Visual Exploration of Large-Scale System Evolution

Richard Wettel and Michele Lanza
Faculty of Informatics - University of Lugano, Switzerland

Abstract

The goal of reverse engineering is to obtain a mental model of software systems. However, evolution adds another dimension to their implicit complexity, effectively making them moving targets: The evolution of software systems still remains an intangible and complex process. Metrics have been extensively used to quantify various facets of evolution, but even the usage of complex metrics often leads to overly simplistic insights, thus failing at adequately characterizing the complex evolutionary processes.

We present an approach based on real-time interactive 3D visualizations, whose goal is to render the structural evolution of object-oriented software systems at both a coarse-grained and a fine-grained level. By providing insights into a system's history, our visualizations allow us to reason about the origins and the causalities which led to the current state of a system. We illustrate our approach on three large open-source systems and report on our findings, which were confirmed by developers of the studied systems.

1 Introduction

Reverse engineering aims at obtaining a mental model of a software system, usually performed on its most recent version. Several researchers have proven that also the history of software hides valuable insights, which are hard to discover outside the evolutionary context. However, taking into account evolutionary information adds to the complexity of the analysis, as the data volume literally explodes, thus posing challenging conceptual and technical problems.

The phenomenon of software evolution has been first investigated by Lehman in the 70s, later resulting in a set of *laws* of software evolution [26] which established that as systems evolve, they become more complex, and consequently more resources are needed to preserve and simplify their structure. This led to the notion that *change* is inevitable and should not be avoided, but actually embraced [3]. In the past decade, software evolution research has been stimulated by the widespread use of versioning systems, such as Subversion and CVS, mostly by the open-source community. This led many researchers to focus their activities on the massive amounts of information recorded by the versioning systems, resulting in a number of publications on "who did what at which time and how/why?".

While we strongly believe in the value of such research, we argue that little attention has been dedicated to the evolutionary phenomenon itself, i.e., the question "what does an evolving system look like?" still lacks an answer that can hardly be given by the usually provided line/bar charts that depict the evolution of one particular aspect of a system over time. Moreover, a number of terms that appear in this context, such as code decay [12], design erosion [36], architectural drift, etc. have found acceptance, but still fail at creating a mental model of what they actually imply.

In the context of reverse engineering and software evolution research, visualization has proven to be a key technique, due to the large amounts of information that need to be processed and understood [34]. Most existing approaches are however focused on depicting only particular aspects (such as lines of code, number of modules/classes, developer activity, etc.) of evolving systems, and not on rendering the structural evolutionary process as such.

We present a set of elaborate interactive 3D visualizations which illustrate the structural evolution of large software systems at both a coarse-grained and a fine-grained level. The visualizations make the complex and intangible process of software evolution tangible and visible, and allow for insights into a system's evolution. We use our approach on one system (ArgoUML) that we previously studied, and on two systems (JHotDraw and Jmol) that we have looked at for the first time.

Structure of the paper. In Section 2 we briefly present the city metaphor we use as a basis for our approach to evolution analysis, followed by our case studies in Section 3. In Section 4 we introduce three visualization techniques and apply them on the systems. We discuss the results in Section 5. In Section 6 we report on our tool support, and discuss related work in Section 7. We conclude in Section 8.

We make extensive use of color pictures. Please read the article on-screen or as a color-printed version. We also provide a set of companion movies which can be accessed by clicking on the provided hyperlinks in the PDF version.

2 The Present is Not Enough

In the context of the EvoSpaces[1] project, which aims at exploiting multi-dimensional navigation spaces to visualize evolving software systems, we have experimented with several metaphors [6, 7] to provide some tangibility to the abstract nature of evolving software. We settled on a *city* metaphor [40], because it offers a clear notion of locality, thus supporting orientation, and features a structural complexity that cannot be oversimplified. This led to the adoption of the metaphor in the project's supporting tool [1].

We represent classes as buildings located in districts representing the packages where the classes are defined. The visual properties of the city artifacts reflect metric values. Our most used configuration is: for classes, the number of methods (NOM) mapped on the buildings' height and the number of attributes (NOA) on their base size, and for packages the nesting level mapped on the districts' color saturation. To efficiently use the city real estate, we implemented a hierarchical layout based on *k*d-tree space partitioning [4].

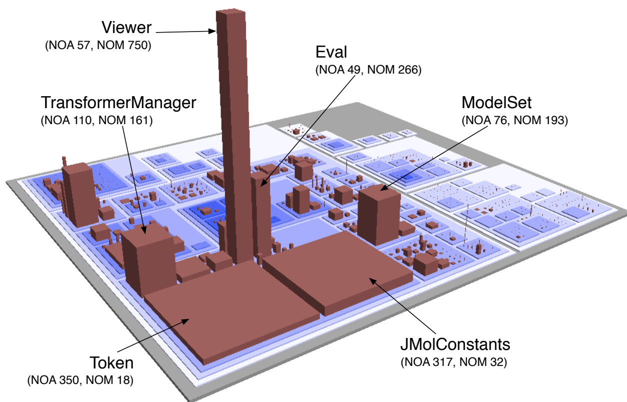


Figure 1. The Jmol system (revision 8065)

The usefulness of this approach for program comprehension tasks is described in detail in a previous publication [39]. We limit ourselves here to briefly illustrate it on a recent version of the Jmol system, depicted in Figure 1: Jmol is composed of ca. 1,000 classes (*i.e.,* the brown buildings) and 100 packages (*i.e.,* the blue districts). We observe outliers, such as the two large platforms (wide and short) in the foreground representing the classes Token and JmolConstants, which define many attributes (large base) and few methods (reduced height), or the "skyscraper" representing the class Viewer with an extremely high number of methods (750) and a much lower number of attributes (57).

