AniMatrix: A Matrix-Based Visualization of Software Evolution

Sébastien Rufiange and Guy Melançon
University of Bordeaux, LaBRI, UMR 5800
INRIA Bordeaux Sud-Ouest, France
Email: sebastien@rufiange.com, guy.melancon@labri.fr

***Abstract*—Software designs are ever changing to adapt to new environments and requirements. Tracking and understanding changes in modules and relationships in a software project is difficult, but even more so when the software goes through several types of changes. The typical complexity and size of software also makes it harder to grasp software evolution patterns. In this paper, we present an interactive matrix-based visualization technique that, combined with animation, depicts how software designs evolve. For example, it shows which new modules and couplings are added and removed over time. Our generic visualization supports dynamic and weighted digraphs and is applied in the context of software evolution. Analyzing source code changes is important to determine the software's structural organization and identify quality issues over time. To demonstrate our approach, we explore open-source repositories and discuss some of our findings regarding these evolving software designs.**

I. Introduction

Information is a critical element of our modern society, and its value is of great importance as its analysis may help discover hidden trends, unexpected relationships, properties or patterns in data. Data often need to be understood from multiple perspectives and may have many forms, e.g., they can be very structured or semi-structured, hierarchical or multi-dimensional. Due to size and complexity, understanding even just a part of this data is often a challenging task, and mistakes may lead to taking decisions affecting people's lives.

Networks (graphs) are a commonly used data structure that can model several real-world systems, such as financial transactions and social interactions. However, these networks are complex and difficult to grasp, even more so as they change over time, and there are growing needs in understanding static and evolving networks. Common techniques to visualize such networks include Small Multiples [1], animation [2] and glyphs [3].

Visualizing dynamic networks is useful in many domains, e.g., to track financial transactions, the forming of social communities or the evolution of software. In particular, software has to change continuously over time or it becomes less useful. Despite advances in understanding the evolution of software, its visualization may provide software engineers and researchers with more insight about dynamic processes that affect modules in software, and the propagation of changes. Software engineers still need tools to help answer important questions such as how did the couplings evolve following the introduction of a pattern? How were the modules restructured during a refactoring phase?

There can be various types of changes affecting software, but complex changes are difficult to describe and may be insufficiently documented in repository logs. Managing and tracking the evolution of these changes is important to achieving good designs and reducing high maintenance costs, and thus more software evolution tools are needed [4].

Current approaches for visualizing evolving software typically use node-link diagrams, which can have cluttering issues. To help address this problem, the visualization that we present in this paper offers an interactive way of exploring evolving software, is scalable and also supports weighted digraphs with multiple types of nodes (multi-modal) and edges (multi-relational). Our contributions are (1) a taxonomy of visualization techniques for dynamic networks, (2) an interactive prototype (AniMatrix) that allows exploring the change histories of software, and (3) a demonstration of our approach on three open-source software repositories.

II. Related Work

Various techniques of visualizing dynamic graphs and changing software have been explored and are discussed in this section.

A. Dynamic network visualization

Small multiples [1] can be adapted to work for any visualization technique to show changes in a graph, by juxtaposing thumbnail images that represent the graph at some point in time. However, they take up more space, making it difficult to track changes over multiple time steps. Difference maps [5], on the contrary, use highlighting to show differences between two time steps, reducing the effort to track changes. They also allow hiding intermediate events, which can be useful if these events are not critical or can be discarded. However, in the case of exploring unknown data, multiple difference maps may be needed to show the complete evolution steps.

Animation also typically uses highlighting, but the user may need to drag back and forth to properly understand the underlying data, which can take time. On the one hand, animation takes time to play, which may affect the performance of tasks. On the other hand, animation can help to reduce error rates [6]. Indeed, objects may move due to changes over time and animation may help in tracking them. Algorithms have been designed to try to facilitate the tracking of nodes over time,

but results obtained so far have not yielded simple conclusions about the effects of preserving the mental map [7].

Static representations were found to be more suitable in some cases (e.g., [6], [8]), although animation was also shown to have benefits (e.g., [9], [10]). Moreover, the integration of animation in a hybrid visualization resulted in increased performance in exploration tasks of dynamic networks [11].

Node-link diagrams have limitations when compared to matrices. Generally, area and color can help to track evolution in representations [12]. However, the tracking of changes may be easier in matrix views, as the area allocated to showing edges (matrix cells) is larger than in node-link views. Although node-link diagrams may facilitate the finding of paths [13], matrices are generally more scalable.

Glyphs (i.e. charts or a representation showing small information summaries) have also been used (e.g., [3], [14]). These glyphs can be used in combination with other visualizations (such as node-link diagrams) to show the evolution of dimensions. However, there is limited space where edges are and so only a few time steps can be shown there as glyphs. Moreover, existing techniques do not focus on showing the evolution of multiple types of edges.

B. Software visualization

Visualization techniques may help developers gain insight about static (at one version of the software), dynamic (at run-time) or evolutionary (over several versions) aspects of source code [15]. Some static visualization approaches may be extended to visualize changing aspects as well. Lattix [16] uses a matrix-based representation to visualize and interact with the layered structure of source code, in addition to showing module dependencies. Package Fingerprints [17] similarly allow visualizing incoming and outgoing couplings to encapsulated packages using a matrix-inspired approach. TreeMatrix [18] combines node-link and matrices to depict high level and low level structures of source code, along with their relationships. Abuthawabeh *et al.* (2013) [19] found that both matrix and parallel node-link visualizations were similarly performant in showing multiple types of couplings. Although these approaches are useful to show the structure of the design at some points in time, they do not focus on representing how the design evolves over time.

Collberg *et al.* (2003) [20] proposed one of the first evolving software visualization based on multiple node-link views. YARN [21] can represent evolving couplings using animated node-link diagrams, but only supports high level abstractions (packages). Langelier *et al.* (2008) [22] uses an animated 3D model to show the evolving hierarchical structure of software, as well as changes in terms of metrics. IHVis [23] allows visualizing the evolution of design structures using node-link diagrams, but may have scalability issues and cannot show the evolution of multiple types of couplings.

In summary, current visualizations focus on various aspects, but in spite of their potential benefits, matrix-based visualizations are still not adapted to show evolving software. In the next section, we explore possibilities of visualizing such changing structures.

