Release Pattern Discovery via Partitioning: Methodology and Case Study

Abram Hindle, Michael W. Godfrey, Richard C. Holt
University of Waterloo
{ahindle,migod,holt}@cs.uwaterloo.ca

Abstract

The development of Open Source systems produces a variety of software artifacts such as source code, version control records, bug reports, and email discussions. Since the development is distributed across different tool environments and developer practices, any analysis of project behavior must be inferred from whatever common artifacts happen to be available. In this paper, we propose an approach to characterizing a project's behavior around the time of major and minor releases; we do this by partitioning the observed activities, such as artifact check-ins, around the dates of major and minor releases, and then look for recognizable patterns. We validate this approach by means of a case study on the MySQL database system; in this case study, we found patterns which suggested MySQL was behaving consistently within itself. These patterns included testing and documenting that took place more before a release than after and that the rate of source code changes dipped around release time.

1. Introduction

This paper describes an approach to analyzing open source process information using the available version control system artifacts. In particular, we are interested in release-to-release iterations of development, and so we examine the behavior of a project around the points of major and minor releases.

The research questions we are investigating concern process extraction: How can we infer details of the development process from the available artifacts, such as the revisions stored in version control? And how much of this can be done automatically and reliably?

Automatic process extraction would allow managers to audit the behaviors of their developers, to inform developers what their actual process looks like, and to educate new employees. For example, if developers are supposed to be following an eXtreme Programming approach, process extraction could help ensure that programmers were following a test-first methodology, and measure how often tests are created or modified when existing code is re-factored.

However, automatic process extraction is a large and difficult problem, so we have limited our scope to the analysis of the behavior of projects around the time of release. A release often demarcates the end of one iteration from the beginning of the next, and so is a natural place to focus our studies on.

In this paper we will propose an approach to analyzing the behavior of a project around release time. Then we will apply this approach in a case study of MySQL's behavior around major and minor releases. Our approach is related to much previous work in the field of software evolution, software life-cycles and processes extraction.

1.1. Background

Over the years the software engineering discipline has proposed various models for how to develop software, such as the Waterfall model [8] and iterative approaches such as the spiral model [1] and OMG's Unified Process [4]. Most of these processes have their roots in classical engineering. Having a defined process encourages repeatable results and performance; in time, processes can be measured and then optimized, as per the SEI's Capability Maturity Model (CMM) [9] aims.

Software development processes — also called software development life cycle (SDLC) models — relate directly to the idea of *software evolution*, which is the study of how software changes over time [7]. SDLC models attempt to tell us how software should be made. Software evolution tells us how it was made. Software evolution is also concerned with study and the measurement of change. If one is to comprehend a system that changes over time one should consider how the measurements of the system change. Some software evolution metrics measure systems before and after a change, as well as measuring change itself [5, 6, 3]. Our problem is: given the repository of a project how do we reverse-engineer the software development process from this data?

In the field of process discovery, the study of what software development processes are being used and how practitioners create software, Cook has described frameworks for event based process data analysis [2]. Our work differs from Cook's; instead of attempting to insert sensors and monitors into the development process as Cook did, we analyze the data available to us and attempt to determine what happened in the past. This is done by analyzing fine-grained changes to version control systems (VCS), such as CVS [10].

Process extraction is important because it can help determine the behavior of the programmers and their project as well as the context of their revisions. This information would be useful to new maintainers or developers joining a project. As well it would be useful as a post-mortem tool to determine which processes were successful. Cook used process extraction to verify if programmers were following their prescribed model, as the programmers worked on the project. We want to recover the project's behavior from the version control repository data. In order to do so we first need to agree on some basic terminology.

1.2. Terminology

In this section we will define some terms which we will use throughout the rest of the paper. *Version control systems* (VCS) are repository-like systems such as CVS or BitKeeper which control and store versions of a project's source code. They record changes to files, these changes are called *revisions* which are *committed* to the VCS. We consider a *commit* to be the act of submitting a set of revisions to the version control system.

