Walls, Pillars and Beams: A 3D Decomposition of Quality Anomalies

Yuriy Tymchuk, Leonel Merino, Mohammad Ghafari, Oscar Nierstrasz
SCG @ Institute of Informatics - University of Bern, Switzerland

***Abstract*—Quality rules are used to capture important implementation and design decisions embedded in a software system's architecture. They can automatically analyze software and assign quality grades to its components. To provide a meaningful evaluation of quality, rules have to stay up-to-date with the continuously evolving system that they describe. However one would encounter unexpected anomalies during a historical overview because the notion of quality is always changing, while the qualitative evolution analysis requires it to remain constant.**

To understand the anomalies in a quality history of a real-world software system we use an immersive visualization that lays out the quality fluctuations in three dimensions based on two co-evolving properties: quality rules and source code. This helps us to identify and separate the impact caused by the changes of each property, and allows us to detect significant mistakes that happened during the development process.

Demonstration video: https://youtu.be/GJ8BONoaF0Q
Dataset artifact: http://dx.doi.org/10.5281/zenodo.56111

This paper makes heavy use of colors. Please read a colored version of this paper to better understand the presented ideas.

I. Introduction

As a software system evolves, its architecture tends to erode, making it harder to understand, change, test and debug [1]. There are different ways to counter the erosion such as code reviewing or precise testing. We are focusing on code quality tools that use static analysis [2] to detect *critiques*: violations of certain rules in the source code. This kind of tool can run automatically and supply developers with important information whether they are developing, reviewing, or sending changes for an integration [3], [4].

Despite the benefits provided by static analysis tools developers are often reluctant to use them due to a large number of false positives [5]. Thus it is natural to update the metrics and thresholds of the tools to suit the architectural requirements and decrease the quantity of false positive reports. For example to improve the acceptance of the Tricorder static analysis tool by Google developers, every rule with more than 10% of false positives was either improved or removed completely [6].

Challenges arise when one has to assess whether the quality tools that are the part of a development process do their job well. The easiest way to do this is by analyzing evolution of the main property that they should affect: quality of code. However it is not easy, because the measure of quality always changes together with the rules' evolution. For example version n contained 5 critiques from a rule that was removed (together with its critiques) in version $n+1$. The removal of 5 critiques in the new version was not caused by improvements in the code but rather by changes of the quality measure: the rules.

We were asked to perform such an analysis on a one year development cycle of a real project measuring 520 thousand lines of code. While not realizing the fact of rule evolution we tried to identify which changes happened to the critiques through the development history. We encountered versions where the number of changed critiques was as high as 5% of the total number of methods in the software system. We manually examined these anomalies and detected that in one of them a rule was fixed to include the violations that it was previously ignoring. Another anomaly was caused by unloading of a big module.

We could not relate the observed anomalies to a single cause, which restricted us from using already available visual and statistical methods. To understand the nature of our data set we created a visualization that uses the changes between quality values as building blocks and lays them out in three dimensions: *software components*, *quality rules* and *software versions*. The visualization relies on the sparse nature of the data, and pre-attentive clustering possibilities of human brain to quickly detect the anomalies. This approach allowed us to see a high level overview of our data set, distinguish the anomalies caused by rule changes from the ones caused by software changes and finally clean the data from the anomalies. Moreover with our approach we identified a dozen anomalies that reveal bad practices or wrong design decisions and can be valuable for the stakeholders.

This paper makes the following contributions:

- A visual approach that enables analysis of evolution anomalies for values dependent on two co-evolving properties: quality rules and source code;
- A case study that demonstrates how the visualization was used on a real project to answer the questions of stakeholders;
- Summary of the changes that can cause anomalies in a historical data of critiques.

Structure of the Paper. Section II provides a description of the problem that we are trying to solve based on a real project. Section III contains an overview of the related work. In Section IV we describe the visualization and its features. In Section V we demonstrate how the problem can solved with our visualization, and present findings. We discuss different aspects of the visualization based on the use case in Section VI. Section VII concludes the paper.

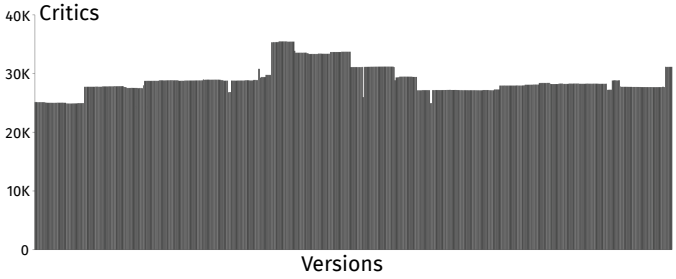


Fig. 1: Critiques histogram

II. Problem Description

The Pharo[1] development environment is an open-source software system developed by both employed engineers and individual contributors [7]. It consists of approximately 92'000 methods and 5'500 classes grouped into 240 packages. The most recent development cycle of Pharo lasted for one year during which 680 incremental updates (also known as versions or patches) took place. During this period, Pharo developers used SmallLint [8], a static analyzer to validate the quality of their code based on 124 unique rules some of which changed over the time. Each rule belongs to a category (like "bugs", "style", "optimization" and others) and specifies its severity (information, warning or error). A violation of a quality rule by a piece of source code is called a *critique*.

SmallLint is used in two stages of Pharo's development process. First of all an integration tool validates each contribution with a large subset of rules before integrating it. This practice has been present in the development process for 3 years already. Secondly, during this development cycle a new tool was integrated. It provides live information about the critiques of the source code that a developer is working on. Maintaining the tools and ensuring a certain development workflow requires additional resources, so the stakeholders who make decisions about Pharo's future features and development process wanted to know whether the tools that they use to maintain good quality of code have a real impact.

Initially we conducted a survey and asked the developers whether they find one of the tools useful [9]. The feedback was positive, but we assume a certain level of bias from the developers who were pleased to obtain a new tool to work with. Although the survey showed that the developers like the tool, it did not answer the question of whether the tool positively influenced the quality of code. To answer this question we decided to analyze the changes in quality throughout the last development cycle. However, we had to work with contemporary critiques: the ones reported to developers when they actually used the quality tool, not post hoc critiques reported by rules introduced later.

