended as a hint for developing similar models and to provide a better traceability between empirical data and simulation model. The key question is: Which type of data was integrated at which point during development? Table 4.3 gives an overview over used empirical knowledge. The modeling stages follow the methodology by Rus et al. [RNM03].

Modeling Stage	Data Source	Usage
Static Process Model Creation	[LD00, MBH+02]	Both sources did not provide empirical data. They were used for creating a static process model that represents a well-known inspection process. (→ Section 4.3)
Influence Diagram Creation	[Arm02]	This source provided qualitative information on which factors influence inspections.
	[BGL+96, CDLM97]	These two sources provided the reference data for the relations between *iDDR* and *dDDR* as well as general context information.
	[Bif00]	This source provided information about the influence of experience and expertise on the *iDDR* values as well as the possible need to change their relative importance.
	[BLW97]	This source provided information about the influence of document size and complexity on the *iDDR* values. (→ Section 4.5)
Dynamic Model Creation	[BGL+96, CDLM97, Bif00, BLW97]	The same data sources as during the creation of the influence diagram were used, because the dynamic model represents the influence diagram translated into the modeling environment. (→ Sections 4.6, 4.7, 4.8)
Model Calibration	[BGL+96, CDLM97]	The quantitative relations between *iDDR* and *dDDR* were derived from these real experiments. Determining the value of *RevCor* and the exponential decrease in *iDDR* weighing was also done comparing simulation results with these sources. The data was used to determine model equations and as input data for calibration. (→ Section 4.9)
Model Validation	[BG01]	This source provided empirical data with a slightly different context, so the model could be validated. The data was used as input data for the simulation and for comparing simulation results. (→ Chapter 6)

Table 4.3: Overview over used empirical knowledge

4.12 Model Limitations

The simulation model has several limitations. It describes a very abstract inspection process and includes only few factors. A standard inspection process is influenced by far more factors [Arm02]. Most of them are not taken into account in this model.

This initial version of the model relies on average defect detection ratios only. While this seems to work fine for homogenous groups of reviewers, problems arise when reviewers with very different iDDRs are combined. With this approach for adjusting each reviewer's iDDR, the result may depend on the time each reviewer starts. By repeatedly running the simulation with randomized starting times of the reviewers, an acceptable average simulation result should be reached nevertheless.

A third limitation concerns the *iDDR Correction* equation. It is based mainly on data from the three simulated experiments, testing and intuition. It needs to be evaluated in other contexts, with more and less reviewers, and with different iDDR values. It served the purpose well in this model, but may need to be altered or replaced for other experiments.

The adjustment variables' influence is largely unknown as well. At the moment, all factors are included in the equations in a linear manner. This raises questions like "With iDDR data available for a certain document size, will iDDR decrease in a linear way with increasing document size?" These questions will need to be answered in order to allow a reasonable and reliable scale-up.

Finally, the model was calibrated using data from three (similar) experiments only. Although the model performed well in a first validation with real-world data (see Chapter 6), all equations, correction factors, and assumptions of this model need to be cross-checked in order to come to a more generally usable simulation model.

4.13 Lessons learned

I have learned several important things with respect to using empirical knowledge during the creation of this model. A lot of information that would have been helpful for creating the simulation model was not available. This was not because it is difficult to gather it, but because I used data from experiments not designed to fit the needs of a simulation model. With only little more effort, the desired data could have been collected, which would have simplified model building a great deal. This might be an issue for the design of future experiments.

Furthermore, deriving valid equations from discrete data points as provided through experiments is also not an easy task. Especially the combination of the RevCor factor and the iDDR Correction equation proved to be difficult. Unfortunately, the experiments themselves could not provide much helpful information. This was not because of the experimental setup, but because of the lack of determined interrelations between influence factors and the amount of different contexts. I tried to overcome this through extensively testing out ideas, but I failed to provide a more systematic approach.

The empirical knowledge I used mainly resulted from one experiment and two replications. Although this was considered in the design of the replications, the contexts were not completely identical. Additionally, I used empirical knowledge from other empirical studies for identifying further impact dependencies. The context of these studies was also different. It seems to be a challenging research field to investigate to which extent context variations are acceptable and what the consequences for the external validity of the model are (further details about threats to validity can be found in [Kra00]). Guidance for this topic is missing. From my point of view, a careful commonality analysis of the contexts could be a starting point.

