Software Evolution

Andy Zaidman, Martin Pinzger, Arie van Deursen

Software Engineering Research Group
Delft University of Technology
The Netherlands
{a.e.zaidman, m.pinzger, arie.vandeursen}@tudelft.nl

Abstract

Software evolution is the term used in software engineering to refer to the process of developing an initial version of the software and then repeatedly updating it to satisfy the user's needs. Software evolution is an inevitable activity, as useful and successful software stimulates users to request new and improved features. However, evolving a software system is typically difficult and costly. In this chapter, we provide an historical overview of the field and survey four important research ares: program comprehension, reverse engineering, reengineering, and software repository mining. We report on key approaches, results, and indicate a number of challenges open to on-going and future research in software evolution.

Keywords: software evolution, software reengineering, software reverse engineering, software repository mining, program comprehension, static analysis, dynamic analysis

1 Introduction

In a recent advertisement aimed at recruiting new software engineers, ING, one of the largest European banks, summarized some of its key information technology figures[1]. ING serves over 85 million customers, conducts 16 million financial transactions each day, making use of 75,000 computers running over 3000 different applications, storing 10 petabytes of data. Furthermore, the underlying software systems are not static, but subject to continuous evolution: ING indicated it was involved in 1449 *change projects*.

[1] Automatisering Gids, nr. 42, 16 oktober 2009.

The situation at ING illustrates what is common in many software-intensive organizations: software systems are business critical and are required to run continuously, often on a 7×24 hour basis. The systems are complex in nature, and should operate under hard to meet performance and scaleability criteria. Under these constraints, engineers are required to adjust the systems to new business opportunities, emerging technologies, law changes, and so on. These changes are to be made in a cost-effective manner, without loss of quality of the existing functionality.

Making changes to existing software systems is one of the key software engineernig challenges, and covered by the field of *software evolution*. We provide a historical overview of this field, and survey four important research areas in software evolution:

- *Program comprehension*, addressing the difficulties people have in understanding the structure of complex software systems;
- *Reverse engineering*, comprising methods and techniques to distill abstract information from existing systems;
- *Reengineering*, offering tools and techniques to restructure existing software systems; and
- *Software repository mining*, involving the study of historical data collected in bug tracking systems, revision control systems, mailing lists, etc. in order to increase our understanding of the underlying systems.

These topics are addressed in Section 3–6, after which we conclude with pointers to research venues and avenues for future research.

2 Historical Perspective

In the 1960's awareness grew that manufacturing software should be less *ad hoc*, and instead should be based on theoretical foundations and practical disciplines. This awareness culminated in the organization of the first software engineering conference in 1967 by the NATO Science Committee [28].

In 1970 then, inspired by established engineering disciplines, Royce proposed the *waterfall life-cycle* process for software development [38]. Of particular importance in this model was the definition of the *maintenance phase* for software systems, which was considered the final phase of the software life-cycle and which happened after its deployment. The IEEE 1219 standard defines software maintenance as: "the modification of a software product *after delivery* to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."

It took a while before software engineers became aware of the inherent limitations of this software process model, namely the fact that the separation in phases was too strict and inflexible, and that it is often unrealistic to assume that the requirements are known before starting the software design phase. In the late seventies, a first attempt was made towards a more evolutionary process model, the so-called "change mini-cycle" as proposed by Yau *et al.* [41].

Also in the seventies, Manny Lehman started to formulate his *laws of software evolution*, see Table 1. The postulated laws were based on earlier work carried out by Lehman to understand the change process being applied to IBM's OS 360 operating system. His original findings were confirmed in later studies involving other software systems [25]. This was probably the first time that the term *software evolution* was explicitly used to stress the difference with the post-deployment activity of software maintenance.

Nevertheless, it took until the nineties until the term software evolution gained widespread acceptance. Also around this time *evolutionary processes* such as Boehm's *spiral model* gained acceptance. In that same category, Bennet and Rajlich's *staged model* explicitly takes into account the inevitable problem of *software aging* [36]. After initial development of a first running version, the "evolution stage" allows for any kind of modification to the software, as long as the architectural integrity remains preserved. If this is no longer the case, there is a loss of evolvability and the "servicing stage" starts. During this stage, only small patches can be applied to keep the software up and running.

Software evolution is also a crucial ingredient of so-called *agile software development*, of which *extreme programming (XP)* [2] is probably the most famous proponent. In brief, agile software development is a lightweight iterative and incremental (evolutionary) approach to software development. It takes into account that software is created in a highly collaborative manner and explicitly accommodates the changing needs of its stakeholders, even late in the development cycle.

