EvoSpaces - Multi-dimensional Navigation Spaces for Software Evolution

Sazzadul Alam[1], Sandro Boccuzzo[2], Richard Wettel[3], Philippe Dugerdil[1], Harald Gall[2], and Michele Lanza[3]

[1] HEG Geneva, Switzerland
[2] University of Zurich, Switzerland
[3] University of Lugano, Switzerland

Abstract. In software development, a major difficulty comes from the intrinsic complexity of software systems and the size of which can easily reach millions of lines of source code. But software is an intangible artifact that does not have any natural visual representation. While many software visualization techniques have been proposed in the literature, they are often difficult to interpret. In fact, the user of such representations is confronted with an artificial world that contains and represents intangible objects. The goal of our EVOSPACES project was to investigate effective visual metaphors (*i.e.,* analogies) between natural objects and software objects so that we can exploit the cognitive understanding of the user. The difficulty of this approach is that the common sense expectations about the displayed world should also apply to the world of software objects. To solve this common sense representation problem for software objects our project addressed both the small-scale (*i.e.,* the level of individual objects) and the large-scale (*i.e.,* the level of groups of objects). After many experiments we decided for a "city" metaphor: At the small scale we include different houses and their shapes as visual objects to cover size, structure and history. At the large-scale level we arrange the different types of houses in districts and include their history in diverse layouts. The user then is able to use the EVOSPACES virtual software city to navigate and explore all kinds of aspects of a city and its houses: size, age, historical evolution, changes, growth, restructurings, and evolution patterns such as code smells or architectural decay. For that we have developed a software environment named EVOSPACES as a plug-in to Eclipse so that visual metaphors can quickly be implemented in an easily navigable virtual space. Due to the large amount of information we complemented the flat 2D world with full-fledged immersive 3D representation. In this virtual software city, the dimensions and appearance of the buildings can be set according to software metrics. The user of the EVOSPACES environment can then explore a given software system by navigating through the corresponding virtual software city.

1 Introduction

Today we rely more and more on software systems. Maintenance has been identified to be the primary factor of the total cost (*i.e.,* up to 90%) of large software systems [10]. One of the key difficulties encountered in software maintenance comes from the intrinsic complexity of those systems, whose size can easily reach millions of lines of source code. Software complexity is recognized as one of the major challenges to the development and maintenance of industrial size software projects. One of the key aspects, which claims more than half the time spent in software maintenance, is software understanding [35]. The goal of this project is to develop visual metaphors to help the maintenance engineer navigate through thousands of modules, classes or any other programming artifacts and identify "locations" of interest to be explored further to get a global understanding of the system.

Software is an intangible artifact that does not have any *natural* visual representation. While many software visualization techniques have been proposed in the literature, they are often difficult to interpret. In fact, the user of such representations is confronted with an artificial world that itself represents intangible objects (*i.e.,* objects of which the user does not have a sensitive experience). The cognitive mapping is then very difficult to do. Our approach is to look for effective visual metaphors (*i.e.,* analogies) between *real-world* and software objects so that when displaying a such natural object one could exploit the common sense understanding of the user. The difficulty of this approach is that the common sense expectations about the displayed world should also apply to the world of software objects. To give a simple example, the difference in size of two displayed objects should reflect a difference in some size metrics of the associated software objects. In summary if, at it is often said, a good picture can replace a thousand words, this is because all aspects of the picture may convey information: sizes, distances, relative positions, colors, shadows, etc. Then, as soon as the displayed objects resemble common objects, then the user begins to have visual expectations about the object.

To solve this common sense representation problem for software objects we have to address both the small scale level of individual objects and at the large scale level of communities of objects. On the small scale this concerns the shape of the visual objects according to the information we want to convey on an individual level. On the large scale this concerns the distribution of large communities of objects in the visual space so that the relative visual feature of objects convey information on the group and peculiar feature of some objects would be easily detectable.

Our work has been pursued in the following directions:

- visual metaphors for software objects;
- mappings between software metrics and features of visual objects;
- ways to display large communities of software objects;
- navigation techniques in this visual world.

Moreover a software environment has been developed under Eclipse so that visual metaphors can quickly be realized in an easily navigable virtual space.

2 Visual Metaphors

Visual metaphors should represent structural and evolutionary metrics of an entity known from our daily life as glyphs. Such glyphs can be real-world things such as houses, cities, streets, cars, or other physical concepts that exhibit a clear understanding by a person. We experimented with some of these concepts and mapped software metrics onto the glyph representation of the concept. For the metric mapping we used evolution metrics similar to the ones described in [23]. For the visualization we applied Polymetric Views by Lanza *et al.* [17]. Within the EVOSPACES project we focused on the usefulness of the third dimension and improvements with respect to the comprehension of a visualized software project.

For each software project we can define the mapping to the visual metaphors (glyphs) individually, therefore, enabling to reflect particular characteristics such as certain metrics or evolutionary changes. We exploit the concepts of metric clusters and metric configuration.

2.1 Metric Clusters

Metric Clusters are defined as a set of specific metrics that in combination enable analysis of particular software entities (see Table 1). Pinzger [22] uses a similar concept to build characteristic views on source code evolution. According to that work, a combination of meaningfully clustered metrics can facilitate the comprehensibility of a software visualization.

