Program Comprehension Risks and Opportunities in Extreme Programming

A. van Deursen

Software Engineering (SEN)

SEN-R0110 May 31, 2001

Report SEN-R0110
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Program Comprehension Risks and Opportunities in Extreme Programming

Arie van Deursen

CWI, P.O. Box 94079
1090 GB Amsterdam, The Netherlands
`http://www.cwi.nl/~arie/`

ABSTRACT

We investigate the relationship between reverse engineering and program comprehension on the one hand, and software process on the other. To understand this relationship, we select one particular existing software process, extreme programming (XP), and study the role played in it by program comprehension and reverse engineering. To that end, we analyze five key XP practices in depth: pair programming, unit testing, refactoring, evolutionary design, and collaborative planning.

The contributions of this paper are (1) the identification of promising research areas in the field of program comprehension; (2) the identification of new application perspectives for reverse engineering technology; (3) a critical analysis of XP resulting in research questions that could help resolve some of the uncertainties surrounding XP; (4) a process assessment framework for analyzing software processes from the comprehension and reverse engineering point of view.

1998 ACM Computing Classification System: D.2.9, D.2.2, D.2.5, D.2.7

Keywords and Phrases: Software process, reverse engineering, pair programming, unit testing, refactoring, evolutionary design, collaborative work.

Note: Work carried out under projects SEN 1.1, *Software Renovation* and SEN 1.3, *Domain-Specific Languages*.

1. Introduction

Every software development team will at some point be confronted with the question how to organize the development process throughout a system's life cycle. A major cost factor in the life cycle of a software system is *program understanding* [11]: trying to understand an existing software system for the purpose of planning, designing, implementing, and testing changes. This suggests that paying attention to program comprehension issues in software process could well pay off in terms of higher quality, longer life time, fewer defects, lower costs, and higher job satisfaction.

Software process in relation to program understanding and reverse engineering technologies raises the following questions:

- How should we organize our process if we want to strengthen the team's program comprehension capabilities?
- How should we organize our process if we want to optimize the code's understandability?
- What (new) directions in software process can we identify in order to leverage advances in program understanding and reverse engineering research?
- What (new) directions in comprehension or reverse engineering research can we identify in order to strengthen their usefulness for industrial software processes?

These are long term research questions, without simple answers. In order to contribute to answering these questions, we will direct our attention to one particular software process: *extreme programming* (XP) [2]. XP is characterized by short cycles, incremental planning, evolutionary design, and its ability to response to changing business needs. It strongly relies on oral communication, and uses tests and source code to communicate system structure and intent. From the comprehension perspective, XP is a process worth studying for at least two reasons:

- First, *source code* plays a dominant role in almost every XP step: Code is documented via test code rather than via prose, the tests themselves are code rather than input data, code is improved continuously to keep the overall design simple, and such code refactorings even replace an explicit design phase.
- Second, team *communication* is an explicit XP value. Software coding is always done in (rotating) pairs, the two people working together and discussing the way they are implementing a given feature. Moreover, planning is done by the complete development team discussing the impact of each of the feature requests.

The emphasis XP puts on people and source code suggests that the need of people to understand pieces of code is at the very heart of XP, making it an interesting case study in analyzing the role comprehension can play in software process.

In this paper, we will carefully analyze the role of program understanding in XP. To that end, we have selected five key practices of XP: pair programming, unit testing, refactoring, simple design, and planning as a team activity. For each of these, we analyze the comprehension risks and opportunities, looking at the effect on the team (whether and how the team gets a better understanding of the code) as well as on the source code (whether and how the code gets more understandable). To conduct this analysis, we look at (1) the published literature on XP [2, 23, 3]; (2) on line discussion fora covering XP subjects;[1] and (3) our own experiences made during an ongoing extreme programming project.[2]

We consider the following the most important contributions of this paper. First, from our analysis of XP we are able to identify a series of promising new research areas in the field of program comprehension. Second, we identify several new application perspectives for reverse engineering technology. Third, our critical analysis of XP enables us to formulate research questions that could help resolve some of the uncertainties surrounding XP. And finally, we return to the general issue of software process, and propose a framework for assessing software processes from the comprehension and reverse engineering points of view.

1.1. Releated Work

Wong analyzes the impact reverse engineering technology can have on software process [36]. He proposes the notion of *continuous program understanding*, in which reverse engineering tools are used continuously to find abstractions and move them forward during software evolution.

Software maintenance is an an area in which it is widely accepted that both program comprehension and an adequate software process are key success factors. An often quoted figure is that 40–60% of the time spent on software maintenance is used for program understanding [32, 11]. In his text book on software maintenance, Pigoski describes a recommended process following a modification request through stages such as analysis, design, coding, testing, and installation [32]. Pigoski's book does not deal with the questions how the recommended steps are related to program comprehension, and what can be done to optimize for program comprehension throughout the software maintenance process.