Finally, using a visual modeling tool such as the one I used proved to be a great help. Visualizing the model modules and their connections allowed for effectively developing the model. A text- or source code-based model would have been significantly harder to develop.

5 Simulation Run

To simplify model usage in real-world environments, this section exemplarily executes on complete simulation cycle. This includes model setup, a simulation run and result identification. First, the simulation model is set up with input data from the real experiments, and then the simulation is run. The three original experiments (generic problem domain) are simulated. For simplification, the results of both documents are combined as during model calibration. Finally, the results file is explained. All input data described here can be found in Appendix A. This should simplify replaying this exemplary simulation run as well as setting up other simulations.

5.1 Scope of Validity of the Model

The simulation model describes a requirements inspection process. A document is inspected by an arbitrary number of reviewers in a classroom setting. Each reviewer works independently from the others. The results of all reviewers are combined, but no real team meetings are held. The reviewers' capability is reflected in their individual defect detection rate. This means that if the individual defect detection rates are known, the model can be used with arbitrarily capable reviewers, and if the defect detection rates are unknown, assumed values can be used in combination with adjusting other model variables. Model calibration and validation, however, was done primarily with data from undergraduate students, so caution is advised when simulating reviewers with a highly different background.

The simulation model assumes that there is sufficient time to thoroughly conduct defect detection. Nevertheless, time pressure can be accounted for through adjusting the document coverage value. The inspected documents were from a generic domain, with 17 pages, each containing an average of 1.7 defects. Model validation suggests that a 50% increase in defect density and a 100% increase in document size do not impair model validity.

5.2 Simulation Model Preparation

First, the documents that will run through the simulation model need to be defined. This is done in the *Documents* block. Since three reviewers are to inspect each document, the *maximum number of reviewers per document* is three. Following the recommendation from Section 4.10.1, documents start at every 3+2=5 time units. The number of *Defects* is set to 28 as the aver-

age of the two documents, while *Size* and *Complexity* attribute values remain 1, because these document properties are already reflected in the reviewer iDDR values.

Next, reviewers need to be prepared. This is done in the *Reviewers* block of the model. Every document is to be inspected by three reviewers. The first reviewer of every group starts one time unit after the respective document, with the others following one by one. *Experience, Expertise* and *DOC Coverage* are set to 1, because these attributes are also already reflected in the reviewer iDDR values. After each set of reviewers, a reviewer with 0 for each of the three attribute values is scheduled, as explained in Section 4.10.2.

Finally, the *iDDR Input* values must be set. This is done directly in a text file that is read at the beginning of the simulation. The first column of the file contains values for each reviewer's *iDDR*, the second the *RevCor* value. The third and fourth columns contain the values for *Experience* and *Expertise Importance.* Since only average values are available from the real experiments, the iDDR values for the reviewers are identical within groups. The data stems from the original experiments, so *RevCor* values are constant here, too. The relative weighing of experience and expertise remains equal because there is no special emphasis on either one. Sets of reviewers are divided by a row with zeros for all values.

5.3 Simulation Results

After setting up all necessary model parameters, the simulation can be started. The experiments were simulated in their timely appearance, which is denoted in Table 4.2, for example. The documents exiting the simulation contain information about the document DDR (*dDDR*) and the remaining *Defects*. More detailed information is recorded by the *Discrete Event Plotter* module (see Figure 5.1). The first column records *reviewer iDDR* values. The standard setting only records changed values. This is why only six values are displayed at simulation times 1, 6, 11, 16, 21, and 26 (all reviewers within one group have the same iDDR values). At simulation times 4, 9, 14, 19, 24, and 29, the zero values from the dummy reviewers are recorded.

