Measuring Architecture Quality by Structure Plus History Analysis

Robert Schwanke
Siemens Corporation, Corporate Technology
Princeton, New Jersey, USA
robert.schwanke@siemens.com

Lu Xiao, Yuanfang Cai
Computer Science Department, Drexel University
Philadelphia, Pennsylvania, USA
{lx52, yc349}@drexel.edu

Abstract—**This case study combines known software structure and revision history analysis techniques, in known and new ways, to predict bug-related change frequency, and uncover architecture-related risks in an agile industrial software development project. We applied a suite of structure and history measures and statistically analyzed the correlations between them. We detected architecture issues by identifying outliers in the distributions of measured values and investigating the architectural significance of the associated classes. We used a clustering method to identify sets of files that often change together without being structurally close together, investigating whether architecture issues were among the root causes. The development team confirmed that the identified clusters reflected significant architectural violations, unstable key interfaces, and important undocumented assumptions shared between modules. The combined structure diagrams and history data justified a refactoring proposal that was accepted by the project manager and implemented.**

Index Terms—**measure, structure, change history, software architecture, fault prediction**

I. Motivation and Background

Software architecture, the "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" [7], often receives inadequate attention, especially in agile software development. For example, the architecture used in a small, agile project may comprise only a few principles, kept in the heads of the developers. However, as small projects grow up, they tend to postpone both architecture specification and necessary restructuring, which then become harder and harder as the code base grows, until the as-built architecture has degraded so severely that it needs major repairs. Sadly, the expense of restructuring is hard to justify — until there is a crisis.

This problem could be mitigated by explicitly tracking "architectural debt" – architecture-related technical debt – so that agile projects can decide when to pay some of it off. However, architectural debt tends to hurt future project sustainability much more than it does current functionality or quality. Therefore, even known architectural issues tend to go unresolved in the rush to deliver the next demo. When the crisis finally happens, the development team needs ways to prioritize the backlog of restructuring tasks, selecting which ones to include in its architecture renovation plan.

In this paper, we report a case study of measuring architecture quality and identifying architecture issues by combining analyses of dependency structure and revision history, following Wong, Cai, Kim and Dalton's [23] work on modularity violation detection through structure/history contrast analysis. This combined analysis allowed us to detect and locate architecture deviation/degradation, discover shared but undocumented assumptions that cut across module boundaries, correlate structural measures with faults and change proneness, and support a restructuring proposal with quantitative measurements and compelling diagrams. The proposal was approved and implemented, cleaning up a large number of the most fault-prone files.

This case study is part of an exploratory program in architecture analysis and improvement techniques. We are investigating the use of architecture-related software metrics both for (a) predicting the locations and impacts of future bugs, and for (b) uncovering and prioritizing current architecture risks. (We use bug and fault interchangeably, except as noted.) We conjecture that, once uncovered, mitigating a risk will also reduce both the metrics that uncovered it and the corresponding future bug impacts. If so, it would suggest that the metrics are measuring aspects of architecture quality.

II. Related Work

In this section, we review the related literature on structure measures, history measures, and fault prediction, which form the background for this research. For this case study, our focus is not on constructing the perfect predictor but on finding the most intuitive, easily obtainable measures that perform "well enough". We started with some representative measures, chosen for convenience, and explored how to use them to analyze software architecture. Future work will study tradeoffs between convenience and usefulness of the measurements.

Size Measures. Size measures are typically used to measure software productivity, predict fault locations for testing, and predict maintenance effort. There are two major types of size measures: 1) physical source lines of code which describe size in terms of the physical length as it appears for people to read (such as SLOC, NCLOC and KSLOC) and 2) logical source statements, which characterize size in terms of number of machine instructions or statements, such as DSI (delivered source instructions), or KDSI.

Park [18] commented that "Nothing in this report should be interpreted as implying that we believe one size measure is

more useful or more informative than another." Based on the assumption that different size measures are highly correlated to each other, and in particular that measures of lines of code are highly correlated with the number of bytes it takes to store them in files, we chose file size (in kilobytes, KB) as our size measure because it is widely available.

Complexity Measures. There are two types of complexity measures: complexity of individual files, and complexity of interconnections between files. McCabe complexity [11] directly measures the number of linearly independent paths through a program's source code. Henry and Kafura [6] were among the first to measure complexity by the fan-in and fan-out of cross-references between files. The widely studied CK metric suite proposed by Chidamber and Kemerer (CK) [3] identifies six object-oriented metrics. MacCormack, Rusnak, and Baldwin [10] measure the complexity (lack of modularity) of a system's dependency graph by its "propagation cost" and its "clustered cost". ("Cost" here refers to "path cost" in routing algorithms, not to real-world cost.)

In our work, we are trying fan-in (the number of modules that depend on a given module) and fan-out (the number of modules that the given module depends on) to measure the complexity of a file's relationships to other files.

Change Measures. The main information sources for measuring change are a project's version control system and its bug and task tracking system. The version control system tracks all the information about what has been changed, when and by whom. The bug and task tracking system records the tasks carried out by the project team, classified into task types such as bug-fix, new feature, or restructuring.

Mockus, Stephen and Karr [13] measured change by size (LOC added/deleted and number of sub systems affected etc.) and purpose. In our work, each time a given file is checked in to the version control system, we count that as one change to the file. We divide the change types into three categories: *bug-fixes*, *features*, and *unknown*.

Effort and Impact Measures. Effort in software projects is often measured by counting tasks, bug reports, file versions, change sets, or staff hours. Although staff hours are the actual units of effort in software projects, it is methodologically impossible to allocate staff hours accurately to any of the other units of impact above, because (a) real developers think about multiple files, bugs, and tasks at the same time, (b) a developer's productivity fluctuates widely due to uncontrolled variables in her environment, and (c) collecting the data relies on the developer's memory of how she spent her time. In this case study, therefore, we treat one change (one file check-in) as one unit of impact. Later, we will look into ways of allocating staff hours to changes.

