# Supporting Software Evolution Analysis with Historical Dependencies and Defect Information

Marco D'Ambros
REVEAL @ Faculty of Informatics
University of Lugano, Switzerland

## Abstract

*More than 90% of the cost of software is due to maintenance and evolution. Understanding the evolution of large software systems is a complex problem, which requires the use of various techniques and the support of tools. Several software evolution approaches put the emphasis on structural entities such as packages, classes and structural relationships. However, software evolution is not only about the history of software artifacts, but it also includes other types of data such as problem reports, mailing list archives etc.*

*We propose an approach which focuses on historical dependencies and defects. We claim that they play an important role in software evolution and they are complementary to techniques based on structural information. We use historical dependencies and defect information to learn about a software system and detect potential problems in the source code. Moreover, based on design flaws detected in the source code, we predict the location of future bugs to focus maintenance activities on the buggy parts of the system. We validated our defect prediction by comparing it with the actual defects reported in the bug tracking system.*

## 1 Introduction

The analysis of the evolution of software [16] has two main goals, namely to infer causes of its current problems, and to predict its future development. Many approaches based on evolutionary information demonstrated that not only can such information be used to predict the future evolution [18], but it can also point out potential problems in the system [12]. Understanding the evolution of large software systems is a complex problem for several reasons: huge amounts of information have to be considered and historical data has to be analyzed to understand the phenomena of evolution and to infer causes of problems. The evolution of a software system is not only the collection of all the versions of its components: developing software is a human activity, and the evolution of a software system also includes the activities performed by developers, testers, users, *etc.* during the entire history of the system. This additional information comes from various sources such as comments committed by developers during the implementation, problem reports delivered by users and stored in bug tracking systems, mailing list archives, *etc.* Several software evolution analysis techniques have been proposed which either focus on the source code and its evolution, without exploiting other data sources such as problem reports, or they use additional information (*e.g.,* e-mail archives), without a direct link to the source code.

We propose an approach to support software evolution which focuses on historical dependencies and defects. We chose these types of information for the following reasons:

- Historical dependencies are visible in the evolution of a system only, and not in its structure, and are therefore the most troublesome since they are not visible to the developers while coding. Violating these dependencies can lead to maintenance problems and generate bugs.
- Software maintenance aims at keeping the quality of software at a good level. Software defects are a tangible effect of bad software quality, and thus they are a measure of the maintenance cost of a system. Defect information is helpful to detect the problematic parts of a software system.
- Several sources of information can be used to study the evolution of a software. However, the final goal of using them is to ease software maintenance activities, by detecting problems in the source code. The benefits of historical dependencies and defects, over other data sources such as mailing list archives, is that they can be directly linked to the source code. Historical dependencies are between software artifacts and defects affect certain parts of the source code.

## 2 Approach and Validation

Figure 1 shows the overall schema of our approach, divided in: (1) Software system, (2) Model and (3) Analysis.

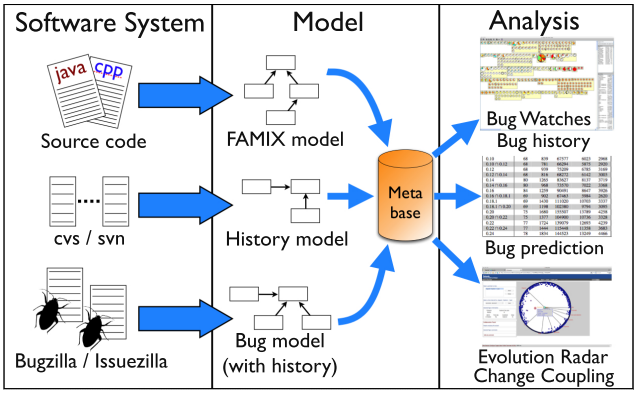


**Figure 1. The general schema of our approach to software evolution analysis.**

**Software System.** To do software evolution analysis we consider the following data sources about a software project: the source code (written in various languages such as Java, C++, *etc.*), the versioning system log files (CVS or SubVersion) and the problem report data (Bugzilla or Issuzilla repository[1]).