Hot-Spot Metric Cluster	Provider-Consumer Metric Cluster	Structural Metric Cluster	Evolution Metric Cluster
Lines of Code	Fan in	Cyclomatic Complexity	Fan in
# of Functions	Call in	Growth rate	Growth rate
Cyclomatic Complexity	Fan out	Lines of Code	# critical Bugs
Halstead Program Diff.	Call out	-	Change rate
-	-	-	Lines of Code

Table 1. Example of several metric clusters

A Hot-Spot Metric Cluster, for example, combines *number of functions*, *lines of code*, *Cyclomatic Complexity* [20] and *Halstead Programm Difficulty* [13] into one particular metrics cluster that accentuates complex software components that exhibit a variety of functionality. Metric Clusters in EVOSPACES help define presets for the metric mappings that can be used independently from the different visualizations.

The reason why Metric Clusters are a powerful concept becomes clear when we want to combine analysis data from completely different analysis methods. We can simply build a new Metric Cluster out of the freshly analyzed data and visualize this context according to our needs.

2.2 Metrics Configuration

The second concept used to facilitate the mapping of metrics with their glyphs is a Software Visualization Mixer (SV-Mixer). With the SV-Mixer we adopt the concepts of an audio mixer for software visualization. The idea is to map particular software metrics to the visual metaphors, like an audio mixer processes audio signals before sending the result to an amplifier. The metric values can then be filtered, normalized or transformed according to the SV-Mixer's configuration before composing a visualization. This allows us to quickly adjust the visual mappings according to our focus while exploring the view.

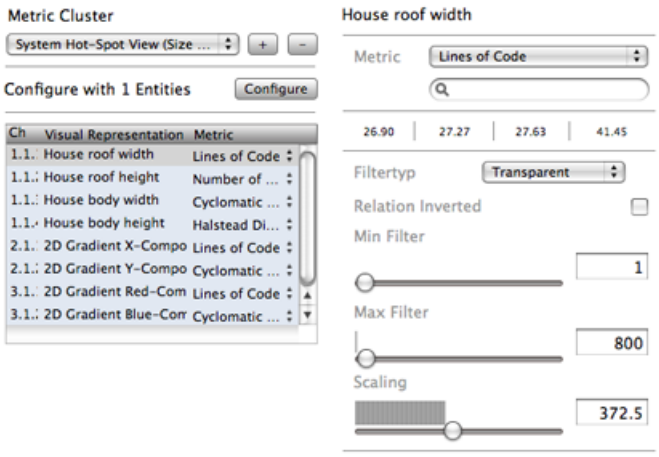


Fig. 1. SV-Mixer: on the left is the general Metric Cluster selection, on the right the editable entities per channel.

2.3 Cognitive Glyphs

Glyphs are visual representations of software metrics. They are generated out of a group of visual components that we call visual representation component. Each visual representation component corresponds to a channel in the SV-Mixer. A Metric Cluster in the SV-Mixer is therefore mapped to the visual representation components used for a particular glyph. As such, the value of the mapped metric specifies the value of the glyph's visual representation. Cognitive glyphs are visual representation components that depict common real-world objects in a particular metaphor. For example, houses in a city could represent classes or packages in a software system. The different dimensions of a house would be represented by software metrics.

Fig. 2 depicts such a house, where the the roof width corresponds to the number of functions, the roof hight to the lines of code, the body width of the house to the Halstead program difficulty, and the body hight to the cyclomatic complexity measure.

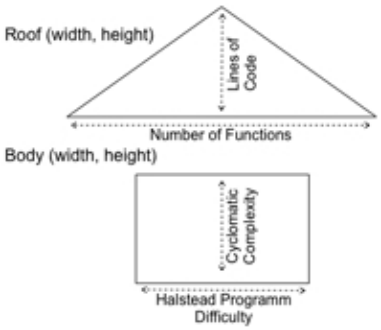


Fig. 2. SV-Mixer metrics mapping on glyphs

The software city of a particular software project then consists of such houses, all of the same metrics cluster mapping to house dimensions. This allows us to cognitively compare houses in a software city and find outliers or somehow special houses or districts of houses. With that software is visualized in a more comprehensible way leading to a focus on the relevant aspects compared to glyphs that are not based on a metaphor (*e.g.,* Starglyphs [11]).

To build a house glyph, for example, we can use four parameters together with their metric mappings. Two metric can than be mapped to represent the width and height of the roof, where as the other two metrics are mapped to the width and height of the body of the house. For these 4 size dimensions of a house we can use different Metric Cluster mappings and therefore visualize different aspects of a house.

In the context of a Hot-Spot Metric Cluster the example in Figure 3 a) represents a complex class, visualized with a large house body, and a comparable small house roof width (*number of functions*) and a medium to large roof height (*lines of code*). The house glyph shows a software component that has reasonably-sized complex code on a few functions like, likely a class implementing a complex algorithm. Such houses would be considered problematic candidates to maintain and evolve.

With cognitive glyphs a viewer can distinguish a well-shaped glyph from a miss-shaped one rather quickly. However to intuitively provide orientation about well-designed aspects and distressed ones in the software system, we need to accurately normalize metrics.

