Software Evolution and Maintenance

Václav Rajlich
Wayne State University
Department of Computer Science
Detroit, MI 48202
001.313.577.5423
rajlich@wayne.edu

ABSTRACT
Successful software requires constant change that is triggered by evolving requirements, technologies, and stakeholder knowledge. This constant change constitutes software evolution. Software evolution has gained steadily in importance and recently moved into the center of attention of software developers. There is the new prominence of evolutionary software development that includes agile, iterative, open source, inner source, and other processes. As a consequence, the bulk of software development now happens in the stage of software evolution and this changed the face of software engineering. This paper discusses evolutionary software development and also discusses the software change, which is the fundamental software evolution task. It further discusses research methodologies, teaching software evolution in undergraduate curriculum, and difference between software evolution and software maintenance. For all these themes, this travelogue paper presents the current state of the art and the perspective of the future advance.

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

General Terms
Management, Measurement, Economics, Experimentation

Keywords
Staged model of software lifespan, evolutionary software development, practices of software development, phased model of software change, concept location, impact analysis, actualization, refactoring, empirical techniques, reasoning about software evolution, software maintenance, teaching software evolution.

1. INTRODUCTION
Successful software requires repeated changes that are triggered by evolving requirements, technologies, and stakeholder knowledge. These repeated changes constitute software evolution. The original “Roadmap” paper discussed several aspects of software evolution [1]. In particular, it presented a new model of software lifespan, called “staged model” that clarified the role of software evolution and software servicing (maintenance) in software lifecycle. It also presented a tentative model of the most important evolutionary task, software change that introduces new functionality or new properties into software. This paper discusses the further development of these themes during the past 14 years and also the future directions. Furthermore it discusses new closely related themes that have emerged or gained in importance. A more complete recent coverage of these themes appears in [2].

Historically, software evolution appeared as an unexpected and unplanned phenomenon that was observed in the original case study [3]. Since that time, it gained steadily in importance and moved into the center of attention of software engineers. There were several recent summaries that mapped the field of software evolution: Godfrey and German contrasted software evolution and software maintenance, two terms that some authors incorrectly interchange [4]. Mens and Demeyer edited a book that addresses research issues in software evolution, reengineering, and so forth [5]. This paper builds on these summaries including [2], maps the present state of select software evolution themes, and outlines their likely future directions.

Perhaps the most significant advance since the “Roadmap” paper is in the new prominence of evolutionary software development. It transferred the bulk of software development into the stage of software evolution and through that changed the mainstream of software engineering. Evolutionary software development includes agile development, open source development, and other processes. These issues are related to staged model and are discussed Section 2 of this paper.

Research of the fundamental task of software evolution, software change, has also substantially progressed. The current state and future directions are summarized in Section 3.

Another significant development in the intervening years has been the progress in research methodologies. This is briefly discussed in Section 4. Section 5 discusses the place of software evolution in the undergraduate curriculum. Section 6 discusses software maintenance as a contrast to software evolution. Finally Section 7 concludes the paper.

2. STAGED MODEL OF SOFTWARE LIFESPAN
The original staged model of software lifespan was published in [1, 6] and it was further expanded in [2]. It divides the software lifespan into five stages: Initial development, evolution, servicing, phase-out, and close-down, as represented in Figure 1.

Initial development produces the first version of the software from scratch. During this stage, the developers select the programming

FOSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2865-4/14/05...$15.00
http://dx.doi.org/10.1145/2593882.2593893

languages, libraries, software tools, system architecture, and so forth. These fundamental choices of the initial development set the course for the whole software lifespan; the reversal of these decisions later is very expensive, and often it is practically impossible. Initial development typically contains tasks that are familiar from waterfall (software plan, requirements elicitation, design, implementation), but it's scope and duration is limited, leaving the bulk of the development to evolution. For the discussion of issues in initial development and its role within the context of staged model, see Chapter 14 of [2].

The next stage, *evolution*, is the main topic of this paper. The developers add new features, correct previous mistakes and misunderstandings, and react to the requirements, technologies, and knowledge volatility as it plays out through the time. Software changes are the basic building blocks of software evolution and each change introduces a new feature or some other new property into software.

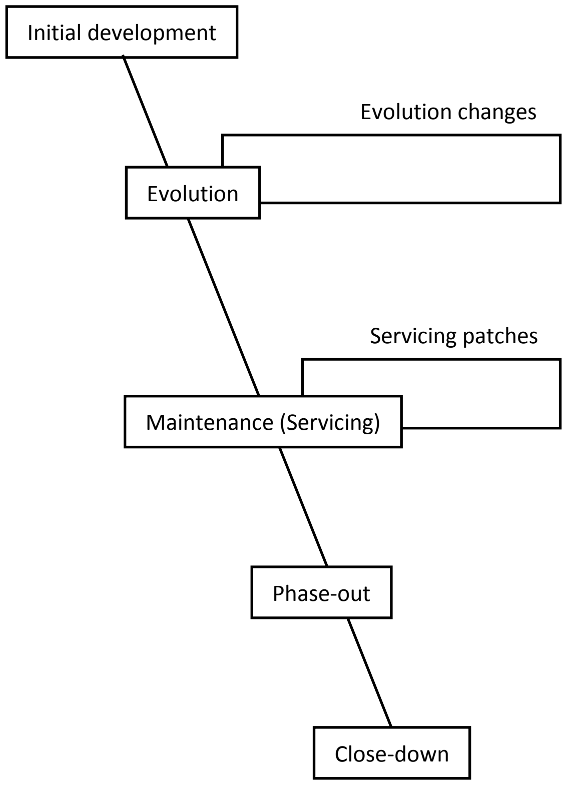

Figure 1. Staged model of software lifespan

The stage that follows is *maintenance* or *servicing*, where the programmers no longer do major changes in the software, but they still make small repairs that keep software usable. Software in the servicing stage has been called "legacy software", "aging software", or "software in maintenance".

When software is not worthy of any further repairs, it enters the *phase-out* stage where it is no longer serviced, although it still may be used. Sometime later, the managers or customers completely withdraw the system from production and this is called *close-down*.

From the point of view of software engineering, software evolution is the central stage of the software lifespan; for typical successful software, an overwhelming amount of time and resources are spent in evolution and hence it merits particular attention of researchers.

2.1 Evolutionary Software Development (ESD)

ESD is a software development practice that moves the bulk of the development from initial development to evolution. Short initial development and early start of evolution is the response to the volatility of requirements, technology, or knowledge, and offers the developers a possibility to react to the volatility in timely manner and offers stakeholders a repeated feedback about the progress of the project.