**Model.** We have defined an evolutionary meta-model of software systems. It describes (1) the source code by means of the FAMIX language independent meta-model [7], (2) the history of software artifacts [4] and (3) the problem reports together with their histories [6]. The model is populated in batch mode, by just providing the url of the SubVersion (or CVS) and Bugzilla (or Issuezilla) repositories from a web interface. In [4] we describe the details of the meta-model and the web interface. We also discuss how we provide flexibility to the meta-model through an engine, called meta-base, able to automatically generate object persistency descriptors based on the EMOF (Essential Meta Object Facilities[2]) meta-meta-model.

**Analysis.** Our software evolution analysis approach has two main goals: first we want to use evolutionary information about dependencies and defects to detect potential problems in the system. Second, we want to detect particular design flaws in the source code and use them to predict locations of future bugs. Both these activities lead us to identify on which parts of the system the maintenance effort should be spent. To achieve these goals, we have introduced three approaches, presented in the remainder of this section: (1) visualizations (Bug Watches) to understand the history of bugs, (2) design flaws detection for defect prediction and (3) visualizations to perform change coupling analysis.

[1] See www.bugzilla.org/ and www.netbeans.org/kb/articles/issuezilla.html
[2] http://www.omg.org/docs/html/06-01-01/Output/06-01-01.htm

In our approaches we make use of various visualizations. We decided to use visualization because it provides effective ways to break down the complexity of information, and because it has proven to be a successful means to study the evolution of software systems [13, 14, 20].

## 2.1 Bug History Analysis

Bug tracking systems are used by developers, quality assurance people, testers, and end users to provide feedback on software systems. They are also used in software evolution research to perform retrospective system analysis [3, 8]. In this context bugs are linked to software artifacts (*e.g.,* files, classes) using different heuristics, with the aim of detecting the most problematic parts of the system, *i.e.,* the ones affected by many bugs. Some approaches model bugs as mere numbers (*e.g.,* file $x$ is affected by $n$ bugs, file $y$ by $m$), while others also model bug properties such as the description, the severity, the person assigned to fix it, *etc.* However, bugs are often considered as an unwanted "side dish" of the evolution phenomenon, and they are modeled as "static" entities affecting the source code.

We have proposed an approach in which we consider bugs as *first-level* entities which can change and evolve over time. In particular we have focused on the bug life cycle, *i.e.,* the history of a bug and the various states it traverses. Our hypothesis is that bug histories represent a valuable source of information that can lead to interesting insights about a system, that would be hard or impossible to obtain by modeling the bugs as static entities. Based on the information we recovered from Bugzilla (or Issuezilla), we have introduced two visualization techniques aimed at understanding bugs at two different levels of granularity:

1. *System Radiography.* This visualization renders bug information at the system level and provides indications about which parts of the system are affected by what kind of bugs at which point in time. It is a high-level indicator of the system health and serves as a basis for reverse engineering activities.

2. *Bug Watch.* This visualization provides information about a specific bug and is helpful to understand the various phases that it traversed. The view supports the characterization of bugs and the identification of the most critical ones, based on their histories.

The proposed approach provided two main contributions: (1) introducing the concept of a bug's life, *i.e.,* bugs are considered as evolving entities which change over time. Studying the history of bugs permits an accurate characterization of them. (2) Introducing a new criterion for bug criticality: besides the severity and priority we have also considered the life cycle. The underlying assumption is that bugs reopened several times are more critical.

We have applied the technique on the Mozilla software project, finding interesting insights into the system, and detecting the most critical components. However, we still need to do a more complete validation of the approach, in particular by giving feedback to the developers and evaluating the usefulness of our findings.

