Evo-Clocks: Software Evolution at a Glance

Carol V. Alexandru*, Sebastian Proksch*, Pooyan Behnamghader†, Harald C. Gall*
* Software Evolution and Architecture Lab
University of Zurich, Switzerland
{alexandru,proksch,gall}@ifi.uzh.ch
† Center for Systems and Software Engineering
University of Southern California, USA
pbehnamg@usc.edu

***Abstract*—Understanding the evolution of a project is crucial in reverse-engineering, auditing and otherwise understanding existing software. Visualizing how software evolves can be challenging, as it typically abstracts a multi-dimensional graph structure where individual components undergo frequent but localized changes. Existing approaches typically consider either only a small number of revisions or they focus on one particular aspect, such as the evolution of code metrics or architecture. Approaches using a static view with a time axis (such as line charts) are limited in their expressiveness regarding structure, and approaches visualizing structure quickly become cluttered with an increasing number of revisions and components. We propose a novel trade-off between displaying global structure over a large time period with reduced accuracy and visualizing fine-grained changes of individual components with absolute accuracy. We demonstrate how our approach displays changes by blending redundant visual features (such as scales or repeating data points) where they are not expressive. We show how using this approach to explore software evolution can reveal ephemeral information when familiarizing oneself with a new project. We provide a working implementation as an extension to our open-source framework for fine-grained evolution analysis, 𝕃ISA.**

I. Introduction

Managing, developing, and maintaining large software projects is fundamentally challenging due to their size and complexity and the distribution of knowledge across many team members and stakeholders. Developers in large projects often only work on small parts of the software and may be unable to stay aware of the entire context surrounding the software artifact they work on. Furthermore, it can be particularly difficult for new members joining the project, outside contractors, or auditors to gain a comprehensive overview of the structure and history of the project in a short time, if relying solely on browsing source code (e.g., in an IDE), reading commit logs, or a combination of the two whilst browsing a version control system.

Numerous existing software visualization techniques address this challenge one way or another, attempting to visualize how both individual components and the structure as a whole evolve over time, typically using one of the two following methods or using a combination of both. The first option is to visualize the evolution of software metrics by using a global time axis, for example, in using line charts or more elaborate visualizations like the Evolution Matrix by Lanza [1]. In these cases, multiple (or even all) data points, stretching the evolution of one or more metrics, are visible at once. The other option relates to changes in structure and can be more challenging because software artifacts appear, disappear, move, and change over time. In this case, the time component is usually implemented on top of a structural visualization, such that changes can be played back as an animation, one time unit at a time, like in interactive variants of "code cities" [2], [3]. In this case, the observer can only inspect one revision at a time, but can move forward and backward in time to investigate the evolution.

While both these approaches can be useful in certain instances, we believe that for visualizing the evolution of software components, both techniques have their respective drawbacks. On one hand, over the lifetime of most individual software components, changes are typically small, so showing them without context can result in low information density of the resulting visualization. On the other hand, the sheer number of commits created in modern software development makes it infeasible to visualize the structure of a project for each revision in its entire history, as such a view would overwhelm the observer, while showing only a single snapshot at a time can make it hard to recognize higher-level tendencies.

We propose a novel method for simultaneously visualizing software structure and the evolution of its individual components: Evo-Clocks. It uses localized, evolutionary representations of individual artifacts embedded within a multi-revision representation of structure that has been merged into a single view. Our approach provides an accurate representation of structure at a single point in time or, alternatively, a close approximation of structure for an extended period, while also displaying the evolution of individual metrics in a single view. Selective information hiding, facilitated by simple yet effective selection and filtering features, and means to further explore hidden details, enable the observer to learn more about specific time ranges and metrics relevant to an exploratory task, like during an audit, when planning developer allocation, or onboarding of new team members.

Our contributions can be summarized as follows:

1) We propose a novel way of visualizing the gradual evolution of localized artifacts in the global structure of graph-based data, such as versioned source code.
2) We show that our approach can answer ephemeral, project-specific questions and provide valuable insights during the explorative phase of project familiarization.
3) We discuss numerous additional improvements that we think feasible with regard to the proposed visualization.
4) We publish the implementation of our entire toolchain as open-source software.[1]

[1]https://bitbucket.org/sealuzh/lisa/ – A separate replication package for the visualization study will be published and linked for the camera ready version.

II. Background and Research Goal

Underlying our method for visualizing graph evolution is an idea that originally relates primarily to the efficient computation of metrics and other analyses over the entire history of large software projects, although the technique applies to any kind of versioned graph, as we discuss in section V-D. To this end, we present a graph compression algorithm and numerous techniques for reducing redundancies when analyzing multiple revisions of the same project in previous work [4]. A central concept in that work is the idea of a “revision range”.

Individual commits to a software repository typically contain only a few localized changes. This is especially true in modern version control systems like Git, where large projects can receive dozens or even hundreds of commits per day. Nonetheless, most software-evolution research is conducted by applying measuring tools intended for analyzing a single (typically the current) state of a given project. Consequently, each revision to be measured is analyzed individually in a laborious, resource-intensive, and slow process. Research shows that more than 95% of data is redundantly analyzed when discounting the multi-revision nature of software and that, all other factors being equal, analyzing revisions individually can be over 50 times slower [4]. Short of expensively parallelizing the workload, a fairly simple improvement is to analyze revisions incrementally, recomputing values only in artifacts where changes occur. But given that changes are usually small and localized, for example, at the method level, this still involves significant redundant computation.

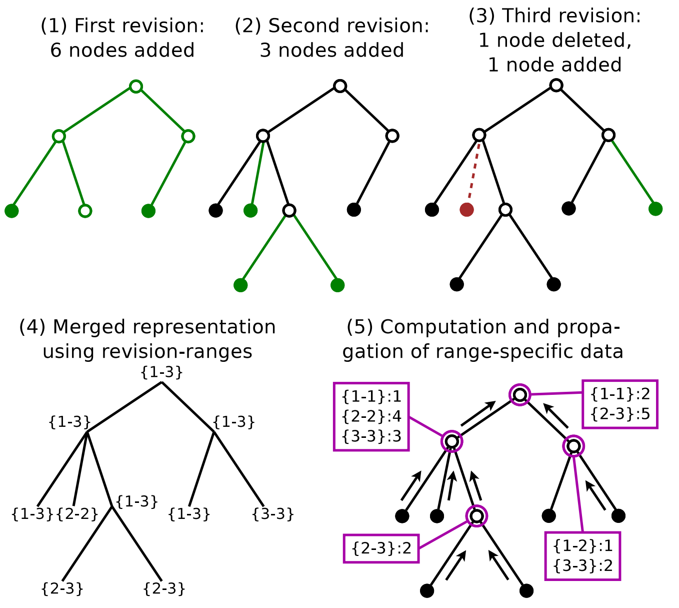


Fig. 1. Illustrative description of 𝕃ISA’s internal graph representation [4]. ○ represent (inner) class nodes, and ● represent method nodes. (1), (2) and (3) show how three commits modify the program structure. (4) shows the merged, annotated representation. (5) shows how a method count is computed and stored. The top-level class, for example, counts two methods in the first revision, and then five in the second and third revisions, for which this value is stored only once. Revision ranges are created, split and merged to accommodate all data with complete accuracy, while avoiding redundancies where there is no change over time.