Several articles describe processes that can be followed during reverse or re-engineering projects [5, 34]. Such processes are an important tool for carrying out these projects in a managed and predictable way. They do not, however, address the issue how to organize the software development process in order to set conditions favorable for program comprehension.

Naturally, many software development processes include elements that are essential for program understanding, and only some of these are used in XP as well. For example, Humphrey stresses the importance of inspections, software quality assurance, and testing [21]. The Rational Unified Process emphasizes short iterations, architecture centric software development, and use cases [26]. An explicit analysis of software processes in view of their support for program comprehension does not appear to be available.

Key publications on extreme programming [2, 23, 3] cover many issues related to comprehension, such as code expressing intent, feedback from the system, and tests to document code. Several publications on XP deal with topics that are close to program comprehension, such as pair program and refactoring [35, 17]. However, an explicit account of what program comprehension means to XP (and vice versa) is not available: a gap we are trying to bridge with the current paper.

2. Background

2.1. Program Understanding

We define[3] program understanding (comprehension) as *the task of building mental models of an underlying software system at various abstraction levels, ranging from models of the code itself to ones of the underlying application domain, for software maintenance, evolution, and re-engineering purposes* [29]. Studies indicate that more than half of the programmer's task during maintenance is involved in program understanding [11].

In this paper, we will be particularly involved with program understanding during *software evolution*, which Benett and Rajlich define as a particular phase in software maintenance [6]. They distinguish between initial development, repeated software evolution, servicing, phase out, and close down. Initial development in extreme programming is very short (a couple of weeks only), after which an XP project enters the software evolution phase.

An important research area in program understanding deals with the cognitive processes involved in constructing the mental model of a software system (see, e.g., [27]). A common element of such cognitive models is generating hypotheses about code and investigating whether they hold or must be rejected. Several *strategies* can be used to arrive at relevant hypothesis, such as bottom up (starting at the code), top down (starting from a high-level goal and expectations), and opportunistic combinations between the two [11]. Strategies

[1]Most notably, the *C2 wiki* at `www.c2.com/cgi/wiki` and `groups.yahoo.com/group/extremeprogramming/`.

[2]The project involves the construction of commercial program understanding tools by the *Software Improvement Group*: `www.software-improvers.com/`.

[3]The definitions in this section are also discussed on line at the *wiki* at `www.program-transformation.org/re/`.

Pair programming	Production code is developed by pairs of programmers
Testing	Unit tests and acceptance tests are run continuously
Refactoring	Continuously improve the design without changing the functionality
Simple Design	The guiding design principle is to do the simplest thing that could possibly work
Planning Game	Development estimates user stories, and business prioritizes them
Collective ownership	Developers can modify any piece of code
Continuous integration	Integrate changes immediately instead of developing them in separate branches
40-hour week	Programmers work 40 hours max., to keep them fresh and creative
On site customer	A customer is on th team to discuss feature requests and domain concepts
Frequent releases	Release code into production as often as possible
System metaphor	Simple shared story of how the system works
Coding standards	Ensure agreement on simple coding conventions

Figure 1. The twelve key practices of Extreme Programming

guide two understanding mechanisms that produce information: *chunking* creates new, higher level abstraction structures from chunks of lower level structures, and *cross referencing* relates different abstraction levels [27]. We will see how the XP practices relate to these program understanding theories.

The construction of mental models at different levels of abstraction can be supported by reverse engineering tools in order to (1) identify a system's components and interrelationships; and (2) create representations of a system in other forms or higher levels of abstraction [9]. We will explore how reverse engineering tools fit in naturally in the XP process.

An emerging research area in reverse engineering and program understanding is *software process* [30]. A software process can be defined as *the coherent set of policies, organizational structures, technologies, procedures and artifacts that are needed to conceive, develop, deploy, and maintain a software product* [20]. Software process involves technical, methodological, organizational and economic issues [20].

2.2. An XP Primer

Extreme Programming (XP) is a light-weight methodology for teams of approximately 10 people developing software in the face of vague or rapidly changing requirements [2]. XP builds upon various existing and common sense practices and principles, but applies these to extreme levels. For example, code review, testing, designing, and refactoring are done *continuously*, rather than at dedicated phases of the software process only. XP assumes that the development team makes use of modern development environments (Smalltalk, Java), and aims at taking maximal advantage of the resulting benefits.

XP is performed in short *iterations*, which are grouped into larger *releases*. The planning process is depicted as a game in which business and development determine the scope of releases and iterations. The customer describes features via *user stories*, informal use cases that fit on an index card. The developers estimate each of the user stories. User stories are the starting point for the planning, design, implementation, and acceptance test activities conducted in XP.

The 12 key practices of XP are listed in Figure 1. We will study the first five in considerable detail in the next sections.

