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*—As software evolves over the lifetime of a project, changes to software objects such as methods or classes occur which can have ripple effects throughout the rest of the system causing relationships to be re-engineered and causing instability. However, projects are sometimes deemed stable at certain release points by projects owners or maintainers. We know that studying a release history of a project can give clues as to a project's history, release process, and how processes change, but can it shed light on what causes a project to be released and deemed stable? This paper presents our findings from studying source code change trends surrounding 109 major release points of 10 open source projects. We found 6 large source code change trends at these release points that can be used to determine if a project is likely near a major release point and thus deemed stable.**

I. Introduction

Release points are a vital milestone of software projects. From major releases of a Waterfall based project to minor iterations of an Agile development, releases form an interesting single point of a project's development history. Third party users of a system often only see a product at a release point either major or minor, and expect the system to come with a sense of reliability and stability at this point. However, the decision as to when a project is ready for public usage as to its reliability, quality and stability can be a difficult decision to make for most project owners or maintainers.

While measuring software quality has had a major focus in software engineering research for many years [1] [2] [3], the sub study of software stability and its implications on quality and reliability remains a difficult subject to understand. The decision of what makes a project stable and ready for a release often comes down to the release manager or maintainer of a project and is often a reflection of the open source community which surrounds the project [4]. Code is an often looked to statistic for stability but can be grossly misleading in terms of pre-release and post-release defects [5], with some exceptions [6]. Creating a different quantitative approach to determining software stability and release preparedness is still a large open area of interest in software engineering research.

We turn our analysis to the notions of software change trends, specifically those trends around major releases. Change trends have been used to detect stability in core architecture [7] as well as evolving dependencies [8]. With the power of major release points in open source projects as a starting point for project stability and the understanding that change trends can be leveraged to detect stability and the evolution of source code, the question we investigate in this paper is: "*Is it possible to identify trends in source code changes surrounding major releases of open source projects as a notion towards a project stability measure?*". By answering this question, we hope to provide industry with the ability to determine a project's stability and perhaps reliability based on its current change trends. We also hope to provide researchers with the ability to perform further analysis of projects in regards to software quality with the findings of this paper.

In this paper, we perform a case study of 10 open source projects in order to study their source code change trends surrounding major release points throughout their history. We studied 26 quantitative and 16 qualitative change trends and identified a core group of 6 change trends which occur prominently at major release points of the projects studied.

The remainder of this paper is laid out as follows. Section II goes over the related work of release study and stability. Sections III explains our methodology and tools used in solving our research problem. Section IV gives a full representation of the results and trends found as well as a summarized results of the 6 change trends found to be most prominent. Finally, Sections V and VI give our future work to follow this study and final conclusions respectively.

II. Related Work

While very little has been published about release quality studies and stability, there have been a few studies which attempt to address the issues directly or indirectly. Wu et al. [9] performed a case study of SoftPM, a widely adopted project management tool, to explore the relationships of pre-release and post-release failures at major releases. Wu et al. found that the ratio of post-release failures to pre-release failures is significantly low and can be used to show reliability and stability. Hindle et al. [10] performed a case study on MySQL which observed a the project's behavior around major and minor release by monitoring artifact check-ins and changes. They found that there are temporary stoppages for source revisions around releases, indicating that a temporary freeze is taking place for developers and that last minute fixes and manual testing may be being performed. Zaidman et al. [11], in comparison, studied the co-evolution of production and

test code with inspections and analysis at major and minor releases.

The study of open source projects revolving around release points has become more available by the work of Tsay et al [12]. Tsay et al. created a resource of historical release dates for open source software projects to be used for future studies by other researchers.

In terms of software stability, many proposed development techniques have been proposed to increase software stability. Fayad [13] [14] suggests that "business objects" (BOs) do not change in nature and that they are inherently stable. These objects only need to change to accommodate external modules at the interface. Some studies such as Chow et al. [15] have investigated the stability of changes to interfaces which are considered a good indication of stability. Mockus et al. [16] used major and minor release points to compare industry process quality to customer-perceived quality of the software project. Mockus et al. found that defect density is lowest at major releases but at the same time software quality is at its lowest all when compared to minor releases. The low software quality here related to end-user errors of installations and configurations. Wermelinger et al. [7] showed that stable core architectures can be detected by using source code changes. Finally, Fayad et al. [17] have investigated the Software Stability Model (SSM) for Software Product Lines to show that the SSM's impact on architecture and design of a software product can help improve the life of the product line and make it more adaptable and applicable.

