Characterizing the Roles of Classes and their Fault-Proneness through Change Metrics

Maximilian Steff & Barbara Russo
Free University of Bozen-Bolzano
Bozen, Italy
{maximilian.steff,
barbara.russo}@unibz.it

ABSTRACT

Many approaches to determine the fault-proneness of code artifacts rely on historical data of and about these artifacts. These data include the code and how it was changed over time, and information about the changes from version control systems. Each of these can be considered at different levels of granularity. The level of granularity can substantially influence the estimated fault-proneness of a code artifact. Typically, the level of detail oscillates between releases and commits on the one hand, and single lines of code and whole files on the other hand. Not every information may be readily available or feasible to collect at every level, though, nor does more detail necessarily improve the results. Our approach is based on time series of changes in method-level dependencies and churn on a commit-to-commit basis for two systems, Spring and Eclipse. We identify sets of classes with distinct properties of the time series of their change histories. We differentiate between classes based on temporal patterns of change. Based on this differentiation, we show that our measure of structural change in concert with its complement, churn, effectively indicates fault-proneness in classes. We also use windows on time series to select sets of commits and show that changes over short amounts of time do effectively indicate the fault-proneness of classes.

Categories and Subject Descriptors

D.2.8 [**Metrics**]: Process metrics, Product metrics

Keywords

product metrics, fault-proneness, software architectures

1. INTRODUCTION

When we write software, we make mistakes. This seems to be an inevitable truth of software development. Knowing when, where and how defects (or faults) are introduced and when they occur is therefore a valuable asset in software development. Most approaches of fault detection and localization rely on mining historical data from various sources like version control systems of the source code, bug and feature trackers, and secondary sources like mailing lists. Version control systems generally provide information on who made which change when. Bug and feature trackers as well as most secondary sources augment the timeline of changes with information as to why which change was made. Historical data can have different levels of granularity. For example, the number of defects can be reported by line of code, method, class and so on. Not all data is available at every level, though. Defects in issue trackers, for instance, might not be (correctly) reported for all code versions. In previous work, we presented a method to measure structural change in software systems and derived a metric for the fault-proneness of classes [21]. We represented software structure at the class-level and our timeline was given by the releases of the systems we examined. Among the open questions we left was the issue of how our method would fare at a higher level of granularity. In this paper, we extend our previous work measuring with a finer measure of structural change and churn - changed lines of code per class - at the commit-level. We use two datasets, each covering about three years of development on the Spring Framework and Eclipse. At the commit-level, however, our previous metric is not able to reliably indicate fault-proneness anymore. To investigate the reasons behind this and qualify defective classes by their change history at the commit-level, we pose the following nested research questions:

RQ1 Are there different types of class change histories? Do they depend on the characteristics of software products?

RQ2 Do change history types indicate code defectiveness? Does this depend on characteristics of software products?

RQ3 Does local change history indicate top defective classes? What are the characteristics of the change history that are better observed in defective classes?

Structural change and churn are two complementary dimensions of complexity in software: complexity between and inside classes. In this work, we collect structural change and churn for each class over commits using bug-fixing commits as proxy for defects in that class. We observe these measures in time series. The new measures we are able to define over time series - length and stationarity - neatly separate the

ESEM'12, September 19–20, 2012, Lund, Sweden.

time series into sets with consistently distinct defect densities for both structural change and churn. Finally, we show the advantage of using both structural change and churn in their complementary nature by further identifying the most fault-prone files and defects over files with these measures.

In the remainder of this paper, we will first have a look at relevant literature. After describing our method in detail, we proceed to describe our datasets and how we collected them. Then, we present the results of applying our previous method on these datasets, followed by an analysis of our new approach. We finish with limitations, a summary of our findings, conclusions and a few remarks on future work.

2. RELATED WORK

There is a wealth of methods for determining and predicting the fault-proneness of files or components in software development. We restrict ourselves to discussing studies that are directly related to our work.

Most relevant for our work are methods that include either product or process metrics. Process metrics have been introduced by Graves *et al.* [7]. They define process metrics as measurements taken on the change history of the code. The software repository is the prime source for these metrics. Graves *et al.* concluded that "process measures based on the change history are more useful in predicting fault rates than product metrics of the code." Thus, they introduced the differentiation between process and product metrics, the latter being measurements of the code and its structure. An example for the use of process metrics is the work by Zimmermann *et al.* [23]. They collected a set of measures on the method-, class-, and file-level for Eclipse and correlated these values on the file- and package-level to pre- and post-release defects. We will later use their results as a benchmark for our results.

Lately, the modeling of histories of process and product metrics as time series has garnered considerable attention. Ratzinger *et al.* [17] collected a number of metrics for three systems on every day for two months. The metrics included number of lines added and deleted, number of different authors of changes, and number of bugs. On the time series of these attributes they applied a genetic algorithm for feature selection to use in predicting the number of defects over the next two months. They obtained rather high correlation coefficients - between .716 and .946. One of their systems was Spring. They found that in Spring there are only few files with a single bug and very few with more than one. This uneven distribution of defects is an obstacle for defect prediction. For Spring they also found that the single best predictor was the number of times a file had been changed, followed by the number of authors of these changes. Kenmei *et al.* [10] collected bi-weekly snapshots over five years for three systems, one of which was Eclipse. From every snapshot they extracted the number of lines of code and identified the number of new change requests, i.e. bugs, from a bug tracker within that period. Then, they modeled and predicted the request density of requests per unit of size using ARIMA. While their results for JBoss and Mozilla are useful for predicting and identifying trends, they found it quite hard to do even one step ahead prediction for Eclipse. Couto *et al.* [1] used an approach similar to Kenmei *et al.*. Also using bi-weekly snapshots collected over several years, they collected a number of metrics including the CK suite of metrics from the code. They used Granger Causality [6] to find the causes for bugs in changes of the values of the metrics. For 64-93% of bugs they could identify a cause in the metrics. However, there was no single best predictor for any system nor any file. Ostrand *et al.* [16] built a negative binomial regression model using data like lines of code, a file's age and previous faults to predict the number of faults per file. They found that 20% of the files contained on average 83% on one and 84% of faults on another system. Menzies *et al.* [13] used their previous results on using static code attributes such as lines of code, Halstead and McCabe, and similar results to conjecture that moving away from maximizing the probability of detection over false alarms towards the approach taken by Ostrand *et al.* and others is, in fact, preferable. They conclude that current results essentially constitute a ceiling for what learners can achieve. Instead, better results can be achieved by maximizing the number of defects in as small a set of files as possible.