3. Pair Programming

Pair programming means that *all production code is written with two people looking at one machine, with one keyboard and one mouse* [2]. It is the pair's job is to write the tests and production code for a given user story. To that end, they discuss design issues, as well as the particular piece of code they are looking at. While the developer holding the keyboard is entering code, his partner is conducting an immediate code review, thinking about the overall design, additional test cases, and potential simplifications.

3.1. Comprehension Benefits

One way in which pair programming affects comprehension is that *pair programmers explore a larger number of alternatives than a single programmer alone might do.* [16]. The partners may have different backgrounds and experiences, and different strategies for solving particular sets of problems. Their combined strengths help them when understanding existing code. Moreover, when designing for change they may come up with ideas that neither of them would have found when working individually. Thus, pair programming might be described as a form of *distributed cognition*, in that multiple parties participate in the development of a final theory [33].

A consequence of this is that pair programming should result in better code. Indeed, several studies into the effects of pair programming have been conducted [16, 35, 10], which report that pair programming results in better code, fewer defects, better design, better team building [35, 10]. Moreover, code written by two programmers together is more readable than code written by one programmer only [31].

Concerning team building, XP requires that programmers frequently change partner. As a result, *system knowledge is shared between the team* rather than isolated with some key programmers. Moreover, *when modifying existing code it will rarely happen that both partners are new to the code to be changed.* When deciding with whom to pair when starting to work on a particular task, a programmer will try to find a partner that is more or less familiar with the code affected, for the simple reason that it will save time. Thus, an XP programmer is likely to have someone else's code explained to him, rather than having to understand it all alone.

Another consequence of pair rotation is that *pair programmers can monitor and learn from each other's program comprehension strategies.* Pairing forces a programmer to think aloud, making the comprehension strategy followed explicit to the programming partner. This not only helps him to understand a given piece of code, but also to see how others attack a program that is hard to grasp. At the same time, the mere act of thinking aloud may help the programmer to understand the source code more quickly.

An XP practice related to pair programming is that all programmers share one room. Thus, *pair programming makes it possible to overhear other programmers code comprehension discussions and join them when needed.* In our project, it happened regularly that a programmer jumped into discussions concerning code he checked in a while ago, and started explaining what he had done. In some cases this resulted in spontaneous design discussions concerning more fundamental aspects of the system than just the given code.

3.2. Comprehension Risks

XP assumes that pair programming replaces explicit code reviews. To quote Beck, "if code reviews are good, we'll review code all the time" [2, p. xv]. It is, however, questionable whether the several reported benefits of code review are still applicable if it is done "all the time". Such reviews are short, intense, organized sessions, specifically focused on finding and removing errors [19], which is quite different from pair programming. Therefore, XP's decision to refrain from explicit code reviews is a potential risk: it may, for example, be the case that pair programming with separate explicit review sessions yield better results, or can be used to find different types of problems [22].

Another concern related to pair programming might be the costs, which at first may seem to be twice as high as with single programming. Studies have shown that this is not the case [35, 10]. In addition to quality considerations mentioned above, a pair is more productive than a single programmer. As result, the total increase of development time is 15%, rather than 100% – not counting time saved due to the increased quality. In our project, we felt that one reason for the productivity gain is that the partner "feeling best" is doing the typing and determining the speed, reducing the loss of productivity due to, e.g., tiredness of one of the programmers.

4. Unit Testing

Unit testing is at the heart of XP. Unit tests are written by the developers, using the same programming language as used to build the system itself. Tests are small, take a white box view on the code, and include a check on the correctness of the results obtained, comparing actual results with expected ones. Tests are an explicit part of the code, they are put under revision control, and all tests are shared by the development team (any one can invoke any test). A unit test is required to run in almost zero time. This makes it possible (and recommended) to run all tests before and after any change, however minor the change may be.

Testing is typically done using a testing framework such as *junit* developed by Gamma and Beck [4]. The framework caters for invoking all test methods of a test class automatically, and for collecting test cases into test suites. Test results can be checked by invoking any of the *assert* methods of the framework with which expected values can be compared to actual values. Testing success is visualized through a graphical user interface showing a growing green bar as the tests progress: as soon as a test fails it becomes red.

The XP process encourages writing a test class for every class in the system. The test code / production code ratio may vary from project to project but can be as high as 1:1. XP prescribes to test *everything that could possibly break* [23]. Moreover, XP encourages programmers to use tests for documentation purposes, in particular if an interface or method is unclear, if the implementation of a method is complicated, if there are circumstances in which the code should work in a special way, and if a bug report is received [2]. In each of these situations, the test is written *before* the corresponding method is written (or modified).

Moreover, test can be added while understanding existing code. In particular, whenever a programmer is tempted to type something into a print statement or debugger instruction, XP's specifically advises to write a test instead and add it to the system's test suite [4].

4.1. Comprehension Benefits

First, *XP's testing policy encourages programmers to explain their code using test cases.* Rather than explaining the behavior of a function using prose in comments or documentation, the extreme programmer adds a test describing that behavior.

