Exploring Characteristics of Code Churn

Master’s Thesis

Jos Kraaijeveld

Exploring Characteristics of Code Churn

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jos Kraaijeveld
born in Gouda, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Software Improvement Group
Rembrandt Tower, 15th floor
Amstelplein 1 - 1096HA
Amsterdam, the Netherlands
`www.sig.eu`

Exploring Characteristics of Code Churn

Author:	Jos Kraaijeveld
Student id:	1509225
Email:	`mail@kaidence.org`

Abstract

Software is a centerpiece in today's society. Because of that, much effort is spent measuring various aspects of software. This is done using software metrics. Code churn is one of these metrics. Code churn is a metric measuring *change volume* between two versions of a system, defined as sum of added, modified and deleted lines. We use code churn to gain more insight into the evolution of software systems. With that in mind, we describe four experiments that we conducted on open source as well as proprietary systems.

First, we show how code churn can be calculated on different time intervals and the effect this can have on studies. This can differ up to 20% between commit-based and week-based intervals. Secondly, we use code churn and related metrics to automatically determine what the primary focus of a development team was during a period of time. We show how we built such a classifier with a precision of 74%. Thirdly, we attempted to find generalizable patterns in the code churn progression of systems. We did not find such patterns, and we think this is inherent to software evolution. Finally we study the effect of change volume on the surroundings and user base of a system. We show there is a correlation between change volume and the amount of activity on issue trackers and Q&A websites.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Company supervisor:	Prof. Dr. Ir. J. Visser, Software Improvement Group B.V.
Company co-supervisor:	Dr. E. Bouwers, Software Improvement Group B.V.
Committee Member:	Dr. M.T.J. Spaan, Faculty EEMCS, TU Delft

Preface

This thesis is the product of my graduation project for the Master of Science degree at the Delft University of Technology. I would like to thank my supervisor from the university, Andy Zaidman, for all the insights and support during this project. The same goes for my company supervisor, Eric Bouwers, who always knew how to challenge me during my time at the Software Improvement Group (SIG). I am grateful for Joost Visser as my other supervisor at SIG, and all of my colleagues there as well. Finally, I want to thank my two friends and housemates Zhi Kang Shao and Pascal 't Hart for being a source of motivation throughout my project. All these people played a a crucial role helping me improve as an engineer, a scientist and a person over the past year.

Jos Kraaijeveld
Delft, the Netherlands
September 5, 2013

Contents

List of Figures **v**

List of Tables **vi**

1 Introduction **1**
1.1 Background . . . 1
1.2 Research questions . . . 2
1.3 Code churn . . . 3
1.4 Research context . . . 5
1.5 Document structure . . . 6

2 Related work **7**
2.1 Software process and evolution . . . 7
2.2 Software metric studies . . . 9
2.3 Classification of activities in software . . . 9

3 Experiments **11**
3.1 The influence of the time interval on code churn calculation . . . 11
3.2 Using churn metrics to identify software activity types . . . 21
3.3 Pattern detection in code churn . . . 29
3.4 Impact of code churn on users of open source software . . . 35

4 Conclusions and future work **42**
4.1 Conclusions . . . 42
4.2 Contributions . . . 43
4.3 Future work . . . 43

Bibliography **44**

A Test set as a result of manual classification **48**

B Training set in ARFF format **50**

List of Figures

1.1 An example of two different code churn measurements. 2

2.1 An example evolution matrix as created by Lanza [29] 8

3.1 Three versions of the same program. 12
3.2 Software change types as defined by Basili et al. [5] 21
3.3 Decision tree when using all our selected metrics as possibilities. 27
3.4 Decision tree when only using relative metrics. 27
3.5 An example code churn pattern, where most of the work is done in the middle of the release cycle. 30
3.6 Weekly code churn progression for JQuery. 34

List of Tables

3.2 The selected open source systems for the interval experiment. 13
3.3 The excluded folders per system 15
3.4 System statistics for the year 2011. 16
3.5 System statistics for the year 2012. 16
3.6 Relative code churn values for the year 2011. 17
3.7 Relative code churn values for the year 2012. 17
3.8 Relative deviation between intervals and commit-based code churn in 2011. . . 18
3.9 Relative deviation between intervals and commit-based code churn in 2012. . . 18
3.10 The median and mean differences between committer date and author date for commits per system. 19
3.11 The chosen churn-related metrics. 23
3.12 The dates of activity on which we based our test set. 24
3.13 The selected open source projects. 36
3.14 The Spearman results for the number of issues and number of asked questions. 38
3.15 The relation of number of questions and issues before and after a release. . . . 39
3.16 The Spearman results between release code churn and the number of questions and answers posted on Stack Overflow 40
3.17 The Spearman results between release code churn, combined questions and answers, and reported issues 40

Chapter 1

Introduction

1.1 Background

We have become irreversibly dependent on software systems. As these systems grow larger and more complex, our understanding of them becomes weaker. Software metrics are used for understanding software systems. Research into software metrics has been done for four decades already, up to the point where it is unclear what the exact state of software metric research is [26]. On another side of the software engineering spectrum, there is the rise of Agile process methodologies like SCRUM [36] and eXtreme Programming [7], which emphasize small iterations of changes and improvement. The emphasis on the software process is a result of the growing complexity of software. By understanding what changes a system went through, we can understand the system. To the best of our knowledge, there has not been much been much work which combines metrics and the software process since the rise of the Agile methodologies.

One of the studies which combines both software metrics and the development process is the work by Basili et al. from 1996 [5]. They surveyed developers regarding the changes they made to a piece of software when going from one version to a next, and gathered information like amount of changed lines and the nature of the change. They show that knowing what type of changes are made during a time period helps with software understanding and the distribution of effort across a release. The downside of their study is that the developers have to fill in a survey each time they make a change to gather the required information.

We want to combine software metrics, software process and software understanding. For this, we decided to study *code churn* [13]. Code churn is the most commonly used basic metric for measuring change between two versions of the same system. Our goals for this study are threefold. Firstly, we want to gather the same information as Basili et al. regarding types of changes automatically. Secondly, we want to use code churn, the base measure of change, to capture patterns in the software development process and further software understanding. Thirdly, we want to know whether code churn and the software process have an impact on the software's surroundings, like user communities.

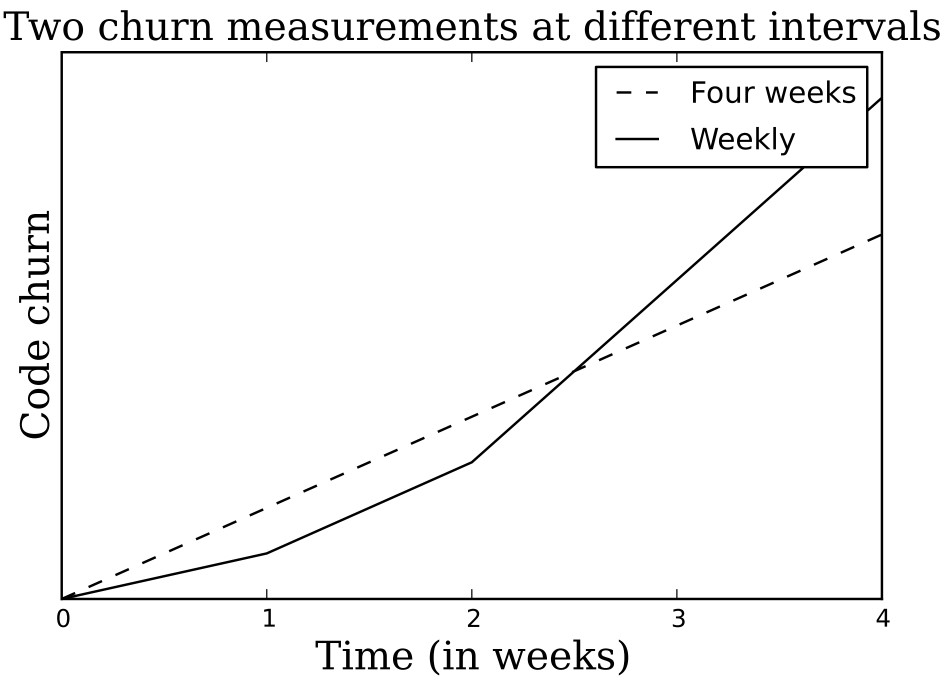

Figure 1.1: An example of two different code churn measurements.

1.2 Research questions

As stated, our goal is to explore code churn's characteristics to develop methods which assist with software understanding. To this avail, we answer the following four research questions.

1. *What is the relationship between time between versions and code churn?*
   Code churn can be calculated for any two versions A and B of a system, as long as A precedes B. However, the time in between the two versions can impact the results gathered. For instance, the churn calculated once on versions a month apart will be different than the sum of all calculated churns of the four consecutive weeks. An example of this is shown in Figure 1.1. By answering this research question, we research the influence of the decision of the time interval on the results.

2. *What churn-related metrics can be used to classify software changes using automated source code measurements?*
   Following the experiment performed by Basili et al. [5], which classified developer activity between versions in different categories by using surveys, we would like similar insight without bothering developers. By answering this research question, we identify what churn-related metrics indicate which type of change, and we show what such a classifier looks like.

3. *Which generic patterns are detectable in code churn?* As a next step, we want to detect generic patterns in a system's code churn progression. This can help with detecting when a project is about to repeat a previous mistake like neglecting tests, when it enters a specific phase of development or is not showing the correct priorities. We study both inter- and intraproject churn patterns. By answering this question, we want to create a clear mapping between a project's process and its actual evolution as it is measureable from the source code.

4. *How much of an influence does churn have on the surroundings of a system?* Code churn is a measure of change, but software change does not necessarily mean change for the end user. By answering this question, we show the effect churn has on the amount of reported issues on the issue tracker of systems, as well as the amount of questions asked on StackOverflow[1]. As such, by answering this question we show how the volume of change (churn) impacts the community of a system.

1.3 Code churn

In the field of software engineering, churn is a measure of change, a measure of progression. In this section, we explain what definition of churn we use throughout the document in section 1.3.1. We show why churn is the preferred metric to measure change in a software system in section 1.3.2. Then, we describe the choices and their effects with churn in section 1.3.3. Finally, we show how churn has been used in past research in section 1.3.4.

1.3.1 Definition

Code churn has been introduced by Elbaum and Munson in 1998 [13]. They define it as the difference between two versions of the same system, as a sum of the added, modified and deleted lines. If we define $\Delta a_{A..B}$ as the amount of added lines if we compare version A and B, where A precedes B, then we define churn as follows:
$CHURN_{A..B} = \Delta a_{A..B} + \Delta m_{A..B} + \Delta d_{A..B}$, where the operators are added, modified and deleted lines respectively. It should be noted that each line can be in at most one category, since the way of measuring between two exact points in time cannot detect whether a line got added first and then modified.

Like studies by, among others, Nagappan et al. [34] [35] and Moser et al. [32], we modify the definition of code churn to only encompass added and modified lines. That is; $CHURN_{A..B} = \Delta a_{A..B} + \Delta m_{A..B}$. This is because deleted lines generally have less impact on the evolution of the system. Taking deleted lines of code in account seperately provides a better view of the actual change that took place.

[1] http://www.stackoverflow.com

1.3.2 Why code churn?

As per our definition, code churn is a metric for measuring the change of a system. Since our goal is to improve the understanding of software and the software process, we believe the metric closest to source code changes, churn, is the most applicable.

Churn is a metric which measures the volume of change a system went through between two moments in time. A comparable metric is the simple *delta*-operation. As apposed to churn, delta simply measures the size at point t_1 and t_2, and gives the difference between the two as result. Churn approximates actual change better than a delta for a simple reason: the size of the system can remain roughly the same even though there have been many adjustments [1].

There are other alternatives for measuring the change a system went through, however. One could look at a system on a higher level, like UML diagrams or API descriptions, but a system goes through many changes which cannot be seen in such high level views. Alternatively, it is possible to extract change information from an IDE instead of comparing files. An approach like this has been proposed by Robbes and Lanza [37]. Using the IDE to measure all the changes as they happen gives a wealth of information, but significantly constraints the projects that can be used in such research. Given our research context, we do not consider such methods since they tie in closely with a specific language. So, even though it is less precise than the fine-grained changes exposed by an IDE, this leads to the another notable advantage of code churn: it can be used in any environment, regardless of operating system, version control or editor used by each developer.

