Capturing the Long-Term Impact of Changes

Kim Sebastian Herzig
Saarland University
Saarbrücken, Germany
kim@cs.uni-saarland.de

ABSTRACT

Developers change source code to add new functionality, fix bugs, or refactor their code. Many of these changes have immediate impact on quality or stability. However, some impact of changes may become evident only in the long term. The goal of this thesis is to explore the *long-term impact of changes* by detecting *dependencies* between code changes and by measuring their influence on software quality, software maintainability, and development effort. Being able to identify the changes with the greatest long-term impact will strengthen our understanding of a project's history and thus shape future code changes and decisions.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*impact measures, effort measures, maintainability measures*

Keywords

Software engineering, metrics, long-term impact of changes, change dependencies

1. INTRODUCTION

Software development is an incremental process that uses earlier stages of the software product to build newer versions. During software development, source code is added, changed and removed. Directly after applying the code change, the software structure and execution behavior may have changed [9, 10]. Some code changes will also have an additional *long-term impact* on the software project that is difficult to estimate.

Consider the following example: Your development team maintains a plug-in based image processing framework; one of the image algorithms is hidden by design and has to be accessed using a wrapper object; to increase performance and accessibility, your team is about to declare the internal algorithm interface public. But what will be the consequences of this design decision? How will this change impact the quality and stability of this project in the long term? How much development effort is necessary to maintain the new member of the API?

ICSE '10, May 2-8 2010, Cape Town, South Africa

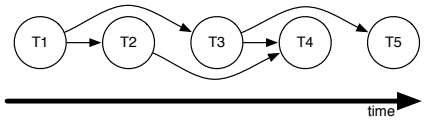

Figure 1: Sample TDG derived from method operations shown in Figure 2. An arrow $T1 \rightarrow T2$ indicates that $T2$ depends on $T1$.

Influencing the development process over a period of time implies having influence on later decisions. Thus, measuring the long-term impact requires recording or reconstructing the dependencies between code changes. The higher the number of later code changes that depend on T_i, the greater the long term impact of T_i. Figure 1 shows a sample dependency graph. The vertices T_1 to T_5 represent applied code changes, an edge between two vertices represents a dependency between the corresponding changes. The long-term impact of code change T_2 is small since only one of the three later applied code changes depends on it. Collecting quality and effort data for code change T_4 allows us to conclude about the impact of T_2 on quality and effort of the later development process.

The goal of this thesis is to detect and investigate dependencies between code changes. Similar to earlier approaches [2, 6], this thesis will use *transaction dependency graphs* (TDGs) to model cause-and-effect chains between sets of code changes and to analyze their impact on software quality, maintenance and development effort.

2. BACKGROUND

The difference between two (consecutive) source code revisions can be determined using a simple diff algorithm. But identifying the consequences of code changes requires more detailed analysis. Zimmermann [13] determined fine-grained differences between revisions on token level. Fluri et al. [5] extracted hierarchically structured changes on statement level along with change type information. Later, they used their own tool to describe development activities using change patterns derived from change type combinations [4]. On method level, Kim et al. [8] presented an approach that represents structural changes as a set of high-level change rules, automatically infers likely change rules, and determines method-level matches based on these rules.

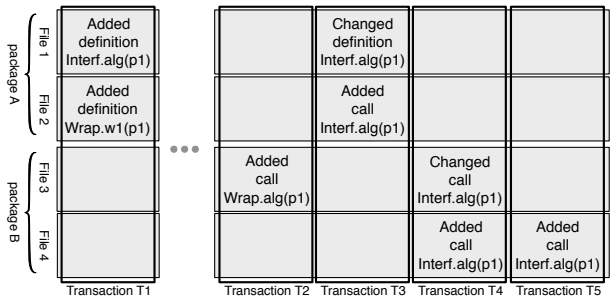

Figure 2: Sample method operation dependencies

Impact analysis techniques such as *CoverageImpact* [10] and *PathImpact* [9] determine the immediate impact of code changes on program executions. Ren et al. [11] presented a tool that decomposes program version differences into sets of atomic changes and reports test cases, whose execution behavior may have been changed. However, these approaches only consider the immediate impact of code changes on program structure and behavior.