Second, *the requirement that all tests must run 100% at all times, ensures that the documentation via unit tests is kept up-to-date.* With regular technical documentation and comments, nothing is more difficult than keeping them consistent with the source code. In XP, all tests must pass before and after every change, ensuring that what the developer writing the tests intended to communicate remains valid.

Third, *adding unit tests provides a repeatable program comprehension strategy*. If a programmer needs to change a piece of code with which he is not familiar, he will try to understand the code by inspecting the test cases. If these do not provide enough understanding, the programmer will try to understand the nature of the code by developing and testing a series of hypotheses, as we have seen in Section 2.1. The advise to write tests instead of using print statements or debugger commands applies here as well: *hypotheses can be translated into unit tests*, which then can be run in order to confirm or refute the hypotheses. Also observe that such test cases are added to the system's test suite. Thus, *the test suite acts as a log of the hypotheses used to understand a particular piece of code*, making them available to other team members as well.

Fourth, *a comprehensive set of unit tests reduces the comprehension space when modifying source code*. To a certain extent a programmer can just try a change and see whether the tests still run. This reduces the risks and complexity of conducting a painstakingly difficult impact analysis. Thus, the XP process attempts to minimize the *need* for building up a mental model of certain program parts since the test help the programmer to see that these parts are not affected by the current code modifications.

The XP testing process not only affects the way the team works, it also has a direct effect on the understandability of the production code written [23, p.199]. *Writing unit tests requires that the code tested is split into many small methods each responsible for a clear and testable task.*

If the tests are written after the production code, it is likely that the production code is difficult to test. For that reason, XP requires that the unit tests are written *before* the code (the "test first" approach). In this way, *testing code and production code are written hand-in-hand, ensuring that the production code is set up in a testable manner*.

4.2. Comprehension Risks

Using tests for documentation leads to the somewhat paradoxical situation that in order to understand a given piece of code a programmer has to read another piece of code. Thus, to support program comprehension, XP increases the code base by up to 100%, code that needs to be maintained as well. In our project, we experienced that maintaining such test code requires special skills and refactorings, some of which we have described in [15].

Another concern is that XP uses the tests (in combination with oral communication and code written to display intent) as a *replacement* for technical documentation. The word "documentation" is mentioned once in Beck's, where he explains why he decided *not* to write documentation [2, p. 156]. For addressing subjects not easily expressed in the tests or code of the system under development, a *technical memorandum* can be written [12], These are short (one or two pages) papers expressing key ideas and motivations of the design. However, if the general tendency is not to write documentation, it is unlikely that the technical memoranda actually get written, leaving important decisions undocumented.

A final concern is that some types of code are inherently hard to test, the best known examples being user interface and database code. In our project, we had trouble building tests for the code listening to an input queue interfacing with the mainframe. Writing tests for such code requires skill, experience, and determination. This will not be always available, leaving the hardest code without tests and thus without documentation.

5. Refactoring

A *refactoring* is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior [17, p.53]. Fowler has collected a catalog of refactorings, which range from very simple modifications to more substantial changes comprising several different smaller refactorings. Many of the larger refactorings deal with the introduction of design patterns into the software, such as *Replace Type code with State/Strategy*, which replaces enumerated constants by a dynamic state.

Refactorings are applied when *bad smells* are detected in the code, such as *duplicated code* (which violates the *once and only once* rule), *long method*, *feature envy* (when a method seems more interested in the features of a class other than the one the method is actually in) or *switch statements* [17].

To apply a refactoring, unit tests for the code affected are needed in order to ensure that the functionality of the code is not changed. Such tests are run before and after each refactoring. The refactorings themselves are applied iteratively and in small steps, running tests in between. When writing the code for a user story, the developers search for potential refactorings both before they start and after they are finished.

5.1. Comprehension Benefits

The primary goal of refactoring is to make the code more understandable. Explicitly including refactoring in the process *gives programmers the right to improve their code*, even under time pressure.

Next, *a catalog of smells and refactorings provides programmers with a vocabulary to discuss their program understanding*: they can recognize bad programming structures and have ways to improve these.

Most interestingly, refactoring can also act as a way of helping developers to understand the programs they are modifying. To quote Fowler [17, p.56],

> I use refactoring to help me understand unfamiliar code. When I look at unfamiliar code, I have to try to understand what it does. I look at a couple of lines

and say to myself, oh yes, that's what this bit of code is doing. With refactoring, I don't stop at the mental note. I actually change the code to better reflect my understanding, and then I test that understanding by rerunning the code to see if it still works.

Early on I do refactoring like this on little details. As the code gets clearer, I find I can see things about the design that I could not see before. (...) When I am studying code I find refactoring leads me to higher levels of understanding that otherwise I would miss.