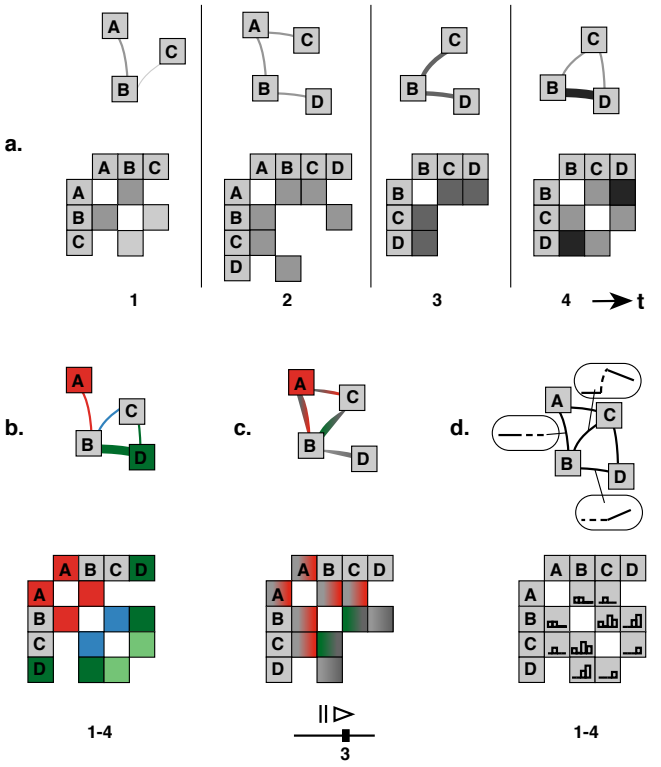

Fig. 1. A taxonomy of visualization techniques for dynamic graphs. The same dynamic graph is shown using four techniques, and each approach is depicted using both node-link and matrix representations. The weights of the edges correspond to the lines' thickness (node-link view) or the color darkness (matrix view). When the differences are highlighted (difference or animation views), the color indicates the type of change, either removal (red), addition (green), modified (blue) or unchanged (light gray). **a. Small Multiples.** A small representation is used at each of the four time steps. (Note that the time slices could also be shown on top of each other in 3D.) **b. Difference Maps.** The changes between time steps 1 and 4 are depicted using colors. **c. Animation.** A user may use a dragging gesture (or a scroll bar) to navigate in time. In this case, the changes between time steps 2 and 3 (i.e. one transition) are interpolated. **d. Glyphs.** Charts, time series or another graphical representation can be used to show small summaries of the evolution of edges.

III. Taxonomy

There are different visualization techniques that can be used to show evolving networks. In this section, we present a taxonomy of dynamic graph visualizations. We used this classification to organize visualization possibilities (a similar strategy has been used in, e.g., [11], [24], [25]). In contrast to previous work, we discuss techniques to visualize evolving networks having multiple types of nodes and edges, focusing on both node-link and matrix views.

Figure 1 illustrates the main approaches that can be used to visualize the same dynamic graph. The techniques are generic and can be used to visualize any weighted dynamic digraph, although undirected graphs are depicted to simplify the illustrations. Nodes in node-link diagrams may also be laid out in different ways, including radial layouts [26] or parallel vertical axes.

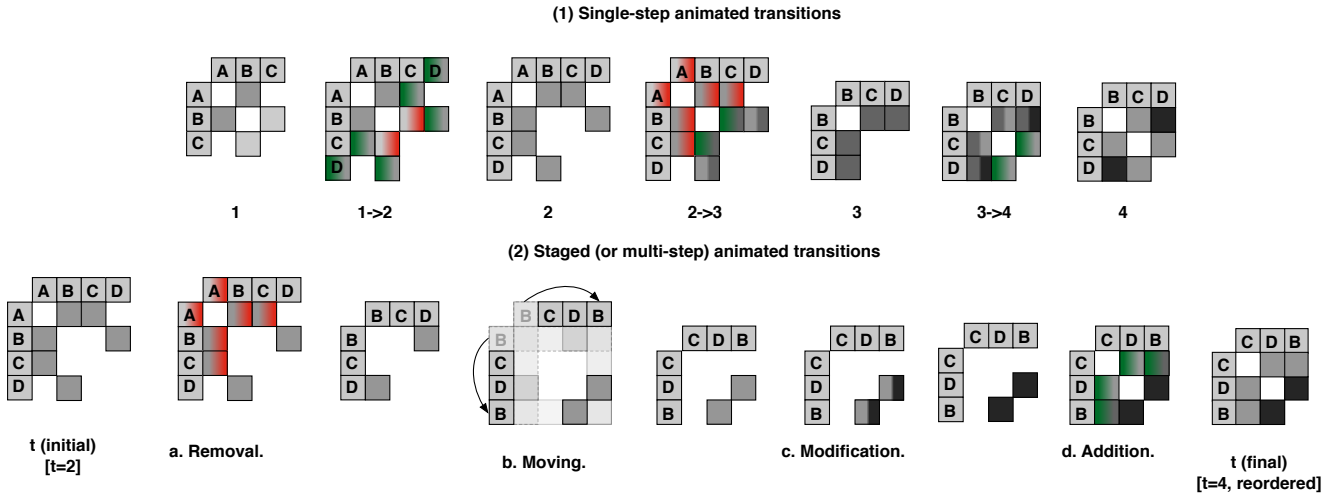

Fig. 2. Smooth transitions showing graph changes between time steps 2 to 4 in a matrix-based visualization using a single-step animation (1) or a staged (multi-step) animation (2). Note that the network shown here is taken from the taxonomy (Figure 1) and animated changes occur in a left-to-right order. In the first case, three transitions are depicted (i.e. 1-2; 2-3; 3-4) and all types of changes are aggregated and shown at the same time. In the second case, a staged animation is used to gradually show the evolution of multiple types of changes. The animation steps occur in the order detailed below. **a. Removal step.** Nodes and edges fade out from their current color to red. **b. Moving step.** Elements move to their new positions (in case of reordering). **c. Modification step.** Elements are modified (e.g., in terms of edge weights) and a cell becomes darker (stronger link) or lighter (weaker link) over time. **d. Addition step.** Added nodes and edges gradually appear, fading in from green to their target color.

Small multiples (1.a) can support a wide range of techniques and layouts, but require the user to manually compare the visualizations at different time steps, and take up more space. In contrast, difference maps (1.b) highlight the changes between two time steps. However, they may also hide potentially critical intermediate states. Animations (1.c) have shown to be useful in facilitating the understanding of changes and are associated with lower error rates, but may be slower and the user often need to manually control the animation speed. Glyphs (1.d) can show the evolution of edges by drawing small graphics summarizing the changes. However, the available space in such visualizations is limited, making it difficult to draw glyphs inside edges of node-link diagrams, for example.

In software engineering, files would generally be depicted as nodes, and edges would represent the dependencies. A software network may have several types of nodes (e.g., classes, interfaces) and edges (e.g., inheritance, method invocations). However, not all visualization techniques may support showing the evolution of multiple types of edges, although some could be extended to support these types of networks. For instance, to visualize an unweighted graph with multiple types of edges, one way would be to map colors to edge types instead of edge weights. However, in the case of a weighted graph with multiple types of edges, another visual encoding must be used, and one that is non-conflicting. Extended glyphs, such as stacked bar charts, multiple line charts, spider webs or even parallel coordinates could be used in place of classical bar charts to show the evolution of multiple types of edges. The combination of animation and bar charts, for example, could achieve similar objectives (i.e. animating the glyphs).

Moreover, there might be many types of changes to animate simultaneously, and there are different approaches, as illustrated in Figure 2. However, node positions may change over time, making it harder to track moving objects. To avoid this potential issue, mental map preservation algorithms aim at minimizing node movements over time, because on the one hand, too many movements make it harder to track nodes and on the other hand, very few movement may not help in revealing related nodes and clusters. Indeed, if reordering is not computed at all (or insufficiently), a software module could appear as part of a cluster while it is no longer the case. However, possibly because of this trade-off, it is unclear whether and how mental map preservation algorithms should be used [7].