A *release* is a set of *revisions* that are bundled together and then distributed as files to end-users, or simply represent the state of the software at the end of an iteration. A release occurs on the day when a new version of the software is officially packaged for distribution. Releases are found via tags with in the VCS, via notes in the project changelog and even from the release packages on the project's FTP server. We distinguish between two main kinds of releases: *major releases* and *minor releases*.

A *major release* indicates that there has been a substantial change in the software, such as the change from Linux kernel 2.4 to 2.6 or from MS Windows 2000 to MS Windows XP.

A minor release indicates that less significant changes have occurred from the previous release, such as Linux kernel 2.4.23 to 2.4.24 or from MS Windows XP SP1 to MS Windows XP SP2.

We note that the criteria used to distinguish between major and minor releases depends on the project. If we refer to *all releases*, this means that both major and minor releases are included. A *release revision* is a revision made during an interval around a release. If we are looking at a release with an interval of a week before and after, that means any revision that occurred within seven days before or after that release is a release revision. *Before release* refers to an interval of time immediately before a release. *After release* refers to an interval immediately after a release. It is important not to confuse *releases* and *revisions*, revisions are fine grained changes while releases are collections of revisions prepared for distribution.

For our analysis we *partition* the files in the version control system into four classes: *source*, *test*, *build*, and *documentation*. A revision belongs to a *revision class* based on which class that their file is associated with:

Source revisions are revisions to source code files. Source code files are identified by the file name suffixes such as `.c, .C, .cpp, .h, .m, .ml, .java,` etc. Note that source files might include files which are also used for testing.

Test revisions are revisions to files that are used for testing the project. Test files include regression tests, unit tests, and other tests that may be added to the repository. Revisions to files that are part of regression test and unit test cases are considered to be test revisions. Generally, any file that has `test` in its name is assumed to be a test file (although there are obvious exceptions).

Build revisions are revisions to build files such as those related with GNU Autotools (make, configure, automake, etc) and other build utilities. Build files include files with names such as configure, Makefile, automake, config.status, or suffixes such as `.m4`, etc.

Documentation revisions are revisions to documentation files, which include files such as README, INSTALL, doxygen files, API documents, and manuals.

A *release pattern* is a behavior that occurs before or after a release. A release pattern includes behaviors such as increased frequency of documentation revisions before a release which drop off after the release, or even the frequency of test revisions maintaining a constant rate during the release. These patterns are primarily found by analyzing a project's release revisions (the revisions associated with the releases of the project). Sometimes we analyze these patterns using aggregate functions like a *window* function, which is a function which takes an interval of values as input and produces an output. For instance, per each day we could sum up all the revisions during the preceding week, this would be a *window* function.

Using our 4 revision classes we can extract and analyze the project's behavior and identify release patterns. We go into further detail on the steps taken to analyze a project in the next section (section 2). These concepts we discussed in this section are consistently used throughout the rest of the paper and the methodology.

2. Methodology

This section presents our methodology for analyzing release patterns of a project; we will present the steps involved and then we will followup with an application of our methodology in a case study (section 2.5). Our methodology relies heavily on the revision classes we discussed in the previous section 1.2.

Our methodology can be summarized as: Extracting data for revisions and releases (section 2.1); Partitioning the revisions (section 2.2); Grouping revisions by aggregation and windowing (section 2.3); Producing plots and tables (section 2.4); Analyzing summaries of the results (section 2.5).

2.1. Extracting Data for Revisions and Releases

Firstly we choose a target project's VCS and either mirror the repository or download each revision individually. From VCS's such as CVS or BitKeeper we extract the revisions and sometimes release information. We will later analyze this extracted data. Per each revision the minimal information extracted includes the date of revision, the name of the revised file and the author of the revision. In section 3.2 we discuss that we use such extraction tools as `softChange` for CVS repositories and `bt2csv` for BitKeeper repositories.