The comprehension theories covered in Section 2.1 can help us understand what Fowler is saying. In the bottom up strategy, a programmer is building a mental model of a piece of code by deriving several representations at different levels of abstraction from it, for which chunking is a key activity. *Refactoring helps to make the chunks explicit.* The programmer can use a catalog of familiar refactorings in order to make the higher-level chunks visible in the code — which then helps him or her to arrive at the next level of abstraction.

In the top down strategy, refactoring may help as well. For example, a programmer may expect the use of a certain design pattern. He will search the code for it, and if necessary refactor the code in order to make the use of that design pattern more explicit. If the pattern is not yet used, he or she will refactor the code so that it follows the design pattern.

Moreover, *each refactoring corresponds to a hypothesis that a given change should not alter the functionality of the code*, which can be verified by checking that the existing unit tests, still pass after the refactoring.

In short, *refactoring provides a systematic program comprehension strategy*: a developer can start with simple refactorings, gradually trying to simplify the code until he understands the code and sees why it is structured the way it is. In addition to that, during refactoring the developer is actually modifying code himself, *which helps him to internalize the structure of that code*.

5.2. Comprehension Risks

Software improvement has a subjective element to it: what is good to one is ugly to another. The same holds for program comprehension: what is intuitive and clear to one programmer may be hard to follow for another. In combination with collective code ownership, this may lead to conflicting refactorings applied repeatedly to the same piece of code. Moreover, as more code gets changed, more programmers need to re-understand software they once understood, even if there where no functional changes.

Another comprehension concern is the way in which XP deals with comments. Fowler's book on refactoring includes a chapter on bad smells, which also discusses the use of comments. Fowler observes that although comments are not a bad smell, they are often used as a "deodorant," and recommends to refactor bad code instead of commenting it [17]. The risk clearly is that this may lead to not writing comments at all. More specific advice is given by McConnell [28], who basically agrees by stating that *comments done well can greatly enhance a program's readability; Comments done poorly can actually hurt it.* and then continues to spend 40 pages with specific guidelines for writing comments well.

6. Evolving a Simple Design

XP's design philosophy is minimalistic and pragmatic. It does not start with a full up-front analysis and design: instead, it only requires a quick analysis of the entire system and then begins working on the first iteration, building the smallest useful system in approximately three weeks. Moreover, XP does not distinguish between analysts, architects and programmers: every developer is responsible for both design and programming. Furthermore, developers evolve a design by continually simplifying the existing code base through refactoring. Thus, instead of a significant design effort up front by a small group of people, XP opts for a little design at every modification, done by who ever is implementing the change.

XP aims at arriving at the the simplest design that runs all the acceptance tests. *Simplest* is defined as follows [2]:

- The code and tests together communicate everything the developer wanted to communicate;
- The system contains no duplicate code;
- It has the fewest possible classes;
- Each class has the fewest possible methods.

Because of this requirement for simplicity, generalizations not needed to get the current acceptance tests running are not implemented. Thus, class structures promising simplifications in forthcoming iterations or features potentially needed at a later stage are not taken into account in the design. XP takes the position that such generalizations are costly to build and to maintain. They will be implemented only when they are needed to eliminate code duplication in the *current* system.

In XP, the design is generally not explicitly documented. Software diagrams tend to get associated with heavyweight processes, in which a lot of time is spent on drawing and maintaining diagrams that are not actually used. Fowler suggest that UML diagrams in XP can be used to explore a design before starting to work on a task or iteration [18]. He advises to draw diagrams only if they can be kept up to date with little effort, to put these drawings on places where everyone can see them, and to throw them away as soon as they are not used anymore. They are not artifacts that have to be maintained.

The habit of not documenting designs is expressed by the XP rule that "the source code is the design". This suggests a strong relationship with reverse engineering tools, which we explore in Section 8.

6.1. Comprehension Benefits

XP vehemently fights over-designs. It asserts that the total costs spent on the construction, maintenance and understanding of such over-designs are higher than the benefits gained from those situations in which the generalization does come of use. The obvious comprehension benefit of this approach is that in order to understand functionality it is not necessary to wade through all sorts of generalizations that may come in of use at a later stage.

This design philosophy is only possible if

- The team already has a good understanding of the existing code base, so that the team can quickly act if, for example, significant redesign is necessary;
- The code itself is in good shape, making it easy to modify when new features require this.

As we have seen, the other XP practices of unit testing, pair programming and refactoring aim at achieving exactly this. Thus, *"design for today" leverages program comprehension*, taking advantage of the way in which the other XP practices support program comprehension. The result is maximal simplicity for today's design, which in turn contributes to program comprehension itself.