Nowadays, software evolution has become a very active and well-respected field of research in software engineering, and the terms *software evolution* and *software maintenance* are often used as synonyms. The fact that the software evolution research area is so active, can in part be explained through the fact that software systems have a prolonged lifetime: some of the early systems written in the 1960's and 1970's are currently still in use. These so-called *legacy systems* are still crucial to the business environment and simply replacing them might involve high risk and high costs. Still, these systems must also evolve in order to stay useful in today's operating context. That is why sub-fields such as program comprehension, reverse engineering, mining software repositories, testing, impact analysis, cost estimation, software configuration management and re-engineering are so important to understand the software and enable the continued evolution of these and more modern software systems.

3 Program Comprehension

Having a sufficient understanding of the software system is a necessary prerequisite to be able to successfully accomplish many software engineering activities, including software evolution and software re-engineering related tasks.

Definition Program comprehension is *the task of building mental models of an underlying software system at various levels of abstraction, ranging from models of the code itself to ones of the underlying application domain* [33].

Being aware of the fact that almost all software evolution activities require understanding of the software system, the link between software evolution and program understanding becomes immediately clear. Unknown perhaps is the fact that building up this knowledge can take up to 60% of the time allocated for a particular task, making program comprehension a costly necessity [10].

Von Mayrhauser and Vans have done research on how software engineers go about their comprehension process and they have identified three distinct strategies, namely: a top-down model, a bottom-up model or a mix of the previous two, the so-called integrated model [39].

Top-down understanding typically applies when the code, problem domain and/or solution space are familiar to the software engineer. Because of these similarities, the software engineer typically forms a number of hypotheses about the structure of the system. Subsequently, hypotheses

1	Continuing Change	Systems must be continually adapted else they become progressively less satisfactory.
2	Increasing Complexity	As a system evolves its complexity increases unless work is done to maintain or reduce it.
3	Self Regulation	System evolution processes are self regulating with distribution of product and process measures close to normal.
4	Conservation of Organizational Stability	Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an evolving system tends to remain constant over product lifetime.
5	Conservation of Familiarity	As a system evolves, all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
6	Continuing Growth	The functional capability of systems must be continually increased to maintain user satisfaction over the system lifetime.
7	Declining Quality	Unless rigorously adapted to take into account for changes in the operational environment, the quality of a system will appear to be declining.
8	Feedback System	Evolution processes are multi-level, multi-loop, multi-agent feedback systems.

Table 1. Laws of Software Evolution

are iteratively refined, passing several levels until they can be matched to specific code in the program.

Pennington found that the bottom-up program comprehension model is often used when the code and/or problem domain are not familiar to the software engineer [37]. This way of understanding starts at the code level and while identifying elementary blocks of source code in the program, microstructures are chunked together to form macrostructures and macrostructures are linked to each other via cross-referencing. Another bottom-up approach that can be employed is the so-called situation model, which concentrates on a dataflow/functional abstraction instead of relying on control-flow, as described earlier.

Finally, the integrated model for program comprehension involves top-down and bottom-up comprehension, and also a knowledge base. The knowledge base, which typically is the human mind, stores (1) any new information that is obtained directly from the application through either of the two program comprehension strategies or (2) information that is inferred. In practice, the integrated model is frequently used when trying to understand large-scale systems, largely because software engineers are typically familiar with certain parts of the source code, while they are less familiar with other parts.

4 Reverse Engineering

Usually, the system's maintainers are not the software engineers that originally designed the system. Thus, before making changes to the software they must first build up sufficient knowledge of the software system at hand. In Section 3 on program comprehension, we have seen that this process can take up to 60% of the allocated time. It is in this context that reverse engineering tools can play an important role as they can facilitate the program comprehension process.

In their seminal paper, Chikofsky and Cross define reverse engineering as follows [9]: *"reverse engineering is the process of analyzing a subject system to identify the systems components and their interrelationships and create representations of the system in another form or at a higher level of abstraction."* Reverse engineering has been traditionally viewed as a two step process: information extraction and abstraction. Information extraction analyses the subject system artifacts to gather raw data, whereas abstraction creates user-oriented documents and views. The primary purpose of reverse engineering a software system is to increase the overall comprehensibility of the system for both maintenance and new development. In particular, reverse engineering provides ways to [9]:

Cope with complexity. It allows to better deal with the shear volume and complexity of systems, by (automatically) abstracting to a higher level.

Generate alternate views. Tools facilitate the (re)generation of graphical representations. In particular, these tools often allow to generate alternate views, thereby offering the chance to study the system from a different perspective.

Recover lost information. In continuously evolving systems, modifications are frequently not reflected in documentation. In this context, reverse engineering allows to recover designs.

Detect side effects. Comparing the initial designs with the current designs as obtained from reverse engineering tools allows to spot deviations from the original design plans.

Synthesize higher abstractions. Reverse engineering tools frequently create alternate views at higher levels of abstraction.

Facilitate reuse. Reverse engineering can help detect candidates for reusable software components from software systems.