Running SmallLint on all versions in this development cycle produces around 19.5 million critiques. Figure 1 shows the total number of critiques per each version. Many of the critiques are related to essential complexity [10] and never change. The number of critiques that were added or removed from version to version is around 64.5 thousand. Inspection of the plot quickly reveals many versions where the number of critiques changed by as many as 5'000 or 15% from the version's total. We refer to these changes as *anomalies* because according to our investigation they are clear outliers and are unlikely to have been caused by a common refactoring or feature implementation.

During the manual investigation of a few anomalies we analyzed the data from the versioning system to understand which source code changes caused the anomalies. Additionally we analyzed the issue tracker entries linked to the software patches to understand what was the reason behind the changes. We discovered that in some cases developers were fixing a faulty rule, cleaning code from certain critiques, or simply applying a refactoring. Here we provide an example of the anomalies that we selectively inspected:

1) An increase of 5'000 critiques, all of which were reported by a single rule responsible for detecting unused methods. Previously the rule was broken. After a fix it reported all 5'000 unused methods. The anomaly does not represent a change in quality: the methods were present previously, but not reported. The anomaly itself can be interesting for stakeholders to learn about improvements happening to the quality support system and their impact.
2) A decrease of 1'000 critiques caused by a removal of a big module that was implementing low-level functionality. The anomaly represents a change in quality, as the removal of a module simplifies maintenance of the whole project. On the other hand the change is mostly related to essential complexity and is not useful when included on the same level with common changes.
3) Another decrease of 2'500 critiques was caused by developers troubleshooting reports caused by a single rule. The anomaly is caused by intentional changes in the source code and rises a question: "when and why were these critics introduced and why were they not addressed earlier?"
4) 2'500 critiques were added because of multiple reasons: a) one rule was fixed, as previously it was including false-positive results; b) a new rule was added; c) a new version of a package management module was integrated. Multiple types of changes that could significantly affect the number of critiques made it very hard to reason about the anomaly.

Additionally, around 90% of all rules were changed to some degree. Not all the changes affected the functionality. They could be related to the changes of a description, group, severity, or could be caused by a refactoring of the critique detection algorithm.

Detecting the anomalies from the chart in Figure 1 may seem easy, but not all the critiques of an affected version are related to an anomaly, and we want to keep the "innocent" critiques for further analysis. Moreover one version can have multiple anomalies of different types that should be handled

[1] http://pharo.org/

separately. We acknowledge that some statistical methods may help to identify the anomalies, but at the moment we are not aware of the "anatomy" of anomalies, thus we want to obtain an overview with the help of visualization.

After analyzing the selected anomalies and other aspects of the data set we compiled the requirements for analyzing the impact of the quality analysis tools. Critique evolution consists of two types of changes: *gradual* and *extreme*. The former are the result of common code evolution that slightly impacts the critiques on each commit. We can analyze gradual changes by using graphical and statistical methods and draw conclusions based on the trend of changes. However, the latter consist of anomalies. It is complicated to provide an evolutionary summary for them as extreme changes are diverse and not frequent. The previous inspections revealed that the anomalies are caused by critiques that represent a single rule in one version, or are related to a single package in one version. We believe that a report about extreme changes should consist of summaries that describe each individual anomaly. Additionally we need a possibility to analyze the data on a time scale and correlate an anomaly with similar ones that have occurred in other versions.

III. Related Work

Our visualization approach is driven by the characteristics of our data set. In our first attempt we used a simple bar chart (as demonstrated on Figure 1). Thus, to understand the behavior of each package and each group of rules we had to create separate charts. We could detect versions, and packages revealing high changes in the number of rules. However, we could not identify the reason behind the anomaly, as in the chart both package changes and rule changes are projected onto a single dimension.

The ChronoTwigger [11] visualization supports the analysis of two co-changing properties. However, it is constrained to properties in the same dimension, and in our case we needed to analyze the evolution of critiques based on two evolving parameters: packages and rules.

The Evolution Matrix [12] technique, uses polymetric views to visualize the evolution of software packages over the time. We visualized the number of critiques and number of rules from where the critiques originated as the metrics on the rectangles' extents. We found that the Evolution matrix provided a good overview of the relation between packages and versions. Although this gave us a better understanding about the places with significantly higher number of critiques from diverse rules, we could not investigate the reason behind the anomalies.

To assess the benefit of visualizing package-rule relations, we created an incidence matrix. In it, one axis represented packages while another one represented rules. The only metric that we applied to matrix cells was the number of changed critiques represented as an amount of blue coloring. We produced several samples for data coming from different versions. They revealed patterns that helped us to identify abnormal changes of critiques. Figure 2 shows part of the visualization of *version 241*, where critiques have changed in many packages, but for only one rule.

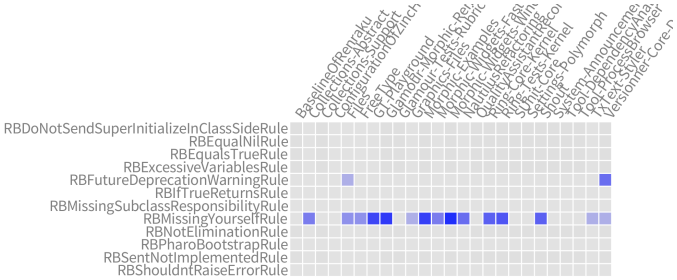


Fig. 2: An incidence matrix to visualize change of critiques.

2-dimensional matrices were already used to visually solve diverse time-related problems. The Small MultiPiles [13] approach clusters similar matrices from the history into piles and presents them as small multiples. Brandes and Nick use glyphs based on gestaltlines [14] to represent an evolution between relations in an incidence matrix [15]. Finally AniMatrix [16] uses animated matrix cells to convey the evolutionary information. While these approaches looked promising we found them difficult to use with our dataset because matrix cells were as small as 2.5mm or 10px in width and height when the matrix was fully expanded on a 27 inch display. Additionally our main concern was to detect anomalies in the project's evolution instead of sequential patterns. We believed that seeing the correlation between packages, rules and versions in one image will solve this problem. Thus, we explored a visualization that uses a 3-dimensional metaphor.