Staged animations can also help in tracking several types of changes [27]. In some cases, understanding the full evolution of the data may not be needed, and so simply interpolating between two time steps (thus hiding intermediate steps) could be sufficient. On the contrary, if a finer level of details is required to better understand evolutionary aspects, multiple transitions steps can be used instead. In this case, the evolution timeline could be first divided by the number of time steps, and then further divided for each animation step.

Based on these possibilities, to help users interactively explore large software and change histories, we implemented a matrix-based visualization. Our prototype allows exploring software by animating its matrix representation in a flexible manner. In the next section, we describe our visual approach to explore evolving software.

IV. Software Evolution Dimensions

Software is developed using various languages and technologies, and designs can be difficult to assess as several

patterns and structures may affect how changes propagate over time. AniMatrix aims to help answering questions such as:

- At which points in time is the design more (or less) stable?
- When are new design elements inserted and do they evolve in a certain way?
- Were there any minor or major reorganizations?

In the remaining of this section, we describe dimensions concerning evolving software, along with the corresponding visual cues that could be observed in an animated matrix-based visualization (such as AniMatrix). We focus on observations that can help understand relationships, structures, confirm or suggest hypotheses, and that may lead to insight [28].

A. Usages

Coupling is widely recognized as a key measure to assess the quality of software designs, and a number of metrics have been proposed in the literature. A design in which there are many highly coupled modules tend to be rigid and harder to reuse and maintain. Dependencies of various types may remain constant, increase, decrease or fluctuate (i.e. increase and decrease) over time. In the following subsections, we present observations that could be found in software designs, based on such dependencies.

1) Central Element: Central elements have relatively strong incoming couplings from several other classes. Over time, it can lead to a Shotgun Surgery anti-pattern, i.e. one change made to the central element could affect several other elements. Indeed, if incoming couplings of an element are constantly increasing, it could indicate a growing need to restructure it. In our visualization, these central elements tend to have multiple filled (and dark-colored) cells in their corresponding columns. Moreover, the addition of edges are shown as fading in, while removed edges fade out.

2) Lazy Element: A *Lazy Element* is the opposite of a *Central Element* in that a heavily dependent element (as in a *Lazy Class*) has many outgoing links. Thus, the corresponding matrix rows would be gradually filled with multiple (dark-colored) cells.

3) Reused Abstraction: Software abstractions are created, but may not be used for some time. These unused generalizations can introduce unnecessary complexity, making it harder to refactor or find and fix problems. On the other hand, abstractions that are reused over time are potentially valuable investments. In a matrix view, rarely used abstractions would contain several empty cells in their corresponding columns.

4) Related Elements: Two modules which are increasingly (or decreasingly) dependent on each other, will develop stronger (or weaker) links. Visually, the corresponding cells would get darker over time to indicate stronger relationships. Also, related modules will have a tendency to move closer to each other as if they were part of the same group (or cluster), while unrelated modules will move apart. Such groups of modules should generally be part of the same unit of development (e.g., to facilitate testing). In some cases, cycles may appear, i.e. two elements may be dependent on each other at some point. These elements would appear on each side of the matrix diagonal.

B. Design Stability

Several design patterns aim to enhance the stability of a design, so that changes are localized and have small impact on unrelated classes to reduce modification costs. Furthermore, the increase in code smells [29] can be a sign of a degrading design. A visual tool could help to check software evolution patterns and prevent issues. Figure 3 illustrates how these stability aspects would appear in a matrix-based view.

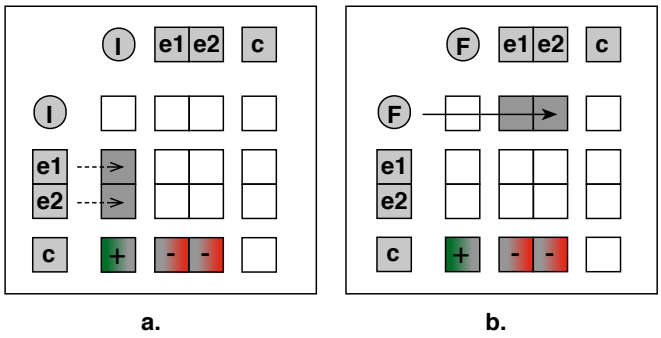

Fig. 3. Visualization of stability aspects of an evolving design. **a. Protected Variations.** An application of *Protected Variations* is typically composed of three parts. An interface *I* is first introduced at some point in time. Then, elements (*e1* and *e2*) implement interface *I*. Finally, a client class *c* relies on the stable interface *I*, and couplings (if any) to the internal elements are being removed. **b. Encapsulation.** Similarly, an intermediary class *F* (such as a Facade) may isolate unstable elements from external clients to preserve the design's stability. At some point in time, a class *F* encapsulates some elements (*e1* and *e2*). A new client class *c* may then no longer need to refer to these encapsulated elements and thus depend on *F* instead.

1) Interface Couplings: Increasing couplings to interfaces (stable couplings) could indicate a shift to a more stable design structure, while decreasing stable couplings may suggest design instabilities. For instance, following the introduction of a new interface design pattern, client classes will often refer to the interface shortly after its introduction to benefit from it. In this case, the column associated with the interface would become increasingly filled with cells.

2) Protected Variations: Increasing couplings to concrete implementations of interfaces (i.e. clients directly referring to classes that implement interfaces or extended interfaces) are a sign of increased instability, as the clients rely on unstable parts. *Protected Variations* [30], as implemented in design patterns (e.g., Strategy, Iterator), hide unstable elements from clients to protect them from changes using a stable interface. Hence, clients should only refer to that interface, to favor stable couplings and to avoid unstable ones (Figure 3.a).

3) Encapsulation: As per information hiding, the introduction of an intermediary class (such as a Facade) may help protect external clients from changes in encapsulated elements (Figure 3.b). Clients relying on a Facade should thus have reduced couplings toward the encapsulated elements, as the clients can refer to the intermediate element instead. In particular, if the outgoing couplings of the intermediary class

are creational ones, this would suggest a behavior similar to that of a Factory (essentially hiding the construction logic).

C. Restructurings

Major or sudden fluctuations (i.e. several changes made during the same period of time) may be caused by significant design changes or restructurings. A lot of changes could also be caused by a hierarchical reorganization of the design (such as changing packages), when unused parts are removed (such as dead code) or after code cleanings.

1) Big Bang: Often at the beginning of a project but also at other stages, multiple types of changes can occur simultaneously during a short period of time. These changes usually pop up visually as several elements are added and modified. This could be caused, for example, by structural changes (as in package reorganizations) or by committing a significant piece of work just before a release.

2) Massive Abstractions: Sudden insertions (or removals) of multiple interfaces or abstract classes may suggest that the design is being reworked, e.g., design patterns could be gradually incorporated. In particular, the insertion (or removal) of abstract classes is often associated with increases (or decreases) in reusability, as it is a key role played by these types of software elements. Similarly, increases (or changes) in couplings might influence reusability aspects. These can be spotted relatively quickly by specifically watching changes in abstract classes and extension couplings.