The release is essentially a record of the time of when the project was distributed or packaged for distribution. This time is determined from the version control system tags, project changelogs, manuals, and even the release datestamps found in the project's FTP repository. Once extraction is complete we are ready to partition our revisions into classes.

2.2. Partitioning the Revisions

Once we have extracted the data, we partition the set of revised files into these four classes: source code, tests, build scripts and documentation. The revisions are partitioned based on how their files are partitioned. In principle, these are disjoint classes, but in the work presented here, there is some overlap between source code and tests.

We partition the revisions into their respective classes mostly by suffix and if their names match. Usually test files are classified as test because they are in a test directory, they have "test" in their pathname, or they have a test related prefix or suffix. Documentation files are determined much the same way. Suffixes help determine source files and build files. One should audit the matched files and determine which ones truly belong to each class as there are sometimes false positives. If revisions are duplicated between branches we will evaluate each duplicate as a separate revision.

2.3. Choose and Apply Aggregate and Window Functions

Our revision data is often quite variant and somewhat messy to plot, to get a clearer picture of the trends involved one often needs to aggregate or smooth the results. For instance, revision frequency data is often exponentially distributed, meaning that points are highly variable and there are lot of points that look like outliers but are a normal part of development (see the distributions of the revision classes of MySQL 5.1 from our case study in figure 2).

This means that a time line plot of the revisions is highly variant and often messy to plot, this implies that we need smoothing to help make trends more apparent to the eye so that we can investigate and validate if those trends truly exist in the data. Aggregate functions such as summation over an interval, summation over a window, average over a window, all smooth out the data and make trends more visible. For an example of a smoothed plot see the figure 1 from the MySQL case study.

We have a choice between aggregating aggregates, averaging aggregates or combining all the revisions and aggregating them together. An example of this is to group all of the revisions that occurred 1 week before a release together and analyze those results, the alternative would be to analyze each release independent of each other and then aggregate the independent results.

2.4. Plot and Analyze

We use graphs, plots and tables to help us understand the release patterns of the project. Here is a key question that we would like to answer: *For each class of file, does the frequency of revision increase (or decrease) preceding (or following) the time of the revision?*

To try to answer a question such as this, we plot the frequency of revisions in a revision class leading up to and following the release. We prepare our data, as suggested in the previous section, by aggregating by a time period such as hours or days. We then would compare the average of the number of revisions in each period. We would do this multiple times with different values for parameters such as interval length or release type. We can generate plots of the revision frequencies and linear regressions of these frequencies. We can also make tables showing how the results change when parameters like interval length change. If we have too many results we might want to compound these results by aggregation functions like majority voting or averages to make them more readable and easily analyzable.

In our experiments with MySQL, as described in our case study, we varied the length of the interval from 7 days up to 42 days. We carried out linear regressions on the aggregated frequencies for the before release and after release intervals to help identify release patterns.

Given the previous steps in our methodology and given the resulting plots and tables, the final step is to analyze these tables and plots to help us understand the release patterns. Our larger goal is to gain insight about the process of software change. As we have mentioned this often requires us to summarize the behavior of attributes of graphs and tables such that we can make general claims about behavior in verifiable and definitive manner. We developed a summary notation which we call STBD Notation, described below in section 2.5, to help summarize trends of revisions before, after and during releases.

2.5. STBD Notation

STBD Notation is a short form summary of the results of a query much like Myers-Briggs Type Indicator (MBTI). MBTIs are short form summaries of a person's preferences and what kind of personality they perceive they have, example summaries of personalities include INTJ and INTP (both are introverts but one is judgmental and the other perceives). These short forms show the preference of the individual for each of the four dichotomies which are represented positionally in order by single characters: Extroversion and Introversion (E/I), Sensing and intuition (S/N), Thinking and Feeling (T/F), Judging and Perceiving (J/P).