We reviewed Sv3D [17] which uses a 3D representation. In it, data is depicted by cylinders that are positioned using three numerical attributes of data. One extra attribute is mapped to the height of cylinders. Finally, cylinders are colored to encode a categorical attribute. Although we found it useful to provide an overview, it did not help us to identify anomalies since occlusion among cylinders hindered our ability to identify anomalies in the data.

Matrix Cubes [18] is a visualization technique based on a space-time cube metaphor of stacked adjacency matrices in chronological order. Since we have to analyze the relation between objects of different kind, we adopted this technique and expand it to use incidence matrices.

IV. Visualization Approach

Our visualization is developed in Pharo itself using a 3D version of Roassal [19] – an agile visualization engine. While we believe that our approach is applicable in many different contexts, we decided to script the exact visualization that we need instead of building a highly customizable application. This is why our main focus is on the explanation of the general approach and discussion of the details that may work differently for other cases.

Our data set can be indexed with triples of the form: *package name*, *rule name*; *version number*. We want to study a metric called the *critique delta*: an integer value that represents the

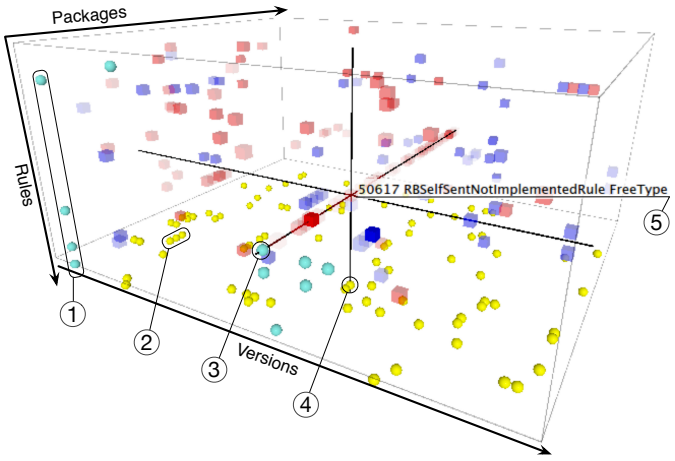


Fig. 3: Visualization Example.

number of critiques that have changed in a *package* based on a *rule* in a *version*. A critique delta of version v is calculated by subtracting the number of critiques in version $v-1$ from the number of critiques in version v.

As we wanted to build a visualization based on three independent values, we decided to use a 3-dimensional space and encode the critique delta values with the help of color intensity. An example of the visualization is presented on Figure 3.

Critique deltas are represented as cubes in a 3D matrix. Versions are natural numbers, and we sort them in an ascending order to represent data in a historical way. Also we sort packages and rules in an alphabetical order. First of all they are represented in this order in many tools that are used inside an IDE that makes this ordering a common way to comprehend them. Secondly, multiple packages form implicit groups by beginning with common prefixes. Alphabetic ordering keeps implicit package groups together and enhances comprehension as it is common for these groups to co-evolve at the same time.

Hovering over cubes displays a popup (⑤ in Figure 3) with information about the cube and a *crosshair* that allows a user to identify which entities are at the same level as the one being hovered over.

We believe our visualization technique is general enough to tackle problems of other domains. Therefore, we classify it using the five dimensions proposed by Maletic *et al.* [20]. The *task* tackled by our visualization is identification of anomalies and cleaning of data; the *audience* consists of software analysts who need to make sense of quality evolution; the *target* is a data set containing a set of critique rules for each package of a range of revisions; the *representation* used can be classified as a geometrically-transformed projection according to Keim's taxonomy [21]); the *medium* used to display the visualization is a high-resolution monitor with at least 2560 x 1440 pixels.

We designed our visualization according to the visualization mantra introduced by Shneidermann [22]. First, users obtain an *overview* to identify places of interest. Once they find one, they *zoom* in to have details, they can also *filter* surrounding data to maximize the focus on the objects of interest, and finally they can obtain *details-on-demand* of the critiques delta of a package within a version.

A. Coloring

We use color coding to determine if the critiques delta is positive or negative. Red represents an increased number of critiques, while blue means that the number of critiques has decreased in that version. Translucency of cubes is determined based on the absolute value of the critique delta: the cube with the biggest absolute critiques delta value will be opaque, while the one with no changes will be transparent. The other cubes will have their translucency proportional to the maximum of absolute values of the critiques deltas. This ensures that the larger changes will have more visual impact in comparison with the smaller ones.

We considered two approaches for calculating translucency: one of them calculates a separate maximum for each version, while another one uses a single maximum based on all the critique deltas. The former approach ensures that in case there is one significantly larger change in the whole history, it will not make all the other cubes barely visible. However we also find it important to base the alpha value on the whole history to get a better idea if at some time there were bigger changes. We determined that an alpha that is 2/3 based on local maximum and 1/3 based on global maximum works well in our case. We cannot generalize this decision, but rather suggest to calculate alpha based on both local and global maximums.

B. Changes. 2D Meta Information

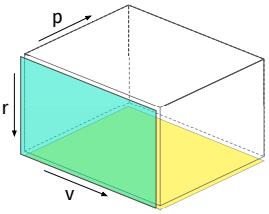


Fig. 4: Meta planes illustration

As can be seen on Figure 3, our visualization also contains cyan and yellow spheres. They are situated in 2-dimensional planes and contain additional information about packages and rules for each version. Cyan spheres reside in a rule-version plane and each of them represents changes made to the rule in the version. Yellow spheres are related to the changes in packages and reside in package-version plane. To better explain the location of the spheres, we provide an illustration in Figure 4. We use a different shape: spheres, as they represent completely different data from cubes. We also color them with distinct colors that are different from the critiques delta color codes.