We developed a representation of change (and lack thereof), in which program components are not represented as individual data points for every individual revision. Instead, components and their metrics are represented in *ranges* that comprise several (unchanged) subsequent revisions. As a result, analyses (for example, counting the number of methods in a class, shown in fig. 1) do not have to analyze each revision individually, but can be executed only once for a set of identical input values stretching across several revisions.

In practice, revision ranges can stretch over thousands of commits, in which a program component does not experience any change to itself or to the metrics computed for it. We have shown that this technique reduces the number of individual states required to be dealt with in individual program nodes by roughly 97% on average over $4,000$ projects we analyzed using our approach, compared to storing state for each node and every revision. Larger and older projects especially benefit from this redundancy-removal technique.

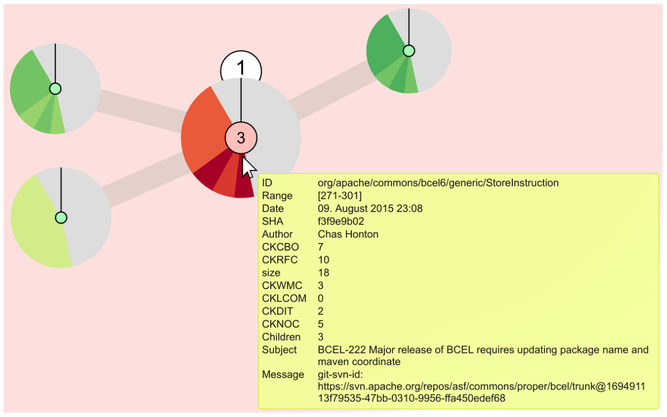
Even though we developed the idea of revision ranges with the primary goal of accelerating software evolution research, we recognize that it may also apply to visualization. Because changes are usually small and localized, a global, multi-frame view of a project does not change much over time. This is why visualizations like Gource [5] remain possible to follow and understand in the first place, as otherwise there would be too much happening on every frame. Thus, we hypothesize that applying the idea of revision ranges for visualizing the evolution of individual software artifacts (such as classes and methods) might have some merit. We ask the question: *How can the evolution of both structure and individual components be visualized without using a multi-frame visualization?*

Since this is a methodological question, we chose not to formulate design goals for the visualization itself, i.e., we do not determine in advance what kind of questions we want the visualization to answer. Instead, we implement the approach in order to discover its strengths and weaknesses, as well as which kinds of questions it is best able to answer. We discuss the results of this investigation in section IV.

III. Approach

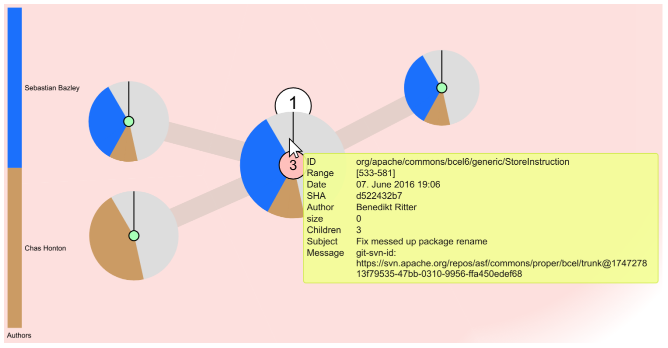
We wish to visualize how individual components evolve as part of a larger structure, independently of the application domain. In our specific use case – the evolution of software metrics and structure – we build a merged representation of multiple revisions as described in [**?**], resulting in a directed, cyclic graph where individual nodes represent program fragments and have arbitrary numerical and categorical attributes (e.g., size and authorship, respectively) which can change over time. Edges in the graph only have a categorical type attribute. We chose to use a node-link diagram for visualizing the subject matter as a whole, while using small, circular visualizations describing the history of individual nodes.

In this section, we describe how representing metrics across revision ranges can be achieved through what we call “*clocks*”. We also show how the node-link diagram is a merged representation of how the structure looked at different points in time. In



ID	org/apache/commons/bcel6/generic/StoreInstruction
Range	[271-301]
Date	09. August 2015 23:08
SHA	f3f9e9b02
Author	Chas Honton
CKCBO	7
CKRFC	10
size	18
CKWMC	3
CKLCOM	0
CKDIT	2
CKNOC	5
Children	3
Subject	BCEL-222 Major release of BCEL requires updating package name and maven coordinate
Message	git-svn-id: https://svn.apache.org/repos/asf/commons/proper/bcel/trunk@1694911 13f79535-47bb-0310-9956-ffa450edef68

Fig. 2. Four *clocks* representing the size history of a class (a 1st-level node) and its three methods (2nd-level nodes). The center of the class node indicates the number of methods, while its red color indicates that this class is abstract. The method nodes are initially hidden – in this case, the user clicked on the center of the class node to spawn them. The class had been co-evolving (see section III-E) with one other class as shown by the number at the top. The class and all three methods were created at the same time in August 2015 but deleted in recent history. The class size changed several times, apparently due to changes in two out of its three methods. The third method did not change in size at all, thus only a single *clock* sector with a uniform color is visible. Additional information for one particular revision range of the class is being displayed as the mouse is hovering over its corresponding *clock* sector.



ID	org/apache/commons/bcel6/generic/StoreInstruction
Range	[533-581]
Date	07. June 2016 19:06
SHA	d522432b7
Author	Benedikt Ritter
size	0
Children	3
Subject	Fix messed up package rename
Message	git-svn-id: https://svn.apache.org/repos/asf/commons/proper/bcel/trunk@1747278 13f79535-47bb-0310-9956-ffa450edef68

Fig. 3. The same nodes as shown in fig. 2, but using a categorical color scale for authors. The total history range is, of course, identical, but the sectors appear to have different sizes exactly because they visualize different dimensions which were changing at different times. Two methods (and thus, the class itself) were modified by two authors, the unchanging method near the bottom left by only the first. In contrast to fig. 2, the mouse is hovering over a commit where the class does not exist, such that only commit metadata is shown in the tooltip.

addition, we discuss how we can reduce information overload and optimize targeted information retrieval for a specific time frame or subcomponent of the project by allowing the user to interact with the visualization and apply simple but effective selection and filtering mechanisms.

We implemented the approach described in this section as a browser front-end to our open-source analysis framework, 𝕃ISA, using Javascript, HTML/CSS, and SVG, utilizing the renown D3 visualization library.[2] While the visualization is designed to represent the evolution of any graph-structured data, our specific use case is visualizing the history of software projects compiled for the JVM.

A. The Evo-clock

To show how individual nodes in a graph evolve independently from each other, a variety of visual tools (such as simple line charts or even polymetric views [6]) could be used. A circular visualization lends itself nicely for our use case, since it makes it easier to avoid overlaps when visually grouping several nodes together in a force-directed layout (which gravitates to hexagonal packing), as we will discuss in section III-D. Thus, just as *revision ranges* are the core conceptual term we use when talking about the evolution of individual coding artifacts, the circular *clock*, four specimens of which are shown in fig. 2, is the primary visual tool we utilize for presenting it. In bare terms, a *clock* is really just a pie chart where different sectors represent different times in the history of the project, just like with a regular clock.