Reverse engineering techniques can be classified according to the artifacts that they analyze: *static analysis* extracts properties from software systems through the analysis of source code, documentation, architectural diagrams, or de-

sign information. *Dynamic analysis* meanwhile analyses data gathered from a running programming and thus studies the actual behavior of the software. *Hybrid approaches* then combine static and dynamic analysis. We will now discuss static and dynamic analysis in more depth.

4.1 Static Analysis

Static analysis, or the analysis of source code, documentation, architectural diagrams, or design information, has been successfully applied in a number of areas. Key challenges in static analysis-based reverse engineering are to abstract low-level information, e.g., source code, into more manageable and easier to understand higher-level abstractions, e.g, control-flow diagrams. Apart from abstracting, another challenge is to establish links between different artifacts, e.g., establish links between existing documentation or requirements and the source code.

Low-level examples of static analysis are the generation of control-flow diagrams and performing data-flow analysis in order to better understand software systems in the small. Passing over program dependence graphs and techniques like *slicing* [40], which enable program comprehension in the large, there are also many other static analysis techniques which have successfully been applied. Some examples of problem areas where reverse engineering with static analysis has been successfully applied are [7]: redocumenting programs and relational databases, identifying reusable assets, recovering architectures, recovering design patterns, building traceability links between code and documentation, identifying code clones, code smells and aspects, performing impact analysis, and many more.

4.2 Dynamic Analysis

Dynamic analysis, or the analysis of data gathered from a running program, has the potential to provide an accurate picture of a software system because it exposes the systems actual behavior. Among the benefits over static analysis are the availability of runtime information and, in the context of object-oriented software, the exposure of object identities and the actual resolution of late binding. Drawbacks are that dynamic analysis can only provide a partial picture of the system, i.e., the results obtained are valid for the scenarios that were exercised during the analysis, and that dynamic analysis typically involves the collection and analysis of large amounts of data, often introducing scalability issues with tools when analyzing large software systems.

Cornelissen *et al.* have performed a survey of dynamic analysis techniques for program understanding purposes [12]. From this broad overview, we see that research in this area revolves around creating ultra-scalable visualizations, e.g., Extravis [11] (see Figure 1), and coming up

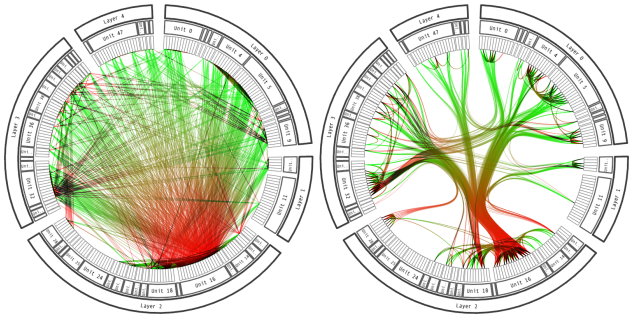

Figure 1. Extravis, an example of visualization of dynamic analysis [11].

with sensible abstraction mechanisms for trace data in order to overcome scalability issues [43]. Research in this area has resulted in tools that allow to do feature localization, bug localization, etc.

4.3 Tools

A number of reverse engineering tools have been produced by the research community. Two of these tools have become real frameworks enabling a wide range of static and/or dynamic analyses for reverse engineering. These tools are Moose[2], which is a platform for software analysis originally developed at the University of Berne, Switzerland, and Rigi[3], an interactive, visual tool to understand and re-document software, developed at the University of Victoria, BC, Canada.

5 Reengineering

From the *laws of software evolution* that we discussed in Section 2 we intuitively understand that systems must continuously be adapted to meet changing requirements from its users (law 1) and that we should take preventive actions to reduce the increasing complexity which is the result of successive adaptations of the software (law 2). Reengineering is the term coined for the process of reorganizing and modifying an existing software system with the aim of making the system easier to maintain. Chikofsky and Cross define reengineering as follows [9]: *"The examination and alteration of a system to reconstitute it in a new form"*. Please note that this definition implies that *reverse engineering* — the examination part — is often part of the reengineering cycle. In the following, we introduce legacy software systems and subsequently ways to measure and resolve typical shortcomings with them.

[2] http://www.moosetechnology.org/
[3] http://www.rigi.csc.uvic.ca/

5.1 Legacy Software Systems

There is a clear connection between *reengineering* and *legacy software systems*. Often, reengineering is the most cost-effective option for an organization to extend the life of their software systems. Legacy software is software that is still very much useful to an organization – quite often even indispensable – but the evolution of which becomes a great burden [3]. Brodie and Stonebraker give an apt description of a legacy system [5]: "Any information system that significantly resists modification and evolution to meet new and constantly changing business requirements." Note that this definition implies that age is no criterion when considering whether a system is a legacy system [14], implying that even relatively new systems can be considered legacy systems if they are of high-value to the organization and resist evolution.