Although helpful in reverse-engineering a single version of a system, this visual representation does not help in understanding the system's evolution.

[1] http://www.inf.unisi.ch/projects/evospaces

3 Our Case Studies

Before presenting and discussing our approach to evolution analysis, we briefly introduce the systems we used as case study (See Table 1).

System	ArgoUML	JHotDraw	Jmol
Packages	144	72	105
Classes	2,542	998	1,032
Lines of Code	137,000	30,000	85,000
Sampling Start	Oct 2002	Oct 2000	Jan 2000
Sampling End	Feb 2007	Apr 2005	Aug 2007
Sampling Period	random	weekly	8 weeks
Samples	9	57	50
Revisions	13,535	267	8,065

Table 1. Systems under study

ArgoUML is a UML modeling tool, whose version 0.23.4 we reverse-engineered in a previous experiment [39]. By the end of the experiment we were left with a number of open questions whose answers were buried in the system's history. For the current experiment we added all the major releases of ArgoUML, resulting in a non-periodic sampling.

JHotDraw is a 2D graphics framework which we sampled using a 1-week interval. By removing duplicate samples (no commits in between) we ended up with 57 versions.

Jmol is a 3D molecular viewer for chemical structures. We sampled the history of Jmol using a sampling period of 8 weeks, resulting in 50 sampled versions.

4 Looking Back in Time

Each of our evolutionary visualizations is characterized by the granularity of the representation and by the technique applied to reveal a particular evolutionary aspect of the system under investigation. We have experimented with two levels of granularity for the representation. At a *coarse-grained* level, classes are represented as monolithic blocks, omitting internal details. At a *fine-grained level* we move the focus on the methods, which appear as bricks that constitute the body of the building corresponding to the class. We developed three visualization techniques —applicable at both granularity levels— for software evolution analysis:

1. *Age map* to depict the age distribution,
2. *Time travel* to step through the system's history, and
3. *Timeline* to capture the entire evolution of a software artifact in a single view.

For each technique and granularity level we provide a *description*, a set of comprehension *goals*, an exemplification of its *application* on one of the systems, for the non-trivial findings an optional "*reality check*" with the actual developers, and a discussion of the *drawbacks*.

4.1 Coarse-Grained Representation

4.1.1 Coarse-Grained Age Map

Description. We overlay the city representing a version of a system with colors mapping the age of the artifacts. The age is an integer value representing the number of sampled versions that the artifact "survived" up to that version. The color scheme ranges from light yellow for new-born classes/packages to dark blue for the oldest ones.

Goals. Obtain a starting point for the evolution analysis. Discover the old parts of the system, discover the recently changed parts of the system. Get an overall feeling on the system's evolution by "looking back in time".

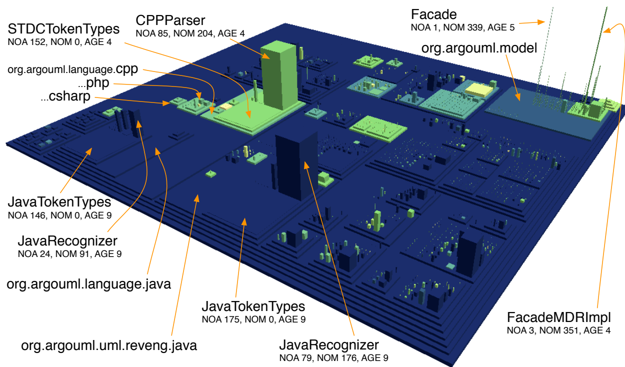


Figure 2. Coarse-grained *age map* of ArgoUML

Application. In Figure 2 we see an *age map* of ArgoUML 0.24. It is a fairly large system with some massive classes. We see that the classes JavaTokenTypes and JavaRecognizer appear twice, namely in org.argouml.language.java and in org.argouml.uml.reveng.java. The *age map* indicates that both pairs are as old as the system's history (denoted by their common dark color). This duplication was an open question in [39] when we only looked at one version. Now we can discard the hypothesis that it is a migration/replacement of one pair of classes with the other, since both pairs have been part of the system since its inception. Another insight is that ArgoUML has supported the Java language since the beginning. The support for C++, C#, and PHP has been added at a later stage, as shown by the light green color of the packages org.argouml.language.cpp/php/csharp, respectively.

Drawbacks. The age map flattens the evolutionary information with respect to the currently visualized system version. What we need is a technique to visualize the process of evolution itself, i.e., we need to *travel through time*.

4.1.2 Coarse-Grained Time Travel

Description. *Time traveling* is achieved by stepping back and forth through the history of the system while the city updates itself to reflect the currently displayed version. Locality plays a major role: We make sure that every artifact is assigned an individual real estate in the city, maintained throughout the entire evolution, i.e., if an artifact is removed from the system or does not yet exist in the system at a certain point in time, this is denoted by an empty space that cannot be occupied by other artifacts. To ease the observation process in the presence of the cities' evolutionary dynamics (*e.g.,* buildings growing, shrinking, disappearing), we use *color tracking*, i.e., we can manually assign particular colors to the entities of interest, which are then consistently maintained in all the visualized versions.

Goals. Observe the evolution both of the entire system and of individual artifacts (*e.g.,* packages, classes). At the system level, discover which districts have been under heavy maintenance or barely touched between two consecutive versions or along their entire evolution. By focusing on a particular artifact in a city, observe its "birth", its evolution in terms of the chosen set of metrics, and in some cases its "death" when it was removed from the system.