If all revisions are displayed, the history begins with the oldest revision at the 12 o'clock position and runs clock-wise around the circle reaching the latest revisions upon returning to the 12 o'clock position approaching from the left. A black line serves as a reminder for the discontinuity. The sectors of each *clock* can be colorized according to various different kinds of information, as laid-out in section III-B. This means that (i) for any range of revisions where the artifact experienced no change, a sector with a larger internal angle, spanning multiple individual revisions, is rendered and (ii) if there is no change in the metric being visualized, even if the artifact has changed otherwise, the colors of adjacent sectors (representing adjacent revision ranges) will still be identical, resulting in a visually larger sector yet again. In this way, irrelevant increments in the history of an artifact (such when unrelated components or metrics changed) are completely hidden, which can be seen when comparing figs. 2 and 3, where sectors *visually* cover different time spans depending on the metric displayed. If an artifact is entirely absent in a given range, the corresponding *clock* sector is colored in gray.

B. Visualizing metrics and other information

When visualizing the history of structured data, a natural place to start with is the age of different components: a metric which any kind of artifact exhibits. By default, *clock* sectors of 1st-level nodes (which are drawn by default) are colorized in shades of blue and 2nd-level nodes (which are hidden by default) in shades of green, darker shades representing older revisions and lighter shades representing newer ones (analogous to the colors used by the range selection tool shown in fig. 4). If, for example, the left half of a *clock* is colorized in a single shade of dark blue, this indicates that the node had been existing in its current state for a long time. Also, quite naturally, multiple sectors with a small internal angle in close succession indicate frequent change.

Sector colors in each *clock* can be used to represent the evolution of both numerical and categorical data. Increasing values of numerical metrics can be placed on a continuous color scale (we use a green-yellow-red gradient). We recali-

[2]https://d3js.org/

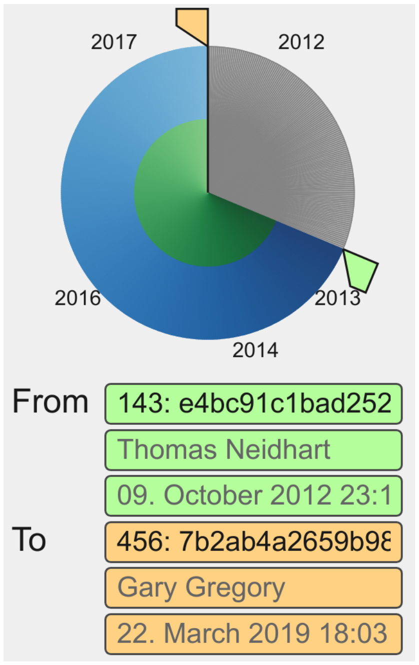
brate scales every time the diagram is updated, so that colors are interpolated between the highest and lowest value for any specific metric within the currently visible selection of nodes and time. For categorical data (such as which person authored which commit, shown in fig. 3), each category is assigned a color picked from the CIECAM02 [7] color model, maximizing the perceived differences between colors even for a larger number of categories.

Many kinds of numerical and categorical coloring conditions can conceivably be visualized in this fashion. We discuss alternative ways of (re-)computing color scales as well as additional potential metrics in section V.

C. Revision range selection

By default, the entire history of a project is visualized in a single view. However, the user might be interested only in a specific period. To view only a specific range and, thus, reduce the complexity of the visualization by filtering out-of-scope data, we designed a range selection tool, shown in fig. 4, which itself resembles a *clock* and follows the same intuition:

- A circle represents the entire history of the project being visualized.
- Blue and green gradients around the center serve as a visual reminder of the colors used to represent the age of 1st and 2nd-level nodes, as described in section III-B.
- Year labels around the circle facilitate a high-level orientation on how much time the history spans and how densely packed revisions are over different periods.



From
143: e4bc91c1bad252
Thomas Neidhart
09. October 2012 23:1

To
456: 7b2ab4a2659b9ε
Gary Gregory
22. March 2019 18:03

Fig. 4. The range selection tool. In this example, the history spans from before 2012 until after 2017. Judging from the spacing of year labels, 2012 and 2016 have seen a much larger number of impactful commits than other years, hence they take up more space on the circle's circumference. The green handle has been moved to the 4 o'clock position, thus the visualization will not render data from before October 2012.

- Two handles can be dragged around the circle, like hands of a clock, to select the start and end revisions of the history range to be displayed. The green handle signifies the selected start, the orange handle the selected end revision. Double-clicking a handle resets it to the 12 o'clock position. Revisions outside the selected range are shaded in gray, as the dark-to-light blue and green gradients move together with the handles, such that the entire gradient is always utilized for the visualization, ensuring strong visual contrast.
- Hovering over different circle sectors reveals commit metadata, including the message.
- Text indicators below the selection tool show additional information on the selected start and end revisions. The first field of each triplet accepts input, such that the user can enter a number or a commit hash to move the handles to specific commits.

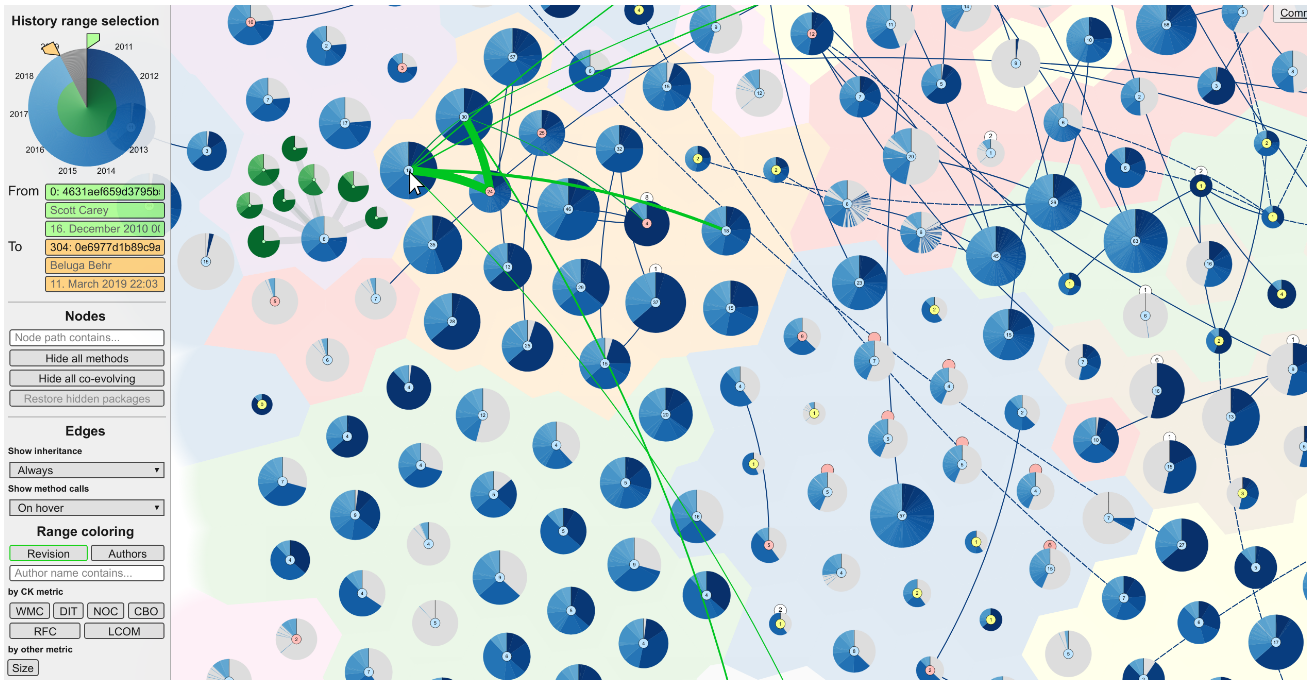
D. Node-Link diagram

When the visualization is first rendered, it displays a node-link diagram (which the user can zoom and pan) representing a limited selection of project components existing in any revision of the entire history, as shown in fig. 5. There are numerous requirements to be considered in designing the diagram worthy of individual discussion:

Nodes and types of nodes: When considering what to display, both performance overhead and information overload are imposing upper limits on the number of nodes that can be meaningfully displayed. For our visualization, we assume that the data being displayed is, at least somewhat, hierarchical. Software artifacts, specifically, can be layered going from classes or units via methods down to statements. Finally, it often makes sense to cluster nodes into groups. For our use case, we render classes as 1st-level nodes and methods as 2nd-level nodes (which are initially hidden). All nodes are grouped by their containing package. It is conceivable to add class fields as 2nd-level members or to add additional 3rd or 4th layers depending on the use case.

Each node, regardless of level, is rendered using a *clock* and displays the evolution of that particular node. Since each node represents an entire history, we use an average of a size metric to determine the radius used to draw each node. For our JVM dataset, the size is determined from the number of tree nodes in 𝕃ISA's internal representation, extracted from the JVM bytecode, although for AST-based data, we could also use the number of lines of code or any other metric.

Links and types of links: Because the diagram we render is a merged representation of multiple revisions, nodes will be rendered even if they do not exist in the most recent revision. Their final *clock* sector will be colored gray to indicate recent absence, but they will be rendered. Any connections these nodes have to other nodes will be drawn if the connections existed at any point in the history of a related node. Thus, a connection might be shown even if it does not exist in recent history. This is a necessary trade-off, which can, however, be



From 0: 4631aef659d3795b
Scott Carey
16. December 2010 0(
To 304: 0e6977d1b89c9a
Beluga Behr
11. March 2019 22:03

Show inheritance: Always
Show method calls: On hover

Fig. 5. A screenshot of the visualization with the controls overlaid on the left side. Node histories are represented as *clocks* (section III-A) using a particular color scale and legend (section III-B), with node centers indicating the number of children and using colors to indicate modifiers (blue for concrete and red for abstract classes, yellow for interfaces). Numbers above certain nodes indicate co-evolving nodes, which are hidden by default. Clicking on such a number spawns the co-evolving nodes and colorizes their tops (section III-E). Near the top-left, a 1st-level node has been expanded to show the history of any methods it had during its existence. Inheritance links are drawn blue, method calls in bright green when hovering over nodes using the mouse (section III-D2). The background behind nodes is shaded according to packages (section III-D7).

alleviated by using the range selection tool: if links or nodes do not exist in the selected range, they will not be drawn. We believe that this is a fair trade-off, as the connection between nodes was relevant at some point, and zooming into a shorter time-frame increases the accuracy of the visualization.

For our specific use case, we currently use three kinds of links. Thin, blue lines indicate inheritance. Thick, transparent bands connect 2nd-level nodes to their owning 1st-level nodes. Finally, thin, green lines indicate outgoing method calls. To avoid information overload, method calls are only shown when the user hover over a calling node by default.

Node identifiers: To position nodes and draw links in the diagram, we need to assign unique identifiers that are valid throughout the history of a node. These IDs largely depend on the application domain. In our case, 𝕃ISA determines IDs for our bytecode data using a simple tree-based path naming scheme, such that every class name is prefixed by the package name, while method nodes are using the JVM-assigned method names and signatures, prefixed by the containing class. This means that if an artifact is renamed once in its history, it will be represented by two separate nodes with non-overlapping revision ranges. Detecting such moves in software evolution is a hard problem though [8] and is out of scope for this paper.

Layout and node placement: We have several requirements on how nodes should be placed in the diagram. We need 1st-level nodes belonging to the same group to be clustered together, and it makes sense to shorten the links between nodes as much as possible to reduce the amount of clutter. We also don't want nodes to overlap. There are many possible strategies for computing node-link layouts, ranging from circle packing to static tree-like structures. Because it satisfies our requirements and is easy to use, we chose to use D3's implementation for drawing *force-directed* graphs [9]. A D3 force simulation initially places all nodes in a phyllotaxis arrangement and then performs an iterative optimization, in which the *forces* are applied to gradually move the nodes. The forces get weaker in later iterations until all movement stops and a static arrangement of nodes has been achieved. Whenever the diagram is modified (for example by adding or removing nodes), the simulation is restarted and forces can again act upon nodes to re-compute a fitting arrangement. We apply the following forces:

- A basic *centering force* ensures that nodes bunch together instead of drifting apart by pulling them somewhat towards the center of the drawing.
- A *collision force* ensures that nodes don't overlap.
- A *link force* applies a positive attraction between nodes which are connected by a link.
- A *clustering force* determines the largest 1st-level node in each group and pulls all other nodes of that group towards it. This force is stronger than the link force to ensure that nodes in the same group are nicely clustered, even if longer link links are required.

These forces in unison result in a layout where nodes are clustered in groups, with more connected graph components placed closer to the center.

Graph modification: Computing and drawing a force-directed graph layout using SVG is computationally intensive. To avoid clutter when first loading the diagram, the layout is computed covertly for $2,000$ ticks and only the final, static placement is rendered to screen. The diagram can, however, be modified in one of several ways:

- The user can choose to view only a specific period in the whole history of the project (section III-C). Nodes which do not exist in the selected range must be removed from the diagram.
- Nodes are clustered by groups (for example, packages). A user can select a subset of groups to display, thus hiding all others (section III-D7).
- The user can enter a search term that is matched against node IDs. Only nodes whose ID match the query should be rendered.
- 2nd-level nodes are not rendered initially. If the user clicks on the center of a 1st-level node, its 2nd-level nodes must be added to the graph.
- Co-evolving 1st-level nodes (section III-E) are summarized, such that only the largest among them should be rendered. The user can choose to expand that node by clicking on the circle above it, upon which all co-evolving nodes are added to the graph.

When adding or removing nodes, it is crucial to ensure *visual continuity*, as it would be hard to keep track of nodes if they were to jump about the screen whenever the layout is recomputed. So instead, the simulation is restarted and the diagram is rendered on every *tick*, giving the impression of existing nodes fluently moving to the new place where they belong.

Sticky nodes: When the diagram is modified and the simulation is running, nodes are moving towards a more ideal placement given the update. However, this movement can take several seconds and during that time it can be tricky to interact with the visualization: it is hard to click on the center of a node if that node is moving around the screen. Thus, on a minor but impactful implementation detail, a node will not move at all, if hovered with the mouse cursor, even if the simulation is still running.