Types of Changes. Many project repositories record links between change sets and the tasks for which the changes were made, enabling analysis by type of task. However, this kind of data often has poor quality [1], because the quality is hard to check at time of capture. In our case study, for example, the developers were expected, at time of check-in, to link each change set to the bug and/or task tickets for which the changes were made. However, the data revealed that some developers had provided ticket links most of the time, some provided them about half the time, and some rarely provided them at all, such that, for about half of the changes, we do not know whether they were bug-fixes or not. This did not make the exercise invalid, but it highlighted the importance of watching for data quality problems and adapting the analysis accordingly.

Software Fault Prediction from Project Repositories. The work of Ostrand, Weyuker and Bell [16] is representative of a wealth of research on how to analyze project development repositories to predict which files will have faults in the future. Their approach identifies a set of software metrics that correlate well with future fault detection, builds a prediction function out of them, based on negative binomial regression, and fits the parameters of the function to historical data of the project. They created a practical tool for prioritizing files to undergo system tests based on the predicted number of remaining bugs. Their tool consistently identifies 20% percent of the files that contain 71% to 92% of the remaining faults. They analyzed a wide variety of project types, differing in programming languages, application domains, and corporate divisions, etc. The tool is now in routine use in several ongoing development projects at AT&T Research. Other approaches use machine learning techniques to create classifiers that separate fault-prone files from others.

Prediction Performance. The usual criteria for comparing the performance of prediction methods include correlation, accuracy, precision, and recall. Correlation is commonly used when comparing predictions of scalar values, whereas accuracy, precision, and recall are used to measure the performance of classifiers [12] [24] [1] and information retrieval methods. In fault prediction work, correlation is often used to compare how well two methods predict the number of faults in a file, and accuracy, precision, and recall are used to compare how well two methods predict which files contain at least one bug, and which do not.

Ohlsson and Alberg [14] were among the first to use variants on the Pareto diagram to compare the performance of fault prediction functions, calling theirs the *Alberg diagram*. (The diagram notation dates back at least to 1905, to M. O. Lorenz [9].) They also noted that the usual correlation functions were not well suited for this comparison.

Fenton and Ohlsson [4] did an extensive study of faults and failures in a large system, reporting quite a few surprising observations such as the low correlation between bugs fixed before release and failures that happened after release. The biggest lesson from this study is that one must explore the data at hand with only minimal prior assumptions, before applying pre-defined analysis techniques to them.

Ostrand and Weyuker[17] noted that correlation is not a sufficient performance measure because the *precise* number of bugs found in a *particular file* is not as important as the *relative* number of bugs found in one file vs. another, and *total* number of bugs found in the *set of files* that one examines first. Furthermore, both accuracy and precision statistics treat false positives and false negatives as equally important, whereas in bug prediction, the consequence of a false positive (testing a

file that has no bugs) is much less severe than the consequence of a false negative (letting a buggy file escape into the field).

Instead, they specialize their performance measures to the way that the predictor functions are intended to be used: the files will be sorted according to their predicted number of bugs, worst first (descending order), and tested for bugs in that order. With that in mind, Ostrand and Weyuker advocate using two performance measures for bug prediction methods: the percentage of remaining faults found in the predicted-worst K files, and the percentage of faulty files *not* found among the predicted-worst K files.

Our work generalizes from this insight, focusing on three performance measures all based on recall: *faulty file recall*, *fault recall*, and *fault impact recall*, defined and discussed in later sections.

Correlation and Regression Analysis of Count Data. Most of the data we want to analyze fits the technical definition of count data. Statisticians define "count variables" as random variables whose values are restricted to the natural numbers (non-negative integers) and represent counted items, not ranks (e.g. 1st, 2nd, 3rd). Such data needs special treatments that are unfamiliar to many, because (a) ordinary least squares (OLS) regression behaves badly near zero, (b) many of the distributions are binomial, Poisson-like, or exponential, and (c) experimental data involving faults tends to be over-dispersed, for example by having more zero values than a standard Poisson distribution.

Therefore, we have refrained from using trend lines or OLS regression on our count data. For correlation analysis we use Kendall's tau-b rank correlation measure[8]. For constructing predictors; we will use *negative binomial regression* in our future work (Cameron and Trivedi [2]).

Architecture Violation Detection. Sangal, Gordon, Sinha, and Jackson [22] detect architecture deviations directly, by detecting the dependencies that violate the designed architecture structure. Their study focused on detecting violations of syntactic dependency specifications. Based on the assumption that the essence of software modularity is to allow for independent module-wise evolution, Wong *et al.* [23] detect modularity violations, supported by their tool, Clio, by identifying change coupling that is not explained by Robillard's heuristic [20]. Schwanke and Hanson [21] identify modularity errors by contrasting module membership with feature-based similarity.

During software evolution, if two components often change together to accommodate modification requests, but they belong to two separate modules, Clio considers this a *modularity violation*. Such violations may be caused, for example, by side effects of a quick and dirty implementation, or mismatches between requirements and the original architecture design. Wong *et al.*'s preliminary work has demonstrated the feasibility and utility of this approach in Hadoop and Eclipse. They identified hundreds of apparent violations in each studied system, and 40% to 66% of them were conservatively confirmed, either by code inspection, by subsequent changes, or by subsequent developer comments. (They were not able to contact the developers to ask questions, but used change logs and community discussion boards to find confirming evidence.)

We re-implemented the Clio concept of contrasting design coupling with change coupling, replacing certain academic assumptions with their industrial counterparts.