Application. In Figure 3 we see the sequence of views obtained during our *time travel* through ArgoUML's sampled history. To provide a better sense of time we marked on the figure the release numbers and dates. The elements that are object of the following discussion are color-tracked with red for emphasis[2]. Another open question from the previous experiment [39] whose answer we hoped to find by analyzing ArgoUML's evolution was the intriguing case of the Facade interface, whose over 300 declared methods are implemented by one class only. Stepping through time reveals the origin of this apparently questionable design decision: In version 0.14 a large building (60 attributes, 180 methods) representing class ModelFacade emerges, then it grows enormously (108 attributes, 405 methods) in version 0.16. In version 0.18.1 ModelFacade dies, but its death coincides with the birth of two other tall and thin buildings: the interface Facade (1 attribute, 306 methods) and the concrete class NSUMLModelFacade (2 attributes, 319 methods) implementing Facade. One possible reason for this change is that the developers realized that ModelFacade was growing into a maintainer's nightmare due to its size, or they needed to define variations of it. They declared the common behavior (306 methods) in an interface and moved the concrete behavior from ModelFacade to the new class NSUMLModelFacade, the first implementor of Facade. Version 0.20 gives birth to a second implementor of Facade, called FacadeMDRImpl (2 attributes, 329 methods). To observe this,

[2]We provide a movie of this *time travel*, located at http://www.inf.unisi.ch/phd/wettel/download/argouml-coarse.mov

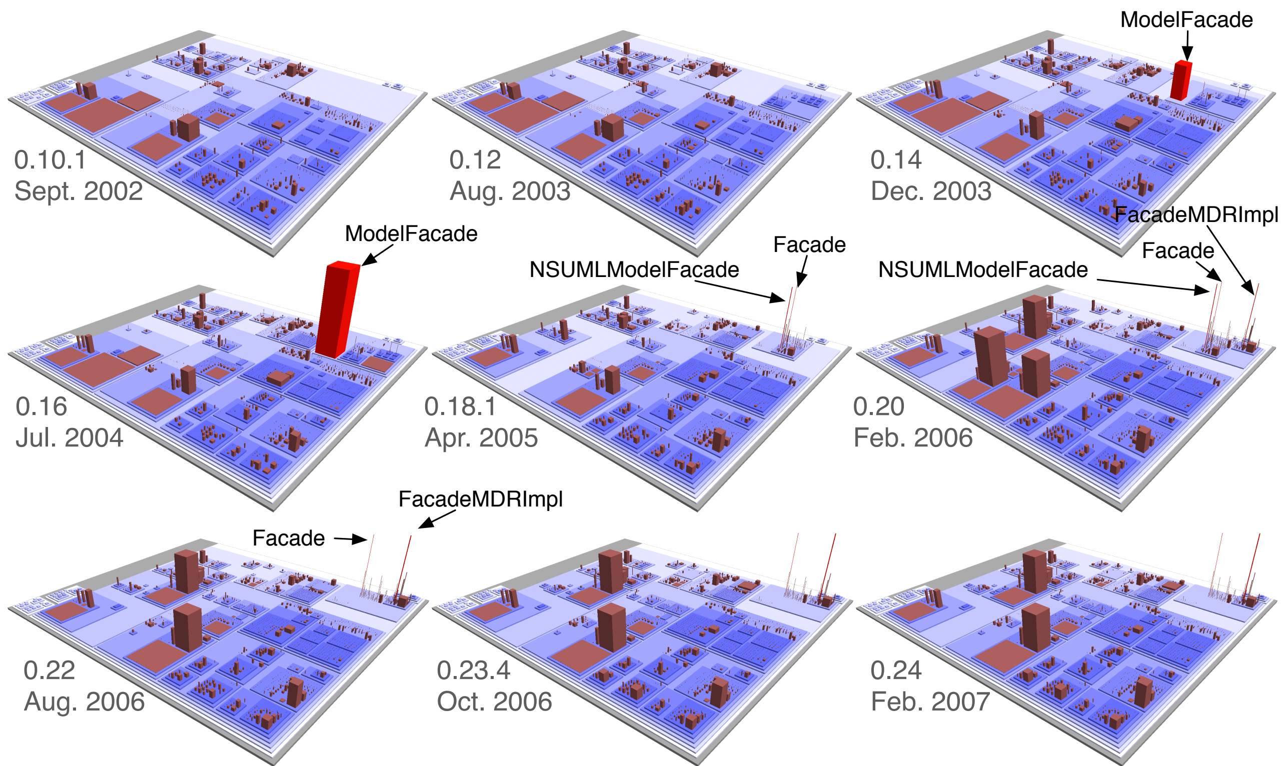


Figure 3. Coarse-grained *time travel* through the history of ArgoUML

we looked for buildings at least as tall (*i.e.,* NOM value) as the one representing the interface, because in Java a class implementing an interface is enforced to implement all the methods declared in the interface. Version 0.20 is the only sampled version of ArgoUML in which the two implementors coexist, as in version 0.22 the class NSUMLModelFacade is removed, leaving FacadeMDRImpl the only implementation of Facade.

Reality Check. A key-developer of ArgoUML confirmed our insights gained during the *time travel* and provided more details. At a higher level, the model repository switch from NSUML to MDR required flexibility, which was indeed obtained by refactoring ModelFacade into a common interface and a concrete class. The numerous attributes of ModelFacade were tokens implemented as constants because of the lack of support for enumerations in Java 1.4.

Drawbacks. At this granularity level, the technique does not provide information about the internal evolution of classes, which is the level at which the changes happen. For example, if a developer removes and adds the same number of methods between two consecutive versions, the NOM metric value and consequently the building's height remain the same, despite the substantial changes. This loss of detail is visible in the last 3 versions of ArgoUML (Aug. 2006 to Feb. 2007), where not much seems to happen in the system.