The cyan plane is located on the side of the matrix. It is in front of the matrix if you are looking from the position where the versions increase from left to right. We find this to be a common position for inspecting the matrix, as in western culture people expect time to travel to the right. We place the spheres in front of the visualization as changes in rules are not frequent and most of the time we want to correlate exceptional changes of critiques with the changes of rules.

Packages have significantly more changes in comparison with rules, in fact each version is an update of some packages. We found that the change metadata can obstruct the rest of visualization. This is why the yellow plane is located at the bottom of the matrix, as it is common to look at 3D visualizations from above the horizon level.

The crosshair extends slightly beyond the change planes making it easy to see if a sphere is on the same line with a square. Hovering over spheres also displays the crosshair in the same way it works with cubes. This allows a user to easily see what cubes are related to a change, what other changes happened in the same version, or in which versions the same rule (package) was changed.

The change spheres can be used in two ways. One of them is to easily see if there was a change in the rules or packages for some set of squares. For example in Figure 3 a cursor is hovered over a cube that is on one line with the other ones and the crosshair is penetrating a cyan sphere ③ revealing that the rule of the hovered cube has changed in this version. Secondly one can start by looking at the patterns in changes ① ② and inspect the impact that they made based on the visualization.

C. Visual Features

The visualization provides many different pieces of information, as we have a cube position based on 3 coordinates, color, translucency and 2 extra planes that have a sphere position based on 2 coordinates. It may seem that this amount of data pollutes the visualization and makes it hard to understand. For this reason we identify 2 sequential questions that a user of our visualization wants to answer.

1) What are the irregularities in the system's evolution?
2) Why did this irregularity occur?

To answer the first question a user can use the camera movement and identify clusters of cubes. We based our approach on the proximity principle: a pre-attentive feature that allows us to cluster closely-situated visual elements in a fraction of a second [23]. The principle works in 2D space so that the 3D visualization is eventually projected on a plane. We took into account many aspects, such as the sparse nature of the critic deltas and translucency of smaller deltas, to avoid occlusion, because it can cause false clusters to appear on a 2D projection. The cube clusters form lines, as seen in Figure 5. At this phase spheres do not obstruct the visualization; the color of cubes is not as important as whether the cubes are there or not, and whether the proximity is preserved during the camera movement.

After a user has identified the pattern of cubes and locked on it, the second question should be answered. In this case the rest of the visualization comes into play and helps a user to understand what is the version, which rules have changed in this version, were the critiques added or removed, *etc.*

D. Interaction

The visualization supports orbiting of the camera around the 3D matrix with a mouse. Also a keyboard can be used to move horizontally or vertically the point at which the camera is looking (the same one used as the center of orbital movement). By hovering with a mouse over visual elements user can see a popup (⑤ in Figure 3) with an information about the version, rule and package of the element. Also a crosshair appears on the hovered element and spans the whole matrix including planes with spheres. This allows a user to easily identify which elements are on the same line. For example on Figure 3 the crosshair's line is passing thorough many red cubes and a cyan sphere. This demonstrates that all the critique changes are on the same line, and are reported by a rule that has changed in this version ③ . Another line of the crosshair is penetrating a yellow sphere, which means that the package related to the hovered cube has changed in this version ④ .

While we rely on the natural clustering, we also provide slicing functionality that allows a user to hide the unneeded parts of the visualization to avoid being distracted by them. These options are accessible from the context menu of any cube. One kind of slicing removes all cubes that are more than two steps away from the selected one. This can be done based on all three dimensions: by versions, rules or packages. As the result only a slice with a thickness of 5 cubes is visible as shown on Figure 8. The other kind of slicing simply generates a 2-dimensional incidence matrix visualization (Figure 9). This slicing approach eliminates the distortion caused by perspective, but also lacks information about the neighbor slices.

We encode a large amount of data into the visualization, but some information like a textual change log summarizing the patch cannot be conveyed by colors or layouts. For this purpose we provide a dialog window with a textual description of the patch release notes together with links to the discussions on an issue tracker.

Finally we envision our visualization as a tool for identifying and removing anomalies. For this purpose we provide an option to log an anomaly which can be:

- rule anomaly: all critiques with a certain rule and version;
- package anomaly: all critiques with a certain package and version;
- version anomaly: all critiques with a certain fixed.

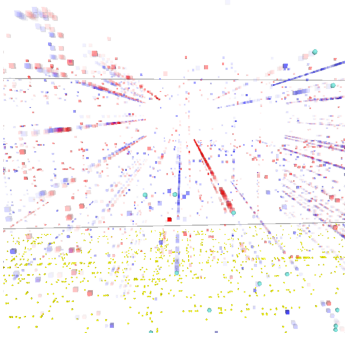
After being logged the cubes related to the anomaly will be removed from the visualization to simplify the detection of other anomalies. This action can be accessed either from a cube's context menu, of from the dialog window with a patch summary.

V. Case Study

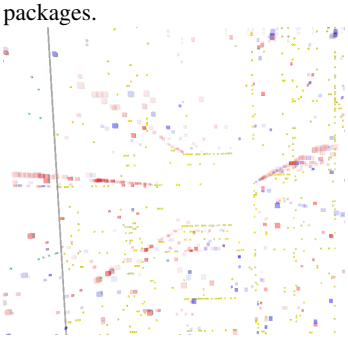
Figure 7 displays our visualization applied on the full Pharo data set described in Section II. In this section we will do a step-by-step walkthrough for decomposing the evolution, and provide the obtained results.

A. Decomposing the Data Set

To find anomalies in the system's evolution we orbited around the visualization and looked for patterns that stood out. All the patterns that we identified by this approach had a line



(a) Beams (side view): critiques of a same rule changed for many packages.



(b) Pillars (top view): critiques of many rules changed for a same package.

Fig. 5: Common critique anomalies.

