Teaching Future Software Developers

Václav Rajlich

Wayne State University
Department of Computer Science
Detroit, MI 48202, U.S.A.
rajlich@wayne.edu

Abstract

Teaching software developer skills should be a fundamental part of software engineering curriculum. The current industry relies on evolutionary and agile processes that add one feature or property at a time. The main task of these processes is software change. Results of the recent research allow this topic to be taught on both undergraduate and graduate level. Phased model of software change (PMSC) divides the task of software change into phases that are sufficiently well-understood and suitable for teaching in the undergraduate course. Among the phases, concept location finds the module(s) to be changed, impact analysis assesses the full extent and difficulty of the change, prefactoring reorganizes software to make it suitable for the change, actualization implements the new feature, and postfactoring cleans up the aftermath.

Categories and Subject Descriptors D.2.9 [**Management**]: Software Process Models

General Terms Management, Verification.

Keywords Phased model of software change, concept location, impact analysis, actualization, refactoring, verification, software development, teaching

1. Introduction

While software engineering is more than just software development, the software development is the core and indispensable part of software engineering education. This is what most of the graduates will do, either in their industrial careers or in their further studies in the graduate schools. Moreover, the other parts of software engineering exist only for the purpose of supporting software development, and hence their proper grasp requires fluency in software development skills.

According to the recent surveys, agile development moved into the position of the current software development mainstream [1]. However for the purpose of this discussion, the following terminological clarification may help: *Evolutionary processes* are all processes that build the program by adding one feature after another to an already existing – although incomplete – program.

Agile processes are special evolutionary processes which have specific and well-defined process roles and practices, like daily meetings, test-first development, and so forth. The following formula describes the relationship of the process categories:

$$Agile \subset Evolutionary\ processes$$

The fundamental task common to all evolutionary processes including agile processes is software change that adds a new feature or new property to software [2]. Since this task is so fundamental, it should be a part of every software engineering curriculum.

In our undergraduate software engineering course, we teach the phased model of software change, see section 2. Section 3 briefly summarizes past research that made this approach possible, and the future research that aims to help software developers in their tasks and further improve teaching software development.

2. Course Topics

Our undergraduate software engineering course (csc4110) teaches software developer skills and emphasizes developer's tasks in agile and evolutionary software development processes. A particularly important task is software change; the rationale for this choice is summarized in [3].

2.1 Phased Model of Software Change (PMSC)

PMSC consists of several phases; PMSC process enactment consists of some or all of these phases. Our course presents all phases and for each, it presents select applicable techniques. It also serves as an introduction to the sizeable research literature dealing with each of the phases. The approach is explained in detail in [4]. Additional references are [2, 5-7]

Software change starts with *initiation* where the programmers decide to implement a specific change in the software. This phase includes activities that are traditionally presented as requirements elicitation, analysis, tasking, and prioritization.

The next phase is *concept location* in which the developer finds the software code modules that ought to be changed. These modules are the places where the new functionality or the bug correction resides. Concept location may be an easy task in small programs, but it can be a very difficult task in very large programs [8, 9].

Very often, software change is not localized in a single program module, but it affects other parts of software. *Impact analysis* is a phase that determines all these other parts. It starts where concept location stopped, i.e. it starts with the modules identified

by concept location as the places where the core of the change should be made. Then, it looks at interacting modules and decides to what degree they are also affected [10, 11]. Impact analysis, together with concept location, constitutes the design of software change, where the strategy and extent of the change is determined and precedes the phases where the code is actually modified.

Actualization is the phase that implements the new functionality. The new functionality is implemented either directly in the old code, or it is implemented and tested separately and then integrated with the old code via *incorporation*. In either case, the change can have repercussions in other parts of software. Change propagation identifies these other parts, making the secondary modifications. Methodologically, change propagation is similar to impact analysis because it also identifies the other affected modules; only this time, the actual code modifications are implemented.

Refactoring changes the structure of software without changing the functionality. During the typical SC, refactoring is done as two different phases: before the actualization, and/or after. When it is done before actualization, it is called *prefactoring*. Prefactoring prepares the old code for the actualization and gives it a structure that will make the actualization easier. For example, it gathers all the bits and pieces of the functionality that is going to change and makes the actualization localized, so that it affects fewer software modules; this makes the actualization simpler and easier. The other refactoring phase is called *postfactoring*. Actualization can make a mess; Postfactoring is an opportunity to clean-up this mess. Postfactoring also can address technical debt that has accumulated during previous software changes [11].

Verification aims to guarantee the quality of the work, and it interleaves with phases of prefactoring, actualization, postfactoring, and conclusion. Verification uses various strategies and techniques, including unit, functional, and structural testing, and inspections.

Conclusion is the last phase of software change. After the new source code is completed and verified, the programmers commit it into a version control system. SC conclusion is an opportunity to create the new baseline, update the documentation, prepare a new software release, and so forth. If SC is done by a programmer within a team, then both the SC initiation and conclusion are team activities and they may differ depending on the process that the team uses.

PMSC is an extensive topic and the explanation of individual phases and the related practices constitutes approximately 50% of the lecture time.

2.2 Other Lecture Topics

At the beginning of the csc4110 lectures, the course covers history of software engineering, software life span models, survey of the most common software project technologies (languages, compilers, version control), and the survey of most common software models (UML class and activity diagrams, dependency graphs).

The end of the course is devoted to a survey of other software engineering topics, with the emphasis on software processes. It surveys team practices of agile and evolutionary software development, initial development of software from scratch, the final stages of software life-span, and reengineering. Additional topics briefly covered at the very end are software engineering ethics, management, and ergonomics.