[131] Plotter, Discrete Event

Point Number1	T1	DDR 2		3 T3	Total DDR 4	Ti4	Remainin...
0	1	0,2464		0	0	0	28
1	4	0		2	0,2464	2	21,1008
2	6	0,3214		3	0,404096	3	16,685312
3	9	0		4	0,50502144	4	13,85939968
4	11	0,2133		5	0	5	28
5	14	0		7	0,3214	7	19,0008
6	16	0,2593		8	0,527096	8	13,241312
7	19	0		9	0,65874144	9	9,55523968
8	21	0,14585		10	0	10	28
9	24	0		12	0,2133	12	22,0276
10	26	0,2246		13	0,349812	13	18,205264
11	29	0		14	0,43717968	14	15,75896896
12				15	0	15	28
13				17	0,2593	17	20,7396
14				18	0,425252	18	16,092944
15				19	0,53146128	19	13,11908416
16				20	0	20	28
17				22	0,14585	22	23,9162
18				23	0,239194	23	21,302568
19				24	0,29893416	24	19,62984352
20				25	0	25	28
21				27	0,2246	27	21,7112
22				28	0,368344	28	17,686368
23				29	0,46034016	29	15,11047552

Figure 5.1: Simulation results plot

The second column is not used. The third column contains the values from the *DOC DDR* holding tank at the respective times, separated by zeros when the tank is reset for a new document. These values can also be found in the *SimulationResults.txt* file, which is written by the *File Out* module. Here, each simulation step results in a new row. The non-zero rows contain the respective *DOC DDR* values from the third plotter column.

The fourth plotter column contains the values calculated for the number of *remaining defects* in the document. The values are linked directly to the *DOC DDR* values from the third column. Table 5.1 gives an overview over simulation document DDR, remaining defects, (real-world) experiment document DDR and the deviation of reality and simulation. The average deviation between simulation results and reality is 1.99 percentage points of DDR.

Simulation dDDR	Remaining Defects	Experiment dDDR	Reality - Simulation
50,5%	14	48,2%	-2,3%
65,87%	10	62,4%	-3,47%
43,72%	16	41,9%	-1,82%
53,15%	13	52,75%	-0,4%
29,89%	20	32,2%	2,31%
46,03%	15	47,7%	1,67%

Table 5.1: Simulation run results overview

6 Validation

Since the model was developed to be used in a real-world environment, a test of the model was needed. This is to verify the equations derived from the calibration data and to evaluate model behavior under different circumstances. A good test data set would have similar context, but slightly changed compared to the calibration data. [BG01] provides such a data set.

6.1 Simulation Model Preparation

As a first test in a different environment, the calibrated extended model was tested with data from another real experiment [BG01]. The experiment originally analyzed the influence of team size and defect detection technique on the inspection effectiveness of a nominal team. Valid data from 169 subjects could be collected. The subjects were all undergraduate students with some software engineering experience. The experimenters distinguished subjects using a checklist-based approach (denoted with "C" for the rest of this chapter), a scenario-based approach using the user viewpoint ("A"), the designer viewpoint ("D") and the tester viewpoint ("T"). All experiment runs were preceded by appropriate trainings.

Since the focus of the experiment was on reading technique mixes and team size, iDDR and team DDR data was only available for two-person teams. This creates a good test for the calculation of the document DDR value. The knowledge of the test subjects can be compared to the subjects from the replications used for calibration, since both were undergraduate students. In both cases, there was a classroom setting. The inspected document was a requirements document as well. It was about twice as large (35 compared to 16/17) as the calibration documents and contained about three times as many defects (86 compared to 27/29). This makes for an average of 2.46 defects per page, which is 45% higher than the calibration documents (1.7). So, there are context changes, but only slight ones so the model may perform well in the changed context as well.

For testing purposes, only combinations of two reviewers using the same reading technique were entered into the model. This should prevent spoiling simulation results due to combining different reading techniques. Translated into model parameters, this means that four equal documents are inspected by two reviewers each: First AA (two "A" reviewers), then TT, CC, and finally DD. All other parameters kept their original values, including *RevCor*. The specific values of all simulation parameters can be found in Appendix B.

6.2 Simulation Results

After preparation, the simulation was run. Four documents were inspected by two reviewers each. The results can again be found in the *Simulation-Results.txt* file and, more detailed, in the *Discrete Event Plotter* module.

The simulation run resulted in document DDR values very close to the experiment values. Table 6.1 gives an overview. The rows contain the values for the AA, TT, CC, and DD reviewer groups. The average deviation between simulation results and reality is 1.1 percentage points of DDR, which equals a maximum deviation of 3.3 percent. This is a very good result, suggesting that the simulation model indeed decently reflects reality.