Clustering and filtering groups: To visualize which higher-level group a given node belongs to, we use a common contour based approach, which has been shown to be versatile in previous research [10]. As visible in fig. 5, nodes are placed upon a colorized backdrop (computed using Voronoi tesselation [11] like in Gansner and Hu's GMap approach [12]). A limited number of colors is used, meaning that in rare cases, adjacent packages may share the same color. To learn which group a cluster of nodes belongs to, the user can hover over the backdrop to spawn a tooltip displaying the group name. Since rendering and viewing all nodes at the same time can be overwhelming, the user can select one or more groups to remain visible while all others are hidden. This is done by holding the SHIFT key and then clicking on the backdrop of each group that should remain visible (like is customary for multiple selections in most file managers, word processors and spreadsheet calculators). Once the SHIFT key is released, all non-selected groups are removed from the diagram, the simulation is restarted, and the remaining nodes are repositioned accordingly. At the click of a menu button, all hidden groups can be restored to the diagram.

E. Condensing co-evolving units

Previous research has shown that co-evolution of software artifacts is an indicator of logical coupling [13], [14]. In line with reducing the number of visible nodes to a feasible minimum, we analyze the revision ranges of all nodes and group those nodes (across all groups), whose history have exactly the same revision ranges. Out of all these nodes, the largest one is selected as the representative for the whole group and will be displayed while all other co-evolving nodes remain hidden. The number of hidden nodes is printed in a small circle above the representative. Clicking this field will (i) make the hidden nodes appear next to the selected node, (ii) colorize the clicked circle, as well as the circles above the corresponding co-evolving nodes with the same color, and (iii) restart the simulation, such that the previously hidden nodes have a chance to move towards their desired place (even as they might be from different packages). Since the circles above all co-evolving nodes have the same color, it remains easy to track them, even if they belong to other packages, as shown in fig. 6.

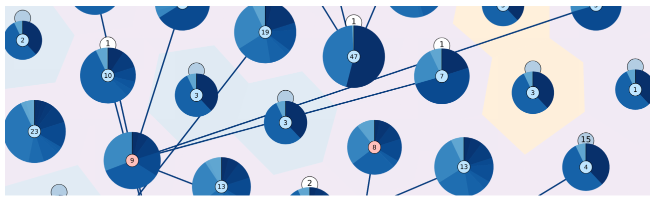


Fig. 6. Co-evolving classes (as indicated by their blue-colored top circle) spread across the `bcel.generic`, `bcel.classfile` and `bcel.verifier.statics` packages.

IV. Evo-Clocks in Practice

Joining a new project as a developer can be an overwhelming experience. It is challenging to familiarize oneself with a project just by browsing source code or reading issues and API documentation. In particular, it is *hard to even know where to start exploring*. Furthermore, the history of a project may be underestimated as a valuable source of information, even though the current state of the code base is the result of a dynamic evolution involving many design decisions, people – and even mishaps – from times long past.

To understand the usability of our approach and the extent to which it facilitates project familiarization, we explored the core module of four subject systems shown in table I, ourselves. These systems, along with others from the Apache

TABLE I
PROJECTS ANALYZED USING EVO-CLOCKS.

Project	Impactful commits
Commons-Codec	456
Commons-Pool	373
Avro	329
Bcel	581

ecosystem, are frequently used as study subjects in software engineering research. To visualize inter-object relationships (such as inheritance and method calls), analyzing source code or abstract syntax trees only was not an option. For this reason, we utilized SQUAAD [15], a framework for compiling software histories, to generate the necessary bytecode to be fed into 𝕃ISA for analysis and visualization. Naturally, there exists a large number of commits which introduce build failures or do not affect the compilation at all. SQUAAD uses commit-impact analysis [16] to reduce the number of uncompilable revisions, increasing the compilation ratio for impactful commits to almost 100%. In total, it found and compiled $1,739$ commits that produce a distinct revision of the core modules in our subject set.

A. *Finding a starting point*

Even without prior knowledge on a project, merely opening Evo-Clocks and seeing a high level overview – of both structure and history – already conveys significant and valuable information about the overall shape of the project, like the edit history of files, the density of development activity, inheritance and access dependencies between artifacts, and thus, potentially interesting candidates for further exploration.

Using Evo-Clocks to get familiar with Apache Avro, a new developer would immediately be able to identify which classes are often changed by many people (e.g., `GenericData`) or remain stable (e.g., `SpecificErrorBuilderBase`). They can recognize important top-level types which are frequently inherited or implemented (e.g., `SpecificRecordBase`), and also how inheritance is structured across package boundaries (e.g., all classes in `io` inherit within the package, except for `DatumReader` and `DatumWriter`, which are inherited by classes in `generic`).

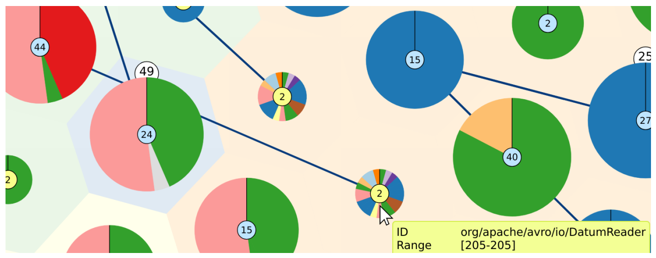
This initial orientation, which draws both from the structure as well as the history of the project, can provide a starting point for continued exploration unlike comparable tools that do not visualize the fine-grained history of entities together with their structure. In the following sections, we describe several discoveries we made while exploring our data and which may be relevant to a developer starting to work on these projects.

B. *Development activity and authors*

When joining a project, auditing existing components or working on related code, it is useful to know who has been working on which parts of the source code, and which parts of the source code need frequent changes.

Our visualization, with *clock* sectors colorized by author, makes it easy to explore both these questions. For example, the two aforementioned interfaces, `DatumReader` and `DatumWriter`, shown in fig. 7, have been modified together, frequently, and by many different authors, while the other classes in the same package have not been modified in a long time. We can also spot groups of classes which have been edited together, albeit less frequently.

Looking at the commit history of individual files, for example by using Git, is an alternative, though tedious, source for this information. The main problem is that while `git log` can identify multiple commits to one file and `git blame` can identify the most recent author for any line of code, there is no direct way to see the most recent couple of authors for multiple closely related files while retaining a mental image of how the files involved relate to each other.



ID org/apache/avro/io/DatumReader
Range [205-205]

Fig. 7. Using a categorical scale for author names, we can distinguish classes frequently changing together or being modified at the same time and thanks to the structural layout, we are able to put these classes into context.

C. *Large-scale architectural changes*

While 𝕃ISA does not detect moves, it is possible to recognize them visually. The example in fig. 8 illustrates how the BCEL project renamed the `org.apache.commons.bcel` package to `org.apache.commons.bcel6` about half way through its evolution, and then back again in recent history. This appears – even visually – to have been an incisive design choice that was nonetheless reverted a year later, likely having had an impact on other software depending on BCEL. We can actually confirm this by hovering over the respective revision sectors of any affected class, revealing the commit message "Major release of BCEL requires updating package name and maven coordinate" in August 2015, and then later: "Fix messed up package rename" in June 2016.

