A Descriptive Model of Software Development to Guide Process Improvement

Kurt Schneider, FG Software Engineering, Universität Hannover

Abstract

Software process improvement can follow a capability maturity model such as SPICE or CMMi. Or it can focus on improving selected software processes in detail. The latter can either be a stand-alone improvement activity, or it can tune processes that were established or modified due to a large-scale assessment. Some software processes need to be adjusted qualitatively: process steps or deliverables need to be added, modified, or dropped from the process. In many cases, however, the complexity of software processes comes with their quantitative aspects: while testing is a good activity, for example, it must neither be carried out too long nor too short. Quantitative analysis of software processes requires an adequate model to capture relevant aspects. This paper presents a descriptive approach to model software development. It covers effort, size of deliverable, and quality, and it helps to better understand effects occurring in software projects. Key concepts and modeling examples are presented. Our FLOW project investigates opportunities to improve software processes based on optimizing project information and experience flows. A descriptive model of software project dynamics is an important ingredient.

1 Introduction

Software process improvement is often seen as a large-scale activity. Assessments and improvement approaches such as CMM and IDEAL have spread throughout the industry [1]. They often help to establish missing processes or to improve their capability. With extended models such as SPICE (ISO 15 504) or CMMi [2], more software-related processes are covered.

There has been an alternative approach to software process improvement: Based on Basili´s seminal work on the so-called Experience Factory [3], several researchers and a growing number of companies have pursued experience-based process improvement [4], [5]. In essence, experience-based process improvement does not come with a checklist of processes, but starts with an analysis of improvement goals. It studies existing processes, often using GQM-based measurement [6], [7]. Findings are usually very specific to the organization and offer a prioritized list of improvement suggestions. Those suggestions are based on an in-depth analysis of fine-grained processes. At the same time, experience-based approaches rarely cover as many processes as capability models do. And they never produce a rating similar to CMM levels.

In our FLOW research project, we have started to take experience and its flow just as seriously as most people take project information flow. Much like experience-based process improvement, we tend to look deeper at selected aspects. Processes are perceived as the carriers of project information flow, and of experience flow as well. Where any flows are disrupted, a process is flawed *qualitatively*. In many cases, however, a process works in principle, but flows are out of order due to *quantitative* reasons. Such flaws are hard to see and they tend to escape attention: quantitative effects are harder to model, to imagine, and to keep in mind. When we attempt to understand effects at this level, we need descriptive models of flows and their interactions over time and effort.

A descriptive model of software projects can serve several purposes: not only can it direct software process improvement (as outlined above); it can also be used to teach software engineering relationships and effects. In both analysis and education, a simpler and more intuitive model will work better than a more complex one.

A comprehensive, yet comprehensible descriptive model will help (1) project participants to reflect and reason about their work, (2) software process improvement supporters to focus their fine-grained activities, and (3) software engineering researchers to study and teach realistic software engineering effects. The challenge is to carry out the right activities, but also at an appropriate intensity and duration. A good software professional (project leader or developer) will take care of both.

Section 2 gives an overview of models used to capture software project dynamics descriptively. Model theory provides a theoretical background to discuss specific modeling requirements in Section 3. In Section 4, the Software Quantum Metaphor for descriptive models of software projects is explained. It was originally created for the SESAM educational game [8]. Section 5 argues how this metaphor can stimulate software process improvement when put into a conceptual framework like our FLOW project. Section 7 concludes by summarizing usage opportunities and potential impacts.

2 Models of Software Development

Whenever software development is described or discussed, a model is being built – either explicitely or implicitely. A model takes a small part of reality and looks only at a certain set of its aspects. In most cases, relevant aspects are mapped to a different domain during modeling. For example, the streets of a real city are mapped to lines on a piece of paper their names written on the “map”.

A software process model is one possible model of software projects. In this case, activities, deliverables and their relationships are among the aspects usually considered relevant. Only those relevant aspects will appear in the process model. The mapping from reality (project) to model (process model) will turn an activity into a labeled box (e.g. “design”) and the deliverables into ovals (e.g. “user manual”) [9] or other symbols. During its peak in the late nineties, the software process modeling community discussed process model notations at workshops and conferences. A majority of process models were built to *prescribe* processes, and enact them wherever possible.