Reflecting on the above methods of analysis, we can identify several approaches to granularity. Measurements can be taken at the class- or package-level, the metrics themselves vary in their level of abstraction. Time-frames for each measurement range from single days (possibly already aggregating changes if several commits occurred that day) to indefinite spans of time given by the dates of releases. This causes two problems. First, if two classes were added to the system in the same commit, the length of their history is considered to be the same, no matter how many changes it includes. At a low granularity, i.e. using releases as time intervals, the change in a metric may degrade to a statistical artifact where the fact that it changed is more important than how it changed or what. Depending on the system, a change in a rarely modified file may be a very good indicator for a defect in that file, no matter what the change actually did. Second, at high granularity, it is apparently difficult to find a single best metric to predict fault-proneness. There are several possible explanations for this, for example that the level of abstraction of the employed metrics does not capture underlying relations or that confounding variables have a disproportionately high impact at this level of detail that they do not have at a lower granularity.

Śliwerski *et al.* [20] linked bug-fixes to the changes responsible for the bug using the commit history of Eclipse. One of their observations was that some classes were almost exclusively and repeatedly changed in bug-fixes. Classes have different characteristics and roles in a system. Thus, lumping them together disregarding their specificities may negatively impact the assessment of their fault-proneness. Ekanayake *et al.* [4] observed that they could track concept drift of software projects using defect prediction quality. On Eclipse and three other systems they found that defect prediction quality did indeed vary significantly over time. Specifically, they identified different periods of time where prediction quality was either stable or drifting. Furthermore, they could link periods of drift to an increase in the number of authors contributing code to the projects. Related to this observation is work on change bursts by Nagappan *et al.* [15] or Rossi *et al.* [19]. Nagappan *et al.* noticed that for commercial software periods of increased activity on the code could predict defects very well. Applying their method on Eclipse, however, they noted that change bursts work just as well as regular change activity as predictors. They attribute this to the open source nature of Eclipse. Rossi *et al.* investigated the relation between code churn, bug fixing activities,

and software release dates to discover bursts in time-series of commits in several OSS projects. They found that there is an increase in code-related activities in the OSS community in the proximity of a software release but no specific activity in proximity of a bug fixing. Misirli *et al.* [22] investigated bugs in Eclipse in beta-releases and discovered that they have an interesting characteristic. They found that these bugs are concentrated in Eclipse files with few changes by few different people. This further indicates that there is value in differentiating files and the characteristics of their change behavior over time.

3. METHOD

In our previous work, we have presented and evaluated a measure for structural change as an indicator of fault-proneness of classes [21]. We use a graph kernel, specifically the Neighborhood Hash Kernel (NHK) [8], to measure structural distance between the graph representations of consecutive releases of a software system. In this section, we first describe these graph representations and how we extended them for our current work. Then, we briefly recapitulate how the NHK works and how we apply it as well as some notes about churn. This includes a description of our representation of structural changes and churn for each class over time. Based on these time series, we present our measurement of temporal change patterns and its implications for the analysis of the time series.

3.1 Graph Representation

The definition of structure for a software system depends on the programming language used and the level of granularity at which we consider the code. We restrict ourselves to the study of systems written in Java. In Java, we have the overall systems, its packages and its classes as the most important levels of granularity. For our purposes, we choose classes as the main entities for the representation of structure. The advantage of this choice is that classes essentially correspond to files in Java. Since version control systems typically manage files, we can easily map classes to files.

Thus, every class is a node in our graph representation of the structure of Java code. We have directed edges between two classes for any inheritance relationship, for every method call, and for every referencing of a public field. We distinguish between different method calls and field references so that the graph can be and typically is a multi-graph with several edges between nodes. For two classes, we collect all distinct method calls from each method in one class to each method in the other class. Previously, we had a single link between classes if there was any number of dependencies. Collecting the cardinality of each method call would entail parsing the structure of conditionals and loops in the code enclosing the call. Determining a definite number for the cardinality of an edge in the graph may thus not always be feasible which is why we do not consider cardinalities. We ignore all classes and relationships that are or go outside of the system. After all, we are interested in the development of the system and that has to exclude everything outside of that scope and beyond the reach of the system's developers.

3.2 Measuring Structural Change

The basic idea of the NHK is this. The name of each node is replaced with a unique bit label. Then we encode the neighborhood of each node into its label. We can repeat this encoding iteratively to include information about a bigger and bigger neighborhood in a node's label. This means that we calculate new labels for each node for each iteration. To compare two graphs, we count the number of labels they have in common for a given iteration, divided by the number of total labels for that iteration.

Originally, the NHK was used in applications in biology and chemistry where node names are not necessarily unique. Computing several iterations thus encodes local structure - neighborhoods - into each node and thus allows for efficient comparison of the global structure of the whole graph. In our case, node names are the classes' fully qualified names and are therefore distinct. Computing several iterations nevertheless yields an advantage. It lets us trace the impact of changes. When we compare the node labels of two graphs, the node labels of nodes whose dependencies are different will change in the first iteration. In the second iteration, the labels of nodes that have edges going to those nodes will change and so forth. This means that the NHK lets us trace change impact along the edges in the graph. For our purposes, we limit the calculation to three iterations, i.e. to nodes two steps removed from the original change. The predictive capabilities of dependencies and their change are firmly established in the literature (e.g. [11]). One more recent example of the influence of flawed classes on their clients in terms of defects is the work by Marinescu and Marinescu [12]. This supports the notion that impact of changes as represented by the NHK is valuable for determining fault-proneness of classes. We add changes in dependencies in our method and believe that they are even more valuable to identifying fault-prone classes.

