# What is the Long-Term Impact of Changes?

**— Short Position Paper —**

Irina Ioana Brudaru
Saarland University, Saarbrücken, Germany
brudaru@cs.uni-sb.de

Andreas Zeller
Saarland University, Saarbrücken, Germany
zeller@cs.uni-sb.de

## ABSTRACT

During their life cycle, programs undergo many changes. Each of these changes may introduce new features—or new problems. While most of the impact of a change is immediate, some of the impact may become evident only in the long term. For instance, suppose we make the internals of a component accessible to its clients. In itself, this does not introduce a problem. In the long term, though, this will most likely lead to maintainability issues.

We are currently exploring ways to identify this long-term impact of change. We want to show how a change eventually impacts program quality (in terms of defects), program maintainability, and development effort. Identifying those changes with the greatest impact will foster our understanding of a program's history, and help us in learning lessons for future projects. Eventually, such lessons may come as automated recommendations regarding long-term impact: "In the long run, this change will cause maintainability issues. Do you want to reconsider?"

## 1. INTRODUCTION

During their life cycle, programs are subject to a constant stream of changes. Changes are made for a number of reasons: to adapt the program to a changing environment, to add new features, to fix bugs, or to enhance the general structure and maintainability. Besides the short-term purpose at hand, changes may also have long-term consequences—some intended, some unintended. Here are some examples of such a long-term impact of changes:

**Example 1.** Suppose that we are changing all the visibility modifiers of a class from private to public. This change in itself does not affect the program quality (or even semantics) in any way, but will likely cause higher coupling and therefore maintainability problems in the future.

**Example 2.** There is a current discussion about removing some code from the Big Kernel Lock (BKL) in the Linux OS that turned the BKL non-preemptible. Removing the BKL entirely is also not a good idea, since in 15 years of project history, a lot of dependencies have been created and removing it might lead to lower stability. Then, what is the long-term impact of adding and removing kernel locks? What is the direction the developers should take?

**Example 3.** Suppose a function *readline()* is introduced in a program. The method reads a line of input from the user and returns the input as a character string. However, if the user enters an end-of-file (Control-D) character, *readline()* returns a null pointer—a condition rarely checked and therefore a source of bugs. Should programmers have used *readline()* in the first place?

## 2. THE IMPACT OF CHANGE

To assess the long-term impact of change, one first needs to *measure* the impact—in terms of quality, effort, and maintainability. All of these features, however, come to be as *future changes:*

**Quality.** A defect eventually manifests itself as a bug fix. Therefore, we can assess the quality of a program by assessing its *fixes.* Likewise, new features also manifest themselves via changes—for instance, by having a new test added to the test suite. The nature of changes thus reflects the quality of the system.

**Effort.** Effort means programmer activity, which manifests itself in terms of *changes* to project artifacts.

**Maintainability.** The greater the extent of a single logical change, i.e. the more files are influenced by this change, the lower the modularity of the program—and therefore, the higher the effort it takes to change and maintain a system.

In all three cases, the long-term impact of a change can be measured by assessing future changes. Therefore, we need to come up with a *model* of how changes impact each other.

## 3. BUILDING CHANGE GENEALOGIES

We are currently exploring how to express the history of a project in a *genealogy of changes*—a directed acyclic graph in which an edge $A \rightarrow B$ means that the change $A$ enables (and thus impacts) a change $B$; the change $B$ thus depends on $A$. For instance, the change $A$ may make class internals accessible (example 1), introduce a kernel lock (example 2),