2.1 Prescriptive and Descriptive

A *prescriptive* software process model tells project participants what to do in which order – and what results they are expected to produce. Osterweil claimed: “Software processes are software, too” – also highlighting their executable character. Workflow tools are a popular class of systems controlled by prescriptive models.

Descriptive models, on the other hand, do not tell anybody what to do – they try to represent what is actually being done. Prescriptive models often describe an ideal process. There are many reasons why actual processes will deviate from planned (prescriptive) process models: shortcuts and a lack of process understanding are frequent reasons. Descriptive models, on the other hand, are supposed to capture non-ideal properties of the processes as well. In fact, it is one of their main purposes to capture and show the weak spots of a given real process.

Descriptive models do not provide instant support to software professionals. Their impact is a more indirect one: They help people to better understand what is going on. A systematic analysis may uncover weaknesses that were never seen as long as there was no model.

2.2 Rare: Descriptive Models of Software Development

Many books on software engineering convey good lessons, but would not suffice to build a descriptive model. A descriptive model of software project dynamics needs to state how a project behaves under what circumstances, *be they desirable or not*. Mistakes must be covered. Usually, the model should be able to respond to a whole range of stimuli. Reactions of the model should be plausible: they should conform with our expectations about real project dynamics.

Abdel-Hamid was one of the first to build simulation models of software projects [10]. His System Dynamics models were mainly used to compare different “policies” in software management. A policy in System Dynamics is basically a function that formalizes a simulated response to some data. For example, the simulated number of defects in a model would cause (through a policy) more or fewer inspections to be carried out. By setting a threshold, one could study the impact of the “modified policy” during simulation runs.

Pfahl and co-authors have used a similar approach (e.g., in [11]). Their focus was on project planning. Simulations were carried out to compare different scenarios before the project starts.

While System Dynamics simulations are running in batch mode after the “policies” have been set, SESAM [8] simulated software projects and presented them as interactive simulation game. A project leader would interact with a simulated project, its simulated team members and simulated products. This approach required a descriptive model of action and reaction. The Software Quantum Metaphor was originally developed for the SESAM project and applied in its simulations [8]. Below, it will be put into a different context, the FLOW project.

During a series of software projects at Hannover University [12] we used a different kind of simulation: We picked four aspects of real software projects students should be able to experience in a University project. Among them was the separation of customer and software engineering coach roles: Customers would only talk about requirements, coaches would not know requirements but help students to fulfil them. In a University course, both roles are often played by the same advisor. By clearly

separating both concerns, we simulated real effects, and the behavior of the projects better resembled real projects. Other Universities have tried similar approaches. Approaches like those may seem similar to modelling software projects at first glance, they are fundamentally different in an important aspect: there are no explicit models nor behavior patterns. Similar effects (in the University) are provoked by similar stimuli, but there is no mapping or description of dependencies. For the purpose of this paper, we need explicit models.

2.3 Model theory

In the context of this work, it is highly important to understand precisely what happens during modeling and what constitutes a model. Therefore, a little glimpse on model theory is in place. A short excerpt from Stachoviak´s [13] thorough discussion of model theory will be sufficient.

1. The first step in modeling is to select a so-called “Original”. The Original is (or will be) part of the real world. When the model is a plan (e.g., of a house), the Original will only be built in the future.
2. Several aspects of the Original are selected as “relevant” (to the purpose of the modeling activity). For example, size of rooms is relevant for an architectural model.
3. A model is created, often in a different domain: A sheet of paper with lines on it represents the house.
4. All relevant attributes of the Original need to be mapped to the Model, while irrelevant aspects should be dropped.
5. At the same time, each Model will introduce additional properties. For example, the color (or prize or size) of paper has no meaning to the architectural model: it is not mapped from a relevant Original attribute.
6. Some of the additional properties or attributes are not relevant in the modeling sense, but they are handy which may be a reason to build the model in the first place: lines on a house plan can be easily (and inexpensively) be erased and redrawn – whereas tearing down walls is much more costly.

