

# API Evolution: A Study of Software Releases and Change Trends Leading to Stability

Jordan Ell  
University of Victoria  
British Columbia, Canada  
jell@uvic.ca

Braden Simpson  
University of Victoria  
British Columbia, Canada  
braden@uvic.ca

Daniela Damian  
University of Victoria  
British Columbia, Canada  
danielad@csc.uvic.ca

## Abstract—Something

### I. INTRODUCTION

Intorduction goes here.

### II. RELATED WORK

Related work goes here.

### III. METHODOLOGY

In order to answer our research question, we decided to use the tool ChangeDistiller created by Fluri et al. [1]. This tool allows us to detect fine grained source code changes in Java projects This tool works by building an abstract syntax tree of a file before and after a code change, then it tries to determine the smallest possible edit distance between the tree. This results in the source code change at a fine grained level performed in the commit.

We took ChangeDistiller and applied it across 10 open source Java projects. For each of the projects, we obtained the software configuration management (SCM) system which is used to store all source code changes of a project. When it was necessary, we converted some forms of SCM system to Git in order to reduce implementation burdens of using multiple SCMs. Once the SCM was obtained, we used ChangeDistiller and iterated over every commit found in a project's git master branch. We stored 34 of ChangeDistiller's built in source code change types for each commit. We noted how many of each change type was performed in each commit and stored that information in a PostgreSQL database. In order to filter and protect our results, we manually inspected the 10 Java projects studied in order to identify code built for test. We separated changes to this test code from all other code to ensure our results only focused on real implementation while allowing us to study changes to test based code separately.

Once the ChangeDistiller information was collected, we decided to examine software change trends surrounding releases of the project's we had selected. Since releases have preconceived notions of software stability, we decided that by studying the types of changes surrounding these releases, we could get a better understanding of what types of source code changes or trends constitute software stability or maturity. In order to study the release points, we went to each of our 10 project's home pages on-line and looked through their release

histories for major, minor, alpha, beta, and release candidate type releases. In total we identified 472 releases across our 10 studied projects.

Once the release dates were collected we set about analyzing out data by creating average change ratios surrounding the release dates of each project as a way to measure the trend of a particular change type at a release type. To do this we used Equation 1. Equation 1 works to create a change ratio by first creating a numerator by summing across all releases of a given release type a sum of a particular change type in commits ( $T_c$ ) from the release date ( $r$ ) to a given number of days after the release ( $d$ ) divided by the number of commits in this date range ( $|c|$ ). Next the denominator is created by summing across all release of a given release type a sum of that same particular change type in commits( $T_c$ ) from a given number of days ( $d$ ) before the release date ( $r$ ) to the release date divided by divided by the number of commits in this date range ( $|c|$ ). This numerator and denominator form the final change ratio. This equation gives us a ratio of a particular change type happening before and after a particular release. If the ratio is above 1 then that particular change type occurs more frequently after the release and if it is below 1 then it occurs more frequently before the release. For the purposes of our study, we set the number of days before and after the release ( $d$ ) to 60 as the projects studies had many months in between their major releases. This quantitative data formed much of the basis for the results to come in Section IV

$$\text{ChangeRatio} = \frac{\sum_{r_0}^{r_n} \sum_{c=r}^{r+d} T_c / |c|}{\sum_{r_0}^{r_n} \sum_{c=r-d}^r T_c / |c|} \quad (1)$$

Aside from generating quantitative data, we also created a web application for the visualization of the data called API Evolution (APIE). This visualizer allowed us to inspect a single project and a single change type metric at a time (see Figure 1) for qualitative analysis of software evolution trends. We used this tool to manually inspect 4 specific change type trends surround release dates. To do this, we aggregated change types across 50 commits, meaning that each point in the graph represented the date of a commit and the sum of the particular change type's occurrences over the last 50 commits. This was used to smooth out the curves presented by the tool to allow easier manual inspection. Manual inspections were

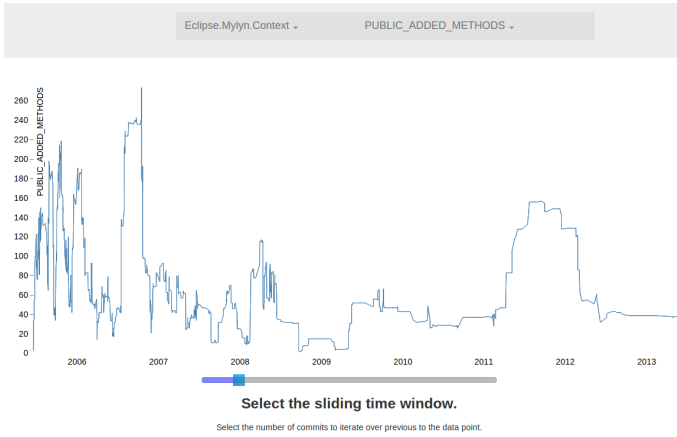


Fig. 1. An screen shot of the APIE visualizer showing project Eclipse.Mylyn.Context with change type PUBLIC\_ADDED\_METHODS being analyzed.

labeled into 4 categories: upward trending, local maximum, downward trending, and local minimum. Since the graphs were quite turbulent, best estimations were given by two judges at each release point to fit the graph into the aforementioned 4 slope categories. The two judges used 1.5 months before and after the release date as start and end points for the graph trend line.