III. Case Study Questions

Although many structure metrics have been investigated for quantifying software quality, the connection between structure and quality is not yet firmly established. When project managers are faced with these sorts of measures, or with architecture deviation reports, the typical response is,

> "*What can I do with this information? There will always be some modules with high complexity, coupling, fan-out, or size. What will I gain by reorganizing them? Why do I care if the code violates the architecture, as long as it works?*"

We hope that our answer will be,

> "*These surface measurements point to deeper quality issues. Finding and fixing the quality problems will save time and effort – and the surface symptoms will also go away.*"

We reasoned this way: if we can show that a structure measure, say "fan-out", applied to today's code, correlates strongly with future bug-related effort, then we know that the developer can find bugs sooner by looking at files with high fan-out. It also suggests (but does not prove) that high fan-out itself is a cause of high bug-fix effort. Therefore, finding out why a file has high fan-out could lead to an explanation for why it is prone to high bug-fix effort. Furthermore, we can use this linkage to justify fixing the problems we find. In this case study, we aim to answer the following questions:

Q1. *Does this combined structure/history measurement reveal critical architecture issues that are worth fixing?*

Q2. *Are there any structure-based measures that can be used to predict quality variation in the absence of adequate revision history data?*

Q3. *What measurements could help the developers make important architectural decisions, and how?*

IV. Case Study Procedure

Our case study consisted of the following steps, with plenty of overlap and backtracking:

1) Data collection: we obtained access to the project's source code version control system and its task and bug tracking system.
2) Structure and history measurement: we selected and adapted well-known, easily-understood measures.
3) Validation: we validated the measures on the project's own data, from releases 1 and 2, showing which measures were good predictors of future faults.
4) Prediction: we used measurements from the first two release cycles to predict the relative number of fault-related changes to each file in the future (after release 2.0).
5) Uncovering architecture problems: we sorted the source code files by predicted future faults, then

a) Inspected each of the most fault-prone files in a graphical structure browser to see its role and connections in the architecture, and
b) Clustered "distant" pairs of files (defined below) that frequently changed together, inspecting each cluster graphically to uncover potential structure problems.

6) Presenting findings: We presented our analysis to the lead developers to find out which of the apparent problems were significant risks for the project. This inspired them to tell us about additional concerns.
7) Investigating developer concerns: we identified the developers' biggest concerns and combined our statistical and structural information to investigate and quantify them.

Notice that steps 5 to 7 rely heavily on human judgments. The potential issues we identified provided the symptoms to the experts, motivating them to do the deep analysis. We are not looking to algorithms for solutions, only to help us find problems. Solving them is, so far, still the job of humans.

V. Case Study and Results

A. The Project and Its Data

The project we studied, code-named *System J*, is a two-year old development project for an industrial software product in an emerging product domain.

We chose the project because we had unusually good access to project data and to the developers, but we tried to treat the developers as (cooperative) customers. This meant that we could not require that the developers spend time talking to us, nor could we impose work on them if they didn't want to do it voluntarily. The project has had up to 20 developers involved at any given time. It comprises about 300 KSLOC of Java in 900 files, in 165 Java packages. The system aggregates a certain type of data from many sources and uses it to support both market and operational decision-making at a time granularity of minutes to hours. It has a service-oriented architecture and a transactional database, both implemented with third-party platform technologies.

The software is being developed with an agile project discipline, where the project manager is also the customer proxy. The sprints are usually two weeks long, the system is rebuilt and automatically tested at least nightly, and each sprint ends with a customer demo and a retrospective. Fixing their bugs from the night before is usually each developer's highest priority. The entire software history is kept in a Mercurial [15] repository, with only one main development branch. At the end of each sprint, the current version of each file is tagged with the sprint ID, so that it is possible to go back to each tagged set and find the code for a complete system version that passed a known set of automatic tests. Bug and task tracking data is kept in a JIRA database (www.atlassian.com/JIRA). Every time code is checked into Mercurial, the developer is expected to insert tag(s) in the change log entry, mapping the set to JIRA ticket(s).

We treated System J's project and package structure as the "as-built" subsystem decomposition tree. System J comprises 25 Java projects, each containing a tree of packages, each containing multiple classes, each class in a separate file.

We extracted the case study data from the Mercurial, JIRA, and Understand™ repositories and stored it in a PostgreSQL object-relational database system. The database consists of 11 tables covering 4 aspects of project information: the file tables (paths, fileinfo_sprint, fileinfo_release), the revision tables (entries, commits, versions), the ticket tables (tickets and solveticket) and the people tables (persons and aliases). This database allows convenient exploration of the project data using simple queries, such as the histogram of change set sizes, or distribution of ticket types over time, as shown in Figure 1.

This article contains many real examples from *System J*. To protect company proprietary information, the domain-specific words appearing in file names and graph labels have been systematically replaced with words from the domain of gardening.

B. Measures of Structure and History

In this case study, we applied 6 single-file measures and 1 file-pair measure, all at the granularity of Java classes, with a one-to-one correspondence between Java classes and source code files. For each file f and each pair of files (f, g), we measure:

1) *File size*: Source code file size of f in kilobytes.
2) *Fan-in*: the number of source code references from other files to elements of f.
3) *Fan-out*: the number of source code references from f to elements of other files.
4) *Change frequency*: number of times that f is checked in.
5) *Ticket frequency*: the number of different JIRA task and bug tickets for which f was modified. This frequency is also broken down by ticket type: *bug*, *feature*, or *unknown*.

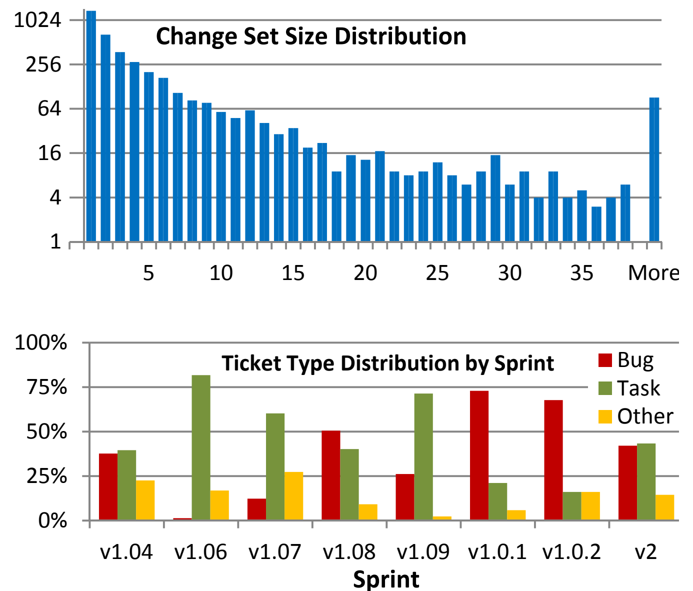

Figure 1: Changes set sizes, ticket types

6) *Bug change frequency*: the number of change sets that contain *f* and reference at least one bug ticket.
7) *Pair change frequency*: the number of change sets in which *f* and *g* both appear.