1.3.3 Churn taxonomy

To use churn to its fullest potential, it is necessary to choose the correct way of using it. There are a number of options to choose from when using churn.

- **Granularity.**
  Although we have focused on code churn so far, churn can be calculated on other levels of granularity. A rule of thumb is that a smaller entity yields a more precise approximation of the performed changes. That is, a code churn metric will give more insight than a file churn metric. However, this does not mean code churn is the best or most practical choice in all cases. We will use both when answering RQ2 in section 3.2.

- **Interval.**
  Like explained when discussing RQ3, the choice of interval is often overlooked when dealing with code churn. The relationship between commit-based and time-based churn calculations is explored in section 3.1.

- **Relativity.**
  A relative measure is a better comparable result than an absolute one, since differently sized projects will have different churn influenced by its size and team size. For instance, when creating a generalized model like Nagappan and Ball ([34]), the purpose is to have the metrics be applicable to all types of systems. They normalize the numbers based on the system's size.

1.3.4 Churn uses in other research

Churn, and in particular code churn, has been used by researchers to varying success for different goals. The most notable areas are fault detection and software evolution analysis.

Churn as a surrogate for software faults

Elbaum and Munson first proposed code churn as a fault surrogate [13]. They showed churn is a better surrogate than relative complexity, which is a fairly successful predictor too. Nagappan and Ball used relative code churn to predict defect densities up to an accuracy of 89 % [34]. They use the eight churn-related metrics to achieve this result, like $\frac{Code\,churn}{Total\,LOC}$ and $\frac{File\,churn}{Total\,files}$. There are other approaches in the area of fault detection, like graph-based approaches [8] and fine-grained source code changes [19], but these methods are not as universally applicable as churn-related studies. At the time of writing, churn is still the baseline to compare fault detection results against.

Churn for software evolution analysis

A side product of the study by Elbaum and Munson is that they showed the magnitude of change is barely related to the number of developers involved with the change [13]. Eick et al. showed that Lehman's law of increasing complexity holds true on a large system using code churn [12]. Gall et al. show it is possible to detect non-obvious (logical) coupling between classes in Java using file churn [16]. As such, churn is a metric used for many different purposes in the field of software evolution.

1.4 Research context

This research is conducted at the Software Improvement Group (SIG) in Amsterdam. The SIG is a third party software evaluator and consultancy firm. They analyze hundreds of industrial software systems on a weekly basis. Because the services of the SIG have to be as general as possible, we perform this research with an extra emphasis on language agnosticism.

1.5 Document structure

The remainder of this work will be structured as follows. In Chapter 2 we discuss related work with regards to code churn and our research questions. In Chapter 3 we describe the design, implementation and results of our experiments to answer all four research questions. We present our conclusions, contributions and suggestions for future work in Chapter 4.

Chapter 2

Related work

2.1 Software process and evolution

Ever since Lehman introduced his Laws of Software Evolution [30], it has become a common notion to say that software is never done. This also spurred popularity in the academic field of software evolution. As it is our goal to find patterns in the code churn metric, and that metric is a commonly used indicator of software change and evolution, we find some works have overlapping intentions with ours.

First off, much work has been done to extract information from various types of software repositories. Hipikat, created by Cubranic et al., tracks changes done on a system and uses it to recommend actions to a developer [10]. Although it is closely tied with the Eclipse[1] development environment, it shows how change metrics can be used to detect coupling between entities which are not evident from the code alone. Similarly, D'Ambros et al. have created a model to extract facts from software repositories called RHDB [11]. They use their system to analyse software evolution from various different viewpoints: distribution of work within a team, change coupling analysis and detecting design issues in an architecture. These works are a basis for ours since they show how to gather metrics from various data sources.

Lanza created a visualization called Evolution Matrix in 2001 [29]. An example use of an evolution matrix is shown in Figure 2.1. Like us, he wanted to see what has happened since a previous version of a system more clearly. He represents each class in a system based on a box, where the box's size depends on the number of methods and the number of instance variables it contains. As such, it is possible to see how each class evolves. However, this method is limited in a couple of aspects. Firstly, as a visualization it scales badly. It's not uncommon for projects to have hundreds of classes, and a visualization of all of them would be difficult to interpret. Secondly, we believe that number of methods and number of instance variables does not capture each aspect related to a class' evolution completely. We do believe that the charaterization of events in the lifecycle of a system is

[1] http://www.eclipse.org

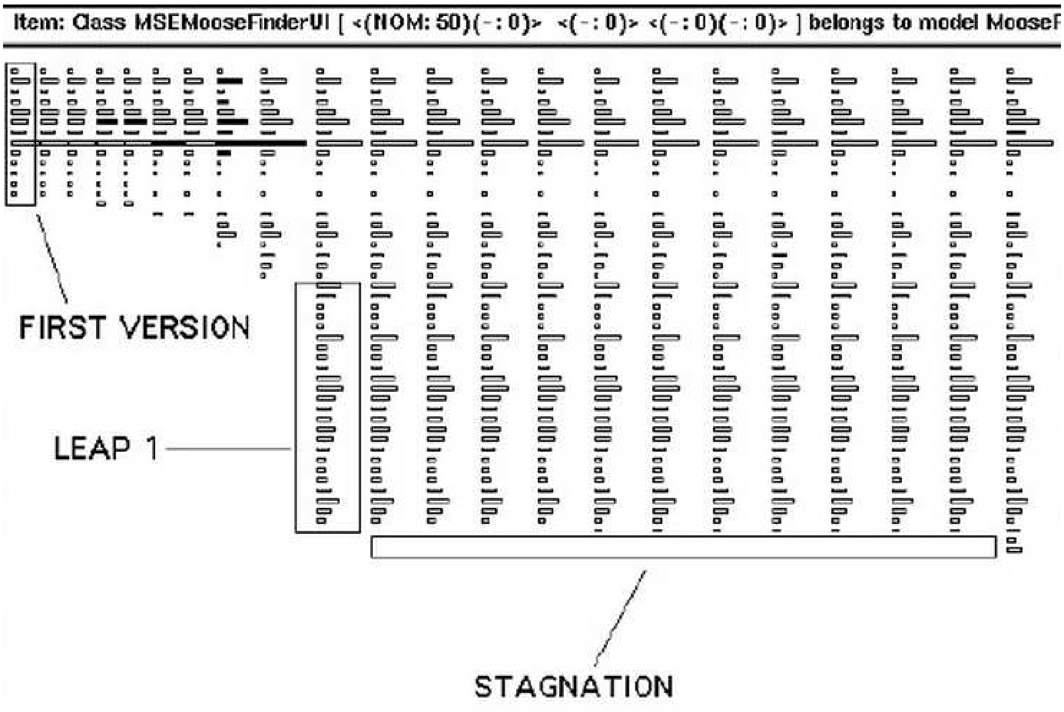
Item: Class MSEMooseFinderUI [<(NOM: 50)(-: 0)> <(-: 0)> <(-: 0)(-: 0)>] belongs to model MooseF
FIRST VERSION
LEAP 1
STAGNATION

Figure 2.1: An example evolution matrix as created by Lanza [29]

.

valuable, and we will work upon this notion when we characterize change activity types in section 3.2.

Alija and Dumitrescu use code churn as one of their key metrics to understand product line evolution [1]. They use the change metrics to determine whether it is safe to release a new version. On top of that, Alija and Dumitrescu mention briefly that for the systems they studied, which are industrial, the majority (88%) of code churn is caused by new lines of code. The relation between code churn and added lines is one we explore more as we characterize activity types. They also note that the number of modules or components of a system does not influence the amount of code churn. This is a relationship we verify properly in section 3.2.

Barry et al. found four different patterns with regards to the volatility of open source systems [4]. That is, they look at the change of a system in terms of its amplitude, periodicity and dispersion. Amplitude is the size of the change, the periodicity is the time it took for the change, and dispersion means the difference in behaviour compared to what you would expect given a set amplitude and periodicity. Based on these three aspects, they classify each change as being part of one out of eight different software volatility classes. Then, they perform phasic analysis to detect patterns in those classifications. The results of Barry et al. are four different patterns of software evolution. However, to get to that point, they reduce each set of change values to ordinal values. This means the change values are ordered in increasing size, but the magnitude of the change is not taken into account. We think valuable information is lost in this conversion. In their analysis, the absolute or relative difference in values is not used, so the set of values $\{10, 11\}$ would give the same result as $\{10, 1000\}$. This means it is possible for very large or very small changes to not be taken into account properly. Furthermore, they only consider snapshots a month apart, without analyzing whether the choice of this time interval impacts their results. Finally, their set of 27 systems might not be enough for a pattern detection study, and we do not know about the differences and similarities of those systems. Our pattern detection experiment, in which we deal take those issues into account, is in section 3.3.

2.2 Software metric studies

There are countless software metric studies, but their numbers shrink greatly if we only look at change metrics. German and Hindle presented a framework through which metrics for change can be classified as one of four types [18]. Their framework is useful since it shows that different metric types have different uses and pitfalls. They promote the use of metrics as precise as possible (on a line level), rather than on a less precise level (like module level or even system level). German and Hindle say that there is not much known about the usefulness of change metrics in software evolution, and more work is required.

Kagdi et al. surveyed and created a taxonomy of software repository mining approaches in relation to software evolution [25]. They identified ten study categories, of which both *evolutionary patterns* and *change classification and comprehension* are directly related to our research questions. However, the evolutionary patterns mentioned refer to associations between entities, like Gall et al.'s work on logical coupling [15]. Gall et al. group elements together if they get changed at the same time, to create a view of which elements are related or even dependent on each other. Although they seek patterns in software evolution, they are different from our goal as we look for repeatable patterns in time, whereas they attempt to link entities together.

2.3 Classification of activities in software

Identifying what type a change in software is, has been the subject of studies ever since Basili et al. classified different types of software tasks based on developer surveys at NASA in 1996 [5]. They showed how the time spent by developers was split up between creating enhancements, fixing bugs and adapting the system without enhancing its functionality. This is basis for our RQ2, since we would like to do the same, but based on source code measurements.

In the footsteps of Basili et al., Mockus and Votta created the same distinction based on textual representations of the changes [31]. Mockus and Votta rely on keyword clustering to build a classification of these text descriptions, in the same categories as defined by Basili et al. Although the aim is the same as our RQ2, they specify that they do not take the source code into account. We believe textual descriptions, like commit messages, can be too ambiguous, and has large differences between projects because of varying standards.

German developed a method to recover the evolution of software projects using its software trails [17]. Software trails are source code releases, CVS logs, issue trackers, mailing lists and change logs. From the source code, he extracted the size in lines and the size in files. He tried his approach on one open source system. Instead of classifying changes, he is able to give more general information about the system as a whole, like that developers focus on their specific part of the system and that most modification requests require changes in only a few files. Although thorough, German's approach is difficult to replicate on a different project because of all the required data sources. On top of that, he barely uses the system's source code besides size metrics. We believe the evolution of a project can be made even more clear if more change metrics are used.

Chapter 3

Experiments

In this chapter, we describe the experiments performed to answer the research questions described in section 1.2. First, we discuss how the choice of interval between versions code churn influences the code churn metric, to answer RQ1.

3.1 The influence of the time interval on code churn calculation

Different studies use code churn in different ways, as shown in section 1.3.4. To see whether those experiments can be compared to each other, we determine the relationship between code churn results and time intervals in this experiment. This gives us an answer to RQ1, and provides us with a basis for future experiments.

3.1.1 Goal and question

The goal of this experiment is to find out whether the interval between the two versions of a system matters significantly with respect to the calculated churn measurement. For instance, how much of a difference exists between the sum of seven churn calculations of versions a day apart, compared to a single measurement of versions a week apart?