For a set of graphs and their nodes we now have three values for each node in each graph. Since each graph represents the system at the time of a commit to the software repository, we can order the graphs by the time of the corresponding commit. Now we can compare the labels for nodes pairwise for two graphs and determine which labels changed due to the changes in that commit. We mark a 1 if the value changes and a 0 if it does not. We assign different weights for changes in the three iterations. If a label changes in the first iteration, we assign a weight of four, two for the second iteration and no weight for the third iteration. For every class we arrive at a vector with a length equal to the number of commits in the repository. Each entry in the vector reflects the change of that class in that commit. If a class is added or deleted in one of the commits we are considering, we do not consider it changed, though. Only if an existing class is changed in terms of dependencies we add the corresponding value to the vector for this class.

3.3 Measuring Size Change

Lines of code are a very simple and intuitive measure of size. However simple and intuitive, lines of code have also been the subject of countless studies and discussions, and a source of many problems when considering the fault-proneness of files (e.g. [18, 5]). Many problems stem from lines of code being an absolute value that makes comparison difficulty. Churn, thus, is a relative measure and is defined as the change of lines of code in a given period of time and has shown great value for the prediction of defect density (e.g. [14]).

Version control systems allow for the simple computation of added and deleted lines in the change to a file. Unfor-

tunately, a modification in a line of code is represented as the deletion of the original line and the subsequent addition of the modified line. This causes several issues for the computation of churn. Some of these issues and a solution are discussed in [9]. We follow their approach to compute the churn for each commit. As with measuring structural change, we create a vector for each class with each entry being the churn value for that class for that commit.

3.4 Temporal Change Behavior and Defects

Typically, the change vectors for each class contain many zeros because most classes do not change most of the time. If a change occurs, it might introduce a defect into that class and necessitate a subsequent bug fix. Viewing only a single class, this chain of events is clear cut. From the point of view of the commit history of the repository, there may be many other commits in between the changes to that class. Assuming that these changes are not relevant for that class, we remove every 0, i.e. commits where the class was not modified, from its change vector.

There are different ways to analyze the properties of the resulting time series vectors. Since we are interested in how the values in the vectors behave over time, we employ the Augmented Dickey-Fuller test (ADF) [3] to check if the time series in the vector is stationary. A time series is stationary if its probability distribution and thus properties like mean and variance of the series do not change with time shift. Stationary time series allow prediction over time series periods and can reveal patterns in the evolution. The null hypothesis of ADF assumes non-stationarity of a time series. In our case, this tells us if there are different, non-periodical intervals in the time series of a class. For example, if there are periods where the class was changed but the churn values are low, and other periods where we have subsequent, high churn values for that class. To get stable results from the ADF test, we have to disregard all time series that have less than 30 values. We will see the implications of this cutoff in our analysis.

Assuming that we know in which commits a class was changed as part of a bug-fixing commit, we can create a second time series for each class. We create two bug vectors for each class, one for structural change and one for churn since these vectors may have different lengths. The bug vectors have the same length as the corresponding change vectors, we set every entry to 1 that corresponds to a bug-fixing commit and 0 otherwise. The bug vectors for churn cover all bug-fixing commits for a class. The bug vectors for structural change do not necessarily cover all bug-fixing commits because not every bug-fixing commit also changes the structure of the system. We do assume, however, that a structural change may cause a defect that does not require a structural change to fix. For this reason, we additionally insert all bug-fixing commits immediately following a structural change in the vector for structural change and the corresponding bug vector.

Since we have two pairs of vectors for each class, we have several options for testing stationarity. For the present study, our main interest is with consistently changing classes. Thus, we test both the metric vectors and the bug vectors for stationarity. In other words, we are interested in classes that have a stationary history of changes as well as a stationary history of bug-fixing commits. If either or both vectors are non-stationary, we count the pair as non-stationary. Further distinguishing between these is matter of future work.

If a time series is non-stationary, we know that its mean and variance change for different sub-series. In our case, we can assume that these different time windows have different influence on the defect vector. In a period of small changes, we also expect there to be few bugs. In times of much activity, the chance of incurring defects is probably higher. Based on this observation, we can already make assumptions about different kinds of time series. Generally, short time series will have few bugs. Following Graves *et al.* [7], the number of changes to a file is a pretty good indicator for its fault-proneness. The fewer changes, the less fault-prone it is. With the help of ADF we can distinguish two different behaviors in longer time series, stationary and non-stationary behavior. If the behavior changes, say, periodically, the kind of period we are in at the moment influences our estimate of the corresponding file's fault-proneness within that time-frame. Similarly, predictions of the fault-proneness in the near future depend on the type of the current period. We propose a simple method to approach this issue. We shift small windows across the time series and sum the values in each window. Then, we correlate the value to the corresponding window on the bug vector for the same file. Changing the granularity like this allows us to consider short periods of time. The correlations we obtain then tell us if and how changes in short periods correlate to defects in the same time frame. We use windows of different size to explore how the vectors influence each other. Since our cutoff for the length of vectors to test in the ADF test is 30, we decided to use evenly sized windows equal or smaller than 30, namely 10, 20, and 30. Since longer histories provide us with more information, we let the windows slightly overlap by having the lower bound for each window increase by only half of the rate of the upper bound. The idea is to use more and more information about a class' history as it becomes available over time.