3) Renaming: Classes might be renamed at some point or be moved to another package, as responsabilities are reassigned. Since their namespaces would change, the elements would first appear to be removed, then re-added shortly after in a second step, possibly at different locations. The renamed elements might also be refactored at the same time.

4) Code Cleaning: Code cleaning and removal of dead code may happen just before a release or during refactorings, and suggest a gradual stabilization of the design. Some matrix cells and elements are typically removed during a cleaning phase and are not added back (these fading out elements naturally pop out).

The observations described in this section are organized in Table I, which may further help in analyzing software evolution dimensions [1]. Our tool shows the history of changes and the user can time travel and focus on specific aspects (e.g., added interfaces, node movements). Further investigations can reveal the reasons for these modifications and their effects on the design.

V. Visualization

Software designs can naturally be modelled using multi-modal and multi-relational networks. As shown in our taxonomy, an approach to visualize the evolution of software would be to juxtapose small multiples, i.e. one for each version of the software, as in [20]. However, in this case, only a limited number of snapshots can be shown on the screen and additional interaction techniques are required to browse large histories. Some types of changes are also harder to understand in such representations, as the user needs to manually compare versions. Animated node-link diagrams were used in previous work (e.g., [11], [21], [31]) but such approaches may have cluttering issues. In contrast, matrix-based visualizations have less cluttering, are scalable and support dense networks (as demonstrated in, e.g., [13], [32]). An overview of our interactive prototype is presented in Figure 4. AniMatrix allows watching software change histories using a matrix-based visualization and multi-step animations. Thus, it allocates space to show connections among software modules, making it easier to track, e.g., changes in couplings over time. To facilitate the understanding of changes, interaction techniques, such as filtering and zooming are also supported.

A. Interaction

We implemented interaction techniques to support a dynamic analysis of software [2]. A history navigator is initially shown to the user (Figure 5). This interactive timeline covers all commits made by the developers over time. The user can zoom in or out using the mouse wheel and pan around by dragging the mouse or by using the scroll bar slider. Hovering over a revision shows more details, such as the full date of the commit, author, and log message.

Between two consecutives commits, a multi-step animated transition is created (as previously described in Figure 2). The user can move in time (backward or forward) inside that multi-step animation by moving the mouse cursor or by pressing the left and right keys. The current step of the animation (e.g., addition of nodes and edges, movement of nodes, removal of nodes and edges) is highlighted in the history navigator. At the same time, the matrix view is updated as the user moves around to highlight the addition of elements (increasing size, in green color), the removal of elements (decreasing size, in red color) or node movements (in orange). Changes with respect to edges are also shown. In this case, the borders of the matrix cells are colored in blue when they are changing, and the cells are gradually colored in varying tones of gray (using darker grays if weights are greater). The user can optionally see the counts (i.e. the weights) for each edge type (in the detailed view) or the total count for all types (i.e. the aggregated value).

Nodes can also be of several types (classes, abstract classes or interfaces) as shown in Figure 6 [3]. Hovering, selecting and viewing source code are supported. Finally, the user may filter or hide elements, by searching for their names or filtering by their types. Whenever one or several filters are applied (e.g., all interfaces named *Shape* or implementation couplings), only the corresponding changes in the animation steps are shown in the history navigator. So, a user may quickly see where a

[1] This categorization table illustrates how evolving software designs may be explored visually but could be expanded or tailored to other contexts.

[2] A video showing the tool and the interaction possibilities is available online at http://ref.rufiange.com/animatrix2014.

[3] There are reasons that motivated our choices. Rectangles and ellipses are also commonly used in UML to represent classes and interfaces. Abstract classes and interfaces are drawn using dashed borders, as they cannot be instantiated, in contrast to classes (which are also shown in darker color).

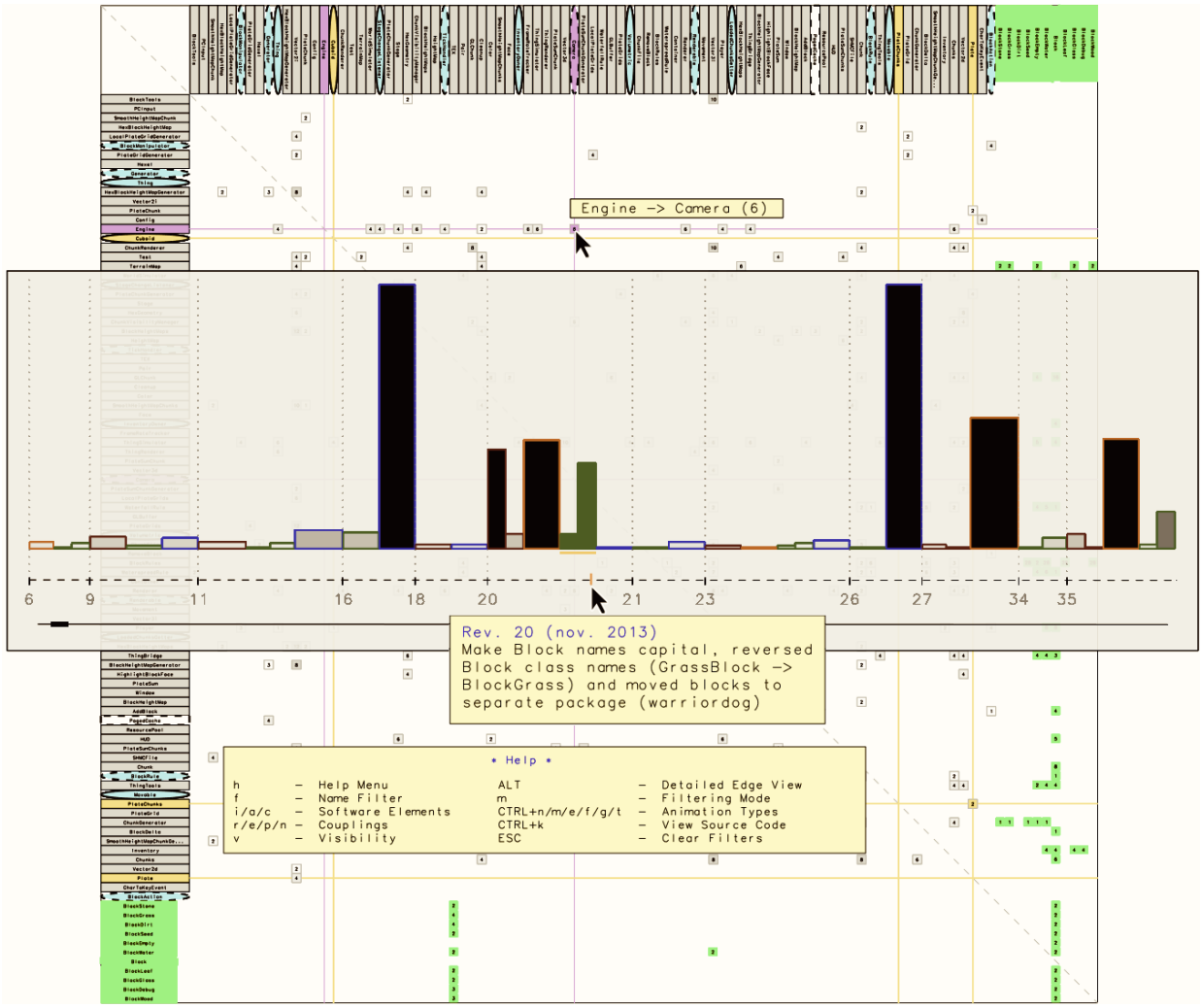

Fig. 4. Overview of our AniMatrix prototype, comprising an interactive and dynamic matrix view, and a history navigator. The implemented controls enable the user to move back and forth in time and in space, as well as search and filter elements. Zooming and panning is supported in the matrix and navigator views. In this screenshot, a matrix cell is hovered (shown in purple, along with a tooltip), and some elements are selected (shown in orange).