2.3 Laboratory and Projects

The art of software change is practiced in software projects of realistic size and quality; they are the main topic of the parallel one-credit co-requisite software engineering lab (csc4111). We use open source projects that students update. For example in the past, we used WinMerge (http://winmerge.org/) and this semester, our project is Easypaint. (http://qt-apps.org/content/show.php/EasyPaint?content=140877)

Each team of students deals with a project and each individual student makes several changes of the project code; they are expected to use PMSC in their software changes. While students work individually on their changes, they have to resolve the conflicts and create new baselines as a team; this is similar to the current common practice. Table I illustrates a 14-week schedule of csc4111, with assignments and due dates for software changes highlighted. As the complexity of the assigned changes increases, so does the available time.

Table I. Schedule of the lab

1	syllabus, project tools
2	SVN, Merge and Diff, Wiki
3	GUI technologies: QT, Cmake
4	divide the class into teams, assign change request 1
5	groups meetings + Q&A about the project
6	change request 1 due + team presentation
7	refactoring - in class exercise
8	groups meetings + Q&A about the project
9	change request 2 due + team presentation
10	unit testing - in class exercise
11	groups meetings + Q&A about the project
12	groups meetings + Q&A about the project
13	change request 3 due + team presentation
14	extra credit due

3. Related Research

This approach to the teaching of software development skills was made possible by the recent research. Several process models of software change have been proposed, among them TDD [12] or "Legacy code change algorithm" [13]; they are special cases of a more general and more recent PMSC used in our course [4].

3.1 Past Research

Individual phases of PMSC received considerable attention from the research community, and the selected research results, explained on undergraduate level, constitute large part of the course. The last missing piece of the PMSC puzzle was concept location that was investigated mostly during the last 10 years; the review of the recent results appears in [8]. Our course presents techniques of concept location that do not require pre-processing of the code and hence they are easy to use by the students. They are dependency search [14] and grep search including grep query formulation [15]. The refactoring also have been a subject of an extensive research and our course presents renaming, function splitting, base class extraction, and component class extraction [16, 17]. Similar selections have been made from the techniques applicable to the other phases of software change.

An evolutionary software process constitutes the core of the undergraduate software engineering course. A discussion of evolutionary process practices in a research context has been published in [18].

3.2 Future Research

The current and future research in evolutionary code development has a goal of improving productivity and quality of developer's work and may impact undergraduate teaching of software development in the future. In particular, the individual phases of PMSC are still investigated and new promising techniques and practices regularly appear on the scene.

Another research issue is IDE that would seamlessly support PMSC. A step towards that goal is JRipples that supports concept location by dependency search, impact analysis, and change propagation [19]. A greater integration of the phases would combine these techniques with additional techniques and phases.

The effectiveness of the techniques proposed by the research are empirically validated by user studies and by software repository mining [20]; software repositories record the past evolution, and study of that can lead to new and better tools.

References

[1] (2009). Survey Finds Majority of Senior Software Business Leaders See Rise in Development Budgets. Available: http://www.softserveinc.com/news/survey-senior-software-business-leaders-rise-development-budgets/

[2] V. Rajlich and P. Gosavi, "Incremental Change in Object-Oriented Programming," IEEE Software, vol. 21, pp. 62-69, July-August 2004.

[3] V. Rajlich, "Teaching Developer Skills in the First Software Engineering Course," in International Conference on Software Engineering (ICSE), San Francisco, 2013, pp. 1109 - 1116.

[4] V. Rajlich, Software Engineering: The Current Practice. Boca Raton, FL: CRC Press, 2012.

[5] N. Febbraro and V. Rajlich, "The Role of Incremental Change in Agile Software Processes," in Agile Conference 2007, Washington D.C., USA, 2007, pp. 92-102.

[6] C. Dorman and V. Rajlich, "Software Change in the Solo Iterative Process: An Experience Report," in Agile, 2012, pp. 22-30.

[7] V. Rajlich. (2013). "Software Engineering: The Current Practice". Available: http://www.cs.wayne.edu/~vip/ProjectAndLabs/index.html

[8] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," Journal of Software: Evolution and Process, vol. 25, pp. 53-95, 2013.

[9] M. Petrenko and V. Rajlich, "Concept location using program dependencies and information retrieval (DepIR)," Inform. Softw. Technol., vol. 55, pp. 651-659, 2013.

[10] B. Li, Z. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," Softw. Test. Verif. Reliab., p. early view http://onlinelibrary.wiley.com/doi/10.1002/stvr.1475/abstract, 2012.

[11] S. Bohner and R. Arnold, Software Change Impact Analysis. Los Alamitos, CA: IEEE Computer Society, 1996.

[12] K. Beck, Test driven development: By example: Addison-Wesley Professional, 2003.

[13] M. Feathers, "Working Effectively with Legacy Code," ed. Upper Saddle River, NJ: Prentice Hall PTR, 2005.

[14] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph," in Proceedings of the 8th International Workshop on Program Comprehension (IWPC'00), Limerick, Ireland, 2000, pp. 241-249.

[15] M. Petrenko, V. Rajlich, and R. Vanciu, "Partial Domain Comprehension in Software Evolution and Maintenance," presented at the IEEE International Conference on Software Comprehension, 2008.

[16] M. Fowler, Refactoring: Improving the Design of Existing Code. Reading, MA: Addison Wesley, 1999.

[17] R. Fanta and V. Rajlich, "Reengineering Object-Oriented Code," in International Conference on Software Maintenance, 1998, pp. 238-246.

[18] V. Rajlich and J. Hua, "Which Practices are Suitable for an Academic Software Project?," in International Conference on Software Maintenace (ICSM), 2013.

[19] M. Petrenko, JRipples. Available: http://jripples.sourceforge.net/ (2011).

[20] H. Kagdi, M. Collard, and J. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," Journal of Software Maintenace and Evolution, vol. 19, pp. 77-131, 2007

.