Towards a Systematic Approach for Driving Micro-Level Software Development Process

Salman Mirghasemi, Olivier Buchwalder, Claude Petitpierre
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne
Lausanne 1015, Switzerland
{*salman.mirghasemi, olivier.buchwalder, claude.petitpierre*}*@epfl.ch*

Abstract

Software developers encounter many new and usually difficult problems in their every day work. Solving every new problem requires taking an appropriate strategy and careful planning, that needs full concentration and focus on the problem and enough consciousness about the current state of development. However, due to many factors, such as working on more than one task at the same time, the high degree of granularity of tasks, interruptions and the complexity of problems, developers usually have difficulties in obtaining the required concentration and focus on their current local problems and making progress in their work. This paper proposes a novel approach, which lets a developer follows a systematic step by step problem solving process in order to be focused on only one target in every moment while all needed details from the past and future steps are present to her. Moreover, this approach helps a developer to better understand her software development process and discover new ways for improving it.

1. Introduction

In the last two decades, software production has been greatly facilitated by significant advances in different aspects of software engineering. In parallel, tools and methods used by software developers to produce programs have immensely improved over time. Nowadays, developers employ high level modeling and programming languages to design and write programs, comprehensive integrated development environments (IDEs) to edit, navigate and manage project artifacts in one place, advanced communication and collaboration tools to be connected and synchronized to other teammates and various kinds of online knowledge repositories to find answers to their questions [5], [6], [13], [14].

In spite of all mentioned facts software development is still a hard and challenging job for developers [5], [9]. Although the nature of software development has not changed a lot in the last decades and, it is still a continual problem solving process, this process has become more complicated. Today, in order to solve a problem, due to the larger size of programs, also the employment of many third party libraries and programs, a developer has more questions to answer, more concerns to worry and more options to choose. It has added more details to the process. Also a developer is not just a programmer anymore but also a divider, finder and integrator. She divides a challenging problem to smaller ones, finds the corresponding appropriate solutions from many available solutions and integrates partial solutions to build a complete solution for the primary problem. It has increased the granularity of the process. Moreover developers encounter many new and different problems that require taking different strategies for developing a solution to them. It has made the process more heterogeneous [9].

Besides the complexity of the development process, which makes it hard to follow for developers, frequent task switches and interruptions distract them from their targets [11]. Developers must remember goals, decisions, hypotheses, and interpretations from the task they were working on and risk inserting bugs if they misremember. The difficulty of recovering from an interruption have been shown by many studies [9], [11], and some of them suggested externalizing developer's task context - methods they have examined, decisions in progress, and other information - in a tool [9], but no method or tool has been provided.

The way developers manage the development process and the strategies they choose during this process have crucial roles on their performance and productivity [13], [2]. Currently developers keep and manage this complex process mostly in their mind [5], [8]. This paper proposes a discipline that by adopting it, developers can free their mind from all details of the process and be more focused on

their local targets while all past and future steps are available to them and also safe against interruptions. Moreover this approach changes the complex process of software development to a systematic step by step process for software developers.

The rest of this paper is organized as follows. In the next section, the issues addressed in this paper are described in more details. In Section 3, the discipline will be presented. In Section 4, existing related works are briefly reviewed and compared to our approach. The last section summarizes the conclusions and describes our future work.

2. Problem Description

Software development is a heuristic process, in which a developer accomplishes a task by proceeding through a long chain of plan-do-check cycles. The details of this path are not known at the beginning, and as the developer proceeds, next possible points become more visible and more precise. Then, she can choose a strategy for reaching them. Developers usually keep the details of the task's progress, such as the past points and their outcomes, also the future points and the strategies and plans for reaching them, in their minds. The immediate consequence of this practice is that with any distraction or gap in work, a developer may lose some parts of the path or even the whole path completely. It has been shown by many studies that due to many different sources of distraction during software development such as various kinds of interruptions, high degree of granularity of tasks and blocking, losing the path happens quite often [9], [11].

Keeping the track of tasks' progress in mind also has a few other drawbacks. Using this practice, the developer is not forced to specify the next point clearly and also to define the strategy and the plan for reaching the next point precisely and properly. Therefore, she continues with an implicit vague understanding of the problem and the plan, which leads to spending more time for reaching the next point, and therefore more time for accomplishing the task. This case is less visible when developers add or change a functionality of a program but it is very prevalent when developers read code to answer to a few implicit questions, or they try to find a defect causes a bug [8], [7].

Moreover, because of inherently complicated nature of software developments, developers are usually completely drown in their work, and therefore they have not the track of time. Sometimes, specially in the case of blocking or very slow progress, it is necessary to stop and rethink about other possible ways to go through the current step or even skip it for the moment.