An experiment as this has, to the best of our knowledge, not been documented yet. Therefore, we assess the relationship between code churn at the time interval it is measured at, with two goals in mind. Firstly, we would like to know how different academic works relate to each other when they use different intervals. Secondly, we want to ensure the choice of time interval does not invalidate our results in our upcoming experiments.

The question we answer in this section is the following:

RQ1. What is the relationship between time between versions and the code churn measurement?

Intuitively, we would expect that a longer time interval leads to lower code churn, since the same line might have been changed twice in the same period. Consider the three following versions of the same system:

```
def foobar():           def foobar():           def foobar():
   foo()                   bar()                   bar("arg")
```

Version A   Version B   Version C

Figure 3.1: Three versions of the same program.

The code churn when going from A to B is 1, and so is the code churn when going from B to C. If we sum these, the total would be 2. If we calculate the code churn between A and C, we get 1. More formally, for the example comparison with intervals set to a day and a week, we get:
$CHURN_{t=7}(System) \leq \Sigma_{t=1..7}(CHURN_{t=1}(System))$.

At first glance, we suspect two things to be the case regarding this question.

- $H1$**.** The influence of the interval between two versions of a system will show noticeable differences.
- $H2$**.** The influence of the interval will vary greatly per project.

As such, we try to reject the following null hypotheses:

- $H_0 1$**.** The interval between two versions of a system will not show noticeable differences.
- $H_0 2$**.** This influence of the interval will not vary greatly per project.

3.1.2 Design

To answer the research question, we have to take into account for a number of things. We want to ensure we have enough systems and enough time periods to get statistically significant results. We want to measure at four different intervals: per commit, weekly, monthly and yearly. We believe the results of this would in practise be discussed in terms of practical time spawns, so this distribution is logical.

For this experiment, we need access to enough projects for which we can calculate the different churn values. We want these projects to be active and large to avoid skewing with regards to new or dead projects. We also want to only consider source code files belonging to the project, and not for instance documentation or dependencies not part of the developed system itself.

System	Main language	Start size (lines)[1]	End size (lines)[2]
Django	Python	211751	342739
Git	C	327718	425779
Linux	C	13557478	15962285
MongoDB	C++	176113	355176
NodeJS	Javascript	97534	192574
Ruby on Rails	Ruby	237306	223583
Spring Framework	Java	697469	815979
Symfony	PHP	111336	195089
PostgreSQL	C	1675161	2099975
Subversion	C	792442	1037991

Table 3.2: The selected open source systems for the interval experiment.

3.1.3 Selection of systems

We decided to use a time period of two years, since with ten systems this would still give us at least 20 data points for the yearly time measurement. For weekly and monthly measurements, this gives us 1040 and 240 data points respectively. We choose active contributors and code size as measures of maturity. A user is an active contributor if she created at least 20 commits during the past year. This leads us to the following inclusion criteria:

- The version control system (VCS) must be publicly accessible.
- The system must have been subject to ongoing changes during the time period [2011-01-01, 2012-12-31].
- The system must have more than 15 active contributors.
- The system must be over 100.000 lines in size on 2012-12-31.
- The system must have a clear main branch which represents the state of the system as accepted by the maintainer(s).

The selected systems are shown in Table 3.2, and are also selected to be of varying sizes, languages and levels of maturity to reduce selection bias.

[1] on 01-01-2011
[2] on 31-12-2012

3.1.4 Implementation

We chose to select systems hosted on the online code collaboration platform GitHub[3] for two reasons. Firstly, this ensures we can select systems based on activity and popularity, since GitHub allows us to quickly scan a repository's activity. Secondly, choosing GitHub ensures we can use Git for much data processing. Git is a light weight version control system which supports strong querying and reasoning over a repository, with algorithms to calculate differences and code move detections already implemented.

To compare the churn numbers, we first need a uniform way of calculating churn. For this experiment, we use the *git diff* command. By default, this uses Myers' Greedy Diff algorithm [33]. Using this, it is possible to give two SHA1 hashes pointing to versions of the system, and getting the number of added and deleted lines between those versions. A modified line is shown as both an added and a deleted line. As per our definition of churn explained in section 1.3.1, we use only the number of added lines returned by a *git diff* command. To prevent that actions such as renaming, moving a file or changes in whitesspace influence the results, we filter them by making use of the Git -*w* and -*C* flags. -*w* ignores all whitespace differences between the two versions, whereas -*C* detects moves and renames. The latter command is an implementation of a move detection algorithm which works on code level. That is, it detects copies of lines or blocks of lines based on similarities, and does not regard this as a change. As such, for this experiment we only detect changes and additions.

The next step is splitting up the time span in lists of dates based on the chosen interval. Based on each of those dates, it is then possible to extract the SHA1 hash identifying the state of the master branch of a system at that moment in time. This can be done through the *git rev-list* command. By passing two dates, it can return all the SHA1 identifiers in between those dates in an ascending order. This means the state of branch at a certain time is associated with the first SHA1 before the date. As a result, we can get an ordered list of hashes all of which can be compared to each subsequent hash.

After having run and acquired the churn numbers for the selected systems, we noticed there were some unwanted artifacts which skewed the results. For instance, some repositories include dependencies and periodically update an dependency by copying the source code of that dependency into the project folder. As a result, the churn numbers were much higher even though the dependency should not be included in the analysis for that project. To remedy this, and other changes in the repository which are not related to the evolution of the actual system, we allowed for excluding specific folders from the analysis based on regular expressions. Every file with a path matching the regular expression is excluded from the analysis. Which folders to exclude is decided on a per project basis. As a guideline, we excluded documentation, translation-related paths and external libraries. What paths have been excluded for each project are shown in Table 3.3.

[3] http://github.com

System	**Excluded folders**
Django	doc/, django/contrib/localflavor, django/conf/locale
Git	Documentation/
Linux	Documentation/
MongoDB	docs/, src/third_party/, pcre/
NodeJS	lib/, deps/
Ruby on Rails	guides/
Spring Framework	
Symfony	
PostgreSQL	doc/, contrib/
Subversion	doc/, contrib/

Table 3.3: The excluded folders per system

As expected, absolute churn numbers are not readily comparable to each other. Therefore, we must use relative churn instead. To calculate relative churn, we need the size of the system in lines of code, adhering to the same path exclusion criteria. To gather the size of each system, we listed each file in the system at the start of the time period and the end of the time period, excluded the unnecessary paths, concatenated the remaining files and performed a simple line count. This is also how the sizes in Table 3.2 are calculated.

To ensure we have enough data points especially for the yearly measurements, we split up the experiment in two parts: one covering the year 2011, and one covering the year 2012.

3.1.5 Results

First off, we got the sizes and the number of total commits per system for both the year 2011 and the year 2012. The results can be found in Table 3.4 and Table 3.5 respectively. We see that not only the sizes of our selected systems differ greatly, also the growth numbers are different. Ruby on Rails even shrunk a substantial amount in 2012. We think these statistics show how varying our selected systems are.

As for the code churn, we calculated relative values based on the starting size of each respective year. The relative churn values on the four intervals we described earlier are shown in Table 3.6 and Table 3.7. Here we see the differences between the various intervals. To get a more clear view of how large the difference between the numbers is, we divide the code churn values per week, month and year by the corresponding commit-based code churn value, resulting in the percentages shown in Table 3.8 and Table 3.9. Across both these tables, the average value for week divided by commit is -9.58%, and month and year yield -11.76% and -18.73% respectively. These values are much more consistent than we expected with hypothesis H0.

System	**Size on 2011-01-01**	**Size on 2011-12-31**	**Growth**	**# Commits**
Django	211751	304321	43.72%	1669
Git	327718	372678	13.72%	2076
Linux	13557478	14648745	8.05%	47082
MongoDB	176113	198467	12.69%	3982
NodeJS	97534	194294	99.21%	1511
Ruby on Rails	237306	246224	3.76%	5099
Spring Framework	697469	777515	11.48%	1242
Symfony	111336	191134	71.67%	4641
PostgreSQL	1675161	2096176	25.13%	1386
Subversion	792442	979926	23.66%	4511

Table 3.4: System statistics for the year 2011.

System	**Size on 2012-01-01**	**Size on 2012-12-31**	**Growth**	**# Commits**
Django	304231	342739	12.62%	1968
Git	372678	425779	14.25%	2098
Linux	14648745	15962285	8.97%	54430
MongoDB	198467	355176	78.96%	4020
NodeJS	194294	192574	-0.89%	1537
Ruby on Rails	246224	223583	-9.20%	4516
Spring Framework	777515	815979	4.95%	945
Symfony	191134	195089	2.07%	2809
PostgreSQL	2096176	2099975	0.18%	1537
Subversion	979926	1037991	5.93%	3819

Table 3.5: System statistics for the year 2012.

	Relative churn			
System	**Per commit**	**Weekly**	**Monthly**	**Yearly**
Django	112.0%	106.7%	103.7%	93.6%
Git	19.3%	17.7%	17.5%	16.8%
Linux	22.9%	19.7%	19.6%	17.7%
MongoDB	70.1%	65.1%	64.0%	61.6%
NodeJS	149.3%	131.7%	126.7%	113.7%
Ruby on Rails	44.0%	38.8%	37.3%	32.0%
Spring Framework	23.7%	21.2%	19.8%	15.1%
Symfony	192.1%	158.6%	150.1%	120.0%
PostgreSQL	27.8%	26.5%	26.3%	25.6%
Subversion	26.7%	25.4%	25.0%	24.8%

Table 3.6: Relative code churn values for the year 2011.

	Relative churn			
System	**Per commit**	**Weekly**	**Monthly**	**Yearly**
Django	48.8%	44.6%	43.1%	41.0%
Git	41.9%	34.8%	34.3%	32.9%
Linux	19.1%	20.2%	20.1%	18.7%
MongoDB	129.0%	119.7%	115.6%	103.4%
NodeJS	46.7%	37.0%	36.3%	34.8%
Ruby on Rails	38.8%	33.9%	33.1%	28.0%
Spring Framework	17.5%	16.7%	16.4%	15.6%
Symfony	76.2%	66.0%	64.6%	60.1%
PostgreSQL	4.6%	4.4%	4.4%	4.1%
Subversion	15.5%	13.2%	12.8%	11.5%

Table 3.7: Relative code churn values for the year 2012.

	Deviation (%)		
System	**Week / Commit**	**Month / Commit**	**Year / Commit**
Django	-4.8%	-7.5%	-16.4%
Git	-8.5%	-9.3%	-13.2%
Linux	-13.9%	-14.4%	-22.4%
MongoDB	-7.2%	-8.7%	-12.1%
NodeJS	-11.8%	-15.1%	-23.9%
Ruby on Rails	-12.0%	-15.3%	-27.3%
Spring Framework	-10.4%	-16.4%	-36.1%
Symfony	-17.4%	-21.8%	-37.5%
PostgreSQL	-4.9%	-5.7%	-8.1%
Subversion	-4.7%	-6.2%	-7.0%

Table 3.8: Relative deviation between intervals and commit-based code churn in 2011.

	Deviation (%)		
System	**Week / Commit**	**Month / Commit**	**Year / Commit**
Django	-8.8%	-11.7%	-15.9%
Git	-16.9%	-18.2%	-21.5%
Linux	6.0%	5.6%	-2.1%
MongoDB	-7.2%	-10.4%	-19.9%
NodeJS	-20.8%	-22.2%	-25.4%
Ruby on Rails	-12.5%	-14.6%	-27.9%
Spring Framework	-4.9%	-6.3%	-11.1%
Symfony	-13.3%	-15.2%	-21.0%
PostgreSQL	-2.9%	-4.2%	-9.7%
Subversion	-14.9%	-17.6%	-25.8%

Table 3.9: Relative deviation between intervals and commit-based code churn in 2012.

System	**Median CD - AD**	**Mean CD - AD**
Django	0	1.25
Git	0	2.7
Linux	3	18.26
MongoDB	0	1.87
NodeJS	0	5.23
Ruby on Rails	0	2.8
Spring Framework	0	3.18
Symfony	0	3.62
PostgreSQL	0	0.03
Subversion	N/A	N/A

Table 3.10: The median and mean differences between committer date and author date for commits per system.

