A Theory of Software Development

Robert Feldt

Department of Computer Engineering
Chalmers University of Technology
Gothenburg, Sweden

November, 2002

Abstract

We present a theory of software development as an incremental learning process. The focus is on the internal models of the developer. There are two main ways in which a development process can make progress: by refining an internal model or by refining an artefact based on an internal model. Refining the internal models is a prerequisite for being able to write a concrete specification and program that show acceptable behaviour. The theory has implications for tools to support software development. By creating novel test cases they can force the developer to question his internal models and realize where they are incomplete or incorrect.

1. Introduction

Our society increasingly relies on computers. Computers are controlled by software that tells them what to do. It is thus important that we theories of how to develop this software.

The fundamental problem of software development is that of refining mental pictures of how a problem could be solved into a program that a computer can run to actually solve the problem. The problems are typically ill-defined and only partly known. The mental pictures about possible solutions are similarly incomplete and fuzzy. Both problems and the mental pictures of how to solve them change over time. How should a software developer approach this situation?

We wish to consider general problems involving software development. To do this it is first necessary to identify the various elements involved and define them. We can then relate them in a model and define what we mean with a software development process and study which steps advances it.

In this paper we assume that the role of the program is fairly well known, ie. the development task is not about helping a problem owner explore the potential ways how a computer plus a program could solve the problem. We assume this connection

between a problem and a computer-based solution has already been established. Even though a more open-ended development situation may better reflect a majority of software development situations we here focus on the simpler problem while noting that more refined theories will be needed in the future.

2. Elements

Our model for software development has five fundamental elements[1]:

1. A *patron* which has a need for a program to solve a problem.
2. A *specification* which states how the program should and should not behave.
3. The actual computer *program* being developed.
4. A *developer* that develops the program.
5. A *library* containing humanity's total accumulated knowledge relevant to the development problem at hand.

In any software development process there are two fundamental types of information: the computer program itself and information about how the program should behave. The software itself is a natural element since we know that the developer has to write it. Another element is information about what the software should do. A specification is where the developer collects this knowledge.

The specification and the implementation are the two main artefacts in a software development process. A third artefact are any tests the developer runs on the software during development. Even though it is thinkable that in some development processes no tests are run, in practice there will always be some tests. The reason that tests are needed is because the implementation is a static artefact. To assess whether it fullfills the specification it has to be executed. However, tests are not fundamental elements of our model since we will see that they are proto-requirements. Requirements are part of the specification.

2.1. Specifications

Informally specifications are any information that specifies the form and behaviour of a computer program and the process by which it is going to be developed. To us

[1]In the rest of the paper we will alter between using he and she when referring to the developer and/or the patron. Of course it does not matter whether they are actually female, male or machines.

the form is not relevant since we are interested in principles and not the specifics of a particular programming language or the naming of parts of the system. Even though the development process is our topic of study it is not fundamental to a specification. Any requirements on the process by which we reach a final software product can only help or hinder development. In theory it should not affect the final behaviour of the system itself.

Fundamental to a specification is that it specifies the behaviour of the running program. Programs are static artefacts we develop to give a computer a certain, useful, dynamic behaviour. Any development effort would be meaningless if it did not pay attention to how the resulting system is supposed to behave. A specification can also describe situations that should never arise, for example inputs that are invalid or outputs that are forbidden. We call such specifications behavioural.

The atomic parts of a behavioural specification are behavioural requirements. They describe an invocation of the program and whether its a valid invocation or not. If its a valid invocation they also state information about different program behaviours in that situation. If the requirement is permissive it states what behaviour is valid. If it is dismissive it states what behaviour is invalid. Formally we have:

Definition 1 (Atomic Behavioural Requirement): An *atomic behavioural requirement*, denoted ρ, is a tuple $\{\sigma, \iota, o, \chi\}$ where

- σ describes the state the program is in before this invocation
- ι the input to the program in this invocation
- o the output of the program in this invocation
- χ the information about this invocation or output where χ is one of
  - InvalidState - this state should never occur ($\iota = \varnothing, o = \varnothing$)
  - InvalidInput - this input is not allowed in this state ($o = \varnothing$)
  - InvalidOutput - this output is not allowed in this state for this input
  - DontCare - it is not important how the program behaves in this state and for this input (o can be $\varnothing$)
  - Valid - this is the valid behaviour for the program for this invocation

Note that in practice requirements are often not written in this atomic form. Require-

ments in traditional specifications often specify a large number of invocations at once. An example would be a mathematical expression relating the outputs to the inputs for some sub-domain of the valid input values. However, for our purposes we simply consider that a notational convenience.