### 2.2 Defect Prediction

Defect prediction deals with guessing where in a system there will be bugs, thus providing valuable information to developers and project managers, since this allows them to focus resources. Previous research has proven that the best predictor for bugs are the bugs themselves [22], *i.e.,* entities affected by bugs in the past usually suffer from them in the future as well, as long as no substantial efforts are spent on restructuring the ailing parts. This however presumes the usage of a bug tracking system. How can one predict bugs in the absence of recorded bugs or in the case of a freshly developed system? Researchers tried to answer this question with complexity metrics, where the assumption is that complex pieces of software generate bugs [19].

We have proposed an approach (under submission) to predict defects based on the presence of so-called "design disharmonies" [15], which can be discovered through a technique based on detection strategies [17]. Detection strategies are metrics-based composed logical conditions, by which design fragments with specific properties are detected in the source code. Design disharmonies are similar to code smells [9], where the difference is that by translating a set of design guidelines or heuristics the former can be automatically uncovered with detection strategies. Our underlying assumption is that pieces of software which exhibit design problems are also prone to generate bugs.

To validate our technique, we applied it on version $n-1$ of a system, obtaining a list of classes with the numbers of predicted bugs. We then compared this with the actual bugs reported in version $n$ of the system, and computed the prediction performance. We experimented the approach on several versions of 3 software systems (Eclipse JDT Core, ArgoUML, AspectJ), and were able to prove that design disharmonies represent a good means to predict defects, obtaining an increase in prediction power over other approaches.

### 2.3 Change Coupling Analysis

Change coupling is the implicit dependency between two or more software artifacts that have been observed to frequently change together during the evolution of a system. This dependency provides additional types of information that are not visible when only one version of a system is considered. Previous research has dealt with low-level couplings between files [21, 23], leading to an explosion of data to be analyzed, or has abstracted the change couplings to module level [10, 11, 20], leading to a loss of detailed information.

We proposed an approach which integrates change coupling information at different levels of abstraction, to detect areas in the system which may lead to maintenance problems. Our technique uses an interactive visualization called the *Evolution Radar* [2,5], which can effectively break down the amount and complexity of the change coupling data.

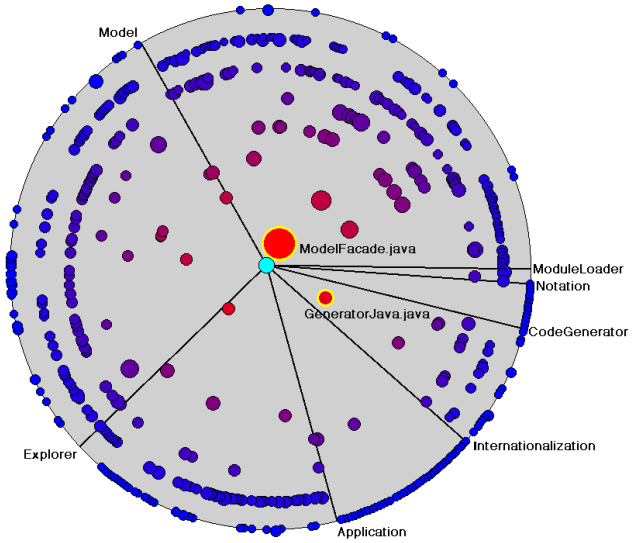


**Figure 2. The Evolution Radar visualization.**

Figure 2 shows an example of the Evolution Radar, applied to the *Diagram* module of ArgoUML. The visualization shows the dependencies between a module, represented as a circle and placed in the center of a pie chart, and all the other modules in the system represented as sectors of the pie chart. In each sector, all the files belonging to the corresponding module are represented as colored circles and positioned according to the change coupling they have with the module in the center (the higher the coupling the closer to the center). The Evolution Radar allows us to understand the dependencies between modules and to detect the main responsible for such dependencies in terms of files, represented by the circles closest to the center.

We have validated our approach on three large open source software systems: Mozilla [5], PostgreSQL [1] and ArgoUML [2]. We have applied the Evolution Radar to understand module dependencies and to detect candidates for reverse engineering. We have found design issues and dependencies between modules not mentioned in the documentation. We have also reduced these dependencies to coupling between small sets of files, which should be reengineered in order to decrease the coupling at the module level.

