Tracking the Impact of Design Changes During Software Development

Frank Padberg
Fakultät für Informatik
Universität Karlsruhe, Germany
padberg@ira.uka.de

Abstract

Design changes occur frequently during software development. Currently, there is a lack of empirical, quantitative data about the extent and impact of such design changes. In this paper, we present a methodology for collecting design change data systematically and thoroughly as a development project proceeds. The methodology is easy to apply and works for any kind of software project. The resulting database can be used to support design decisions and project planning.

1 Introduction

It's an everyday experience in software development that the design of some part of the software changes. As a software product grows towards completion, design changes are made to implement requirements or to fix design problems. In a sense, a software development project is a series of design changes.

The design of a software product and its development project are closely connected, see [18]. In particular, changes to the design have a strong impact on the progress of the development project. Since the components of a software are coupled, design changes in one component often cause changes and rework in other components. For example, when an interface offered by some component gets extended, all components which use that interface must be reworked. The impact of a design change need not be limited to reworking some of the components. A change can make some components more complicated or introduce additional components into the software. Design changes in many cases increase the total development effort and delay the project completion.

Early software design decisions often are benefit-driven. For example, designing a software to work over the Internet, even if this will increase the development cost, is a strategic design decision which may be required by the customer to strengthen the market position of the company in the future. Once the strategic design decisions have been made, a number of design decisions must be made which are cost-driven, aiming at reaching the benefits at the lowest possible cost. In particular, software designers and project managers must consider that design changes will occur frequently during the project. For a given early design of a software, designers and managers must address a number of questions, for example:

- How many design changes must be expected to occur during the project when using that design?
- Which components are likely to undergo frequent design changes?

- How much rework effort must be planned for?
- Which components are likely to cause much rework in other components when they are changed?
- Which components should be developed early so as to minimize the overall rework effort caused by design changes?

The last question points out that design decisions can't be completely separated from scheduling considerations.

To evaluate a given software design with respect to the impact of possible design changes, the components in the design must be *compared* with the components in past projects. For example, to find out how likely it is that a particular component will undergo design changes, one must study how often comparable components in past projects have undergone design changes. Two components are comparable when they have a similar degree of programming difficulty. In addition, the components must be coupled to their respective neighbor components with a similar strength, because the coupling between the components in a software plays a major role in the propagation of changes. To compare a design with the designs in past projects we must have a detailed *database* describing the design history in past projects.

The design changes during software development and their impact on the software's architecture and the development effort have not been studied empirically and systematically so far (there is some work in software maintenance, see the section on related work below). Although it is immediate that design changes have a strong impact on the progress of a development project, *we lack reliable, empirical, quantitative data about the extent and impact of design changes in software projects.* How often have design changes occured in past projects? How many components had to be reworked because of design changes? How much rework effort has been caused by design changes? We also lack empirical data about why the design changes were made and how the different design changes in a project were related.

Reconstructing the design history of a software product after its development project has been completed is practically impossible. Usually, we are left with some description of the requirements and high-level design, the deltas of the source code, and the developers' knowledge of what happened during the project. The developers' knowledge about the project is informal and fades away with time. The code deltas are recorded at the level of source statement changes and do not provide the logical links between the individual changes of the code. *Therefore, we need a methodology for collecting design change data systematically and thoroughly as a development project proceeds.* The methodology must tell what data to collect, how to collect the data, and how to evaluate the data.

In this paper, we present a methodology for building a *database of design changes* during software development projects. With each design change, the methodology documents what is changed, why it is changed, and how much effort the change causes. Design changes in different parts of the software (or at different points in time) which are logically related are linked in the database. Using the database, one can reconstruct at any time how the design of the software evolved during the project. The methodology is easy to apply and works for any kind of software development project.

2 Collecting Data

Filing and approving formal change requests, as is done in software maintenance, is much too rigid for most software development. Therefore, the methodology aims at *consistently documenting* the changes which are applied to a software's design. Whenever a developer applies a design change to the software, he creates an entry in a *design change database* to describe the change. We'll show the format of a database entry in a moment. The database is shared by all developers in a project.