We chose the measures above to investigate first because they were readily available, easily applied, and widely understood.

C. Exploratory Data Analysis

To assure ourselves of the quality and relevance of the data, we used several exploratory data analysis techniques, including histogram inspection, scatter-plotting relationships, and comparing two sets of the same kind of data from different time intervals. The data we used spanned two development cycles of the subject system, release 1 (R1) and release 2 (R2). For some types of analysis we treated the changes during release 1 as “the past”, the code structure at the end of release 1 as “the present”, and the changes during release 2 as well as the structure at the end of release 2 as “the future”. For other analyses we combined the changes in release 1 and release 2 into a single, large time interval (R1+R2), “the past”, using the code structure at release 2 as “the present”, and subsequent changes as “the future”. Here are some analysis examples, which also begin to unfold the story of the data.

1) *Distribution of Each Measure’s Values.* We started by looking at the distribution of each measure’s values. For example, in Figure 2 we see a histogram of the number of changes per file. (Note the logarithmic scale, used to conserve column-inches.) It shows a typical exponential decay curve, up to a frequency of about 60. Beyond that, we see about two dozen “outliers” that change much more frequently than the rest. All six single-file measures had similar histograms.

2) *Scatter Plots of Relationships between Measures.* The first two measures for which we examined the outliers were fan-in and fan-out. While the files with high fan-out tended to be error-prone, those with high fan-in did not. Instead, they were frequently infrastructure classes, with many instances or many sub-classes, suggesting that “high fan-in” is architecturally significant.

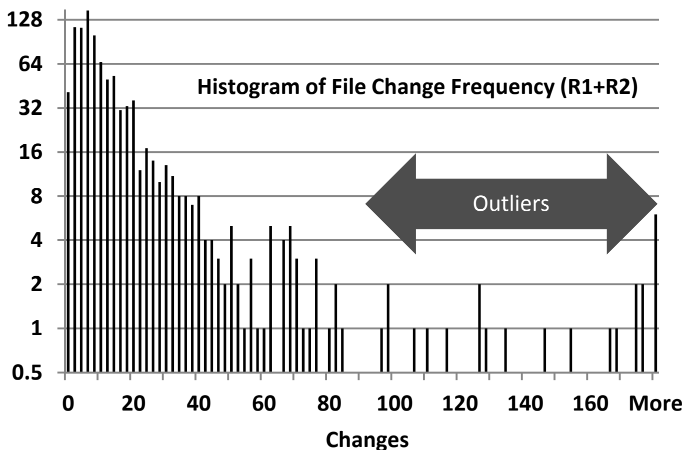

Figure 2. Histogram, showing outliers

We used scatter plots to compare each of the 6 single-file measures to each other. For example, the plot in Figure 3 shows a likely correlation between R1 fan-out and R2 change frequency. However, this plot reminds us that we cannot use the Pearson correlation measure, nor ordinary least squares (OLS) regression, on this kind of data, because most of the data points lie on or close to zero in some dimension.

Instead, we compute correlations using Kendall’s *tau-b* rank correlation measure [8], which calculates the extent to which two measures rank-order the same sample points in the same order.

$$\tau_B(F,G) \stackrel{\text{def}}{=} \frac{concord(F,G) - discord(F,G)}{concord(F,G) + discord(F,G)}$$

where $concord$ and $discord$ count the number of pairs of entities that are in the same order (resp. opposite order) in rankings F and G. The value of τ_B ranges from -1 (exactly the opposite order) to +1 (exactly the same order), with zero indicating no correlation. Pairs of points that have the same value under either of the measures (“ties”) are ignored in the calculation, and the correlation is not affected by how close or far apart two points are in the two orderings.

Table 1 gives the correlations between each pair of single-file measures, calculated over R1+R2. The table is ordered by how closely each of the other measures correlated with bug change frequency (**bugs**). It shows statistically significant correlations ($\alpha<1\%$) between each pair of measures except those highlighted, but the correlation is much weaker when one of the measures is fan-in.

3) *Comparing Release 1 with Release2.* We compared R1 and R2 values of all six single-file measures (see Table 2), and found that the R1 structure measures each had a strong

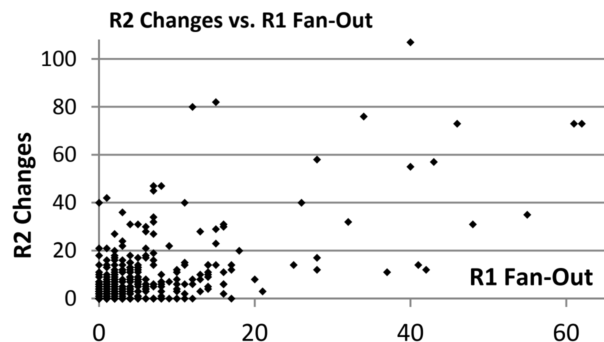

Figure 3. Scatter plot, suggesting correlation

Table 1: Rank correlations between measures

R1+R2	fan-in	fan-out	size	changes	tickets	bugs
fan-in	1	-0.028	0.023	0.108	0.092	0.086
fan-out	-0.028	1	0.454	0.460	0.443	0.445
size	0.023	0.454	1	0.507	0.529	0.578
changes	0.108	0.460	0.507	1	0.628	0.659
tickets	0.092	0.443	0.529	0.628	1	0.941
bugs	0.086	0.445	0.578	0.659	0.941	1

Table 2: Rank Correlations: R1 vs. R2

R1 vs. R2	vs. same measure	vs. bugs
fan-in	0.921	0.171
fan-out	0.942	0.508
size	0.902	0.598
changes	0.478	0.586
tickets	0.650	0.648
bugs	0.661	0.661

correlation (greater than 0.90) with the same measurement in R2, confirming that structure changes slowly. For history measures, the same-measure correlations between R1 and R2 were not as strong, but still evident, ranging from .48 to .66. These correlations give us some confidence that each measure can at least predict future values of itself.