TABLE I
OBSERVATIONS CONCERNING EVOLVING SOFTWARE DESIGNS AND VISUAL REPRESENTATIONS.

Category	Observation	Visual pattern
Usages	Central Element	-Matching column (vertical direction) is increasingly filled with cells.
	Lazy Element	-Matching row (horizontal direction) is increasingly filled with cells.
	Reused Abstraction	-An abstraction with many incoming couplings (i.e. a column with several non-empty and dark-colored cells) suggests a reused abstraction. In contrast, rarely used software abstractions only have a few non-empty (or light-colored) cells.
	Related Elements	-Related elements tend to get closer to each other over time, while the distance between unrelated elements increases. Furthermore, elements placed on each side of the matrix diagonal are generally interdependent.
Design Stability	Interface Couplings	-Increasing couplings toward interfaces or extensions of interfaces are signs of increased stability (e.g., corresponding columns are increasingly filled with cells).
	Protected Variations	-Clients increasingly rely on interfaces and not on internal implementations. Thus, couplings toward concrete implementations decrease (Figure 3.a).
	Encapsulation	-An intermediary class has couplings toward encapsulated elements. Client classes refer to the intermediate element, and their couplings toward the encapsulated elements decrease (Figure 3.b).
Restructurings	Big Bang	-Many changes occur during a small time span (e.g., classes and links are added and removed during a reorganization).
	Massive Abstractions	-Several insertions of abstract classes, interfaces or extension couplings in a short amount of time.
	Renaming	-The same elements are removed then re-added shortly after.
	Code Cleaning	-Many unused elements (i.e. with several empty cells at their matching columns) are definitely removed at some point.

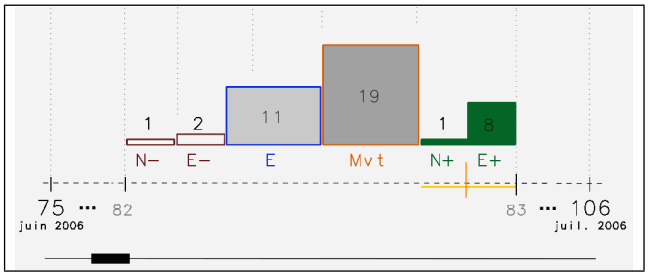

Fig. 5. History Navigator. The changes between revisions 82 and 83 are grouped by types and shown with distinct colors: removal of nodes and edges (N-, E-), node movements (Mvt), addition of nodes and edges (N+, E+) and changes in edges' weights (E). The interactive orange line indicates the current position in time and also the active animation step. The displayed changes take under account any filters that could have been applied. Also, the matched elements in the active animation step are gradually animated and highlighted in orange in the matrix view to make them easier to track.

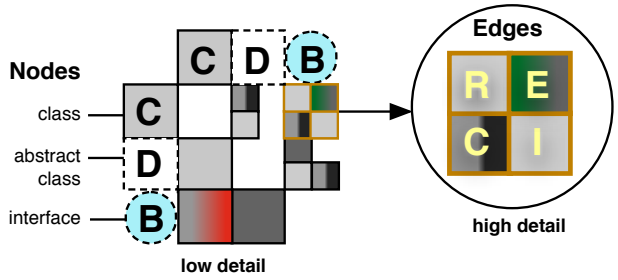

Fig. 6. Staged animations used in our matrix-based visualization can show changes concerning multiple types of nodes and edges. Node types include classes (gray rectangles), abstract classes (white rectangles with dashed borders) and interfaces (cyan ellipses with dashed borders). Edge types represent couplings among these software elements and include references (importations and declarations) (R), extensions (E), creations (constructor calls) (C) and implementations (I).

class is changed over time or when specific types of couplings are created. A user may find pattern occurrences by searching for their names, and can then further check how the instances evolve over time.

B. Implementation

We implemented our prototype in Java, using the Java OpenGL wrapper library. Dynamic networks modelling the evolution of these software projects were extracted from source code repositories using our framework (IHVis) [23]. As software elements are increasingly interconnected, cliques may appear at some point as a result of the reordering of the matrix. Also, there is a trade-off between the degree of generated movement on the visualization and the potential visibility of such cliques. So, we let the user decide when she/he wants to perform reordering. There are several matrix reordering algorithms [33] but no one is clearly better in the context of evolving software. Thus, we used a barycenter algorithm that iteratively places interconnected nodes near each other (as described in [34]). Furthermore, the color scales used in our visualization are interpolated based on ColorBrewer [35].

VI. Software Exploration

In this section, we present some of our findings, discovered using AniMatrix on three Java software projects (Table II). Hexel (http://github.com/es92/Hexel) is a voxel-based game allowing players to explore a 3D world. EasyNote (http://github.com/rui1989/EasyNote) is a note taking application. Finally, Buddi (http://buddi.digitalcave.ca) is a personal finance software.

TABLE II
Projects that were explored using AniMatrix.

Project	# Commits	# Files	Size	Dates
EasyNote	131	141	1.6 MB	2013-2014
Hexel	134	153	2.3 MB	2013-2014
Buddi	1273	290	1.6 MB	2006-2013

A. EasyNote

1) Usages: At the beginning of the project, *MainPanel* is the sole *Lazy Element* and thus interacts with several other classes to display and process user input. Couplings are increasing over time, e.g., at revisions 21 and 56 (Figure 7). As shown in Figure 8.a, a few central elements are also visible (e.g., *Property*, *SoundFactory*, *SoundTheme*) and remain central as the project evolve. There are only a few abstract classes and they are not used by several elements (such as *AbstractNoteDAO*, *AbstractDocument*). In fact, at revision 103, *AbstractNoteDAO* is refactored and is no longer abstract ("Change AbstractNoteDAO to DocumentNoteDAO and make it concrete"), possibly because this part of the design was unnecessary abstract and not reused a lot. As can be expected, unit tests are related to the tested classes (e.g., *WorksheetUnitTests ->Worksheet*).

2) Design Stability: Despite its name, *SoundFactory* (previously found to be a central element) does not focus on creating complex objects, but rather facilitates the access to sound resources. We discovered this by comparing this visual occurrence to the theoretical case illustrated in Figure 3.b, and also by checking the source code. In particular, *SoundFactory* encapsulates the *Property* class that is used by several other clients. Classes rely on *Property* as it is generally stable and thus can be reused without causing too much instabilities.