6.2. Comprehension Risks

In XP, the design activity is distributed among many different people, each with his own preferences or earlier experiences. The result may be a mixture of design styles, resulting in a lack of *conceptual integrity*, the most important consideration in system design according to Brooks [8]. XP's defense is the *system metaphor*, which is the guiding principle of how the system works. The notion of metaphor is not unproblematic, though. Fowler, for example, states "I still don't think I've seen metaphor explained in a convincing manner" [18], while Jeffries *et al.* confess that "creating a good metaphor for your program is something we can't yet teach you to do—-we manage to do it about half the time we try" [23].

We have seen that XP tries to benefit from the fact that the team is good in understanding and modifying the current code base: there is no need to worry about potential future changes, and time is saved by maintaining neither design documents nor technical documentation. But what if half way the project comprehension conditions get less favorable? What happens when several people leave, or what if the refactorings done or test cases written turn out to be inadequate? XP plays it high risk / high return: if it works, you save time and money and develop at high speed; if things turn out not to work, you are left to your own devices, without even the normal defense mechanisms such as explicit design and documentation.

7. The Planning Game

Planning in XP is an activity in which the development team and business decide on what to do in each *release* (3–6 months) and *iteration* (1–3 weeks)

A release starts with an *exploration* phase, in which business and development discuss what the system should eventually do. Business writes *stories* for feature requests, and development estimates how long each story will take to implement. If a story is too complex to estimate, it is split into smaller stories. In the subsequent *commitment* phase the stories are ordered by business priority. These stories then get implement in the longest, so-called *steering* phase, which consists of a series of small iterations. The beginning of each iteration acts as a planning synchronization point, and can be used to add or delete stories and to adjust estimates if necessary [2].

The iteration structure is similar to the release structure. In the (short) exploration phase, the team translates this iteration's stories into tasks – pieces of work that can be implemented in a couple of days. During the commitment phase, programmers sign up for tasks, and estimate how long the task will take to implement. During the steering phase, tasks get implemented, progress is measured, completed stories are verified via functional tests, and the division of tasks may get adjusted depending on the progress [23].

7.1. Comprehension Benefits

The developers and the understanding they have of the existing code base play an important role in XP's planning approach Thus, collaborative planning first of all *reduces the likelihood of unexpected complications*, as knowledge of the code base is taken into account during planning.

Moreover, both the release planning and each iteration planning start with team-wide discussions on the estimated effort needed for the stories and the way to split each story into clear tasks. This has the effects that the *team shares its understanding* of what is meant by each of the stories, as well as the design issues involved in breaking up the story into tasks. This sharing can easily result in design discussions, *helping the team to identify the weak spots in the code*, potentially leading to new refactoring tasks.

7.2. Comprehension Risks

XP lacks an explicit design phase, in which the consequences of several alternatives are analyzed and compared. Jeffries *et al.* suggest to organize *quick design sessions* [23], but these are so quick ("ten minutes is ideal"), that it is hardly possible to understand the consequences of a single design option. In our project it happened several times that bad designs creeped into the system, which then stayed in the code for a while before we could manage to refactor them away. Making design a more visible activity in the process would have helped to avoid a number of these (in retrospect) bad decisions.

8. Lessons Learned

In the previous sections we have analyzed XP from the program comprehension perspective. In this section, we summarize the lessons learned.

Program Comprehension Research Directions Our study into how XP helps programmers in constructing mental models of the software, raises several interesting program comprehension research directions:

- How can tests be used to record program understanding? Can tests replace the traditional role of documentation? Which category of hypotheses generated during program comprehension can be captured into unit tests?
- How does a "test first" approach affect the understandability of the production code?
- How can a process of *changing* code help to internalize the understanding of that code? What sort of chunks or other aspects of the mental model constructed during understanding can be made explicit through refactorings?
- How does working in pairs affect program understanding? Can we use the discussions between pair partners to improve our models of how program understanding works in the mind?

These are important and promising areas of further research, which, to the best of my knowledge, so far have attracted little or no attention from the program comprehension research community.

Reverse Engineering Application Perspectives The traditional application area for reverse engineering is legacy systems: over the years software has become hard to understand and modify, and recovering system structures at higher levels may help dealing with this unpleasant situation.

The XP process gives reverse engineering a much more pro-active role. In XP, *the source code is the design*. Instead of spending time writing documentation that gets out of date all too soon, XP is satisfied with just inspecting the source. In some cases this will take longer than reading documentation, but this is compensated for by the time saved by not having to maintain that documentation. In this way, XP formalizes what is the de facto standard in many projects anyway.

For such source inspections, XP projects rely on tools such as the Java or Smalltalk development environments, or the reverse engineering capabilities of TogetherJ[4] or RationalRose[5].

[4]www.togethersoft.com
[5]www.rational.com

Reverse engineering research has more to offer than just these tools, and many XP projects could benefit from that. System browsers with architecture extraction capabilities (such as PBS [7] or DocGen [13]) could help to make it easier to navigate through the design. Design patterns in the code can be made explicit using techniques from [24]. Moreover, refactoring could be supported by smell detectors, such as clone detection tools for finding duplicated code [1, 25].

