Vision, Modeling, and Visualization (2016)
M. Hullin, M. Stamminger, and T. Weinkauf (Eds.)

Metric Evolution Maps: Multidimensional Attribute-driven Exploration of Software Repositories

R. R. O. da Silva[1,2] and E. F. Vernier[3] and P. E. Rauber[2] and J. L. D. Comba[3] and R. Minghim[1] and A. C. Telea[2]

[1]Institute of Mathematical and Computer Sciences, University of São Paulo, Brazil
[2]Johann Bernoulli Institute, University of Groningen, the Netherlands
[3]Informatics Institute, Federal University of Rio Grande do Sul, Brazil

Abstract

Understanding how software entities in a repository evolve over time is challenging, as an entity has many aspects that undergo such changes. We cast this problem in a multidimensional visualization context: First, we capture change by extracting quality metrics from all software entities in all revisions in a software repository, yielding a multidimensional time-dependent dataset. Next, we propose Metric Evolution Maps (MEMs), a new visual approach to create dynamic maps that show the similarity of entities in a revision and changes across revisions. We enrich MEMs with visual cues to show which metrics and metric values are key to formation of similar-entity patterns. Additionally, we show how entities change between revisions, and due to which metrics. We illustrate our approach by exploring changes in two real-world software repositories.

1. Introduction

A key issue in software engineering is to understand how a project is organized during its lifetime. For this, two approaches can be taken. First, one can analyze changes of the *explicit* project structure, captured by its physical or logical hierarchy and dependencies [BD08,TA08,HvW08]. Alternatively, one can mine the repository to find its *implicit* structure, *i.e.*, aspects which create groups of highly-related entities. Such aspects can be inferred from the perspective of software quality metrics [LM06], source code (clones) [RCK08], co-change [Ant05], and lexical term frequencies [KELN10]. Detecting implicit structure can uncover previously unknown patterns and changes that, in turn, lead to more insights on the software evolution.

A challenge of analyzing implicit structure and its change is that code analysis delivers tens of such metrics, each capturing a different facet of the software [Sci16]. Understanding how entities in a software system relate to each other and how such relations change in time implies understanding how tens of measurements on hundreds of items change over hundreds of time steps. Using just a few metrics (over all entities) can easily miss important underlying aspects relating the entities [VT06]. Conversely, considering all metrics over a few entities offers only a coarse-grained view [LWC07]. To explore the full data space, we essentially have to understand the evolution of a multidimensional dataset D of n metrics, measured on m entities, over T time moments, for large values of n, m, and T. To do this, we need ways to capture and depict metric-implied patterns formed by groups of entities, and ways to track pattern changes over time.

We approach this problem by proposing Metric Evolution Maps (MEMs), a new approach for the visual exploration of multidimensional time-dependent data. We extract software quality metrics, mined on all entities, *e.g.* classes, of all revisions of a repository to find groups of highly-similar entities, by using multidimensional projection (MP) techniques. We explain such groups in terms of their shared metrics and metric-value properties. Finally, we explain change patterns at entity and group level in terms of the underlying metric changes. Overall, we aim to answer two types of questions:

Q1: How do entities group in a given revision? Which are the main groups? Which metrics determine these groups?
Q2: How do groups change in time? How do entities migrate between groups, and due to which metric changes?

We address the above by two contributions. To support Q1, we enhance static MPs with automatically labeled heat-map-like plots. To support Q2, we enhance a recent time-dependent MP with bundled trajectories and interaction.

The rest of this paper is organized as follows. Section 2 overviews related work on visualizing high-dimensional data with a focus on software understanding. Section 3 describes our approach. Section 4 shows how MEMs can be used to discover several aspects, and their changes, in two real-world software repositories. Section 5 discusses our results. Section 6 concludes the paper.

2. Related Work

Many techniques have been proposed to visualize similarity and changes of software entities such as code lines [VTvW05], syntactic blocks [TA08], hierarchies [BD08, HvW08], and code clones [Han13]. Such techniques typically consider just a few attributes to model change and hence fall out of our scope. At the other end, static analysis and repository mining extract tens of metrics such as code size, complexity, cohesion, coupling, number and type of bugs, and identity of developers [LM06, MD08, VT06]. Depicting patterns of similar and/or related entities and their changes is hard, due to the high dimensionality of the data. Several techniques try

to attack this, as follows. Space-filling techniques such as table lenses [RVET14], evolution lines [VTvW05], and evolution matrices [Lan01] use a 2D Cartesian layout populated with sparkline-like encodings [Tuf06] to show the change of metrics *vs* entities *vs* time. Variations include 3D Cartesian layouts and UML layouts using bar charts [LWC07, GJR99]. No such approach can, however, show a data space having hundreds of entities, tens of metrics, and hundreds of change moments. Also, finding groups of similar entities, or finding how entities change in time with respect to each other and with respect to the underlying metrics, is hard.

The second type of methods uses a map metaphor to place entities to reflect their metric-induced similarity. Self-Organizing Maps (SOM) visualize multidimensional data in terms of a 2D grid where cells map entities placed based on their multidimensional similarity [Mac05, RPP02]. However, the user must define the number of cells in the SOM to give the desired granularity level, which may not be known beforehand. Other methods use multidimensional projections (MPs): For a set of entities $E = \{\mathbf{e}_i\} \subset \mathbb{R}^n$ having n real-valued metrics each, an MP creates a set of typically 2D points $P = \{\mathbf{q}_i\} \subset \mathbb{R}^2$ so that the pairwise distances $\|\mathbf{q}_i - \mathbf{q}_j\|$ are as close as possible to $\|\mathbf{e}_i - \mathbf{e}_j\|$. The resulting 2D scatterplot-like image P can be used to find groups of similar entities as well as outliers in E. Modern MP techniques score highly in distance preservation for a *static* dataset E, *e.g.*, ISOMAP [TdSL00], LAMP [JPC*11], LSP [PNML08], and t-SNE [vdMH08]. Compared to other high-dimensional visualization techniques such as table lenses, evolution lines, evolution matrices, parallel coordinates [ID90], and scatterplot matrices [Har75], MPs are much more scalable in number of entities m and dimensions n, and support better finding groups of related entities. Yet, MPs are susceptible to two problems:

Group explanation: MPs do not show by default which dimensions are responsible for group formation. Recently, Da Silva *et al.* addressed this by color-coding points $\mathbf{p}_i \in P$ to show the dimension(s) best explaining the neighborhood around $\mathbf{p}_i$ [dSRM*15]. However, the delineation and labeling of groups is done manually, and dimension-values are not used in the explanation.