Hassan and Holt [7] used source sticky notes to provide historical records of system structures and its evolution. Directly related to this thesis is the concept of *change genealogies* [2] and *change impact graphs* [6]. Both model the dependency structure between code changes in a directed acyclic graph. In contrast to change genealogy graphs [2], German et al. [6] extract time dependency information directly from revision repositories and lift time dependency information to source code entity level (e.g. classes). Later, Alam et al. [1] used change impact graphs to determine the fraction of new changes that are built on older, stable code.

3. CHANGE DEPENDENCIES

Many code changes applied to a software project are based on other code changes applied before. More precisely:

> *Code change C_2 depends on code change C_1 if and only if C_2 cannot be applied without applying C_1 before.*

Detecting all possible dependencies between atomic code changes on method level requires full type and cross-reference resolution over the complete project history. As an example: adding the statement `y = x + 10` depends on those changes that added the statements declaring the two variables `x` and `y`. It is desirable to have a dependency detection that is almost complete but also includes a large number of possible inter-change dependencies.

To ease the concept of code change dependencies, I separate code changes into sets of method definition and method call operations: added method definition (AD), changed method definition (CD), removed method definition (RD), added method call (AC), changed method call (CC) and removed method call (RC). This reduction of code changes will disregard many inter-change dependencies that will not add additional method calls or change any method interface. But concentrating on code changes that cross the natural method border implies focusing on changes that are more likely to have a wider impact. Additionally, the reduction eases the definition of code change dependencies dramatically.

CDs and *RDs* depend on transactions that previously added or changed the definition of the very same method. *AC*s, *CC*s and *RC*s depend on those transactions that added or changed the definition of the called method. Additionally, *CC*s and *RC*s depend on the changes that added the changed or deleted method call. Given the definition of code change dependencies and given type information based on source code [3], it is possible to detect and model code change dependencies.

Figure 2 shows a revision history that corresponds to the example given in the introduction. The change turning the interface public (T_3) depends on the previous definition of the interface in T_1. All new calls to the now public interface depend on T_3. So do all changes that replace old wrapper calls with new direct calls. At the same time these replacement changes also depend on the changes that added or changed the wrapper class. The set of changes depending on T_3 determines the set of development activities influenced by making the interface public and can be used to measure the impact on the later development process.

4. TRANSACTION DEPENDENCY GRAPH

Brudaru and Zeller [2] and German et al. [6] used directed, acyclic graphs to model code change dependencies. These graphs contain edges $A \rightarrow B$ if and only if change B can be applied only if change A was applied before.

Instead of modeling change dependencies on atomic code changes, *transaction dependency graphs* (TDGs) used in this thesis are based on dependencies between sets of simultaneously applied code changes: so called *transactions*. The underlying assumption is that code changes applied within the same transactions are dependent on each other. Thus, any such inter-transaction dependency will not refer to a long term impact. But lifting the granularity to transaction level reduces the size and complexity of the dependency graphs.

The TDG corresponding to our example from the introduction (see Figure 1) contains no edge from T_3 to itself, even though transaction T_2 applied two changes depending on each other (see Figure 2). TDG vertices are annotated with an exact description of all changes committed within the corresponding revision control transaction, the transaction id, the author and the timestamp of the commit. These annotations provide enough information to compute change dependencies within a transaction, if necessary. Each edge in a TDG represent a *temporal dependency* — dependency between two sets of code changes applied across multiple project revisions. Each edge is annotated with a *rationale* — the precise information which method definition or method call is responsible for adding this edge. If there are multiple methods responsible, the rationale comprises them all.

TDGs described in this thesis are based on Java source code and use change dependencies determined on method level using *partial program analysis* [3]. But adapting them to other object-oriented programming languages and to support more fine grained granularity levels is possible.

5. TRANSACTION IMPACT

The definition of an impact highly depends on the target domain (impact on what?). The goal of this thesis is to measure the long-term impact of code changes on the quality and maintainability of the software project as well as on development efforts. In particular, we raise the following

research question:

Q1 Is it possible to detect and analyze major design/development decisions using TDGs (e.g. refactorings and their effectiveness)?