4.2 Fine-Grained Representation

To address the need for a more fine-grained representation, we depict methods as cuboids ("bricks") laid out on top of each other in layers of 4 (see Figure 4), in the order of their creation (*i.e.,* older down, newer up). Removed methods are represented as empty spaces. The height of a building continues to be proportional to the number of methods. The base platform provides immediate access to the class for interaction.

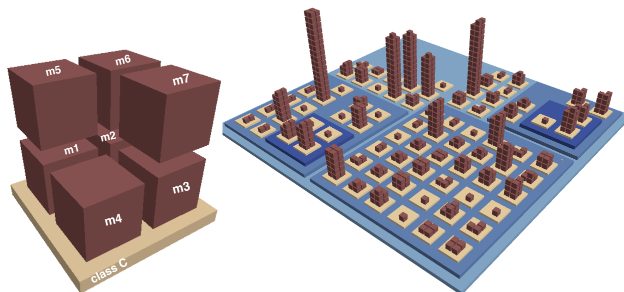


Figure 4. Fine-grained representation (left) applied to ArgoUML's cognitive **package (right).**

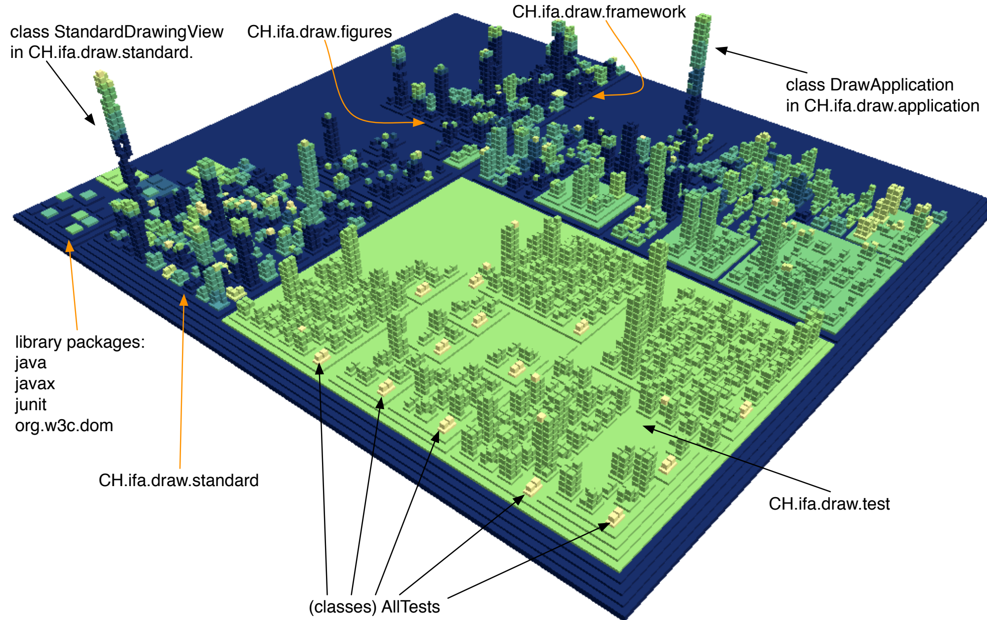


Figure 5. Fine-grained *age map* applied to the most recent version of JHotDraw

4.2.1 Fine-Grained Age Map and Time Travel

Description. The two techniques, described previously, are applied on the fine-grained representation.

Goals. Obtain insights into the method-level evolution. Discover classes created in one "bang" versus classes grown in an incremental manner.

Application. The fine-grained *age map* applied to the last version of JHotDraw (See Figure 5) reveals interesting facts about its evolution. The districts colored in a dark shade (*e.g.,*CH.ifa.draw.standard/framework/figures) represent the most rooted packages within the system, because they have been there since the system's inception. The CH.ifa.draw.test package is relatively new, it appeared in the sampled version 6 (out of 8), denoted by its light (green) color. We can observe light yellow small buildings, representing classes called AllTests that have been added in sampled version 7. Each of these classes contains two methods: main and suite. The main method is the starting point for running a suite of tests (defined in the suite method) for every sub-package of test. The buildings painted in a wide palette of colors (e.g., classes DrawApplication and StandardDrawingView) not only have been part of the system from the first version (their color starting at the base is the same as the city ground), but they have permanently required adaptation during the system's evolution (with each version of the system the developers added/removed methods). We then focused our attention on package test using the *time travel* technique[3], which confirmed what the *age map* depicted: the test package appeared all at once towards the end of JHotDraw's lifetime. Since the first sampled history of JHotDraw contained only 8 versions covering $3^1/_2$ years, we did not want to jump to conclusions. The sudden appearance of all the unit tests could have happened gradually over a period of 6 months, between Jan. and Jul. 2003. To reason more accurately about this evolutionary fact, we sampled the system using a weekly sampling period and to our surprise, we obtained the same result: The test package appears entirely from revision 121 (24/01/03) to revision 155 (31/01/03), which reduces the possible period to these 7 days , during which 34 commits were performed. Writing all tests at once and at a late stage in a project is curious.

Reality Check. The developer who created these classes told us that he used a JavaDoc-based code generator to automatically generate test cases for JHotDraw, in the form of unimplemented skeletons for the work to be done.

Drawbacks. One cannot know *when* a particular method disappeared, since its representation is an empty space. Also, we are not able to distinguish the methods whose bodies have been changed from the ones that remained the same. It also raises some scalability issues, *e.g.,* ArgoUML is represented by over 16,000 elements. Finally, if one visualizes complete systems, the methods of a class are hard to track. To remedy this, we use the *timeline* technique to depict the entire history of specific artifact(s), discussed next.