While knowledge of ESD processes can be traced to the historical beginnings of software development, new and popular ESD processes have emerged in the last two decades and took the central stage of software development. All ESD processes share the fundamental software evolution practices, but differ in other aspects. This section lists some of the most popular ESD processes.

Iterative development is characterized by repeated iterations. Iterations are milestones with specific goals and produce executable – albeit incomplete – versions of software that give stakeholders opportunity to assess the state of the project and plan for the future. Iterative development has been used since the early days of software development although it has gained prominence lately [7].

Agile development is a widely discussed variant of iterative development that is based on very short iterations and autonomous decision makings by developer teams during the iterations. An example of a well-defined and successful agile process is Scrum [8]; the circumstances in which Scrum is effective are discussed in [9]. Another well-known agile process is Extreme Programming [10].

Exploratory development is used when developers do not know in advance details of the features that they are developing, and establish them by trial-and-error. Exploratory programming is very common in research projects [11].

Centralized development is characterized by the presence of the guardians of the code base who either accept or disallow changes to the code base, removing that right from individual developers. Software with very high quality expectations, including avionics software, is an example of such projects [12].

Open source development is a widely discussed variant of centralized development where a wide community of developers can contribute their code to the project, but code owners guard the code base and accept or reject these contributions [13, 14]. *Inner source development* leverages the tools and techniques of the open source development, but it is practiced within a corporation for the development of proprietary software. Successful projects have been conducted in this way [15, 16].

Directed development is another ESD process in which managers assign tasks to the developers. It has been used as a default when other processes do not apply and is applicable to very large or distributed projects [17, 18].

Solo development is the process employed by a single programmer; with the availability of powerful new tools and libraries, this process is becoming more and more common [2, 19].

All these software development processes are variants of software evolution and share the fundamental characteristics, which is repeated change to existing software. Software developers should always choose the most appropriate variant of ESD for their specific project circumstances.

ESD processes increased probability of the success of *small software projects*. The data indicate that small projects (with a budget of less than $1 million) have now an unprecedented success rate of 76%, and only 4% of small projects are cancelled, see p. 3 of [20].

2.2 Future Directions in Staged Model

The successes and popularity of software evolution, including ESD, opened a host of issues that deserve attention of researchers.

2.2.1 Towards Unbundling Practices of ESD

The ESD processes listed in the previous section are all different instances of software evolution that differ from each other by some organizational practices. They are often presented as a fixed bundle of practices and practitioners are expected to accept or reject the whole bundle. However this bundling – while it represents a reasonable current stage in our understanding of software evolution – is becoming an obstacle to further progress. This section presents a different view: It proposes to associate individual practices with specific project circumstances that they address, and based on that, to tailor new processes that exactly fit specific project needs.

We note that each ESD project has a long list of practices. Some of them are obvious and are present in all reasonable projects, while others are a response to specific circumstances which the ESD team must deal with. For the sake of this discussion, we divide the practices into three groups:

1. Practices of code development
2. Practices of developer team organization
3. Practices involving all stakeholders

Examples of practices of the code development include practices of code verification, naming conventions, and so forth.

Practices of developer team organization include practices of the developer team governance [21]. For example, task assignment can be done by managers or it can be done autonomously by self-organizing developer team [22].

Practices involving all stakeholders include planning of iteration goals and their length, decision about release of the product, prioritization of evolution tasks, and so forth.

At the very beginning of a new software development project, an important practice draws a correct boundary between the initial development and evolution. When deciding on this boundary, there are two conflicting considerations: On the one hand, the initial development should resolve the most critical issues and set the project on a good course. To do justice to this criterion requires a large sized initial development. On the other hand, developers need a timely feedback from other stakeholders about the directions of the project and have to respond to the volatility; this requires short initial development. The correct boundary between initial development and evolution reconciles these two contradictory considerations. An extensive discussion of this issue has been done in human factors community where many authors try to combine user centered design with agile development [23]. However this is a general issue of all ESD and it deserves an additional attention of researchers.

Table I presents a sample of project circumstances (in the left column) and corresponding practices that address them (in the right column). It is a very preliminary version; a large research effort is necessary to get a comprehensive and practically useful future table. Some authors did an extensive work on the description of the factors that characterize project circumstances that appear in the left column of Table I and also noted that some circumstances may not have a suitable mitigating practice [24, 25]. Nevertheless, a credible table that extends Table I and takes into account different stages of software lifecycle, may substantially impact ESD and software engineering practice in the future.

Table I. Preliminary table of ESD practices as answers to project circumstances

Project circumstances	**Practice suitable for the circumstances**
Exploratory programming is necessary	Developers are domain experts
Gap between programmer capability and expected quality	Code guardians, permission to commit
Frequent turnover of developers	Concept location practices
High volatility of requirements, technologies, or knowledge	Short initial development, short iterations
Low volatility	Long initial development
. . .	

2.2.2 Problems with Large Software Projects

In contrast to small software projects, there is an industry-wide problem with large software projects, identified in the previously mentioned report [20]. According to this report, large software projects with budget larger than $10 million have low 10% probability of success, and 38% of them were cancelled. These numbers represent large financial losses and are an embarrassment to our community. Note that among the ESD processes mentioned in Section 2.1, there are documented success stories involving large software development projects and involve iterative, directed, open source, inner source, or centralized processes; extraction of the positive experience from these projects and its popularization is a compelling goal for software evolution researchers.

Confusions about terminology may be a contributing factor to this state of the affairs. Unfortunately, many debates about process selection are reduced to a false dichotomy: "waterfall vs. agile". These debates ignore the wide selection of other available ESD processes, and this faulty reasoning appears in both surveys [26] and in industry discussions [27].

The confused debate about the available options may be a contributing factor to the problem with large software development. It is up to the researchers to reframe this community-wide debate, to move it from the over-simplified binary choice "waterfall vs. agile" to a more accurate debate that takes into account availability of all ESD processes of Section 2.1.

Unbundling of practices, discussed in previous section, may offer additional options.

We may consider "waterfall" of these debates as a code word for projects with large initial development and non-existent or very short evolution. "Agile" in this context is a code word for projects with very short initial development (sometimes called "0.th iteration") followed by ESD that uses at least some agile practices. A more informed and nuanced debate would pose the following questions:

- What is the appropriate size of the initial development for this project?
- After initial development, which ESD process or which combination of ESD practices is the most appropriate for further development, given the specific circumstances of this project?