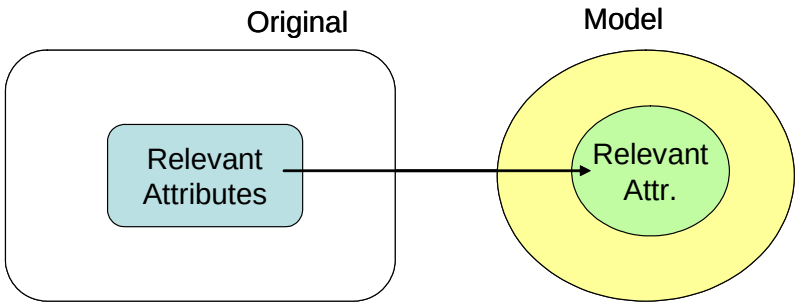

Figure 1: Original and Model. Relevant attributes are mapped.

When we accept the above foundation of modelling, it becomes obvious how models should be characterized, compared and evaluated.

Four questions should be asked about every model:

- What is the Original?
- Who will use the model?
- At what time?
- For what purpose?

A model should not simply be called “good” or “bad”, but rather evaluated with respect to the above four questions. A change in any of these facets will usually demand a new or adjusted model. For a more detailed analysis, one can look at the relevant attributes and how they were mapped.

With this little excerpt from Stachoviak´s [13] rich modeling theory, we are well-equipped to specify what we expect a descriptive model of software projects to do and to look like.

3 Requirements for a Descriptive Model

The original mission was to study and teach software engineering phenomena that are due to both qualitative and quantitative effects during a software project.

By looking at model theory, we can specify the models we are looking for:

Original. The original whose behavior we are trying to understand better is “a” software project. Of course, there are many different kinds of projects. For our teaching purposes, we would do with “any typical” project our students are likely to run into after finishing their degrees. In this sense, the Original is a *future* software project. For a specific company that tries to do a fine-grained process improvement, “a” software project should be typical *for that company*. It will be a good idea to explicitely name two or three past projects to frame this characterization and validate the models. Those selected projects, however, are not the Original!

Purpose. The main purpose is to understand qualitative-quantitative aspects of software project behavior in order to avoid future misconceptions. These insights should be easy to memorize and to repeat mentally. In a new situations, the model should be easily varied an used to investigate different lines of action. The model should be appropriate to be animated by a computer simulation.

Target users. Researchers, teachers, students, and practitioneers in companies should be able to work with the models, at least as mental models. Only researchers and teachers should necessarily be enabled to run simulations.

Time. We are looking at current projects (including agile and extreme programming methods) and their propperties. The models should be able to capture important 2004 software project trends.

Relevant attributes. Since we are interested in software process improvement, our main relevant attributes are process attributes:

- What activities, roles, and documents are used?
- How long and intense are those activities and what is the consequence?
- What is the resulting software quality?
- How much time and effort is spent?

In addition to these standard concerns, I would like to raise:

- Where are information bottlenecks?
- What activity is carried out at an insufficient experience level?
- Where have activities (not) reached a quantitatively satisfying level of completion?

Taken as requirements for a descriptive model of software projects, some challenges are obvious:

- When models are able to capture quantitative effects in order to simulate project behavior – such as Abdel-Hamid´s or Pfahl´s - they are usually not easy to understand and not appropriate as mental models.
- Relationships between effort, time, and quality may be stated from an empirical perspective (like in COCOMO [14]), but there is no mechanism that would explain their interrelationships – not even at a qualitative level.
- Models that are easy to understand (like explanations in many textbooks), they cover only isolated phenomena (like “adding more manpower to a late project”), but do not provide the bigger picture of project dynamics.
- Highly complex, computer-based simulations (like SESAM [8]) need substantial effort for model validation. This is only rarely possible in models for specific companies. When model users see only the behavior of the system, but cannot follow the model, validation in detail becomes even more important.

4 Software Quantum Metaphor

The “Software Quantum Metaphor” was first introduced in SESAM [8]. It allows quantitative simulation of an entire software project – with the exception of the project leader, who is a real person interacting with her or his simulated SESAM project. The project leader´s actions have an impact on the project; its later behavior depends on earlier decisions. Mistakes are possible, but have consequences. Since every team member, every document, and every event is simulated, the model had to be fairly comprehensive.