STBD Notation is similar to MTBI, but is meant to summarize comparisons and representations of values related to revision classes. An example instance of STBD Notation for comparing the average frequency of release revisions would be `S+T-B-D-`; a '+' would indicate that revisions were more frequent before a release and a '-' would indicate revisions were more frequent after a release. Example values represented include comparisons of the average frequency of revisions before and after a release. We assign a letter to each file class (S for source, T for test, B for build, D for documentation). We order the class characters from most frequent class to least frequent class: S, T, B, D.

The format of the summary is `S*T*B*D*` where `*` could be '+', '-', '=' or '?'. In the case of comparing the average frequency of release revisions (as we just previously described), '=' would indicate the averages were very close or the same and '?' would indicate we had no data. Of course these four characters can be arbitrarily assigned depending on the metric. The symbols '+', '-' and '=' are particularly good at describing the direction of the slope for a given interval.

We repeat the letters just to aid deciphering the summary in case someone has forgotten the order. Alternatively the letter prefix allows us to pick and choose what information we want to show. For instance if we only care about change we can omit those classes which did not change.

Project	Source	Test	Build	Doc
MySQL 3.23	4220	1410	421	21
MySQL 4.0	11593	4936	1033	34
MySQL 4.1	31451	16430	2990	88
MySQL 5.0	45946	26373	3908	105
MySQL 5.1	52897	31389	4772	122

Table 1. Total Number of Revisions per class

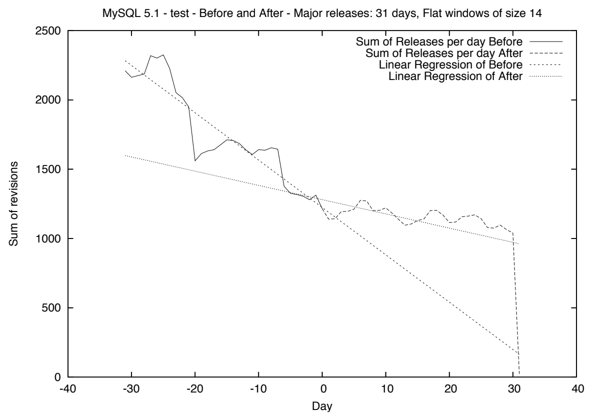

Figure 1. Windowed plot of Test revisions

Example metrics and classes of metrics one could use with this notation include: linear regression slopes, average LOC per revision, average frequency of revision, relative comparison of frequencies, the sign of a metric, concavity of quadratic regression, etc.

An example case would be if we measured the number of lines changed per class then compared the number before a release and after a release. We could use '+' to mean if more lines changed before the release, and we could use '-' if more lines changed after the release. If there were more source lines before, more test lines before, more build lines after and equal documentation lines we'd get a STBD Notation value of `S+T+B-D=`.

In the next section we use our methodology of extracting, partitioning, plotting, and analyzing on MySQL. Our analysis of MySQL relies heavily on STBD Notation, enabling us to summarize our results, infer and investigate the release patterns of MySQL.

3. Case Study of MySQL

For our case study we chose to study MySQL. MySQL is an Open Source SQL RDBMS that is used by many other applications and websites. We chose MySQL because it is