We also took an advance peek at how well these measures could predict bug-related changes. The correlations "vs. bugs" in the table above show that the R1 values of each of them has a significant correlation with R2 bug-related changes, and that, except for fan-in, the correlations are competitive with each other.

From inspecting the scatter-plots and histograms and calculating the correlations, we inferred that all of the 6 single-file measures were sufficiently well-behaved to be suitable for the next step, except that fan-in would be a poor predictor of bugs. By contrast, most of the pair change coupling measures we looked at were ill-behaved or hard to relate to practical tasks. We decided to use only *pair change frequency* at first, and revisit the other pair change measures later.

D. Validation on Project Data

Extending the approach of Ostrand and Weyuker [17], we chose three variations of information retrieval's *recall* measure as the primary bug-prediction performance measures. We assume that the software measure whose performance is being evaluated will be used to sort the files, "worst first", after which the developers will test or examine the first K files to look for faults. *Recall* is defined as the fraction of the relevant 'instances' that are 'retrieved' by examining those K files. The value of *K* is uncertain, because it will depend on available resources and competing tasks, so the measure should perform well over a wide range of values for *K*. Therefore, the performance of a bug prediction method is the area under its recall curve. The three variations differ by what an 'instance' is:

Faulty file recall. An 'instance' is a file that is changed (in the future) at least once due to any bug ticket, but additional bug tickets and additional changes don't count extra.

Fault recall. An 'instance' is a pair <file, bug ticket>, where the file is changed at least once due to *that* bug ticket. Additional changes *due to the same ticket* don't count extra.

Fault impact recall. An 'instance' is a triple <file, change set, bug ticket>, where the file is a member of the change set and the change set is associated with the bug ticket. If a file is checked in several times *due to the same bug ticket*, each time is a separate instance.

Fault recall puts more emphasis (compared to *faulty file recall*) on files with multiple faults than on files with just one fault. *Fault impact recall* puts even more emphasis on faults that are hard to fix (inducing multiple check-ins).

The validation process calculated the measures on R1 data and used them to predict bugs that were found in R2. It then plotted each measure's R2 recall curve for each of the three recall performance measures. Figure 4 shows the R2 *fault impact recall* curves. These curves are *Alberg diagrams* [14]. (Out of 528 files, the grid line at 106 represents the first 20% of the files. The graph stops at 264 files (50%) to save column-inches, without loss of insight.)

Each colored curve on the chart represents sorting the files according to a different measure, for example, decreasing size, decreasing fan-out, decreasing past bugs, etc. A given point *<X,Y>* on curve *Z* means that, when sorted according to measure Z, the first X files will incur Y% of the total R2 bug-related changes.

The top curve, labeled "Oracle", represents the best possible performance, achievable only if the files are sorted in descending order of R2 bug-related changes. The remaining curves are closely spaced. Each of them shows 70% to 80% bug impact recall with the first 20% of the files. This performance is consistent with the findings of Ostrand, Weyuker, and Bell [16].

Combining the measures was a little better than using just one. *Median R1 Rank* looked at each file's rank with respect to code size, fan-out, and past changes, and used the median value of these three ranks to sort the files. Thus sorted, 20% of the files accounted for about 80% of the R2 bug-related changes.

We also compared the performance of these measures for fault recall and faulty file recall. The spacing and ordering of

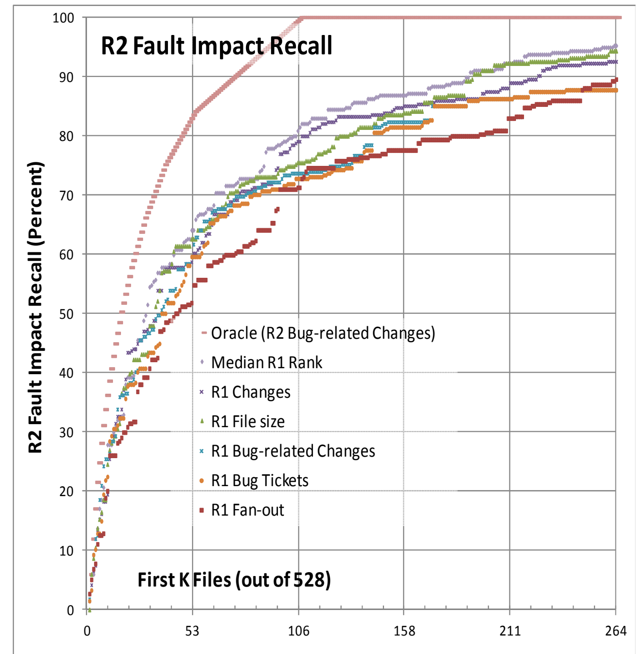

Figure 4: R2 fault impact recall

the curves varied little, but the curves for faulty file recall were lower: the top 20% of the files by each stand-alone measure identified only 50-60% of the buggy files.

Doing less well on the faulty file recall measure is not surprising, because a file with only a single fault has little effect on the other performance measures, but has the same effect as a file with a dozen faults or a hundred bug changes under the faulty file recall performance measure.

This analysis gives us optimism that ranking the System J files according to one or more of the top 5 measures will give a good order to look at the files. Although it is tempting to draw further inferences, these experiments did not analyze nearly enough data to distinguish among the top 5 measures, and, since they only involved one project, do not tell us what to expect on other projects.

Further research will be needed to create an optimized composite measure, presumably using negative binomial regression, and measure its performance.

E. Uncovering Architecture Problems

Having confirmed that the R1 measures predict R2 faults, we next used them to discover architecture issues by: analyzing outliers; visualizing their positions within the architecture; and, clustering "distant" files that frequently changed together.

Since we would need the developers' knowledge and good will to complete the analysis, we could no longer treat a date in the past (the end of R1) as if it were "the present". Since the analysis took place at the time that R2 actually ended, we tabulated the measures on R1+R2 and used them to predict where faults would be found in the next release, "R3". We would judge our success solely by the developers' reactions and follow-up actions.