We performed 1888 manual inspections across 10 projects, 472 release dates and 4 change types, and used this data to form the basis of our qualitative data. Quantitative data was cross validated with qualitative data to form some of the final results to be seen in the next section.

#### IV. RESULTS

To answer our research question, we conducted a case study of 10 open source Java projects. These projects are: eclipse.jdt.core, eclipse.jdt.ui, eclipse.jetty.project, eclipse.mylyn.common, eclipse.mylyn.context, hibernate-orm, hibernate-ogm, hibernate-search, eclipse.maven.core, and eclipse.maven.surefire. These project were chosen because of their high use amongst other Java projects and to study specific ecosystems of projects and their evolution trends.

Due to the space requirements of this paper, we focus our results on Major releases of the 10 case study projects and a select few of the calculated change ratios. There were 109 major releases across the 10 studied projects. All of the major findings as per values computed from Equation 1 for non test metrics can be seen in Table I.

As it can be seen in Table I, there a few notable change type trends around major releases. We can see that any changes to public classes are more likely to occur before a major release than after, noting that the addition and deletion of public classes have change ratios of 1.14, and 1.16 respectively. This means that on average classes are added and removed 10% more often after major releases than before. The two most drastic change type trends to note from this table however are the adding and removing of private classes. We can see that

TABLE I: Implementation oriented change types and their normalized average change ratios at 60 days on each side of releases.

Object	Added	Changed	Removed
Public Classes	1.14	0.86	1.16
Public Methods (Signature)	1.07	0.92	1.34
Public Methods (Bodies)	-	1.06	-
Private Classes	0.81	1.18	1.44
Private Methods (Signatures)	1.00	1.10	1.22
Private Methods (Bodies)	-	1.08	-
Files	1.12	0.96	1.14
Documentation	-	0.99	-

private classes are 44% more likely to be added after a major release and that adding private classes to a project is roughly 24% more likely to occur before a major release. All results in Table I could be used as identified trends of major software releases, while we have just highlighted the larger ratios in this paper.

Some of the more interesting trends found around major release points from Table I are private methods being removed and public method signatures being changed. We hypothesized that public method signatures being changed would be heavily favored towards before a major release indicating a larger sens of stability in the public API, however we found that only a change ratio of 0.92 exists meaning that public method signatures undergo changed almost through the project steadily. The private methods being removed is also suggestive of possible maintenance being done post release, meaning that code is being cleaned up in order to reduce the number of overall methods in a project which are not publicly available.

Our qualitative results from manual graph inspections can be seen in Table II. These results show that adding, changing signatures and bodies of, and removing public methods tend to all be at a local minimum of change type trends at major releases.

Lastly we found that software changes related to testing can be highly used as an indicator of a major release point within the projects studied. The change ratios found can be seen in Table III. As it can be seen, the three largest indicators of a project release and stability with regards to test based changes are the removal of test classes, the changing of method signatures (both occurring more before releases), and test classes being changed. An interesting side note is that documentation is changed heavily before releases as opposed to after.

TABLE III: Test oriented change types and their normalized average change ratios at 60 days on each side of releases.

Object	Added	Changed	Removed
Classes	1.07	1.21	0.76
Methods (Signatures)	1.23	0.83	1.01
Methods (Bodies)	-	0.90	-
Documentation	-	0.72	-

TABLE II: Qualitative graph analysis results.

Change Type	Upward Trend	Local Maximum	Downward Trend	Local Minimum
Added Public Methods	21.6%	17.2%	14.7%	33.6%
Changed Public Methods (Signature)	6.0%	19.8%	19.0%	39.7%
Changed Public Methods (Bodies)	9.2%	16.5%	26.6%	37.6%
Removed Public Methods	7.8%	16.4%	12.9%	41.4%

We summarize our key findings as to source code change type trends which can be found at points of stability as the following: the addition of private classes more often before a release point, the removal of private classes more often after a release point, the removal of public methods more often after a release point, the removal of private methods more often after a release point, the removal of test classes more often after a release point, and the changing of test method signatures more often before a release point. While all change ratios may need to be considered for a final analysis or taxonomy, we have offered the strongest change trends.

## V. FUTURE WORK

In future work of change trend analysis, we have planned further statistical tests of the change trends identified in this paper. We would like to be able to determine if these patterns are unique to major releases versus other types of releases (minor, alpha, beta, release candidate) and to be able to determine if these pattern are unique to releases alone or can be spotted throughout the history of a project.

If it is deemed that these change trends are present in no other releases, some, all, or in non release history of the project, we will compare the history of the projects as to where these trends are found against preexisting software stability and quality metrics. We will use this data to create a new method of discovering when software can be considered stable in reference to the types of software changes being made.

## VI. CONCLUSIONS

Software stability is an often used measure in determining some aspects of code quality. This paper has present some initial indications as to how software stability may be found, based on major release stabilities, by analyzing source code change type trends both before and after major software releases. The source code change trends found in this paper, in combination with future work and analysis, will allow for the detection of stable code points throughout a software projects life time. This paper has also shown the beginnings of a visualization for source code change trends which may be used as a visual cue towards project stability and potential areas of instability where action may need to be taken.

## REFERENCES

- [1] B. Fluri, M. Wuersch, M. Plnzer, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007.