There are also Data Access Objects (DAO) realizing the *AbstractNoteDAO* abstraction (e.g., *ArticleNoteDAO*) that may prevent client classes from knowing how elements are actually stored in the database. However, as we observed, couplings toward concrete DAO classes are present and sometimes increase (e.g., at revision 19), which might cause issues at some point. The use of an intermediary class (such as a Factory) could have been an option to reducing the number of unstable couplings to concrete DAO classes.

A few interfaces are initially present (e.g., *Note*, *Document*) and some others are added between revisions 57 and 65 (e.g., *AuthorsAware*, *CreatedTimeAware*, *LastUpdatedTimeAware*), as new functionality regarding worksheet data is added ("Add

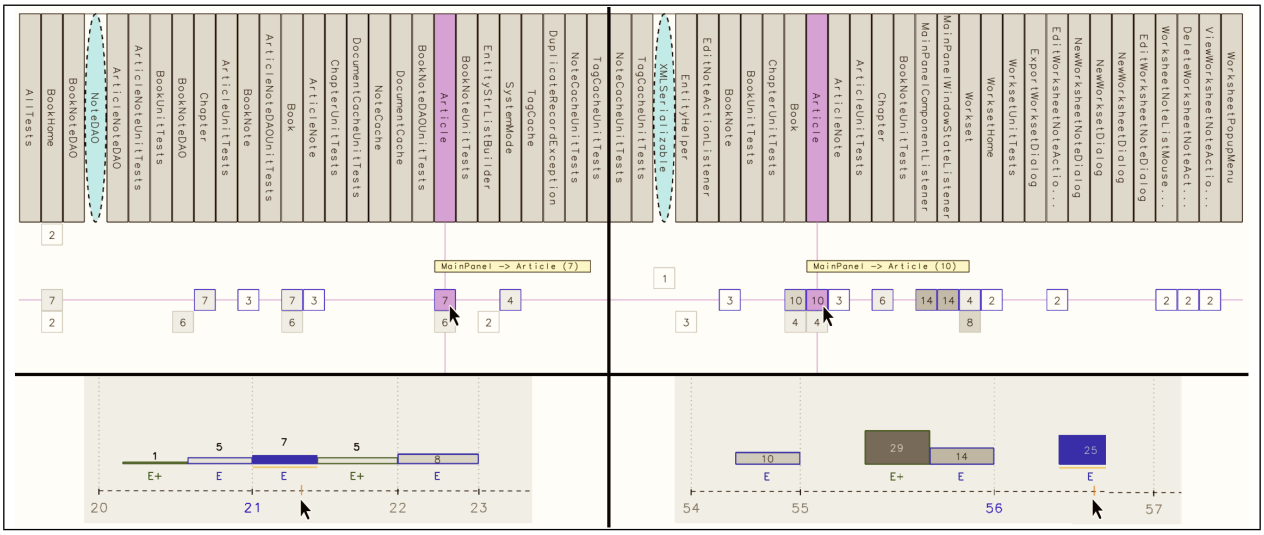

Fig. 7. EasyNote. Evolution of *MainPanel*'s couplings (a *Lazy Element*) between revisions 21 (left) and 56 (right).

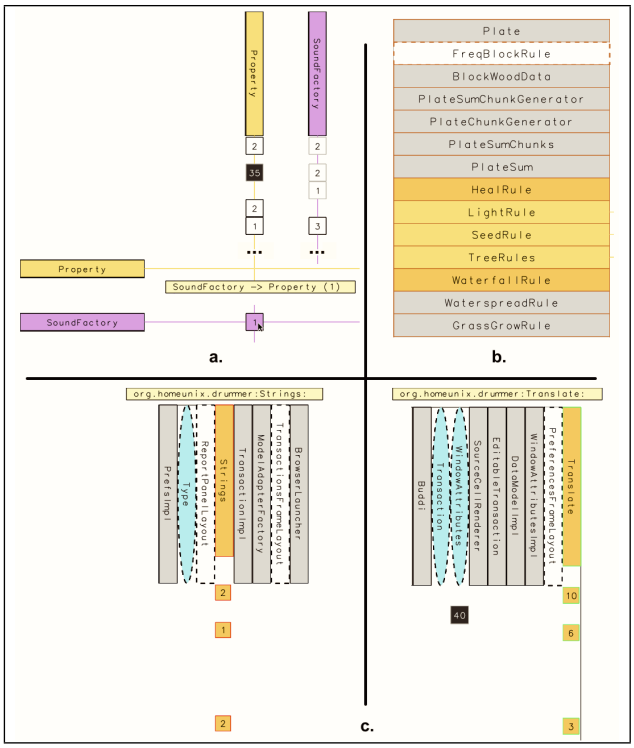

Fig. 8. **a.** *Central Elements* in EasyNote. *SoundFactory* is not really a Factory instance as it does not encapsulates *Property*, which is reused by several other classes (revision 102). **b.** Different kinds of *Rules* classes are placed close to each other in Hexel (revision 54). **c.** In Buddi, the central element *Strings* is renamed and refactored to *Translate* (revision 42).

comment field to worksheet"). Couplings toward these stable interfaces slowly increase over time, e.g., between *MainPanel* and the *Document* interface at revision 110. Thus, the panel component does not need to know about the concrete document types.

3) Restructurings: There are several moving objects at, e.g., revision 49 as data model classes are reorganized ("Move book and article entities to be under entity package"). At revision 119, there is a (small) Big Bang, in which test cases (e.g., *NoteCacheUnitTests*, *BookNoteDAOUnitTests*, *ArticleNoteDAOUnitTests*) are moved around ("Refactoring of cache and DAO layers").

B. Hexel

1) Usages: Core utility elements, such as vectors (*Vector2i*, *Vector3i*) and blocks classes (e.g., *Block*) are central in this project. *Block* is abstract and this abstraction is reused at several places in this block-based game. On the other hand, *PagedCache* and *FreqBlockRule* are rarely reused abstractions but they seem to exist for very specific needs. Also, different kinds of Rules (e.g., *HealRule*, *LightRule*, *WaterfallRule*) are related and often placed near each other (Figure 8.b).

2) Design Stability: There are some interfaces in this project but except for a few reorganizations most of them are added during the initial commit, which suggests a (relatively) stable design structure. For instance, *Wanderer* and *Player* are concrete implementations of the *Movable* interface, and describe how they should be moved in space. Using filterings, we discover that *ThingSimulator* is essentially the only class that directly refers to the *Movable* interface over time (Figure 9). Until revision 92, *ThingSimulator* did not know about the concrete implementations of *Movable*. However, when the *Humanoid* class (implementing *Movable*) is introduced, *Wanderer* is refactored and now extends *Humanoid*. So, the coupling from *Wanderer* to *Movable* is removed, and the *Humanoid* class now refers to *Movable* instead. Also, *ThingSimulator* is now coupled to the *Player* class. Checking the source code of *ThingSimulator*, this coupling (that should be avoided) is caused by the casting of objects to *Player* ("if (t instanceof Player)").