This shift in the debate may better clarify options present in the development process selection, and help to avoid some disasters that are otherwise waiting to happen.

3. PHASED MODEL OF SOFTWARE CHANGE

Another topic discussed in [1] is the software change (SC); the original paper contains a tentative model of SC and makes references to earlier models. SC is the foundation of all evolutionary software processes and its central role in software development is affirmed by survey data [28]. Numerous researchers have dealt with various facets of SC. Several SC models have been proposed, among them "Test Driven Development" [29] or "Legacy code change algorithm" [30]. They are special cases of a more general *phased model of software change (PMSC)* that also incorporates recent research results.

3.1 Phases of Software Change

PMSC is explained in detail in the previously mentioned book [2] and in supporting case studies that affirmed its validity [19, 31, 32]. A schematic model of PMSC is in Figure 2; PMSC enactment may contain all phases of Figure 2 or only a subset. This section contains a brief summary of PMSC phases.

Initiation phase deals with requirements: their elicitation, analysis, prioritization, and so forth. These issues are discussed in [33].

The next phase is *concept location* that finds the module or code snippet that must be changed in order to implement the change request. Once the location of the change is found, *impact analysis* determines the strategy and the full extent of the change. Both concept location and impact analysis require comprehension of the existing program and are discussed in section 3.2.

Actualization implements the new functionality and incorporates it into the old code. Actualization may also require change propagation that makes secondary changes in the old code [2].

Refactoring changes the structure of software without changing its functionality. 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, prefactoring can gather all bits and pieces of the functionality that is going to change and make the actualization localized, so that it affects fewer software modules. The other refactoring phase is called *postfactoring* and it is a clean-up after the actualization. There has been a great progress in refactoring and there are numerous papers and books that have advanced this topic. Summaries of this subfield appear in [34, 35].

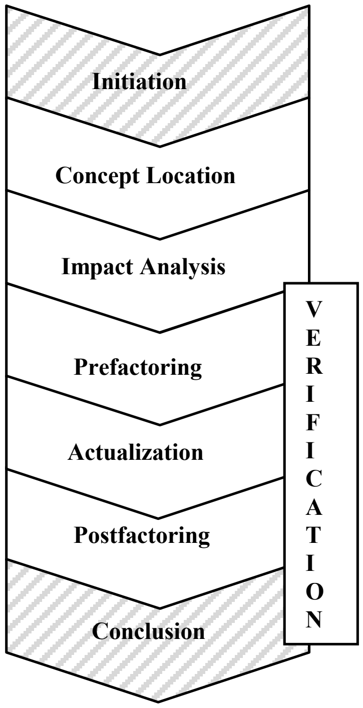

Figure 2. Phased model of software change (PMSC)

Verification guarantees the quality of the work, and it interleaves with phases of prefactoring, actualization, postfactoring, and conclusion. Although no amount of verification can give a complete guarantee of software correctness, numerous bugs and problems can be identified and removed through systematic verification. These issues are discussed in detail in [36].

Conclusion is the last phase of software change; after the new code is completed and verified, the programmers commit it into the version control system repository. Conclusion also creates new baseline, updates documentation, prepares new release, and so forth. A practice of daily build is described in [18], while practice of continuous integration is described in [37].

3.2 "As Needed" Program Comprehension

During evolution, the programmers must comprehend the existing program to be able to add new functionalities or new properties to it. The existing software is sometimes very large and developers employ as-needed approach that concentrates only on that part of the code that they need to understand for their current task. They behave like tourists who visit a large city like Prague, and who also instinctively adopt an as-needed approach: They learn how to reach their hotel and how to get to the historical sites they want to see, but they pay no attention to the rest of the city and all its complexity.

A theory that underlines the as-needed comprehension is adopted from linguistics and mathematics. *Concept* is the fundamental notion of this theory, and following the work of Frege [38] and de Saussure [39], each concept has three attributes: name, intension, and extension, see Figure 3. The name is the label that identifies the concept, intension explains the meaning of the concept, and extensions are all things that the concept describes, including snippets in the code that implement the concept [40]. The six fundamental concept-based comprehension processes are the following [41]:

- location: intension → extension
- recognition: extension → intension
- naming: intension → name
- definition (or account): name → intension
- annotation: extension → name
- traceability: name → extension

These processes play a role in as-needed program comprehension. Concept location is of particular importance as it is a phase in PMSC. It was the progress in the as-needed comprehension that filled this last missing piece of puzzle and allowed creation of the integrated PMSC.

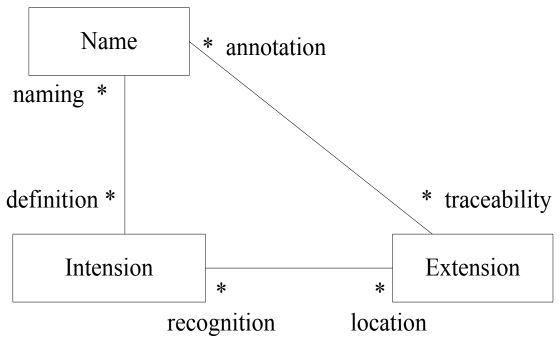

Figure 3. Concept triangle

3.2.1 *Concept Location*

Concept location is the search for concept's extensions in the code. The concepts the developers search for usually originate from change requests.

Concept location can be a small task in small programs, but it can be a considerable task in large or unfamiliar programs. When done in large programs, it resembles internet queries; however, it is done within the code and/or other software work products. Concept location is usually the first step of the software change, and its role is to find the code snippet that implements the extensions of the concepts. The code modifications then start in these concept extensions.

Programmers who know their code perfectly can find the concept extensions immediately, but this approach fails in large or unfamiliar programs. Ideally, in that situation, there should be an external documentation that provides traceability, i.e. guides the programmers from the concept name to the proper place in the code. However, such documentation is very rare. Programmers traditionally use a string pattern matching by "grep" that locates identifiers or comments that indicate the presence of the concept extension in the code.

In spite of its wide use, "grep" has serious limitations. It frequently fails because of the use of synonyms or homonyms that lower precision and recall of the grep queries. The search for better concept location techniques is an important software issue in evolution research. The new concept location techniques broadly fall into static and dynamic categories [42]. Static techniques analyze the software source code or documentation while dynamic techniques analyze results of the program execution. There has been a significant progress in various concept location techniques; for a survey, see [43].