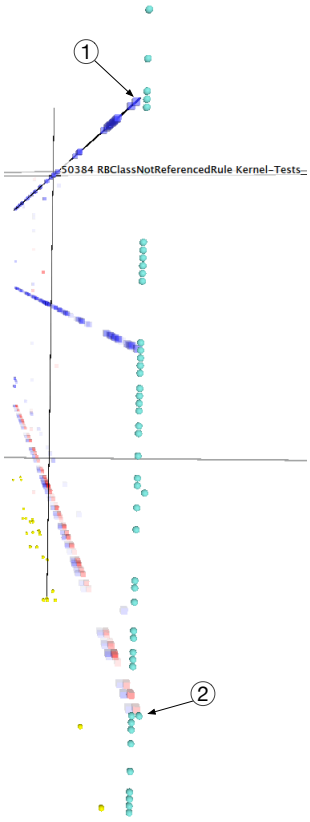


Fig. 6: Core rule refactoring.

made out of cubes as their base component. The cubes that make these lines represent critiques from the same version. There are two types of this lines: *beams* – the critiques are related to a single rule and form a horizontal line (Figure 5a) and *pillars* – the vertical counterpart where the critiques are about a single package (Figure 5b).

A few patterns especially attracted our attention. The critiques in this case form a *wall* of cubes by spanning both multiple rules and packages (Figure 8). All anomalies of this kind were related to critical issues in the system that were immediately fixed. This is why all the walls came in pairs of opposite colors separated by at most one version. For these anomalies slicing the visualization to present only a subset of cubes in a range of 5 versions was useful to remove all the noise around and investigate the pair of walls alone. The two dimensional representation shown on Figure 9 helped us to isolate one plane even more and remove the perspective distortion. We viewed the patch comments for each version and analyzed changes made. After this we saved the version numbers together with comments about the reason of each anomaly. In the end we hid the walls to remove unneeded obstructions.

The second kind of special pattern that drew our attention appeared in cyan spheres and was related to rule changes. There were two versions where changes occurred in almost every rule, which is most likely a sign of refactoring, as many similar components of a working system have changed simultaneously. One of them did not have any beams, and the other one is shown on Figure 6. The visualization contains 4 beams. The crosshair on one of them does not penetrate any sphere from the cyan pillar (1) . This allows us to easily see that the beam is not aligned with the pillar which means that they are from different versions. The only red beam (2) is also not in the version with rule refactoring, but it follows a blue beam and also has a cyan sphere on its end. The next hypothesis can be formed by simply looking at the visualization: "There was a refactoring globally performed on all the rules, because of which two rules were broken and one of them was fixed in the following patch." By looking at the patch summary we confirmed that our hypothesis was correct except for the detail that one of the rules was not broken but rather fixed during the refactoring session. This also explains why it didn't receive any more attention in comparison with other rule that was immediately fixed.

After dealing with walls and rule refactoring we started to process other beams, as they were more prominent in comparison with pillars. The standard workflow went as follows:

1) **visually locate**: we visually explored our visualization and focused on the lines that can be seen at Figure 5a. We used camera movement to change the angle of view and viewpoint to ensure that the cubes are not forming a line only in one projection.
2) **analyze relations**: we used a crosshair as demonstrated in Figure 6 to better understand how are the other elements situated relatively to the beam. We also used slicing to focus only on the critiques of a few versions (Figure 8), or on the critiques of a single rule by using 2-dimensional slice similar to the one in Figure 9. The slicing functionality was used to identify if there were other beams in the same version or in the whole history but related to the same rule.
3) **understand the cause**: at this point we mainly relied on the patch summaries, issue tracker messages and source code diffs to understand the reason of changes and the cause of the anomaly.
4) **log and hide**: we annotated the anomaly with the explanation about the changes that caused it. Finally we hid the anomaly to avoid distractions during further explorations.

After dealing with beams we moved to pillars. We quickly noticed that most of them are related to the changes that were introduced in the package that they represented. It is arguable whether there is a benefit of logging and removing such kind of anomalies from the data set. They are related to the one of the main questions of software quality analysis: "how does this change impact the software quality?" However we decided to log these anomalies anyway, as we wanted to investigate if there are other causes and also by removing or hiding them we could reveal other less prominent anomalies. The strategy for processing pillars was the same one as for processing beams. We naturally finished our analysis when we were not able to

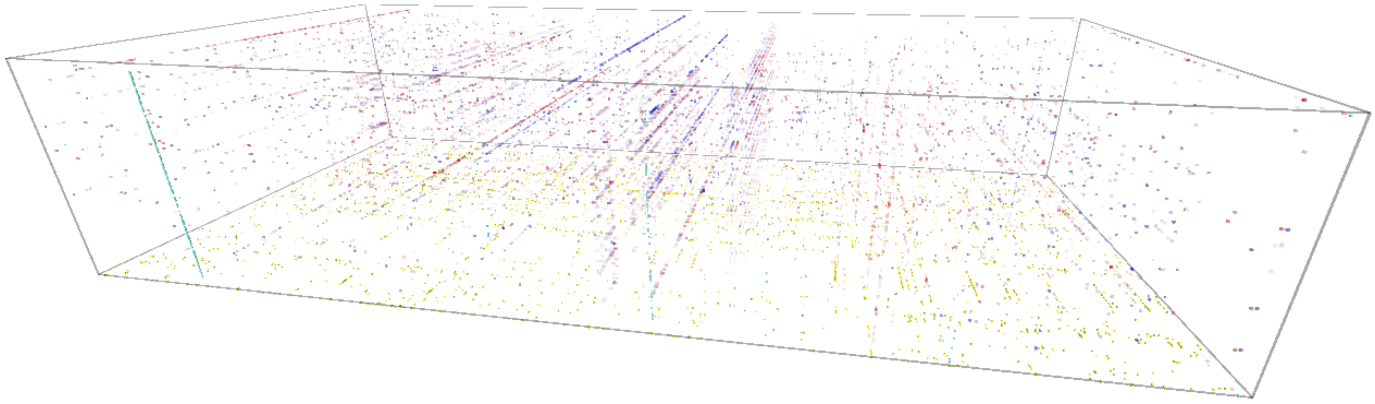


Fig. 7: A complete visualization of Pharo critiques over the history of 680 incremental patches.