Simulation dDDR	Remaining Defects	Experiment dDDR	Reality - Simulation
26,24%	63	27,0%	0,76%
24,11%	65	25,6%	1,49%
32,96%	58	33,5%	0,54%
24,11%	65	25,7%	1,59%

Table 6.1: Model validation results overview

7 Conclusions

7.1 Summary

This work presented a discrete-event simulation model of an inspection process. The model is based on empirical data from an inspection experiment at the NASA/GSFC Software Engineering Laboratory and two replications at Kaiserslautern University. The experiments provided data about average individual defect detection rates of the reviewers and average total team defect detection rates. While the experiments examined both NASA-specific and generic type documents, only the generic part was considered for the model.

The simulation model reconstructs the inspection process in an abstract form. Several adjustment variables allow for future adaptation of the model to different contexts. The model itself is based on average defect detection ratios.

The first chapter provided an introduction into this work. The second chapter gave an overview about empirical software engineering and introduced and structured different types of real and virtual experiments. The third chapter explained the three main possibilities of combining real and virtual software engineering experiments, and pointed out typical problems that can be expected when doing so.

The fourth chapter implemented an exemplary simulation model using empirical knowledge from real experiments. This was done following a methodology proposed by Rus et al. [RNM03]. All modeling steps are explained in detail, as well as an extension to better support real-world application of the model. Model limitations and lessons learned round off the chapter. The fourth chapter serves as a guide to creating other models. Finally, an exemplary simulation run is described in the fifth chapter, with detailed explanation of the steps necessary for setting up the model and interpreting simulation results. Chapters six and seven provide a real-world model test and an outlook to one possible usage of the model.

The model results lead to the conclusion that the model does represent reality to a certain degree. Nevertheless, it should not be forgotten that many details were abstracted from. Additionally, only three real experiments were used as calibration data. However, the model performed well in a first test with real-world data. It has become clear that integrating empirical data into models is a feasible way to obtain good simulation models. Furthermore, the methodology introduced by Rus et al. [RNM03] provides a

systematic help for developing the model. Although the method is intended for modeling larger processes from real-world businesses, the suggested steps towards a model helped in systematically developing the model. Of course, some steps were left out, e.g., customer interviews. Nevertheless, the method proved to support modeling processes in practice.

It took about three person weeks to develop the simulation model, including the initial calibration and the extensions for the multi-document, multi-reviewer capabilities, with about one extra week for tool familiarization. An arbitrary amount of time can be spent for further calibration with different data, of course. Up to now, no information is available on whether the model will pay off, and how many real experiments need to be replaced by simulations until break-even. This is also closely related to the reliability of the simulation results.

Using empirical knowledge during simulation model development provided several advantages. While the static model can be developed using expert knowledge, determining the quantitative relations seems rather difficult without real-world data. Using the same analogy as in Section 3.2.5, the static process model is the black-and-white photograph, whereas the empirically enriched dynamic model represents the color video. The same advantages apply for calibration and model adaptation: When the process context changes, e.g., because other employees are conducting the inspection, this change can easily be reflected in the simulation model through varied input parameters. An example for this is given in Chapter 6. Finally, the incremental approach is supported well, too. The first raw version of the model presented in this thesis was developed using empirical knowledge from the original experiments only. When relations had gotten clearer, more data from the other experiments was included to refine the model. Ultimately, this approach led to the multi-document, multi-reviewer simulation model.

Several problems appeared during modeling and integration of empirical knowledge. The original experiments did not provide all the data that would have been helpful for the model. Also, deriving valid equations from discrete data points is not an easy task and requires extensive explorative studies.

7.2 Further Usage of the Simulation Model

As already mentioned earlier, simulations can be used to evaluate (slight) context changes in experiments. One experiment may provide data for a certain context, another one for a slightly different context. Using simulation technology, these two isolated spots may be expanded and combined into a larger field where certain predictions can be made through simulations.

Figure 7.1 depicts this situation. Two real experiments supply information about the areas denoted with “A”. Although this is certainly valuable information, it only concerns isolated spots in the much larger field of possible effects. Now, through replicating each isolated real experiment using simulation, these areas may be expanded and enlarged through slight variation of simulation parameters. This could lead to decent knowledge about the “B” areas. Since it has not been gained through real experiments, it is only reliable to a certain degree. Determining this degree (“The statements in area “B” are true with a probability of 0.8”) could be a field of future research.