Our approach is, in a nutshell, to differentiate between different time series of structural changes for classes. This differentiation leads us to the realization that different properties of these time series require us to consider windows on these time series. Thus, we shift discrete, slightly overlapping windows of various sizes over the change and defect vectors, sum the values in each vector for each window and then correlate the values for change and defects for all classes.

4. DATA COLLECTION AND PROCESSING

4.1 Data Sources

We collected two datasets. The first dataset is the Subversion repository of the Spring Framework[1] for the development of version 3 of the system and the corresponding JIRA bug tracker. The second dataset is the CVS repository for Eclipse[2] and the corresponding BugZilla bug-tracker. The Spring repository covers the time from July 2008 to mid-October 2011 when we cloned it. We selected a comparable interval for the Eclipse repository and chose the time from January 2002 to December 2004. This is approximately the same time span covered by another, popular Eclipse data set [23]. In contrast to that dataset, however, we limited

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

ourselves to one module in the repository, the JDT module. For both datasets, we restrict our analysis to the *trunk* since development activity for both projects mainly takes place in the *trunk* rather than *branches*. We do not consider refactorings, e.g. moving or renaming of classes. Doing so would probably increase the average length of our time series. However, we assume that a refactoring may also indicate a change in characteristic of class. To not bias our analysis, we therefore consider classes subject to refactorings as distinct entities. Also, we exclude all test files since they do not contribute to the core functionality of either system.

From the bug trackers we extracted all bug reports that affected versions developed in the respective time frames, i.e. bugs where we can assume that they were introduced during the time of our observation. From this subset, we selected all bug reports that were fixed within the respective intervals.

4.2 Data Processing

We converted the Eclipse repository to Subversion using cvs2svn[3] and then processed both Subversion repositories using SVNPlot[4]. Since Subversion only counts the lines added and deleted for every file per commit, we used Hofmann and Riehle's [9] technique to calculate churn values for each change to a file. To create graph representations for every commit in each repository, we had to build each commit into byte-code. Byte-code allows for easy and very fast extraction of classes and relationships between classes by using a library such as Apache BCEL[5]. Unfortunately, building commits typically fails because of unfulfillable dependencies to external libraries. For this reason we used PPA [2], a tool that creates partial builds from source code ignoring all dependencies to external libraries. From these partial builds we created our graph representations of their structure with a parser we wrote using BCEL. We save each graph to a database on which we then apply the NHK to calculate the structural distance between consecutive versions of the graphs.

Both bug trackers assign unique IDs to every bug report. Following common practice, we used regular expressions to find occurrences of these IDs in the commit messages of the repositories. For every match, we mark the commit as a bug-fixing commit. We cannot readily deduce bug-introducing changes or the actual location of defects, though. The actual location of a bug might be different from the place of the fix because the actual site may be too sensitive to changes. We use bug-fixing commits and the files therein as a proxy for the actual defects and their locations, following common practice. Since we use time series, we can assume that in most cases we will at least have a reasonable connection between bug-introducing and bug-fixing commits.

For each class, we have two pairs of vectors with structural change and churn, and their corresponding bug vectors. Each entry in the vectors represents a commit in which the class was changed according to our metrics. The change vectors contain those values, the bug vectors are boolean and indicate if the corresponding commit was a bug-fixing commit. As mentioned above, we added those commits to the vectors for structural change that only had churn but were bug-fixing commits and immediately followed a commit with structural change for the respective class. This is based on the assumption that structural changes may cause defects but are not necessarily fixed in commits with structural change. Since they are presumably caused by a structural change, though, we still want to include them in that vector.

[3]http://cvs2svn.tigris.org/
[4]http://code.google.com/p/svnplot/
[5]http://commons.apache.org/bcel/

Table 1: Summary Information of Spring and Eclipse.

		Spring	Eclipse
#classes	beginning	1,791	1,150
	end	2,709	2,278
	total	3,457	3,093
#dependencies	beginning	6,819	11,619
	end	12,169	21,627
	total	21,671	42,896
#commits	total	2,960	17,488
	structural	1,051	2,145
	bug-fixing	449	4,546
	struct. bug-fixing	147	803

4.3 Summary Description of the Datasets

4.3.1 Spring

The Spring Framework is a Java application framework. The system connects several functionalities like an inversion of control container, support for aspect-oriented programming, or a model-view-controller framework for web applications. These functionalities are spread out among a large number of packages, indicating a high degree of modularization.

The period we are considering started in July 2008 when the development of version 3 of Spring started. After four milestone releases and three release candidates, Spring 3 was released in December 2009. The release of version 3.0 was followed by seven maintenance releases with minor version numbers. Meanwhile, development on version 3.1 started. Version 3.1 was released after two milestones and two release candidates in December 2011, just after the end of the time frame we consider.

Table 1 lists some figures describing the system, its structure and growth. The number of classes grows by about 50% while the number of dependencies doubles. Typically, we can expect a faster growth of dependencies than classes when adding new features. New classes have to be both connected among themselves and plugged into existing structures. This process may cause the breakdown of modularity. Comparing the values for density, average path length, and average in- and out-degree of its graphs, we found that the development team is upholding the modularity of the system. Furthermore, we observe that about 80% of classes that were in the system at one point are not there anymore in the end as shown by the number of total classes in the repository. We can attribute this to refactorings as well as deletions of obsolete classes. Given the growth in number of dependencies, the ratio of commits to commits that change the structure of the system is about 3:1, meaning every third commit changes one or more dependencies. This is also reflected in the ratio of bug-fixing commits to bug-fixing commits that change the structure.

4.3.2 Eclipse

As already mentioned, we retrieved the part of the Eclipse repository containing the JDT module and removed all tests and non-Java files. Eclipse started out as an Integrated Development Environment (IDE) but has since evolved into an extendable platform. The Java Development Tools (JDT) are a core part of the development environment, responsible for parsing and representing source code, among other things. Within JDT, there is a *core* package containing the most important features around which the rest of JDT is clustered.