Evolution: Most MPs do not handle collections of time-varying entities E_t, each defined for a given time step t. We know only two exceptions, as follows. Kuhn *et al.* [KELN10] use multidimensional scaling (MDS) [STSS05] to show the evolution of lexical similarity of source-code entities, computed based on term (identifier) frequencies processed by Latent Semantic Indexing (LSI) to factor out synonymy and polysemy. To make the projections P_t consistent over time, they propose two variants – offline MDS, *i.e.*, projecting the union of revisions $\bigcup_{1 \le t \le T} E_t$; and online MDS, *i.e.*, constructing P_{t+1} by projecting E_{t+1} using P_t as an initialization of MDS. Both above variants have issues: Offline MDS creates high projection errors for a large number of time steps T, as it tries to preserve distances between points in *any* two revisions E_{t1} and E_{t2}. Online MDS is strongly biased and can easily converge in a local minimum. Both above issues also apply to the well-known t-SNE MP technique [vdMH08] by Rauber *et al.* [RFT16]. The recent dynamic t-SNE (dt-SNE) technique in [RFT16] is, to our knowledge, the only MP for time-dependent datasets that offers verified guarantees in terms of spatial and temporal coherence. Trade-off between preservation of distances in the same projection *vs* preservation of distances across projections which are close in time is controlled by a user parameter. However, dt-SNE has not yet been used for software evolution exploration, nor for any other real-world dataset. Also, dt-SNE shows change patterns by animation, which makes tracing such patterns hard for more than a few time frames P_t.

In the following, we show how we extend the explanatory approach in [dSRM*15] and combine it with dt-SNE [RFT16] to create our explanatory maps for metric evolution.

3. Construction of Metric Evolution Maps

We construct MEMs in three steps: metric extraction, revision visualization, and evolution visualization, as described next.

3.1. Metric Extraction

We considered eight popular tools for extracting code quality metrics: CCCC [Tim16], also used to analyze repositories in [VT06]; SourceMeter [Fro16]; CppCheck [Cpp16]; iPlasma [MMMW05]; SonarQube [Son16]; Analizo [TCM*10]; Analytix [Goo16a]; and Understand [Sci16]. We compared the tools on 12 open-source projects written in C++ and Java, and having between 26 and 635 classes and between 2 and 500 KLOC. Requirements included full automatic batch-mode usage since we need to automatically process entire repositories; handling code that does not build, since many repositories miss libraries or build rules; generating a large number of metrics, since this is the purpose of using MPs; speed, since real-world repositories contain hundreds of revisions each having thousands of files; and good documentation. The first four tools were quickly discarded as they did not comply with most of the requirements. SonarQube proved to have a complicated manual set-up. Analizo proved much slower than Analytix and Understand. Analytix, the second-best found tool, delivers about a third of the metrics of Understand, and is also slower. Overall, Understand matched all our requirements well. Using Understand, we built a Git repository metric extractor that lets users select the files, start revision r_S, end revision r_E, and the number of revision samples in $[r_s, r_e]$ to analyze. Metrics are saved in CSV tables, one per analyzed revision.

3.2. Revision Visualization

For a revision $r_t \in [r_s, r_e]$, our analysis delivers a set $E_t = \{\mathbf{e}_i\}$, where $\mathbf{e}_i$ are classes, files, or packages in r_t. We next consider classes, for exposition simplicity. Each class $\mathbf{e}_i$ has $n = 43$ metrics. For a complete description, see [Sci16]. For each r_t, we show E_t by projecting it to 2D using dt-SNE [RFT16], which preserves distances in *both* space, *i.e.*, between entities $\mathbf{e}_i$ in the same E_t, and time *i.e*, between entities $\mathbf{e}_i$ in consecutive time-frames (Sec. 2). This way, the resulting projections $P_t = dtSNE(E_t)$ can be used to reason about groups of related classes *and* how these change in time.

Displaying a projection P_t as a scatterplot, height plot [KELN10], or heat map [TMR02], shows related entity (class) groups, but not *what* they mean and *why* they appear, *i.e.*, which metrics and metric-values make these similar. To address this, we extend the projection-explanation technique in [dSRM*15], explained next.

Attribute ranking: In [dSRM*15], for each projected point $\mathbf{q}_i \in P$, the n dimensions of $\mathbf{e}_i$ are ranked in terms of how important they are in making $\mathbf{q}_i$ similar to its neighbors in P. Consider the 2D neighborhood $\nu_i^P = \{\mathbf{q} \in P | \|\mathbf{q} - \mathbf{q}_i\| \le \rho\} \subset P$ of $\mathbf{q}$, and the corresponding nD neighborhood $\nu_i = \{\mathbf{e} \in E | \mathbf{p} \in \nu_i^P\}$. Here, ρ is set to about 10% of the distance between the two farthest points in the projection. A ranking $\mu_i = (\mu_i^1, \ldots \mu_i^n) \in \mathbb{R}_+^n$ is computed for all n dimensions of $\mathbf{e}_i$, as follows. Let $GV = (var(\mathbf{e}^1), \ldots, var(\mathbf{e}^n))$ be the variance of all n dimensions over E. The rank μ_i^j, of dimension j to the similarity

$\mathbf{e}_i$ to all its neighbors in ν_i is the ratio of the local variance LV_i^j of dimension j over ν_i and global variance GV^j, *i.e.*

$$\mu_i^j = \frac{LV_i^j / GV^j}{\sum_{j=1}^{n} (LV_i^j / GV^j)}. \quad (1)$$

Low μ_i^j values give metrics j which vary less over a neighborhood, *i.e.*, explain better the local cohesion of classes, than metrics with high μ_i^j values. For full details, we refer to [dSRM*15].

Basic visual encoding: For each $\mathbf{q}_i \in P$, we compute a vector $\{(j, \mu_i^j)\}_{1 \le j \le n}$ with the IDs and ranks of all its n metrics, sorted increasingly on ranks. Next, we color map the IDs of the $C = 9$ metrics having top-ranks for most points in P via a categorical colormap built with ColorBrewer [BH16]. Metrics which are top-rank for many points get thus distinct colors. Metrics which are top-rank for few points are shown by the reserved color dark blue. The highest-rank metric for a point $\mathbf{q}_i$, *i.e.* $m = \arg\min_{1 \le j \le n} \mu_i^j$, however, may not fully explain why q_i is similar to its neighbors in P. To show this, [dSRM*15] compute a *confidence* c_i^m defined as

$$c_i^m = \frac{\sum_{\mathbf{q}_j \in \nu_c^P \wedge \arg\max_k \mu_j^k = m} \mu_j^m}{\sum_{\mathbf{q}_j \in \nu_c^P} \max_k \mu_j^k}, \quad (2)$$

where ν_c^P is a 2D neighborhood centered at $\mathbf{q}_i$, defined like ν^P but with a smaller radius $\rho_c < \rho$. This acts as a smoothing filter with kernel radius ρ_c that gives high confidence to homogeneous (same top-rank) regions and low confidence to mixed regions having points with different top-ranks. The final scatterplot P is constructed by mapping top-ranks and confidences for all $\mathbf{q}_i \in P$ to hues and brightnesses, respectively, of 2D Gaussian texture-splats centered at $\mathbf{q}_i$. This technique, introduced by [MCMT14], creates a compact plot where different-color areas indicate clusters of points related by different dimensions. Figure 1a shows this for a synthetic dataset of 3000 points randomly sampled from three faces of a cube, and next perturbed by uniform spatial random noise of amplitude equal to 5% of the dataset's extent, projected with PCA. The projection consists of three 'zones' – the cube's faces – each being well explained by a single dimension, as expected. Points close to cube edges are darker, as their explanation by a *single* dimension is less confident, as expected. A global ranking legend (Fig. 1c) shows how colors map to dimensions, and the number of points in each colored area. Bars are sorted decreasingly on this value to show the importance of each dimension in explaining the *entire* projection. We see that the point count is divided in three roughly equal parts, which is correct, since each cube face has roughly the same sample count.