All these facts show that developers need a more effective way to maintain and manage the process of development, which is guarded against interruptions and provide them enough awareness and consciousness about what have been done, what is the current goal, what is the current step and what are the next steps to reach the final goal.

3. Discipline

Before getting into the discipline explanation, it is required to describe a few concepts with more details. Developers usually have more than one *tasks* at hand in a *working session*. For example a developer might have these tasks at hand, "adding feature X", "fixing bug Y" and also "participating in an on-line meeting and discussing the next release design of a module". An *active task* is the task that the developer is working on right now.

If we consider the *path* traversed by a developer for accomplishing a task, we can usually recognize some *internal points*. For example for "adding feature X", we can see the following internal points after the developer started the task:

1. She decided about the implementation of this feature. She decided to use one of the available libraries support this feature.
2. She found the appropriate library.
3. She has studied the library and knows how it must be customized and integrated.
4. She has implemented feature X using the library. The task is done at this point.

Similarly, it is possible to recognize deeper internal points in the path from one point to the next one. For example if we consider the path from the first point to the second point, we can name these internal points :

1.1. The developer searched the Internet and have a list of libraries that support feature X.
1.2. She read about the differences of these libraries on the technical forums,
1.3. She chose the appropriate library. At this point, the developer is at point 2.

If we continue going to deeper internal points, we will finally reach to atomic actions of the developer. When a developer starts a task, She has some ideas about the upcoming next points in her mind and as she goes forward, she updates these possible future internal points. We name these future internal points, the *strategy* of the developer for reaching the next point or accomplishing the task . Sometimes, the strategy covers all the path between two points, but there exists also cases in which the developer just knows a few initial steps of her plan and she will figure out more details later. For example in order to fix bug Y, this might be a strategy:

1. Reproduce the bug.
2. Recognize the defect.
3. Fix the defect.

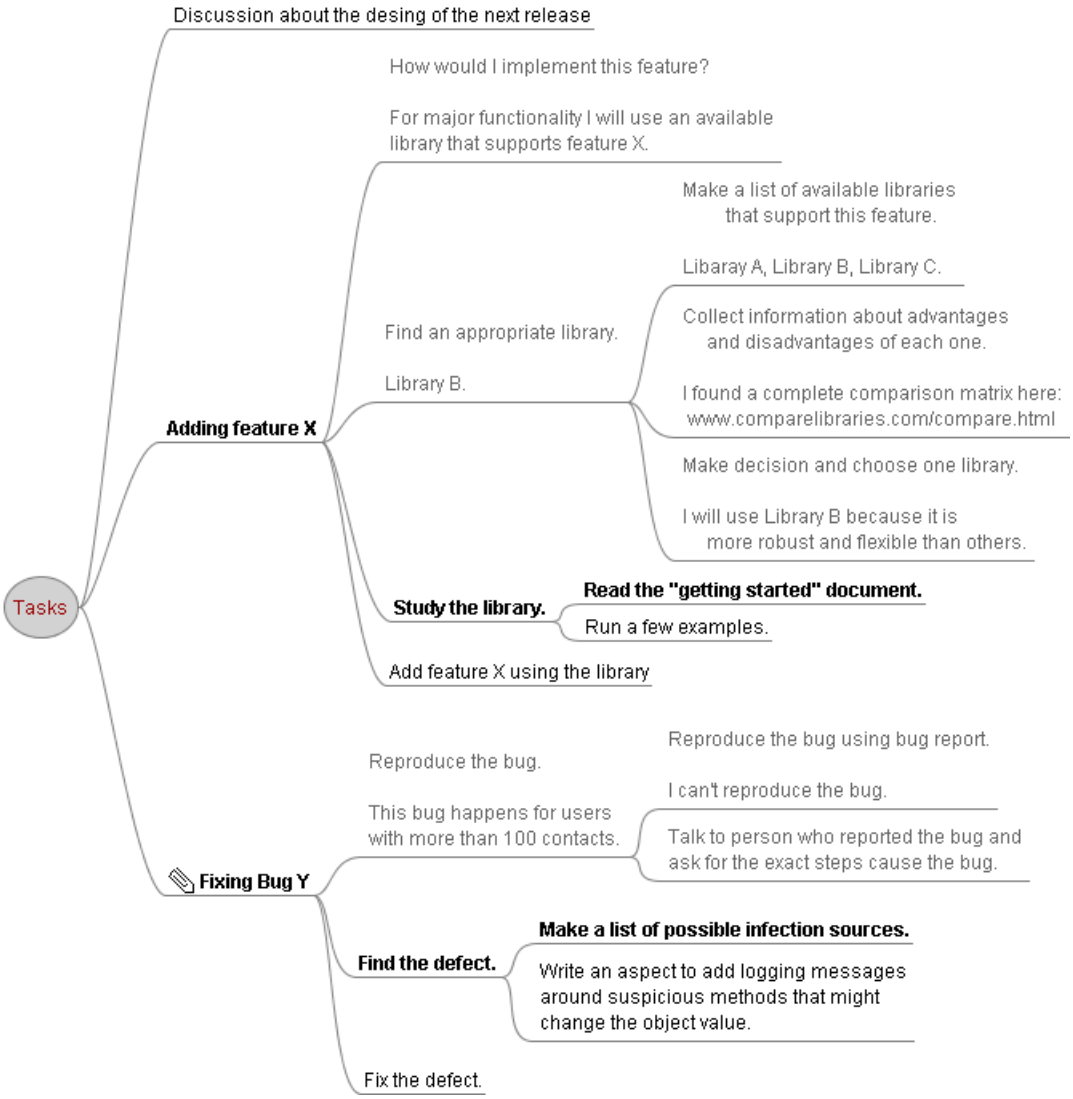

Figure 1. An example of Work Progress Tree (WPT).

The strategy of the developer covers whole the path at the first level, but once she has reproduced the bug and she is planning for finding the defect, she just knows a few initial steps and she will decide about the next steps later. We name the first strategy a *complete strategy* and the second one a *partial strategy*. Also, we say a developer is in a *blocking* situation when she has no strategy for reaching the next point and she is progressing very slowly.

We can see that all the past points and future points can be presented as a tree for each task. We name this tree a Task Progress Tree (TPT). If we put all tasks' progress trees under one shared parent node we get a Work Progress Tree (WPT). An example of one possible WPT three is shown in Figure 1. In this picture, each main branch shows one task progress tree. The nodes with gray text show the past points, the nodes with bold text show the current next point and the nodes with normal text show future points for each task. The pen icon marks the current active task in the WPT. We can see that the developer has written short notes at some past points about the outcome of the step or the facts she has found during that step.

The discipline is defined in three levels, where every level adds new features to the previous one. These levels are shown in Table 1 .For every level, *Goals* (the benefits gained by developer by adopting this level), *Principles and Rules*, *Guide Lines* and *Tool Support* (the features should be supported by the tool used by developers to adopt the discipline) have been described.

At the first level, the developer must keep the WPT out of her mind. The developer is not required to update WPT

Discipline Level	Goals	Principles and Rules	Guide Lines	Tool Support
First	- Reducing the disruptive effect of interruptions, task switches and gaps. - Employing a systematic approach for planning and making progress. - Being focused on one target until it is achieved or cancelled.	- Keep the WPT out of your mind on a sheet of paper or a program. - Update WPT when you finish one step or switch to another task.	- Focus on your current target and just write down everything else which is not related to that. - If you are unsuccessful in reaching the next point (you are in a blocking situation), define a strategy for this step.	- Creating and Updating WPT.
Second	- Improving time management	- Estimate and specify the required time for each step you are going to start.	- Try to be committed to your estimates. - Find reasons behind overrun estimates and improve your estimates.	- Accepting time estimates for each step and Alarming the developer when the time is passed.
Third	- Improving the process	- Gather and Analyze data about the process.	- Find ways to reduce the time spent on each step.	- Gathering timing and programs' usage data automatically.

Table 1. The three levels of discipline.

regularly, but she can update it every time she comes back to the WPT to check her current state and make decision about next steps. By adopting this level, a developer can reduce the effect of task switches, interruptions and gaps in her work. Furthermore, he employs a systematic step by step approach in order to accomplish her tasks. This let her free her mind from all details of process while she is only focused on her current local target.

At the second level, the developer can specify a time estimation for each step, and he will be notified after this period passed. In this way, the developer has more control on the time she spends on each step.

At the third level, the developer uses the gathered data of the process to analyze her performance and improve the development process. There is one main question that should be answered by the developer: "How could I reach the next point in a shorter time?". However, many more specific questions can be derived from this question. For example, "How good are the strategies I have chosen?", "Which skills can help me to speed up my progress?', "Which parts in the process can be automated or semi-automated?" and "How much time I would gain from upgrading my computer's hardware?".

The first level can be adopted using a simple mind mapping tool like Freemind [3] for drawing and updating WPT, but the second and third levels require a tool that supports time estimation for each step, alarming, gathering timing data and profiling programs' usage in addition to creating and updating WPT. For this purpose, we have developed MindRoute [10]. A user interacts with MindRoute through a command line interface which shows the current active task and the associated path at every moment and it also provides the required actions for creating and updating WPT. To support time estimation and alarming, MindRoute associates a dedicated timer for each task and, when the developer switches to another task or pauses the current task because of an interruption, it stops the current task's timer and when she resumes a task, it also resumes its timer. Currently MindRoute has only a command line interface, but the addition of a graphical interface has been planned for the next release.