Project	Major	7 days	14 days	31 days	42 days
MySQL 3.23	Major	S+T-B-D=	S-T+B-D+	S-T+B-D+	S-T+B-D+
MySQL 3.23	Minor	S+T+B+D+	S+T+B+D+	S+T+B+D+	S+T+B+D+
MySQL 3.23	All	S+T+B+D+	S+T+B+D+	S+T+B+D+	S+T+B+D+
MySQL 4.0	Major	S-T-B-D=	S-T+B-D+	S-T+B-D+	S-T+B-D+
MySQL 4.0	Minor	S+T+B-D+	S+T-B-D+	S+T+B+D+	S+T-B+D+
MySQL 4.0	All	S+T+B-D+	S+T-B-D+	S+T+B+D+	S+T-B+D+
MySQL 4.1	Major	S+T-B-D=	S+T+B-D+	S+T+B-D=	S+T+B-D+
MySQL 4.1	Minor	S+T+B+D+	S+T+B-D+	S+T+B-D+	S+T+B+D+
MySQL 4.1	All	S+T+B+D+	S+T+B-D+	S+T+B-D+	S+T+B+D+
MySQL 5.0	Major	S+T+B-D+	S+T+B-D+	S-T+B-D+	S+T+B-D+
MySQL 5.0	Minor	S+T+B+D+	S+T+B-D+	S+T+B-D+	S+T+B+D+
MySQL 5.0	All	S+T+B+D+	S+T+B-D+	S+T+B-D+	S+T+B+D+
MySQL 5.1	Major	S+T+B-D+	S+T+B-D+	S-T+B-D+	S+T+B-D+
MySQL 5.1	Minor	S+T+B+D+	S+T-B-D+	S+T-B+D+	S+T-B+D+
MySQL 5.1	All	S+T+B+D+	S+T-B-D+	S+T-B-D+	S+T-B+D+

Table 2. A STBD Notation summary table of MySQL. S - source, T - test, D - documentation, B - build. + indicates that the preceding class of revisions are more frequent before a release than after.

Project	Major	Minor	All
MySQL 3.23	S-T+B-D+	S+T+B+D+	S+T+B+D+
MySQL 4.0	S-T+B-D+	S+T?B?D+	S+T?B?D+
MySQL 4.1	S+T+B-D?	S+T+B?D+	S+T+B?D+
MySQL 5.0	S+T+B-D+	S+T+B?D+	S+T+B?D+
MySQL 5.1	S+T+B-D+	S+T-B+D+	S+T-B?D+

Table 3. Summary of table 2 using majority voting where '?' means no majority

relatively unstudied from a fine grained revision level perspective, although it has been studied from a release level perspective before. Also, MySQL has a good number of major and minor releases for us to study. MySQL is a large software package that for maintenance reasons is split into multiple branches. Each branch is one version of MySQL (3.23, 4.0, 4.1, 5.0, 5.1) that is stored in a separate BitKeeper repository. Note that new branches contain all the revisions of old branches up to the point of the creation of the newer branch. Some revisions, such as bug fixes are shared between branches.

3.1. Assumptions

Our initial assumption is that we'll see all 4 revision classes increase in activity as they approach the release and then drop off after release. We expect source and test revisions to have a positive slope after release as bug fixes come in.

3.2. Tools and Datasets

Our data was extracted from the MySQL BitKeeper repositories for MySQL 3.23, MySQL 4.0, MySQL 4.1, MySQL 5.0, and MySQL 5.1 (fetched 2006-07-26). We used `bt2csv` to extract and convert the BitKeeper repositories to facts stored in a CSV database.

To analyze the data that was extracted we used: *Hiraldo-Grok*, an OCaml based spin off of Grok used for answering statistical based queries; *R*, a plotting and statistics package; *GNUPlot*, a graph plotting package.

3.3. Applying our method

We extracted the release dates from the MySQL manual, and we marked the first releases that were packaged and released to the public as the major releases. The rest of the releases were considered to be minor releases. We then extracted each revision from the BitKeeper repository with our `bt2csv` tool and produced some CSV files and `softChange` databases. Once we had the revision databases we used Hiraldo-Grok to partition the revisions into their revision classes. These 4 classes of revisions were then aggregated per day. Hiraldo-Grok then produced the histograms of the revisions per day distributions.