One of the enhancements is a broadening of the concept location beyond a search for a single concept. Concept maps consist of several related concepts, and there is an evidence that concepts that are related in the concept map are likely to be related in the code [44]. Hence locating any related concept will lead to location in the code from which the code modification can start [45].

3.2.2 *Impact Analysis*

Impact analysis expands the results of the concept location and determines the full extent of the change by finding all software components that will be affected by the software change. It is a planning phase that determines the strategy of the change and it is also a part of as-needed program comprehension. However it is methodologically very different from the previously described concept location and it constitutes a separate research topic. For a recent survey, see [46].

One of the relevant practices is *incremental impact analysis* that uses step-by-step navigation of program dependencies. If a component C is scheduled to be modified, the developer inspects all components that interact with C through dependencies and decides which ones are to be modified and which ones are not. The process iterates until all components that interact with modified components have been visited. JRipples is the tool that supports this process [47]. A paper that discusses this topic is [48].

3.3 Future Directions in Software Change

Since software change is such a fundamental task in software engineering, both related practices and supporting tools need continuous improvement.

Concept location presents numerous open issues; some of them are summarized in Section 7 of [43]. Other open issues include investigation of concept location in the presence of external documentation, in particular the annotations [49]. Concept location can be enhanced by a practice of creating and maintaining traceability links that provide a list of concept names and links to the concept extensions in the code, see Figure 3. There have been numerous papers dealing with traceability links; see for example [50, 51]. Also the use of concept maps or domain ontologies in concept location is a promising direction that should receive further attention [52].

Impact analysis is a planning phase that assists developers with planning the rest of the SC. However most of the current research on impact analysis considers only whether modules are impacted or not, and this distinction is often too crude for planning; module modifications range from complete overhaul to minor adjustments and impact analysis should quantify the complexity of the impact, as it can play a role in selecting the strategy for SC.

The incremental impact analysis is based on the navigation through program dependencies; some dependencies propagate the change while others do not. To make this navigation more tractable, heuristics and tools that identify "likely to propagate" dependencies need further attention [47, 53]. Another set of heuristics is necessary for determining when the incremental impact analysis is completed.

In the case studies that we conducted, impact analysis was able to predict prefactoring and actualization but was unable to predict postfactoring. We believe that postfactoring is often done reactively, and postfactoring that cleans up a minor technical debt is often postponed. Technical debt is often resolved only when it accumulates beyond acceptable limit [54]. These factors make prediction of postfactoring difficult. However for a better planning of software changes, this issue needs further studies.

There is also a need for a *seamless software environment* that would support all phases of PMSC and monitor the quality of the resulting code. Currently, typical SC is supported by several technologies. For example, software changes reported in [19] were supported by JRipples, an Eclipse plugin that supports concept location, impact analysis, and change propagation [47]. Mylyn assisted the programmer in managing the workflow and measuring the effort. To record and export timing data required an additional plugin, Tasktop. Unit tests were done by JUnit and functional tests used the Abbot Java GUI test framework. Moreover there were tools measuring size of the code and size of the code changes, and tools measuring test coverage. While using these tools, it became clear that a lack of tool integration was a major hindrance. A unified and seamless software environment that integrates the tools and supports all SC needs is a topic for future work.

4. RESEARCH METHODOLOGIES

Software evolution currently represents a large research agenda and applicable research methodologies have received an increased attention. This section briefly describes the current state and future directions.

Software evolution was discovered empirically by observation of a specific software project [3]. Ever since, empirical work has been the driving factor of the software evolution research.

4.1 Empirical Work

In the research of software evolution, the researchers use a wide spectrum of empirical methodologies [55]. The most common ones are briefly surveyed in this section.

Case studies are observations of a single instance of a phenomenon, or just few instances. The pioneering empirical work that discovered existence of software evolution belongs to the category of case studies [3]. The successful ESD processes were first published as case studies [8, 12, 18]. Original concept location papers were also case studies [56, 57]. Very often, case studies offer the first evidence of important facts about software evolution and open new horizons of research; hence case studies are an indispensable part of the empirical work in software evolution. Guidelines indicating when to use case studies and how to conduct them are summarized in [58].

Controlled experiments are empirical processes that control independent variables and study their effect on dependent variables. Among them, *work product experiments* use independent variables that represent the work products of software evolution, most often the code. These experiments do not involve human subjects and hence they may be easier to conduct, yet still they can reveal important empirical observations. As an example, a study found that an easy-to-compute and scalable "static execute after" relationship can often replace hard-to-compute traditional slicing dependencies, with only a very minor loss of precision [59].

Software repositories are work products that contains a wealth of information about the past software evolution. *Mining software repositories* is a very active and promising research area [60, 61].

An emerging empirical technique available for the study of newly proposed software evolution tools and techniques is *reenactment*. Reenactment is a process by which information about past software changes is extracted from a repository and developers repeat these past changes using new tools or new techniques. The results of a reenactment indicate the effectiveness of these new tools or techniques. In *simulated reenactment*, an algorithm simulates actions of human developers. Simulated reenactment is common in the research of impact analysis [48, 62-64] or concept location [40, 65].

Another class of controlled experiments involves developers and other human subjects [66]. An important consideration in this context is the difference between novices and experts; substantial differences between the two groups have been documented in some contexts [67].

Surveys use a questionnaire that is filled out by the recipients and the results are tabulated by the researchers. There are many surveys that impacted the way we understand software evolution; among the findings that were discovered by surveys is the previously mentioned finding of serious problems with evolution of large software systems [20]. Surveys, of course, can be conducted when the phenomenon that is being studied is already widespread and well-known, and hence they work on the opposite end of the spectrum from case studies.

From this brief overview, it is obvious that there is a wide spectrum of empirical techniques and approaches that are available to a researcher in software evolution and these techniques helped us to discover the existence of software evolution and deepened our understanding of it.

4.2 Reasoning about Evolution

While empirical work discovers facts of software evolution, reasoning integrates these facts into models (theories). Based on these models, researchers formulate hypotheses, predictions, and practical recommendations. If these models (theories) are falsified by new empirically obtained facts, they have to be replaced by better ones which accommodate these new facts [68]. This alternation of empirical observations and creation of models is the foundation of scientific discovery and it also has been applied in the field of software evolution. The models in software evolution have been mostly informal; nevertheless, they have explained many aspects of software evolution. They summarize our current understanding of software evolution and support our reasoning about it.