Although nodes not present in the most recent revision are being rendered, it is possible to recognize which parts of the graph are relevant for which part of the project's past by visual intuition, given the cardinal direction on the face of any *clock* towards which sectors are drawn. Restricting the revision range using the range selector can confirm intuition if necessary and provide a deeper insight into specific periods of time.

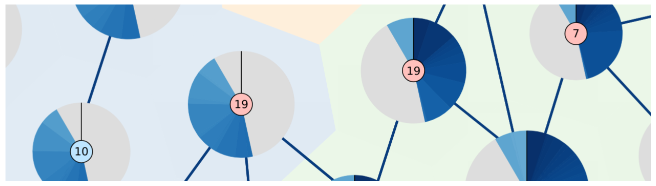
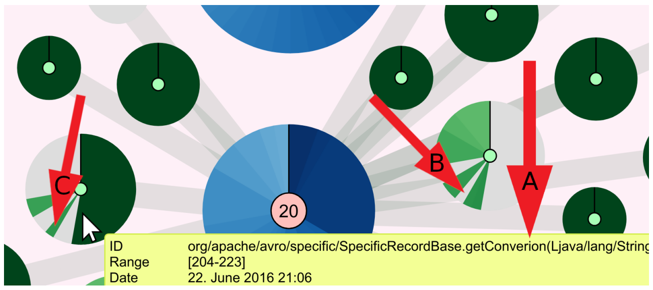


Fig. 8. Renamings of packages can be recognized via several nodes exhibiting complementary revision ranges.

D. Discovering pitfalls

`SpecificRecordBase` is the most inherited class of the Avro project. With a single click on the central number, its 20 methods that existed in its history are revealed, as shown in fig. 9. Two methods appear to be alternating in their presence. A quick glance at their history reveals that the second method in question fixed a typo in the method name of the `getConver`**`s`**`ion(String fieldName)` method. However, the fix was reverted just two impactful commits later. It was introduced a second time, and reverted yet again. It is possible that the fix broke too much depending software. A third fix was finally permanent. Given that there is another method, spelled correctly but accepting an Integer (`getConversion(int field)`), this could have been a source of confusion when working with or on this project for quite some time. The deprecation warning is still present in the API even today.



ID org/apache/avro/specific/SpecificRecordBase.getConverion(Ljava/lang/String
Range [204-223]
Date 22. June 2016 21:06

Fig. 9. `SpecificRecordBase` and its methods. The large arrow A points to the typo, the small arrows B and C indicate where fixing the typo was reverted for the first time.

E. Summary

Our findings show that *at a glance*, the user can gain detailed knowledge regarding hot spots, developer activity, frequently changed components, and important architectural changes, as well as the high-level structure of a project. Irregular parts of the software or its evolution signify interesting starting points for understanding the intricacies of a project without any prior knowledge and without knowing where to start. Spotting outliers is interesting because they usually have a story behind them. Our visualization arguably helps revealing interesting matters, albeit without letting the user drown in information. We believe that placing historical data on each software component into a structural context makes it easier to get one's bearings in an unfamiliar software project, compared to existing visualizations that lack one or the other. That said, additional research is needed to confirm these assumptions.

V. Discussion

The design of any visualization is subject to trade-offs. We chose to visualize quite a lot of information simultaneously, while providing users with the opportunity to limit the amount of data being rendered. Having the option to filter, collapse, and expand information on-demand strongly increases the explorability of our visualization.

We are not rendering a perfect picture of the structure of the project, but we think the chosen view provides a sufficient, intuitive understanding of its shape. For very specific questions, tailored tools are likely better suited for finding an answer. However, we see great potential in our approach as an exploratory tool, exploration often being the first step towards asking more specific questions.

We envision numerous extensions through which the Evo-Clock visualization can be improved in future iterations. Furthermore, we wish to conduct purpose-specific comparisons with other approaches as well as controlled lab studies with test subjects using our tool. In this section, we reflect on many different aspects of the visualization, indicate shortcomings and possible improvements.

A. Polymetric "clocks"

Disregarding the numbers and colored center spot, our *clocks* currently depict only a single metric at a time, which is sufficient given the scope of this paper, but limits the expressiveness of the visualization. We are considering to expand the compactified visualization of component histories by adding additional visual features. Besides colorizing entire node sectors and the center of a node, we could also add an outer border to each node to indicate an additional dimension. For example, the border color could be used to highlight search results when implementing additional search functionality. Furthermore, instead of coloring entire sectors with a single color, multiple metrics could be displayed using one of the methods sketched in fig. 10.

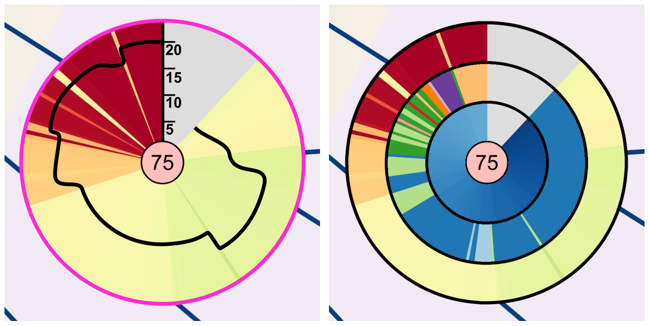


Fig. 10. Two artificially created concepts for extending the *clock* to render additional dimensions. On the left, a line chart indicating a metric ranging from 5 to 20 is wrapped around the center with an axis drawn at the 12 o'clock position. Also, the outer edge, in pink, may highlight this node as a search result. On the right, the evolution of three different dimensions is drawn in concentric rings around the center.

B. Improving scales

We mention in section III-B that scales are recalibrated every time the graph is updated. We made this decision with the goal of utilizing the entire scale no matter which nodes are drawn. This could, however, be made optional, so that color scales will always reflect the magnitude of values relative to *all* other nodes, whether visible or not. Furthermore, the same scale is currently used for 1st and 2nd-level nodes, which, for large 1st-level nodes containing many small 2nd-level nodes, results in one overly red and many overly green nodes. Thus, it makes sense to compute separate scales for different levels, or even within a group of siblings. It is also feasible to add additional options, like using a logarithmic instead of linear scale. In addition, although our current approach of using a green-yellow-red scale for continuous numerical metrics is arguably intuitive, color blind people may appreciate additional color options.

With regard to the time scale, currently all time-steps have the same internal angle, thus periods of high-frequency change take up more space than periods with less change. We could use the actual time passing between commits to determine the internal angle, which would result in time being uniformly distributed around the *clock*.

C. Evolution of edges

Edges between two nodes in the graph are currently rendered if they exist in at least one of the revisions being viewed, which may be misleading in some cases. It could be useful to give the user more control:

- An additional handle on the range selection tool could be used to select a single revision for which the edges should be drawn. Moving the handle back and forth would show the change in structure with edges popping in and out of existence. As only the edges for a single revision would be visible, this would be a playback-based solution, albeit only partially so, since nodes in the visualization would nonetheless remain static.
- A color gradient spanning the length of the edge could inform the user of its evolution, just like the sectors of the *clock* do. This option would avoid the need for playback, but it may be overly colorful and thus hard to understand.
- The tooltip shown when hovering over edges could be augmented with its own *clock* to render the evolution of edge-related metrics.