*RSSE '08,* November 10, Atlanta, Georgia, USA

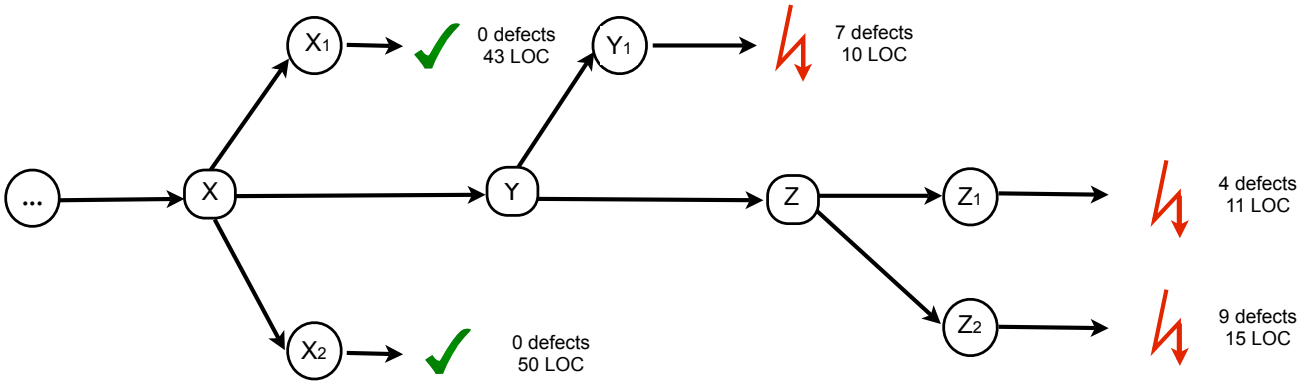


**Figure 1: A change genealogy graph. The change $Y$ enables the changes $Y_1$ and $Z$, which both lead to defects.**

or define the *readline()* function (example 3); the change $B$ accesses the class internals (example 1), uses the kernel lock (example 2), or uses the *readline()* function (example 3). Dependencies are *transitive*: that is, if change $B$ depends on change $A$ and change $C$ depends on change $B$, then change $C$ also depends on change $A$.

*A change genealogy graph* is a graph that contains the sequences of changes that take place in a project, as extracted from version archives. This is a graph which incorporates further dependencies between changes, derived from the changed code.

As an example of a genealogy graph, consider Figure 1. Here change $X$ enables $X_1$, $X_2$ and $Y$, $Y$ enables $Y_1$ and $Z$ and $Z$ enables $Z_1$ and $Z_2$. Some of these changes $(Y_1, Z_1, Z_2)$ are *bug-introducing*, that is, they introduce code that was later found to contain a defect (because it was fixed in a later change).

These changes, $(Y_1, Z_1, Z_2)$, were *made possible* by earlier changes: By following back the dependencies, we can find that change $Y$ enabled both "bad" changes $Y_1$ and $Z$. Change $Y$ thus is a change with a bad long-term impact—in contrast to, say, change $X$ which had a more positive long-term impact.

The first question we are currently working on is: *How do we know a change $B$ depends on change $A$?* We want to determine such dependencies from version archives and from code.

One straightforward, but computationally intensive way is to *experiment:* we try to build the project without change $A$, but with change $B$ applied. If the project cannot be constructed, then $B$ depends on $A$.

For projects with hundreds of thousands of changes, this becomes far too expensive to determine. We therefore need to come up with *heuristics* based on the definition and usage of individual program items.

Our first attempt in defining dependencies is by finding common data (variables, methods) that different changes access to modify or use. When such a connection is found between two changes, an edge in the genealogy graph is added, between the nodes that define the respective changes (their revision numbers). We further define the dependency concept between two changes.

**Definition. Dependency between two changes.** A change $A$ *enables* a change $B$ iff:

- Change $A$ has an earlier timestamp than change $B$,
- There exists a set of common identifiers that appear in the two changes, and
- The project containing the changes from $B$ and not from $A$, would not compile.

# 4. LEARNING FROM CHANGE IMPACT

Once we have extracted full-fledged genealogies from real systems, we need to develop *models* on how to compute which changes have the greatest impact on program features. For this, we are currently defining appropriate metrics for *maintainability*, *quality*, and *effort*—and setting up models to determine how these are impacted in the long term.

## 4.1 Evaluating Quality

If a particular change has issued a lot of code, one can say that this change has had a high contribution to the project. On the other hand it might also have lead to a high number of errors. One can measure the *quality score* as the average of the ratios between the number of defects and the amount of code introduced, on the path between the particular change we chose to look at, and the last descendants of this change, coming from the version history data of the project.