[3] A movie of this *time travel* is located at http://www.inf.unisi.ch/phd/wettel/download/jhotdraw-fine.mov

4.2.2 Fine-Grained Timeline

Description. The class versions are represented as platforms next to each other along a timeline, from left (first version) to right (last version) and the methods are represented as "bricks". We combine this with the *age map* technique to enable a clearer visual distinction between the different "generations" (*i.e.,* groups of methods created in the same version) of methods.

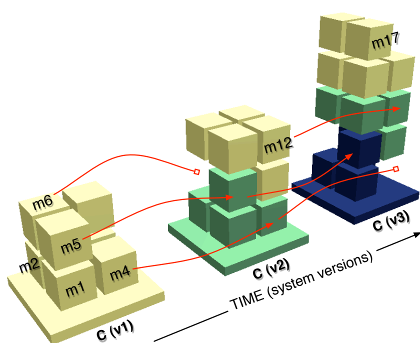


Figure 6. Principles of fine-grained *timeline*

Figure 6 illustrates these principles applied to class C throughout a history of 3 versions. In the first version, class C has 7 methods (m1 to m7). In the second version, method m6 is removed and other 5 methods (m8 to m12) are added to the class, appearing at the top of the building in a lighter color than the rest of the methods. The place formerly occupied by m6 will be represented by an empty space. In version 3, the older method m4 is removed and 6 methods are added. The benefit of this visualization is that it provides a complete representation of an artifact's evolution, thus allowing for the detection of evolutionary patterns. We use this technique mostly at the class level, by visualizing the evolution of a class in terms of all its methods, but it can also be applied at the package level, showing a package and its sub-packages and classes.

Goals. Isolate a reduced set of artifacts to create a view that presents their entire history including all the inner components, and observe evolution patterns, such as incrementally grown classes, recurring methods, etc.

Application. We analyzed a number of Jmol's classes with the *timeline*. The first example is the class Graphics3D, which was present in the last 23 versions of the system's 50 version-long sampled history (see Figure 7). It is probably a core class, because since its first version it defined a large number (103) of methods. Signs of decay appear in its 9th sampled version and intensify with every subsequent one. At the same time, new functionality (*i.e.,* methods) is gradually added to the class. The final version reflects its history of continuous adaptation: out of the 311 methods throughout its evolution, only 158 made it to this version.

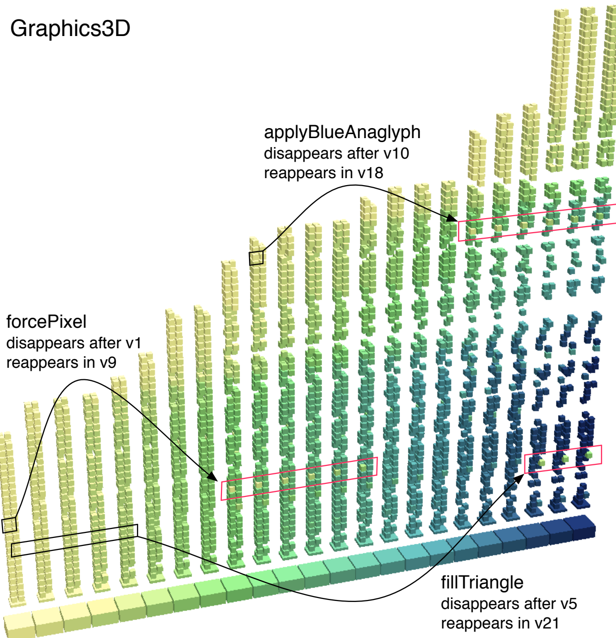
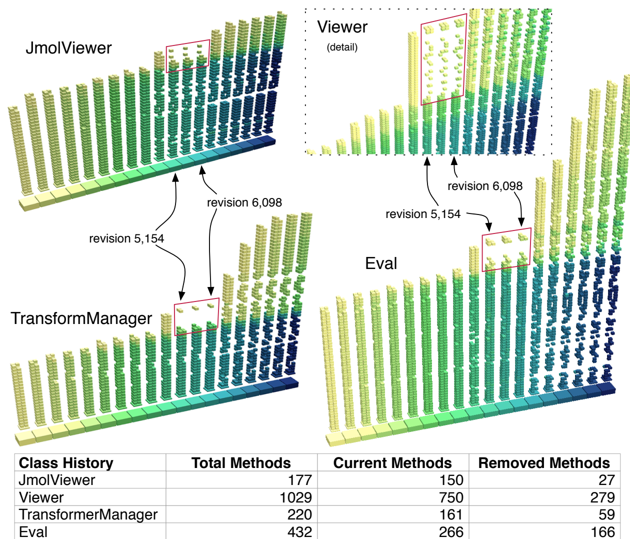


Figure 7. *Timeline* **of class** Graphics3D

Figure 7 also illustrates an interesting pattern: Some of the building's missing bricks reappear after a number of versions, which happens when removed methods are later restored. This pattern is particularly visible with our approach because of the combined effects of the *age map* technique and of the chronological order imposed on the layout. After a method is resuscitated, the color of the brick representing it stands out as an anomaly in every subsequent version of the class, *i.e.,* it breaks the color pattern of the bricks around it. Although its position denotes the fact that it was born in the same version as its neighbor bricks, the color reflects a shorter life (*i.e.,* fewer versions) than the one of its neighbors. Figure 7 shows 3 methods of the class exhibiting this behavior. The more versions pass between the removal and the restoration, the more striking is this color anomaly, as in the case of method fillTriangle (an interval of 15 versions).