3.1.6 Threats to validity

As shown in Table 3.7, the results for the Linux project are not theoretically possible. That it, it shows there was more churn when measured on a weekly basis than there was on a per commit basis. This is an inaccuracy which arises from the different way we measure the churn based on a per-commit basis. The time at which a commit occurs can be different from the time it is inserted into the repository we monitor. This is the difference in author date and committer date. The more time in between these two dates, the more inaccurate the results of our experiment become since the commits which are included in the time based approaches might be out of range when calculating the churn on a per-commit basis. This is a threat to construct validity in the terms of Wohlin et al. [45].

This problem is inherent to our way of measuring, but we can mitigate the damage. For each project, we calculate the median and mean difference between the committer date (CD) and author date (AD). Low values mean our results are trustworthy, while high values indicate a potential large differences in our results. These values are shown in Table 3.10. N/A means there is no difference between the committer date and author date because of the project's version control organization. As shown, the mean and median difference between the two dates are very low for each project except for Linux. Linux is famous for its strict commit acceptance policy, so this is no surprise. Therefore, we note that the Linux results are not reliable enough, but we are confident in the results for all other projects.

An experimental setup like this is susceptible to selection bias. Although we attempted to reduce the influence of this bias through our selection criteria, we note it it still possible that the results are not fully generalizable to every other system.

3.1.7 Answer to the research question

As we expected in hypothesis H1, there are clear differences in the measured code churn values between the different intervals. This leads us to reject null hypothesis $H_0 1$. What is most surprising though is that this relation is consistent across various projects, with the percentage deviation not straying far from the mean. Therefore, we cannot reject null hypothesis $H_0 2$.

In this section, we set out to answer the following research question:

RQ1. What is the relationship between time between versions and the code churn measurement?

We see that on average, for around 3-20% of lines which are changed, it holds that they change again within a week. This range increases slightly when considering months instead, and ends up being up to 37.5% if we look at an entire year.

3.2 Using churn metrics to identify software activity types

Understanding software has been shown to be a difficult and time consuming task [42]. It becomes especially difficult when analyzing the code as a third party or when a new employee first encountering an existing codebase. Storey surveyed work in program understanding and shows most of it is done in the area of visualization [41]. To the best of our knowledge, no software understanding study has taken into account the change history of the system to classify the type of work performed. When looking at a system's history of changes, specific types of changes can be linked to problems which occured, which in turn can be avoided in the future.

3.2.1 Goal and question

In 1996, Basili et al. performed a case study on a set of NASA systems in so-called maintenance mode [5]. That is, software systems which are in active use still undergo evolution. Basili et al. were concerned with the amount of effort spent to perform different types of maintenance tasks, and wanted to estimate the cost of a new maintenance release better. As part of their study, they analyzed the distribution of a release in different change types, namely *error correction*, *enhancement* and *adaptation*. This is also show in Figure 3.2.

Basili et al. have interviewed the engineers working on the software systems, and had them fill in a form each time they implemented a change. The form described the change, how much time had been spent on the change, what type of change it was and how many lines of code belonged to the change, among other pieces of information. As a result , they showed that 61% of all time was spent doing enhancement type changes, compared to 14% for error correction and 5% for adaptation. The remaining 20% could not be fit in those three categories.

The work by Basili et al. was focused on the predictive aspect of the change types, that is - how well can they predict how much a new release will cost on this activity distribution? We think such a distribution is also valuable when interpreting information regarding the *past* evolution of a software system. Often times, software engineers do not have the

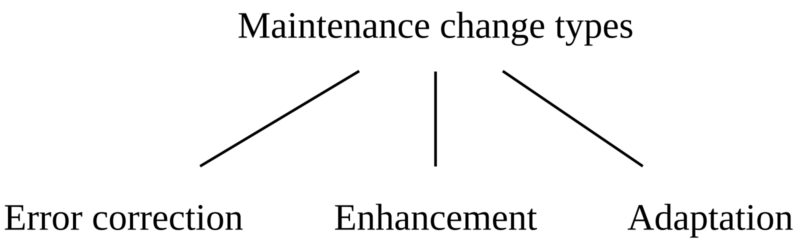

Figure 3.2: Software change types as defined by Basili et al. [5]

luxury of proper documentation or thorough change logs. This is even more relevant for engineers who do not work on a system regularly, like for instance a third party evaluator. Therefore, knowing how to characterize changes can assist with the understanding of the program.

With this experiment, we show how it is possible to make it easier to understand a software system's evolution. To that avail, we answer the following question:

RQ2. What churn-related metrics can be used to classify software changes using automated source code measurements?

For the sake of generality, we wish to remain independent of any type of software or time interval, as per our research context. As such, we restrict our method to a way which still works based on only snapshots of a system, and not use the information pertained in for instance commit logs of a version control system.

In the remainder of this section, we explain the approach to the problem, the results obtained so far and an answer to the posed research question.

3.2.2 Design

When attempting to classify software changes based on source code measurements, we assume there are metric profiles which group the different changes into groups. Like Basili et al., we want to decide between *bug fixes* (error corrections), *new features* (enhancements) and *refactorings* (adaptations). As we do not know what metrics have a positive correlation with a change belonging to a specific group, we take a black box approach.

It is possible to calculate a lot of change metrics based on two snapshots of a system. These range from added, deleted and moved lines of code to the percentage of files with relatively small changes. We gather as many of these change metrics as we can to cover as many aspects as possible. These metrics are shown in Table 3.11. The first step is finding out which of these metrics are influential when it comes to classifying them in one of our three groups. This is a *feature selection* problem, which is common in machine learning and statistics [21].

To be able to use existing feature selection algorithms, we need to build a reasonably sized test set, which contains per change all the metrics, and a manually added classification. We want this test set to be large enough and ensure the systems and snapshots are different enough to represent a wide range of changes. As shown in experiment 1 (section 3.1), we can take the metrics on any time interval. For this experiment, we choose a time interval of one week. We think one week is long enough to capture enough change, but short enough to not have too many changes in one go. To build the test set, we inspect 50 different time spans of a week across different projects and classify them on their primary focus. We validate our classification by letting an independent software developer perform the same classification. We set up a checklist of aspects of a list of commits such a third party validator should consider when performing the classification. Any junior level software engineer

Metric	**Description**
CHURN	The code churn of the entire system.
pCHURN	The code churn of the part of the system which is production code.
tCHURN	The code churn of the part of the system which is test code.
ADDED	The amount of added lines of the entire system.
pADDED	The amount of added lines to production code.
tADDED	The amount of added lines to test code.
CHANGED	The amount of modified lines of the entire system.
pCHANGED	The amount of modified lines of production code.
tCHANGED	The amount of modified lines of test code.
DELETED	The amount of deleted lines of the entire system.
pDELETED	The amount of deleted lines from production code.
tDELETED	The amount of deleted lines from test code.
FILECHURN	The file churn of the entire system.
COMPGINI	Code churn distribution among the system's components, as Gini coefficient [20].
pCOMPGINI	Code churn distribution among production components, as Gini coefficient.
tCOMPGINI	Code churn distribution among test code components, as Gini coefficient.
FILEGINI	Code churn among the changed files, as Gini coefficient.
pFILEGINI	Code churn among the production files, as Gini coefficient.
tFILEGINI	Code churn among the test files, as Gini coefficient.
PERC_SMALLCHURN	Percentage of changed files with less than 10 lines hanged.
PERC_MOSTLYADD	Perc. of changed files where over 90% of the churn is lines added.
PERC_MOSTLYCHN	Perc. of changed files where over 90% of the churn is lines changed.
PERC_MOSTLYDEL	Perc. of changed files where over 90% of the total change is deleted lines.
tCHURNDIVpCHURN	tCHURN divided by pCHURN.
CHURNDIVDELETED	CHURN divided by DELETED.
ADDEDDIVDELETED	ADDED divided by DELETED.
RELCHURN	Code churn of the system relative to its size.
RELpCHURN	Code churn of the production code relative to the system size.
RELtCHURN	Code churn of the test code relative to the system size.
...	Similar relative metrics for all ADDED, CHANGED and DELETED measures.

Table 3.11: The chosen churn-related metrics.

should have enough knowledge to be a validator. As such, we have all the necessary elements to use an existing feature selection algorithm.

With the influential metrics selected, we have to use them to classify the change sets. Given the nature of our problem and the fact that we already have a test set, machine learning is an obvious choice. More precisely, we use supervised machine learning, which is common for classification problems [27]. To decide which classifier algorithm best suits our need, we consider linear classifiers, kernel estimation algorithms, neural networks and decision trees. We also consider the difference between absolute and relative metrics, and the influence of each on the resulting classifier.

With a classifier in place, we test its performance on the test set using a ten-fold cross validation approach. Based on the results, we can iterate to improve it. We rate the classifier based on the precision it achieves on our test set and the Kappa statistic [9]. The Kappa statistic indicates the probability of the results being pure chance, where 0 means the results might be purely chance, and 1 means a perfect classifier. Landis and Koch state that a value of 0 is poor, .10 - .20 is slight, .21 - .40 is fair, .41 - .60 is moderate, .61 - .80 is substantial and > .80 is near perfect agreement [28]. We are satisfied when we reach a precision of over 65% and a moderate Kappa statistic.

System	Year	Dates
Django	2010	01-11, 01-25, 03-01, 04-12, 10-18
	2011	10-17, 11-21
	2012	06-11, 08-13, 10-01
PostgreSQL	2010	02-22, 03-29, 05-17, 06-28, 10-11, 11-15
	2011	01-03, 08-22
	2012	04-23, 10-01
Eclipse JDT Core	2010	02-22, 03-01, 04-26, 11-01, 12-20
	2011	03-07, 04-18
	2012	01-16, 04-16, 08-27
Subversion	2010	02-15, 03-22, 05-10, 05-31
	2011	05-09, 05-30, 09-12, 10-03
	2012	02-13, 07-09
Ruby on Rails	2010	03-15, 08-16, 09-06, 11-22
	2011	02-21, 07-04, 03-28
	2012	07-16, 09-10, 10-01

Table 3.12: The dates of activity on which we based our test set.

3.2.3 Implementation

First, we perform the manual classification to create a test set. We selected ten different dates from five different projects. The dates are chosen arbitrarily among the possibilities where there was a peak in activity in the week preceding the date. Time periods in which there was no activity are not considered for this experiment. The projects and dates are shown in Table 3.12. We selected projects based on their popularity and mainly the difference between them. Django and Ruby on Rails are web frameworks written in Python and Ruby, PostgreSQL is a database system written in C, Eclipse is an IDE written in Java, and Subversion is a version control system written in C. These dates are chosen because they are a peak in code churn compared to their neighbouring dates. The absolute values of the code churn values differs greatly, however.

Classification of each date is done as follows. For each date, every commit on the project's repository for the period of seven days preceding the date is shown, including files changed, the size of the changes and the commit messages created by authors. On top of that, all the closed issues on the project's issue tracker in the same time period are listed. Using these data sources, we aggregate related commits and issues, and classify them as either a refactoring, a bugfix or a new feature. Then, based on the most prevalent type, we determine the main focus during the past week as one of those three categories. Doing this for all chosen dates from the five projects, we have a test set of 50 entries. We validated our classification against those of an independent software developer who is instructed to classify using the same methodology. This yielded an interrater agreement of .94, meaning we disagreed on three instances. After discussion, we came to a consensus on the instances we disagreed upon. The full test set is shown in Appendix A.

To perform the described tasks, we use Weka[4]. Weka is a software suite written in Java which contains tools for data pre-processing, classification, regression, clustering, association rules and visualization [22]. To use Weka, our input needs to be in Attribute-Relation File Format (ARFF). This is a plain text data format similar to common separate values (CSV) files, but with added type information instead. In our case, We combine the metric data and test set classification into a single ARFF file, which is given in Appendix B.

Creating a classifier in Weka is done by supplying a test set, indicating which field is the target to classify, choosing algorithms for selecting the relevant metrics and building the actual classifier. Selecting the relevant metrics can be done a few ways. According to Hall, a correlation-based feature selection approach is most suited when building a classifier which groups entries in at most one category [23]. Therefore, we use Hall's implementation of the correlation-based feature selection algorithm in Weka.