For this purpose we implemented a concept to normalize the mapped metrics to each other. In the context of the SV-Mixer the idea is to do the normalization similar to an audio mixer's pitcher. The pitcher stretches or condenses different audio sources to bring them in tune. Similarly we tune the visual representation values of the mapped metrics to meet a specific project's needs. With the normalization the mapped metrics are customized to represent a well-shaped piece of the software as a well-shaped glyph. As with a pitcher the mapped metrics normalization factor is adjusted with a slider. In the default configuration a linear function is used, but non-linear regression would fit as well for normalization. The default normalization value ($scalV$) for each metric is

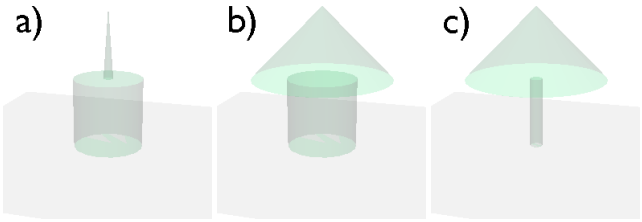


Fig. 3. Cognitive Glyphs showing two miss-shaped a), c) and a well-shaped house b).

calculated with the minimal ($minMV$) and the maximal value ($maxMV$) as well as the current scaling ($curSV$) slider value of the mapped metric:

$$scalV = minMV + \frac{(maxMV - minMV)}{curSV}$$

The concept of normalization further *enables comparability of projects through visualization even if the analyzed context differs substantially*. This is the case since the normalized cognitive glyphs represent a similar well-shaped glyph from a well-shaped piece of the software.

2.4 Code smells

Instead of mapping metrics directly to cognitive glyphs, we can provide a user with extended information based on a code smell analysis of the entities of interest. Different approaches exist to find code smells or anti-patterns in source code. In our work, we use the approach developed by Lanza and Marinescu in [18]. Based on the detected code smells the glyphs would be visualized differently. This can be done with a traditional approach such as coloring the detected entities according to a color concept (see Figure 4). The bottleneck in coloring the entities based on their detected code smells is that we cannot apply another color concept to the data set. This is even worse whenever an entity includes two or more different code smells, we would need to color the entities with a default color. However, a default color leaves us with just little, and imprecise piece of information. We still would need to further dig into the entity to see which code smells were detected. On the other hand we can use various different cognitive glyphs based on the code smell analysis result. The advantage is that the combination of code smells would be visible as a further glyph.

2.5 Releases and evolution

With cognitive glyphs we visualize time in two different but convenient ways: First, we present the glyphs of all the releases of a software component, laid out accordingly on a time axes to depict whenever relevant changes have happened. Second, we visualize the components as a set of visual snapshots, where we then can switch through those snapshots and get a small animation like presentation showing the key changes between releases.

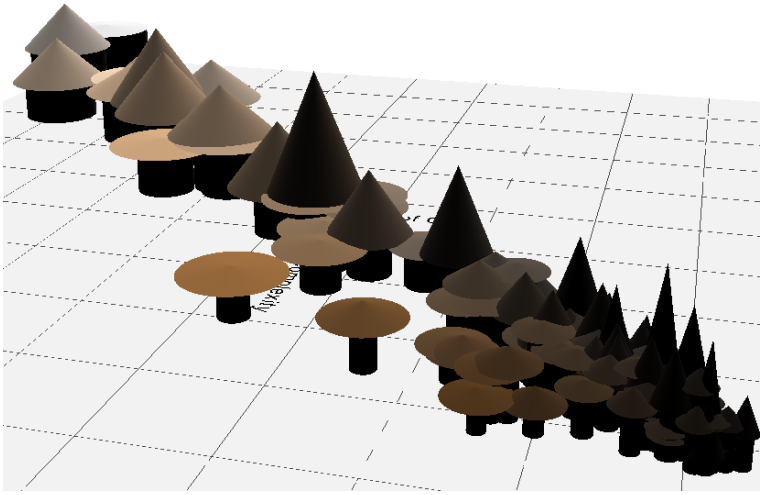


Fig. 4. House glyphs showing a set of classes of a commercial web application framework using the Hot-Spot metric cluster mapping; the brighter the color the higher the god class potential

As changes to components between releases are harder to perceive in the snapshots and an all-in-one view offers more comparability, we typically prefer the first approach on small data sets.

2.6 Tagging glyphs and visualization states

To preserve visualization states or remember interesting glyphs for later analysis we implemented a concept for tagging. This is convenient since during the navigation within a visualization, relevant aspects are spotted and need to be remembered before proceeding with the software exploration. The remembered aspects can then later be analyzed or shared within the working group. Furthermore such a tagging functionality is useful whenever one wants to switch to other visualizations and likes to further examine the spotted components from another view.

3 Code (as a) City

To provide some tangibility to the abstract nature of software systems, we experimented with the various metaphors presented in Section 2. Eventually, we settled on a city metaphor [33], because it offers a clear notion of locality and orientation, and also features a structural complexity which cannot be oversimplified. These advantages led to adoption of the city metaphor, described next, in the project's supporting tool.

3.1 The City Metaphor

Although the idea of using a 3D city-inspired metaphor to depict software systems has been explored before [2,7,15,16,19], our solution provides a good compromise between level of detail and scalability. According to our domain mapping, a software system is visualized as a 3D city, in which the buildings represent the classes of the system, while the city's districts represent the packages in which the classes are defined.

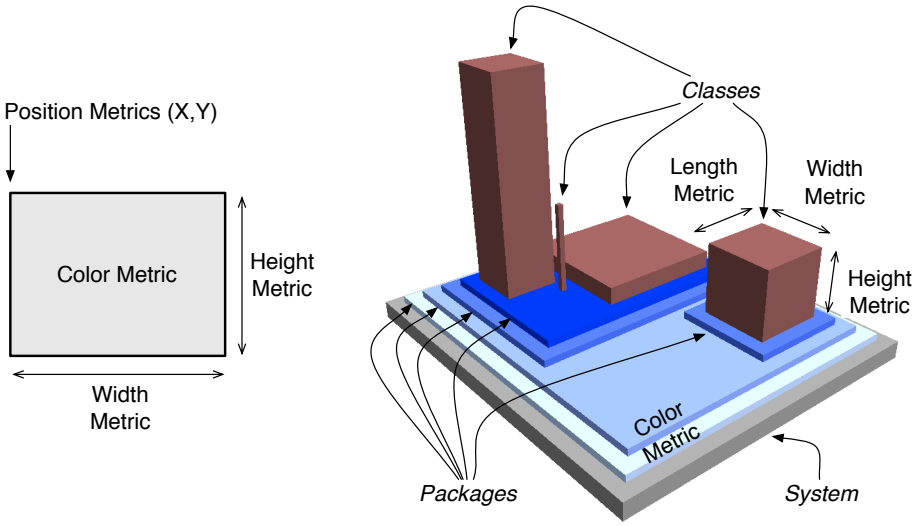


Fig. 5. Our city metaphor (right) drew inspiration from the polymetric views (left)