Q2 Can we use TDGs to define and measure software development process metrics that will correlate with software quality and maintainability?

Q3 Can TDGs be used to extract development process models that will help to estimate future development efforts and to estimate the risk of future development decisions?

The answer to *Q3* highly depends on the answers to *Q1* and *Q2*. If TDGs cannot be used to detect major development process changes and if TDGs cannot be used to detect quality or stability issues, it will not be possible to extract useful decision models from TDGs. Since TDGs are too big to be analyzed manually, I will use software development process metrics based on the properties of TDGs to find an answer to *Q1* and *Q2*.

5.1 Dependency Metrics

InDegree and OutDegree

Similar to the code complexity metrics *FanIn*[1] and *FanOut*[2], the *InDegree*[3] and *OutDegree*[4] of a TDG vertex will indicate it's impact. Having a high number of outgoing edges indicates a high direct impact on many later changes. A high number of edges ending in a vertex means that this change depends on many other previous changes. Vertices having a high InDegree and a high OutDegree are expected to detect refactorings and thus may be useful to answer *Q1*.

Long-Term Impact

The relation between the number of dependent transactions and the number of transactions committed later determines the impact of a single transaction on the later development process. But different than the out degree of a vertex, it is necessary to consider indirect dependencies, too. Of course impact should decrease with increasing path length.

$$Impact(T_i) = \frac{\sum_{i \in depend(i)} \frac{1}{depth^2(i,j)}}{\#later(i)}$$

where *depend*(i) is the set of transactions dependent on T_i, *depth(i,j)* is the length of the shortest path from T_i to T_j, and *later*(i) is the set of all transactions committed later than T_i.

The impact of transaction T_3 from our example (see Figure 1) is one — all later applied transactions directly depend on it. In contrast, the impact measurement of T_1 is only three third since not all transactions depend directly on it.

[1]FanIn is the number of functions calling a given function.
[2]FanOut is the number of functions being called from a given function.
[3]InDegree is the number of directed edges pointing to a vertex.
[4]OutDegree is the number of directed edges leaving a vertex.

Long-Term Negative Impact

The above notion of impact does not state weather the impact was positive or negative. Code changes that have to be revised or undone later might have large impact but it is likely that a developer would consider such impact as negative. Measuring the fraction of dependent transactions whose dependencies are only based on code changes revised or undone later determine the negative impact of a transaction. Of course, not all development effort put in revised or undone changes can be classified as unnecessary or negative but the identification of "bad" transaction helps to identify and prevent future critical decisions and unstable code fragments.

Long-Term Impact on Quality

Quality refers to the number of bugs introduced by the code changes committed within a transaction. Sliwerski et al. [12] describe an approach that determines *fix-inducing changes* — changes that had to be revised in order to fix a bug. They map bug reports to revision transactions that contain those code changes to be applied in order to fix a bug. Those transactions that committed the lines to be fixed are identified as fix-inducing.

$$DefectRate(t_i) = \frac{\#fixind(t_i)}{\#delta(t_i)}$$

where *fixind*(t_i) is the set of fix inducing changes t_i committed and *delta*(t_i) is the set of code changes applied within t_i.

The quality of a transaction might influence the quality of later, dependent transactions, e.g. by changing return values or exception handling. The lower the quality of preceding transactions, the higher the defect risk. Vice versa, low quality of dependent transactions might indicate low quality of the common predecessor. Quality feedback might expose quality issues of transactions unknown so far. In order to find an answer to research question *Q2*, I want to investigate which such defect rates propagate along TDG edges and which factors of a transaction lower the defect risk.

Long-Term Impact on Stability

The more changes a code entity gets applied (in a specified period of time), the more unstable the code entity. Instability often refers to low quality and is often used as an indicator for quality. On code change level, a transaction T_i is unstable if the number of transactions applying code changes that revise changes applied within T_i is large. Similar to code entities, transactions whose code changes had to be revised are of poor quality. Unstable transactions with high impact are likely to cause further instability (on transaction and code entity level) and cause unnecessary development effort.