1) Outlier Analysis – Individual Files. Table 3 lists the predicted fault-prone files "worst first", sorting them according to the median of their ranks by R1+R2 fan-out, size, and change frequency.

First, we noticed that at the very top of the list, all five measures seem to agree. The top 8 files were ranked among the top 14 by every measure.

The first time we tried to present such results to the developers, to elicit feedback, we just showed them a list like this and asked them to comment on each of the top 10 or so.

Table 3: Most error-prone files, by median R1 rank.

file	fan-out	size	changes	bugs	tickets
Che	8th	5th	3rd	1st	1st
Fig	6th	7th	2nd	4th	2nd
Yew	1st	4th	1st	5th	7th
Abiu	2nd	8th	11th	2nd	3rd
Bael	5th	12th	5th	3rd	6th
Date	11th	6th	12th	7th	5th
Duku	4th	10th	9th	8th	8th
Imbe	3rd	14th	4th	9th	9th
Lime	51st	19th	13th	6th	4th
Neem	160th	11th	7th	11th	11th

However, we found that staring at lists of file names and measure data was overwhelming; the developers gave us little response. So, before going back to them, we tried investigating the top few files ourselves, using a graphical structure browser.

2) Structure Browsing with Understand™. Scientific Toolworks, Inc. makes a static code analysis tool called *Understand*™. It creates a database of code structure and cross-reference information, processes queries on this information, and renders the results using Graphviz, a popular, free graph drawing engine originally developed at AT&T Research. Understand's structure diagrams render files and dependencies as boxes and arrows, and render sub-system structure as nested boxes. Its graph browsing user interface provides a convenient way to incrementally refine a query until you get the diagram you want.

Using this tool, we would analyze a file by creating a diagram that included the file itself and all of its immediate neighbors in the cross-reference model, including the packages in which these files are located. Showing the enclosing packages provides guidance for analysts who are unfamiliar with some of the files.

Such a diagram can then be further customized, e.g., by adding neighbors-of-neighbors, to help the code expert discover or remember what was causing the file to be fault-prone. We found that showing such diagrams to the code experts elicited many more comments than the lists of data alone. Sometimes the comments were directly related to the diagram at hand, but sometimes the diagram reminded the developer of another issue, not seen in the diagram at all. The point is that helping the developers visualize structure was an effective problem elicitation method.

By this method, we were able to discuss many of the most fault-prone files with the developers and, in most cases, either draw their attention to a problem they hadn't noticed, or give them concrete evidence of a problem they had already suspected. In two cases, however, the developers stood their ground, saying that, although the structure looked suspicious, it was necessary for the job that the class in question was doing.

3) Outlier Analysis – Pairs and Clusters. In addition to measuring individual files, we investigated the structure and history of pairs and clusters of files. According to Parnas [19], the decomposition of a system should be based on mature, stable design decisions and should encapsulate known future variabilities, so that the impact of each kind of future change is as limited as practical.

We first applied the approach of Wong *et al*. [23] to System J, unmodified, but encountered several challenges. First, Clio's algorithm assumes that each change originated in a "starting change set" and was propagated to others. However, in reality, when developers commit changes, the "starting change set", if it exists, is only in the developer's mind and there is no way for us to determine later what it was.

Second, Clio's definition of *modules*, that is, structural proximity, was based on a transitive closure of Robillard's relevancy heuristic, weighted by path length. However, the heuristic itself was hard to explain, and not experimentally validated by Wong.

We also realized that, in industrial settings, each project – indeed, each developer – may have a slightly different way of mapping code structures, files and folders to modules and subsystems. We should first agree on what a *module* is, before presenting the "modularity violations" to the developers.

To overcome the challenge of not having the "starting change set", we chose to ignore the question of which file or files "caused" a change, and simply counted the frequency with which pairs of files changed together. To avoid conflicting definitions of "module", we used the project's own definition, namely, that a Java package is a module.

We then defined structural proximity in terms of "local" and "distant" change pairs: a change pair <X,Y> is considered "local" if (a) class X depends directly on class Y, or Y depends on X, or (b) X and Y belong to the same Java package. Otherwise they are considered "distant". This definition will be refined in future research to consider dependency path lengths greater than 1 and to consider nested packages.

Based on these two modifications, we used a simple "single link" clustering algorithm to group distant change pairs: the similarity between two distant files was defined to be their joint change frequency, provided that the frequency exceeded a specified threshold. Each file that was part of at least one frequent, distant change pair was placed in the same cluster as the distant file with which it changed most frequently. After all distant file pairs were thus clustered. Any remaining files were then added to existing clusters based on local rather than distant change pair frequencies. (Exploration of better clustering algorithms is left to future work.)

For each cluster, we generated a structure diagram containing the cluster members themselves and their shared neighbors in the dependency graph. These diagrams give a sense of how severely a cluster cross-cuts the layer-dependency architecture. Their shared neighbors give hints about why they are frequently changed together.

For example, Figure 5 shows the four files (highlighted in yellow) that made up the three most frequent, distant change pairs in System J, along with five of their shared neighbors (colored lavender) in the dependency graph. Two of the files belong to RestEJB and two belong to Yew. Together, the four files and the five shared neighbors span four of the five main system layers. This suggests that the changes have been rippling across many, far-flung parts of the system.

In another cluster (not shown), a file in the Entry Points layer was changing jointly with several key files in the DataAccess package. That Entry Point file was also ranked number one in the whole system by fan-out. It looked as if changes in the Data Access layer were propagating to this Entry Point file, suggesting that some critical architectural interfaces are not stabilized. This observation eventually led to the renovation task described in the next section.

We also inspected the source code of the most-frequently changed pairs, to see if the reasons they were changing together were obvious. Some example reasons we found were: cloned code; logic moving from one file to the other; and shared dependencies that caused them to receive the same propagated changes.