Different classifiers work well in different situations. We considered three approaches discussed by Hall: Naive Bayes, Decision Trees and IB1-Instance Based Learners. In the end, we chose the Decision Tree approach because its intermittent output, the decision tree, is

[4] http://www.cs.waikato.ac.nz/ml/weka/

easy to interpret by us. That said, all three approaches are known to work well, and the difference in performance on our data set was minimal. However, the Naive Bayes approach was less precise, with a 64% precision when using all metrics, since it takes all our selected metrics into account and tries to have all of them weigh in. We think this is unnecessary because we selected redundant metrics, which often cover the same aspect of the system and its changes.

3.2.4 Results

Using the J48graft decision tree algorithm by Webb [43], we used the commonly used tenfold cross validation approach to create and validate the classifier using the test set. We do this for the set of metrics listed in table Table 3.11 and the set of only relative metrics. The resulting decision trees are shown in Figure 3.3 and Figure 3.4. When using all the metrics, we reach a 74% precision with a Kappa statistic of 0.60. When we only use the relative metrics, we get a precision of 70% with a Kappa statistic of 0.54.

It is surprising to see that using all metrics yields a better result, since in other fields like fault prediction metrics of different projects can be compared better using relative measures only [14]. However, we think that the size of the system does not have a linear relation with the change it is subject to. Therefore, absolute measures might be more close to each other than the relative values are.

The threshold values seem a bit arbitrary at some points in the tree, especially the very small relative values. This is most likely a result of our specific test set, and these threshold values are subject to change as the test set is expanded in future work. However, the decisions between *larger than* and *smaller or equal than* show a logical reasoning across the metrics to decide upon a change type.

The Kappa values of both results indicate that they are solid enough to draw tentative conclusions, according to the guidelines set by Landis and Koch [28]. Assessing whether a larger test set indeed does further increase the precision and the Kappa agreement statistic is part of future work.

3.2.5 Threats to validity

For this experiment we have not taken into account how many developers work on each system. Of course, the absolute amount of churn depends partially on the number of developers and the time they work on each system. This effect may have an influence on the numbers between different systems we have used to create our test set, but we believe the impact of the differences is not significant enough because we have used various systems. Determining the exact impact and validating that the impact reduces as the test set grows larger is part of future work.

A threat to validity in this classification experiment is the manual classification done by

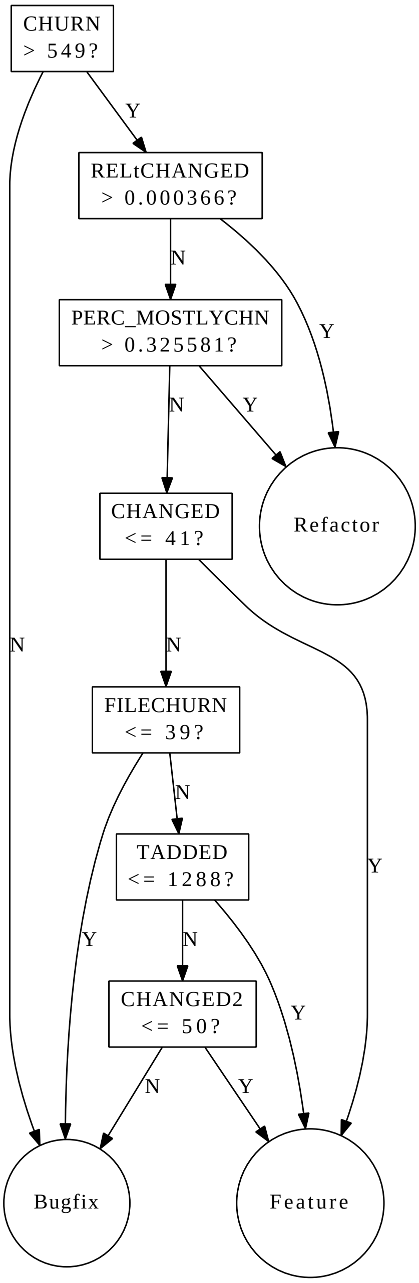

Figure 3.3: Decision tree when using all our selected metrics as possibilities.

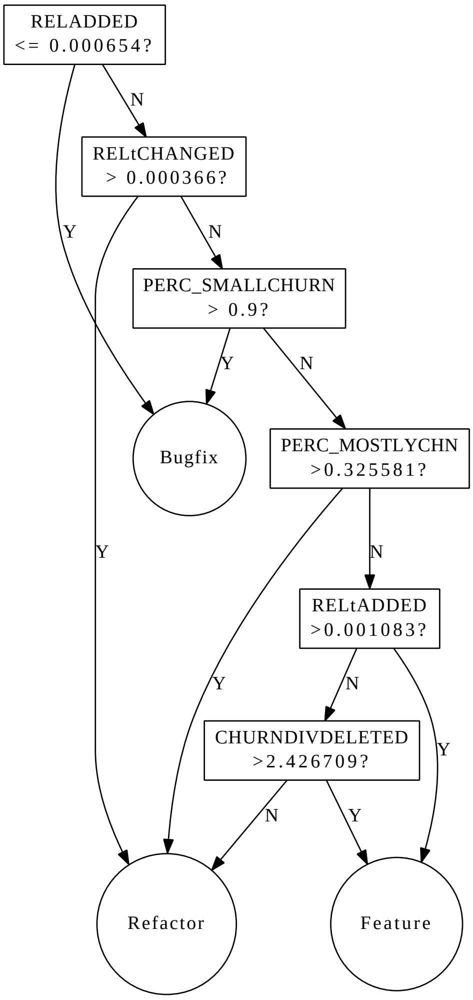

Figure 3.4: Decision tree when only using relative metrics.

ourselves. Although we used a second opinion approach, it is still possible our classifications are not flawless. Of course, during a week's time it is possible for a development team to do multiple types of changes. Therefore, working on a different test set and having developers of the teams themselves classify the test set is part of future work. Alternatively, improving the precision by classifying per system component instead of the entire system will improve overall reliability of the manual classification.

3.2.6 Answer to the research question

In this section, we showed our approach to answering the following question:

RQ2. What churn-related metrics can be used to classify software changes using automated source code measurements?

As shown in this section, we can use software change metrics to build a classifier to determine what change types have been most prevalent during specific time periods. The only requirements for our approach are the two snapshots of the system's code base which should be compared against each other. By using a machine learning approach, we have set up a general approach which allows for constant improvement. This is done by checking the classifier's output and using that to increase its test set.

The most influential metrics are found in our decision trees shown in Figure 3.3 and Figure 3.4, where the upper branches indicate more influence. This means absolute code churn, changed lines, file churn, and amount of files with a small number of changes are among the influential metrics. It also shows that activity in test code is an indicator of various types of software activity.

3.3 Pattern detection in code churn

With code churn being the basis for determining what activities have been the focus of a development team, we can assist with understanding the process a project went through to get to its current state. In this section, we attempt to find patterns in code churn to get a view of the software process.

3.3.1 Goal and question

There has been much attention to different software processes since the rise of Agile methods like eXtreme Programming [7] and SCRUM [36] in the early 2000s. The aim of these processes is to have small, consistent, iterations on the project to remain flexible with regards to changing requirements. Older projects, often working with a traditional waterfall process [39], also know iterations, but generally with a larger time frames in between. The key here is the notion of iterations, or cycles.

Since each project supposedly has some sort of development cycle, we suspect there are detectable, probably matching patterns in the resulting code too. To the best of our knowledge, no such patterns have been found so far. We set out to answer the following question in this section:

RQ3. Are there detectable patterns in code churn?

Patterns is left intentionally vague, since this is an exploratory question. We aim too see whether we can detect peaks of activity based on time in between released versions, whether there is a pattern followed by a set of projects, and whether the same project repeats its on cycle according to its process description.

Although this is an exploratory experiment, we do have expectations based on our own experiences. We formulate the following set of hypotheses:

- $H3$**.** The churn pattern of a project in between releases will approximate a normal curve, as illustrated in Figure 3.5.
- $H4$**.** Projects with a similar process will show a similar churn pattern.
- $H5$**.** Within one project, there are recurring churn patterns which are repeated each release cycle or even multiple times per release cycle.
- $H6$**.** The volume of production churn nearing a release will go down, while the volume of test churn goes up.

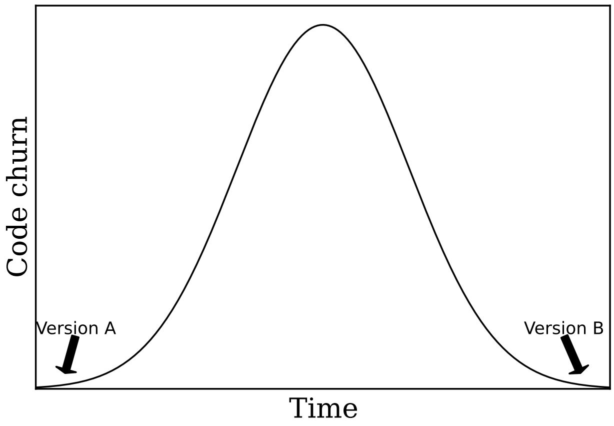

Figure 3.5: An example code churn pattern, where most of the work is done in the middle of the release cycle.

As such, we try to reject the following set of null hypothesis:

- H_0**3.** The churn pattern of a project in between releases will not approximate a normal curve.
- H_0**4.** Projects with a similar process do not show a similar churn pattern.
- H_0**5.** Within one project, there are no recurring churn patterns.
- H_0**6.** There is no notable change in the relationship between production churn and test churn nearing a release.

In the remainder of this section, we explain the design and implementation involved with testing these hypotheses and answering the research question. Unlike the previous experiments, we explain in a fully chronological order due to the exploratory nature of this experiment.

3.3.2 Design

For this experiment, we take an exploratory approach, similar to the approach taken by Barry et al. [4]. Unlike Barry et al., we look on a per week basis, instead of per month. We believe this gives us more precision with regards to the actual size of the changes. Since we are uncertain about what patterns exist, we set up a number of situations for which we can test for patterns.

We use the database at SIG to acquire code churn for more than one hundred industrial systems, as well as ten additional open source systems. The open source systems are the ones listed in Table 3.2, except we substituted Linux with JQuery because of the issues we had studying Linux. We have a weekly snapshot of each system for at least 52 weeks per system, and each system is larger than 10000 lines. We employ different pattern detection strategies to find interproject patterns, if we can find any, on just the code churn metric, both on the absolute and relative values. If we do not find anything substantial, we split up the churn in a production part and a test part, and attempt to do the same again. We expect some sort of consistency within the same projects, so we will look specifically for approximations of distributions. We most expect an activity curve as shown in Figure 3.5 because we suspect there will be more effort into designing solutions at the start of a release cycle, and more effort into quality assurance near the end of a release cycle. Regardless, we also check for constant activity distributions, monotonously increasing and decreasing series . We do this for all projects while automatically smoothing step by step while we apply linear regression to find a pattern. We stop smoothing when we have lost too much data and we have not found a pattern, and mark the system as having nothing found.

As for intraproject patterns, we expect there to be multiple groups of patterns as per hypothesis H4. Therefore, we approach this as an unsupervised classification problem, similarly to Barry et al. [4]. As such, we also choose a phase sequence analyzer.

For the purpose of testing hypothesis H3, we also gather the major release dates of the open source systems involved in this experiment. This is done by manually inspecting each system's website and collecting the dates.

3.3.3 Implementation

First off, we attempt the naive approach. That is, we do not take into account multiple releases or cross referencing between projects. We set up an example function of values approximating the normal distribution using SciPy[5]. We do the same for constant activity, monotonously increasing and monotonously decreasing functions. We treat these as targets for our curve fitting.