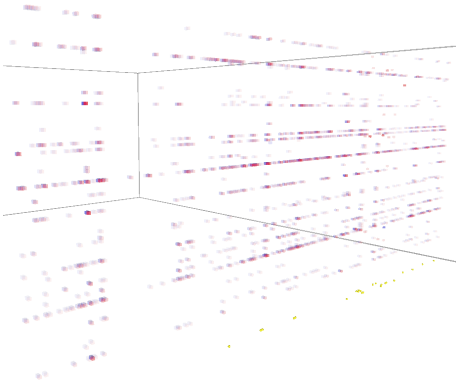


Fig. 8: "Wall": Critiques of a significant amount of rules changed for many packages.

Fig. 9: 2D representation of one version.

Type	Subtype	Number
Complete versions		6
Rules (32)	added or removed	8
	fixed or broken	17
	other (non-related to rule changes)	7
Packages (45)	added or removed	42
	modified	3

TABLE I: Number of recorded anomalies by type.

detect anomalies any more.

B. Obtained Results

The quantitative results of the decomposition that we performed are presented in Table I. The minority of anomalies affected both many rules and packages of a version. This is natural, as such anomalies are related to severe issues in the system. In our case there were 6 such cases that formed 3 pairs, as each defective patch was instantly fixed or reverted. Only one such pair was related to changes in the quality validation system. It was very hard to identify the cause of all such anomalies, and this involved reading patch summaries, bug tracker issues and even code that was changed.

For the most of the logged anomalies critiques of many rules affected a single package. Out of the total of 45 anomalies, 42 were related to the packages being added or removed. This could in fact be easily detected automatically. The remaining 3 anomalies were caused also by a package-related changes, where a significant amount of code was changed in one patch.

The third type of anomaly – critiques about a single package that originated from many rules, had 32 occurrences. 8 of them were related to an addition or a removal of the rule. This subtype of anomaly could be detected automatically. 17 anomalies were related to the rules being fixed or broken. And the smallest subtype with only 7 cases is related to the anomalies that are not related to the rule changes. Some of them were results of a planned eradication of the critiques of a certain rule. The others were related to the specific changes of the source code that had an impact only on a single rule.

Despite eliminating all the visual anomalies, we missed a few cases where a single cube had a large delta of critiques. For example the average delta is around 10 critiques and two cubes had a delta of more than 2000 critiques. These cases are very rare and very hard to detect, as in the 3D matrix they are represented by a lonely completely opaque cube that is not very different from its surroundings. On the other hand these anomalies can be easily detected by sorting all deltas by their absolute value and inspecting the largest ones.

The most important findings were concluded from the anomalies related to the critiques of a whole version, critiques related to rules that were not caused by the rule changes and critiques of a single rule that affected only a single package in a version. These findings show weak points in the system, and the integration approach. Rule-related anomalies caused by the rule changes allowed us to understand how requirements to the code quality were changing over time.

C. Anatomy of the Anomalies

Most of the **package-related** anomalies were caused by addition or removal of the packages themselves. These changes were caused by replacing old submodules by new alternatives, integration of new features or removal of the unused ones. The smaller amount of anomalies caused by dramatic package changes happened in the packages that belong to external submodules. They are versioned separately from the main project and the integrated versions contain more changes.

Rule-related anomalies have a more diverse nature. Poor value of the critiques reported by rules was the main reason for their removal. The rules that were added captured the design decisions of different parts of the project. Some of them were related to a method invocation order, others provided suggestions about the usage of core API migrations, or about the methods that have to be defined under certain circumstances. Rule fixes either were focused on capturing the violations that were missed or excluding false positives from the results. Also few rules had their scope reduced to avoid the overlap of critiques. The regressions in rule functionality happened because of two reasons: either a mistake was made during a refactoring or the precision of a rule was sacrificed in favor of performance. After analyzing the data set and rule anomalies in particular we can suggest stakeholders a test that can warn about these kinds of changes in rules prior to integration.

Some rule-related anomalies were caused by changes in the code. For example one of them reported many invocations of undefined methods. This was caused by the changes to the API of an icon factory. Another case involved deprecation of a widely used API, which caused many deprecation warning critiques. A third case involved the addition of support classes that reported a high number of "unused class" critiques. The last two cases were negated by counter-anomalies where issues introduced previously were fixed. We suggest the stakeholders to review the quality validation in their integration process, because according to our findings the critiques that can be easily solved with a simple automatic refactoring were ignored and integrated.

Wall anomalies are the most interesting type. We identified three pairs of them and only one was related to the changes is the quality validation framework. It occurred when the server-side validation system was broken, and the changes made were intended to fix the issue. As the result integrated changes broke the validation system completely and were instantly reverted. Other two anomalies were caused by integration of a changes with invalid source code. Beside breaking the quality validation the changes also causes issue with source code recompilation and were fixed in the following versions. We encourage stakeholders to investigate the integration process, as two changes that broke the validation were nevertheless integrated. We also advise to add a test of source code integrity to detect the similar issues more easily.

Finally, our use case contained two **single-cube anomalies** that were related to a single issue. The rule violated by this anomaly is checking whether a class contains methods identical to the ones defines in traits [24] that the class is using. First anomaly was caused by a package rename refactoring during which all trait methods were copied into the classes of that package. The second anomaly appeared 170 versions later when the duplicated methods were removed. The issue was identified because developers noticed the related critiques. However we advise the stakeholders to investigate why this changes were integrated in the first place, and solve the duplication bug of rename package refactoring.

VI. Discussion

In this section we reflect on our use case experience and discuss both positive and negative aspects of the visualization.

The visualization represents anomalies as natural clusters of data that are easily detectable by visual exploration. The orbital camera movement was essential to identify whether the detected pattern is not an accidental alignment of the elements in the current projection. For the same reason we suggest to use the same size for all the cubes, as different sizes will complicate the perception of dimensional positioning. The sparse nature of the data is also very important for the visualization. Because the changes to the critiques should not be frequent and large, most cubes are highly translucent or completely transparent and do not obstruct the view of the ones positioned behind them.