As an example of an informal model, Fred Brooks identified the essential software difficulties as complexity, interoperability, invisibility, and changeability [69], and used them to correctly predict that the development of better software tools will be slow and difficult. The theory also proved useful in explaining the past paradigm shifts in software engineering, see Chapter 1 of [2]: The first paradigm shift in 1970's was caused by the software complexity and it led towards better understanding of the ways how to deal with it, including composition of software out of simpler parts (modules), information hiding, abstractions, architectures, patterns, and so forth. The second paradigm shift in 2000's was caused by interoperability: Any change in software domain has to be reflected in a corresponding software change, and volatile domains require repeated software change, i.e. software evolution. It moved ESD into the mainstream of software development.

The staged model explained in Section 2 is also a product of the process of knowledge discovery. Sneed's model of software lifespan laid the first foundation [70]. Based on it, Lehner surveyed the lifespan of 13 business systems from Upper Austria and found a clear distinction between what he called software "growth" and "saturation" [71]. Thus, he empirically refuted earlier opinions that the evolution and growth of software continues indefinitely. These empirical observations led to the staged model presented in [1]. Hence the staged model is the result of the process of knowledge discovery that consisted of the following steps:

model → empirical data → new model

Similar process of knowledge discovery led to PMSC of section 3 and other models (theories) that help us to understand software evolution.

4.3 Future Directions in Research Methodologies

4.3.1 Empirical Work

Various empirical techniques offer various levels of confidence and different threats to validity; for that reason, it is often impossible or inappropriate to replace one technique by the others. Full range of the empirical techniques is necessary and attempts to narrow down available choices are not reasonable.

For important observed facts, there should be more effort devoted for replications. The replications can remove some of the threats to validity by using different empirical techniques than the ones used in the original study. For example, a study that used simulated reenactment may be replicated by a controlled experiment that involves human developers, or vice versa. Combinations of different empirical methodologies that have different threats to validity greatly enhance the confidence into empirical results.

4.3.2 Reasoning about Evolution

In spite of the successes of reasoning about evolution, it should be noted that important reasoning happened only episodically. On the other hand, there have been many instances of faulty reasoning, including "waterfall vs. agile" choice discussed in Section 2.2.2. Note that creating practical recommendations based on empirical results always requires reasoning and the goal is to make that reasoning explicit, pragmatic, and sound. Expansion of Table I into a practically usable tool will require not only collection of empirical evidence, but also sound reasoning about that evidence.

The instances of faulty reasoning and their consequences illustrate the urgency of the task to put the reasoning about software evolution on sound footing. Additional empirical work provides only marginal protection against reasoning mistakes; instead, more systematic process in reasoning about software evolution is necessary, in order to improve the process of knowledge discovery. The community should reach a consensus what constitutes acceptable reasoning and it took recent steps towards that goal [72]. A very promising approach advocates to create "small" models geared towards specific problems [73]. This incremental approach has a better probability of success than large all-encompassing models that attempt to explain everything. This paper proposes the following checklist for a good model, based on [68]:

1. Definitions and assumptions that constitute the model
2. Empirical evidence that supports the model (justification)
3. Useful predictions and practical recommendations derived from the model (reasoning)
4. Predictions that are particularly suitable for future empirical exploration and can corroborate or falsify the model (falsifiability)

While presence of this checklist cannot guarantee that the practical recommendations are always correct, it can improve their "batting average" by revealing confusing definitions, insufficient justifications, fallacies in reasoning, or models that lack falsifiability and hence do not represent scientific knowledge.

5. TEACHING SOFTWARE EVOLUTION

Software evolution research reached the maturity level where the fundamental concepts of software evolution can be taught in the undergraduate curriculum. This section summarizes reasons for that and proposed topics to be covered.

Both employers and graduate schools expect computer science graduates to be able to work as developers in software projects. However many academic programs miss this educational goal, as Bjarne Stroustrop observed: "… many graduates have essentially no education or training in software development outside their hobbyist activities. In particular, most see programming as a minimal effort to complete homework and rarely take a broader view that includes systematic testing, maintenance, documentation, and the use of their code by others. ... For many, "programming" has become a strange combination of unprincipled hacking and invoking other people's libraries" [74].

Teaching software evolution builds a bridge between the current academic programs and the expectations placed on the graduates, as overwhelming number of software projects employ some form of ESD. The first software engineering course (1SEC) can be reorganized towards software evolution skills [2, 75]. However the required reorganization may be larger than many instructors suspect, as the gap between the traditional approach and the current expectations is wider than generally assumed.

In order to explain this reorganization, the student learning can be described as a process where students acquire a sequence of incremental skills that build on top of each other. The first skill is the introductory programming that allows the students to develop small programs and gives them confidence in their rudimentary programming abilities. In the current curriculum, this level of competence is achieved in introductory programming courses.

The next step is knowledge of a portfolio of technologies that are used in realistic projects. The technologies include the version control system, GUI and testing frameworks, modelling and monitoring tools, and so forth.

That is followed by learning PMSC basics that is the foundation of all ESD processes. PMSC is the main objective of the revised 1SEC and the missing link in many current curricula. 1SEC structured like this opens the door to learning advanced practices of various ESD processes and the rest of software engineering discipline. The semester schedule of 1SEC as taught at Wayne State University is in Table II. For more detail, see [2, 75, 76].

Learning PMSC requires practice on realistic projects that represent current software development and at the same time are suitable for beginning developers. At Wayne State University, we use open source projects that are evolved in a lab that runs in parallel with the lectures. Students are divided into teams and each team is assigned a specific open source project. Each student in the team is given different change requests and there are deadlines by which the students commit their code. The instructor and a teaching assistant manage the process: They assign the change requests to the individual students, help them to solve the conflicts, and help to create new baselines after each deadline.

The required code changes are of increasing complexity. The first change typically requires only concept location and a minimal code modification that does not propagate to other classes. Second change already propagates to several classes and may involve all phases of PMSC. The third change usually involves conflict with other team members that the students must resolve. During the semester, a student typically writes approximately 300 lines of code that has to fit into the existing project code and has to be thoroughly tested, resulting in expected code quality [76]. Our lab schedule within a semester is in Table III.