The deployment of such tools could also help to widen the scope of projects to which XP applies, bringing legacy systems within reach of XP. The tool and process implications this involves are discussed in [14].

A key lesson is that reverse engineering techniques should not be simply inserted into the software process, but that they should be tightly integrated with the entire process. Thus, the use of reverse engineering tools should be supported by a comprehensive testing process, to augment the reverse engineering results. Moreover, reverse engineering tools should be supported by well-established means of team communication, such as pair programming. This also opens up new reverse engineering research directions, such as the use of testing to support reverse and re-engineering processes, or the generation of test cases to populate a test suite for a legacy system.

Extreme Programming Revisited We have analyzed the risks and opportunities of XP from the program comprehension perspective. What is the overall evaluation?

The most important benefits seem to be in pair programming, unit testing, and refactoring. These are practices that could be adopted in all sorts of processes, not necessarily in an XP-context. The focus on simplicity and the involvement of developers in the planning process are other aspects positively affecting comprehension without much risk.

The highest risks seem to be in the lack of a design phase or document, and in the extreme way in which "design for today" has to be adopted. In order to get the most benefit out of XP, these practices must be followed. If the project gets into trouble, however, there are few fall back scenarios, as the project may end up in an undocumented system that is hard to modify and not prepared for the changes of tomorrow.

Providing a conclusive overall evaluation is virtually impossible as there is very little hard data about the effects of the various XP practices. XP is largely based on anecdotical success stories, for example in the Chrysler Comprehensive Compensation project (see [23]). The only experiments we are aware of analyze the effects of pair programming [16, 35, 10]. This is an important weakness of XP, and one of the reasons why XP faces significant resistance.

Setting up convincing experiments to assess the XP practices is expensive, as this typically involves developing software using XP and non-XP processes in two separate groups, under otherwise similar conditions. Before such costly experiments can be conducted, it is necessary to have a clear understanding of what risks or benefits should be measured, and how the results of such measurements should be interpreted.

We believe that our analysis can help to achieve exactly this. We have analyzed XP from one particular perspective, program comprehension. We have isolated many individual risks and benefits, and for each of them experiments can be devised in order to grasp the true effects of the corresponding practice. As such, this paper can also be seen as a series of potential empirical study topics and research questions that could help to resolve many of the controversies of XP.

Software Process Design The program comprehension assessment we have conducted for XP can be repeated for other processes. The steps involved include selecting the practices that are most relevant to program comprehension, identifying the risks and benefits for these practices, and an overall evaluation resulting in potential research questions. Processes that come to mind for such an assessment are the (Rational) Unified Process (RUP), the Dynamic Systems Development Method (DSDM), and the Team Software Process (TSP).

We have seen that the practices of XP can have a profound effect on program comprehension, and we expect that certain practices of other processes and the way in which they are combined will have similar effects. This strongly suggest that we can influence program comprehension by the sort of process we choose.

This also suggests that program comprehension should be an explicit element of the software process. The most explicit route would be to make program comprehension a management concern. That is, project management should care about:

- Measuring the understandability of code through software metrics including McCabe or size metrics, effort measurements, such as the time spent per feature implemented when working on given part of the code, or quality criteria, such as coverage of the unit test suite.
- Listening to how the team perceives the understandability of the system and the level of system understanding it has acquired. This can be achieved by periodical interviews or checklists, in which team members express their (subjective) opinion on code parts that are, for example, inherently complex, or badly maintained, as well as their own understanding of certain parts of the system.
- Organizing external comprehension assessments.
- Establishing process support for program comprehension. This includes a careful selection of the individual practices, as well as a proper integration of reverse engineering tools in the software process.

With such actions in place, program comprehension would take a central role in the software development process. Moreover, the team would be able to assess and optimize their comprehension processes, based on the feedback obtained from the comprehension measurements.

9. Concluding Remarks

In this paper, we have seen that extreme programming includes a number of practices that have a very interesting effect on program comprehension: testing can help to get code that is easier to understand, testing can be used to document program understanding, refactoring can act as a program comprehension strategy, and planning sessions can help to spread the design throughout the team. These constitute a largely unexplored territory of important program comprehension research questions.

Second, including reverse engineering technology into the software process requires that this process is adapted in various ways in order to give program comprehension a central role. Thus, the benefits of reverse engineering tools should be leveraged by devising a software process in which reverse engineering is closely integrated with other practices, such as testing, pair programming, and refactoring.

Third, reverse engineering technologies can play a key role in extreme programming. XP relies on the source code as its principal design document, and reverse engineering technologies can help to extract the design from the code. This can also help to widen the scope of XP to the area of legacy systems.

Fourth, extreme programming is in need of empirical support. We have deconstructed the practices into individual risks and benefits concerning program comprehension. These can be used to design experiments for an empirical validation of the program comprehension risks and benefits of XP.