One by one, we take the entire weekly code churn time series and try to fit them onto our target functions using least square approximation. Using this method, we do not find any matches across more than 100 systems. We use smoothing techniques described by Shumway and Stoffer [40] to smooth our data step by step and attempt least square approximation again until we find a fit, but each resulting fit is one onto the constant activity function, when all the smoothing has resulted into a flat activity line. This means a naive approach of taking as much code churn we have about a system does not work. Instead of using purely the total code churn, we use the separated values of production churn and test churn. If we apply the same methodology again, we do not find any results. To further illustrate why this approach does not work, the initial churn pattern and the pattern after one smoothing step for JQuery[6] are shown in Figure 3.6.

[5] http://www.scipy.org/

[6] http://jquery.com/

As the naive approach does not work, we gather the release dates of as many systems as possible to only consider the time period between two major versions of a system. We gathered these release dates from the websites of each open source system and by inquiring people involved with the proprietary systems. If we only try to fit code churn progression from within a release cycle onto our example functions, we still do not get a single match.

Hence, we suspect none of our projects can be fitted onto our example functions. The next step is to try and find recurring patterns within each project. This is similar to the goal of Barry et al. [4], so we use the same approach which had been successful for them, with different measurements. They use the WinPhaser software written by Holmes [24]. However, WinPhaser does not allow for anything else than ordinal values to be analyzed, and we want to keep the relationship between different churn values on at least a ratio scale since we get any number of invaluable patterns if we drop the ratio scale constraint. Instead, we implement the Motif Tracker as more recently described by Wilson et al. [44]. The Motif Tracker is meant to find recurring patterns in one single data stream. Although meant for the financial market, we deem it suitable for our data sets too. Motif Tracker is also the most recent and most efficient pattern detection algorithm we could find.

Using our Motif Tracker, there are only four systems out of more than 100 in which we can detect a pattern. Each of these patterns is only found once after smoothing, and they do not correspond with our known release cycles. When we attempt to group similar systems together, like web frameworks or systems in the financial sector, we still cannot improve this result. The same goes when we group the system based on primary programming language.

3.3.4 Results

As explained in the implementation section, we came up empty handed when trying to find patterns in the churn progression of the systems. First, we tested for distributions and based on the results obtained from our open source systems, we cannot reject null hypothesis $H_0 3$. Based on all activity in between the registered release dates, none of them resemble our test distributions. Various degrees of smoothing the data does not change this.

Although some projects in the data set have overlap with themselves at some points, this did not occur often. This leads us to believe that the overlapping points are by chance and not a recurring pattern. We argue that the number of systems we used in this analysis is large enough to make such a claim, since grouping them per primary technology or even business area did not yield results. However, because we did not find the development process for the large majority of the systems, we cannot readily reject null hypothesis $H_0 4$.

When inspecting projects without comparing them to others, we find that the volume of change changes drastically all the time. When we separate between production code and test code, we see that the way test churn and production churn behave is related, as shown in Figure 3.6. However, we cannot reject null hypothesis $H_0 5$ because none of the projects had recurring patterns, regardless of our smoothing strategies.

Finally, we found no inverse relationship between production and test code before a release, like we hypothesized in $H6$. We note that production and test churn are both correlated positively. Whenever there is more production churn, there is often more test churn as well. These findings are similar to the ones found by Zaidman et al. [46]. As such, we cannot reject null hypothesis H_06 either.

3.3.5 Threats to validity

As this is an exploratory experiment, there remains the threat that we were not thorough enough in our analysis. Although our negative results do exclude the generic patterns for which we have tested, there might be others which our method of analysis did not reveal.

Although we use over a hundred systems for this experiment, is still a possibility for selection bias. The vast majority of systems are monitored actively by SIG, meaning there is more supervision of the systems than most projects have.

Lastly, we have not verified the correctness of our Motif Tracker, although the pseudocode supplied by Wilson et al. [44] was comprehensive. Future work would include verifying the correctness and comparing its performance to other generic pattern detection algorithms.

3.3.6 Answer to the research question

As discussed, we could not rejected any null hypothesis. Although we anticipated we would find patterns, this sends a strong signal with regards to the evolution of different software projects. It points towards the observation that most likely no two systems are alike in terms of the way they are evolved. This is supported by studies by Robles et al. ([38]) and Baysal et al. ([6]), who studied the evolution in size of a number of open source systems. Baysal et al. studied two open source browsers and found that, even though they share the same domain and purpose, their developments differed greatly in terms of when new version are released. We confirm these findings on many more systems.

In this section, we set out to answer the following research question:

RQ3. Are there detectable patterns in code churn?

Regardless of us not finding any patterns, we cannot fully state there are no patterns in code churn over time. We did not prove the absence of patterns. However, we believe that if recurring patterns exist within software activity, they are sufficiently blurred by other influences that they will have lost value. From our experiment, it seems likely that there are no code churn patterns and every project distributes its activity in its own way.

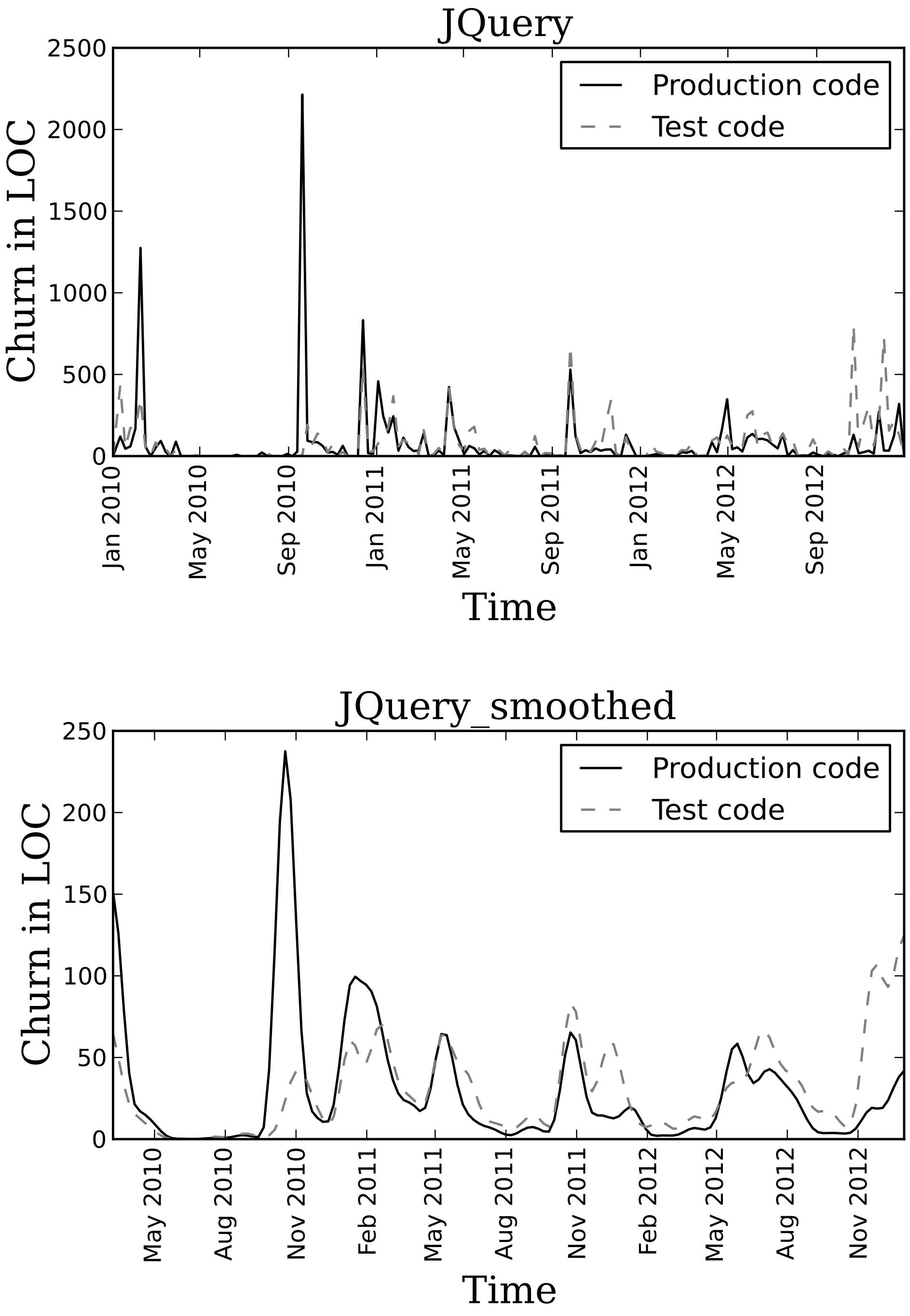

Figure 3.6: Weekly code churn progression for JQuery.

3.4 Impact of code churn on users of open source software

A software system is nearly always part of an ecosystem of developers and users. Any changes made to a software system can influence users in different ways, and due to the nature of open source software, problems users run in to do not always reach the developer again. Sometimes, new issues are created on the project's issue tracker, and the developers learn about introduced issues that way. However, often times when changes do not necessarily lead to faults but simply introduce changes users do not understand, they look towards the popular Q&A website Stack Overflow[7] instead.

3.4.1 Goal and question

In line with the exploratory nature of this work, and after the lack of findings with regards to RQ3, we would still like to know more about other aspects of code churn. Churn is a measure of change, but change to the software does not necessarily mean change for the user. In this section, we want to assess the influence code churn has on the end user of a system. More specifically, we will answer the following question:

RQ3. How much of an influence does code churn have on the surroundings of a system?

In this context, we use *surroundings* loosely. For open source systems, it often holds that its users are other developers. In the context of this experiment, we will restrict ourselves to this case. To guide our gathering of data, we suspect a number of things to be the case. These hypotheses are:

- $H7$**.** There exists a correlation between the number of reported issues and the number of posed questions following a release.
- $H8$**.** Code churn influences the number of questions, answers and issues created after a release.

As such, we aim to reject the following null hypotheses:

- $H_0 7$. There is no correlation between the number of reported issues and the number of posed questions following a release.
- $H_0 8$. Code churn does not influence the number of questions, answers and issues created after a release.

In the remainder of this section we discuss the design and implementation involved with addressing these hypotheses and answering the research question.

[7] http://stackoverflow.com

Table 3.13: The selected open source projects.

Name	Language	Versions used
CouchDB	Erlang	0.9.0 - 1.1.0
Django	Python	1.1 - 1.3
Node.js	JavaScript	0.2.0 - 0.8.0
Ruby on Rails	Ruby	2.3.0 - 3.1.0
Redis	C	2.2.0 - 2.4.0

3.4.2 Design

To answer the question, we require different types of information. First of all, we once again need a representative set of systems for which we can acquire enough release dates. We also need churn data for each of these systems, as well as information about asked questions and reported issues.

We can gather churn data for each of these newly selected systems as we described in section 3.1. For issues, we have to query the issue tracker of each system individually. To gather the number of questions and answers belong to a specific system, we use the Stack Overflow database dump from August 2012 [3].

Using all the information, we apply a rank correlation test to test hypothesis H7. We ensure the precision on this is significant enough to yield good results. To test hypothesis H8, we correlate the code churn to each of the other entities.

Based on the availability of all the required data, we selected the systems and versions shown in Table 3.13 for this experiment.

3.4.3 Mining the required data

Release dates

Each of the selected projects is hosted on GitHub as a Git repository. In Git, it is possible to *tag* a specific state of the repository, which is often used to mark a new version of the system. From this list, we filtered out every entry matching an alpha, beta or release candidate tag. Each system uses similar numbering, so it was possible to determine whether the release was a minor or a major release. The versions are of the form ‘a.b.c’ or a slight deviation thereof. A change in ‘a’ indicates a major release, a change in ‘b’ shows a minor release, and a change in ‘c’ marks a maintenance release. Each tag has an associated tag date which we use to mark the moment of release. We limited ourselves to releases with changes in ‘a’ or ‘b’, since ‘c’ versions are generally not influential with regards to a system’s functionality. To avoid confusion with Stack Overflow tags, we refer to Git tags as releases.