The visual properties of the city artifacts carry information about the software elements they represent. The set of properties able to reflect the values of the chosen metrics is composed of the three dimensions (width, length, and height), the color, and sometimes the position. We extended the idea of the polymetric views [17] to 3D. Apart from the familiarity of the city metaphor, its main benefit is that it visually represents the software entities in terms of a number of metrics in the context of the system's structure (*i.e.,* given by the package hierarchy). By combining this domain mapping with property mappings [32] and efficient layouts we provide the viewer with an overview of the structure of a system, while pointing out the outliers.

3.2 Exploring a City

Figure 6 presents the CodeCity [34] representation of ArgoUML version 0.24, which is a 137 kLOC Java system with about 2,500 classes and 140 packages. We mapped the classes' *number of methods* (NOM) metric on the height of the buildings and the *number of attributes* (NOA) metric on their base size. The districts of the city are colored according to the nesting level of the package, based on a blue color scheme: the higher the nesting level of the package, the darker the shade of blue of the district.

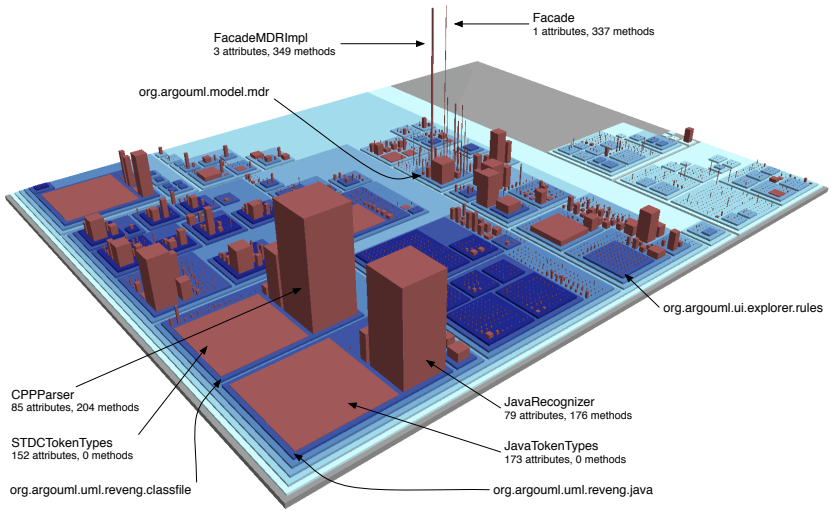


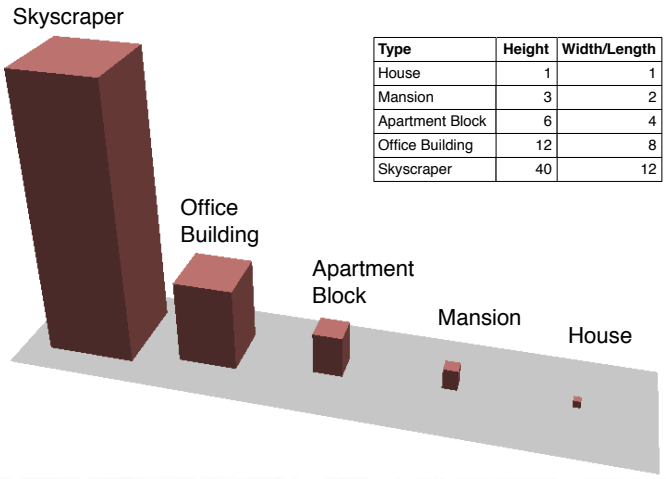
Fig. 6. The city of ArgoUML, release 0.24.

This overview gives us a comprehensive image of ArgoUML system. The city is composed of one large district, which is ArgoUML itself and two other districts (at the upper-right periphery), representing the subset of the java and the javax libraries used by ArgoUML. In the main district we see two skyscrapers, which represent the interface Facade and the class FacadeMDRImpl implementing it, both in the package org.argouml.model.mdr. The two are the absolute outliers in terms of the number of methods metric, which is visible from the very first look. There are two other pairs, both made of one massive building and one flat and wide building, which provide the parsing functionality for Java and C++ code. These pairs are part of ArgoUML's mechanism for generating source code from UML diagrams. The two buildings that look like parking-lots are containers for constants, while the two enormous ones have large amounts of functionality and also many attributes, which classifies them as potential *God classes* [24]. After our eyes get used to the outliers, we see some uniformly built districts, such as the one of the package org.argouml.ui.explorer.rules, in which the buildings are all of the same size. This happens because the majority of the classes in that package are concrete implementations of an abstract class which defines 4 methods, and all of them implement exactly 4 methods, a case depicted by buildings of equal heights.