Improved encoding: Even for the simple cube dataset, we see a distracting reticular pattern between the plot points (Fig. 1a, insets). The pattern follows the edges of the Voronoi cells of the 2D scatterplot P, and conveys no information, being just the effect of the nearest-neighbor interpolation used by the splatting technique in [dSRM*15]. We improve this by using Shepard interpolation for both hue and brightness: For each pixel $\mathbf{x}$ in the image, we find all projected points $\mathbf{q}_i$ in a neighborhood $\nu_r(\mathbf{x})$ of radius $r = 5$ pixels centered at $\mathbf{x}$. Let $\phi : \mathbb{R}^2 \to \mathbb{R}^+$ be a smooth decaying function, set to $\phi(x) = \exp(-x^2/r^2)$. We interpolate the ranks μ_i^j of all $\mathbf{q}_i$ for all dimensions j at pixel $\mathbf{x}$ by

$$\mu(\mathbf{x})^j = \frac{\sum_{\mathbf{q}_i \in \nu_r(\mathbf{x})} \phi(\|\mathbf{q}_i - \mathbf{x}\|) \mu_i^j}{\sum_{\mathbf{q}_i \in \nu_r(\mathbf{x})} \phi(\|\mathbf{q}_i - \mathbf{x}\|)}, \quad (3)$$

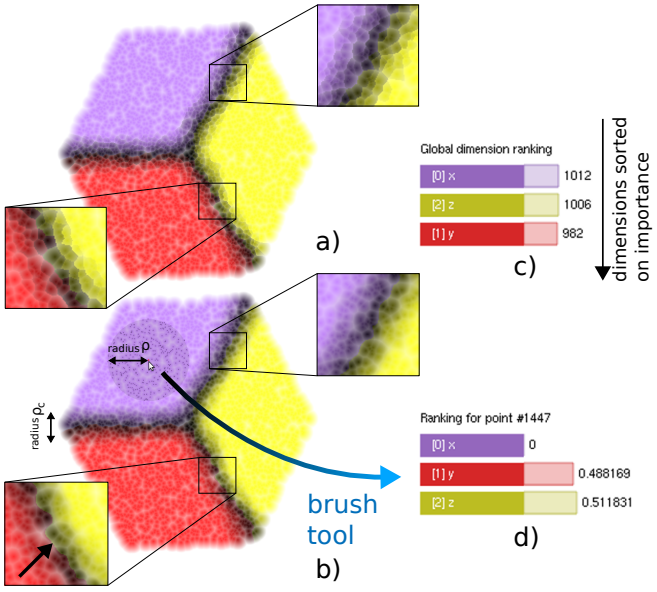

Figure 1: Visual explanation of synthetic cube dataset. (a) Original interpolation from [dSRM*15]. (b) Our improved interpolation.

and set the color of $\mathbf{x}$ by finding the dimension that maximizes $\mu(\mathbf{x})^j$ for all j. The brightness of $\mathbf{x}$ is computed analogously, by using the interpolation in Eqn. 3 for the confidences c_i^j. Figure 1 shows the result. As we can see in the insets, all Voronoi artifacts are removed, and we can see a clear *and* smooth border separating the different-hue regions. Implementing Eqn. 3 using NVidia's CUDA makes our artifact-free approach as fast as the original splat-based idea in [dSRM*15], *i.e.*, real-time for thousands of points.

Point inspection: Brushing the points allows to interactively inspect the ranks μ_i^j for a given point $\mathbf{q}_i$ via a second bar chart (Fig. 1d). Here, dimensions are sorted top-down in the same order as in the global ranking legend (Fig. 1c), so one sees how important are dimensions *locally* as opposed to globally. The top-rank dimension x (purple) of the point brushed in Fig. 1 has variance 0, which is correct, as the point is on the cube face orthogonal to the x axis.

Overall, we produce a projection where points are *implicitly* grouped, by color-coding, on the dimension best explaining their local similarity. Reading the explanation of a group involves searching its color in the color legend, which can be tedious for datasets having tens of dimensions or time-dependent datasets. Hence, answering **Q1** is not completely supported. We address this by computing *explicit* same-explanation clusters, in four steps: cluster identification, labeling, and dimension-value explanation, as explained next.

Cluster identification: We segment a projection P using a connected components approach, where a point is connected to its nearest neighbor having the same explanation *and* a confidence above 50%. This yields a set of compact same-explanation clusters C^k which, intuitively, contain same-hue points and meet at the dark, low-confidence, areas. This is fast to execute, simple to implement, and needs no user parameter setting, unlike *e.g.* hierarchical clustering used for similar tasks [NB12]. Note we do not produce a *partition* of P: very low-confident points in P are not included in any cluster. This is desirable, as we want next to reason only about the most confident point groups.

We next show the clusters by outlining them. First, we compute the convex hull $H(C^k)$ of all points in a cluster, and sample it uniformly in arc-length space, yielding a closed 2D polyline $L(C^k)$. Next, we shrink L iteratively by moving its points with a small step along its inward normals. Shrinking iterations alternate with Laplacian smoothing iterations, like in mean curvature flow [CV01], so that the contour's curvature stays low, in order to get a robust normal estimation. After each shrink-smooth pass, 10 in total, we resample L to preserve uniform point density. Contour points are not moved if they get closer than an offset value α to a point in C^k or if L would self-intersect. Figure 2 shows two cluster outlines for two α values: Small values create a tighter, but more tortuous, outline; larger values create smoother, but looser, outlines. Outline construction is fully automatic; handles convex, concave, compact, and variable-density clusters; guarantees outlines that surround all points, have a given smoothness, and are intersection-free; and has a single user parameter (α) which is simple to set. Arguably the best known related technique is alpha shapes [EKS83] which also produces contours surrounding all given points, but does not guarantee contour smoothness or user-prescribed offsets. Other techniques, such as isolines [KELN10]), do not guarantee connected single-piece outlines.