Legacy software is omni-present: think of the large software systems that were designed and first put to use in the 1960s or 1970s; these software systems are nowadays often the backbone of large multinational corporations. For banks, healthcare institutions, etc. these systems are vital for their daily operations. As such, failure of these software systems is not an option and that is why these trusted "oldtimers" are still cared for every day. Furthermore, they are still being evolved to keep up with the current and future business requirements. This is where reengineering approaches can be useful: reengineering allows to make the future evolution of these legacy systems easier.

5.2 Software Metrics

Considering the quality of a software system requires to take a multidimensional viewpoint. Indeed, the ISO 9126 standard identifies six key quality attributes for computer software: (1) functionality, (2) reliability, (3) usability, (4) efficiency, (5) maintainability, and (6) portability. From a software evolution perspective, the fifth quality attribute, *maintainability*, is of great importance, as this attribute determines the ease with which the software can be adapted.

In particular, a number of internal properties of the software can be measured and captured with *metrics*. A number of these metrics have shown to be affecting the maintainability of the software system [24]. For example, complexity is often correlated with maintenance effort; that is, the more complex the code, the more effort is required to maintain it. One of the most frequently used measures during maintenance is the *McCabe cyclomatic complexity* [27], which measures the number of linearly independent paths through the code. Other measures that influence the maintainability are the more traditional object-oriented design metrics (see Chidamber and Kemerer [8]) and simple metrics like lines of code, number of comment lines, number of modules, number of methods/procedures per class/module, etc.

5.3 Code Smells and Problem Detection

Code smells are symptoms in the source code or the behavior of a software system that possibly indicate a deeper problem [18]. Many code smells are associated with difficulties in maintaining the software system, e.g., the *duplicate code* code smell. In general, code smells can be identified in several ways: through the use of code metrics, particular relations between source code elements, specific behavior of the software system or through a series of changes that have been made to the code.

Table 2 gives an overview of some common code smells. Fowler considers the *duplicate code* code smell as the *number 1* code smell, a smell that should absolutely be avoided. Duplicated pieces of source code are typically referred to as *'clones'*, which Basit and Jarzabek define as [1]: "code fragments of considerable length and significant similarity". Because code clones are considered an important code smell, the area has seen a great interest of the research community. This interest can be explained by the fact that code clones are seen as plausible arguments for an increased maintenance effort, in particular, changes have to be made multiple times because the code is redundant. Another important reason for the research interest in code clones is that clones can lead to bugs. In particular, when only a few instances of a cloning relation are changed, there might be side-effects from those instances that have erroneously not been changed.

Research in the area has on the one hand concentrated on developing code clone *detection* and *removal* techniques [23]. Many of these removal techniques are based upon series of refactorings (see Section 5.4). On the other hand, researchers have also concentrated on *code clone management*, which entails that the Integrated Development Environment (IDE) which the software engineer is using, is *aware* of the clones and can warn the software engineer when he is actually making changes to an instance of a code clone [15].

Nevertheless, code cloning is not all bad. Kapser and Godfrey's study has shown that code cloning is sometimes a purposeful implementation strategy which makes sense under certain circumstances [22].

5.4 Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code [18]. Crucial to this restructuring approach is that while the internal structure of the software is improved, the external behavior is not changed. Typically, refactoring a software system is composed of a series

Duplicate code	identical or very similar pieces of source code
God class	an object that controls too many other objects in the system; an object that does everything
Long method	a method that tries to do too many things
Large class	a class that contains too many subparts and methods
Long parameter list	a method that tries to do too much, too far away from home, with too many subparts of the system
Divergent change	occurs when a class is frequently changed in different ways for different reasons, an early sign of a god class in the making
Shotgun surgery	each time you want to make a single, seemingly coherent change, you have to change lots of classes in little ways
Feature envy	a method seems more interested in another class than the one it is defined in.
Inappropriate intimacy	a method that has too much intimate knowledge of another class or method's inner workings, inner data, etc

Table 2. An overview of some typical code smells [18]

of small behavior preserving transformations. These small steps ensure that (1) there is less chance that something can go wrong, and (2) they make it easier to verify that the behavior has indeed been preserved by making use of unit testing [31].

One should however be careful that the refactorings do not break the unit tests and thus invalidate the safety net that is provided by the unit tests. Research has shown that at least 20 of the refactorings that are listed by Fowler in [18] break the unit tests [31]. This in turn implies that unit tests can also become an evolutionary burden, because they too need to be refactored.

6 Mining Software Repositories

With the advent of open source projects, configuration management systems, bug-tracking systems, open source projects, and, last but not least, the Internet as communication platform new information sources became available for empirical software engineering research. The growing interest in this research led to special tracks at software engineering conferences and later on to workshops and conferences on its own. The most popular venue is called Mining Software Repositories (MSR)[4] that addresses the following research topics:

- Meta models to integrate, represent, and exchange information stored in software repositories;
- Meta models to model social, organizational, software development processes;
- Meta models to represent software quality aspects;
- Mining techniques to analyze the information;
- Application areas in which the results can be applied, e.g., search-based software engineering, software evolution analysis, software reliability assessment, cost estimation, bug and change impact analysis.
- Visualization techniques to represent data and mining results;

A survey on corresponding approaches for mining software repositories has been presented by Kagdi *et al.* in [21]. We further would like to refer interested readers to the special issue on MSR that has been edited by Nagappan *et al.* [35].