3.3 Property Mapping Policies

The most straightforward mapping of software metrics onto visual properties is the *identity mapping*, *i.e.,* one that uses the identity function. In this case, the visual properties accurately reflect the real magnitude of the software systems in terms of the chosen metrics. However, the large variety of sizes cannot be easily processed by humans, as it has been assessed that humans can preattentively process a limited number of different sizes [12].

To address this issue, we looked into representing only a limited number of building types. In Figure 7 we present the types of buildings with their assigned sizes, in which the unit is "storey", *e.g.,* an apartment block is a six-storeyed building, and an example of their visual representations. Since we map two different metrics, it is possible to have different combinations, such as a one-storeyed house with the width and length of an apartment block: It would be a class with few methods and a large number of attributes. The metric values now need to be appropriately mapped on this building types. We designed and implemented two policies for the categorized mapping, namely *boxplot-based* mapping and *threshold-based* mapping.



Type	Height	Width/Length
House	1	1
Mansion	3	2
Apartment Block	6	4
Office Building	12	8
Skyscraper	40	12

Fig. 7. Building categories

The *boxplot-based mapping* uses the boxplot technique, widely used in statistics [29,30], which reveals the center of the data, its spread, its distribution, and the presence of outliers. Building a boxplot requires the computation of the following values: the minimum and maximum non-outlier values (*i.e.,* whiskers) and the three quartiles (*i.e.,* lower quartile, median, and upper quartile). For our mapping (see Figure 8) we use the whiskers and the lower and upper quartiles as thresholds which split the values for a software metric within a system in 5 categories, corresponding to: lower outliers, lower values, average values, upper values, and upper outliers. Two metric values within the same category, even if they greatly differ, are mapped on the same building type.

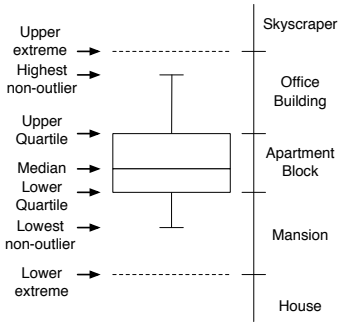
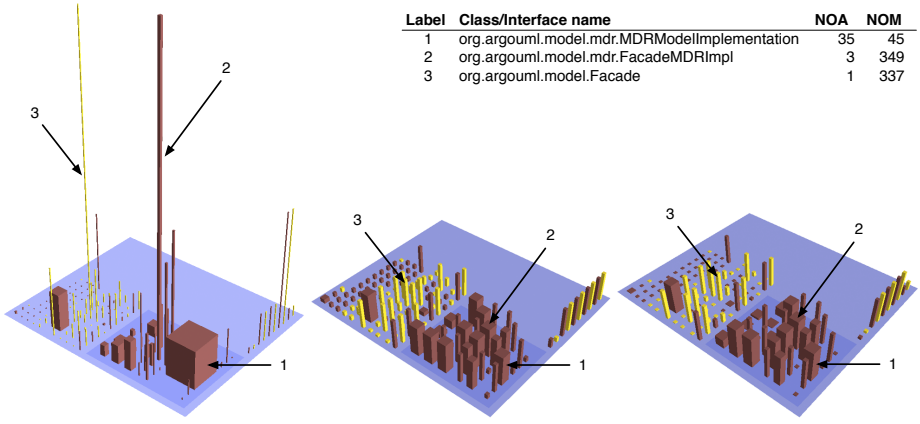


Fig. 8. Boxplot-based mapping and the corresponding building types

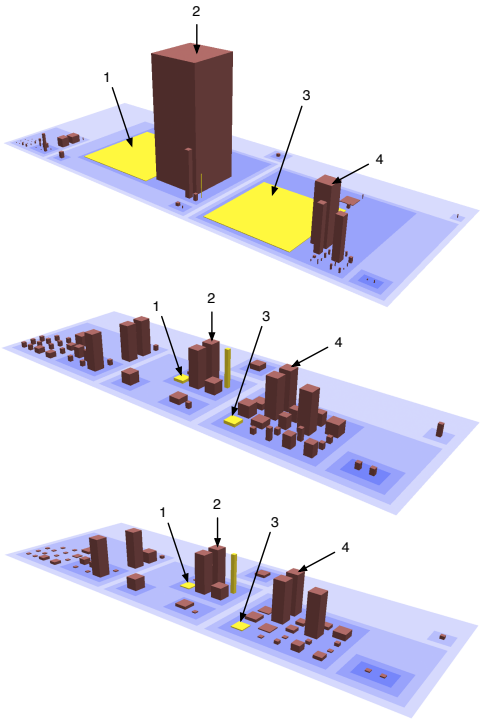
An important property of the boxplot is that the interquartile range (*i.e.,* between the lower and the upper quartile) hosts the central 50 % of the data, which insures well-balanced cities in terms of its buildings, *i.e.,* at least half of them are apartment blocks. This very property is also its main drawback: The category distribution is always balanced, which obstructs system comparison. To tackle this, we provide a *threshold-based mapping* policy which allows the definition of the category thresholds, based on statistical information. The threshold "magic numbers" need to hold across system boundaries and are thus not easy to obtain. We use values presented in [18], where the authors measured many widely different systems in terms of size, domain, and type (*e.g.,* commercial, open-source) and produced threshold values for many software metrics. For Java systems, the threshold values for number of methods are 4 (low), 7 (average), 10 (high), and 15 (very high).