Every design change results in a number of development activities to implement the change. To permit measuring the impact of a design change on the development effort, the developers also create entries in the database to describe the development activities corresponding to a change and the effort spent on the activities. Typically, the activities are documented on a daily basis.

A design change can occur at different design levels. The change can be at the level of the architecture of the software, involving several components; at the level of a single component, involving several modules (or classes) of that component; or, at the level of a single module. In most cases, a change will *imply* other changes at "lower" levels. For example, to implement some requirement it might be necessary to change the architecture by introducing an additional component but also to extend some modules in existing components to connect with the new component. Therefore, changes are naturally organized into *coherent sets.* To permit measuring the "span" of a design change, all changes belonging to the same coherent set are linked in the database by refering to the same identification number. This is similar to software maintenance where changes are linked to modification requests.

Here is a description of the fields contained in a database entry:

- *Identification Number.* Each coherent set of design changes gets an identification number. All changes in a coherent set refer to the same identification number.
- *Description.* The purpose and content of the change. For the first change of a coherent set ("triggering change") the description indicates whether the change is made to implement some (possibly new) requirement or to fix some design problem.
- *Changed Item.* The name of the component, module, function, or shared variable whose design is changed. The name must be unique. Thus, the naming scheme "component.module.function" is used.
- *Type of Change.* One of the four categories Changed, Extended, New, Dropped.
- *Date.* The day and month when the entry was made. For most entries, this will be the date of some development activity performed to implement the change.
- *Activity Times.* For each development activity performed on the day specified in the Date field to implement the change, the time spent on the activity is recorded. The activities belong to the five categories Designing, Coding, Testing, Documenting, Other.
- *Developer.* The name of the developer who made the entry. Each developer who spent some time on a change must make his own entries for that change in the database.

In most cases, a particular change results in multiple entries in the database. The entries correspond to different times at which a developer worked on implementing the change and different developers

implementing the change. For example, a developer might spend a total of 40 minutes of designing, 20 minutes of coding, and 10 minutes of testing for some change on one day, and an additional 10 minutes of coding and 30 minutes of testing for the same change on the next day. That would lead to two entries in the database for the change. It also is natural that a component or module occurs in the Changed Item field of database entries which belong to different coherent sets of changes, for example, when fixing a design error made during some earlier design change.

Partial entries in the database can and should be made as soon as some components and modules are known to surely be affected by a change. Partial entries will include the names of the components and modules, but no effort data. Recording such information helps to get an overview of the extent of a change and maintain a list of tasks yet to be accomplished to implement the change. In particular, looking at the functions and shared variables which are affected by a change helps to identify which other parts of the software will be affected by the change. The first entries for a project should be made when the initial high-level design has stabilized.

3 Related Work

Currently, tracking design changes and their impact during software development is not well supported. Version control systems record changes at the level of source code statements and not at the design level. In addition, it is difficult to properly link version control data to time sheet data, see [3, 12]. Bug trackers, such as Bugzilla [10], Gnats [11], and Jitterbug [13], are limited in scope.

Changes are well documented in software *maintenance* using formal modification requests. In [4], a methodology for collecting change data during maintenance is presented which includes collecting effort data. That approach is similar to ours, and the lessons learned about the data collection process are valuable.

Recent work in design metrics for object-oriented software provides empirical evidence that the strength of the coupling between classes has a large impact on the fault-proneness of the classes [6, 7, 8, 9] and on the effort to correct faulty classes in maintenance [7]. Earlier work studied the suitability of source code metrics as predictors for the effort to correct faulty components [1, 5, 14]. Practical guidelines how to design a software to minimize faults and rework effort are scarce. How should components be designed and coupled to limit the propagation of changes? A qualitative approach for analyzing the tradeoffs in a software's design has been presented in [2, 15]. We have outlined a quantitative approach in [18].

For the purpose of project effort estimation, a quantitative model for software projects is required which takes the software's design as input and reflects the impact of design changes on the progress of a project. We have already presented such a model in [16, 17, 18, 19]. A design change database, as the database presented here, yields part of the input data needed for the project estimation model.

4 Discussion

It does not make much sense to collect data about each minor change or bug fix, especially while coding and unit testing. The emphasis with the methodology is on documenting *significant* changes.

In particular, changes which require coordination among the developers or which affect more than one component are documented.