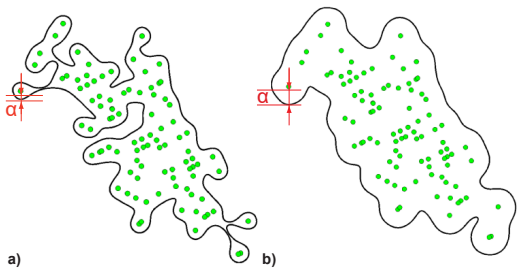

Figure 2: Cluster outlines for two α values.

Group labeling: We use the outlines $L(C^k)$ to explain the clusters C^k by labels showing their top-ranked dimensions. For each C^k, we compute the major eigenvector of the covariance matrix of the points in C^k, orient the top-rank dimension label for C^k along this direction, center the label in the centroid of C^k, and scale it so as to fit in $L(C^k)$. This makes labels 'stretch' most while staying confined in the cluster outlines, thus providing maximal readability. We have also tested other label-placement methods, such as used in tag clouds [PTT*12]. While these use only two reading orientations (horizontal and vertical), they cannot scale labels maximally, thus decrease readability. Figure 3 shows our cluster-based explanation for a dataset of 6773 projects attributed by 12 quality metrics, coming from [MSM*10]. We used here the same projection method (LAMP [JPC*11]) as in the original technique [dSRM*15]. Our explanation explicitly shows that the data consists of five clusters that capture nearly all points; a few outliers do not fall in any cluster (black points, Fig. 3b bottom). These are also points for which the top-ranked dimension has a very low confidence. Labels are well-centered in their clusters, oriented to take maximum advantage of the cluster shape, and placed fully automatically. The labels in Fig. 3 are manually placed. We can directly read the top-ranked dimensions *on the projection* itself rather than having to search colors in the color legend. This is helpful since we use both hue and luminance to encode data, which can lead to visual confusions, *e.g.*, a light brown region being mistaken for a dark orange one. Finally, labeling can handle

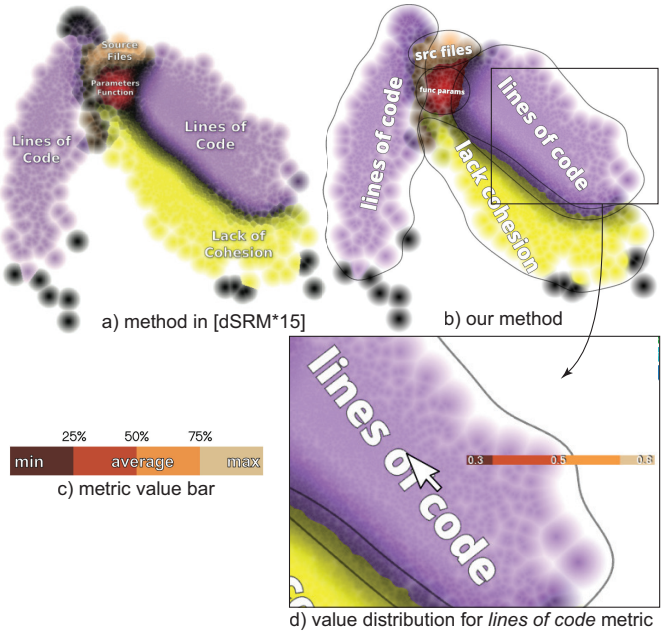

Figure 3: MEM projection explanation for dataset from [MSM*10].

any number of dimension names, while color coding is limited by the size of categorical color maps, typically under 10 entries [BH16].

Metric-value bars: So far, we partition the projection P into clusters explained by metric *names*. This can create several same-hue clusters, see *e.g.* the two purple clusters in Fig. 3b. These are point groups being similar mainly in the *lines of code* metric, but having different metric values. To clarify this, we show metric values following the model of Oliveira *et al* [OCT*13]: A 1D bar is divided into maximally four parts, colored using an ordinal colormap (Fig. 3c). Each part maps an equal-sized interval between the minimum and maximum value, for a given cluster, of its top-rank metric. The length of a part shows the number of data values in that range. Three labels atop the bar show the minimum, average, and maximum values of the metric in a cluster – the bar is thus a compact four-bin histogram of the metric values in a cluster. Metric-value bars are shown when brushing a cluster label, so they do not clutter the visualization. Figure 3d shows how we use such bars to explain the two purple clusters (*lines of code* metric) in Fig. 3b: The minimum value for *lines of code* in the brushed right purple cluster is 0.3, attained only for few data points (short dark brown bar segment). About half of the points have values from 25% up to 50% of the maximum *lines of code*, as shown by the dark orange bar segment. Other points have values from 50% to 75% of the maximum, see the light orange bar segment, which is roughly 30% of the cluster size. The remaining points have values from 75% to 100% of the maximum, see the beige bar segment. Brushing the left purple cluster in Fig. 3 shows much lower metric values. Hence, the right purple cluster in Fig. 3b contains projects that are similar because they are large (many lines of code), and the left purple cluster contains projects that are similar because they are small (few lines of code).

3.3. Evolution Visualization

The techniques presented in Sec. 3.2 visualize a single revision from a repository by a colored and annotated projection. For a set of T revisions E_t, $r_s \leq t \leq r_e$, we construct T corresponding projections P_t,

using the dynamic t-SNE (dt-SNE) technique in [RFT16]. As outlined in Sec. 3.2, dt-SNE preserves well *both* (a) distances between points in a projection and (b) between points in projections for close time-frames. Feature (a) enables us to reason about groups of similar classes at a given time, *i.e.* supports answering **Q1**. Feature (b) allows us to reason about the amount of change (of class metrics) by looking at the amount of visual change, *i.e.* supports answering **Q2**.

To visualize change, we propose two options. First, we use a small-multiple or animation of projections P_t, drawn all using the same metric-ID-to-color mapping (see Fig. 5 and Sec. 4.1 for details). This works well for analyzing a short time range, but does not scale to hundreds of revisions. For this, we use a second method: Let $\pi_i = (\mathbf{q}_i^{r_s}, \ldots, \mathbf{q}_i^{r_e})$ be the 2D projections in $P_{r_s}, \ldots, P_{r_e}$ for a class $\mathbf{e}_i$. We next construct polylines from the points π_i for all classes. These can be thought of as 'trails' showing the evolution of classes. For projects with hundreds of classes and revisions and large change dynamics, drawing the raw trail-set $\{\pi_i\}$ creates a cluttered image. Instead, we show a simplified view by bundling trails using the technique in [vdZCT16]. Any other trail or graph bundling techniques can be used equally well. Figure 4a shows the trails for the Guice repository evolution discussed in Sec. 4.2. Trails are colored from green (first revision r_s) to red (last revision r_e), similar to other bundled visualizations [HvW08]. The trail pattern allows interpreting change dynamics: Short trails show classes that stay very similar to each other, metric-wise, while long trails suggest the opposite. The splitting and merging of classes may show how entities have grown to be more similar or diverse throughout the project's history. Trails thus give a high-level *overview* of the project dynamics. For *detail* insights, we provide two extra views: First, we can select a group of classes in any time-frame P_t and explain their entire evolution from a metrics perspective (Fig. 4b). Trails that do not pass through the selection are drawn gray to provide context; a trail that contains a selected class $\mathbf{e}_i$ is colored, at each point $\mathbf{q}_i^t$, with the color of the top-ranked metric for revision t for the cluster containing $\mathbf{e}_i$ at t. Trails change colors, thus, showing how the most important metrics explaining the similarity of a class at each moment of its evolution changed. In Fig. 4b, for instance, we see that most of the selected trails are orange, which shows *average essential cyclomatic complexity*, see the top-right color legend. Separately, we can also select a revision of interest t, and show the evolution of all classes around this moment (Fig. 4c). For this, we show only trail fragments in the interval $[t-\delta, t+\delta]$, where δ is the user-set size of a window of interest centered at t. Fragments are alpha blended with a Gaussian profile centered at t and vanishing at the window borders, to focus the visualization on the moment of interest, and are colored like described earlier for Fig. 4b. Drawing these trails atop of projection P_t shows the local dynamics of each class, *i.e.*, where it came from, and where it will go next, from the perspective of revision t.

