Formal Methods and Extreme Programming*

Hubert Baumeister

Institut für Informatik
Universität München
Oettingenstr. 67
baumeist@informatik.uni-muenchen.de

1 Introduction

Lately an agile software development process called Extreme Programming (XP) has received a lot of attention. For small projects XP promises to produce software that provides business value faster than traditional software processes. In addition, XP is designed to cope with changing requirements and to provide early feedback to cope with unclear requirements. On the other hand, people believe that XP is not very well suited to produce provably correct software because XP does not do up front specification and design.

In this position paper I discuss how the software development process of XP can be adapted to produce provably correct software and what kind of support by formal methods is needed to make this work.

2 XP

In the traditional approach, within an iteration or release, first a set of features (use-cases) that should be implemented and their requirements is determined, then a specification and/or a design model of the software is produced, finally, the system is implemented and tested. Thus the system can be used (and thus produces business value) only after the first iteration, which may take several months to complete. Further, the farther the process is within an iteration the more difficult and costly it is to react to change in the requirements.

Instead of looking at a set of features at once and producing a design for these features, in XP the software is designed and implemented feature by feature. Each feature is designed and implemented only by looking at the new feature and the already implemented features. When looking at a new feature, one tries to find the simplest design that implements the new feature together with the old features. Features that are going to be developed in the future do not contribute to the current design. When adding a new feature it might happen that the old design does not fit with the new feature. In this case the software needs to be refactored, that is changing its structure without changing its functionality. However, refactoring may be dangerous as there is no guarantee that the refactoring steps do not change the implemented functionality. To address

* to be presented at the workshop on Evolutionary Formal Software Development 2002.

this problem, XP requires automated tests. Automation is very important as the tests are intended to run very often to ensure that after each change of the code the code still has its intended functionality.

The advantage of XP is that almost from day one software is produced that can be shown to the customer for feedback and that could be sold to produces business value. In addition, since the design of the current implementation does not take into account any features to be implemented in the future, it is easy to react to changes in the requirements by exchanging features.

3 Formal Methods and XP

Thus it seems that XP and formal methods do not go together because XP lacks a design or specification phase. However, as described above, tests are very important in XP and XP advocates writing the tests before writing the code. Thus writing tests first can be regarded as specifying the intended behavior. Of course, tests are not sufficient as specifications as they can only show that a program fails and cannot guarantee correctness. Therefor, to apply formal methods to XP, tests should be replaced by formal specifications. However, one of the main principles of XP is that it should be easy to run tests. Therefore, when replacing tests by specifications, it should be easy to produce an initial proof that the program is correct and to replay the proof even when program and specification have changed.

The proposal of this paper is a software development process using formal methods that builds incrementally the specification, the program, and the proof that the program satisfies the specification. In the first step, a property of the intended software is selected, then that property is implemented by a program and a prove is produced that the program satisfies this property. This can also be done by extracting the program from its proof. Next, a new property is added and implemented. The old property should still hold and one would like to reuse the original proof to show that. However, chances are high that, to keep the program simple, the program needs to be refactored. In this case, the proof for the old property should be still reusable, maybe after being transformed according to the refactorings applied.