VU VRIJE UNIVERSITEIT AMSTERDAM

VU Research Portal

Supporting Architecture Evolution by Mining Software Repositories

Vanya, A.

2012

document version
Publisher's PDF, also known as Version of record

Link to publication in VU Research Portal

citation for published version (APA)
Vanya, A. (2012). *Supporting Architecture Evolution by Mining Software Repositories*. [PhD-Thesis - Research and graduation internal, Vrije Universiteit Amsterdam].

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

E-mail address:
vuresearchportal.ub@vu.nl

Download date: 21. Apr. 2025

order to reach a goal. The works of Hall and Fenton [1997] and Niazi et al. [2006] are examples thereof. Other papers focus more on negative success factors, i.e. what has to be avoided in order to reach a goal. See, for instance, the work of Baddoo and Hall [2003].

Our work is similar to the ones identifying critical success factors for software process improvement. Most of the lessons learned we report in this chapter are in the form of what one should or should not do in order to successfully execute a process. In that sense, those lessons learned resemble the critical success factors for software process improvement. Also, both those critical success factors and our lessons learned address a similar question: how to collect and use data such that it will eventually bring benefits to the organization where the data is collected and analyzed.

Although they are similar, there are some differences between the lessons learned reported here and the critical success factors for software process improvement. First, we gained our experiences by (repeatedly) executing a process, while critical success factors are typically identified based on interviews with developers, experts and managers. Second, our lessons learned come from a single study environment. Critical success factors on the other hand are identified after gathering experiences from multiple projects / study environments.

6.6 Conclusion

Having a decomposition of a software system where components can evolve as independent as possible may improve evolvability in many ways. It may decrease delays in development, communication costs, the number of tests needed and alike.

In this chapter, we report the lessons learned while supporting architects at Philips Healthcare MRI improve the evolvability of their software system. In several iterations, we developed a process for doing so. The lessons learned can be organized into three major, but interrelated, categories:

- *frequent feedback loops* with stakeholders such as architects and developers are necessary so that they understand the approach used.
- *knowing the process* of how the software is being modified is crucial for the proper interpretation of the data retrieved from the software archive.
- *knowing the data* we use is needed for identifying the unwanted couplings that really matter. The key for deriving useful information is in the details

Section 1.6 defines the three concrete research questions of this thesis (**RQ-1**, **RQ-2** and **RQ-3**). All these research questions have been addressed in the previous chapters. The answers to these research questions can be summarized as follows:

RQ-1 *How to identify groups of frequently co-changing software entities?*

Answer: Knowing which software entities changed *frequently* together requires us to know first which software entities co-changed. There are several industrial environments, for in-

stance the one studied, where that knowledge is not even captured. In other words, change sets are not always recorded. Furthermore, in the studied environment the meta-data related to individual changes is rather incomplete and the process to introduce changes to the software system (i.e. using postlists) is not completely according to the standard assumptions of previous work. Therefore, in such environments one needs to think about alternative ways to approximate change sets for retrieving co-changes.

In Section 2.4 we described five variants of the well-known sliding window algorithm, see also the work of Zimmerman et al. [2004], to approximate change sets from the historical meta-data of Philips Healthcare MRI. The development process executed had a significant impact on which change set approximation alternatives we identified. The approach described in this thesis to help software architects relies on the accuracy of the change sets approximated. For instance, change sets are used to derive co-changes. Co-changes are used to create evolutionary clusters which are further selected and analyzed by the software architect and engineers. At the end, the accuracy of change sets approximated has an impact on the decisions of the software architect. Since those decisions have to be taken with care, it is important to evaluate the accuracy of the change sets approximated.

We have described two approaches to evaluate the change sets approximated, see Section 2.5. One of them makes use of an on-line survey which developers need to fill in, while the other one is based on carefully selected postlists, i.e. lists of modifications sent to the system integrator. The results from both approaches were very similar. Based on those results we could select an approximation alternative for further usage where the precision and recall values where optimally combined (89% and 84% respectively).

Having the change-sets accurately approximated, the next step is to group those software entities which changed frequently together. For this, we described a clustering approach in Chapter 3. The result of this clustering activity is a binary parameterized tree or dendrogram where nodes, also referred to as evolutionary clusters, are sets of software entities changing together. The position of the evolutionary clusters in the dendrogram shows how frequently the related entities co-changed.

RQ-2 *How to select those groups of co-changing software entities which may indicate unwanted couplings?*

Answer: To answer this research question, we present an approach in Chapter 4 which we executed in Philips Healthcare MRI and on an open source, called ArgoUML. The approach first characterizes evolutionary clusters based on a few justified properties, for instance, the number of co-changes or co-change tendencies. The architects are contacted to elicit costly past evolution anti-scenarios which should be avoided in the future. Based on those anti-scenarios, a query is formulated on the evolutionary clusters characterized. The goal of that query is to find only those evolutionary clusters which point to unwanted couplings. The result of that query gives us the potential unwanted couplings which need to be further analyzed.

It is key in the approach presented that it makes it possible to consider the input from the software architects on unwanted couplings. As discussed in Section 4.3, what is considered to be an unwanted coupling is subjective and therefore every architect needs to be helped to

find unwanted couplings in alignment with their definition of unwanted couplings. Extracting evolution anti-scenarios from the architects is our way to explore what they find to be an unwanted coupling.

The results of the approach show that the selected unwanted couplings were pointing to unwanted couplings. Also, we rejected many such evolutionary clusters which the architect did not want to analyze for a good reason but which previous approaches would not have filtered out. The ability to filter evolutionary clusters is crucial also because there are hundreds of evolutionary clusters and architects are interested only in a few of them. Those few which point to unwanted couplings.

RQ-3 *How can interactive visualizations help facilitate a detailed analysis of potential unwanted couplings?*

Answer: Interactive visualizations provide a means to its user to customize what and how to visualize. By interacting with visualizations the user gets information iteratively. In every iteration, information is collected first from the visualization and later on the user interacts with the visualization to get even more information. The interactions are guided by the interest of the user and his interest is influenced by the information previously collected.

In Chapter 5 we investigated how such an interactive exploration process can help architects and software engineers analyze and resolve unwanted couplings. In Section 5.3.2 we presented a range of possible interactions which we also implemented in our interactive visualization tool called iVis. We used this tool to visualize those evolutionary clusters which we previously filtered out for the software architect supported. For each of those clusters working sessions were organized with the architect and engineers to analyze them. During the working sessions we observed the benefits of using interactive visualizations. It turned out that interactive visualizations can not only help to reason about unwanted couplings but they can also help to find solution alternatives and assess their impact. In Section 5.5 we provide a list of observations and key lessons learned.

6.7 Future Work

Doing research is a continuous problem solving activity; while working on a research question new questions are encountered which need to be answered at a later stage. In that sense, a piece of research can rarely be finished without leaving some open questions behind. So is it with this thesis. The major areas of the envisioned future work is further elaborated on in this section.

From the laws of Lehman [1997] follows that improving the decomposition of a software system needs to be done continuously or periodically rather than only once. Future work would therefore need to investigate how the architect can be supported to *maintain* the required status of the decomposition over time. In other words, how to monitor the “health” of the software system’s decomposition.

In this thesis, mainly co-change information derived from a version management system

was used to analyze unwanted couplings. It would be interesting to research how the detailed analysis of such unwanted couplings can be helped by presenting multiple type of relations to the architect and developers. Such types of relations could be, for instance: run-time couplings, static relations, semantic couplings.

All the studies in this thesis have been carried our in a single development environment of Philips Healthcare MRI. Applying the work presented in this thesis in other environments could result in a better understanding about how to generalize the results.