Label	Class/Interface name	NOA	NOM
1	org.argouml.model.mdr.MDRModelImplementation	35	45
2	org.argouml.model.mdr.FacadeMDRImpl	3	349
3	org.argouml.model.Facade	1	337

Fig. 9. Identity (left), boxplot-based (center), and threshold-based (bottom) mappings applied to the district of package org.argouml.model (ArgoUML)

The result of applying the three presented mapping policies on the same system varies significantly. To discuss this issue, we use two examples taken from ArgoUML (Figure 9 and Figure 10), in which the interfaces are colored yellow and the classes brown, and the most extreme-sized buildings are annotated. While with identity mapping one can visually compare the actual metric values by looking at the building heights, this apparent advantage vanishes in front of the large number of different sizes which are hardly distinguishable. Another disadvantage of the identity mapping is that classes with very low metric values are difficult to spot, due to their small size. In Figure 9 we see that even on such a small subsystem, it is already very difficult to have an overview and yet be able to see the small houses. The best visibility is obtained with the boxplot-based mapping, where there is a visible balance among the categories. While the difference between the classes of the same category is not visible anymore, the very small buildings are still reasonably sized and do not risk to appear as invisible. The threshold-based mapping provides a better overview than the identity mapping and still offers a sense of the distribution of the very large and very small classes.



Label	Class/Interface name	NOA	NOM
1	org.argouml.language.cpp.reveng.STDCTokenTypes	152	0
2	org.argouml.language.cpp.reveng.CPPParser	85	204
3	org.argouml.language.java.generator.JavaTokenTypes	146	0
4	org.argouml.language.java.generator.JavaRecognizer	24	91

Fig. 10. Identity (top), boxplot-based (center), and threshold-based (bottom) mappings applied to the district of package org.argouml.language (ArgoUML)

3.4 Layout

To achieve a scalable visualization for large-scale software systems, we need an efficient layout which satisfies the following requirements:

1. does not waste the real-estate of the cities,
2. reflects the given containment relationships, and
3. takes into account the metric-dependent dimensions of the buildings.

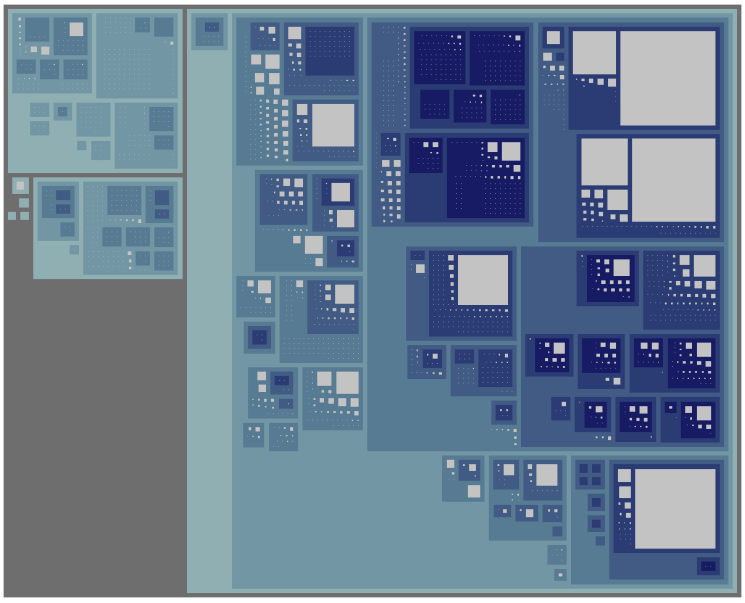


Fig. 11. A birds-eye view of the layout in the code city of ArgoUML.

Figure 11 presents the layout of the ArgoUML system, in which the buildings appear as white (for increased clarity) and the districts in shades of blue, according to the nesting level of the package. In spite of the similarities to treemaps [26], our layout must deal with fixed dimensions (as opposed to percentages) of the elements as the result of the metric mappings, which makes it more similar to a rectangle packing problem. Unlike a treemap algorithm, our layout cannot establish *a priori* how much space will it need to place the element.

We implemented a space-efficient layout, which uses a 2-dimensional *k*d-tree [3] to do the space partitioning and keeps track of the space currently covered. To chose the place for an element within the city's real estate, the layout algorithm takes into account the following criteria, ordered by priority: (1) a place which does not cause the expansion of the city's covered area and if there are several nodes, the one using the space most efficiently (the available space of the node is closer to dimension of the element), (2) a place which causes the expansion, and among these the one which produces a covered area closest to a square (for aesthetics).

3.5 Topology

We introduced the notion of topology by representing the package hierarchy as stacked platforms, thus placing the buildings at different altitudes, as can be seen in Figure 12.

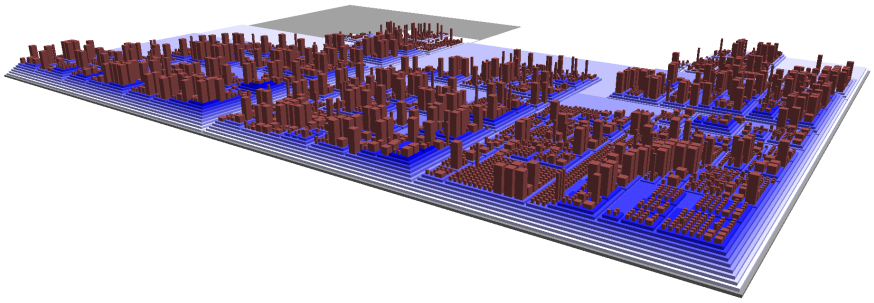


Fig. 12. Topology of the Azureus systems

3.6 Prototyping with CodeCity

We concretized the domain mapping, several property mapping policies, a number of layouts and the topology in a prototyping tool called CodeCity [34]. As a proof of the scalability and language-independence Figure 13 shows a visualization of ten systems written in Java, Smalltalk, and C++, comprising over 1.25 million lines of source code.