Table II. Semester schedule of the lectures

Week	Topics
1	syllabus, history of software engineering
2	software life span models, technologies
3	software change initiation
4	concept location
5	impact analysis
6	actualization
7	refactoring
8	verification
9	verification, cont.
10	software change conclusion, software processes introduction
11	evolutionary software development
12	initial development
13	final stages of software lifespan
14	professional ethics

Table III. Semester schedule of the lab

Week	Topics
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 1 due + team presentation
7	refactoring - in class exercise
8	groups meetings + Q&A about the project
9	change 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 3 due + team presentation
14	extra credit due

While the grade in the lectures is based on the exams, the project student grades are based on student's code commits and the corresponding reports that the students are asked to prepare. Through analysis of the version control system data, the student reports, and interviews whenever necessary, the instructor can assess the individual responsibility and identify those team members who caused problems; hence there is both a team collaboration and an individual accountability in the teams.

5.1 Future Directions in Teaching

Integrated software environment that supports PMSC was mentioned in Section 3.3 and it would enhance not only practice of ESD, but also enhance teaching as it would guide the students when they are implementing software changes. It could also help with the monitoring and grading. A step towards that is JRipples that supports concept location by dependency search, impact analysis, and change propagation [47], but support for additional phases and additional tools that provide feedback are also desired.

Another direction is to develop courses that build on 1SEC and teach more advanced topics, for example other roles in various ESD processes, including testers, managers, and so forth [77].

PMSC and its individual phases are still being investigated and additional experience is being accumulated. The new tools and techniques that improve productivity or quality of software change should be incorporated into 1SEC in the future, as they become available.

6. SOFTWARE MAINTENANCE

Software enters maintenance stage when it is transferred to users. According to standard ISO/IEC 1207 [78], "The purpose of the Software Maintenance Process is to provide cost-effective support to a delivered software product." While many authors call this stage software *maintenance*, others call it software *servicing* [1, 79].

Like software evolution, maintenance also consists of repeated changes to software, but the objectives are drastically reduced: The only goal is to keep software usable in a cost-effective way, and there is no longer any aspiration to add new features or functionalities. The changes that the developers undertake in this stage are mostly corrective (bug fixes) or adaptive (adapting to changes in technology or use).

Since servicing aims at scaled down objectives, the servicing processes are usually simpler and more predictable than the processes of evolution. As examples, Software Maintenance Maturity Model was proposed in [80], and a variant of agile process suitable for software maintenance appeared in [81].

Many issues specific to software maintenance originate from the fact that software is in use while being maintained [82, 83], and that a different team than the team of original developers may maintain software [84, 85]. During maintenance, it is very common that there are at least two versions of software in existence: One is the version that is in production and is maintained (serviced), while in parallel the developer team evolves the another version for the future releases; this situation is described by *versioned model of software life span* [1, 2].

An important issue is the *end of software evolution.* Software evolution is an expensive process and managers may decide that they no longer want to invest in it. Then it ends by *management* *decision* while maintenance may continue as long as software is used.

Stabilization is the situation where software evolution is no longer necessary. The earlier mentioned reasons for software evolution are the volatility of requirements, technology, or stakeholder knowledge. Once the volatility stops, the software project reaches stabilization.

Stabilization frequently occurs in embedded systems that control a specific mechanism: Once the mechanism is produced, it does not change and therefore the software requirements are stable. The system is equipped with a specific chip and specific operating system and this technology is also stable. The only reason for evolution is the volatility of stakeholder knowledge, where the stakeholders are learning how to control this mechanism. Once the stakeholders complete their learning and find the solution, there is no reason for further evolution and the software reaches the point of stabilization. However the servicing still may be required.

Code decay is a reason that may cause involuntary end of evolution. Although software is not material and is not subject to wear and tear, its structure decays under the impact of software changes [86, 87]. The symptoms of code decay are increased difficulty of making software changes and decreased code quality, particularly an increase in the presence of bugs introduced by the changes. The situation may reach the point where further evolution is beyond the capabilities of the programming team.

Insufficient developer knowledge is one of the reasons for code decay. To understand a software system, developers must understand the domain of the application, the architecture, the algorithms, the data structures, the concept extensions in the code, and so forth. If that knowledge is not available, the new code they produce does not mesh with the old code and software changes aggravate code decay. Even refactoring requires a thorough knowledge of software and without it, it is no longer possible and the system further decays. As the symptoms of decay proliferate, the code becomes more and more complicated and the knowledge that is necessary for further evolution actually increases. When the gap between the knowledge of the team and the growing complexity of software becomes too large, the software evolvability is lost.

A special instance of loss of knowledge is *cultural shift*. Software engineering has more than half a century of history and there are programs still in use that were created a half century ago. These programs were created in a time of completely different properties of hardware; computers were slower and had much less memory, often requiring elaborate algorithms to deal with these limitations. Moreover the programmers wrote in obsolete languages and for obsolete operating systems, and held different opinions about what constitutes a good structure of software. The current programmers who try to change these old programs face a double problem: not only do they have to recover the knowledge that is necessary for that specific program, but they also have to recover the culture within which these programs were created. Without that, they may be unable to make the evolutionary changes in the program [88].

6.1 Future Directions in Software Maintenance

End of software evolution is a very important event and as such, it deserves continuous study. Accidental end through code decay or software aging is a particularly serious issue; techniques that avoid decay and keep software evolvable deserve continuous research attention. A complementary approach is the research of techniques and tools that allow addition of substantial new functionality to a decayed code, and through that to extend software evolution to the code that is currently considered to be non-evolvable.

7. CONCLUSIONS

Software evolution has moved from an unexpected and peripheral phenomenon into the center of software development and center of software engineering. As such, it gained enormous attention of both developers and researchers. This paper covers several themes that are likely to have a significant impact in the future software evolution and future software development.

It discusses processes of evolutionary software development that include agile, iterative, open source, inner source, and other processes, and advocates investigation of the individual practices within these processes, so that additional processes, more fitting to specific project circumstances, can be created. The paper also discusses the basic task of software evolution, software change, and presents phased model of software change (PMSC) together with directions how it can be enhanced.

It lists research methodologies that have been used in investigation of software evolution, and besides empirical work, it proposes to pay attention to model creation and reasoning, so that the process of knowledge discovery is more complete.

It reiterates that software evolution can be taught in undergraduate curriculum and through it, the gap between the curriculum and the practice can be substantially narrowed. Finally it briefly contrasts software evolution with software maintenance.

8. REFERENCES

[1] Bennett, K. H. and Rajlich, V. T. 2000. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering,* pp. 73-87.

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

[3] Belady, L. A. and Lehman, M. M. 1976. A model of large program development. *IBM Systems Journal*, vol. 15, pp. 225-252.

[4] Godfrey, M. W. and German, D. M. 2008. The past, present, and future of software evolution. In Proceedings of the *Frontiers of Software Maintenance*, 2008. pp. 129-138.