D. Different application domains

While we designed the visualization of evolutionary data using *clocks* with software in mind, the visualization's applicability is not limited to software evolution and neither is it limited to two levels of data nesting (classes and methods in our example). The same visualization of localized change embedded in a larger structure could apply to a variety of other domains, such as (i) social networks and marketing, where nodes visualize historical data on individuals and edges visualize their relationships, (ii) logistics networks, where nodes could visualize the evolution of stock or other data, (iii) geographical data, where nodes could have several nesting levels from countries down to municipalities or even individual households, and finally (iv) versioned Semantic Web (RDF) data for arbitrary knowledge domains

𝕃ISA already supports a variety of programming languages and it has built-in generic data structures which supporting versioned graphs for arbitrary application domains. Thus, it is feasible to adapt our open-source implementation to support visualizing different kinds of data.

E. Data linking

Coming back to the context of software development, it could be useful to integrate additional information sources, such as issue trackers or communication channels. At the very least, we plan to implement facilities for linking artifacts to their source inside the version control system, such that a user can reach the source code revision corresponding to any *clock* sector. Naturally, the idea of linking data to other channels also applies to any other application domain.

F. Improvements to visuals and performance

Using a different drawing technology (such as HTML5 Canvas or even just OpenGL) might certainly improve the performance of the visualization even when rendering a large number of nodes. However, it could be argued that a larger number nodes being displayed would be of no benefit due to information overload. It would be possible to simply set a node limit, such that only the *n* largest or most recently added nodes are rendered, to retain the intuitive understanding of the project structure while reducing the actual number of nodes rendered. Our approach scales well to hundreds of impactful revisions, with however if there were thousands, the individual arcs would be extremely small and SVG rendering would likely become unresponsive. In this case, individual arcs would need to be merged not just visually (i.e. drawing them with the same color), but using the underlying drawing technology. The Avro and BCEL projects in our sample are not small and the approach works well. For much larger projects it would likely be necessary to add an additional level of indirection to view sub-components of a project individually.

As there are no labels on individual nodes it can be hard identify which node represents which software artifact. After browsing for a while, one recognizes certain entities by their history, edge placement or numbers, but most of the time, it is necessary to hover over the node to reveal its ID. A possible solution to this problem would be to only show node labels when zooming in close to a group of nodes, such that the labels will not overlap with other drawing elements.

Another improvement to both visual clarity and performance would be to make edge drawing optional, unless the user hovers over a node that is connected by an edge.

We can also extend the amount of information available for 2nd-level nodes, utilizing their centers to convey information, just like with 1st-level nodes.

G. Improvements to graph modification and comprehension

At the moment, expanding a 1st-level node which has many children does not provide the most pleasing visual experience, as dozens of moving method nodes clutter the display until they have settled at their most ideal placement. It might be possible to limit the number of additional nodes spawned on a single click to, for example, load only the largest methods on the first expansion. We also consider hiding all nodes which are not directly linked to the expanded 2nd-level nodes.

It would also be beneficial to add a node filter that works by selecting one or more nodes (for example using a "lasso" tool) and then only showing these nodes plus any nodes coupled to them by inheritance, co-evolution or access, similarly to how multiple packages can be selected at the moment.

H. Various minor improvements

We think that it might be beneficial to be able to specify the *strictness* of how co-evolving nodes are grouped. It might make sense to group nodes even if their history slightly differs, as long as most of it is identical. This would further reduce clutter and convey additional information on logical coupling within a project.

2nd-level nodes are not providing much data in our current proof-of-concept implementation. It is possible to implement additional metrics, such as cyclomatic complexity or call counting, to be displayed when rendering method nodes.

VI. Related Work

As gathered in systematic reviews by Novais et al. [17] and Khan et al. [18], there is a large corpus of existing research in software evolution visualization, thus we focus on the most closely related work.

Ever since polymetric views were first formalized by Lanza et al. for the use of software visualization [6], the concept has been used extensively. Lanza et al. themselves designed a visualization superficially resembling Evo-Clocks, where multiple metrics of a few modules, evolving over a small number of revisions, are drawn on the radial axes of a Kiviat diagram [19]. The diagrams themselves are arranged in a graph, with edges indicating coupling. Evo-Clocks, however, visualize a much larger number of revisions (arguably with less clutter, since axes are not repeated on every node) at the class and method, rather than the module level, albeit currently just one metric at a time. Later, Lanza et al. also experimented with 3D variations of polymetric views [20]. Both Telea et al. and Hassan et al. have presented tools for exploring source code evolution that include fine-grained evolution views using similar color-coding as our *clocks*, but they do not show structure simultaneously, as evolving components are visualized in parallel stripes instead of circles in a graph [21]–[23]. On the other hand, Telea et al. have also visualized changes in the structure of code by matching elements in different revisions, flattening their syntax trees vertically, and then connecting matched elements using horizontal lines [24]. This visualization, however, does not visualize code metrics or other properties, only structure, and only works for a small number of revisions.

With the goal of combining structural and metric information, Gonzales et al. have proposed an approach that visualizes metrics and commits in a circular view that also starts the history at the 12 o'clock position, going around clockwise [25]. However, it only provides a single, global overview and, thus a separate tree view is utilized to browse individual artifacts, with metrics being displayed using regular line charts.

Nam et al. have developed EVA, a tool for understanding changes in software architecture [26], which uses small circles to visualize methods and classes, packed into larger circles representing packages, without rendering edges for inheritance or method access. Unlike our visualization, EVA is better suited for comparing a very small number of major revisions. It renders the complete program representation for every revision being compared, stacks the versions in layers, and connects entities that are present in different revisions when the user hovers over them with the mouse. It incorporates issue tracker data to help developers understand architectural changes.

Numerous multi-frame approaches have been developed, where only one revision is shown at a time and change is visualized by animation. Examples are evolving versions of "software cities" [27] by Langelier et al. [28], Steinbrückner and Lewerentz [2], Boccuzzo and Gall [29] and Wettel and Lanza [3], the latter two also supporting some kind of static timeline view for a limited number of revisions. Another multi-frame visualization, drawing individual releases at a fine-grained level (including inheritance, call graphs, and control flow) and using color coding to indicate authorship, has been demonstrated by Collberg et al. [30]. Gource [5] is a playback-based tool for visualizing the evolution of files in version control systems, which renders the file tree as a graph, with additions and removals visualized step-by-step.

What distinguishes Evo-Clocks from all existing approaches is that it is able to visualize very fine-grained history for individual components within a structure context, while avoiding visual axes and tick marks unlike comparable visualizations. Despite high information density, the opaque rendering of unchanging metrics and the ability to filter, collapse, and expand details help to avoid draining the viewers attention unnecessarily, even though every node's history, comprising hundreds of (impactful) commits, is accurately represented.

VII. Conclusion

With the goal of designing a visualization which can accurately describe fine-grained changes without the need for animated playback, we have created a generic solution for visualizing evolving graphs and show how it functions on the example of evolving software. In contrast to other approaches, our tool is not tailored to a specific question. It does, however, facilitate the familiarization with a project through simple exploration and does so by embedding the fine-grained histories of individual components into the global context of a graph. In our case study, we have found that Evo-Clocks indeed allowed us to discover ephemeral information that provides a valuable and, most importantly, tangible insight into the project's hotspots and legacy, without having any prior knowledge about these projects. The visualization is extensible and provides ample opportunity to be further enriched with additional edge representations, metrics, and information hiding techniques. In conclusion, we believe that embedding component-specific, fine-grained historical information into a structural context can amplify intuitive understanding of unknown software.