4. Sample applications

We use our MEMs to analyze two open source repositories: JUnit [JUn16] and Google Guice [Goo16b]. For each, we select 100 revisions and extract 43 metrics/class, see Sec. 3.1. To simplify the exposition, we show only classes present in all considered revisions.

4.1. JUnit 4

JUnit is a popular Java testing framework. It allows unit testing by providing helper classes which repeatedly invoke methods from the tested class and compare with expected results. We consider here revisions from March 2010 to March 2016. Data acquisition delivered us 314 classes. Per revision, JUnit has 20.5 KLOC. Figure 5 shows an overview of the project start using small multiples. We see that the overall cluster layout is well kept by dt-SNE, which is expected for a relatively small time-span of 4 revisions. This helps when comparing consecutive maps. Let us now compare the first with the last (100^{th}) revision (Figs.5e,f). While the spatial distribution of clusters strongly changes, which is expected given the 6 year time span, we see that the main clusters of similar classes are given by the *same* metrics – see the similarity of the color legends in Figs. 5e,f in terms of bar colors and bar order. Most classes in both revisions 1 and 100 are similar due to *number of default methods* (yellow) and *average cyclomatic complexity* (purple). This means that, even though *individual* classes do change, there is a strong underlying grouping of classes in terms of aspects captured by the above metrics. We also see, in all Figs. 5a-f, a stable outlier orange group, explained by the metric *average cyclomatic modified*. While we did not dig deeper into the semantics of this class-set, it is clear that these classes stay very different from the rest of the code.

The evolution trails for this dataset show large change (Fig. 5g). The separate bundles show classes which changed their metric values *together*, *i.e.*, evolved as a 'block'. One instance is the bottom horizontal bundle in Fig. 5g which shows a motion to the left. To understand this better, we select one of the end clusters of this bundle and show evolution details (Fig. 5h). The selected group (16 classes) are related mainly by *average cyclomatic complexity* (yellow) in revision 1, and evolved together, being finally mainly similar due to *number of default methods* (purple). Brushing the views shows us that these classes belong to packages *org.junit.internal.runners* and *org.junit.internal.builder*. A stray trail (purple, Fig. 5h) shows a single class diverging from the grouped change. Fig. 5i shows a second group-analysis. The selected classes change together, as told by the bundles, but due to several metrics, as told by the bundle colors: First, this group was similar due to *average cyclomatic codified* (orange), next due to *average line comment* (yellow), and finally due to *average strict cyclomatic complexity* (purple).

4.2. Google Guice

Google Guice is a framework which provides dependency injection for Java objects [Goo16b]. Here, we analyze how 524 classes evolve during 100 revisions between April 2007 and March 2016. The maps of revision 1 and 100 show that most classes share similar values of *average modified cyclomatic complexity* (purple) and *average cyclomatic complexity* (yellow) (Figs. 6a,b). The trails (Fig. 6c) are, in contrast to the JUnit repository (Sec.4.1), much shorter. This tells that Guice has many more stable classes, which do not change much. The color legend in Fig. 6c tells that the values of *average modified cyclomatic complexity* are the most similar for most classes during the analyzed period. The largest group in revision 1, explained by metric *average strict cyclomatic complexity* (purple), splits during the evolution. We use the animation of the projections (Sec. 3.3) to find out that splitting occurs at revision 22. This is also visible in the revision-centric visualization in Fig. 6d. During this process, the splitting bulk of purple classes is joined by a distinct group of yellow classes, explained by *average cyclomatic complexity*. We next focus on what explains the change of class-groups having a high dynamics. We select a group of such classes, *i.e.*, part of long trails (Fig. 6e). Interestingly, this group arrives in the last revision (100) very close to its original position. We want to see which were its intermediate values during its evolution. For this, we analyze a prominent region of blue trail fragments half-way of the group's trajectory. According to

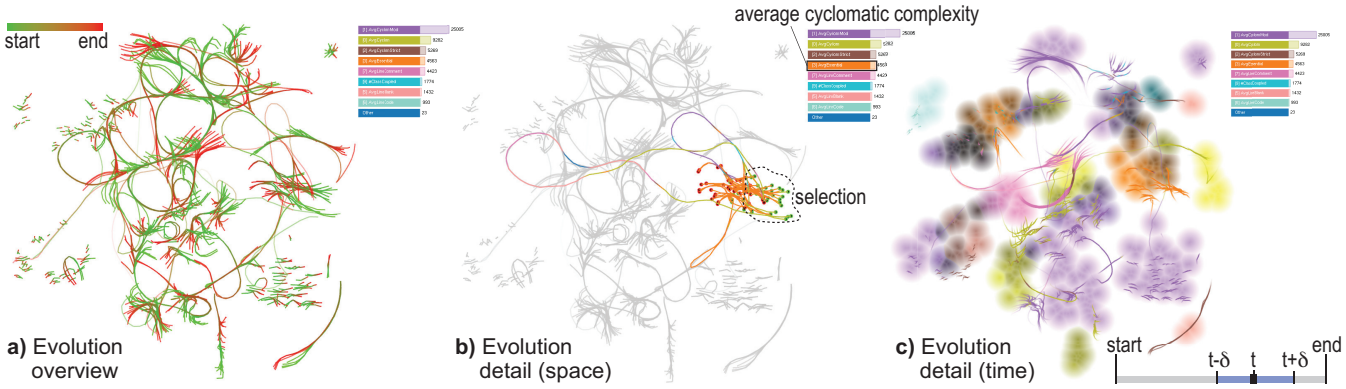

Figure 4: Evolution visualization. (a) Overview showing changes of all classes in all revisions. (b) Evolution of selected classes color-coded by top-ranked metrics. (c) Evolution of all classes around a selected revision.