In June 2002 and June 2004, Eclipse saw two major releases, 2.0 and 3.0. Our dataset covers an additional half year prior and past those two years. This includes version 2.1 and two minor versions leading up to it as well as three minor releases following it. In September 2004 we have one maintenance release following version 3.0.

In Table 1, we have a summary of some key figures of the system and repository data. The number of classes almost doubles in the span of three years while the number of dependencies grows by slightly less than 100%. However, we have a much higher turnover in dependencies, almost twice the number of the dependencies that are present in the end. Despite the turnover, the ratio between commits and structural commits is much smaller than in Spring, slightly more than 1:8. There are, however, many more commits than in Spring. Another difference is that Eclipse has many more bug-fixing commits. Similar to the ratio among commits, a smaller fraction of commits is structural. From these observations we can infer that Spring and Eclipse are indeed different in their internal structure.

As mentioned before, we assume that structural changes that cause a defect do not necessarily require a fix that changes the structure. For this reason, we chose to include bug-fixing commits without structural change in the vectors for structural change for each class. We tried two strategies, to add all bug-fixing commits and to add only bug-fixing commits preceded by structural changes. Surprisingly, we found the difference in total defects present in the different vectors to be very small (less than 5% difference). This means that classes that do have structural changes at all do not have many issues that require more than a single (non-structural) fix if any. This includes possibly interspersed non-structural changes which for this reason hardly occur neither. There is one major observation to be made. Changes in structure and changes in size are somewhat mutually exclusive (not considering the change in size that is a necessary consequence of structural change in any case) in Eclipse, specifically that we can distinguish sets of files for which one or the other holds true. This spells out the difference between classes that are themselves complex and classes that reuse functionality in other classes: one kind changes in size only, the other changes in structure as well. This observation is limited to classes with a considerable length of their history, though, in our case a history of more than 30 changes. It is interesting, however, that this difference is so pronounced for the more important classes of Eclipse. On the other hand, there are a few classes with histories shorter than 30 that have accumulated a lot of bug-fixes. In fact, for about a dozen classes, we find that they have been changed at most once not in a bug-fixing commit over a history of 20 or more changes. This corresponds to the findings in [20].

5. DATA ANALYSIS

5.1 Replicating our previous study

Spearman correlation between the sum of all values in the structural change vector, the churn vector, and the bug vector for each class is near 0 and significant for both systems. So there is no correlation. In our previous study, we used 74 releases of Spring and determined how many structural changes and bugs each class underwent and compared it to churn. We found rather strong correlations between change and defects, supporting similar results from the literature.

We explain this different outcome in two ways. At a low granularity, the fact that a class changed is already a pretty good indication of fault-proneness. Any change carries the risk of introducing a defect whereas the chance of having a defect in a class that did not change is considerably smaller. Even at a very high granularity, i.e. single commits, we find that large parts of Spring and Eclipse are essentially never modified structurally after their addition to the system (see Table 2 for the number of unchanged classes in each system in terms of structure). Typically, these classes have only an in-degree in our graphs, meaning that they are, for example, helper classes that provide functionality factored out for a single purpose from another class. Mistakes are rare in this kind of class. Very few classes do never have any churn, though. However, most classes have churn only in less than a handful of commits meaning that they reach their final state after few changes and are left unchanged afterwards. So we have a majority of classes that never or hardly ever change. This observation has to be kept in mind when analyzing the data.

On the other hand, we find that there is no correlation between structural change or churn and defects at a high level of granularity. Arguably, a large change does not necessarily have to be defective and vice versa. Other factors like the locus of a change in the overall system, the person who implemented the modification, and the purpose of the change play a role as well. So there is no straightforward relationship between the quantity and the quality of a change. Thus, any method using either quantity or quality of changes can benefit from using the other variable as well.

Another finding corroborates this argumentation. We calculated the similarity of the overall graphs for consecutive commits and wrote the values to a vector. In a second vector, we marked bug-fixing and non-bug-fixing commits. Using Granger Causality [6] (which was also used in [1] and is comparable to ARIMA used in [10]), we found that neither churn nor structural change Granger-caused defects for the overall system. So neither change at the class- nor at the system-level was correlated to defects at the scale of single commits.

However, if we evaluate our new scheme outlined in Section 3.2, we obtain very strong correlations. Stronger, in fact, than our previous results. Apparently, having longer histories requires a higher level of detail in terms of the dependency structure and where changes originate and how they propagate. On the other hand, though, when having longer histories it makes less sense to use correlations over the entire length of the histories. This insight is the basic reason for developing our new approach.

5.2 Evaluating our new approach

To answer our first research question, we analyze time

Table 2: Descriptive analysis of Spring and Eclipse time series. Length cutoff is 30, avg are rounded.

		Spring	Eclipse
NHK	avg length	9	98
	#long	124	405
	#stationary	3	188
	avg length stationary	70	408
	#non-stationary	121	217
	avg length non-stationary	55	155
	#short (< 30)	1381	753
	#0-length	1946	1875
Churn	avg. length	4	13
	#long	19	284
	#stationary	0	18
	avg. length stationary	-	166
	#non-stationary	19	266
	avg. length non-stationary	45	53
	#short (< 30)	3423	2596
	#0-length	15	213

series by their length. Table 2 lists the three different kinds of time series we derived: long, short, and 0-length time series. Long and short time series are defined by their length being greater or smaller 30 respectively. 0-length time series correspond to those classes that remain unchanged over all commits (i.e. with zero commits). Long time series are further classified into stationary and non-stationary. In the following, we preliminarily show that this classification of histories - long, short, stationary, non-stationary - can be used to characterize products' change history highlighting their differences.