One of the main objectives is to *improve current software development practices by learning from historical evidence provided by information stored in software archives*. In the following, we list the main research directions of software repository mining and for each summarize key approaches and research results.

6.1 Data Modeling

Recent literature highlights source-control systems, defect-tracking systems, and mailing lists archives as the main data sources for MSR. Source-control systems, such as CVS or subversion, are used for managing and storing the various versions of source code artifacts and the modification reports that led to the new versions. Defect-tracking systems, such as Bugzilla or Jira, are used to report and manage defects and enhancement requests. Mailing lists keep track of discussions between software developers, testers, and also end users. Furthermore, the advent of Web 2.0 technologies (i.e., Wiki, Blog, Twitter, etc.) provides various new alternatives to share information and discussions over the process, design, implementation, etc. of software systems.

One of the key problems faced by researchers in the domain of mining software repositories is to extract and link all the different information sources to obtain a more complete picture over the software project. As noted by Kagdi *et al.*, software repositories vary in their usage, information content, and storage format. Typically, they are managed and operated in isolation and have no explicit direct relationship with each other [21]. Furthermore, not all software projects follow the same development process (or a development process at all). For example, while some projects agreed on using a versioning system, they do not use a bug-

[4]More information on this series of conferences can be found at: http://msr.uwaterloo.ca/

tracking system. Although a project team is using a versioning system, a bug-tracking system, and mailing lists, they operate them in isolation. There is no strict process rule that enforces developers to link commits to bugs and comments in mailing lists. Last but not least, these tools were explicitly made for developing software but not for performing software archeology as done with software repository mining.

Improvements and a step towards a more measurable software development process can be expected from recent trends in Integrated Development Environments (e.g., IBM's Jazz, Microsoft Team Foundation Server) and open source products, such as Hudson[5], CruiseControl[6], and Continuum[7]. They provide support for continuous integration that better integrates the various tools and repositories for developing software systems.

A number of approaches have been introduced in the past to re-establish the links between the information residing in the different software repositories. The main objective is to obtain a common information source which then can be input into mining algorithms to analyze various aspects of the software system. In the following we summarize approaches that have been frequently used and cited by the MSR community.

The *release history database* (RHDB) integrates data from versioning systems obtained from CVS or SVN with bug report data obtained from Bugzilla [17]. Figure 2 depicts the corresponding data model whereas the link is denoted by the association between the `FileVersion` and `BugReport` entities.

The file history for each source file is extracted from the log information that is queried from the versioning systems, for example with the CVS log command. Bug report data is obtained from the Bugzilla repository by requesting the bug in XML format. The XML then is parsed and extracted information is stored into the RHDB. The links between file revisions and bug reports are established in a post-processing step. Regular expressions, such as

```
bugi?d?:?=?\s*#?\s*(\d\d\d+)(.*)
```

are used to query bug numbers in the log messages stored at each file revision. Whenever a valid bug number is found a link to that bug report is stored into the database. A similar approaches has been presented by Bevan *et al.* with the Kenyon framework [4].

There have been several extensions to the RHDB approach that take into account additional data sources. For example, Hipikat forms an implicit group memory that, in addition to versioning and bug report data, stores data from mail archives, and online project documentation [13]. Figure 3 depicts the Hipikat data model. It comes with a text similarity matcher to infer links between the various data elements. Through these links the group memory then can be queried and navigated by developers to obtain recommendations on a task at hand. Another interesting feature is the incremental update of the group memory when new reports, mails, file revisions, and emails are added.

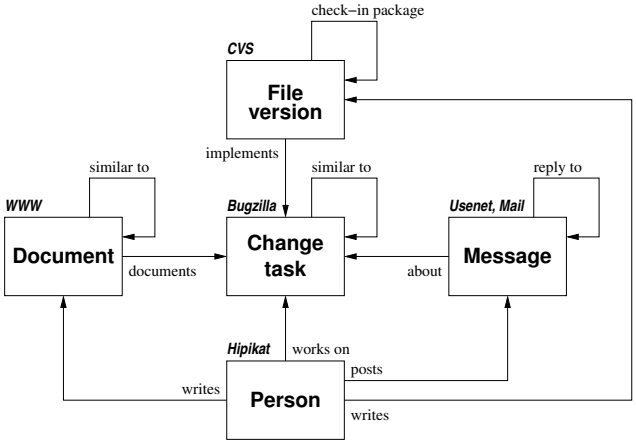

Figure 3. The Hipikat data model integrating versioning and bug report data with mails and project documentation.