## 3 Conclusion

To support software evolution analysis we have proposed a technique based on historical dependencies and defect information. We chose these two kinds of data about a software system because they are complementary to structural information and, differently from other data sources (*e.g.,* mail archives), they can be directly linked to the source code.

The main contributions of our work so far are:

- A meta-model for defects, which takes time into account. It considers bugs as evolving entities, while in previous research they were considered static entities.
- Two visualizations (System Radiography and Bug Watch) which, exploiting the defect meta-model, supports the understanding of bugs evolution and the detection of critical bugs. The riskiness of a bug is defined according to its history, and in particular to its life cycle, whereas in previous approaches only static attributes (such as severity and priority) were considered.
- A defect prediction technique based on particular design flaws called design disharmonies. The approach represents an improvement over the state of the art in metrics-based defect prediction.
- The Evolution Radar visualization, which shows change coupling information at different levels of abstraction, supporting the understanding of both module dependencies and the causes of the dependencies. Previous techniques focused on either coarse-grained coupling, *i.e.,* at the module level, or fine-grained coupling, *i.e.,* at the file (or finer) level.

We plan to continue our work in two directions: first we want to investigate other types of historical dependencies, for example the one based on bug sharing. The assumption is that two entities sharing a bug, over the system's history, have an implicit dependency. The greater the number of bugs they share, and the longer the time during which they share bugs, the stronger the dependency is. In the second research direction we plan to use historical dependency information to predict bugs. This would integrate our approaches for historical dependency analysis and bug prediction.

**Acknowledgments.** We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "DiCoSA - Distributed Collaborative Software Analysis" (SNF Project No. 118063).

## References

[1] M. D'Ambros and M. Lanza. Applying the evolution radar to postgresql. In *Proceedings of MSR 2006*, pages 177–178.

[2] M. D'Ambros and M. Lanza. Reverse engineering with logical coupling. In *Proceedings of WCRE 2006*, pages 189–198. IEEE CS Press, 2006.

[3] M. D'Ambros and M. Lanza. Software bugs and evolution: A visual approach to uncover their relationship. In *Proceedings of CSMR 2006*, pages 227–236. IEEE CS Press, 2006.

[4] M. D'Ambros and M. Lanza. A flexible framework to support collaborative software evolution analysis. In *Proceedings of CSMR 2008*, pages 3–12. IEEE Computer Society, 2008.

[5] M. D'Ambros, M. Lanza, and M. Lungu. The evolution radar: Visualizing integrated logical coupling information. In *Proceedings of MSR 2006*, pages 26–32, 2006.

[6] M. D'Ambros, M. Lanza, and M. Pinzger. "a bug's life" - visualizing a bug database. In *Proceedings of VISSOFT 2007*, pages 113–120. IEEE CS Press, 2007.

[7] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[8] M. Fischer and H. Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):385–403, 2004.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[10] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of ICSM 1998)*, pages 190–198. IEEE CS Press, 1998.

[11] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proceedings of IWPSE 2003*, pages 13–23. IEEE Computer Society Press, 2003.

[12] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of ICSM 2004*, pages 40–49. IEEE CS Press, 2004.

[13] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of CSMR 2005*, pages 2–11. IEEE CS Press, 2005.

[14] M. Jazayeri, H. Gall, and C. Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. In *Proceedings of ICSM 1999*, pages 99–108. IEEE Press, 1999.

[15] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[16] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[17] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of ICSM 2004*, pages 350–359. IEEE CS Press, 2004.

[18] T. Mens and S. Demeyer. Future trends in software evolution metrics. In *Proceedings of IWPSE 2001*, pages 83–86, 2001.

[19] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of ICSE 2006*, pages 452–461. ACM, 2006.

[20] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005*, pages 67–75, 2005.

[21] J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. In *Proceedings of MSR 2005*, pages 1–5. ACM Press, 2005.

[22] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of ICSEW 2007*, page 76. IEEE CS Press, 2007.

[23] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of ICSE 2004*, pages 563–572. IEEE CS Press, 2004.