To further investigate this evolutionary pattern, we correlated the *timelines* of 4 class histories, presented in Figure 8. At a first glance, we see how each *timeline* reflects the evolutionary characteristics of one class history. For instance, the peak of each *timeline* (the height of the last version of the building) depicts the number of method histories: Eval is twice as tall as JmolViewer or TransformManager. Viewer is, due to its over 1,000 method histories, difficult to display entirely. Eval lost many of its methods during its evolution (166 out of 432) and looks unstable with many missing bricks in its current version (*i.e.,* last column). This shows that the fine-grained *timeline* depicts the decay of software in a very suggestive way.



Class History	Total Methods	Current Methods	Removed Methods
JmolViewer	177	150	27
Viewer	1029	750	279
TransformerManager	220	161	59
Eval	432	266	166

Figure 8. Similar pattern in 4 classes of Jmol

Figure 8 presents a detail of the *timeline* of class Viewer, which had 1,029 methods throughout its history, and the complete *timelines* of the classes JmolViewer (the superclass of Viewer), Eval, and TransformManager. Each of these timelines contains a set of bricks which disappears in revision 5,154 from 22/05/2006 and reappears after exactly three sampled versions, in revision 6,098 from 6/11/2006. Our hypothesis was that the developers massively removed methods from these logically coupled classes, thus generating bugs which were not detected right away and which could only be fixed later by reviving the removed methods.
Reality Check. The revision log 5,091 from 10/05/2006 says: "No more javax.vecmath.Point3f in g3d shape drawing routines. There were some cases where screen coordinates were being passed in as Point3f objects[..]" and the log of revision 5,579 from 17/09/2006 acknowledges the recovery of the previously removed methods: "Revert of vecmath lib change". Also, three Jmol developers confirmed this and told us that after some refactorings they reverted to an earlier state because of performance problems in their graphic display module which resulted in slow rendering.
Drawbacks. Scalability issues in the case of an artifact having hundreds or more of revisions.

5 Discussion

Tracking events in time. The actual time at which a particular event happened can be anywhere between the commit times of the earliest sampled version in which the event occurs and of the previous sampled version. In this context, the sampling period plays a major role in establishing accurate time localization of events.

Sampling policies. The sampling policy has a major influence on the information we can extract from the history, which encourages us to strive towards shorter sampling periods and thus richer histories. However, manipulating tens to hundreds of versions of an industrial system raises the issue of scalability with respect to memory requirements and processing time. We believe that an incremental approach can be applied in such cases, by starting first with a sparse history, i.e., few samples distant from each other in time, and then focusing on interesting intervals by increasing the number of samples and decreasing their temporal distance. We applied this approach in the case of the JHotDraw system. We also experimented two different sampling policies, namely time-based (JHotDraw and Jmol) and release-based (ArgoUML). The time-based sampling allows us to observe the evolutionary process as a "slow motion movie" with the drawback of potential duplicated samples. The release-based sampling has the advantage that the frames are steps of the actual development cycle. This eliminates duplicate samples but has the disadvantage that one needs to correlate the development steps with the actually elapsed time.

Color scheme limitations. Applying a color scheme to depict age limits the number of versions one can clearly distinguish. In our experience, a color scheme works best up to a maximum of 10 versions. Increasing this number hinders the visual distinction among consecutive versions.

City metaphor. The city metaphor is a promising way of building evolutionary visualizations. Due to the evolutionary layout which takes into account the entire history of every software element in the system, we support consistent locality, which helps in keeping the viewer oriented at any time. This enables the user to observe hotspot neighborhoods with respect to the evolutionary phenomenon: conservative districts (which rarely change), districts permanently "under construction" or moving districts. The price of providing consistent locality is the extra space used by the layout for allocating lifetime estates to every entity (even to ones with a short life).

The views. Despite the evolutionary structural overview that the *age map* and the *time travel* techniques provide with the coarse-grained representation, their drawback is the low level of detail provided for classes. To make up for this, we created a fine-grained representation, to see how method addition and removal drives the evolution of classes. However, this level of detail raises scalability problems in the case of very large systems. Moreover, the overview is lost for such large systems, i.e., the details are not visible from far away and after zooming in, we lose the overall context. Since our views present only one version of the system at a time, with the possibility to perform *time travels*, we needed to be able to focus on a single element throughout its entire evolution. The *timeline* technique makes this possible and allows for the detection of evolutionary patterns.

6 Tool Support

We built our tool called CodeCity on top of the Moose framework [11], which provides an implementation of a meta-model for history, called Hismo [16]. In Hismo, a history is a sequence of versions of the same kind of entity (e.g., class history, package history, *etc.*), where a version is a snapshot of an entity at a certain point in time. Since we do not work directly with the source code, but with a FAMIX model, we first have to load the models for each version and create a model history of the system that we analyze. For parsing Java projects, we use iPlasma [29]. Once we have the model of the system's history, we create interactive visualizations with our tool.

CodeCity allows us to define various view configurations, where we can specify which types of elements in our model to represent, the figure types for each element type, the mapping set between software metrics and visual properties of the figures, layouts, etc. The created visualizations allow the viewer to navigate the urban environment and to interact with the entities by means of manual inspection or a query mechanism. The viewer can apply the *age map* color scheme, perform *time travels* and generate *timelines* for the selected artifacts. An extended discussion on CodeCity's functionality and configurability is presented in [41].