[5] Mens, T. and Demeyer, S. 2008. *Software evolution*. Springer.

[6] Rajlich, V. and Bennett, K. H. 2000. A Staged Model for the Software Life Cycle. *IEEE Computer*, vol. 33, pp. 66-71.

[7] Larman, C. and Basili, V. R. 2003. Iterative and Incremental Development: A Brief History, *IEEE Computer*, vol. 36, pp. 47 - 56.

[8] Schwaber, K. and Beedle, M. 2002. *Agile Software Development with Scrum.* Upper Saddle River, NJ: Prentice Hall.

[9] Rising, L. and Janoff, N. S. 2000. The Scrum software development process for small teams. *IEEE Software*, vol. 17, pp. 26-32.

[10] Beck, K. 2000. *Extreme Programming Explained*. Reading, MA: Addison Wesley.

[11] Rajlich, V. and Hua, J. 2013. Which Practices are Suitable for an Academic Software Project? In *Proceedings of the IEEE International Conference on Software Maintenace (ICSM)*, pp. 440 - 443.

[12] Madden, W. A. and Rone, K. Y. 1984. Design, development, integration: space shuttle primary flight software system, *Commun. ACM*, vol. 27, pp. 914-925.

[13] Weber, S. 2004. *The success of open source*. Cambridge University Press.

[14] Feller, J. and Fitzgerald, B. 2002. *Understanding open source software development*: Addison-Wesley London.

[15] Gurbani, V. K., Garvert, A., and Herbsleb, J. D. 2006. A case study of a corporate open source development model. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 472-481.

[16] Lindman, J., Rossi, M., and Marttiin, P. 2008. Applying open source development practices inside a company. In *Open Source Development, Communities and Quality*, Springer, pp. 381-387.

[17] Humphrey, W. S. 1999. *Introduction to the team software process* Addison-Wesley Professional.

[18] Cusumano, M. A. and Selby, R. W. 1997. How Microsoft builds software. *Communications of the ACM*, vol. 40, pp. 53-61.

[19] Dorman, C. and Rajlich, V. 2012. Software Change in the Solo Iterative Process: An Experience Report. .In *Proceedings of the Agile (AGILE)*, IEEE Computer Society, pp. 22-30.

[20] 2013. *Chaos Manifesto 2013*. Available at: http://versionone.com/assets/img/files/ChaosManifesto2013.pdf, accessed on

[21] Shah, S. K. 2006. Motivation, governance, and the viability of hybrid forms in open source software development. *Management Science*, vol. 52, pp. 1000-1014.

[22] Friebel, G. and Schnedler, W. 2011. Team governance: Empowerment or hierarchical control. J*ournal of Economic Behavior & Organization*, vol. 78, pp. 1-13.

[23] Silva da Silva, T., Martin, A., Maurer, F., and Silveira, M. 2011. User-centered design and Agile methods: a systematic review. *In Proceedings of the Agile Conference (AGILE)*, IEEE Computer Society, pp. 77-86.

[24] McLeod, L. and MacDonell, S. G. 2011. Factors that affect software systems development project outcomes: A survey of research. *ACM Computing Surveys*, vol. 43.

[25] Clarke, P. and O'Connor, R. V. 2012. The situational factors that affect the software development process: Towards a comprehensive reference framework. *Information and Software Technology*, vol. 54, pp. 433-447.

[26] 2009. Survey Finds Majority of Senior Software Business Leaders See Rise in Development Budgets. Available at: http://www.softserveinc.com/news/survey-senior-software-business-leaders-rise-development-budgets/, accessed on 3/12/2014

[27] Hu, E. 2013. This Slide Shows Why HealthCare.gov Wouldn't Work At Launch. Available at: http://www.npr.org/blogs/alltechconsidered/2013/11/19/246132770/this-slide-shows-why-healthcare-gov-wouldnt-work-at-launch, accessed on 3/12/2014

[28] Carroll, P. B. 1988.Computer glitch: Patching up software occupies programmers and disables systems. *Wall Street Journal*, vol. 118, p. 1.

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

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

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

[32] Febbraro, N., Rajlich, V. 2007. The Role of Incremental Change in Agile Software Processes. In *Proceedings of the Agile (AGILE)*, IEEE Computer Society, pp. 92-103.

[33] Nuseibeh, B. 2014. Requirements Engineering Travelogue. *In Proceedings of the IEEE International Conference on Software Engineering (ICSE), Future of Software Engineering*.

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

[35] Mens, T. and Tourwé, T. 2004. A survey of software refactoring," *Software Engineering, IEEE Transactions on*, vol. 30, pp. 126-139.

[36] Orso, A. and Rothermel, G. 2014. Software Testing: A Research Travelogue (2000 – 2014). In *Proceedings of the IEEE International conference on Software Engineering (ICSE), Future of Software Engineering*.

[37] Duvall, P. M., Matyas, S., and Glover, A. 2007. *Continuous integration: improving software quality and reducing risk*: Pearson Education.

[38] Frege, G. 1964. *The basic laws of arithmetic* University of California Press.

[39] De Saussure, F. 2006. *Writings in general linguistics* Oxford University Press.

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

[41] Rajlich, V. 2009. Intensions are a key to program comprehension.In *Proceedings of the IEEE International Conference on Program Comprehension (ICPC)*, pp. 1-9.

[42] Wilde, N., Buckellew, M., Page, H., Rajlich, V., and Pounds, L. 2003. A Comparison of Methods for Locating Features in Legacy Software. *Journal of Systems and Software*, vol. 65, pp. 105-114.

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

[44] Ratiu, D. and Deissenboeck, F. 2007. From reality to programs and (not quite) back again. In *Proceedings of the IEEE International Conference on Program Comprehension (ICPC)*, pp. 91-102.

[45] Wilson, L. A., Petrenko, M., and Rajlich, V. 2012. Using Concept Maps to Assist Program Comprehension and Concept Location: An Empirical Study. *Journal of Information & Knowledge Management*, vol. 11.

[46] Li, B., Sun, Z., Leung, H., and Zhang, S. 2012. A survey of code-based change impact analysis techniques. S*oftw. Test. Verif. Reliab*. Vol. 23(8), pp. 613-646

[47] Petrenko, M. 2011. JRipples. Available at: http://jripples.sourceforge.net/

[48] Hassan, A. E. and Holt, R. C. 2006. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Software Engineering*, vol. 11, pp. 335 - 367.