We experimented with the model and even applied it in a project management class at the University of Stuttgart before there was any computer support for the simulation: all calculations were done manually, with the help of a Spreadsheet program [15]. Later on, SESAM could carry out all interactions and calculations without manual intervention.

1.1 Principles of Software Quantums

The name “Software Quantum” comes from the concept of an atomic unit of project information. This unit was called a “software quantum”. It corresponded to about one tenth of a function point in size.

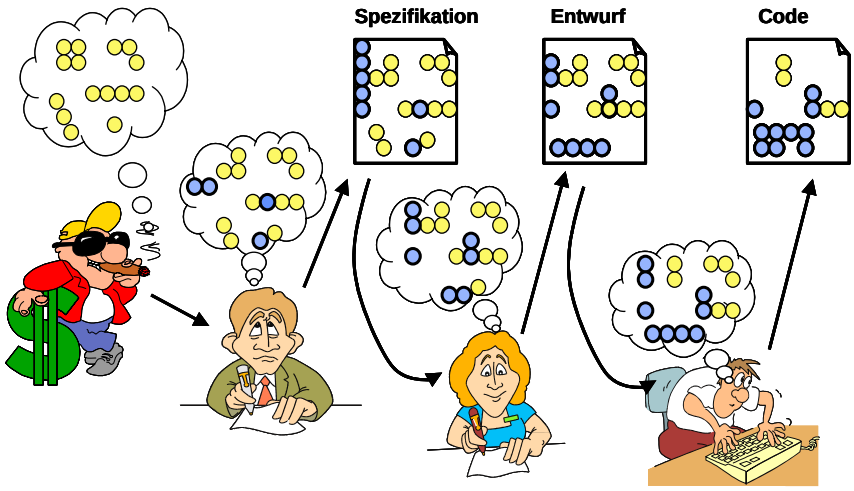

Figure 2: Flow of software quantums through people and documents

The Metaphor maps the entire requirements at the beginning of the project to a bag of balls (the visualization of software quantums). They are all yellow (printed in light gray), which stands for a real, legitimate requirement. When analysts interview the customer, they “grab a handful” of those balls and put it in their own “mental bag”. In this bag, they mix with pre-existing balls that they may have from previous projects in that same domain. Some are appropriate (yellow), others not (blue, printed as dark gray). The longer analysts interact with customers, the more balls they can grab. Formally they grab with putting balls back, since a customer does not forget requirements when he or she tells analysts about. On the contrary: there is a good chance that (important) requirements will be mentioned repeatedly. In this case, some of the yellow balls may already be present in the analysts mental bag.

Mathematically, there is a constant operation of random selection (with putting back) of n balls per time unit. Balls are individuals, so the same ball can be identified in the analyst´s mental bag. The longer the operation takes place, the more (different) balls will have been captured. However, there will be fewer and fewer new balls coming in every time. When plotted over time, the number of (different) balls looks like in the following Figure 3.