References

[1] M. Lanza, "The evolution matrix: Recovering software evolution using software visualization techniques," in *Proceedings of the 4th International Workshop on Principles of Software Evolution*, ser.

IWPSE '01. New York, NY, USA: ACM, 2001, pp. 37–42. [Online]. Available: http://doi.acm.org/10.1145/602461.602467

[2] F. Steinbrückner and C. Lewerentz, "Representing development history in software cities," in *Proceedings of the 5th International Symposium on Software Visualization*, ser. SOFTVIS '10. New York, NY, USA: ACM, 2010, pp. 193–202. [Online]. Available: http://doi.acm.org/10.1145/1879211.1879239

[3] R. Wettel and M. Lanza, "Visual exploration of large-scale system evolution," in *2008 15th Working Conference on Reverse Engineering*, Oct 2008, pp. 219–228. [Online]. Available: http://doi.acm.org/10.1109/WCRE.2008.55

[4] C. V. Alexandru, S. Panichella, S. Proksch, and H. C. Gall, "Redundancy-free analysis of multi-revision software artifacts," *Empirical Software Engineering*, Jul 2018. [Online]. Available: https://doi.org/10.1007/s10664-018-9630-9

[5] "Gource," https://gource.io, accessed: 2019-04-20.

[6] M. Lanza and S. Ducasse, "Polymetric views - a lightweight visual approach to reverse engineering," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 782–795, Sep. 2003.

[7] N. Moroney, M. D. Fairchild, R. W. G. Hunt, C. Li, M. R. Luo, and T. Newman, "The ciecam02 color appearance model," in *Color Imaging Conference*, 2002.

[8] M. Kim and D. Notkin, "Program element matching for multi-version program analyses," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 58–64. [Online]. Available: http://doi.acm.org/10.1145/1137983.1137999

[9] S. G. Kobourov, "Spring embedders and force directed graph drawing algorithms," *CoRR*, vol. abs/1201.3011, 2012. [Online]. Available: http://arxiv.org/abs/1201.3011

[10] C. Vehlow, F. Beck, and D. Weiskopf, "Visualizing group structures in graphs: A survey," *Computer Graphics Forum*, vol. 36, no. 6, pp. 201–225, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12872

[11] F. Aurenhammer, "Voronoi diagrams&mdash;a survey of a fundamental geometric data structure," *ACM Comput. Surv.*, vol. 23, no. 3, pp. 345–405, Sep. 1991. [Online]. Available: http://doi.acm.org/10.1145/116873.116880

[12] E. R. Gansner, Y. Hu, and S. Kobourov, "Gmap: Visualizing graphs and clusters as maps," in *2010 IEEE Pacific Visualization Symposium (PacificVis)*, March 2010, pp. 201–208. [Online]. Available: http://doi.org/10.1109/PACIFICVIS.2010.5429590

[13] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, Nov 1998, pp. 190–198.

[14] R. Robbes, D. Pollet, and M. Lanza, "Logical coupling based on fine-grained change information," in *2008 15th Working Conference on Reverse Engineering*, Oct 2008, pp. 42–46.

[15] P. Behnamghader and B. Boehm, "Towards better understanding of software maintainability evolution," in *2018 Conference on Systems Engineering Research (CSER 2018)*, Charlottesville, USA, May 2018.

[16] P. Behnamghader, R. Alfayez, K. Srisopha, and B. Boehm, "Towards better understanding of software quality evolution through commit-impact analysis," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, July 2017, pp. 251–262.

[17] R. L. Novais, A. Torres, T. S. Mendes, M. Mendonça, and N. Zazworka, "Software evolution visualization: A systematic mapping study," *Information and Software Technology*, vol. 55, no. 11, pp. 1860 – 1883, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584913001298

[18] T. Khan, H. Barthel, A. Ebert, and P. Liggesmeyer, "Visualization and Evolution of Software Architectures," in *Visualization of Large and Unstructured Data Sets: Applications in Geospatial Planning, Modeling and Engineering - Proceedings of IRTG 1131 Workshop 2011*, ser. OpenAccess Series in Informatics (OASIcs), C. Garth, A. Middel, and H. Hagen, Eds., vol. 27. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012, pp. 25–42. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2012/3739

[19] M. Pinzger, H. Gall, M. Fischer, and M. Lanza, "Visualizing multiple evolution metrics," in *Proceedings of the 2005 ACM Symposium on Software Visualization*, ser. SoftVis '05. New York, NY, USA: ACM, 2005, pp. 67–75. [Online]. Available: http://doi.acm.org/10.1145/1056018.1056027

[20] C. Mesnage and M. Lanza, "White coats: Web-visualization of evolving software in 3d," in *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, Sep. 2005, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/VISSOF.2005.1684302

[21] A. Telea and L. Voinea, "Interactive visual mechanisms for exploring source code evolution," *Visualizing Software for Understanding and Analysis, International Workshop on*, vol. 0, p. 17, 09 2005.

[22] L. Voinea and A. Telea, "Multiscale and multivariate visualizations of software evolution," in *Proceedings of the 2006 ACM Symposium on Software Visualization*, ser. SoftVis '06. New York, NY, USA: ACM, 2006, pp. 115–124. [Online]. Available: http://doi.acm.org/10.1145/1148493.1148510

[23] Jingwei Wu, R. C. Holt, and A. E. Hassan, "Exploring software evolution using spectrographs," in *11th Working Conference on Reverse Engineering*, Nov 2004, pp. 80–89.

[24] A. Telea and D. Auber, "Code flows: Visualizing structural evolution of source code," *Computer Graphics Forum*, vol. 27, no. 3, pp. 831–838, 2008. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2008.01214.x

[25] A. Gonzalez, R. Theron, A. Telea, and F. J. Garcia, "Combined visualization of structural and metric information for software evolution analysis," in *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, ser. IWPSE-Evol '09. New York, NY, USA: ACM, 2009, pp. 25–30. [Online]. Available: http://doi.acm.org/10.1145/1595808.1595815

[26] D. Nam, Y. K. Lee, and N. Medvidovic, "Eva: A tool for visualizing software architectural evolution," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 53–56. [Online]. Available: http://doi.acm.org/10.1145/3183440.3183490

[27] R. Wettel and M. Lanza, "Codecity: 3d visualization of large-scale software," in *Companion of the 30th International Conference on Software Engineering*, ser. ICSE Companion '08. New York, NY, USA: ACM, 2008, pp. 921–922. [Online]. Available: http://doi.acm.org/10.1145/1370175.1370188

[28] G. Langelier, H. Sahraoui, and P. Poulin, "Exploring the evolution of software quality with animated visualization," in *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, Sep. 2008, pp. 13–20.

[29] S. Boccuzzo and H. Gall, "Cocoviz: Towards cognitive software visualizations," in *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, June 2007, pp. 72–79. [Online]. Available: http://dx.doi.org/10.1109/VISSOF.2007.4290703

[30] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," *Proceedings of ACM Symposium on Software Visualization*, 03 2003.