We note how closely related behavioural requirements are to tests. A typical test has one part that sets it up and another part describing the inputs to the program[1]. When a test runs the program gives some output behaviour. A test can thus be described with the tuple $\{\sigma, \iota\}$ and a test run with the tuple $\{\sigma, \iota, o\}$. Thus, a test can be seen as a half-baked requirement; it simply lacks a statement about the validity of the invocation and the program behaviour.

A specification is simply a list of individual requirements.

Definition 2 (Behavioural Specification): A *behavioural specification*, denoted β, is a finite list of behavioural requirements: $\beta = [\rho_1, \rho_2, \ldots, \rho_n]$.

We also define the space of all possible specifications.

Definition 3 (Specification Space): The *specification space*, denoted $\hat{B}$, is the set of all possible specifications of the dynamic behaviour of computer programs.

Now for a certain development problem there will typically be many specifications in $\hat{B}$ that specify the wanted behaviour in a way that is good enough for the problem at hand. This is because there are situations in which we do not care how the system behaves. Given a development problem we can imagine a set of acceptable specifications that all describe the behaviour of the sought for program in a way that would be acceptable for the patron.

Definition 4 (Acceptable Specifications): The *acceptable specifications*, denoted B, is the set of all possible specifications that describe a behaviour of a program that would solve the problem at hand in an acceptable way.

We use the term *acceptable* since there will always be a trade-off for the resources we have at hand. A basic goal with a development process must be to get enough information about the set of ideal specifications that are acceptable so that we can write a concrete specification that is close enough to the ideal for our purpose and the resources we have at hand. Formally we write:

Development goal 1: Get to know $B \subset \hat{B}$ enough so that you can write a concrete specification $\bar{B} \in B$.

---

[1]Throughout this paper we use input to refer to the wider notion of stimuli, ie. including both the data and the functions / methods to be called

The ultimate source of information on B must be the patron who has given us the task of developing the program.

2.2. Patrons

Patrons are the owners of the problems that motivate the development process. At the highest level they are some outside agent (customer, boss, contractor etc) which needs a program to solve their problem. However, development processes often have multiple levels and on lower levels the patron might be the developer herself. As an example this would be the case if the developer has identified a sub-component that is needed to solve the customers problem. Even though the ultimate adjudicator of acceptable behaviour is the customer, he might not know how the sub-component must behave. Thus on this lower level of development the developer might be her own patron.

One way to learn about the acceptable specifications is by engaging in Q&A-sessions with the patron. The type of questions that can be answered by the patron range from basic to very complex. The most basic patron can only distinguish invalid from valid invocations and valid from invalid program behaviour. Formally

> **Definition 5 (Least Knowledgeable Patron):** The *least knowledgeable patron*, denoted LKP, is the patron that can only answer questions of the form 'Given a program invocation $\{\sigma, \iota, o\}$ what is χ?'.

At the other end of the scale we could think of an patron that would give a full acceptable specification if we simply ask her.

> **Definition 6 (Fully Knowledgeable Patron):** The *fully knowledgeable patron*, denoted FKP, can give a full acceptable specification, with a complete and consistent list of behavioural requirements, if we simply ask.

The LKP requires a process of interactive specification, where the developer generates questions based on its current knowledge and then updates its knowledge based on the patrons answers. The FKP does not need any questions at all; she can give a complete behavioural specification without any knowledge from the developer.

In practice most patrons are typically between these two endpoints and can give different types of answers depending on the particular invocation. Often they can only reply to direct questions but sometimes give answers for a whole class of invocations. One common patron can give expected outputs when given a state and input.

> **Definition 7 (Expected Output Patron):** The *expected output patron*, denoted EOP, answers with χ and o when asked about the program behaviour for $\{\sigma, \iota\}$.

Patron interaction is even more problematic in practice. The patron might not have the same framework as the developers so might not understand or know the answer to a question. The developer might have a different picture of the problem than the patron leading to misunderstanding and confusion. In this theory we do not model this although we note it as important and a topic for further study.