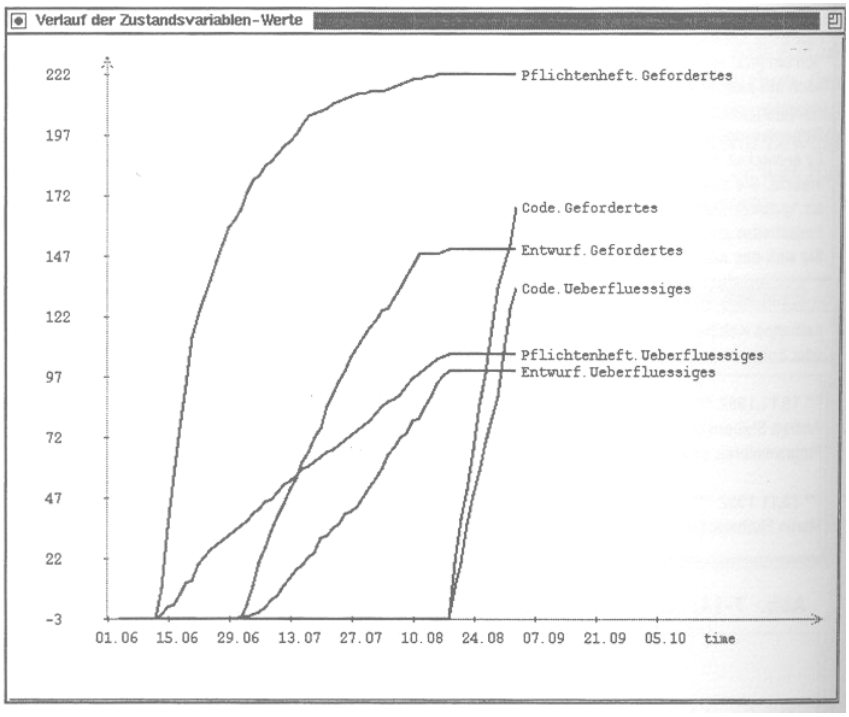

Figure 3: Attributes of simulated SESAM documents [8]

This plot shows the number of software quantums for three subsequent documents: specification (“Pflichtenheft”), design (“Entwurf”), and Code. Each document contains desired (yellow, “Gefordertes”) and undesired (blue, “Ueberflussiges”) software quantums. All curves saturate at the amount of yellow or blue quantums in their authors´ mental bags. Obviously, a subsequent document will hardly contain more yellow software quantums than its predecessor. Curves like in Figure 3 are a good discussion catalyst.

4.2 Assumptions

With the basic idea come a number of assumptions that make the model as useful as it is.

- Moving one software quantum from a mental bag to a document (or vice versa) takes the same amount of time and effort – indepent of its color.
- While grabbing quantums, one cannot see the color.
- During reviews, not the absolute color can be compared, but it can only be detected whether a given ball in the document under review was already present in the reference document.
- Some quality aspects can be defined with respect to comparing software quantum sets. For example, missing yellow quantums means missing functionality or non-functional aspects. Maintainability needs good coverage throughout all development documents: all documents need to have a large overlap with the final code document.
- Other quality aspects cannot be easily captured: Code quality (except for features) is not reflected, and thus, not treated as a relevant attribute.

- Due to their own misconceptions, involved people (software engineers) tend to continue doing what they do beyond a reasonable time. When there is no time limit, they will continue grabbing (the same) previous quantums over and over again. At the same time, they will pollute them with an increasing amount of preconceptions that may be correct or wrong.
- The size of a document is modeled as proportional to the amount of software quantums. Since there may be "blue" quantums involved, a bigger document is not necessarily be a better one. From the size, one cannot tell the quality or completeness.

4.3 Early use in SESAM

In SESAM, this model was animated by a Smalltalk simulation system [8].

Playing the game was an interesting experience. However, studying the games in hindsight was even more rewarding. When one knew the model and the assumptions, and could follow what had happended "inside the brains and documents", there would be many "aha!"s.

Even our early usage of the software quantum model [15] was rewarding for those of us who used the model. It seems, working with the model was the most interesting activity, not just looking at its behavior. Of course, it also implied a much deeper involvement which usually leads to deeper insights as well.

A potential that was never fully exploited in SESAM was the ability of the model to aid in mental experiments.

4.4 Why coming back to the idea

So far, the Software Quantum Metaphor has mainly be used internally within the SESAM system. I think, it should be treated as an important result in its own right. In this sense, SESAM is one application – but not the only one imaginable.

A model as simple as the Software Quantum Metaphor that can express so many aspects of software projects at the same time also has the potential to guide learning and process improvement.

- Learning, when a generic project is modeled.
- Process improvement, when the concrete sequence (or iteration) of activities and the flow of project information in a given project is modeled.
- In its current version, only one kind of "project information" is being represented by Software Quantums. Research in experience-based process improvement highlights the importance of understanding where experiences in software projects originate, and where experience is needed [16].
- Experience flows separately through one or many projects, and it should be covered with the metaphor. This could be done with different ball types or different bags to store them. Important is the notion of something else flowing in a different way and direction; something that is just as important as project information. In fact, experience can be a catalyst, improving the transmission function of a project activity: Someone with long experience in software design will be able to "grab faster" and "write faster", and maybe also act differently.