This way, the set of release dates is dependent on the date a release got created in Git. This limited the releases we could gather because of various reasons. For instance, the Django project switched to Git late 2011, and all the versions which had been released before that had an invalid date associated with it. To remedy this, we searched the project website for earlier release dates and combined these with the automatically gathered data. The selection of only ‘a’ and ‘b’ releases is done manually.

Reported issues

Each of the selected projects has a public issue tracker. These issues all have creation dates and, where applicable, resolution dates. The CouchDB issue tracker is part of the large Jira-based Apache one, which has a REST interface to communicate with it programmatically. Django hosts their own TRAC server, which allowed for web requests to return comma seperated value files with the requested information. The remaining projects, Node.js, Ruby on Rails and Redis, all use the GitHub issue tracker, which is accessible via the Github API. We selected the creation dates of issues created during the time frame described by the first and last release found earlier.

Stack Overflow questions and answers

To get access to the Stack Overflow questions, we used the database dump by Stack Overflow of August 2012 [3]. After importing the PostgreSQL dump into a database, we filtered the posts to be of type ‘question’, and retrieved their creation date. Getting answers based on a time frame and a tag proved a bit more tricky. Both questions and answers are in the same table, but answers do not have specified tags. Answers have a field ‘parent_id’, which questions do not have. Therefore, we got the answers by joining the table on itself based on the id matching the parent_id, and then filtering on date and tag once again.

Churn

Similar to our experiment described in section 3.1, we queried Git to gather all the data. This means we excluded folders which would skew the results and are not part of the system we are interested in. For this experiment, we use the per commit measure to fully see the amount of activity developers have put in.

Consolidation

The gathered releases, issues, questions and churn numbers did not perfectly align. For some cases, there were no issues for the time frame of a release, like for instance Ruby on Rails’ 2.3.0 release. This is because the project moved towards using GitHub’s issue tracker from their own solution from version 3.0.0. onwards. For others, the questions were incomplete for a release since the Stack Overflow snapshot is from August 2012. We cut out versions for which we lacked full question data. This left us with the versions described in Table 3.13.

System	Number of weeks	Spearman (ρ)
CouchDB	173	N/A ($p = 0.13$)
Django	192	0.36 ($p < 0.01$)
Node.js	160	0.91 ($p < 0.01$)
Ruby on Rails	190	0.83 ($p < 0.01$)
Redis	75	0.60 ($p < 0.01$)

Table 3.14: The Spearman results for the number of issues and number of asked questions.

3.4.4 Results

Reported issues and questions asked

For each of the selected projects, we found a time span as large as possible where the found issues and questions overlapped. To correlate the two, we binned them in frames of one week. This allowed us to count the number of issues and questions per week. We have chosen to bin them per week since it is a large enough time span to have each bin filled with multiple questions or issues. On the other hand, it is a short enough time span to observe patterns over time. After grouping both issues and questions per week, it was possible to correlate the two. Table 3.14 shows the relation between the number of questions and the number of issues per project using the Spearman rank correlation.

We see that the results are inconclusive for CouchDB, but the four other systems point towards a strong correlation between issues and questions. Of course, this might be due to projects gaining more users over time, automatically leading to more questions and raised issues. It does show, however, that more questions can imply more issues and the other way around and the relation between the two is worth inspecting further.

To see whether the amount of questions and the amount of issues raised is higher or lower after a release, we selected four per-week groups from before and after a release. If the mean number of questions from these weeks is higher after a release, we call it a *question-influential* release. The same way, a higher number of issues after a release means it is an *issue-influential* release. To be comparable, the results are normalized by the amount of questions or issues actually involved in the weeks before and after the release. To formalize, the influence values are calculated as $\frac{Q_a - Q_b}{Q_{total}}$, where Q_a is the mean number of questions in weeks after the release, Q_b the mean number of questions in the weeks before the release, and Q_{total} the total amount of questions involved. The formula for issues is similar. The results are shown in Table 3.15.

The first finding is that the creation of Stack Overflow questions is largely release-agnostic. Most values hover around 0, meaning there is not much of a difference between the amount of questions asked before or after a release. The same is found for issues, which is more surprising. Manual inspection shows that the phases leading up to the release have a similar amount of bug reports due to beta and release candidate versions. This is similar to the

Release	Influence on questions	Influence on issues
CouchDB 0.9.0	-0.250	-0.073
CouchDB 0.10.0	-0.028	0.067
CouchDB 0.11.0	0.000	0.010
CouchDB 1.0.0	0.005	-0.026
CouchDB 1.1.0	-0.057	-0.042
Django 1.1	0.000	0.000
Django 1.2	0.014	0.035
Django 1.3	0.010	0.035
Node.js 0.2.0	-0.003	0.024
Node.js 0.3.0	-0.027	-0.001
Node.js 0.4.0	0.045	0.023
Node.js 0.5.0	-0.002	0.044
Node.js 0.6.0	-0.031	0.038
Node.js 0.7.0	0.000	0.020
Node.js 0.8.0	-0.002	0.015
Ruby on Rails 2.3.0	0.003	N/A
Ruby on Rails 3.0.0	0.015	N/A
Ruby on Rails 3.1.0	-0.005	0.015
Redis 2.2.0	-0.014	-0.083
Redis 2.4.0	-0.061	-0.036

Table 3.15: The relation of number of questions and issues before and after a release.

findings of Anvik et al. in 2006, who studied the Eclipse and Firefox repositories [2]. They also found that the amount of reported issues did not necessarily increase after a release, but did see higher activity around a release. This does not explain why Stack Overflow questions are release-agnostic though. A possible explanation is that users do not adopt a new version straight away for various reasons. More research is required to confirm this, however. Based on these findings, we cannot reject $H_0 7$, since it seems these releases have no influence on either.

Amount of change

The amount of churn of a release is a measure for how much a system has changed between two versions. More churn implies more change. More change implies, for instance, a higher probability of API changes, which might cause more questions to be asked on Stack Overflow. According to Nagappan et al., this is the case for reported issues [34]. To verify whether it holds for questions and answers, we tried to correlate the amount of total churn before a version to the number of questions asked after a release. We performed the same experiment for posted answers and reported issues. Due to the lack of available versions for Redis, we excluded it from this experiment. To compare the different releases against each other, we normalized the amount of questions, answers and issues per release by dividing

System (# versions)	**Questions (ρ)**	**Answers (ρ)**
CouchDB (5)	N/A ($p = 0.62$)	N/A ($p = 0.62$)
Django (3)	1.0 ($p = 0$)	1.0 ($p = 0$)
Node.js (7)	N/A ($p = 0.76$)	N/A ($p = 0.76$)
Ruby on Rails (3)	N/A ($p = 0.67$)	N/A ($p = 0.67$)
Combined (18)	0.68 ($p < 0.01$)	0.71 ($p < 0.01$)

Table 3.16: The Spearman results between release code churn and the number of questions and answers posted on Stack Overflow

.

System (# versions)	**Questions + Answers (ρ)**	**Issues (ρ)**
CouchDB (5)	N/A ($p = 0.62$)	N/A ($p = 0.50$)
Django (3)	1.0 ($p = 0$)	N/A ($p = 0.67$)
Node.js (7)	N/A ($p = 0.76$)	N/A ($p = 0.64$)
Ruby on Rails (3)	N/A ($p = 0.67$)	N/A ($p = 0.67$)
Combined (18)	0.70 ($p < 0.01$)	N/A ($p = 0.72$)

Table 3.17: The Spearman results between release code churn, combined questions and answers, and reported issues

them by the amount of days which passed until the new version. To see whether churn influences the amount of questions, answers and issues, we performed Spearman rank correlation tests on the systems. We chose the Spearman test since we have no knowledge about the distribution of the data. The results can be found in Tables 3.16 and 3.17.

It is apparent that the values for the individual projects are meaningless because we do not have enough releases per project. The values for Django are also not reliable due to the set of versions being too small, even though the p-value is low. If we envision them being one large project, however, it shows that code churn is correlated to the amount of questions being asked on Stack Overflow. However, this might be caused by the projects becoming more popular over time. On top of that, the projects are very different and lumping them together like this is not the preferred way of analyzing them. However, this crude analysis indicates that more research is warranted to uncover the actual relationship between software change and the number of questions and answers.

A similar story unfolds when we look at the reported issues after a release. None of the individual projects can be used to say something meaningful, and even the combination of all releases does not yield a reliable precision factor. Nagappan et al. [34] have shown that this correlation is present in proprietary software, so expanding our experiment will assist in verifying those findings on more recent OSS systems. Therefore, based on our findings, we reject null hypothesis $H_0 8$ since code churn, issues, questions and answers are correlated if we group our data set together.

We observe that there is barely any difference between questions and answers. This is to be expected since the amount of answers depends on the amount of questions. The relationship between the two proves to be very stable for questions about our selected systems.

3.4.5 Threats to validity

The relationship between issues and questions seems to be positive, but this can also be explained by there naturally being more of both as the system gets more popular. This can in fact be the case for code churn as well, as more popular open source systems tend to attract more developers. This relationship and the correlation between these numbers is part of future work.

The number of versions we studied in this section is limited. This is because of the strong requirements of data from three different data sources. Although the individual systems do not yield strict results, we believe the combined results are still statistically significant. Expanding the set of systems and versions for this study is part of future work.

3.4.6 Answer to the research question

In this section, we set out to answer the following research question:

RQ3. How much of an influence does code churn have on the surroundings of a system?

As discussed, we have not enough data points to answer this with certainty, but our experiment strongly points at a positive correlation between code churn and number of questions asked and answers given. Like said before, this can be caused by a correlation between the two, the notion that both increase as the system gets more popular, or both. We think we cannot fully attribute it to the growing popularity of the systems, since for instance Ruby on Rails has not grown much across the past two years, as shown in section 3.1.

On the other hand, our data points at no correlation between code churn and number of reported issues. This is surprising, since we hypothesized they would be related. This observation might be explained by some versions implementing new features, which is a lot of code churn for very few reported issues. Conversely, a lot of bugs can be solved using very little code.

Chapter 4

Conclusions and future work

In this chapter, we recap our findings, conclusions and contributions. We also indicate possible future work which has presented itself throughout this work.

4.1 Conclusions

For each of our research questions, we summarize our findings.

1. *What is the relationship between time between versions and code churn?*
   As shown in section 3.1, the relationship between commit-based and various interval-based approaches are fairly consistent across projects. For week-based approaches, the difference with commit-based measurement ranges between 3% and 20%. On a monthly basis, this range becomes 4% to 22%. Finally, the range on an annual basis is 7% to 37%. The mean values are 9.58%, 11.76% and 18.73% respectively.

2. *What churn-related metrics can be used to classify software changes using automated source code measurements?*
   As shown in Figure 3.3 and Figure 3.4, absolute code churn, changed lines, file churn and *amount of files with a small number of changes* are significant metrics when building a classifier. Test-related metrics, like added lines of test code are also influential in determining the change type.

3. *Which generic patterns are detectable in code churn?* Based on all the approaches described in section 3.3, we have not found any generic patterns across over a hundred different systems. This might imply that each system is developed differently and is evolving differently. More future work is required, however.

4. *How much of an influence does churn have on the surroundings of a system?* We have shown in Table 3.15 that a new release of a piece of software does not necessarily impact the surroundings on a system. However, we have shown in Table 3.16 code churn has a positive correlation with asked questions and posed answers. We have found no such correlations for reported issues.

4.2 Contributions

In this work, we have made the following contributions.

1. We have explored the influence of a chosen time interval on the code churn calculation.
2. We have shown how to create a classifier to determine the primary focus of developers during a time period. This can be used to have a more high level overview of a system's evolution.
3. We have shown how we were unable to detect generic churn patterns in many software systems. Although this does not mean patterns don't exist, it shows the uniqueness of each project and each iteration.
4. We have shown the influence code churn has on its surroundings, namely the system's issue tracker and Stack Overflow.

4.3 Future work

We built a successful classifier, but it can still be improved by adding more and different metrics to our possible pool. We also think expanding the test set will increase performance and reliability of the classifier. Future work would be to create a self-improving classifier which can use correctly classified instances as new entries to its test set.