CodeCity is written in Smalltalk, runs on every major platform and is freely available at: `http://www.inf.unisi.ch/phd/wettel/codecity.html`

7 Related Work

Visualizing repositories. Gall *et al.* analyze, by means of diagrams, the evolution of the structure and relate that to the adaptations made to the system, based on product release history [15]. With this very coarse-grained approach they target the module level of detail and describe in large evolutionary aspects, such as growth rates. Van Rysselberghe *et al.* [37] use version management systems as source for the analysis and visualize it by plotting the file releases against release dates to identify components that are unstable or change together, relations that changed, or the system's productivity pace. However, seeing a file that was often changed does not say anything about how the software entities inside evolved. The EvoGraph by Fischer *et al.* [14] is based on the information extracted from the systems' release history and produces 2D visual representations of the evolution of structural dependencies. Taylor *et al.*'s reviewing towers view [35] shows how versioning systems repositories evolve. Voinea *et al.* [38] aim at enriching the quantity of information extracted from software configuration management systems that can be displayed in a view. They use a combination of color and texture for the representation of as many attributes as possible and clustering for the reduction of the complexity in such visualizations. Collberg *et al.* [9] visualize the evolution of a software system as large evolving graphs, using colors to depict the changes.

Visualizing evolving dependencies. Holt *et al.* [19] look at the evolution of systems from the perspective of dependencies and provide comparisons of pairs of versions of the system and focus on the common dependencies on the new ones. Keast *et al.* [21] implement a Rigi [22, 30] extension for visualizations of the evolution of relationships.

Ratzinger *et al.* [33], who represent systems as nested graphs, built their approach around the concept of view lens, which provides the user with enhanced zooming capabilities and allows him to define a focal point for the lens view and navigate along the time dimension by user-defined time windows. Beyer *et al.*'s approach called animated storyboards [5] aims at visualizing how dependencies evolve and uses color schemes to depict how much each element changed. Hindle *et al.* [18] use animation based on modules connected by animated edges to observe the evolution of dependencies. Lungu *et al.* [27] work at the module level of granularity and explore evolution of the inter-module relationships by means of filmstrips that depict "stories of relationships" between modules. Due to their focus on the evolution of the dependencies between modules, the only common characteristics between these approaches and ours is that we all target several versions of a system, looking at different aspects of the evolution.

Visualizing evolving structure. Our timeline representation is partially inspired from Lanza's Evolution Matrix [25], which shows the evolution of classes, represented as rectangles, in terms of a set of metrics mapped on the dimensions of the rectangles. Girba *et al.* [17] raise the granularity level and look into the evolution of class hierarchies using a 2D visualization which correlates the histories of classes and inheritance relationships. Eick *et al.* [13] also use color to depict the age, however at at a lower abstraction level: Each line of code is visualized as a row and the files are visualized as columns in SeeSoft, in which the color of the row depicts the age of a line of code. The age maps in CodeCity depict information at a higher abstraction level and from an object-oriented point of view (*i.e.,* the age of packages, classes, and methods, rather than of each line of code in a file) and puts all of these in the structural context of the system. Pinzger *et al.* [32] work at a higher granularity and visualize various evolutionary aspects of complex software systems using Kiviat diagrams to depict multiple evolution metrics, which provide static visualizations of a large number of metrics for the entire evolution of modules, yet lacks both an overview of the entire system and a fine-grained level of detail. Wu *et al.* [42] propose a visualization technique called evolution spectrographs, which portrays the evolution of a spectrum of components based on a particular property measurement which reduces the each

version of a file to a number. Jazayeri *et al.* [20] use 3dSoftVis to visualize evolution by means of a compacted 3D visualization that shows 2D tree graphs aligned in time and a compact 2D visualization obtained by projecting 3D diagrams onto 2D space. We argue that in spite of the useful information they provide, these visualizations oversimplify the representation of the evolution.

The city metaphor. Knight *et al.* [23] and Charters *et al.* [8] use a city metaphor, while Marcus *et al.* [28] and Balzer *et al.* [2] use a similar 3D metaphor to visualize single versions of software systems. None of these works embeds the time factor to visualize the evolution of software systems in their 3D environments. Ducasse *et al.* [10] use the city metaphor of the SimCity game to express challenges behind software evolution, which remained an idea. Another metaphor idea without implementation is proposed in Panas's [31] description of such a 3D city. Langelier *et al.* [24] use 3D visualizations to display structural information, by representing classes as boxes with metrics mapped on height, color and twist, and packages as borders around the classes placed using a tree layout or a sunburst layout. The fact that the space occupied by their buildings is always the same eases the layout procedure at the price of less-realistic looking cities. They also look at the evolution of individual classes or packages by aligning the representation of each version of that class/package. This provides some useful information about the evolution of that particular entity, however at a too high granularity, reducing thus the effects of the change over one entity to a number.

8 Conclusions

We proposed an approach based on a set of visualization techniques built around a 3D city metaphor to explore the evolution of object-oriented software systems. Applying our approach to correlate the current status of such systems with their history helped us to better understand the evolutionary process and revealed important information that cannot be extracted from any of the versions considered separately, but only in the historical context.

We applied the approach on three large open-source industrial case-studies. In the case of one of the systems for which we already had some prior knowledge about one version (ArgoUML) we finally succeeded in answering two questions that were left suspended due to lack of evidence. We also applied the approach on two systems we have not analyzed before and obtained interesting insights, later confirmed by the developers of the systems.

The main contribution of this article is a set of 3D visualizations, that scale up both to real-world software systems and across several versions of the systems. Our techniques provide tangibility to the otherwise ephemeral nature of the phenomenon of software evolution.

Our future work in this area is applying our approach on larger, more long-lived systems, such as GCC, and to investigate additional views to make software evolution tangible.

Acknowledgments. We gratefully acknowledge the financial support of the Hasler Foundation for the project "EvoSpaces" (MMI Project No. 1976).