[49] Rajlich, V. 2000. Incremental Redocumentation Using the Web. *IEEE Software*, September/October, pp. 102-106.

[50] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. 2002. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, vol. 28, pp. 970-983.

[51] Marcus, A. and Maletic, J. I. 2003. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing.In *Proceedings of the International Conference on Software Engineering (ICSE)*, Portland, Oregon, USA, pp. 125-135.

[52] Petrenko, M., Rajlich, V., and Vanciu, R. 2008. Partial Domain Comprehension in Software Evolution and Maintenance. In *Proceedings of the IEEE International Conference on Software Comprehension (ICPC)*, pp.13 - 22.

[53] Petrenko, M.. 2009. *On use of dependency and semantics information in incremental change*. PhD thesis. Department of Computer Science, Wayne State University.

[54] Murphy-Hill, E. and Black, A. P. 2008. Refactoring tools: Fitness for purpose. *IEEE Software*, vol. 25, pp. 38-44.

[55] Wohlin, C., Höst, M., and Henningsson, K. 2003. Empirical research methods in software engineering. in *Empirical Methods and Studies in Software Engineering*, Springer, pp. 7-23.

[56] Wilde, N., Gust, T., Gomez, J. A., and Strasburg, D. 1992. Locating User Functionality in Old Code. In *Proceedings of the IEEE Conference on Software Maintenance (CSM)*, pp. 200-205.

[57] Chen, K. and Rajlich, V. 2000. Case Study of Feature Location Using Dependency Graph. In *Proceedings of the IEEE International Workshop on Program Comprehension (IWPC)*, pp. 241-249.

[58] Runeson, P. and Höst, M. 2009. Guidelines for conducting and reporting case study research in software engineering, *Empirical Software Engineering*, vol. 14, pp. 131-164.

[59] Jász, J., Beszédes, Á., Gyimóthy, T., and Rajlich, V. 2008. Static execute after/before as a replacement of traditional software dependencies. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pp. 137-146.

[60] Kagdi, H., Collard, M., and Maletic, J. 2007. 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.

[61] Demeyer, S., Murgia, A., Wyckmans, K., and Lamkanfi, A. 2013. Happy birthday! a trend analysis on past MSR papers. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, pp. 353-362.

[62] Petrenko, M. and Rajlich, V. 2009. Variable Granularity for Improving Precision of Impact Analysis. In *Proceedings of the IEEE International Conference on Program Comprehension (ICPC)*, pp. 10-19.

[63] Canfora, G. and Cerulo, L. 2005. Impact Analysis by Mining Software and Change Request Repositories. In *Proceedings of the IEEE International Symposium on Software Metrics*, pp. 9-29.

[64] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A. 2000. Identifying the Starting Impact Set of a Maintenance Request: a Case Study. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 227-230.

[65] Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., and Rajlich, V. 2007. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering*, vol. 33, pp. 420-432.

[66] Kitchenham, B. A., et al. 2002. Preliminary guidelines for empirical research in software engineering, *IEEE Transactions on Software Engineering*, vol. 28, pp. 721-734.

[67] Shah, H. B., Gorg, C., and Harrold, M. J. 2010. Understanding exception handling: Viewpoints of novices and experts. *Software Engineering, IEEE Transactions on*, vol. 36, pp. 150-161.

[68] Popper, K. R. 2002. *The logic of scientific discovery*. London and New York: Routledge.

[69] Brooks Jr, F. P. 1987. No silver bullet - essence and accidents of software engineering, *IEEE Computer*, vol. 20, pp. 10-19.

[70] Sneed, H. M. 1989.*Software engineering management*: Halsted Press.

[71] Lehner, F. 1991. Software life cycle management based on a phase distinction method. *Microprocessing and Microprogramming*, vol. 32, pp. 603-608.

[72] Johnson, P., Jacobson, I., Goedicke, M., and Kajko-Mattsson, M. 2013. 2nd SEMAT workshop on a general theory of software engineering. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 1525-1526.

[73] Stol, K. J. and Fitzgerald, B. 2013. Uncovering Theories in Software Engineering. In *Proceedings of the General Theories of Software Engineering* (GTSE 2013), pp. 5 - 14.

[74] Stroustrup, B. 2010. What Should We Teach New Software Developers? Why? *Communications of the ACM*, vol. 53, pp. 40-42.

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

[76] Rajlich, V. 2013. Software Engineering: The Current Practice", Lab manual. Available at:

http://www.cs.wayne.edu/rajlich/ProjectAndLabs/index.html, accessed on 3/12/2014

[77] Shaw, M. 2000. Software engineering education: a roadmap.In *Proceedings of the Conference on The Future of Software Engineering*, pp. 371-380.

[78] 2008. "ISO/IEC/IEEE Standard for Systems and Software Engineering - Software Life Cycle Processes". Available at: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4475822, accessed on 1/06.

[79] Bugde, S., Nagappan, N., Rajamani, S., and Ramalingam, G. 2008. Global software servicing: Observational experiences at microsoft. In *Proceedings of the IEEE International Conference on Global Software Engineering, (ICGSE*). pp. 182-191.

[80] April, A., Huffman Hayes, J., Abran, A., and Dumke, R. 2005. Software Maintenance Maturity Model (SMmm): the software maintenance process model. *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, pp. 197-223.

[81] Pino, F. J., Ruiz, F., García, F., and Piattini, M. 2012. A software maintenance methodology for small organizations: Agile_MANTEMA, *Journal of Software: Evolution and Process*, vol. 24, pp. 851-876.

[82] Xiong, C. J., Xie, M., and Ng, S. H. 2011. Optimal software maintenance policy considering unavailable time. *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, pp. 21-33.

[83] Gkantsidis, C., Karagiannis, T., and Vojnovic, M. 2006.Planet scale software updates. *ACM SIGCOMM Computer Communication Review*, vol. 36, pp. 423-434.

[84] Khan, A. S. and Kajko-Mattsson, M. 2010. Taxonomy of handover activities. In *Proceedings of the International Conference on Product Focused Software*, pp. 131-134.

[85] Pigoski, T. M. 1996. *Practical software maintenance: best practices for managing your software investment*. John Wiley & Sons, Inc.

[86] Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., and Mockus, A. 2001. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Transactions on Software Engineering*, vol. 27, pp. 1-12.

[87] Parnas, D. L. 1994. Software aging.*In Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 279-287.

[88] Rajlich, V., Wilde, N. B., M., and Page, H. 2001. Software Cultures and Evolution. *IEEE Computer*, pp. 24-29.