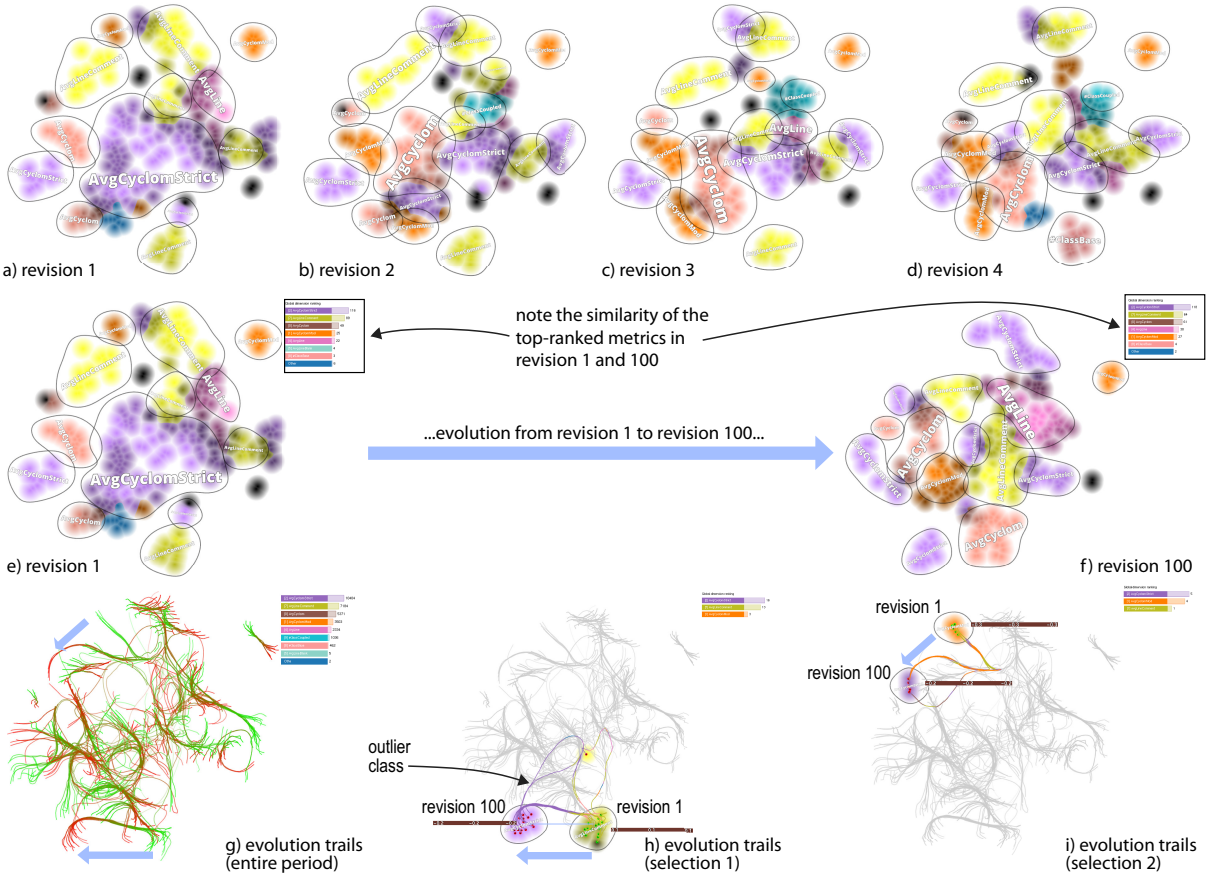

Figure 5: Metric Evolution Maps for four revisions of the JUnit project. See Sec. 4.1.

the color legends, blue can map the metrics *count of coupled classes* or *average number of lines of code*, so we decided to refine the analysis. By showing the windowed trails (see Sec. 3.3), we found that the blue fragments appear around revision 24. Brushing the metric values, we confirmed that the most similar metric values in revision 24 are, indeed, *count of coupled classes* and *average number of lines of code*.

5. Discussion

We next discuss the main aspects of our method.

Advantages: Our method is easy to use; runs in real-time for datasets up to 10K entities on a modern PC with a recent NVidia card for a C++ CPU single-threaded implementation; and is generic, *i.e.*, can be used on any set of entities having n metrics. MEMs explain projections of the data entities in terms of groups of entities that are most similar from the perspective of any of the underlying metrics. Users do not have to *select* which these metrics are – they are determined automatically by the visualization. The found groups are explained both *implicitly*, *i.e.*, as a colored image consisting of several same-color zones; or *explicitly*, *i.e.*, as disjoint clusters.

MEMs extend naturally to time-dependent data by using a dynamic projection technique (dt-SNE). This allows explaining how