The movement interactions were not very user-friendly and could benefit from improvements. For example visual elements could be selectable, after which they will serve as a center of the orbital movement. Also the effort spent on getting closer to a desired element to inspect it can be enhanced by using semantic zooming [25]. As the visualization presents data in an immersive 3-dimensional environment and mainly relies on pre-attentive processing possibilities of a human brain we believe that it can be interesting for researchers that explore visualizations in virtual reality [26].

Slicing was another important feature. It allowed us to isolate an interesting piece of information from the rest of the visualization that was obstructing the view. We found out that 3-dimensional slicing (Figure 8) was the most useful when applied to the version axis. This allowed us to see the changes in the adjacent versions and often we were able to detect cases where some changes were rolled back, or continued on other entities. The same kind of slicing was useful for packages axis, however this is related to the nature of our data set. As mentioned previously the packages form implicit groups that have same base name and different suffixes. These groups usually change together, so having a 5 block deep slice allowed us to capture up to 5 co-changing packages. This was not

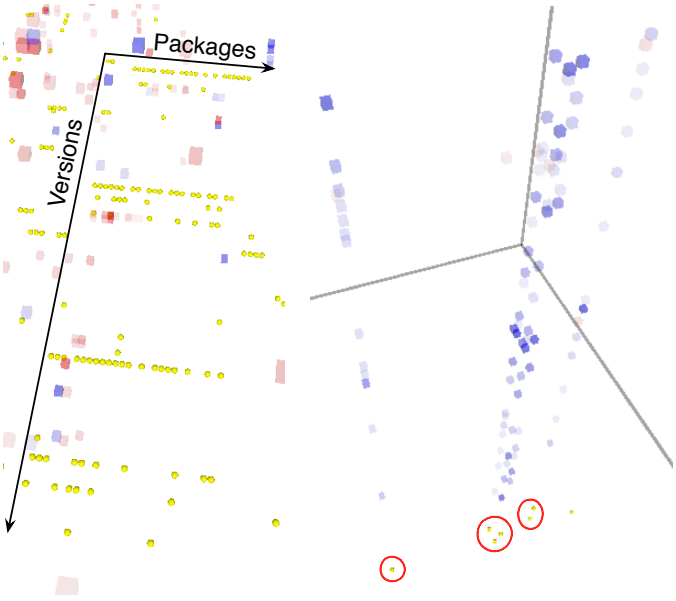
always practical as sometimes more than 5 packages formed a group. This suggests that we need to have a support of a variable slice depth. 3-dimensional slicing was not applicable to rules, as every rule evolves independently of the others. The main goal of slicing the rule axis is to see if there were similar anomalies for the rules throughout the whole history. If the slice contains more than 1 rule, the anomalies from other rules will also appear in the slice and make the analysis more complex. Thus 2-dimensional slicing (Figure 9) worked the best for the rule axis. Similarly 2-dimensional slicing was useful in every case where a single relation between two properties (rule, package or version) had to be examined. Also the possibility of creating a multiple slices can be useful when inspecting similar changes separated by a large period of time.

While obtaining the information about an inspected patch, the main summary and issue tracker discussions were not always enough. Sometimes we had to analyze which classes and methods were changes in the particular patch. Additionally it may be useful to have a support of calculating difference for non-adjacent versions, this can help in detecting rollbacks. We detected a few anomalies that were related to each other in our use case. This requires not only a possibility of multiple slices or selection, but also some features to record this relation between anomalies.

A unique feature of our visualization was the metadata representation by spheres. We found the information about the rule changes extremely useful. It allowed us to easily identify if there were changes made to the rule related to a visual element, and see if it was also changed in the nearby version. Similarly we could see if the other rules changed in the same version. In some cases changes to the rules were driving our exploration because we were able to detect patterns of cyan spheres.

On the other hand information about the package changes was not very useful. Because of the nature of our data set changes to the package are frequent, and yellow spheres obstruct the view if placed on top. We placed them at the bottom and then it was hard to see how they are related to the data. There were some use cases where yellow spheres clearly revealed groups of packages that changed together (Figure 10a). Also during the pillar inspection yellow spheres at the bottom of pillars were clearly identifying that the critiques are related to a historical group of package changes (Figure 10b).

The difference between the usability of cyan and yellow spheres can be explained by the nature of our problem. The yellow spheres represent the changes of packages. These changes are the the building blocks of software evolution. They are frequent and we are considering their existence to be natural. Rules are also evolving, but at a much slower pace and they do not clutter the view. Our main focus is to identify the changes in rules, because they are not as common to us as the changes in packages. These relations can be different in another use case that will focus on something other than changes in rules and packages. This is why we encourage the users of our approach to experiment with positioning the meta



(a) Co-changing packages. (b) Spheres at the bottom of pillars.

Fig. 10: Package change metadata.

information planes on the different sides of the visualization.

We already mentioned in Section V-B that many of the anomalies were related to the addition or removal for rules and packages. Before decomposing the visualization into the anomalies we were not expecting such high percentage of them to be caused by addition or removal. Now we can recommend the users of our approach to automatically detect and remove from visualization the anomalies based on this criteria. Also we suspect that some of the other anomalies can be detected by a statistical approach, or at least be shortlisted statistically. We have not investigated this idea, but without building the visualization we did not know how our data looks like and what the statistical approaches should look for.

We presented a use case where quality critiques were influenced by the changes of both quality rules and source code. We believe that this visualization can be applicable to many problems where one value depends on the other two co-evolving values. The immediate related problems that can be tackled by the approach are concerning failing tests and changes in the performance.

Many visualizations suffer from scalability issues, as the visual elements become too small and the encoded metrics can not be read. In contrast, our approach relies on the significant amount of data that allows a user to detect anomalies that span the whole visualization. We expect that at some point the number of visual elements will decrease the performance of visualization, but this can be mitigated with a sliding time window approach [27]. Also at some point the lines that form anomalies may become too thin to identify them. In this case we suggest to group the entities into blocks that unite the entities with some feature but evolve independently of each other. For example in our case packages can be grouped by their base name, while rules can be grouped by their category.