6.2 Applications of Software Repository Mining

The integrated data models, such as provided by RHDB and Hipikat offer various application areas to analyze different aspects of a software system related to software evolution. Kagdi *et al.* [21] list a number of MSR tasks that have been addressed by recent approaches. These are: Evolutionary couplings/patterns, change classification/representation, change comprehension, defect classification and analysis, source code differencing, origin analysis and refactoring, software reuse, development processes and communication, contribution analysis, evolution metrics. Out of these application areas and tasks we focus on defect prediction and recommender systems which we will detail in the following.

Defect Prediction The main goal of defect prediction is to calculate a model that when applied to the current system tells the developers which modules will most likely be affected by a bug in the next release and/or how man bugs that will be. The results of the prediction then can be used to take preventive and corrective actions, such as refactoring the highlighted software modules and increase testing

[5] `http://hudson-ci.org`
[6] `http://cruisecontrol.sourceforge.net`
[7] `http://continuum.apache.org`

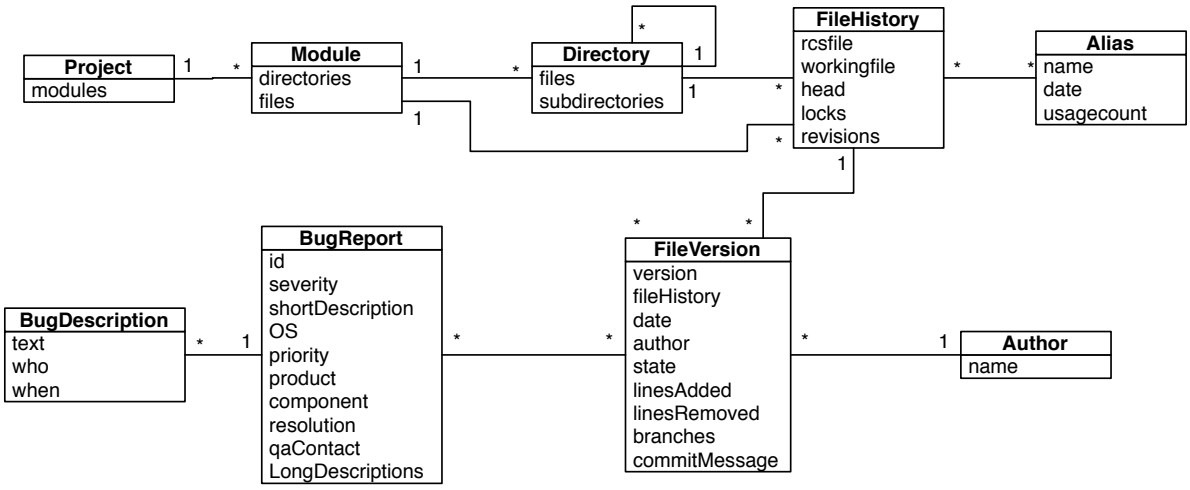

Figure 2. The Release History Database model for integrating versioning with bug report data.

efforts for these modules.

Various machine learning techniques (binary and linear regression analysis, decision trees, naive Bayes, super vector machines, etc.) can be used to train and test prediction models. Basically, resulting modules establish relationships between a set of independent variables and a dependent variable. Typically, the number of defects per module denotes the dependent variable as being predicted by the model. Product and process metrics, such as modules size (e.g., lines of code) and complexity (e.g., McCabe), number of changes and defects in the past, age of the module, are typically used as the independent variables.

A prediction model first is trained and tested with metric values obtained from *past* software releases. Ideally, the validation then is performed with metrics obtained from *subsequent* software releases. Often, however, it is performed with the same data, but then using standard validation techniques, such as ten-fold-cross validation. Depending on the machine learning algorithm used, various performance measures are used to evaluate the predictive power of obtained models, such as precision, recall, and area under ROC curve (AUC). Models with high performance are assumed to perform well for predicting the defects of future release.

Early approaches mostly favored product metrics, such as size and complexity metrics, for defect prediction. Soon process metrics have been added and discussions arose which ones perform best. For example, Graves *et al.* explored the extent to which the change history can be used to predict defects [19]. They found that the number of changes and the age of modules outperform product measures, such as lines of code. Their results were confirmed by a number of recent studies, such as presented by Nagappan *et al.* [34] and Moser *et al.* [32]. In contrast, Menzies *et al.* used static code attributes, such as program size and complexity metrics, to predict defects [30]. Their results showed predictors with a mean probability of detection of 71 percent and mean false alarms rates of 25 percent which denote models with reasonable quality. These authors also made an important point: the choice of the learning method is far more important than which subset of the available data is used for learning. They further advise to assess defect prediction methods with multiple data sets and multiple learners. Another interesting discussion with valuable feedback on previous research in defect prediction is presented by Fenton and Neil [16]. They point out relevant issues, such as the unknown relationship between defects and failures, the use of multivariate approaches, and issues in the statistical methodology and data quality. These issues should be dealt with when doing research in this direction.