Discussions on the exact differences between an experienced and a non-experienced software designer can stimulate research and will be a good exercise for students – one they will remember. In a second-year software engineering class, I used the Software Quantum Metaphor repeatedly to illustrate what specific activities (prototyping, testing, iterative development, early reviews etc.) would imply in terms of Software Quantums.

4.5 Limitations of the Metaphor

Many aspects are not covered by the metaphor, and some restrictions apply.

Of course, content of documents is not modeled – except whether it deals with one of the initial requirements.

Each and every of the assumptions mentioned above imply restrictions. However, they were considered acceptable.

A rather severe modeling problem is the changing nature of balls in different documents and mental bags: A ball, say #17, in the requirements document cannot mean exactly the same as in the design document. What it really means is “the piece or aspect of design that addresses requirement #17”. This obviously might collide with the “same size and effort for all balls” assumption. A requirement will hardly cause the same amount of effort during every step of the software process. However, this has not caused problems so far, maybe due to the small size (and larger number) of software quantums: those effect may even cancel each other out in many cases. I suggest a pragmatic solution: I propose to use *both* (conflicting) modeling assumptions (same effort and continuing reference to requirement #17) as a (given) premise until there will be a better approach that resolves the (theoretical) conflict.

Changing requirements *can* be dealt with. There would just be additional yellow balls that would probably be missing in all the documents and would require repetition of many steps.

5 FLOW: The Bigger Picture

During the Software Engineering lecture, I used the Software Quantum Metaphor several times to visualize certain effects that combine qualitative and quantitative aspects. For example, the number and duration of reviews depends on the number of “yellow” and “blue” software quantums – and neither moderator nor any other participant can tell them apart. If you review a “clean” document, you waste money. If you stop reviewing early, you end up with a overly large and hard-to-handle document that is full of “blue”, unnecessary items. Without any review, your documents become completely useless, as more and more handling effort is wasted with blue software quantums.

In our FLOW research project, we want to study and improve flows of project information and flows of experience within software projects. Working in experience based process improvement [17], building Experience Bases [18], and observing how experience is mostly treated as second-class object, we wanted to create a richer picture.

Figure 4 sketches a situation in which two projects are running in parallel. The above project looks similar to the lower one, except for a few details. In both projects, a customer (head) debates with two analysts who then write a specification document.

When they are finished, designer (heads) take over and produce a design document. After this, work is split up to produce pieces of code, database interfaces, and other documents that are not worth labelling. During integration, code and database components are integrated and the System is produced.