Let $A_1$, ..., $A_n$ be the changes that depend on change $A$, with $n > 0$. Let $QS(A)$ denote the quality score of the change $A$. Then, if change $A$ is not a final change in the change genealogy graph, its quality score is:

$$QS(A) = \frac{\sum_{i=1}^{n} QS(A_i)}{n}$$

If change $A$ is a final change in the change genealogy graph, then the quality score cannot be computed, simply

because we do not know about any future defects fixed. If a change is a change that induces a final change, its quality score is the ratio between the number of defects and the number of lines of code, meaning that the general formula above will be computed recursively. In Figure 1, the changes $X_1$, $X_1$, $Y_1$, $Z_1$ and $Z_2$ are annotated with the number of defects and lines of code. With this information we can compute recursively that:

$$QS(Z) = (QS(Z_1) + QS(Z_2))/2 = 0.48,$$

$$QS(Y) = (QS(Y_1) + QS(Z))/2 = 0.59,$$

$$QS(X) = (QS(Y) + QS(X_1) + QS(X_2))/3 = 0.19$$

from which we suspect that change $Y$ is a change with a bad long-term impact, in contrast to $X$ which has a more positive impact — some of its descendants issued no defects.

The lower the score of a change is, the higher its quality. Moreover, for each node on these paths, the ratios above or their partial averages can be computed. Moving from the last descendants of the change, back in time, towards the considered change, the values of the quality in each node will vary. By analyzing the variation of these numbers I hope to find a way to compute *tipping points*. In the neighborhood of a change that has induced a lot of problems (a tipping point), the quality numbers of the nodes will be high in comparison with changes further away from the bad change (see Figure 1: branches that bring 0 defects decrease the score, i.e. have a higher quality).

If a change leads to deleted code in the end, then it might be a clue that the particular code that was added in the change initially should have maybe never been added.

# 5. RECOMMENDATIONS

Once we know which changes are the ones with the greatest impact, the next step is to find out whether they would have specific characteristics to learn from—be it in the nature of the change, or the components affected. If we can determine appropriate characteristics, these may be helpful features to learn from—both by humans as well as by machine learners. As data collection is fully automatic, the recommendations would be produced automatically as well: "This change is risky, because in the past, similar changes have led to severe long-term maintenance issues."

To make such recommendations, we need to define a notion of *similarity* between changes. Useful features may include changes to types and accessibility (example 1); interaction between modules (example 2); or program structures and data flows (example 3). Large-scale evaluations will help establishing the most useful similarity features.

# 6. RELATED WORK

Maintenance of continuously evolving software systems is one of the most frequent activity performed by software developers. There are various approaches and tools that analyze the effects of the current programming task.

Hassan and Holt [2] have built a tool that attaches various code properties to static dependencies, naming these attached properties *Software Sticky Notes*. The static dependencies come from source control repositories. The tool assists developers who investigate dependencies in a large system to annotate the structure of the current system, providing a historical record of its evolution.

The *Chianti* [3] tool analyzes two versions of an application and decomposes their difference into a set of atomic changes, later reporting the change impact, in terms of affected regression or unit tests whose behavior may be modified by the performed changes.

*Change distilling* [1] finds fine-grained changes that occur across different versions of a project.

Hatari [4] highlights locations in the source code, indicating the level of risk for each part of code, by relating the version history data with a bug database for a particular project. The tool detects those locations where changes have been risky in the past versions and makes this visible to developers, also helping them by providing ways to analyze the risk history.

# 7. CONCLUSION

Most of us have been involved in projects where, with hindsight, a great deal of pain could have been avoided with a little more up-front assessment of the consequences. With the present project, we want to provide some of this hindsight—and with a little luck, turn it into foresight. We look forward to fruitful discussions at the RSSE workshop.

**Acknowledgments.** The concept of long-term impact of changes was originally sketched by Michael Godfrey, Andreas Zeller, and Tom Zimmermann at the SARS 2007 workshop. Rahul Premraj provided helpful feedback on earlier revisions of this paper.

# 8. REFERENCES

[1] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, 2007.

[2] A. Hassan and R. Holt. Using development history sticky notes to understand software architecture, 2004.

[3] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. *SIGPLAN Not.*, 39(10):432–448, 2004.

[4] J. Śliwerski, T. Zimmermann, and A. Zeller. Hatari: raising risk awareness. In *ESEC/FSE-13*, pages 107–110, New York, NY, USA, 2005. ACM.