Deciding whether a design change is implied by some other change or spans a new coherent set of changes (with its own identification number) may be somewhat subjective and pose some difficulty in practice. In particular, when documenting changes to changes the question is similar to deciding whether fixing some bug is still part of one and the same cycle of coding and unit testing, or not. Fortunately, developers usually can tell with good confidence whether a change is an immediate consequence of some other change or is different. The amount of time which has passed between two changes plays a role here. Practical experience with the methodology will show whether properly organizing the design changes into coherent sets really poses a problem in practice.

Although documenting the design changes places some extra work on the developers, they will also benefit from the data. Systematically describing the changes they apply will help them to identify all consequences of their changes. The database also helps with communicating changes to the other developers and partially covers the usual requests for design and code documentation. The data collected has some overlap with the usual time data collected for accounting purposes.

The overhead caused by the data collection process can be minimized using a suitable tool. Such a tool might be integrated with version control. It should offer intelligent defaults and menues to the developer for some fields when creating a database entry. For example, the tool should "know" which components and modules currently are part of the software and offer a menue with a list of their names when a developer fills out the Changed Item field. The tool should also support intelligent search in the database.

To fix our ideas, we have developed two paper forms for data collection. One form documents the purpose and level of the design changes and the other form documents the activities and effort caused by the changes. We have used the forms during our university programming classes. Based on the experiences gained, we are currently merging the forms into a web-based tool.* The next steps will be to use the tool in industrial software development environments and to integrate the tool with version control systems.

References

1. Almeida, Lounis, Melo: "An Investigation on the Use of Machine Learned Models for Estimating Correction Costs", Proceedings ICSE 20 (1998) 473-476
2. Asundi, Kazman, Klein: "An Architectural Approach to Software Economic Modeling", EDSER-2 (2000)
3. Atkins, Ball, Graves, Mockus: "Using Version Control Data to Evaluate the Impact of Software Tools", Proceedings ICSE 21 (1999) 324-333
4. Basili, Briand, Condon, Kim, Melo, Valett: "Understanding and Predicting the Process of Software Maintenance Releases", Proceedings ICSE 18 (1996) 464-474
5. Basili, Condon, El Emam, Hendrick, Melo: "Characterizing and Modeling the Cost of Rework in a Library of Reusable Software Components", Proceedings ICSE 19 (1997) 282-291

* A prototype of the tool will be available at the time of the workshop.

6. Benlarbi, Melo: "Polymorphism Measures for Early Risk Prediction", Proceedings ICSE 21 (1999) 334-344

7. Binkley, Schach: "Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures", Proceedings ICSE 20 (1998) 452-455

8. Briand, Devanbu, Melo: "An Investigation into Coupling Measures for C++", Proceedings ICSE 19 (1997) 412-421

9. Briand, Wüst, Ikonomovski, Lounis: "Investigating Quality Factors in Object-Oriented Designs: An Industrial Case Study", Proceedings ICSE 21 (1999) 345-354

10. Bugzilla, `http://webglimpse.org/bugzilla/`

11. Gnats, `http://sources.redhat.com/gnats/`

12. Graves, Mockus: "Inferring Change Effort from Configuration Management Databases", Fifth International Symposium on Software Metrics (1998) 267-273

13. Jitterbug, `http://jitterbug.samba.org/`

14. Jorgensen: "Experience with the Accuracy of Software Maintenance Task Effort Prediction Models", IEEE TSE 21:8 (1995) 674-681

15. Kazman, Barbacci, Klein, Carriere, Woods: "Experience with Performing Architecture Tradeoff Analysis", Proceedings ICSE 21 (1999) 54-63

16. Padberg: "A Fresh Look at Cost Estimation, Process Models and Risk Analysis", EDSER-1 (1999)

17. Padberg: "A Probabilistic Model for Software Projects", Proceedings ESEC / FSE 7, Springer Lecture Notes in Computer Science 1687 (1999) 109-126

18. Padberg: "Linking Software Design with Business Requirements – Quantitatively", EDSER-2 (2000)

19. Padberg: "Estimating the Impact of the Programming Language on the Development Time of a Software Project", Proceedings International Software Development and Management Conference ISDM / AP-SEPG (2000) 287-298