Although we have not found any in this work, we think detecting generic patterns in evolving software systems can still be helpful. Therefore, extending this research to include more measurements than just code churn for that purpose is part of future work. Another option is extending the work by Barry et al. [4] to work on a ratio scale instead of an ordinal scale, to obtain more applicable patterns.

The ecosystem around pieces of software are arguably as important as the software itself. Therefore, we argue that good future work is in the relationship between the product and the process of the system and its surroundings. We have shown how software releases do not affect surroundings, but the size of changes of a release do have an impact. Finding out what product and process metrics influence the surroundings can help developers with predicting the impact their release will have up front.

Bibliography

[1] S.A. Ajila and R.T. Dumitrescu. Experimental use of code delta, code churn, and rate of change to understand software product line evolution. *Journal of Systems and Software*, 80(1):74 – 91, 2007.

[2] J. Anvik, L. Hiew, and G.C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, New York, NY, USA, 2006. ACM.

[3] A. Bacchelli. Mining challenge 2013: Stack overflow. In *Working Conference on Mining Software Repositories*, page to appear, 2013.

[4] E.J. Barry, C.F. Kemerer, and S.A. Slaughter. On the uniformity of software evolution patterns. In *International Conference on Software Engineering*, pages 106–113, 2003.

[5] V. Basili, L. Briand, S. Condon, Y. Kim, W.L. Melo, and J.D. Valett. Understanding and predicting the process of software maintenance release. In *International Conference on Software Engineering*, ICSE '96, pages 464–474, Washington, DC, USA, 1996. IEEE.

[6] O. Baysal, I. Davis, and M.W. Godfrey. A tale of two browsers. In *Working Conference on Mining Software Repositories*, MSR '11, pages 238–241, New York, NY, USA, 2011. ACM.

[7] K. Beck and C. Andres. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.

[8] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *International Conference on Software Engineering*, ICSE '12, pages 419–429, Piscataway, NJ, USA, 2012. IEEE.

[9] J. Carletta. Assessing agreement on classification tasks: the kappa statistic. *Computational linguistics*, 22(2):249–254, 1996.

[10] D. Cubranic, G.C. Murphy, J. Singer, and K.S. Booth. Hipikat: a project memory for software development. *International Conference on Software Engineering*, 31(6):446–465, 2005.

[11] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger. Analysing software repositories to understand software evolution. In *Software Evolution*, pages 37–67. Springer Berlin Heidelberg, 2008.

[12] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *Software Engineering*, 27(1):1–12, 2001.

[13] S. Elbaum and J. Munson. Code churn: a measure for estimating the impact of code change. In *International Conference on Software Maintenance*, ICSM '98, pages 24–31. IEEE, 1998.

[14] N.E. Fenton and M. Neil. Software metrics: roadmap. In *The Future of Software Engineering*, pages 357–370, New York, NY, USA, 2000. ACM.

[15] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *International Conference on Software Maintenance*, ICSM '98. IEEE, 1998.

[16] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *International Workshop on Principles of Software Evolution*, pages 13–23, 2003.

[17] D.M. German. Using software trails to reconstruct the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):367–384, 2004.

[18] D.M. German and A. Hindle. Measuring fine-grained change in software: towards modification-aware change metrics. In *International Symposium on Software Metrics*, pages 10 pp.–28, 2005.

[19] E. Giger, M. Pinzger, and H.C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Working Conference on Mining Software Repositories*, MSR '11, pages 83–92, New York, NY, USA, 2011. ACM/IEEE.

[20] C. Gini. Measurement of inequality of incomes. *The Economic Journal*, 31(121):124–126, 1921.

[21] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, March 2003.

[22] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The weka data mining software: an update. *SIGKDD Explorer Newsletter*, 11(1):10–18, November 2009.

[23] M.A. Hall. *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.

[24] M.E. Holmes and M.S. Poole. Longitudinal analysis. *Studying interpersonal interaction*, 286:301, 1991.

[25] H. Kagdi, M.L. Collard, and J.I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[26] B. Kitchenham. What's up with software metrics? a preliminary mapping study. *Systems and Software*, 83(1):37 – 51, 2010. SI: Top Scholars.

[27] S.B. Kotsiantis, I.D. Zaharakis, and P.E. Pintelas. *Supervised machine learning: A review of classification techniques*. 2007.

[28] J.R. Landis and G.G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, pages 159–174, 1977.

[29] M. Lanza. The evolution matrix: recovering software evolution using software visualization techniques. In *International Workshop on Principles of Software Evolution*, pages 37–42, New York, NY, USA, 2001. ACM.

[30] M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[31] A. Mockus and L.G. Votta. Identifying reasons for software changes using historic databases. In *International Conference on Software Maintenance*, pages 120–130, 2000.

[32] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *International Conference on Software Engineering*, pages 181–190, 2008.

[33] E.W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[34] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *International Conference on Software Engineering*, pages 284–292. IEEE, 2005.

[35] N. Nagappan, L. Williams, J. Hudepohl, W. Snipes, and M. Vouk. Preliminary results on using static analysis tools for software inspection. In *International Symposium on Software Reliability Engineering*, pages 429–439. IEEE, 2004.

[36] L. Rising and N.S. Janoff. The scrum software development process for small teams. *Software*, 17(4):26–32, 2000.

[37] R. Robbes and M. Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, 166(0):93 – 109, 2007.

[38] G. Robles, J.J. Amor, J.M. Gonzalez-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *International Workshop on Principles of Software Evolution*, pages 165–174, 2005.

[39] W.W. Royce. Managing the development of large software systems. In *WESCON*, volume 26. Los Angeles, 1970.

[40] R.H. Shumway and D.S. Stoffer. An approach to time series smoothing and forecasting using the em algorithm. *Time Series Analysis*, 3(4):253–264, 1982.

[41] M. Storey. Theories, methods and tools in program comprehension: past, present and future. In *International Workshop on Program Comprehension*, pages 181–191, 2005.

[42] A. von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[43] G. Webb. Decision tree grafting from the all-tests-but-one partition. San Francisco, CA, 1999. Morgan Kaufmann.

[44] W. Wilson, P. Birkin, and U. Aickelin. The motif tracking algorithm. *Automation and Computing*, 5(1):32–44, 2008.

[45] C. Wohlin, P. Runeson, M. Hst, M.C. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[46] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen. In *International Conference on Software Testing, Verification, and Validation*, ICST ’08.

Appendix A

Test set as a result of manual classification

An online version of this appendix can be found at:
http://kaidence.org/thesis/appendixa.py

```
testset = {
    ’opensource_django’ : {
        datetime(2010, 1, 11) : "feature",
        datetime(2010, 1, 25) : "feature",
        datetime(2010, 3, 1) : "refactor",
        datetime(2010, 4, 12) : "bugfix",
        datetime(2010, 10, 18) : "refactor",
        datetime(2011, 10, 17) : "refactor",
        datetime(2011, 11, 21) : "feature",
        datetime(2012, 6, 11) : "refactor",
        datetime(2012, 8, 13) : "refactor",
        datetime(2012, 10, 1) : "feature"
   },
    ’opensource_postgres’ : {
        datetime(2010, 2, 22) : "feature",
        datetime(2010, 3, 29) : "bugfix",
        datetime(2010, 5, 17) : "feature",
        datetime(2010, 6, 28) : "bugfix",
        datetime(2010, 10, 11) : "refactor",
        datetime(2010, 11, 15) : "bugfix",
        datetime(2011, 1, 3) : "refactor",
        datetime(2011, 8, 22) : "refactor",
        datetime(2012, 4, 23) : "refactor",
        datetime(2012, 10, 01) : "feature"
    },
    ’opensource_eclipse_jdt_core’ : {
        datetime(2010, 2, 22) : "feature",
        datetime(2010, 3, 1) : "feature",
        datetime(2010, 4, 26) : "bugfix",
        datetime(2010, 11, 1) : "feature",
        datetime(2010, 12, 20) : "feature",
        datetime(2011, 3, 7) : "refactor",
        datetime(2011, 4, 18) : "bugfix",
        datetime(2012, 1, 16) : "feature",
        datetime(2012, 4, 16) : "bugfix",
        datetime(2012, 8, 27) : "bugfix"
    },

    ’opensource_rails’ : {
        datetime(2010, 3, 15) : "refactor",
        datetime(2010, 8, 16) : "refactor",
        datetime(2010, 9, 6) : "feature",
        datetime(2010, 11, 22) : "feature",
        datetime(2011, 2, 21) : "refactor",
        datetime(2011, 7, 4) : "bugfix",
        datetime(2011, 3, 28) : "bugfix",
        datetime(2012, 7, 16) : "bugfix",
        datetime(2012, 9, 10) : "refactor",
        datetime(2012, 10, 1) : "feature"
    },

    ’opensource_subversion’ : {
        datetime(2010, 2, 15) : "refactor",
        datetime(2010, 3, 22) : "refactor",
        datetime(2010, 5, 10) : "feature",
        datetime(2010, 5, 31) : "bugfix",
        datetime(2011, 5, 9) : "feature",
        datetime(2011, 5, 30): "feature",
        datetime(2011, 9, 12) : "feature",
        datetime(2011, 10, 3) : "bugfix",
        datetime(2012, 2, 13) : "refactor",
        datetime(2012, 7, 9) : "feature"
    },
```

Appendix B

Training set in ARFF format

This appendix is limited to one entry. The full training set can be found at:
http://kaidence.org/thesis/appendixb.arff

```
@relation ’Software Event: System’

@attribute datetime date "yyyy-MM-dd HH:mm:ss"
@attribute name string
@attribute CHURN numeric
@attribute pCHURN numeric
@attribute tCHURN numeric
@attribute ADDED numeric
@attribute pADDED numeric
@attribute tADDED numeric
@attribute CHANGED numeric
@attribute pCHANGED numeric
@attribute tCHANGED numeric
@attribute DELETED numeric
@attribute pDELETED numeric
@attribute tDELETED numeric
@attribute FILECHURN numeric
@attribute GINI numeric
@attribute pGINI numeric
@attribute tGINI numeric
@attribute FILEGINI numeric
@attribute pFILEGINI numeric
@attribute tFILEGINI numeric
@attribute PERC_FILESSMALLCHURN numeric
@attribute PERC_FILESMOSTLYADDED numeric
@attribute PERC_FILESMOSTLYCHANGED numeric
@attribute PERC_FILESMOSTLYDELETED numeric
@attribute tCHURNDIVpCHURN numeric
```

```
@attribute CHURNDIVDELETED numeric
@attribute ADDEDDIVDELETED numeric
@attribute RELCHURN numeric
@attribute RELpCHURN numeric
@attribute RELtCHURN numeric
@attribute RELADDED numeric
@attribute RELpADDED numeric
@attribute RELtADDED numeric
@attribute RELDELETED numeric
@attribute RELpDELETED numeric
@attribute RELtDELETED numeric
@attribute RELCHANGED numeric
@attribute RELpCHANGED numeric
@attribute RELtCHANGED numeric
@attribute POTENTIALLYMOVEDLINES numeric
@attribute sclass {bugfix,refactor,feature}

@data
{0 "2010-01-11 00:00:00", 1 opensource_django, 2 1322.0, 3 617.0, 4 705.0, 5
    1259.0, 6 564.0, 7 695.0, 8 63.0, 9 53.0, 10 10.0, 11 269.0, 12 238.0, 13
        31.0, 14 106, 15 0.7257942511346446, 16 0.7619997506545318, 17
        0.3498817966903073, 18 0.6499106344950851, 19 0.6499106344950849, 20
        0.7070422535211267, 21 0.839622641509434, 22 0.5943396226415094, 23
        0.2830188679245283, 24 0.02830188679245283, 25 1.1426256077795787, 26
        4.91449814126394, 27 4.680297397769516, 28 0.030543169373657093, 29
        0.01425501929163875, 30 0.016288150082018345, 31 0.029087632557817156,
        32 0.013030520065614675, 33 0.016057112492202482, 34
            0.0062149111660467155, 35 0.0054986946376175405, 36
            0.0007162165284291754, 37 0.0014555368158399372, 38
            0.001224499226024074, 39 0.00023103758981586305, 41 feature}
```