2.3. Programs

The final product of any software development process is a program. It consists of source code in one or several programming languages that can be compiled to a form that a computer can execute.

In a theory we do not want to care about the details of the programming language or about the form that is needed of the final program so that a computer can understand it. We simply note that a program must be one particular instance in the space of all possible programs. For a particular development problem there is also a set of programs that all show acceptable behaviour.

Definition 8 (Program Space): The *program space*, denoted $\hat{\Pi}$, is the set of all possible computer programs.

Definition 9 (Program): A *program*, denoted π, is a finite list of program instructions that can be executed by a computer.

Definition 10 (Acceptable Programs): The *acceptable programs*, denoted Π, is the set of all possible programs that have a behaviour that would solve the problem at hand in an acceptable way.

In the same way as for specifications we have the obvious goal:

Development goal 2: Get to know $\Pi \subset \hat{\Pi}$ enough so that you can write a concrete program $\overline{\Pi} \in \Pi$.

2.4. The Developer

No theory for a development process can be complete without modeling the developer. We need ways to represent the knowledge she has gained about different parts of the problem and to represent any other knowledge she has that can be put to use during development.

A basic observation is that the developer cannot directly access either B, the acceptable specifications, or Π, the programs showing acceptable behaviour. At any

step during the development process she has some knowledge about these sets. This knowledge typically differs in many ways from the ideals she is trying to achieve. Below we formalize this.

> **Definition 11 (Developer's Specification Knowledge at step t):** The *developer's specification knowledge*, denoted $\tilde{B}_i$, is any information about the acceptable specifications, B, that a developer has at development step *i*.

For human developers $\tilde{B}_i$ is the mental picture or idea that the developer has about the acceptable specifications at a certain development step.

Similarly we have for the program space:

> **Definition 12 (Developer's Program Knowledge at step t):** The *developer's program knowledge*, denoted $\tilde{\Pi}_i$, is any information about the acceptable programs, Π, that a developer has at development step *i*.

Informally a step is an atomic action that drives the development forward. To analyze what different steps are possible we also need to define different parts of developer experience.

> **Definition 13 (Developer's Experience):** The *developer's experience*, denoted $\tilde{K}_{dev}$, is any information that the developer has and can brought to bear to help solve the problem at hand.

When such experience relates to a particular type of development artefact we note the artefact with a superscript. $\tilde{K}^{spec}_{dev}$ is any prior knowledge the developer has on how to write specifications, while $\tilde{K}^{prog}_{dev}$ is the knowledge pertaining to writing programs.

2.5. The Library

The library represents the total knowledge that humanity and its machines have accumulated that are relevant to software development processes and that is explicitly available. We must require that the knowledge has been written down in some form since it is of no use if it sits within the head of another developer without any way to access it.

> **Definition 14 (Humanity's Total Experience):** The *humanity's total experience*, denoted K_{total}, is any information that humanity has previously acquired and that can be brought to bear to help solve the problem at hand.

In the same way as with developer knowledge we note the artefact that the experience relates to with a superscript. $\tilde{K}^{spec}_{total}$ for example is any developers prior knowledge

on how to define and write specifications, while $\tilde{K}^{prog}_{total}$ is the knowledge pertaining to programs. $\overline{K}_{total}$ would be knowledge written down in a form that other developers can understand.

In fact K_{total} is primarily books and snippets of code for testing and programs that we can look up and use since it is hard to tap into the parts of if that is in the heads of other developers. But there is nothing stopping that the library also has a number of experienced developers that one can ask questions pertaining to the development problem at hand. However, we think formalizing the knowledge in $\tilde{K}_{total}$ so that it can be put to direct use in development projects is an important goal for the software engineering community as a whole.

With the basic elements in place we can now relate them in a model.

3. An Aggregate Model for Software Development

The basic observation on which our theory is built is that a software development process is a learning process in which the developer needs to refine her internal models[1] of the acceptable specifications and the programs showing such behaviour to the point where she can clearly formulate this knowledge in executable form. It is not simply a matter of formalizing existing knowledge. It is about first acquiring that knowledge and then formalizing it.

For both specifications and programs there are three main entities to keep in mind: the ideal set, the internal model of the ideal, and the concrete artefact. In the space of all possible entities $\hat{M}$ there is a sought-for set of ideal enitites M. Through its internal models of M denoted $\tilde{M}$ the developer wants to write a concrete entity $\overline{M}$ that as the development process goes on gets closer and closer to the ideals.