Combining several real experiments with simulations could lead to an even larger area of understanding, denoted with “C” in Figure 7.1. Isolated data bits could serve as simulation input to cover a larger area. Of course, the degree of trust for the “C” area would probably be lower than for the “B” areas. Nevertheless, there was at least some information for an area that was completely uncharted before.

This is only one possible use of the combination of real experiments and simulation. A lot of research is still necessary before a map like Figure 7.1 can be drawn quantitatively. Combining different maps into one single, more comprehensive model of the software development process requires further research. Nevertheless, the benefits seem promising, so this goal should be pursued further.

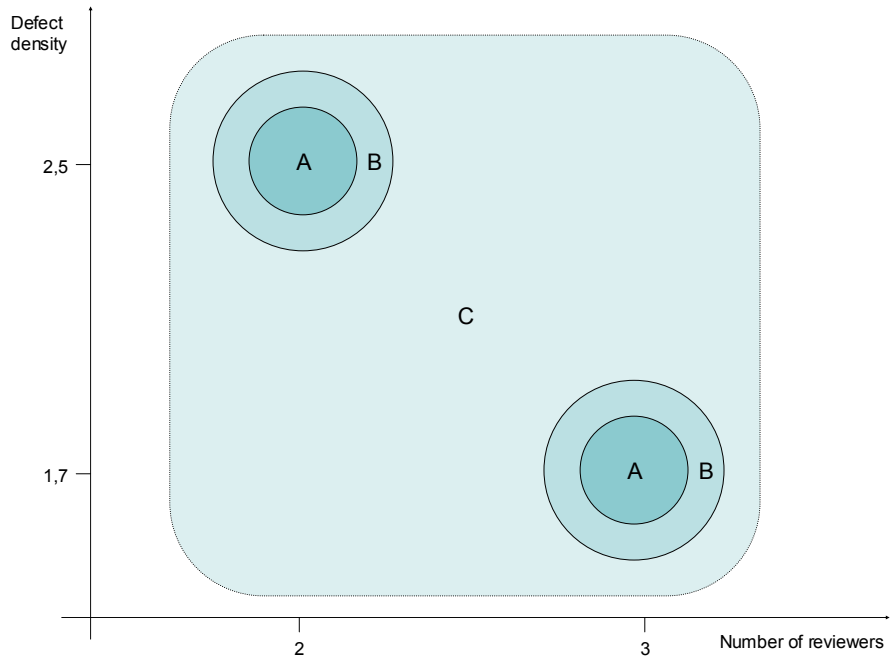


Figure 7.1: Expanding understanding of parameter change effects

7.3 Outlook

A possible direction for future research concerning the model itself would be the *iDDR Correction/RevCor* area. More data from different experiments is needed to determine the validity of the proposals presented in this work. Another important direction would be to explore the possibilities of adapting the model to different contexts. Here, especially the integration of the "adjustment variables" could be enhanced. Generally, more knowledge about the quantitative relations of influence factors is needed.

Other open research questions concern better methodological support for combining empirical data and simulation modeling. As an example, controlled experiments could be designed specifically for simulator development, for example. When conducting case studies, simulation support could be improved by recording more context and individual data. This could lead to a new and advanced type of experimental laboratory that uses process simulation as a virtual capability.

Finally, determining the reliability of the results gained through simulation as addressed in Section 7.2 is important for the usage of simulation results. To what degree do they actually match reality, how much do they typically differ? Questions like these need to be answered in order to take full advantage of the combination of real experiments and simulation, and to analyze the potential pay-off of the simulation model.

Acknowledgements

I would like to thank Jürgen Münch whose comments substantially improved this work. I am also grateful for Andreina Byrd and Simone Wittke reviewing this thesis.

List of figures

List of tables

Bibliography

[1299] *The Journal of Systems and Software – Special Issue on Process Simulation Modeling*, volume 46. Elsevier Science Inc., 1999.

[AHM91] Tarek Abdel-Hamid and Stuart E. Madnick. Software Project Dynamics. *Prentice Hall, Englewood Cliffs, NJ,* 1991.

[Arm02] Ove Armbrust. Developing a Characterization Scheme for Inspection Experiments. *Project Thesis,* Kaiserslautern University, 2002.

[Bas92] Victor R. Basili. The Experimental Paradigm in Software Engineering. In H. Dieter Rombach, Victor R. Basili, and R. W. Selby, editors, *Experimental Software Engineering Issues: A critical assessment and future directions*, 706, pages 3–12, September 1992.