VII. Conclusion

We have presented an approach for visualizing the evolution of a value that depends on two co-evolution properties. The main goal of the approach is to detect, identify and log the anomalies that prevent the evolutionary analysis of dependent values. The visualization is constructed in 3-dimensional space and relies on the sparse nature of analyzed data. It enables quick detection of the anomalies with the help of pre-attentive cluster recognition and provides multiple visual features that enable a user to obtain more detailed information. While many visualizations try to provide meaningful information in each visual element, our approach can be referred to as "anti-matrix", because the data provided by the matrix serves secondary purposes while we focus on detection of "structures" in the 3D space that indicate anomalies. This makes our approach resistant to large dataset sizes *e.g.,* we don't analyze individual cells of a 200x100 matrix, but detect walls, pillars and beams that can consist of different number of elements.

We evaluated our approach by analyzing quality evolution of a real project measuring 520 thousand lines of code. The quality was affected by both changes in the source code and changes in the rules that define quality concerns. We were able to successfully identify most of the anomalies, while the remaining ones can be easily detected by using statistical approaches. We analyzed 85 anomalies and categorized them into different types. Some of the types turned out to be easily detectable automatically, the summary about the others can help to deal with the anomalies in similar problems.

We described all the possible scenarios that can be followed with our visualization, but one can also benefit by using it for a single task such as: 1) obtaining a general overview of the system to understand the status of anomalies; 2) extracting anomalies caused by only one of the co-evolving parameters; 3) completely cleaning the system of anomalies. Also our approach can be combined with others to perform a more advanced analysis.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018).

References

[1] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, Jan. 2012.

[2] P. Louridas, "Static code analysis," *Software, IEEE*, vol. 23, no. 4, pp. 58 –61, 2006.

[3] N. Ayewah, D. Hovemeyer, J. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *Software, IEEE*, vol. 25, no. 5, pp. 22–29, Sep. 2008.

[4] G. A. Campbell and P. P. Papapetrou, *SonarQube in Action*, 1st. Greenwich, CT, USA: Manning Publications Co., 2013.

[5] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" In *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 672–681.

[6] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, Florence, Italy: IEEE Press, 2015, pp. 598–608.

[7] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker, *Pharo by Example*. Square Bracket Associates, 2009.

[8] D. Roberts, J. Brant, R. E. Johnson, and B. Opdyke, "An automated refactoring tool," in *Proceedings of ICAST '96, Chicago, IL*, Apr. 1996.

[9] Y. Tymchuk, "What if clippy would criticize your code?" In *BENEVOL'15: Proceedings of the 14th edition of the Belgian-Netherlands software evoLution seminar*, Dec. 2015.

[10] F. P. Brooks Jr., *The Mythical Man-Month*, 2nd. Reading, Mass.: Addison Wesley Longman, 1995.

[11] B. Ens, D. Rea, R. Shpaner, H. Hemmati, J. E. Young, and P. Irani, "Chronotwigger: A visual analytics tool for understanding source and test co-evolution," in *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, 2014, pp. 117–126.

[12] M. Lanza and S. Ducasse, "Understanding software evolution using a combination of software visualization and software metrics," in *Proceedings of Langages et Modèles à Objets (LMO'02)*, Paris: Lavoisier, 2002, pp. 135–149.

[13] B. Bach, N. Henry-Riche, T. Dwyer, T. Madhyastha, J.-D. Fekete, and T. Grabowski, "Small multipiles: Piling time to explore temporal patterns in dynamic networks," *Computer Graphics Forum*, vol. 34, no. 3, pp. 31–40, 2015.

[14] U. Brandes, B. Nick, B. Rockstroh, and A. Steffen, "Gestaltlines," *Computer Graphics Forum*, vol. 32, no. 3pt2, pp. 171–180, 2013.

[15] U. Brandes and B. Nick, "Asymmetric relations in longitudinal social networks," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2283–2290, Dec. 2011.

[16] S. Rufiange and G. Melanon, "Animatrix: A matrix-based visualization of software evolution," in *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, Sep. 2014, pp. 137–146.

[17] A. Marcus, L. Feng, and J. I. Maletic, "3D representations for software visualization," in *Proceedings of the ACM Symposium on Software Visualization*, IEEE, 2003, 27–ff.

[18] B. Bach, E. Pietriga, and J.-D. Fekete, "Visualizing dynamic networks with matrix cubes," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '14, Toronto, Ontario, Canada: ACM, 2014, pp. 877–886.

[19] V. P. Araya, A. Bergel, D. Cassou, S. Ducasse, and J. Laval, "Agile visualization with Roassal," in *Deep Into Pharo*, Square Bracket Associates, Sep. 2013, pp. 209–239.

[20] J. I. Maletic, A. Marcus, and M. Collard, "A task oriented view of software visualization," in *Proceedings of the 1st Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2002)*, IEEE, Jun. 2002, pp. 32–40.

[21] D. A. Keim and H.-P. Kriegel, "Visualization techniques for mining large databases: A comparison," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 8, no. 6, pp. 923–938, 1996.

[22] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *IEEE Visual Languages*, College Park, Maryland 20742, U.S.A., 1996, pp. 336–343.

[23] C. Ware, *Information Visualisation*. Sansome Street, San Fransico: Elsevier, 2004.

[24] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, "Traits: Composable units of behavior," Institut für Informatik, Universität Bern, Switzerland, Technical Report IAM-02-005, Nov. 2002, Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.

[25] A. Woodruff, J. Landay, and M. Stonebraker, "Goal-directed zoom," in *CHI 98 conference summary on Human factors in computing systems*, ser. CHI '98, Los Angeles, California, United States: ACM, 1998, pp. 305–306.

[26] F. Fittkau, A. Krause, and W. Hasselbring, "Exploring software cities in virtual reality," in *Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on*, 2015, pp. 130–134.

[27] T. Zimmermann and P. Weißgerber, "Preprocessing CVS data for fine-grained analysis," in *Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004)*, Los Alamitos CA: IEEE Computer Society Press, 2004, pp. 2–6.