From this abstract picture we can see that there are two basic ways to make progress. The developer can either make $\overline{M}$ better reflect $\tilde{M}$ or she can make $\tilde{M}$ better reflect M. In some sense, the latter is harder and more fundamental. If there is a discrepancy between $\tilde{M}$ and M the goal we are striving for when writing $\overline{M}$ is wrong. We cannot identify such a situation without extending our view of the goal, ie. refining $\tilde{M}$. This needs an outside force that cannot be found when operating[2] inside $\tilde{M}$. Finding discrepancies between $\overline{M}$ and $\tilde{M}$ is easier since it is a matter of reading $\overline{M}$ and comparing it to the internal model.

Lets now incorporate the other elements into this picture. The patron is the adjudicator of what is and is not inside B. To clarify this division we ask her questions about

[1]For human developers we could say mental models, pictures or ideas

[2]For human developers we could say thinking

(sets of) invocations and she classifies the behaviour. Based on such Q&A-sessions with the patron the developer refines $\tilde{B}$. Based on this model she can refine $\overline{B}$ and her model of what acceptable programs must look like, $\tilde{\Pi}$. This model is used to write a concrete program, $\overline{\Pi}$.

The internal models of acceptable specifications and programs depend on the developers previous knowledge. Some of this knowledge may be explicit in the sense that the developer can write it down in a formal way ($\overline{K}_{dev}$). Some of it is tacit knowledge, distilled through previous experience ($\tilde{K}_{dev}$).

The developer can also tap on some of the total knowledge available in the library. By default this knowledge is formal in the sense that it has been written down ($\overline{K}_{total}$). However, the developer might also be able to tap into tacit portions of K_{total} by, for example, asking fellow developers.

From the patrons point of view each development project should include the best practices in K_{total} that are relevant to this problem. An over-arching goal for the software development community as a whole should be to support that.

The developers previous knowledge is a bag of models that affects his thinking. He can widen this knowledge by going to the library and seeking information that is relevant to the problem at hand. By working with the concrete specification and program, and by questioning the patron she can refine her internal models of the ideal specifications and programs. This in turn helps refine the actual artefacts and another round starts.

With this model we can define what we mean by a software development process.

> **Definition 15 (Software Development Process):** A *software development process* is a sequence of steps in which a developer refines $\tilde{B}$ and $\tilde{\Pi}$ to reflect B and Π faithfully enough so that she can write down $\overline{B} \in B$ and $\overline{\Pi} \in \Pi$.

From this definition we see that a development process is a process of incremental learning. No matter what previous experience the developer has he must learn what parts and how to apply his knowledge in this particular case.

To see what propels this process of learning forward we need to see what can happen in each development step.

3.1. Development steps

There are two main ways in which a development process can make progress: refining an internal model or refining an artefact based on an internal model.

Both these refinement processes have two main components: identifying a

discrepancy and overcoming it. We call the event that makes the developer identify a discrepancy a *trigger*. When the discrepancy has been identified and described the developer decides to take some action to overcome the discrepancy. We call the latter the *remedy* and use the three-part model, of trigger, discrepancy and remedy to describe important development steps.

Triggers are numerous[1]. A common trigger is to do a review of an artefact. By reading the code the developer confronts it with his internal model and can note omissions, inconsistencies and codification faults. Reviewing the specification has similar benefits but with the additional possibility of including the patron during the review. If that is practically possible it is often very valuable since the developers model will confront the patron's view of the system.

Another source for triggers is to use check lists. A check list codifies knowledge that have been important in the past. Each developer has their own internal check lists with common gripes and problematic points. However, writing them down has the benefit that the developer will not forget important findings in later projects. It also has the benefit that check lists can more easily be communicated to others. As an example a developer could get check lists from the 'library' and use any additional items not on their own list when doing a review.

A third common trigger is to write and conduct tests of the system. Since tests concern themselves with the dynamic behaviour of the system they are a natural link between the program and the behavioural specification. When the developer writes a new test she considers how the program behaves in a new situation. Even though she may not know the expected behaviour, the test is valuable since it can be run and the program's output considered. Considering whether a certain output is valid might be simpler than writing down the expected output since the latter requires creativity while the latter is a matter of classification. And if the developer realizes she cannot classify the behaviour of the program she can note that she should ask the patron to help her.