Figure 2 shows the histogram of MySQL 5.1. The diagram is a histogram of the the distributions of the 4 revision classes. This diagram uses log scaling on the proportional y axis, with bezier smoothing on the curves. Since it is a histogram the values are scaled proportionally. The histogram is a 100 bin histogram, so the x axis is scaled for each class. As we can see the distributions look very exponential for 3

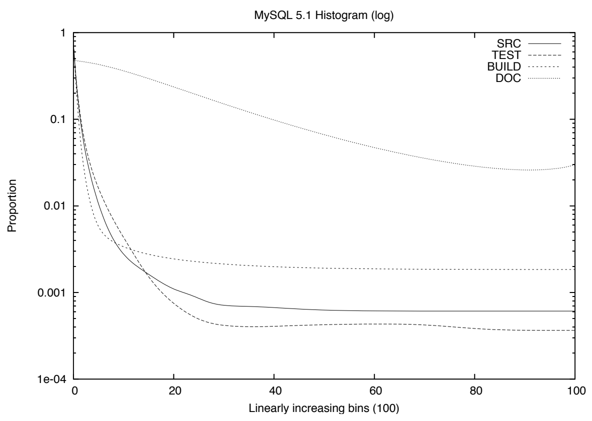

Figure 2. Distribution of revision classes for MySQL 5.1

of the 4 revision classes. Only documentation is different, partially because there are not a lot of documentation revisions but also because there is a peak at the end of the of the distribution, which implies that there are a rare few days where documentation revisions are very frequent.

We then filtered out the revisions which were not within our intervals around a release (intervals of 7, 14, 31, and 42 days) and then filtered the remaining revisions into *before release revisions* and *after release revisions*. Then Hiraldo-Grok produced the necessary tables that allowed R and GNUplot to plot our various graphs. These tables were of the frequency of revisions per day per revision class for *release revisions*. These results were aggregated by our Hiraldo-Grok STBD Notation scripts to produce our summary table (table 2) and majority summary table (table 3). We then used GNUPlot to calculate the linear regression of the frequency of revisions and provided our Hiraldo-Grok STBD Notation scripts with the slopes of the linear regressions. Using those results we produced the linear regression summary table (table 4).

3.4. Indicators of Process

Table 2 shows the results of comparing average frequencies of revisions in intervals before and after a release, where '+' indicates the frequency is greater before a release and '-' indicates the frequency is greater after a release.

Table 3 using majority voting across all of the intervals in table 2. Votes where there is no clear majority result in a '?' symbol otherwise the symbol with the majority is shown. We produced the majority summary table to help average out the noise we see between the measurements in table 2 at different intervals. Table 3 shows us that minor and major releases act both similarly and differently. For example, `S+D+` is consistent across all of the minor release branches

Project	Before	After	Both
MySQL 3.23	`S-T+B+D+`	`S+T-B+D=`	`S+T-B+D+`
MySQL 4.0	`S-T-B+D+`	`S+T-B+D=`	`S+T-B+D-`
MySQL 4.1	`S+T+B-D+`	`S+T+B+D=`	`S+T+B+D+`
MySQL 5.0	`S-T-B+D-`	`S-T-B+D=`	`S-T-B-D-`
MySQL 5.1	`S-T-B-D-`	`S-T-B-D=`	`S-T-B-D-`

Table 4. Linear Regressions of daily revisions class totals: + indicates a positive slope, - indicates a negative slope, = indicates a slope near 0 (Major releases, 42 day interval)

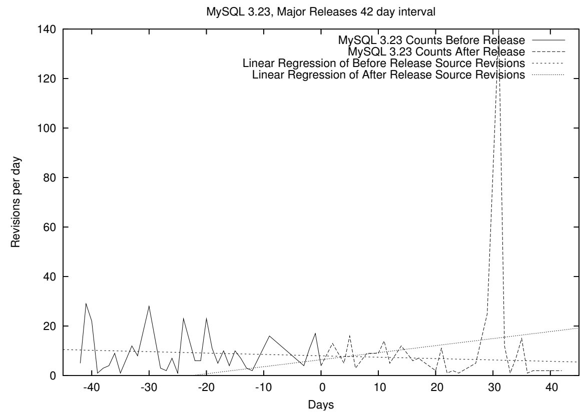

Figure 3. MySQL 3.23 Linear Regression on Source files of Major Release

where as it is only consistent across half of the major release branches. `B-` is consistent for major releases but inconsistent for minor releases. Perhaps build revisions are more frequent during minor releases than during major releases. An interesting observation (table 3) is that `T+D(+/=)` is common among all the MySQLs which suggests that in the MySQL process documentation and testing are done more before a release than after.