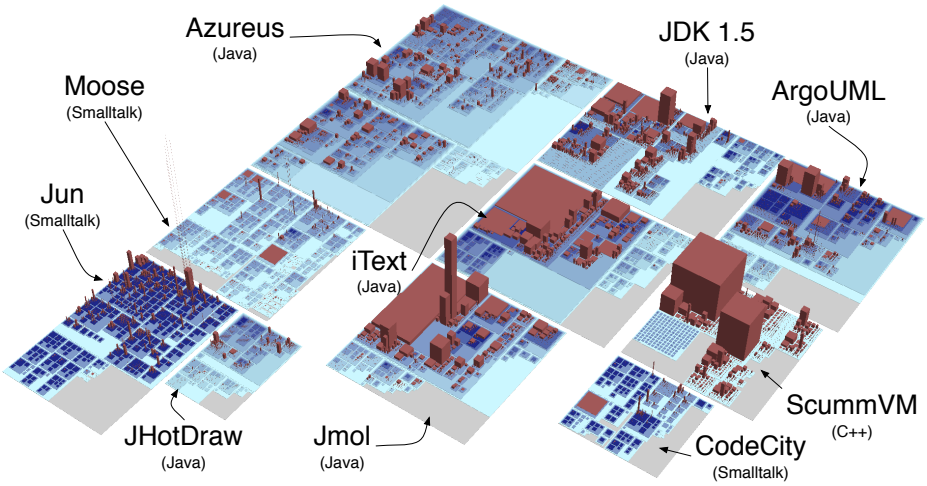


Fig. 13. A proof of CodeCity's scalability.

4 Night view: displaying the working elements of the software

All ideas presented earlier in this chapter are dealing with static data obtained from the source code of the analyzed software. In this section we will cover another part of our reverse engineering toolbox: the visualization of the execution of programs or, rather, the execution trace of the software when it executes a specific task. An execution trace is the record of all the method called when a user is working with the software. This will tell us which classes and methods are involved in the implememantation of the scenario run by the user. This complements the static understanding of the software by the maintenance engineers by showing them exactly what are the software elements involved in a specific task. In our project we used Mozilla Firefox as the test bench and the trace was recorded when interacting with this application. The corresponding data volume is huge: about seventeen millions events (including system calls).

4.1 The night view metaphor

While walking in a city during the night, if one sees an office building with lights on, this is usually associated with workers on duty. Since our project has developed the city metaphor, the night view is easily understandable: we light the buildings depending on the involvment of the corresponding class in the trace. Considering the data volume of an execution trace, it is not possible to display it without some preprocessing. Then, based on a segmentation technique we developed, we could create an animated view of the running of the software at two granularity level: macroscopic and microscopic. This visualisation is directly implemented in the EvoSpaces tool which will be detailed in the next section.

4.2 Segmentation of the trace

To cope with the quantity of information of the execution trace, the current trend in literature is to compress it to remove the redundancies. There are two categories of compression algorithms: lossy and lossless. Basically the lossless algorithms try to identify and eliminate the recurring patterns in the trace. In the compressed trace, such an algorithm keeps only one occurence of the pattern and replaces the others by a reference to the first. The simplest example of this technique is when a set of contiguous similar events are replaced by a single occurrence associated with the number of repetitions. As an example of a much more sophisticated algorithm Hamou-Lhadj and Lethbridge used the common subexpression algorithm [1, 14] originally developed to identify patterns in DNA analysis. The lossy algorithms, i.e. the ones that do not preserve the information after compression, use approximate match techniques to find the recurring patterns to eliminate. For example, the depth of the search in the call tree of methods can be bounded. In De Pauw et al. [31] half a dozen of such lossy techniques are presented. Since our work focuses on identifying the classes or files that are specific to the implementation of a given business function and the way it is implemented, our concern is less to find general techniques for trace compression than to actually ”see” the involvement of classes and files in the trace as time passes. However, it is clear that one cannot present the sequence of building illuminations in our city metaphor according

to each event in the trace. This sequence would take hours to display. Therefore, we developed a segmentation technique to summarize the information. The key idea is to split it the execution trace into contiguous segments of a given duration (width). Next, we analyze each segment to extract some information about class occurence (fig. 14). Finally, we present the information found as a sequence of images. The results can then be displayed as a movie, each image (frame) representing one segment of the execution trace.

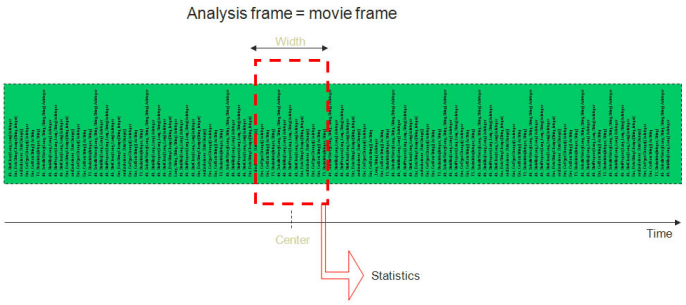


Fig. 14. The segmentation of the trace

In figure 14 the text displayed vertically represents the events (method calls) in the trace. Starting from this segmentation idea we defined two representations for the trace: - The macroscopic view that represents the trace as the sequence of segments - The microscopic view that represents the sequence of events in a selected segment.

In the macroscopic view, one computes the number of occurrences of each class i.e. the number of method executions (events) of the class in each segment. Next, this number is associated to one of the three categories of occurrences we defined: low, medium or high. Each category is mapped to a specific colour and the limit bewteen the categories are user-defined. For example, the occurrences of a given class below 10 could be displayed in blue, the occurrences between 10 and 49 in green and the occurrence above 50 in red (fig. ?).