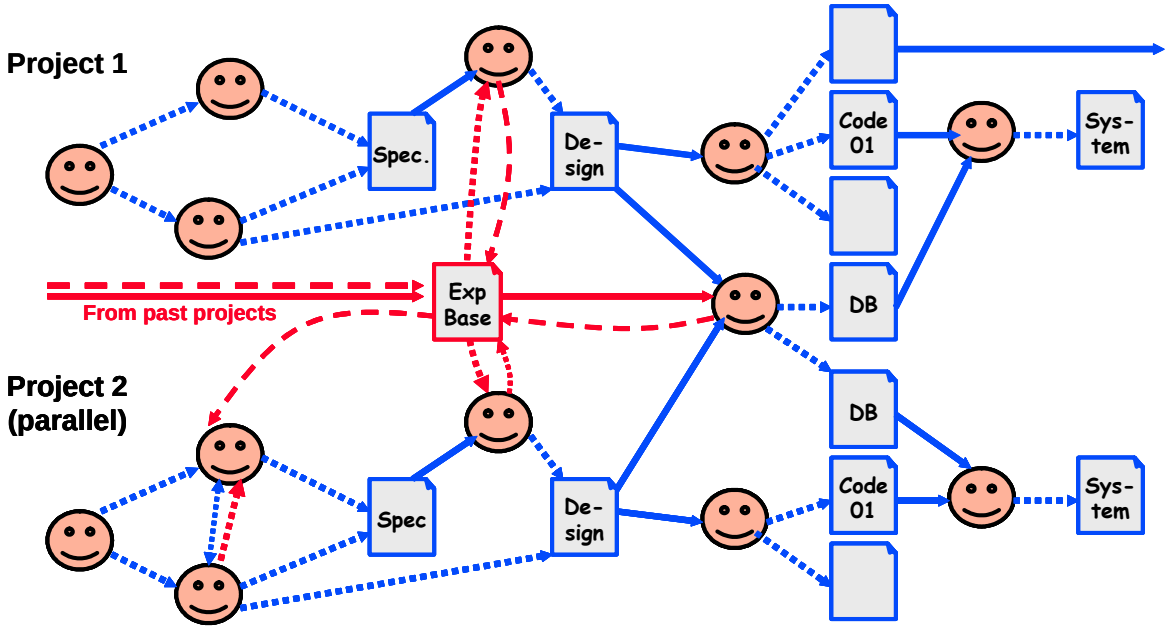

Figure 4: FLOW diagram of two parallel projects, including experience flow

Some details are different in the lower project: There is a newcomer in analysis who works with a more experienced person. Due to that fact, there is an additional project information line between the two, and an experience flow from the senior to the junior person (distinguished in color in the original; different shades of gray in Figure 4).

Designers of both projects exchange their experiences, as they are doing a similar job in different environments. The (sole) database (DB) expert uses an Experience Base (sketched as a document; would usually be a web repository) to exploit previous project experience. All stakeholders return their experience into the Experience Base. All processes around an Experience Base are far from trivial, and Figure 4 is a simplification [16]. It shows, where information flows, and a missing flow is a qualitative finding.

Obviously, flow (of both project information and experience) could be quantified with the Software Quantum Metaphor. SESAM did that in a single-project setting that did not care for experience. Adding more than one project and adding experience flows would enrich the model, but at the same time make its validation much more difficult. Therefore, I suggest a different approach: Plausible (as opposed to validated) quantitative effects would be sufficient to provide comprehensible mental models.

As a consequence, one would hardly expect to predict the future of a software project in all its phases and documents; but one would see that at a certain activity (say, design), experience flow needed a saturation period that was not granted. Our first, most pragmatic goal, in FLOW is to provide aids for understanding local effects qualitatively although those effects are dependent on (plausible) quantitative interactions of flows. A more ambitious goal will be to improve flows and interaction of flows. Earlier attempts in FOCUS [19] and LIDs [20] are encouraging for the development of techniques that look at experiences and flows explicitly.

6 Conclusions

Software process improvement at a fine granularity level is helpful to fine-tune software processes or to start an improvement initiative in a more focused way then through a big assessment or appraisal. For such a fine-grained improvement, a better understanding of quantitative reasons for some software engineering effects are needed.

I propose descriptive models of typical software projects as one way to face this challenge. There have been several earlier attempts in this direction, but we raise a set of requirements that point into a slightly different direction. Most important, models need to be both easy to understand, and at the same time able to express and combine and to explain how effort and time and quality are interrelated.

Obviously, meeting all those requirements in a fully validated simulation model is very ambitious and not the goal of our work in FLOW. At the same time, a rather simple modelling scheme, the Software Quantum Metaphor, provides easy-to-use modelling material that gets close to meeting many of the requirements raised.

Descriptive software quantum models can be used to visualize and to teach software engineering phenomena with a quantitative root. They can also be put in a much bigger picture, such as our FLOW project, in which recommendations for software process improvement can be derived – even from plausible models!

Software quantums have been used in simulations and Software Engineering classes, they have proven their potential for comprehensive, quantitative modelling. Their new use in FLOW calls for a more modest usage, but with increasing awareness for experience flows. We hope to stimulate other researchers and maybe software professionals who would like to use software quantums in their company for internal education.