Graph Measurements

The above described impact measurements are determined by examination of single transactions. But they do not describe the overall graph status, such as the average quality or average impact over all transactions. Trends in transaction impact measurements captured over time can describe the evolution of the overall project health. For this purpose, all transaction impact measurements are averaged to the overall graph level considering a specific time frame (e.g.

two weeks). Limiting the considered time frame puts the average value into temporal context and prevents the average numbers to converge. The result is time lines that show transaction impact trends over the project's history. Using such trends it should be possible to identify effective refactorings (sudden drop in the average transaction impact) or software architecture decay (slow but steady increase in the average transaction impact).

Q1: To determine weather TDGs are suitable to detect and analyze major design and development decisions requires the prior knowledge of such decisions. Mining software archives such as version repositories, ticket systems and mailing lists, and using techniques like specification mining and reverse engineering makes it possible to extract such information about performed refactorings and major code changes. Knowing approximate time frames of major code changes, it will be possible to see if impact measurements such as *InDegree* and *OutDegree* will reflect the importance and impact of these development phases.

Q2: To determine the usefulness and predictive power of development process metrics, I will investigate weather these measurements will correlate with quality, stability and effort figures of the software project. If some of these metrics show significant correlation I will use these metrics to predict the correspondent system health measurement (e.g. bugs). Comparing prediction models based on TDG metrics with prediction models based on code metrics and mixed sets of metrics will show the predictive power of TDG measurements.

Q3: Similar to the evaluation of *Q2*, I need to show that derived change process models are capable of estimating future changes. For this purpose, I will split up the development history of a project into a two third training period and a one third testing period. Models extracted from the two third training period should be able to estimate risks and effort for changes applied in the test period. But this would imply that changes in both periods are similar to each other. If such models are really helpful for managers and developers to understand past changes and to learn from them would have to be evaluated in a large scale survey.

6. CONCLUSION

Constant changes in requirements induce constant code changes. It is more than natural that code changes depend on each other. Understanding the complexity of software development activities and being able to track back on which decisions my own code change is based, provides fundamental information that determines the quality of a change. The goal of this thesis is to determine the long-term impact of code changes on the software quality, maintainability, and development effort. *Transaction dependency graphs* model inter-transaction dependencies based on the level of method definitions and method calls. TDGs can be used to estimate and measure the long-term impact of changes. Long-term impact measurements might help improve automated recommendation system that warn the developer against risky changes.

Acknowledgments.

The concept of long-term impact of changes was originally sketched by Michael Godfrey, Andreas Zeller, and Thomas Zimmermann at the SARS 2007 workshop.

7. REFERENCES

[1] O. Alam, B. Adams, and A. E. Hassan. Measuring the progress of projects using the time dependence of code changes. *ICSM '09: Proceedings of the 25th IEEE International Conference on Software Maintenance*, 2009.

[2] I. I. Brudaru and A. Zeller. What is the long-term impact of changes? In *RSSE '08: Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*, pages 30–32, 2008.

[3] B. Dagenais and L. Hendren. Enabling static analysis for partial java programs. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 313–328, 2008.

[4] B. Fluri, E. Giger, and H. Gall. Discovering patterns of change types. In *ASE '08: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 463–466, 2008.

[5] B. Fluri, M. Würsch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[6] D. M. German, A. E. Hassan, and G. Robles. Change impact graphs: Determining the impact of prior code changes. *Information and Software Technology*, 51(10):1394–1408, 2009.

[7] A. Hassan and R. Holt. Using development history sticky notes to understand software architecture. In *IWPC '04: Proceedings of the 12th IEEE International Workshop on Program Comprehension*, page 183, 2004.

[8] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 333–343, 2007.

[9] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, 2003.

[10] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference*, pages 128–137, 2003.

[11] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. pages 432–448, 2004.

[12] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the Second International Workshop on Mining Software Repositories*, pages 1–5, 2005.

[13] T. Zimmermann. Fine-grained processing of cvs archives with apfel. In *eclipse '06: Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, pages 16–20, 2006.