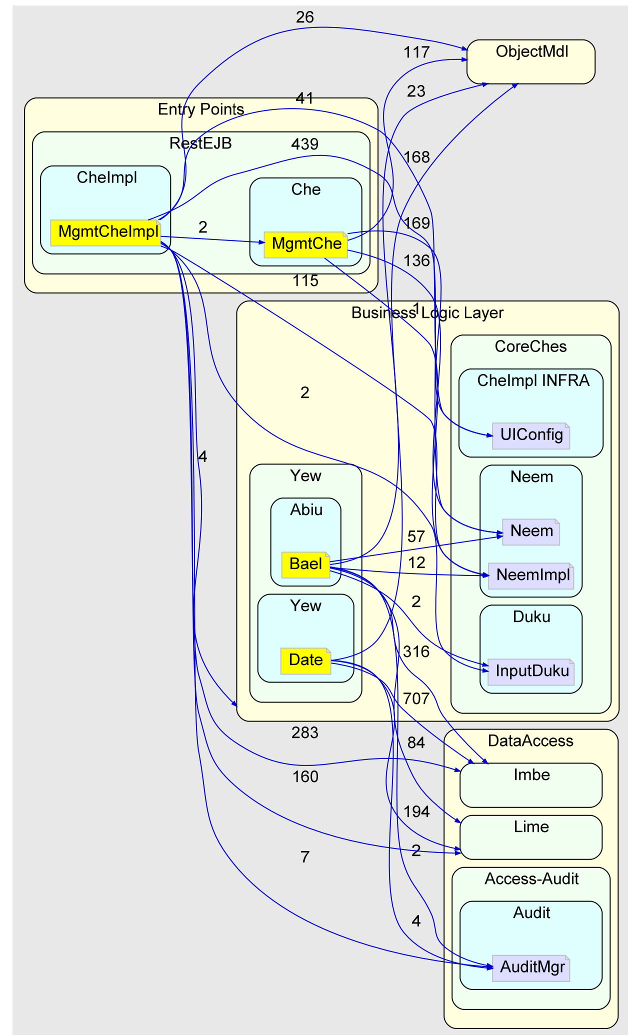

Figure 5. Three most frequent, distant change pairs

In two cases we found shared, unencapsulated assumptions about the representation of time. In one case, the changes happened when the granularity of time changed from minutes to milliseconds. In another case, the representation of time intervals split into two forms: "[a,b)" and "(a,b]". (That is, both forms are half-open, but one is open at the beginning and the other is open at the end.

Although space does not allow us to describe these examples more fully, in most cases we found a definite cause for the files changing together that was not documented and not readily detected by dependency analysis. Therefore, unusually high joint change frequency data, in the absence of syntactic "explanations", seemed to be significant and useful information.

F. Investigating a Developer Concern

In the course of discussing the R1+R2 outliers with the developers, they began volunteering information about their own architectural concerns. We chose a few of those concerns to investigate, producing measurements and diagrams to quantify and illustrate them. For example, a senior developer noticed that several of the most fault-prone files belonged to

the same entry-point package, and mentioned that there was "a lot" of business logic being written in the entry point layer instead of the business logic layer. We used the layer-dependency model to investigate and illustrate the problem, as shown in Figure 6, where (we were told) all of the connections from *RestEJB* to *dao-SIMPLE* and *dao-Carob* violate the architecture. We found a total of 8 files with many of the same violations. One consequence was that the misplaced business logic was not being adequately tested in the nightly builds, because the test driver bypassed the entry-point layer to test the business logic directly. Inadequate testing in turn led to late detection of bugs and many bug-related changes. The root cause was inadequate architecture awareness by junior developers. The senior developer wrote a small renovation proposal, using our data and diagrams to justify the work of moving the business logic to the proper layer. After the proposal was approved and the renovation completed, many of the top 50 fault-prone files had been moved, split, or deleted, in order to move the business logic to the proper layer. Afterwards, the senior developer asked to see the dependency diagram again, after the clean-up – and spotted a faulty file she had overlooked!

Note that our measurement methods did not, by themselves, diagnose the problem. Also, the moving, splitting, and deleting were only side-effects of the clean-up. It was the synergy between measurement, prediction, visualization, and expert insight that led to a successful renovation.

VI. Discussion of Results

A. Answering the Case Study Questions

Q1. *Does this combined structure/history measurement reveal critical architecture issues that are worth fixing?*

Indeed, by combining evolution history information with file dependency structure, we were able to identify the following types of issues. First, there are key interface files that are supposed to be stable and correct, but actually have faults and change frequently. Second, frequent, distant change pairs often correspond to important architecture weaknesses or violations. Third, structure gives us a visual context for analyzing change associations found in the history, discovering important, but undocumented shared assumptions that should be made explicit.

Q2. *Are there any structure-based measures that can be used, without history, to predict quality variation?*

For System J, size and fan-out were each fairly good stand-alone predictors of fault-proneness. When a large file also has high fan-out, it should be examined for accidental complexity and architecture violations. Although fan-in was not a good predictor of quality, high fan-in files did tend to be architecturally- significant infrastructure classes.

Q3. *What measurements could help the developers make important architectural decisions, and how?*

In addition to the well-known measures we set out to apply, we discovered several helpful evaluation techniques. Outlier investigation was a good way to find low-hanging, bad-smelling fruit. Predicting fault-proneness helped to prioritize

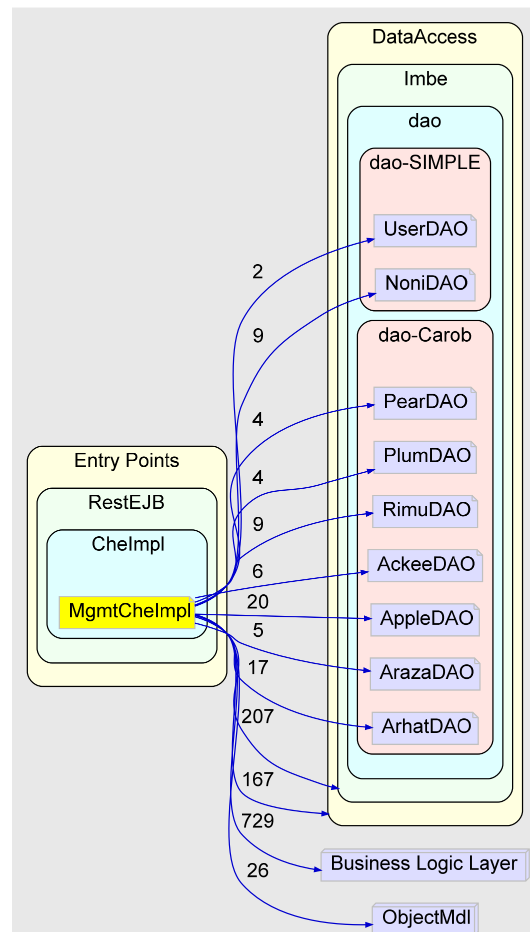

Figure 6. Entry point class with business logic

architecture risks. Clustering distant change pairs seemed to bring together locality violations with the same root cause.