[BCR94a] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Experience Factory. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 469–476, 1994.

[BCR94b] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Goal Question Metric Paradigm. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 528–532, 1994.

[BG01] Stefan Biffl and Walter Gutjahr. Analyzing the Influence of Team Size and Defect Detection Technique on the Inspection Effectiveness of a Nominal Team. In *Proceedings of the 7th International Software Metric Symposium*, April 2001.

[BGL+96] Victor R. Basili, Scott Green, Oliver Laitenberger, Filippo Lanubile, Forrest Shull, Sivert Sørumgård, and Marvin V. Zelkowitz. The Empirical Investigation of Perspective-based Reading. *Empirical Software Engineering*, 1(2), pages 133–164, October 1996.

[Bif00] Stefan Biffl. Analysis of the Impact of Reading Technique and Inspector Capability on Individual Inspection Performance. In *Proceedings of the 7th Asia Pacific Software Engineering Conference (APSEC) Singapore*, December 2000.

[BK01] Ulrike Becker-Kornstaedt. Towards Systematic Knowledge Elicitation for Descriptive Software Process Modeling. In F. Bomarius and S. Komi-Sirviö, editors, *Proceedings of the Third International Conference on Product Focused Software Processes Improvement (PROFES)*, 2188, pages 312–325. Lecture Notes in Computer Science, September 2001.

[BLW97] Lionel C. Briand, Oliver Laitenberger, and Isabella Wieczorek. Building Resource and Quality Management Models for Software Inspections. *Technical Report 97-06, Fraunhofer* Institute for Experimental Software Engineering, 1997.

[Bro95] Frederic P. Brooks. The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition. *Addison-Wesley,* 2nd edition, 1995.

[BSH86] Victor R. Basili, Richard W. Selby, and David H. Hutchens. Experimentation in Software Engineering. *IEEE Transactions on Software Engineering*, SE-12(7), pages 733–743, July 1986.

[BW84] Victor R. Basili and David M. Weiss. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering*, pages 728–738, November 1984.

[CDLM97] Marcus Ciolkowski, Christiane Differding, Oliver Laitenberger, and Jürgen Münch. Empirical Investigation of Perspective-based Reading: A Replicated Experiment. *Technical Report 048.97/E,* Fraunhofer IESE, December 1997.

[Fag76] Michael E. Fagan. Design and Code Inspections to reduce Errors in Program Development. *IBM System Journal*, 15(3), pages 182–211, 1976.

[Fli02] Rudolf Flierl. Verbrennungsmotoren. *Lecture at Kaiserslautern University,* 2002.

[Hag03] Hans Hagen, editor. Virtuelle Laboratorien. *Universität Kaiserslautern, Fachbereich Informatik, Konzeptpapier für die Einrichtung eines DFG-Sonderforschungsbereiches,* 2003.

[Jos88] Mathai Joseph. Software engineering: Theory, Experiment, Practice or Performance. *Technical report,* University of Warwick, 1988.

[KMR99] Marc I. Kellner, Raymond J. Madachay, and David M. Raffo. Software Process Simulation Modeling: Why? What? How? *The Journal of Systems and Software*, 46, pages 91–105, 1999.

[Kra00] David Krahl. The Extend Simulation Environment. In J.A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, editors, *Proceedings of the 2000 Winter Simulation Conference*, pages 280–289. IEEE Press, 2000.

[Lad94] Peter B. Ladkin. Report on the Accident to Airbus A320-211 Aircraft in Warsaw, 1994. http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Warsaw/warsaw-report.html. Last visited 2002-12-04.

[LD00] Oliver Laitenberger and Jean-Marc DeBaud. An Encompassing Life-cycle Centric Survey of Software Inspection. *Journal of Systems and Software*, 50(1), pages 5–31, 2000.

[Lur02] Marty Lurie. Winning Database Configurations: An IBM Informix Database Survey. http://www7b.software.ibm.com/dmdd/zones/informix/library/techarticle/lurie/0201lurie.html; Last visited: 2003-01-21, January 2002.

[MBH+02] Jürgen Münch, Thomas Berlage, Thomas Hanne, Holger Neu, Stefan Nickel, Sascha von Stockum, and Andreas Wirsen. Simulation-based Evaluation and Improvement of Software Development Processes. *Technical Report SEV Progress Report No. 1, Technical Report No. 048.02/E,* Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, 2002.