Figure 1 depicts a smoothed plot of test revision frequency around release time combined with 2 linear regressions of the before and after release revisions aggregated by a flat window function with a window size of 14 days. The window function in this case is a summation over 14 days starting at the current day. We can see that the slope is negative and there are more test revisions before the release than after. This behavior correlates to the results in tables 2, 3 and table 4 described below.

3.5. Linear Regression

Table 4 shows the sign of the slope of the linear regression of the revision frequency for the revision classes. The releases used were the major releases and the interval was 42 days (the both column uses an interval of 84 days). These slopes were calculated by GNUplot and then Hiraldo-Grok summarized the results using STBD Notation. A negative *before* slope and a positive *after* slope indicates a concave up shape in the revision rate, while a positive *before* slope and negative *after* slope indicate a concave down shape or a peak around the release. If both *before* and *after* are positive it suggests that the rate is increasing, although it doesn't mean that the *after* rate is greater than the *before* rate. Only if the result across the release (*both*) is positive does it imply a continuously increasing rate. We can see that documentation revisions increase before a release and then drop off. There doesn't seem to be much documentation after a release.

Figure 3 shows the aggregation of revisions before and after each release and the linear regression of the plot. The general shape of the frequency of source revisions across the release seems to be a downward slope or a concave up curve with the minima near the release. The later versions of MySQL seem more consistent across the releases with a more subtle slope.

3.6. Discussion

This method enables us to explore the release patterns of MySQL. We can now make claims about MySQL's release patterns and back up our claims with actual data. We can reason about the process MySQL uses (we would need to analyze more projects before we could make general claims about software). We can say that build revisions occur more frequently after a major release than before. We can show that revisions to documentation files occur more frequently before releases (major or minor) than after releases. We can show a general downward slope of the frequency of source revisions across a major release. We can show a general trend that testing occurs more before release than after. The STBD Notation helps us reason about release patterns by summarizing the results down a single analyzable result.

Our assumptions in section 3.1 were partially off. For major releases, source revision patterns and build revision patterns are not consistent with our initial assumptions. Test and documentation classes were consistent though. As per the shape of the graph, we assumed a concave down shape but it was in fact a concave up shape for source revisions.

An interesting release pattern observed is that our linear regression results for source revisions indicate there is a dip and rebound for source revisions around release. Perhaps this indicates a temporary freeze is taking place and the developers are doing last minute fixes and manual testing to prepare the project for release.

If we look at table 3 we see that MySQL versions 3.23 to 5.1 transition from `S-` to `S+`. Using table 4 we can see that from MySQL 3.23 to 4.1 the slope across the release (both) was positive. Perhaps this indicates that releases are handled differently between the versions or the development process of MySQL evolved over time. Maybe in earlier versions more bug reports came back immediately after a release thus prompting patching. According to the release history a minor release immediately preceded the first major release of MySQL 3.23 by 4 days. Perhaps immediate patching was required and those source revisions were committed to the VCS. Given that testing was decreasing yet source and build revisions were increasing it is doubtful new features were being added, perhaps maintenance patches for other architectures that the maintainers don't use, but developed for, were added. Build files are changed more after a release than before, perhaps this indicates that there is some stability before a release and that new features which would affect the configuration of the project are held off till after a release.

Given that source revision frequency often increases across a release while test revisions decreased we can probably conclude that the behavior of MySQL does not indicate that test driven development or test-first development was taking place. If that were the case one might expect tests to mirror the source changes. Testing and source commits don't seem to be heavily correlated around release time.

Documentation seems to be sporadically done before the release as documentation revisions drop off immediately after release. This behavior seems to indicate that documentation is not focused on after a release, and this might suggest that documentation files are updated primarily before a release, especially minor releases where we see consistent behavior.