Given the number of overall commits in Table 1, it is no surprise that there are fewer long histories in Spring than in Eclipse for both NHK and Churn. It is also no surprise that the average length of any type of time series for NHK is longer than the one for Churn. This is because the NHK time series values are positive also when neighbor classes change in structure whereas Churn times series values are positive only if they include churn of the class itself. What appears not to be obvious is that the number of unchanged classes is significantly greater with NHK. If compared with the total number of classes in Table 1, this indicates that half of the classes do not actually change their dependencies - and neither do their neighbors - whereas the majority of the classes have lines of code-changes at some point in time. If we further look at stationarity, we find very few (NHK) or no (Churn) stationary time series for Spring. This means that, in general, either class history is too short to draw any conclusions about stationary behavior or does not follow a stationary evolution - i.e. does not show a specific pattern - both for structural change and churn. For Eclipse, there are almost as many stationary as non-stationary NHK time series showing that there is a good number of classes whose dependency change behavior can be predictable over time. In addition, as structural changes always imply churn (but not vice versa) and as the number of stationary time series significantly drops for Churn, we can infer that the majority of classes with stationary NHK time series has either short or non-stationary Churn time series. This indicates that this majority of classes have predictable changes in their neighborhoods while their own changes are much more rarely predictable. This observation has implications for the localization of bugs, as discussed in the next section.

The few classes with stationary churn times series include some of the core classes of JDT, classes representing code and project entities, various editors, and parsers. As such, Spring and Eclipse are very different in their internal structure and history. In Eclipse, there are a few core classes that are under constant (probably planned) development and many more that are directly affected and influenced by them. In Spring, on the other hand, development activity shifts between components. There are no classes that have stationary Churn time series. The few that have stationary NHK time series are classes that are well-connected in the dependency graph of the system through many out-going edges. They accumulate NHK values through the changes of their neighbors. All of them have been added during the development of Spring 3.0, i.e. they are classes added during the time of our observation. They are abstract classes or standard implementations of interfaces in the converter functionality, for the Spring Expression Language, and for the resolution of beans.

To answer our second research question, we analyze bug-fixing commits and their density in long and short time series to see whether different types of history indicate different activity of bug-fixing and, therefore, different class defectiveness. Tables 3 and 4 summarize this analysis. To read these tables we first need to make two remarks. First, the overall NHK totals for Spring and Eclipse exceed the total number of bug-fixing commits of Table 1. This is due to the fact that we include bug-fixing commits without structural changes if they immediately follow a structural change for a file. As files in a commit have all churn, the totals for Churn instead correspond to the total number of bug-fixing commits. Second, as bug-fixing commits typically change more than one file, time series for different classes might share the same bug-fixing commits. In Table 3 we removed bug-fixing commits that occur in more than one time series eliminating bug-fixing commits of non-stationary time series that are already present in stationary time series and bug-fixing commits of short time series that are already present in the non-stationary time series. With these assumptions, data in Table 3 and Table 4 demonstrate again a clear difference between Spring and Eclipse. For Spring, most bug-fixing commits are in short time series, but with a relatively low density. In NHK long time series, bug-fixing commits are more and more dense in non-stationary time series whereas for Eclipse bug-fixing commits are more and more dense in the stationary ones. Long Churn time series also show a different pattern in that for Eclipse bug-fixing commits are much more dense in stationary time series (Table 4).

Overall, bug-fixing commits for structural changes in Spring and Eclipse occur in a different manner in that Eclipse bug fixes occur more in short histories but are more likely in deterministic long histories (stationary time series) than in Spring. In absolute churn values, non-deterministic occurrences of bug fixes in long histories (non-stationary time series) are the norm in both software products - particularly in Eclipse - but in relative churn values (densities), bug-fixing commits in Eclipse are again much more likely to occur in deterministic histories (46.2). As such, history types can tell much about defective classes and the structure of the product surrounding them. First, we can say that files that

Table 3: Bug-fixing commits in time series

		Spring	Eclipse
NHK	stationary	12	735
	non-stationary	62	290
	short	216	1,594
	total	290	2,619
Churn	stationary	-	831
	non-stationary	100	2,424
	short	349	1,291
	total	449	4,546

Table 4: Densities of bug-fixes.

		Spring	Eclipse
NHK	stationary	4.0	3.9
	non-stationary	6.4	1.3
	short	.15	2.1
Churn	stationary	-	46.2
	non-stationary	5.3	9.1
	short	.10	.50

change frequently and over a long time experience more bug-fixing activities. This supports the claim that changed files are defective [7], in particular files that changed in structure. In addition, as evolving systems inevitable change to accommodate new features and functionalities and changes typically revolve around the core parts of the existing system, our findings indicate that the most important files are the most defective. Second, histories of changes reveal different structures of the systems and the way defects propagate in them. More deterministic long histories of structural changes in Eclipse can be due to the many more code dependencies that propagate defects.

As already mentioned, there are always exceptions. There are about a dozen files in Eclipse that have short time series for churn but are almost exclusively changed in bug-fixing commits. This was also noted in [20] where they found that bug-fixes are three times as likely to incur further, subsequent fixes in Eclipse.

5.2.1 Shifting windows on time series

With our previous analysis, we see that length and stationary nature of time series influences defect density. As such, we might hypothesize that defects might be indicated by the local nature of commits time series. Therefore, to answer our last research question, we investigate time series with commit windows of varying sizes and their correlation with bug-fixes. In this way, we are able to relate the local behavior of class change histories (structural, churn, long, short, stationary, or non-stationary) and code defectiveness. In other words, we discuss the local evolution of a class change to observe defect incidence that can be used to identify fault-prone classes. In particular, we analyze the correlation between churn and structural change with bug-fixing commits in the given windows. For each class, we have two change vectors and two corresponding bug vectors. We shift windows across each pair of vectors and sum the values in each vector for the window. We add these sums as entries to pairs of vectors for all classes in the given category. The correlation values are all in Table 5.