There are three important questions about the WPT updating mechanism. The first one is "How much a node's description should be self-descriptive?". The answer of this question depends on the goal a developer seeks from keeping the WPT. For example, in Figure 1 every node is self-descriptive enough to be understandable by the reader. At minimum level, it is enough that a node contains sufficient information that a developer could remember her mental state in the same or at most in the next working session (in case of a gap in work). If a developer seeks other goals such as reporting, detailed analysis of her work or tagging changed parts of code by the related subtree of WPT, then node descriptions should be written with more details.

The second question is that "How many levels the developer should go down in the WPT tree?". There is no strict rule to be given as the answer of this question but there are two main factors which should be considered: The complexity of task and the time required for passing the step. Each one of these factors increases, it is more needed to split up a step to smaller ones.

The third question is that "How often the WPT should

be updated?". It would be ideal if WPT be updated every time the developer changes her current next point due to a task switch, an interruption or finishing/canceling the current step. At the same time, it should be considered that this action itself doesn't distract the developer.

4. Related Work

The Personal Software Process(PSP) [4], [12] was proposed by Watts S. Humphrey to help developers to improve their personal software process. The main idea of the PSP is to understand and improve software process thorough planning, tracking, measuring, and analyzing the defined process. The PSP approach is different from our approach in a few ways. First, PSP doesn't provide any advantage to developers in driving the development process. In order to adopt PSP, developers are required to collect data about their development process without gaining any benefit from this data collection at this stage. But by adopting the proposed discipline in this paper, all required timing data will be collected automatically according to WPT updates. Second, using our approach the collected data has more semantic due to the availability of WPT. Third, analyzing the collected data and exploring ways to improve the process is much simpler. The developer has only one main question to answer: "how can I finish steps in shorter time? " and all other questions will be derived from this question.

Pair Programming [15] is a software development practice mainly introduced by Extreme Programming methodologies. Pair programming refers to the practice whereby two programmers work together at one computer, collaborating on the same algorithm, code, or test. One member of the pair is the *driver*, who actively types at the computer, or records a design or architecture. The other plays the role of *navigator*. One of the roles of navigator in Pair Programming is keeping and updating the WPT and freeing the mind of the driver from all details of the process and, from this perspective it is similar to the proposed discipline. Among many benefits named for Pair Programming, similar benefits that we are following from adopting the discipline such as increased work focus, faster work pace and fewer distractions, can be found [1].

5. Conclusions and Future work

In this paper, we introduced the Work Progress Tree(WPT) model for micro-level software development process and based on this model, we have proposed a discipline for driving and improving the software development process.

In the future, besides adding a graphical user interface to MindRoute, we plan to enhance it by adding the ability of recognizing potential parts of process for improvement and proposing them to the developer.

References

[1] A. Begel and N. Nagappan. Pair programming: what's in it for me? In *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 120–128, New York, NY, USA, 2008. ACM.

[2] B. de Alwis, B. de Alwis, G. C. Murphy, and M. P. Robillard. A comparative study of three program exploration tools. In G. C. Murphy, editor, *Proc. 15th IEEE International Conference on Program Comprehension ICPC '07*, pages 103–112, 2007.

[3] Freemind. http://freemind.sourceforge.net.

[4] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995.

[5] A. Ko, B. Myers, M. Coblenz, and H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Transactions on Software Engineering*, 32(12):971–987, 2006.

[6] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proc. 29th International Conference on Software Engineering ICSE 2007*, pages 344–353, 20–26 May 2007.

[7] T. D. LaToza. Answering common questions about code. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 983–986, New York, NY, USA, 2008. ACM.

[8] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 361–370, New York, NY, USA, 2007. ACM.

[9] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 492–501, New York, NY, USA, 2006. ACM.

[10] Mindroute. http://ltiwww.epfl.ch/~mirghase/mindroute.

[11] D. E. Perry, N. A. Staudenmayer, and L. G. Votta. *Understanding and Improving Time Usage in Software Development*. 1995.

[12] Personal software process (psp). http://www.sei.cmu.edu/tsp/psp.html.

[13] M. Robillard, W. Coelho, and G. Murphy. How effective developers investigate source code: an exploratory study. *Transactions on Software Engineering*, 30(12):889–903, 2004.

[14] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 21. IBM Press, 1997.

[15] L. Williams and R. Kessler. *Pair Programming Illuminated*. Reading, Massachusetts: Addison Wesley, 2003.