4. Validity Threats

Our four main threats to validity are: deciding when a release occurred and the severity of that release (major or minor); whether or not that branching into separate repositories affected our results; the statistical significance of our results with respect to number of revisions, projects and releases; choosing an appropriate interval length for analysis.

Our interpretation of a major or minor release might differ from what the developers consider a major or minor release. For instance, we consider the internal release of MySQL 4.0.0 a minor release whereas the publicly distributed release that follows it that was considered the major release. A solution would be to ask the developers directly or to rely on information in the mailing list discussions.

The MySQL repositories are the major release branches of MySQL. Every repository was forked off from a previous repository. Yet even as the repositories lived on as maintenance forks, bug fix revisions from different branches were passed around between each other. A possible solution to this ambiguity would be to combine all the revisions from all the branches into one set and analyze that.

We might have issues with bias and statistical significance. We don't have many major releases, and some releases don't have many revisions thus in some cases we might not have enough data to have statistically significant results. Table 2 suggests that our choice of interval length seems to affect the results somewhat. We attempted to solve this problem by using the majority summary table (table 3) and averaging results. Unfortunately this could've been biased by our interval length choices; each consecutively larger interval contains the previous smaller interval.

We have shown that we are concerned about validity but our method does enable us to talk about what happened based on actual evidence stored within the project's VCS.

5. Conclusions

Our work is an initial step towards automated process extraction from version control systems: we started by analyzing revisions around releases, partitioning those revisions and then further reasoning about release patterns via the interaction of the revision classes.

We provide a method of partitioning revisions into source code revisions, test revisions, build revisions and documentations revisions in order to better characterize release patterns. We provide a method to characterize the release patterns of a project and we demonstrate this on a case study of the branches of MySQL.

We can see that by partitioning the revisions into different classes such as source and testing we can make claims about the behavior of developers and their projects. We can claim and back up our claim that for a project such as MySQL, that documentation and testing occur more before a release than after. By splitting the revisions into revisions associated with one kind of behavior (testing, building, coding, documenting) we gain a clearer picture of the actual release patterns. We can see if developers prepare for a release by testing, documenting, adding code or modifying build files. We demonstrated our methodology with our case study of MySQL where we found both consistent and inconsistent release patterns between branches.

5.1. Future Work

Future work that would rely on our methodology of partitioning revisions could include a study of non-release revisions. If we can correlate behavior during non-release time and release time we can better model the project's behavior.

We should evaluate the interaction of authors with respect to revision classes. We should answer questions such as: do authors practice test-first programming; do they commit tests just after commits; when do authors document the project; can we characterize an author's roles or behavior from their revisions?

Different analysis techniques such as text mining and clone detection could be used to infer even more information. Is there a general difference architecturally between major and minor releases? Major and minor release revisions should be analyzed with respect to architectural change.

We cannot make any global generalizations since we have only studied one project. We will need to analyze many more project repositories before our results are statistically significant enough to allow us to generalize globally about Open Source software processes and project behavior.

References

[1] B. Boehm. A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11(4):14–24, 1986.

[2] J. E. Cook and A. L. Wolf. Automating process discovery through event-data analysis. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 73–82, New York, NY, USA, 1995. ACM Press.

[3] D. M. German and A. Hindle. Measuring fine-grained change in software: towards modification-aware change metrics. In *Proceedings of 11th International Software Metrics Symposium (Metrics 2005)*, 2005.

[4] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[5] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Langages et Modles Objets (LMO 2002)*, pages 135–149, 2002.

[6] T. Mens and S. Demeyer. Evolution metrics. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, New York, NY, USA, 2001. ACM Press.

[7] A. Mockus, R. T. Fielding, and J. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[8] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, pages 328–339. IEEE Computer Society Press, Mar. 1987.

[9] C. C. M. University. *The capability maturity model: guidelines for improving the software process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[10] T. Zimmermann and P. Weisgerber. Preprocessing CVS data for fine-grained analysis. In *1st International Workshop on Mining Software Repositories*, May 2004.