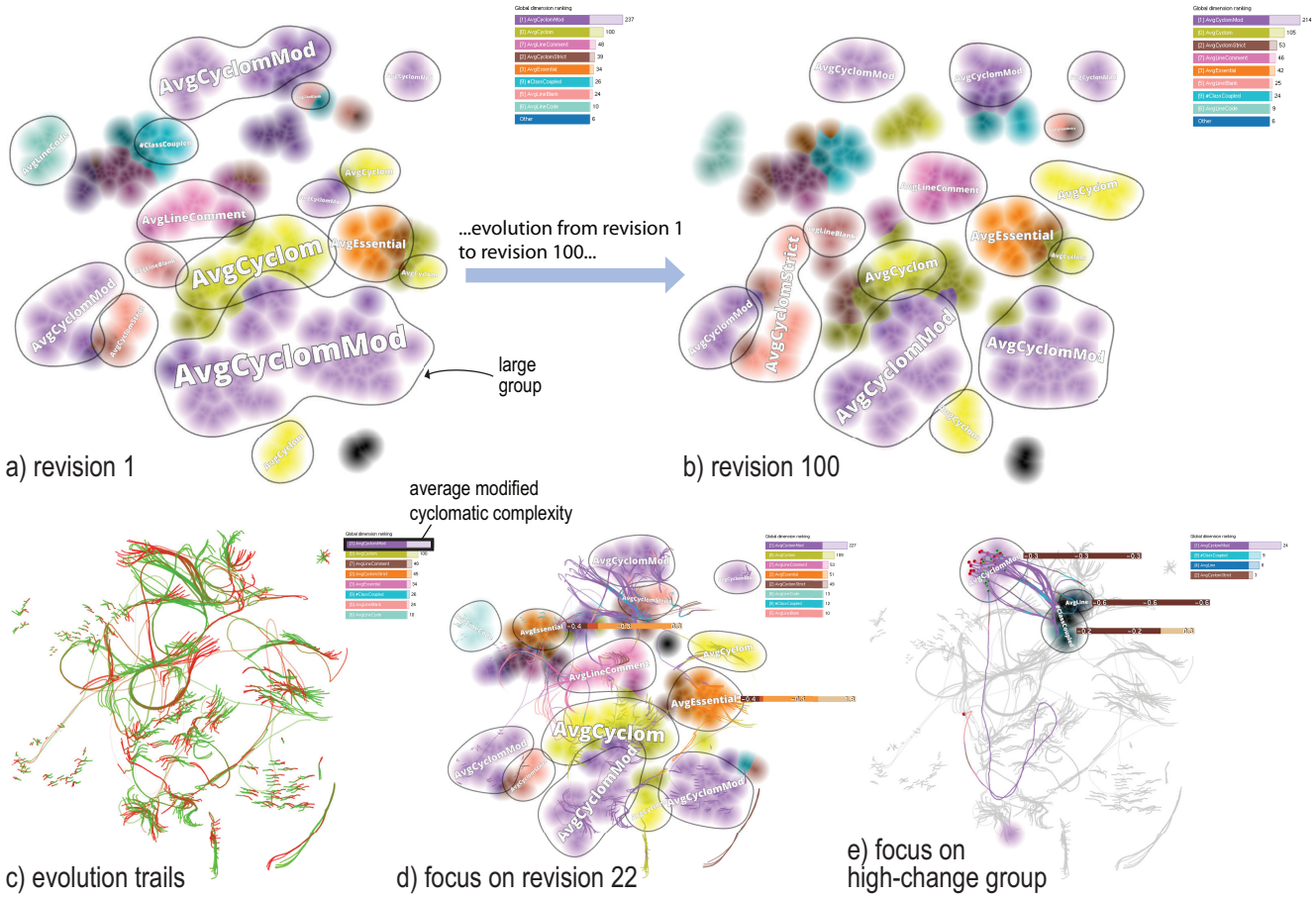

Figure 6: Guice repository. (a,b) First and last revisions. (c) Evolution trails, entire period. (d) Focus on revision 22. See Sec. 4.2.

entities are similar to each other at any time moment (revision), and also how they change across time. The key advantage of using MPs is the high visual scalability thereof: Datasets are essentially reduced to scatterplots, irrespective of the number of underlying dimensions (metrics). Change visualization can be done by animations and small multiples (for examining short time-spans in detail) or bundled trails (for getting simplified clutter-free overviews of long time-spans). Using bundling in combination with time-dependent MPs is, to our knowledge, the first technique that generates compact overviews of large time-dependent high-dimensional datasets.

MEMs have three easy-to-set parameters: ρ acts as a scale factor – large values create fewer color-regions but thicker fuzzy borders, *i.e.* a coarse scale explanation; small values emphasize details but also outliers; ρ_c acts as a filter: large values create smooth regions but thicker borders; small values create noisier regions but thinner borders. α controls the level-of-detail of the shown clusters: small values give more accurate, but less smooth, outlines; large values produce simpler envelopes but remove local details. Besides the above, users can also select entities and/or time-frames for details-on-demand.

Limitations: The similarity-ranking metric (Sec. 3.2) cannot handle well datasets having beyond roughly 40..50 dimensions. Better ranking metrics can be envisaged for 2D neighborhoods, *e.g.* based on feature scoring and discrimination techniques [RdSF*15]. If preferred, any such metric can be trivially added to our MEMs. Secondly, our current visual design is geared towards showing the evolution of entities that exist through the entire time period. However, the arguably main limitation of MEMs relates to their value in assisting software maintenance tasks. While our work shows that MEMs compactly summarize the evolution of large projects and help detecting interesting events and outliers which we could not find by other methods, more validation work is needed to assess their end-to-end effectiveness. Hence, the main contribution of this paper is addressing the hard technical challenge of being able to show the dynamics of large sets of entities, as captured by large sets of quality metrics (answering questions **Q1** and **Q2**). This must be done before we can use such insights to solve concrete application-domain problems.

6. Conclusion

We have presented a set of techniques that allow exploring the evolution of entities in software repositories from a metric-centric perspective. For this, we extend and combine a set of recent multidimensional visualization techniques for both static and time-dependent datasets. Our techniques visually explain groups formed in multidimensional projections, and the evolution in time of (parts of) these groups, in terms of individual entity attributes, such as software metrics. Our techniques include a mix of annotations and segmentations of static 2D projections, bundle-based and metric-based simplified visualizations of the evolution of entities, and interactive mechanisms to provide level-or-detail insight on selected entities and metrics. We demonstrate the technical applicability of the proposed methods on two real-world software repositories.

Future work can address enhancing the proposed visualizations to

show additional metrics, such as speed of change and identity of the changed artifacts, in line with earlier repository exploration methods *cvsgrab*. Secondly, validating the end-to-end added-value of the proposed exploration methods can be done by user studies involving concrete maintenance tasks. Finally, our methods could be applied to the most general challenge of understanding any multidimensional time-dependent dataset.

Acknowledgements: This work is supported the Brazilian research financial agencies FAPESP (grants 2011/18838-5 and 2012/24121-9), CAPES (grant 88888.101004/2015-00) and CNPq (grant 487186/2013-3).

References

[Ant05] ANTONIOL G.: Detecting groups of co-changing files in CVS repositories. In *Proc. IEEE IWPSE* (2005), pp. 23–32. 1

[BD08] BURCH M., DIEHL S.: TimeRadarTrees: Visualizing dynamic compound digraphs. *CGF 27*, 3 (2008), 823–830. 1

[BH16] BREWER C., HARROWER M.: ColorBrewer tool, 2016. http://www.colorbrewer.org. 3, 4

[Cpp16] CPPCHECK: A tool for static c/c++ code analysis, 2016. http://cppcheck.sourceforge.net. 2

[CV01] CHAN T., VESE L.: Active contours without edges. *IEEE Transactions on Image Processing 10*, 2 (2001), 266–277. 4

[dSRM*15] DA SILVA R., RAUBER P., MARTINS R., MINGHIM R., TELEA A.: Attribute-based visual explanation of multidimensional projections. In *Proc. EuroVA* (2015), Eurographics, pp. 37–42. 2, 3, 4

[EKS83] EDELSBRUNNER H., KIRKPATRICK D., SEIDEL R.: On the shape of a set of points in the plane. *IEEE Trans Inf Theor 29*, 4 (1983), 551–559. 4

[Fro16] FRONTENDART: SourceMeter static source code analysis, 2016. http://www.sourcemeter.com. 2

[GJR99] GALL H., JAZAYERI M., RIVA C.: Application of information visualization to the analysis of software release history. In *Proc. VisSym* (1999), Springer, pp. 132–139. 2

[Goo16a] GOOGLE: CodePro AnalytiX, 2016. https://marketplace.eclipse.org/content/codepro-analytix. 2

[Goo16b] GOOGLE GUICE: Java dependency injection framework, 2016. github.com/google/guice. 5

[Han13] HANJALIC A.: ClonEvol: Visualizing software evolution with code clones. In *Proceedings of the First IEEE Working Conference on Software Visualization (VISSOFT)* (2013), IEEE Press, pp. 1–4. 1

