Utilizing the Micro-processes of Software Development for Defect Prevention

Sebastian Jekutsch

Freie Universität Berlin, Institut für Informatik
AG Software Engineering
jekutsch @ inf.fu-berlin.de

This research proposal is based on the assumption that examination of the detailed coding and design process can be utilized to prevent software defect insertion. This *micro-process* of software development, which can be captured during programming work, describes root causes and typical episodes of making errors. It also aims for other process improvements.

Means of quality assurance in software engineering

In recent years progress in quality assurance for software products has mainly been made in the area of testing, i.e. executing the software with the aim to cause failures which in turn are signs for software defects. Inspection techniques seek defects in software artifacts or any other intermediate document directly. As it is, quality assurance departments in software organizations recognize their job mainly as "hunting for bugs". Of course, after finding such problems they still need to be resolved. It's a strange phenomenon that software engineering seems to accept software defects as to be removed after being created by professionals.

In contrast, the research presented here focuses on preventing defects from being inserted in the first place. We assume that

- defects are actively being built into software, because programmers – being humans – make errors
- making errors is not bad luck and therefore defects do not occur by chance and should not be treated by statistical means only
- defect prevention, i.e. prevention of making errors, is mainly a task of process observation and process adjustment
- analysis of defects' root causes is a matter of empirical investigation done on site

Many efforts have already been undertaken by others like strengthening the programming language semantics (e.g. type systems), supporting psychologically good practices (modularization, information hiding, structured programming, object-oriented design, etc), and supporting as well as investigating the programmer's work (modern development environments, graphical debuggers, pair programming, design-

by-contract, among others). In contrast to these, we will focus on the work of the programmer and enhancement of her process and practices in coding as well as in software design. Only little work focused on this, the best known being the Personal Software Process [1].

Some terminology

Before introducing the *micro-process* of programming we need some consensus on the terms used. A software *defect* is a weakness of the software code, i.e. a bad state of the code. A defect often but not necessarily results in a software *failure* during runtime, i.e. the event of not acting as specified. In only rare cases, failures are not caused by defective software but by other reasons.

The well-known techniques of quality assurance mostly ignore the cause of the defect, the human *error*. An error is a human failure: the event of acting against intention (consciously or unconsciously) and making defects in code or intermediate software artifacts like design documents. The errors in turn are caused by human *misbelief* about requirements, computational issues, other system parts behavior, or simply programming language semantics. Like defects, misbeliefs may but do not need to result in errors. Unlike failures, errors are not caused by misbeliefs only. Other causes typically are typing errors, oversights, or interruptions during work, just to mention a few. So far, only little research systematically investigated human error and misbelief as a cause for defects in software development.

Micro-processes of software development

The *micro-process* is a description of the actions undertaken in actually creating low-level design documents or code. It is named in opposition to the concept of (macro-)"process" in software development which doesn't capture human's performed work but only the team participants' high-level tasks, and the work's result. Usually, the atoms of the macro-process are the code changes stored in revision control. In contrast, we will examine the "particles", i.e. intermediate code or document changes, how they occurred as well as tool and time usage.

There has been some work on examining programmer's work to build better development tools, for example shortcuts which support typical activities undertaken [2]. Additionally, cognitive psychologists have developed micro-process models of programming and program comprehension [3]. Others use micro-process examination to analyze a team member's overall contribution and the distribution of defect insertion rate [4].

In this paper, we will concentrate on the task of programming. With ongoing investment in formalizing the design process and in using tools to support the

construction of well-defined design documents, the ideas presented here can be extended to design activities of software development as well.

The first research goal is to develop a general vocabulary for describing the micro-process of coding. It will be based on events occurring on some code part, like

- Changing code parts
- Browsing through code
- Executing the program
- Pausing for a while, etc.

These events will be grouped in typical episodes like

- Trial-and-error cycles (which itself is a special kind of removing defects)
- Copy code, paste it and change it (as a way of changing code fragments)
- Interrupted work, etc

The aim is to log events and episodes and find statistically relevant relations to introduced defects, i.e. identifying typical patterns of defect insertion. For example, typical “anti-patterns” and indicators are

- Being interrupted and not resuming work where left off
- Changing a small part of code very often
- Not changing copied code fragments correctly, etc

By carefully examining the causes it will be possible to predict defects “just-in-time” and therefore prevent them. At least it will be possible for the developer to reinvestigate how the defect has been introduced, i.e. performing some root cause analysis, which may result in some micro-process improvement.

Other chances on micro-process analysis

As an outlook and a source of research ideas, some analogies of micro-processes with other concepts of software development may lead to new possibilities of utilizing recorded micro-processes. For example, micro-processes record way more single code changes and rejected code fragments than are represented in typical revision control. Applying techniques from software archaeology and software evolution for micro-changes may give additional insights like generating product-specific anti-code-patterns or tracking the evolution of code copies.

Not only static changes are recorded but also the dynamics of coding. This may be used for enhancing the programmer’s tools with “macro-fying” typical work episodes or for suggesting places to look at because of past browsing sessions. Defect prevention will benefit from analysing the dynamics of programming in the past as well as parallel to actually coding.

Since the micro-process is represented in an abstract, event-based, and episodic way one gets the chance to re-examine (and even re-execute) post coding sessions, for

example to refresh one's mind about what has been done lately. Additionally, re-playing after some time may help in finding faults which haven't been detected during work because of inattentiveness. The work also may be automatically analysed for quality criteria such as locality of work. Psychologists' discoveries on programming models can also be validated more easily.

Note that typically micro-processes from every programmer in a team can be collected as a whole, improving knowledge sharing.

Mid-term tasks

The next steps of research are as follows.

1. Exploratory view on defects and errors to get a feeling for their occurrences and to later systematically describing the genesis of expensive "bugs".
2. Applying psychologists' work on programming models and observation of programming episodes needs to be undertaken to finally define the vocabulary of micro-processes.
3. Tools for capturing the micro-process automatically while coding need to be developed. This will mainly be the focus of student's theses.
4. After we built a framework for measuring the programmer's work, a lot of effort need to be put in collecting coding episodes and analyzing them to develop and finally support hypotheses on when and why errors are being made and defects are being inserted.
5. This will be supported by transferring general knowledge on human error to the specific task of programming.
6. In parallel some of the other possibilities of utilizing micro-processes may be examined in more detail.

References

1. Watts S. Humphrey: Using A Defined and Measured Personal Software Process, IEEE Software May 1996 (Vol. 13, No. 3)
2. Janice Singer, Timothy Lethbridge, Norman Vinson, Nicolas Anquetil: An Examination of Software Engineering Work Practices, Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, Toronto, Ontario, Canada
3. Simon P. Davis: Models and theories of programming strategy, International Journal of Man-Machine Studies, 1993, vol. 39, no. 2, pp. 237 - 267
4. Philip M. Johnson : Project Hackystat: Accelerating adoption of empirically guided software development through non-disruptive, developer-centric, in-process data collection and analysis, Technical Report csdl2-01-13, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, November, 2001.