At revision 93, the *Controllable* interface is inserted, and

a number of clients refer to it (such as *Engine*, *Renderer*, *GameOverlay*). However, only the *Player* class implements this interface and it does not change over time. So, this suggests that the *Controllable* interface was preventively added but is not used too much at this point.

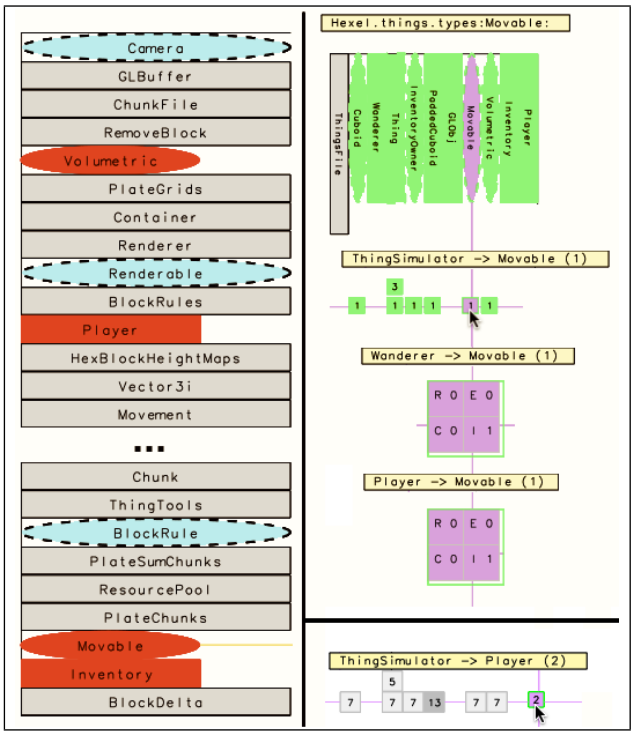

Fig. 9. Hexel. At revision 72 (left), interfaces (e.g., *Movable*) are moved to a new subpackage (*Hexel.things.types*). Then, as an example of a protected variation, *Movable* has two concrete implementations (*Wanderer* and *Player*) and one usage (*ThingSimulator*). Over some time, *ThingSimulator* is never coupled to *Wanderer* nor *Player*, and rely on the stable interface *Movable*. However, at some point (at revision 92), *ThingSimulator* becomes coupled to the *Player* class.

3) Restructurings: Between revisions 50 and 58, some elements are restructured, and also several others are added to support terrain generation features in a new *terrainGenerator* package. At revision 72, some interfaces (e.g., *Movable*, *InventoryOwner*, *Volumetric*) are moved to a new subpackage (*Hexel.things.types*) as shown in Figure 9. Finally, *WaterFallRule* and *WaterSpreadRule* are removed, and several links involving *BlockRules* (derived from the *Block* central element) are also changed for a "simplified and improved water logic" (revision 97).

C. Buddi

1) Usages: Central elements include *Strings* and *Log* classes at the initial commit. *Strings* (which is renamed to *Translate* later) along with *TranslateKeys* are used by several other elements over the duration of the project. *PrefsInstance* is a Singleton class managing the application's settings and it is also reused several times.

2) Design Stability: *PrefsPackageImpl* has increased couplings toward several interfaces, especially at revision 296. Some of these interfaces are in fact encapsulating a *DataModel* (for example, the *Intervals* Interface is implemented by the *IntervalsImpl* class), based on the Eclipse Modeling Framework Project (EMF). At revision 334, a "plugin architecture" is refactored, and a Factory instance (*PluginFactory*) plays a role in the new design structure (Figure 10). Some clients classes using this Factory are *ReportPanel* and *BuddiMenu* to access plug-in features from within the application's main menu. Other client classes are added over time (showing the usefulness of this Factory), such as *PreferencesDialog* at revision 336. Exploring the edge types and checking the source code reveal that the Factory uses reflections (instead of instantiating the objects).

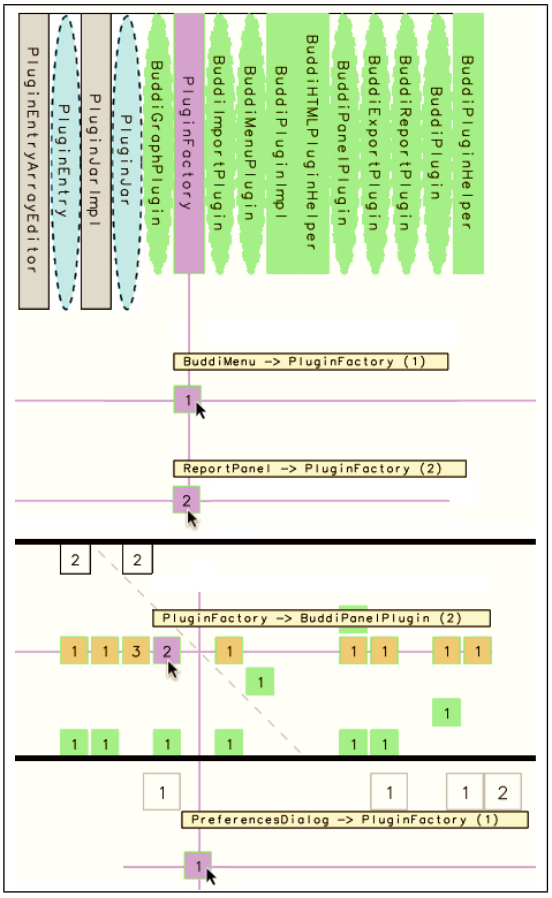

Fig. 10. Buddi. *PluginFactory* is reworked at revision 334 and is responsible for creating and initializing plug-ins. *BuddiMenu* and *ReportPanel* are initial clients of that Factory, while others (e.g., *PreferencesDialog*) are added at revision 336. These clients are not coupled to the encapsulated plug-ins (e.g., *BuddiPanelPlugin*) over time and refer solely to the Factory.

3) Restructurings: The central element *Strings* is renamed and refactored to a *Translate* class at revision 42, causing many small changes (Figure 8.c). At revision 112, some classes are removed and then re-added (e.g., *JTextFieldHint* and *JDecimalLabel*) as they are "Re-arranged". At revision 224, some packages are "moved around to more accurately model the MVC separation". Finally, several abstract classes are added at revision 710 (e.g., *SourceImpl*, *BudgetCategoryType*, *BudgetCategoryTypeImpl*) to "rework the data model".

VII. Conclusion and Future work

We introduced AniMatrix, an interactive and animated matrix-based visualization that we used to explore evolving software. We first presented a taxonomy of dynamic network visualizations, along with a classification of observations concerning changing software. We then used this classification to analyze real-world software repositories. Modules with many incoming couplings generally remained central as the software evolved. Indeed, some functions (such as logging) are needed by several classes and are often placed into separate elements for reusability. We also discovered abstractions that were not used a lot, or that were created and then removed later on as the design evolved. Evolution patterns of software elements and couplings suggested adequate applications of information hiding principles in some cases but also revealed opportunities for improvement. AniMatrix was useful to browse large change histories and also allowed to discover cases of restructurings (such as package reorganizations). In future work, we plan on supporting other types of changes as well as module hierarchies. Interaction techniques, hybrid visualizations and advanced filtering options could also improve the performance of tasks, such as inspecting and evaluating patterns in evolving software.