Bibliography

1. Paulk, M.C., et al., *The Capability Maturity Model: Guidelines for Improving the Software Process*. 1 ed. SEI Series in Software Engineering, ed. M.C. Paulk, et al. Vol. 1. 1994, Reading, Massachusetts: Addison Wesley Longman, Inc. 441.
2. Dennis M. Ahern, A.C., Richard Turner, *CMMI® Distilled: A Practical Introduction to Integrated Process Improvement,*. 2001, Reading, MA: ,Addison-Wesley.
3. Basili, V., G. Caldiera, and R.H. D., *Experience factory*, in *Encyclopedia of Software Engineering*, J.J. Marciniak, Editor. 1994, John Wiley & Sons: New York. p. 469-476.
4. Kontio, J., G. Getto, and D. Landes. *Experiences in improving risk management processes using the concepts of the Riskit method*. in *Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*. 1998. Orlando, USA.
5. Landes, D., K. Schneider, and F. Houdek, *Organizational Learning and Experience Documentation in Industrial Software Projects.* International Journal on Human-Computer Studies (IJHCS), 1999. **51**(Organizational Memories).

6. Basili, V., G. Caldiera, and H. Rombach, *Goal question metric paradigm*, in *Encyclopedia of Software Engineering*, J.J. Marciniak, Editor. 1994, John Wiley & Sons: New York. p. 528-532.
7. Gantner, T. and K. Schneider. *Zwei Anwendungen von GQM: Ähnlich, aber doch nicht gleich*. in *MetriKon*. 2003. Ulm, Germany.
8. Schneider, K., *Ausführbare Modelle der Software-Entwicklung. Struktur und Realisierung eines Simulationssystemes.* Doctoral Dissertation. 1994, Zürich: vdf.
9. Bröhl, A.-P. and W. Dröschel, *The V-Model*. 1995: Oldenbourg-Verlag.
10. Abdel-Hamid, T. and S.E. Madnick, *Software Project Dynamics*. 1991, Englewood Cliffs, NJ: Prentice Hall.
11. Rodriguez, D., M. Satpathy, and D. Pfahl. *Effective software project management education through simulation models. An externally replicated experiment*. in *International Conference on Product Focused Software Process Improvement (PROFES)*. 2004. Kansai Science City, Japan: Bomarius, F.
12. Lübke, D., T. Flohr, and K. Schneider. *Serious Insights through Fun Software-Projects*. in *EuroSPI: European Software Process Improvement Conference*. 2004. Trondheim, Norway.
13. Stachoviak, H., *Allgemeine Modelltheorie*. 1973, Wien, New York: Springer Verlag.
14. Boehm, B., *Software Engineering Economics*. 1981: Prentice Hall, Englewood Cliffs, NJ.
15. Deininger, M. and K. Schneider. *Teaching Software Project Management by Simulation - Experiences with a Comprehensive Model*. in *Conference on Software Engineering Education (CSEE)*. 1994. Austin, Texas.
16. Schneider, K., *What to Expect from Software Experience Exploitation.* Journal of Universal Computer Science (J.UCS). www.jucs.org,, 2002. **8**(6): p. 44-54.
17. Schneider, K. *Active Probes: Synergy in Experience-Based Process Improvement*. in *Product Focused Software Process Improvement (PROFES 2000)*. 2000. Oulo, Finland: Springer.
18. Schneider, K. ; T. Schwinn, *Maturing Experience Base Concepts at Daimler-Chrysler.* Software Process Improvement and Practice, 2001. **6**: p. 85-96.
19. Schneider, K. *Prototypes as Assets, not Toys. Why and How to Extract Knowledge from Prototypes*. in *18th International Conference on Software Engineering (ICSE-18)*. 1996. Berlin, Germany.
20. Schneider, K. *LIDs: A Light-Weight Approach to Experience Elicitation and Reuse*. in *Product Focused Software Process Improvement (PROFES 2000)*. 2000. Oulo, Finland: Springer.

Author's biography

Kurt Schneider

Prof. Kurt Schneider studied Computer Science at the Friedrich-Alexander-Universität Erlangen-Nürnberg. He received his doctoral degree in Software Engineering from Universität Stuttgart. He held a Postdoctoral Research position at the Center for LifeLong Learning and Design at the University of Colorado at Boulder.

For about eight years, he worked at DaimlerChrysler Research Center Ulm in the realm of Software Process Improvement and Systematic Experience Exploitation. Since September 2003, Kurt Schneider is a Professor of Software Engineering at the Universität Hannover.