Recommender Systems Another promising application of the integrated data models are recommender systems. Whenever a modification task is performed the recommender system observes the behavior of developers and based on that context provides a set of recommendations aiming at automating the change task and improving the quality of a software system.

For example, Hipikat is a tool that provides recommendations about project information a developer should consider during a modification task [13]. For that it mines a broad set of information sources including source file revisions, bug reports, email archives, and online project documentation and establishes the links between these information sources. When a new bug is reported the developer can query for similar bug reports and obtain recommendations

on linked source code modifications, email discussions, and the project documentation which might aid in fixing the bug. ROSE is an approach and tool that infers recommendations from source file revisions by using association rule mining [44]. Whenever a developer is performing a modification task ROSE's recommendations guide him along related changes in the way: "Programmers who changed this function also changed these other functions." In addition to these suggestions, ROSE also uses the association rules to warn of incomplete changes. A similar approach, but on the level of source files and using frequent item set mining, has been presented by Ying *et al.* in [42].

In addition to approaches that mine the project history, several approaches exist that mine source code repositories to recommend examples on how to use an API or to improve code completion. Code completion is a popular feature offered by modern integrated development environments that is extensively used by developers. Basically, it helps to prevent developers from writing not compilable source code, and to speed up programming by proposing program elements that are syntactically correct. Investigating current code completion implementations, Bruch *et al.* argued that the quality of suggestions and hence the productivity of software development canbe improved by employing intelligent code completion systems [6]. They evaluated three such code completion systems of which the one using a modified version of the k-nearest-neighbor (kNN) machine learning algorithm performed best. 82% of the recommended method calls were actually needed by the programmer (recall) and 72% of the recommended method calls were relevant (precision).

Concerning the usage of an application programming interface (API), developers often encounter difficulties, such as, which objects to instantiate, how to initialize them, and which methods to call. The problem has been addressed by a number of approaches, such as Strathcona [20]. Typically, they form a knowledge base by mining a set of framework usage examples. Given the current context of the developer the tool queries the knowledge base and recommends code snippets of potential interest. However, it is still the task of a developer to integrated the appropriate code snipped into the source code.

7 Software Evolution in Research

Software evolution has received considerable attention over the past few years. If the reader is interested in consulting the body of knowledge in the area of software evolution, the conferences listed in Table 3 form a good starting point.

Furthermore, also journals like the IEEE Transactions on Software Engineering, ACM's Transactions on Software Engineering and Methodology, Wiley's Journal of Software Maintenance and Evolution, Springer's Emprical Software Engineering, Elsevier's Journal of Systems and Software and Wiley's Software Practice and Experience have frequent contributions in the broader area of software evolution.

[8]Formerly IWPC, the International Workshop on Program Comprehension.

8 Future Challenges

Besides the general main areas (comprehension, reverse engineering, reengineering, and repository mining), a number of trends and application areas can be distinguished. The most important ones include the following.

Requirements Traceability The need for change is reflected in evolving requirements. To assess the impact of changing requirements, it is essential that source code and design decisions can be traced back to requirements. Unfortunately, maintaining accurate requirements traceability links has proven to be hard and costly in practice. Promising research directions aim at partially automating this process, for example through the use of information retrieval techniques such as *latent semantic indexing* [26]. By computing textual similarities between different work documents, candidate traceability links between, for example, code, test cases, and requirements can be computed. Turning the candidate links into confirmed links is still a manual process, but initial results are promising.

Service-Oriented Architectures Software portfolios of large enterprises can easily comprise hundreds of applications. The evolution of the application landscape of such a company over the years has typically resulted in a complex network of connected systems, in which system dependencies are often critical, but equally often poorly understood. In order to regain control over the system dependencies many companies are investigating the use of service-oriented architectures. In this approach, systems are loosely coupled, and communicate via an *enterprise service bus*. Service interfaces are made as stable as possible, but service implementations can be (dynamically) identified and modified.

While promising in many ways, the adoption of service orientation brings in a number of challenges. These include migrating legacy components into services (for example via wrapping), performance implications of data conversion that is required to integrate components developed by different organizations, and systematic testing strategies for dealing with large collections of services not under direct control of the testing organization. This testing may require obtaining production data and connecting to services in production. Since replication of test data and production

Conference acronym	Conference name	Year first organized
ICSM	the International Conference on Software Maintenance	1983
ICPC	the International Conference on Program Comprehension[8]	1993
WCRE	the Working Conference on Reverse Engineering	1994
CSMR	the European Conference on Software Maintenance and Reengineering	1997
MSR	the Working Conference on Mining Software Repositories	2004
IWPSE	the International Workshop on the Principles of Software Evolution	1998

Table 3. Conferences in the area of software evolution

services can be challenging, an interesting research area is to devise service infrastructures permitting (controlled) test execution in the production environment.