For Spring, we get two very different results. The correlation for stationary time series is strong, but there are only few stationary time series. The correlations for non-stationary time series are about as strong as comparable results from literature ([23]). Interestingly, the correlation values increase slightly with larger windows. This may indicate that different periods in those time series are sometimes rather long. As there are no stationary time series, we do not get any correlations for stationary time series and churn (Table 3). The non-stationary time series are either weakly correlated or not significant. As opposed to structural changes, churn does not only vary over time for single classes, the amount of churn is not even an indicator for defects.

The correlations for Eclipse deliver a much clearer picture. Stationary time series for churn and structural change are strongly correlated to bug-fixing commits as compared to [23]. The strength of the correlations decreases with increasing window sizes, though. This means that, in fact, consecutive changes to a file do immediately contribute to the fault-proneness of that file. The same holds true for non-stationary time series for churn. Here, the immediate influence is even more evident through the much weaker correlations for larger windows. For non-stationary time series, correlations between defects and structural change are slightly negative. This indicates that changes and defects are somehow not related locally and any possible relation can occur in longer windows. Similarly, we found small, slightly negative, or no correlation when we investigated short time series. For example, for Eclipse, the correlation between structural changes or churn and defects for a window size of 10 and 20 is negative and below -.18. For Spring, the correlation is positive but slightly above zero (0.09). In both cases, change is not directly related to local activity of bug-fixing and can even indicate a specific stage in the development process in which the focus is not on fixing bugs [4].

As hinted at above, we used windows of sizes 10 and 20 on the short time series. We found no or only weak correlations for structural change for either system. For these classes, churn was a better albeit not a good indicator. For Spring, we obtained significant correlations of .174 and .384 for the two window sizes respectively. For Eclipse, significant correlations are at .283 and .472, respectively.

Our condition for stationarity is rather strong since both the metric vector and the bug vector have to have stationary behavior. Thus, the rows for non-stationary behavior in Table 5 also include time series where either vector might be stationary. We found that a further distinction between those categories did not immediately lead to greater insights. A more in-depth investigation of different combinations of kinds of vectors would, however, change the low or non-significant correlations for what are now non-stationary vectors. We have a strong statement about the characteristics of stationary classes here. Relaxing the current, strong condition would probably also entail relaxing other conditions. This is matter of future work. This strong condition allows us to derive an implication of the impact of structural changes, though. In previous work, structural change was usually measured in terms of fan-in and fan-out for each class individually. We show that regular changes in neighborhoods of classes can also be a solid predictor of defectiveness. A relaxation of our condition for stationarity might shed further light on this.

Table 5: Correlations between windows of changes and bug-fixing commits, all significant. Omitted entries are either not significant (*) or too few to calculate correlation (-).

		window	Spring	Eclipse
NHK	stationary	10	**.745**	**.604**
		20	-	.594
		30	-	.581
	non-stationary	10	.483	-.043
		20	.481	*
		30	**.506**	-.064
Churn	stationary	10	-	**.608**
		20	-	.586
		30	-	.572
	non-stationary	10	.164	**.514**
		20	*	.452
		30	*	.381

5.2.2 *Structural change and churn as complements*

In the description of Table 3, we mentioned that we excluded bug-fixing commits from the count where they intersected with a different time series. As a final measure, we explore how structural change and churn complement each other. After all, we can assume that there are some classes that change often and extensively in terms of structure and churn. Since this is about the number of changes rather than their behavior over time, we consider stationary and non-stationary time series together.

For Spring, we have total of 124 long time series and 74 bug-fixing commits for structural change, and 19 time series and 100 defects for churn. Nine classes have time series in both sets and there are 39 bug-fixing commits in both sets of defects. Overall, we find 134 files with 135 defects. This amounts to about 30% of defects in 3% of the files, which are probably the most important files in the system. If we lower the length threshold for the time series down to 10, we get 75% of the defects in 15% of the files as a compound measure of structural change and churn.

In Eclipse, this result is even more pronounced. For a length threshold of 30, we get a total of 599 files with 3,384 defects (19% and 74%). However, the intersection of classes for structural change and churn is a mere 90 and the intersection of defects is 896. This means that in 90 classes we find 896 defects. According to Table 3, these are 90% of the defects in all structural change time series. This means that if we neglect the other about 300 time series, we have gained 9% of the files with 72% of the defects. For both Eclipse and Spring, there are strong correlations for most of the time series we gather in the intersecting sets. This means that for most files, we have strong correlations between small windows on time series of changes and incidence of defects.

6. LIMITATIONS

We consider two systems over about three years of development time. Nevertheless, the average length of the time series for one system, Spring, was too short to make as much use of our distinction of stationary and non-stationary time series as we did for Eclipse. This includes the issue that our approach requires a considerable amount of history and changes. However, this requirement is based on using the ADF test. Replacing it with another measure for temporal change behavior may remedy this problem. Furthermore, we made several assumptions about the severity and impact of defects depending on where in the system they occur. It may be argued that defects in core classes are, on average, not as severe as in subordinate classes since they are more easily discovered. In any case, defects in core classes are typically more noticeable by users which makes them worse in this sense. We have mentioned before that the number of changes to a file is a good indicator for the defectiveness of that file. This is true if there are also periods of time where the file was not changed. Since we are considering only occasions where a file was changed in our change vectors, a boolean distinction for the occurrence of changes does not make much sense for us.

7. FINDINGS

In this section, we briefly recap the discussion of Section 5, highlighting the meta-level results we believe can be discussed and replicated for other software products.

We find that classes in two large software systems do, in fact, have varying temporal patterns in terms of structural change and churn. In particular, they are different in history length and stationarity. This differentiation then reveals different sets of files with different roles in the system and different defect incidence depending on the overall dependency structure of the systems. In particular, we find that the deterministic nature of histories is a promising instrument to discuss and identify core classes or classes prone to defects. In addition, we find that fine-grained correlations between changes and defects time series are strong, improving on our and previous results in the literature on defects in files. We also show that the two low-level measures that we employ (NHK and Churn) complement each other on both systems in identifying classes with a high fault-proneness.