Last but not least, program comprehension should be an explicit element in the software process. Our study shows that the practices such as those incorporated in XP can have a profound effect on program comprehension. These effects should play a role when devising the software process to be used in a development project. We have sketched how program comprehension can become a management concern, by including measurements to monitor and optimize the way in which the the team understands code, as well as the understandability of the resulting code base.

Acknowledgments I would like to thank the *Software Improvement Group* team for great XP moments: Thanks to Alex van den Bergh, Steven Klusener, Gerard Kok, Tobias Kuipers, Leon Moonen, Indra Polak, and Ernst-Jan Verhoeven. Moreover, I would like to thank Tobias Kuipers (SIG), Leon Moonen (CWI), Pieter Jan ’t Hoen (CWI), and Joost Visser (CWI) for reading and discussing earlier drafts of this paper.

References

[1] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance, ICSM'98*, pages 368–377. IEEE Computer Society Press, 1998.

[2] K. Beck. *Extreme Programming Explained. Embrace Change*. Addison Wesley, 1999.

[3] K. Beck and M. Fowler. *Planning Extreme Programming*. Addison-Wesley, 2001.

[4] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[5] P. Benedusi, A. Cimitile, and U. De Carlini. Reverse engineering processes, design document production, and structure charts. *J. of Systems and Software*, 19(3):225–24, 1992.

[6] K. Benett and V. Rajlich. A staged model for the software life cycle. *IEEE Computer*, pages 66–71, July 2000.

[7] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *21st Int. Conf. on Software Engineering, ICSE-99*, pages 555–563. ACM, 1999.

[8] F. Brooks. *The Mythical Man Month*. Addison-Wesley, anniversary edition, 1995.

[9] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[10] A. Cockburn and L. Williams. The costs and benefits of pair programming. In *Extreme Programming Examined; Proceedings XP2000*. Addison-Wesley, 2001.

[11] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[12] W. Cunningham. Episodes: A pattern language of competitive development. In J. Vlissides, editor, *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

[13] A. van Deursen and T. Kuipers. Building documentation generators. In *International Conference on Software Maintenance, ICSM'99*, pages 40–49. IEEE Computer Society, 1999.

[14] A. van Deursen, T. Kuipers, and L. Moonen. Legacy to the extreme. In M. Marchesi and G. Succi, editors, *Extreme Programming Examined*. Addison-Wesley, 2001.

[15] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In M. Marchesi, editor, *Extreme Programming and Flexible Processes; Proc. XP2001*, 2001.

[16] N. V. Flor and E. L. Hutchins. Analyzing distributed cognition in software teams: A case study of team programming during perfective software maintenance. In *Empirical Studies of Programmers; Fourth Workshop*, 1991.

[17] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[18] M. Fowler. Is design dead? In *Extreme Programming Examined; Proceedings XP2000*. Addison-Wesley, 2001.

[19] D. P. Freedman and G. M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews*. Dorset House, 3d edition, 1990.

[20] A. Fuggetta. Software process: A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[21] W. Humphrey. *Managing the Software Process*. Addison Wesley, 1989.

[22] W. Humphrey. Comments on extreme programming. IEEE Computer Dynabook, 2000. `http://computer.org/SEweb/Dynabook/`.

[23] R. Jeffries, A. Anderson, and C. Hendrickson. *Extreme Programming Installed*. Addison-Wesley, 2000.

[24] R. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based reverse-engineering of design components. In *21st Int. Conf. on Softw. Eng.; ICSE'99*, pages 226–235. ACM Press, 1999.

[25] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, 3:77–108, 1996.

[26] P. Kruchten. *The Rational Unified Process. An Introduction*. Addison-Wesley, 1998.

[27] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, pages 44–55, August 1995.

[28] S. McConnell. *Code Complete*. Microsoft Press, 1993.

[29] H. Müller. Understanding software systems using reverse engineering technologies: Research and practice. ICSE-18 Tutorial. See `www.rigi.csc.uvic.ca/UVicRevTut/UVicRevTut.html`, 1996.

[30] H. Müller, J. Jahnke, D. Smith, M.-A. Storey, S. Tilley, and K. Wong. Reverse engineering: A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[31] J. T. Nosek. The case for collaborative programming. *Communications of the ACM*, 41(3):105–108, March 1998.

[32] T. M. Pigoski. *Practical Software Maintenance – Best Practices for Managing Your Software Investment*. John Wiley and Sons, 1997.

[33] G. Salomon, editor. *Distributed Cognitions: Psychological and Educational Considerations*. Cambridge Univ. Press, 1993.

[34] H. Sneed. Planning the reengineering of legacy systems. *IEEE Software*, 12(1):24–33, January 1995.

[35] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair-programming. *IEEE Software*, pages 19–25, July/August 2000.

[36] K. Wong. *The Reverse Engineering Notebook*. PhD thesis, University of Victoria, 1999.