III. Methodology

In order to answer our research question, we decided to use the tool ChangeDistiller created by Fluri et al. [18]. 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 is 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

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

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 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

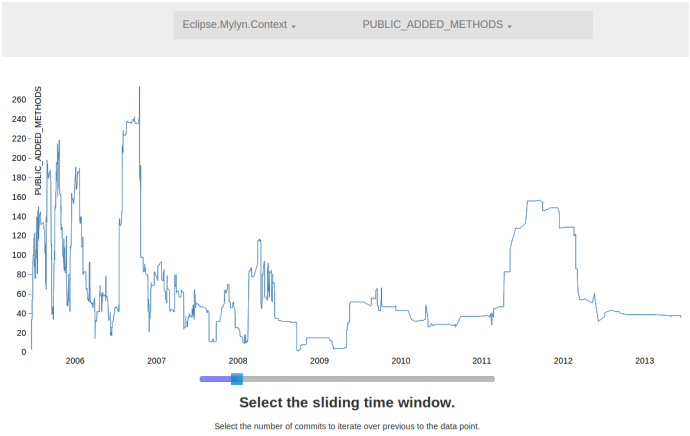

Fig. 1. An screen shot of the APIE visualizer showing project Eclipse.Mylyn.Context with change type PUBLIC_ADDED_METHODS being analyzed.

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.commons, 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.

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	-

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 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	-

We summarize our key findings as to these 6 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

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%

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] J. B. Bowen, "Are current approaches sufficient for measuring software quality?" in *Proceedings of the software quality assurance workshop on Functional and performance issues*. New York, NY, USA: ACM, 1978, pp. 148–155.

[2] R. B. Grady, "Practical results from measuring software quality," *Commun. ACM*, vol. 36, no. 11, pp. 62–68, Nov. 1993.

[3] ISO/IEC, *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[4] M. Conway, "How do committees invent," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[5] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, pp. 797–814, Aug. 2000. [Online]. Available: http://dx.doi.org/10.1109/32.879815

[6] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 284–292. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062514

[7] M. Wermelinger and Y. Yu, "Analyzing the evolution of eclipse plugins," in *Proceedings of the 2008 international working conference on Mining software repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 133–136.

[8] J. Businge, A. Serebrenik, and M. van den Brand, "An empirical study of the evolution of eclipse third-party plug-ins," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, ser. IWPSE-EVOL '10. New York, NY, USA: ACM, 2010, pp. 63–72.

[9] S. Wu, Q. Wang, and Y. Yang, "Quantitative analysis of faults and failures with multiple releases of softpm," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ser. ESEM '08. New York, NY, USA: ACM, 2008, pp. 198–205.

[10] A. Hindle, M. W. Godfrey, and R. C. Holt, "Release pattern discovery via partitioning: Methodology and case study," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 19–.

[11] A. Zaidman, B. Rompaey, A. Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Softw. Engg.*, vol. 16, no. 3, pp. 325–364, Jun. 2011.

[12] J. Tsay, H. K. Wright, and D. E. Perry, "Experiences mining open source release histories," in *Proceedings of the 2011 International Conference on Software and Systems Process*, ser. ICSSP '11. New York, NY, USA: ACM, 2011, pp. 208–212.

[13] M. E. Fayad and A. Altman, "Thinking objectively: an introduction to software stability," *Commun. ACM*, vol. 44, no. 9, pp. 95–, Sep. 2001. [Online]. Available: http://doi.acm.org/10.1145/383694.383713

[14] M. Fayad, "Accomplishing software stability," *Commun. ACM*, vol. 45, no. 1, pp. 111–115, Jan. 2002. [Online]. Available: http://doi.acm.org/10.1145/502269.502308

[15] J. Chow and E. Tempero, "Stability of java interfaces: a preliminary investigation," in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*, ser. WETSoM '11. New York, NY, USA: ACM, 2011, pp. 38–44.

[16] A. Mockus and D. Weiss, "Interval quality: relating customer-perceived quality to process quality," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 723–732.

[17] M. E. Fayad and S. K. Singh, "Software stability model: software product line engineering overhauled," in *Proceedings of the 2010 Workshop on Knowledge-Oriented Product Line Engineering*, ser. KOPLE '10. New York, NY, USA: ACM, 2010, pp. 4:1–4:4.

[18] B. Fluri, M. Wuersch, M. PInzger, 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.