References

[1] S. Alam and P. Dugerdil. Evospaces visualization tool: Exploring software architecture in 3d. In *Proceedings of 14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 269–270. IEEE Computer Society, 2007.

[2] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz. Software landscapes: Visualizing the structure of large software systems. In *VisSym 2004, Symposium on Visualization*, pages 261–266. Eurographics Association, 2004.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[5] D. Beyer and A. Hassan. Animated visualization of software history using evolution storyboards. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*. IEEE Computer Society, 2006.

[6] S. Boccuzzo and H. C. Gall. Cocoviz: Supported cognitive software visualization. In *Proceedings of 14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 273–274. IEEE Computer Society, 2007.

[7] S. Boccuzzo and H. C. Gall. Cocoviz: Towards cognitive software visualizations. In *Proceedings of IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007)*, pages 72–79. IEEE Computer Society, 2007.

[8] S. M. Charters, C. Knight, N. Thomas, and M. Munro. Visualisation for informed decision making; from code to components. In *International Conference on Software Engineering and Knowledge Engineering (SEKE '02)*, pages 765–772. ACM Press, 2002.

[9] 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 2003 ACM Symposium on Software Visualization*, pages 77–86, New York NY, 2003. ACM Press.

[10] S. Ducasse and T. Gîrba. Being a long-living software mayor — the simcity metaphor to explain the challenges behind software evolution. In *Proceedings of CHASE International Workshop 2005*, 2005.

[11] S. Ducasse, T. Gîrba, and O. Nierstrasz. Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, pages 99–102, Sept. 2005. Tool demo.

[12] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[13] S. G. Eick, J. L. Steffen, and S. Eric E., Jr. SeeSoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, Nov. 1992.

[14] M. Fischer and H. C. Gall. Evograph: A lightweight approach to evolutionary and structural analysis of large software systems. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE)*, pages 179–188. IEEE Computer Society, 2006.

[15] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM'97)*, pages 160–166, Los Alamitos CA, 1997. IEEE Computer Society Press.

[16] T. Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, Nov. 2005.

[17] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering)*, pages 2–11. IEEE CS Press, 2005.

[18] A. Hindle, Z. M. Jiang, W. Koleilat, M. Godfrey, and R. Holt. Yarn: Animating software evolution. In *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007 (VISSOFT 2007)*, pages 129–136, 2007.

[19] R. Holt and J. Pak. GASE: Visualizing software evolution-in-the-large. In *Proceedings of Working Conference on Reverse Engineering (WCRE 1996)*, pages 163–167, Los Alamitos CA, 1996. IEEE Computer Society Press.

[20] M. Jazayeri, H. Gall, and C. Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, pages 99–108. IEEE Computer Society Press, 1999.

[21] J. Keast, M. Adams, and M. Godfrey. Visualizing architectural evolution. In *Proceedings of ICSE'99 - Workshop on Software Change and Evolution (SCE'99)*, 1999.

[22] H. Kienle and H. Muller. The Rigi reverse engineering environment. In *Proceedings of WASDeTT 2008 (1st International Workshop on Advanced Software Development Tools and Techniques)*, 2008.

[23] C. Knight and M. C. Munro. Virtual but visible software. In *International Conference on Information Visualisation*, pages 198–205, 2000.

[24] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 214–223. ACM, 2005.

[25] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (4th International Workshop on Principles of Software Evolution)*, pages 37–42. ACM Press, 2001.

[26] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[27] M. Lungu and M. Lanza. Exploring inter-module relationships in evolving software systems. In *Proceedings of CSMR 2007 (11th European Conference on Software Maintenance and Reengineering)*, pages 91–100. IEEE CS Press, 2007.

[28] A. Marcus, L. Feng, and J. I. Maletic. 3d representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization*, pages 27–36. IEEE, 2003.

[29] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *ICSM (Industrial and Tool Volume)*, pages 77–80, 2005.

[30] H. Muller and K. Klashinsky. Rigi: a system for programming-in-the-large. *Proceedings of the 10th International Conference on Software Engineering (ICSE '97)*, pages 80–86, 1988.

[31] T. Panas, R. Berrigan, and J. Grundy. A 3d metaphor for software production visualization. *International Conference on Information Visualization*, page 314, 2003.

[32] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization)*, pages 67–75, 2005.

[33] J. Ratzinger, M. Fischer, and H. Gall. Evolens: lens-view visualizations of evolution data. In *International Workshop on Principles of Software Evolution*, pages 103–112, 2005.

[34] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization — Programming as a Multimedia Experience*. The MIT Press, 1998.

[35] C. Taylor and M. Munro. Revision towers. In *Proceedings 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50, Los Alamitos CA, 2002. IEEE Computer Society.

[36] J. van Gurp and J. Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105–119, 2002.

[37] F. Van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 328–337, Los Alamitos CA, Sept. 2004. IEEE Computer Society Press.

[38] L. Voinea and A. Telea. Multiscale and multivariate visualizations of software evolution. In *Proceedings of the 2006 ACM symposium on Software Visualization*, pages 115–124. IEEE Computer Society, 2006.

[39] R. Wettel and M. Lanza. Program comprehension through software habitability. In *Proceedings of ICPC 2007 (15th International Conference on Program Comprehension)*, pages 231–240, 2007.

[40] R. Wettel and M. Lanza. Visualizing software systems as cities. In *Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, pages 92–99, 2007.

[41] R. Wettel and M. Lanza. CodeCity. In *Proceedings of WASDeTT 2008 (1st International Workshop on Advanced Software Development Tools and Techniques)*, 2008.

[42] J. Wu, A. Hassan, and R. Holt. Exploring software evolution using spectrographs. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 80–89. IEEE Computer Society, 2004.