[Har75] HARTIGAN J. A.: Printer graphics for clustering. *Journal of Statistical Computation and Simulation 4*, 3 (1975), 187–213. 2

[HvW08] HOLTEN D., VAN WIJK J. J.: Visual comparison of hierarchically organized data. *CGF 27*, 3 (2008), 759–766. 1, 5

[ID90] INSELBERG A., DIMSDALE B.: Parallel coordinates: A tool for visualizing multi-dimensional geometry. In *Proc. IEEE Visualization* (1990), pp. 361–378. 2

[JPC*11] JOIA P., PAULOVICH F., COIMBRA D., CUMINATO J., NONATO L.: Local affine multidimensional projection. *IEEE TVCG 17*, 12 (2011), 2563–2571. 2, 4

[JUn16] JUNIT: JUnit testing tool, 2016. github.com/junit-team/junit4. 5

[KELN10] KUHN A., ERNI D., LORETAN P., NIERSTRASZ O.: Software cartography: Thematic software visualization with consistent layout. *J Softw Maint Evol-R 22*, 3 (Apr. 2010), 191–210. 1, 2, 4

[Lan01] LANZA M.: The evolution matrix: recovering software evolution using software visualization techniques. In *Proc. IWPSE* (2001), ACM, pp. 37–42. 2

[LM06] LANZA M., MARINESCU R.: *Object-Oriented Metrics in Practice*. Springer, 2006. 1

[LWC07] LANGE C., WIJNS M., CHAUDRON M.: MetricViewEvolution: UML-based views for monitoring model evolution and quality. In *Proc. IEEE CSMR* (2007), pp. 327–328. 1, 2

[Mac05] MACDONELL S. G.: Visualization and analysis of software engineering data using self-organizing maps. In *Proc. ESE* (2005), pp. 233–242. 2

[MCMT14] MARTINS R., COIMBRA D., MINGHIM R., TELEA A.: Visual analysis of dimensionality reduction quality for parameterized projections. *Computers & Graphics 41* (2014), 26–42. 3

[MD08] MENS T., DEMEYER S.: *Software Evolution*. Springer, 2008. 1

[MMMW05] MARINESCU C., MARINESCU R., MIHANCEA P. F., WETTEL R.: iPlasma: An integrated platform for quality assessment of object-oriented design. In *Proc. IEEE ICSM* (2005), pp. 77–80. 2

[MSM*10] MEIRELLES P., SANTOS C., MIRANDA J., KON F., TERCEIRO A., CHAVEZ C.: A study of the relationships between source code metrics and attractiveness in free software projects. In *Proc. Brazilian Symp. on Software Engineering* (2010), pp. 11–20. 4

[NB12] NOCAJ A., BRANDES U.: Organizing search results with a reference map. *IEEE TVCG 18*, 12 (2012), 2546–2555. 3

[OCT*13] OLIVEIRA G., COMBA J., TORCHELSEN R., PADILHA M., SILVA C.: Visualizing running races through the multivariate time-series of multiple runners. In *Proc. IEEE SIBGRAPI* (2013), pp. 99–106. 4

[PNML08] PAULOVICH F. V., NONATO L. G., MINGHIM R., LEVKOWITZ H.: Least square projection: A fast high-precision multidimensional projection technique and its application to document mapping. *IEEE TVCG 14*, 3 (2008), 564–575. 2

[PTT*12] PAULOVICH F. V., TOLEDO F. M. B., TELLES G. P., MINGHIM R., NONATO L. G.: Semantic wordification of document collections. *CGF 31*, 3 (2012), 1145–1153. 4

[RCK08] ROY C., CORDY J., KOSCHKE R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci Comp Prog 74*, 7 (2008), 470–495. 1

[RdSF*15] RAUBER P., DA SILVA R., FERINGA S., CELEBI M., FALCÃO A., TELEA A.: Interactive image feature selection aided by dimensionality reduction. In *Proc. EuroVA* (2015), Eurographics, pp. 54–61. 7

[RFT16] RAUBER P., FALCAO A., TELEA A.: Visualizing time-dependent data using dynamic t-SNE. In *Proc. EuroVis Short Papers* (2016). 2, 5

[RPP02] REFORMAT M., PEDRYCZ W., PIZZI N. J.: Software quality analysis with the use of computational intelligence. In *Proc. FUZZ-IEEE* (2002), vol. 2, pp. 1156–1161. 2

[RVET14] RENIERS D., VOINEA L., ERSOY O., TELEA A.: The Solid* toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product. *Sci Comp Prog 79* (2014), 224–240. 2

[Sci16] SCITOOLS: Understand tool, 2016. https://scitools.com. 1, 2

[Son16] SONARSOURCE: Sonarqube tool, 2016. www.sonarqube.org. 2

[STSS05] STRICKERT M., TEICHMANN S., SREENIVASULU N., SEIFFERT U.: High-throughput multi-dimensional scaling (hit-mds) for cDNA-array expression data. In *Proc. ICANN* (2005), vol. 3696, Springer, pp. 625–633. 2

[TA08] TELEA A., AUBER D.: Code flows: Visualizing structural evolution of source code. *CGF 27*, 3 (2008), 831–838. 1

[TCM*10] TERCEIRO A., COSTA J., MIRANDA J., MEIRELLES P., RIOS L., ALMEIDA L., CHAVEZ C., KON F.: Analizo: an extensible multi-language source code analysis and visualization toolkit. In *Proc. CBSoft* (Salvador-Brazil, 2010). 2

[TdSL00] TENENBAUM J. B., DE SILVA V., LANGFORD J. C.: A global geometric framework for nonlinear dimensionality reduction. *Science 290*, 5500 (2000), 2319. 2

[Tim16] TIM LITTLEFAIR: CCCC - C and C++ code counter, 2016. http://cccc.sourceforge.net. 2

[TMR02] TELEA A., MACCARI A., RIVA C.: An open toolkit for prototyping reverse engineering visualizations. In *Proc. VisSym* (2002), Springer, pp. 241–249. 2

[Tuf06] TUFTE E.: *Beautiful evidence*. Graphics Press, 2006. 2

[vdMH08] VAN DER MAATEN L., HINTON G. E.: Visualizing high-dimensional data using t-SNE. *J Mach Learn Res 9* (2008), 2579–2605. 2

[vdZCT16] VAN DER ZWAN M., CODREANU V., TELEA A.: CUBu: Universal real-time bundling for large graphs. *IEEE TVCG* (2016). doi:10.1109/TVCG.2016.2515611. 5

[VT06] VOINEA L., TELEA A.: CVSgrab: Mining the history of large software projects. In *Proc. EuroVis* (2006), IEEE, pp. 187–194. 1, 2

[VTvW05] VOINEA L., TELEA A., VAN WIJK J. J.: Cvsscan: Visualization of code evolution. In *Proc. ACM SoftVis* (2005), pp. 47–56. 1, 2