8. CONCLUSION

Building on our previous work, we found that the level of granularity used for time and code artifacts in relation to defects oscillates between several extremes. We discussed the problems of different levels of granularity. Our method begins at a high level of detail and then differentiates between files based on the temporal characteristics of their changes. Then, after selecting sets of files, we accumulate sets of commits for these classes and correlate their changes to defects. We also outlined a possible way to navigate levels of detail in the dimensions of time, change, and files concerned. We started from a high level of granularity by using low-level code measures of structural change and churn on the one side and single commits as a proxy of time on the oder side. Unfortunately, the relationship between change metrics and defect incidence vanishes at this level of detail. Based on this observation, we hypothesized that the effect of this relationship must be better visible at file level and we analyzed the evolution of changes and defects for each class of the system. In the two projects we analyzed, we actually find that classes and their type of change history can describe structural differences, defect occurrences, and identify core classes. Based on this first result we hypothesized that difference in defectiveness would have been even more visible locally and we investigated the fault-proneness of individual files and how well defect incidence correlates

to changes in short windows of commits. The correlations turn out comparably well and we find that we can use intersections of files for the two kinds of changes to further guide the study of defect incidence. This way, we can also harness our strong correlations to a bigger extent. Finally, we observed that it is not only how much a file is changed but also for how long and how it has been changed that might reveal its proneness to defects. How long and how a file is changed before it needs some maintenance also depend on the specfic structure of the software product, though.

There are some open issues to our work. So far, we differentiate between structural change and churn. As mentioned before, these two metrics measure the complexity in the relationships between classes and in classes themselves. As demonstrated by the intersections, they do complement each other, though. Thus, a possible extension to this work is considering them together somehow. Similarly, we do not consider effects classes have on each other outside of these metrics. Including, for example, logical couplings could improve our results. Further, different strategies for the windows could be explored. For instance, different types of classes might be better described using different window sizes depending on the type's characteristics. Finally, our approach to structural change does not include information on the extent of a change in terms of the number of dependencies that changed. This information could be added as a weight to our measure.

9. REFERENCES

[1] C. Couto, S. Christofer, M. Tulio Valente, R. Bigonha, and N. Anquetil. Uncovering Causal Relationships between Software Metrics and Bugs. In *CSMR'12*, Szeged, Hongrie, 2012.

[2] B. Dagenais and L. Hendren. Enabling static analysis for partial java programs. In *OOPSLA '08*, pages 313–328, 2008.

[3] D. A. Dickey and W. A. Fuller. Distribution of the estimators for autoregressive time series with a unit root. *Journal of the American Statistical Association*, 74(366):427–431, 1979.

[4] J. Ekanayake, J. Tappolet, H. Gall, and A. Bernstein. Tracking concept drift of software projects using defect prediction quality. In *MSR'09*, pages 51 –60, may 2009.

[5] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Softw. Eng.*, 27(7):630–650, 2001.

[6] C. W. J. Granger. Investigating causal relations by econometric models and cross-spectral methods. *Econometrica*, 37(3):424–38, 1969.

[7] T. Graves, A. Karr, J. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653 –661, July 2000.

[8] S. Hido and H. Kashima. A linear-time graph kernel. *ICDM'09*, 0:179–188, 2009.

[9] P. Hofmann and D. Riehle. Estimating commit sizes efficiently. In C. Boldyreff, K. Crowston, B. Lundell, and A. Wasserman, editors, *Open Source Ecosystems: Diverse Communities Interacting*, volume 299 of *IFIP Advances in Information and Communication Technology*, pages 105–115, 2009.

[10] B. Kenmei, G. Antoniol, and M. Di Penta. Trend analysis and issue prediction in large-scale open source systems. In *CSMR'08*, pages 73 –82, april 2008.

[11] B. A. Kitchenham, L. M. Pickard, and S. J. Linkman. Evaluation of some design metrics. *Software engineering journal*, 5(1):50–58, 1990.

[12] R. Marinescu and C. Marinescu. Are the clients of flawed classes (also) defect prone? In *SCAM'11*, pages 65 –74, sept. 2011.

[13] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: currentÊresults, limitations, new approaches. *Automated Software Engineering*, 17:375–407, 2010.

[14] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE'05*, pages 284–292, New York, NY, USA, 2005. ACM.

[15] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *ISSRE'10*, pages 309 –318, nov. 2010.

[16] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *ISSTA'04*, pages 86–96, 2004.

[17] J. Ratzinger, H. Gall, and M. Pinzger. Quality assessment based on attribute series of software evolution. In *WCRE'07*, pages 80 –89, oct. 2007.

[18] J. Rosenberg. Some misconceptions about lines of code. In *METRICS'97*, page 137, Washington, DC, USA, 1997. IEEE Computer Society.

[19] B. Rossi, B. Russo, and G. Succi. Analysis of open source software development iterations by means of burst detection techniques. In C. Boldyreff, K. Crowston, B. Lundell, and A. Wasserman, editors, *Open Source Ecosystems: Diverse Communities Interacting*, volume 299 of *IFIP Advances in Information and Communication Technology*, pages 83–93, 2009.

[20] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05*, pages 1–5, 2005.

[21] M. Steff and B. Russo. Measuring architectural change for defect estimation and localization. In *ESEM'11*, pages 225 –234, Sept. 2011.

[22] A. Tosun Misirli, B. Murphy, T. Zimmermann, and A. Basar Bener. An explanatory analysis on eclipse beta-release bugs through in-process metrics. In *WoSQ '11*, pages 26–33, 2011.

[23] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *PROMISE'07*, May 2007.