4. Related work

Exploratory programming (EP) focus on developing a prototype to explore the problem domain and to discover good solution strategies [6]. The prototype is then refined until it performs in an adequeate way. The approach is suitable for projects where a detailed requirements specification cannot be written or for research where it is not clear if a solution even exists [6]. The model has some similarities to our model by not forcing developers into writing a specification up-front. However, our model

[1]We do not consider the trivial trigger of a TODO list listing parts of the program or specification that we explicitly know of but have not yet written down.

highlights the importance of the specification and promotes that properties that the program must have are written down as they are found.

Extreme programming (XP) was recently proposed as a lightweight software development process and it has attracted much interest [1]. It is a package of several practices and ideas that in isolation are not new but taken together form a different approach to software engineering [3]. XP focuses on the importance of writing automatically executable tests with tools that support unit testing. In for example JUnit, a tool supporting the writing of XP-style unit tests for and in Java, you specify how to setup and execute the test but also the expected results [2]. Tests written in such a way would be behavioural requirements in our model and considered part of the specification.

An important element of XP is the use of pair programming in which two programmers together program at the same terminal. One takes the driving role and writes on the keyboard while the other 'looks over the shoulder' and thinks about strategic issues. Developers frequently change roles. Both anecdotal evidence and experimental research have shown that pair programing leads to higher productivity and higher programmer satisfaction [8]. There are possibly many reasons for this effectiveness but we note that one possible explanation is that the models of the developers are constantly rubbed against each other so that they are questioned and discrepancies identified. Our model supports this explanation by highlighting the importance of having some external force that triggers the questioning and refinement of internal models.

In his PhD thesis, Andrew Walenstein presents a framework for understanding software engineering tools in terms of how they support the cognitive processes of the developer [7]. His core ideas is that cognition can be usefully modeled as computation and that the cognitive support offered by the tool is the computational advantage it provides. The tool together with the developer is seen as a distributed cognitive system. The stance taken by Walenstein is similar to ours, in that the focus is on the developer and supporting his cognitive processes. However, Walenstein does not place the same emphasis on the possibility of tools to be creative and help the developer 'think outside the box'. In particluar his qualitative theory of cognitive support lists four prinicples of computational advantage: task reduction, algorithmic optimization, distribution and specialization. Even though several of them applies in some sense to the creative abilities of a tool (the tool creates a test invocation so reduces the number of invocations that a developer needs to create) none of them captures it in its entirety.

The capturing and codification of knowledge on software development so that it can be used automatically has been adressed in the past. An example is the CREATOR2 system for automatic software design built by Koono et al [4] which uses an expert system with knowledge about different deisgns to help in designing switching software. At the core the system is based on a unified representation scheme for modeling the design process and the design product and using multiple strategies in applying the knowledge. Even though the authors does not explicitly mention it the knowledge

in the system could be exchanged between different sites and thus be a candidate to put in a library.

5. Discussion

5.1. On assumptions

An assumption in our model is that the ideal artefacts are static in the sense that for a certain development project there exists a set of ideal solutions. Over time the environment where the system is used might change as might the user requirements. This will lead to a possible change in the ideal artefacts. This would complicate the task of the developer since he is now targeting a moving target. We do not model this but note it as an important topic for refined models.

5.2. On why tests are not elements of the model

Tests are not as fundamental as specifications and programs. It is thinkable that development processes exist where there is no need for tests. However, in practice any development problem of interesting complexity will have a non-empty set of tests. The reason is that while a program is a static artefact, a specification states its dynamic behaviour. Tests are needed to bridge the static domain of programs to the dynamic domain of specifications.

We have chosen not to include the tests since there are other possible verification and validation activities and it is not clear on what grounds some should be included but not others. For example, inspections is a fundamental way to identify discrepancies between an artefact and our model.

However, tests have a special place by being so closely related to specifications. Tests are incomplete atomic behavioural requirements. They are invocations that lack values for the output and classification of the behaviour shown by the invocation. A test can be written $\{\sigma, \iota, ?, ?\}$ and a test run as $\{\sigma, \iota, o, ?\}$ and by filling in the missing information we get a full requirement.