Acknowledgment

Thanks to the anonymous reviewers and to Christopher Fuhrman for their insightful comments. The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 284625.

References

[1] E. R. Tufte and P. R. Graves-Morris, *The visual display of quantitative information*. Graphics press Cheshire, CT, 1983, vol. 2.

[2] C. Erten, P. Harding, S. Kobourov, K. Wampler, and G. Yee, "GraphAEL: Graph Animations with Evolving Layouts," in *Graph Drawing*. Springer, 2004, pp. 98–110.

[3] J. S. Yi, N. Elmqvist, and S. Lee, "TimeMatrix: Analyzing Temporal Social Networks Using Interactive Matrix-Based Visualizations," *International Journal of Human-Computer Interaction*, vol. 26, no. 11-12, pp. 1031–1051, 2010.

[4] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," in *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, 2005, pp. 13–22.

[5] D. Archambault, H. C. Purchase, and B. Pinaud, "Difference Map Readability for Dynamic Graphs," in *Graph Drawing*. Springer, 2011, pp. 50–61.

[6] D. Archambault, H. Purchase, and B. Pinaud, "Animation, Small Multiples, and the Effect of Mental Map Preservation in Dynamic Graphs," *IEEE TVCG*, vol. 17, no. 4, pp. 539–552, Apr. 2011.

[7] H. Purchase and A. Samra, "Extremes are better: Investigating mental map preservation in dynamic graphs," in *Diagrammatic Representation and Inference*, ser. LNCS. Springer, 2008, vol. 5223, pp. 60–73.

[8] B. Tversky, J. B. Morrison, and M. Betrancourt, "Animation: can it facilitate?" *International Journal of Human-Computer Studies*, vol. 57, no. 4, pp. 247–262, Oct. 2002.

[9] L. Zaman, A. Kalra, and W. Stuerzlinger, "The effect of animation, dual view, difference layers, and relative re-layout in hierarchical diagram differencing," in *Proceedings of Graphics Interface*, 2011.

[10] A. L. Griffin, A. M. MacEachren, F. Hardisty, E. Steiner, and B. Li, "A Comparison of Animated Maps with Static Small-Multiple Maps for Visually Identifying Space-Time Clusters," *Annals of the Association of American Geographers*, vol. 96, no. 4, pp. 740–753, Dec. 2006.

[11] S. Rufiange and M. J. McGuffin, "DiffAni: Visualizing Dynamic Graphs with a Hybrid of Difference Maps and Animation," *IEEE TVCG*, vol. 19, no. 12, pp. 2556–2565, 2013.

[12] J. Mackinlay, "Automating the design of graphical presentations of relational information," *Transactions on Graphics*, vol. 5, no. 2, pp. 110–141, Apr. 1986.

[13] M. Ghoniem, J.-D. Fekete, and P. Castagliola, "On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis," *Information Visualization*, vol. 4, pp. 114–135, 2005.

[14] N. Elmqvist, T.-N. Do, H. Goodell, N. Henry, and J. D. Fekete, "ZAME: Interactive large-scale graph visualization," in *Proceedings of the IEEE Pacific Visualization Symposium*, 2008, pp. 215–222.

[15] S. Diehl, *Visualizing The Structure, Behaviour, and Evolution Of Software*. Germany: Springer, 2007.

[16] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *Proceedings of the OOPSLA Conference*, vol. 40, no. 10, 2005, pp. 167–176.

[17] H. Abdeen, S. Ducasse, D. Pollet, and I. Alloui, "Package Fingerprints: A visual summary of package interface usage," *Information and Software Technology*, vol. 52, pp. 1312–1330, 2010.

[18] S. Rufiange, M. J. McGuffin, and C. P. Fuhrman, "TreeMatrix: A Hybrid Visualization of Compound Graphs," *Computer Graphics Forum*, vol. 31, no. 1, pp. 89–101, 2012.

[19] A. Abuthawabeh, F. Beck, D. Zeckzer, and S. Diehl, "Finding structures in multi-type code couplings with node-link and matrix visualizations," in *Proceedings of the VISSOFT Conference*, 2013.

[20] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," in *Proceedings of the ACM Symposium on Software Visualization*. ACM, 2003, pp. 77–ff.

[21] A. Hindle, Z. M. Jiang, W. Koleilat, M. W. Godfrey, and R. C. Holt, "YARN: Animating Software Evolution," in *Proceedings of the VISSOFT Conference*, 2007, pp. 129–136.

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

[23] S. Rufiange and C. P. Fuhrman, "Visualizing protected variations in evolving software designs," *Journal of Systems and Software*, vol. 88, Feb. 2014.

[24] J. A. Cottam and A. Lumsdaine, "Watch this: A taxonomy for dynamic data visualization," *Visual Analytics Science*, 2012.

[25] F. Beck, M. Burch, S. Diehl, and D. Weiskopf, "The state of the art in visualizing dynamic graphs," in *EuroVis STAR*, 2014.

[26] M. Burch, F. Beck, and D. Weiskopf, "Radial Edge Splatting for Visualizing Dynamic Directed Graphs," in *Proceedings of Information Visualization Theory and Applications*, 2012, pp. 603–612.

[27] C. Plaisant, J. Grosjean, and B. B. Bederson, "SpaceTree: supporting exploration in large node link tree, design evolution and empirical evaluation," *IEEE Symposium on Information Visualization*, pp. 57–64, 2002.

[28] P. Saraiya, C. North, and K. Duca, "An insight-based methodology for evaluating bioinformatics visualizations," *IEEE TVCG*, vol. 11, no. 4, pp. 443–456, 2005.

[29] M. Fowler, *Refactoring : improving the design of existing code*. Boston: Addison-Wesley, 2000.

[30] C. Larman, "Protected variation: the importance of being closed," *IEEE Software*, vol. 18, pp. 89–91, 2001.

[31] B. Bach, E. Pietriga, and J. D. Fekete, "GraphDiaries: Animated Transitions and Temporal Navigation for Dynamic Networks," *IEEE TVCG*, vol. 20, no. 5, pp. 740–754, May 2014.

[32] N. Henry, J. D. Fekete, and M. J. McGuffin, "NodeTrix: a Hybrid Visualization of Social Networks," *IEEE TVCG*, vol. 13, pp. 1302–1309, 2007.

[33] A. Pilhofer, A. Gribov, and A. Unwin, "Comparing Clusterings Using Bertin's Idea," *IEEE TVCG*, vol. 18, no. 12, pp. 2506–2515, Dec. 2012.

[34] M. J. McGuffin, "Simple algorithms for network visualization: A tutorial," *Tsinghua Science and Technology*, vol. 17, no. 4, pp. 383–398, 2012.

[35] M. Harrower and C. A. Brewer, "ColorBrewer.org: An Online Tool for Selecting Colour Schemes for Maps," *The Cartographic Journal*, vol. 40, no. 1, pp. 27–37, Jun. 2003.