[McC76] Thomas McCabe. A Software Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4), pages 308–320, 1976.

[Mey02] Angela Meyer. Wer verdient wie viel? http://www.heise.de/ct/02/06/110/; Last visited: 2003-01-21, February 2002.

[Mün02] Jürgen Münch. Process Modeling Tool Structure. *Process Modeling Lecture,* Kaiserslautern University, 2002.

[MPSV96] Patricia McCarthy, Adam A. Porter, Harvey P. Siy, and Lawrence G. Votta. An Experiment to Assess Cost-benefits of Inspection Meetings and their Alternatives. In *Proceedings of the International Metrics Symposium*, pages 123–134, 1996.

[MR00] Robert H. Martin and David A. Raffo. A Model of the Software Development Process using both Continuous and Discrete Models. *International Journal of Software Process Improvement and Practice*, 5(2/3), 2000.

[Nav02] Emily O. Navarro. SimSE – An Educational Software Engineering Simulation Environment, 2002. http://www.ics.uci.edu/emilyo/SimSE/main.html. Last visited 2002-12-04.

[NHM+02] Holger Neu, Thomas Hanne, Jürgen Münch, Stefan Nickel, and Andreas Wirsen. Simulation-based Risk Reduction for Planning Inspections. In Markku Oivo and Seija Komi-Sirviö, editors, *Proceedings of the 4th International Conference on Product Focused Software Process Improvement (Profes 2002)*, 2559, pages 78–93. Lecture Notes in Computer Science, December 2002.

[NHM+03] Holger Neu, Thomas Hanne, Jürgen Münch, Stefan Nickel, and Andreas Wirsen. Creating a Code Inspection Model for Simulation-based Decision Support. In *Proceedings of the International Workshop on Software Process Simulation and Modeling (ProSim),* Portland, Oregon, USA, May 2003.

[oE94] United States Department of Energy. Human Radiation Experiments. http://tis.eh.doe.gov/ohre/; Last visited 2003-01-21, 1994.

[Pfa01] Dietmar Pfahl. An Integrated Approach to Simulation-based Learning in Support of Strategical and Project Management in Software Organizations. *PhD thesis,* Kaiserslautern University, 2001.

[RBH02] Ioana Rus, Stefan Biffl, and Michael Halling. Systematically Combining Process Simulation and Empirical Sata in Support of Decision Analysis in Software Development. *Technical report, SEKE,* 2002.

[RBS93] H. Dieter Rombach, Victor R. Basili, and Richard W. Selby. Experimental Software Engineering Issues: Critical Assessment and Future Directions. *Lecture Notes in Computer Science.* Springer Verlag, 1993.

[RKP+99] David Raffo, Timo Kaltio, Derek Partridge, Keith Phalp, and Juan F. Ramil. Empirical Studies Applied to Software Process Models. *Empirical Software Engineering*, 4(4), pages 353–369, December 1999.

[RNM03] Ioana Rus, Holger Neu, and Jürgen Münch. A Systematic Methodology for Seveloping Discrete Event Simulation Models of Software Development Processes. In *Proceedings of the International Workshop on Software Process Simulation and Modeling (ProSim 2003),* Portland, Oregon, USA, May 2003.

[Rom01] H. Dieter Rombach. Software Engineering 2. *Lecture Notes,* Kaiserslautern University, 2001.

[Som87] Ian Sommerville. Software Engineering. *Addison-Wesley,* 1987.

[Vot93] Lawrence G. Votta. Does Every Inspection Need a Meeting? In *Proceedings of the first ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 107–114, December 1993.

[WSH+00] Claes Wohlin, Per Suneson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. Experimentation in Software Engineering: An Introduction. *Kluwer Academic Publishers,* 2000.