5.3. On why the specification needs to be written down

There could be situations where it is not needed to write down $\overline{B}$ but the patron would take a risk. If the developer is no longer tied to the patron the knowledge about B is lost. Also opportunities for automation and tool support is lost if it is not written down.

5.4. On implications for tool support

Writing down a $\overline{B}$ that covers all of $\tilde{B}$ can be hard since not all parts of $\tilde{B}$ is conscious. There are parts of our models that are implicit and assumptions we make without thinking about them. However, when shown a particular invocation of a program few developers have a problem classifying it as valid or invalid. If they cannot say if it is valid or invalid it has pointed to a lack of knowledge. An opportunity for a tool to support the incremental learning would be to search for tests that show the current behaviour of the program in novel situations.

5.5. On why there is only a single developer?

We talk about our developer in singularis although software is often developed in teams. Even if the learning process would be more complicated for a team, we see no reason that the principles involved would be different. It would only mean that our developer could be more competent (even though that is not self-evident and might depend on the task and individual developers). Multiple developers have a larger pool of knowledge and experience to draw from and can thus, as a group, make more informed decisions. There is also the possibility of parallelizing tasks which could lead to productivity boosts. However, this assumes that the increased communication burden does not lead to misunderstandings and problems.

5.6. On the connections to Software Engineering

A common factor in definitions of Software Engineering is that they are concerned with teams of developers and includes issues both of technical and non-technical nature. For example it is important to have good documentation. Our model does not deal with documentation and neither with maintenance, user interfaces or any of the many other sub-fields of Software Engineering. We simply note that a fundamental problem in any software development project is to develop a program that shows acceptable behaviour. In relation to this problem many other issues studied in Software Engineering are, although important, secondary.

5.7. On whether knowledge can be separated into different parts

When modeling the user we separate out different parts of his experience such as $\tilde{K}_{dev}^{spec}$ and $\tilde{K}_{dev}^{prog}$. Is that not a gross simplification with many hidden assumptions about the form and structure of human knowledge?

In a sense yes. But we are not claiming that it is plausible to assume that human knowledge can be separated into independent units of information pertaining to

different aspects of the world. We simply identify that different parts of human knowledge are relevant to different questions. By collecting that together and giving it a name we can then reason with it. So essentially the actual form and structure of human knowledge is not an issue for the theory. This is similar to the argument in [5] about the ability to delineate a model's boundaries.

6. Conclusions

A theory for software development was presented built from the five elements fundamental to any development process: a *patron* which has a need for a program or program component, a *specification* which states how the program should and should not behave, a program, a *developer* writing it and a *library* containing all of humanities total knowledge relevant to the problem at hand. Based on these elements a software development process is defined as an incremental learning process in which there are two main ways to make progress: refining an internal model of the developer or refining an artefact based on an internal model. Since the former underlies the latter it was identified as more fundamental.

The theory has implications for tools supporting software development. They should trigger the identification of discrepancies between the internal models of the developer and the ideal artefacts that would lead to acceptable behaviour. One way for them to do that would be to create novel test invocations for the developer to consider. If the tool is creative in creating these invocations it could help the developer 'think outside the box' and realize his knowledge is incomplete.

References

[1] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[2] JUnit development team. JUnit - A Cook's Tour. Tech. Rep. (2002). URL *http://junit.sourceforge.net/doc/cookstour/cookstour.htm*.

[3] Daniel Karlström. Introducing Extreme Programming - An Experience Report. In *Extreme Programming Conference 2002 (XP2002)*, 2002.

[4] Z. Koono, B. H. Far, T. Takizawa, M. Ohmori, K. Hatae and T. Baba. Software Creation: Implementation and Application of Design Process Knowledge in Automatic Software Design. In *The 5th International Conference on Software Engineering and Knowledge Engineering, San Fransisco Bay, USA*, 1993.

[5] Marvin Minsky. Matter, Mind and Models. Tech. Rep. (1997), MIT's Artificial

Intelligence Laboratory. URL *http://web.media.mit.edu/~minsky/papers/MatterMindModels.html*.

[6] Ian Sommerville. *Software Engineering*. Addison-Wesley, 1994.

[7] Andrew Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. Ph.D. thesis, Simon Fraser University.

[8] Laurie Williams and Robert Kessler. All I Really Need to Know about Pair Programming I Learned in Kindergarten. *to appear in Communications of the ACM*.