Collaborative Engineering Software development and evolution is a team activity. Much of the existing body of work in reverse engineering and program comprehension is focused on the *individual* developer. Methods and techniques have been proposed to visualize architectures, to identify features from execution traces, or to establish traceability links. The question arises how such tools can be used to *collaboratively* understand, for example, the implementation of a particular feature. Such tools should support the incremental growth of understanding, allowing different developers to record their intermediate knowledge, and share it with their co-developers.

Globally Distributed Development More and more, software teams work on different locations spread across the globe. In addition to this physical distance, the team members come from different cultures, often working in different time zones. Many of the techniques developed to support evolution (tools for program comprehension, impact analysis, or configuration management) will also help to support teams counter the effects of these increased distances.

Software Testing Automated testing is an important evolution enabler. An automated test suite can be used for regression testing, to ensure that software modifications do not break existing functionality. Furthermore, continuous integration in combination with nightly execution of the test suite will help to identify problems caused by changes made to the software as early as possible.

In agile development methods, test suites furthermore are used to facilitate program comprehension. As an example, in the Ruby community software is described via "executable examples" either expressed in Ruby itself (when using rspec[9]), or in natural language which via a simple pattern recognition mechanism is translated into an executable test suite (when using Cucumber[10]).

This interplay between testing, test automation, and documentation comprises an interesting and highly promising route in further supporting software evolution.

References

[1] H. A. Basit and S. Jarzabek. Efficient token based clone detection with flexible tokenization. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 513–516. ACM, 2007.

[2] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999.

[3] K. Bennett. Legacy systems: Coping with success. *IEEE Software*, 12(1):19–23, 1995.

[4] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 177–186. ACM, 2005.

[5] M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces & The Incremental Approach*. Morgan Kaufmann, 1995.

[6] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 213–222. ACM, 2009.

[7] G. Canfora and M. Di Penta. New frontiers of reverse engineering. In L. C. Briand and A. L. Wolf, editors, *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007*, pages 326–341, 2007.

[8] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[9] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[10] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[9] http://rspec.info/

[10] http://github.com/aslakhellesoy/cucumber

[11] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252–2268, 2008.

[12] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[13] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.

[14] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-oriented reengineering patterns*. Morgan Kaufmann, 2003.

[15] E. Duala-Ekoko and M. P. Robillard. Clonetracker: tool support for code clone management. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 843–846. ACM, 2008.

[16] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

[17] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 23–32. IEEE Computer Society, 2003.

[18] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[19] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[20] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, 2006.

[21] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[22] C. Kapser and M. W. Godfrey. "cloning considered harmfu" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE)*, pages 19–28. IEEE Computer Society, 2006.

[23] R. Koschke. Identifying and removing software clones. In Mens and Demeyer [29], pages 15–36.

[24] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[25] M. M. Lehman and L. A. Belady. *Program Evolution: Processes of Software Change*. Apic Studies In Data Processing. Academic Press, 1985.

[26] M. Lormans, A. van Deursen, and H.-G. Gross. An industrial case study in reconstructing requirements views. *Empirical Software Engineering*, 13:727–760.

[27] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[28] T. Mens. Introduction and roadmap: History and challenges of software evolution. In Mens and Demeyer [29], pages 1–11.

[29] T. Mens and S. Demeyer, editors. *Software Evolution*. Springer, 2008.

[30] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[31] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink. On the interplay between software testing and evolution and its effect on program comprehension. In Mens and Demeyer [29], pages 173–202.

[32] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 181–190. ACM, 2008.

[33] H. A. Müller, S. R. Tilley, and K. Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 217–226. IBM Press, 1993.

[34] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 284–292. ACM, 2005.

[35] N. Nagappan, A. Zeller, and T. Zimmermann. Guest editors' introduction: Mining software archives. *IEEE Software*, 26(1):24–25, 2009.

[36] D. L. Parnas. Software aging. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 279–287. IEEE Computer Society Press, 1994.

[37] N. Pennington. Stimulus structures and mental prepresentations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3), 1987.

[38] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proc. IEEE WESTCON*. IEEE Computer Society Press, August 1970. Reprinted in Proc. ICSE 1989, ACM Press, pp. 328-338.

[39] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, August 1995.

[40] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering (ICSE)*, pages 439–449. IEEE Press, 1981.

[41] S. S. Yau, J. S. Colofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *Proc. COMPSAC*, pages 60–65. IEEE Computer Society Press, 1978.

[42] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[43] A. Zaidman and S. Demeyer. Automatic identification of key classes in a software system using webmining techniques. *Journal of Software Maintenance*, 20(6):387–417, 2008.

[44] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.