B. Visualizing Structural Concerns

Although the software metric outliers often pointed to architecture issues, we had trouble getting the developers' attention until we showed them layer-dependency diagrams that illustrated the problems. These diagrams also elicited other architecture issues we had not yet found. From this experience we are convinced that interactive, visual architecture models are essential for analyzing and *communicating* architecture issues.

C. Overcoming Dirty Data

The incomplete data linking change sets to JIRA tickets confirmed the discoveries of others, but it also suggested its own solution(s). Future studies should check for such problems early, check correlations between change types, developer habits, and other measurable attributes, then tailor the prediction functions to account for them.

D. Novelty and Effectiveness

To our knowledge, this is the first case study that demonstrates the applicability and utility of both combining and contrasting structure and history measurements in a realistic industrial setting. The clustering algorithm, adapted

from Wong *et al.*'s work, makes it possible to apply modularity violation detection techniques in a practical way. Our case study is effective in that the experiment had minimal disturbance to development process. Instead, all the measures and data mining and analysis were accomplished independently and we only reported to the team when our analysis suggested suspicious problems. Finally, since System J uses JIRA and Mercurial, which are both popular project support tools, we expect that the approach is repeatable on other projects using similar tools.

E. Assumption and Limitations

This work is limited first because we only studied one project, and we cannot claim that the result can be generalized to other projects with different domains or using different tools or languages. We only used the simplest structure and history measures. In the future, we plan to apply more advanced measures, such as conditional change probability, shortest directed/undirected dependency path, age-weighted history measures, and betweenness centrality [5].

VII. CONCLUSION

We have reported one case study of measuring code structure and history to identify architecture problems in an industrial project. It demonstrated the effectiveness of the combination by enabling developers to visualize the architectural roles and impact scope of files with high fault and change frequency, which justified and supported a successful renovation task.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under grants CCF-0916891, CCF-1065189 and CCF-1116980.

REFERENCES

[1] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced? Bias in bug-fix datasets," ESEC-FSE'09, August 23–28, 2009, Amsterdam, The Netherlands.

[2] A.C. Cameron, and P.K. Trivedi (1998). *Regression Analysis of Count Data.* Cambridge University Press. ISBN 0-521-63201-3

[3] S.R. Childamber, and C.F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on Software Engineering, vol.20, pp. 476-493, 1994

[4] N. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," IEEE Transactions on Software Engineering, vol. 26, pp. 797-814, August 2000.

[5] L. Freeman (1977). "A set of measures of centrality based upon betweenness," Sociometry 40: 35–41

[6] S. M. Henry, and D. Kafura, "Software structure metrics based on information flow," IEEE Transactions on Software Engineering, vol. 7, pp. 510-518, 1981.

[7] Institute of Electrical and Electronics Engineers. *Systems and Software Engineering – Architecture description.* (ISO/IEC/ IEEE Std 4210:2011). New York, NY: Institute of Electrical and Electronics Engineers, 2011. Also iso-architecture.org/42010/

[8] M. Kendall, "A new measure of rank correlation," Biometrika 30 (1–2): 81–93, 1938.

[9] M.O. Lorenz, "Methods of measuring the concentration of wealth," *Publications of the American Statistical Association* (Publications of the American Statistical Association, Vol. 9, No. 70) **9** (70): 209–219, 1905.

[10] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: an empirical study of open source and proprietary code," Manage. Sci., vol. 52, pp. 1015–1030, July 2006.

[11] T. McCabe, "A complexity measure," IEEE Transactions on Software Engineering, vol. 2, no. 4, pp. 308–320, Dec. 1976.

[12] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," IEEE Transactions on Software Engineering, vol. 33, pp. 2-13, January 2007.

[13] A. Mockus, G. E. Stephen, and A. F. Karr, "On measurement and analysis of software changes," 1999.

[14] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," IEEE Transactions on Software Engineering, vol. 22, pp. 886-894, December 1996.

[15] B. O'Sullivan, *Mercurial: The Definitive Guide.* O'Reilly Media: 2009.

[16] T.J. Ostrand, and E.J. Weyuker, and R.M. Bell, "Predicting the location and number of faults in large software systems," IEEE Transactions on Software Engineering, vol 31, pp. 340-355, April, 2005.

[17] T.J. Ostrand, and E.J. Weyuker, "How to measure success of fault prediction models," SOQUA '07 Fourth international workshop on Software quality assurance, pp. 25-30, 2007.

[18] R. Park, "Software size measurement: a framework for counting source statements," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU/SEI-92-TR-020, 1992.

[19] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," CACM, 15(12):1053–8, Dec. 1972.

[20] M. P. Robillard. "Topology analysis of software dependencies," TOSEM, 17(4):18:1–18:36, Aug. 2008

[21] R.W. Schwanke and S.J. Hanson, "Using neural networks to modularize software," Journal Machine Learning, vol.15, pp. 137-168, May 1994.

[22] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," In Proc. 20th OOPSLA, pages 167–176,Oct. 2005.

[23] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," Proceedings of the 33rd International Conference on Software Engineering, pp. 411-420, 2011.

[24] T. Zimmermann, and N. Nagappan, "Predicting defects using network analysis on dependency graphs," ICSE '08. 30th International Conference, pp. 531-540, May 2008.