Appendix A: Simulation Run Data

Output time	Defects	Size	Complexity
0	28	1	1
5	28	1	1
10	28	1	1
15	28	1	1
20	28	1	1
28	28	1	1

Table A-1: Documents input

Output time	Experience	Expertise	DOC Coverage
1	1	1	1
2	1	1	1
3	1	1	1
4	0	0	0
5	1	1	1
6	1	1	1
7	1	1	1
8	0	0	0
10	1	1	1
11	1	1	1
12	1	1	1
13	0	0	0
15	1	1	1
16	1	1	1
17	1	1	1
18	0	0	0
20	1	1	1
21	1	1	1
22	1	1	1
23	0	0	0

Table A-2: Reviewers input

iDDR	RevCor	Eclm	Etlm
0,2464	0,64	1	1
0,2464	0,64	1	1
0,2464	0,64	1	1
0	0	0	0
0,3124	0,64	1	1
0,3124	0,64	1	1
0,3124	0,64	1	1
0	0	0	0
0,2133	0,64	1	1
0,2133	0,64	1	1
0,2133	0,64	1	1
0	0	0	0
0,2593	0,64	1	1
0,2593	0,64	1	1
0,2593	0,64	1	1
0	0	0	0
0,14585	0,64	1	1
0,14585	0,64	1	1
0,14585	0,64	1	1
0	0	0	0
0,2246	0,64	1	1
0,2246	0,64	1	1
0,2246	0,64	1	1
0	0	0	0

Table A-3: iDDR input

Appendix B: Validation Data

Output time	Defects	Size	Complexity
0	86	1	1
4	86	1	1
8	86	1	1
12	86	1	1

Table B-1: Documents input

Output time	Experience	Expertise	DOC Coverage
1	1	1	1
2	1	1	1
3	0	0	0
5	1	1	1
6	1	1	1
7	0	0	0
9	1	1	1
10	1	1	1
11	0	0	0
13	1	1	1
14	1	1	1
15	0	0	0

Table B-2: Reviewers input

iDDR	RevCor	EcIm	EtIm
0,16	0,64	1	1
0,16	0,64	1	1
0	0	0	0
0,147	0,64	1	1
0,147	0,64	1	1
0	0	0	0
0,201	0,64	1	1
0,201	0,64	1	1
0	0	0	0
0,147	0,64	1	1
0,147	0,64	1	1
0	0	0	0

Table B-3: iDDR input

Appendix C: Terminology

Descriptive Process Model

A process model that is derived from the actual process. It describes the current process as it is seen by the modeler.

DDR

Defect detection rate. The defect detection rate is defined as the percentage of the defects found with respect to all defects.

Experience Factory (EF)

The Experience Factory provides a methodological model for the storage of knowledge gained, for example, in QIP circles. It contains an experiment database and project databases. The experience database contains process models, product models, project plans and other packaged information. Each project database contains "raw" products and measurement data. After project completion, this knowledge is analyzed, packaged and added to the experience database.

Experiment

A test, trial or tentative procedure policy; an act or operation for the purpose of discovering something unknown or of testing a principle, supposition, etc.; an operation carried out under controlled conditions in order to discover an unknown effect or law, to test or establish a hypothesis, or to illustrate a known law [Bas92].

iDDR

Individual defect detection rate. This describes the percentage of defects that a reviewer detects of all defects in a document.

Model

Abstraction, usually a simplification, of a real or imaginary system. The model represents only the aspects of the system deemed important for the current purpose. In most cases, this is only a fraction of the actually existing aspects.

Object, Experimental

The object of interest that is to be investigated in the experiment.

Process

Number of partially ordered steps to reach a goal. It may be refined into sub-processes.

Prescriptive Process Model

A process model that provides a plan that determines the course of action. A prescriptive process model is derived from theory and often symbolizes the ideal process and therefore needs to be adapted to reality. Example: The waterfall process model.

Subject, Experimental

Since software engineering is a human-based profession, humans need to carry out the experiment tasks. In this context, they are called experimental subjects.

Treatment

Single experimental run with specifically set parameters.

QIP

The Quality Improvement Paradigm has been introduced in 1984 [BW84] as a result of the research effort of NASA SEL and the University of Maryland, Department of Computer Science. QIP supports systematic planning and executing processes as well as analyzing the results and packaging the knowledge gained for reuse. QIP describes a methodological model for advanced learning in software organizations.

Research Domain

Well-defined science area where new insights are sought for by systematic effort. A number of research objects are identified by defining a research domain. Example: Software Engineering.

Virtual Laboratory

Controlled environment for conducting virtual experiments in a research domain. The (virtual) research object consists of a model of the real research object.

Virtual Experiment

Experiment where the research object is recreated (modeled) and examined (simulated, visualized) in a computer environment.