Finally, each building in a ”frame” (image) is illuminated using this colour. The resulting ”movie” is obtained by the sequential display of the illuminated building for each segment. With this representation we can observe, by looking at the whole landscape and the colour of the buildings, what are the active ”regions” of the software (i.e. packages, subsystems) at any moment in time and what are the most active classes in each of these regions. In the microscopic view the time scale is reduced to a single segment as shown in figure 15. Then we display sequentially each method called in each of the classes by drawing solid lines between the buildings representing the classes (fig. 16). While the increment of time in the macroscopic view is one segment, the increment of time in the microscopic view is the elementary invocation of a single method call (event). It is worth noting that the buildings keep the same colours in both views to show the most active classes in the segments.

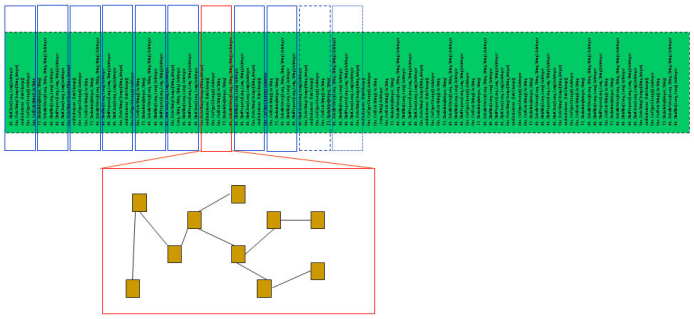


Fig. 15. Events belonging to a segment

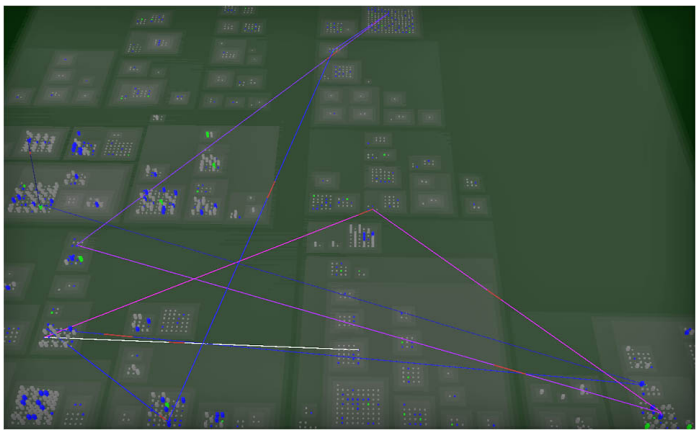


Fig. 16. Microscopic view

4.3 Filtering uninteresting classes

Since we want to understand the role of the classes through the visualization of the program execution, we must be able to associate the classes to the externally observable behaviour of the program. Therefore since this behaviour changes while the program is running, we naturally focus on the classes that occur (are involved) in specific locations of the trace. In other words, the classes that are evenly distributed throughout the trace cannot be linked to some specific business task. In fact these classes often have a transversal responsibility implementing some aspect of the software like logging, security check and the like. By analogy with signal theory, these classes are considered as noise and must be filtered out. Traditional frequency analysis work defines a class frequency simply as the number of times the class appears in the execution trace [8,28]. However, to identify the noise, we cannot simply count the occurrences of some classes but we must know where these occurrences happen in the trace. For example, a class that is found 10'000 times in the first 20% of the trace will be considered differently from a class that is found 10 times in 1000 evenly distributed locations. A class whose

location is evenly distributed throughout the trace is called temporally omnipresent [21]. To interpret this idea in the Software City metaphor we could say that a building that stays illuminated all night long does not provide much information about when exactly people are working in the night. By removing the temporally omnipresent classes, we greatly simplify the visualization of the trace on both the macroscopic and microscopic view.

5 EvoSpaces: implementing best ideas in the same prototype

Concepts and metaphors described before are all developed in separate prototypes specific to each group of research of the EvoSpaces project, this allowed more freedom to each group to try ideas faster. But the best ideas must at some point in time be stabilized to be integrated into the same integrated prototype. This section will first present how the platform is build, then how the ideas presented earlier have been implemented to EvoSpaces. In a second part, the interactions with the environment and tool specific feature will be presented.

5.1 The platform's architecture

To allow our tool to display systems written in different programming paradigms and to be able to quickly integrate new visualization metaphors in the tool's architecture is made of five layers. Thanks to the well-defined interfaces between layers, changes made inside a given layer will have a limited impact on the other layers. For example, the rendering engine will not be affected if one analyzes a system written in a new programming language, provided it follows the same paradigm (object oriented for example).

Source code layer

This layer represents the raw source code of the software under investigation structured as files. Those files are parsed off-line to fill the database of code elements. Since we do not know at parsing time what information the users will look for, we have chosen to extract as much structural information as possible from the source code. Moreover, a set of widely used software engineering metrics is computed for the target system while the database is loaded. This layer is also used to retrieve the source code of some selected element on the screen, when this code must be displayed.

Database layer

At the database level, the source code elements are represented as tables and relationship. Elements like classes, methods, variables, attributes, packages, files or modules are mapped to tables. The way those elements are structured (through programming language constructs), communicate or work together is represented as relationships. The database contains one table per software entity and one table per "relationships" between software entities. Therefore, in the database, the software under investigation

is modeled as a huge entity-relationship diagram like the one presented in figure 16. Moreover, all entities and relationships have extra properties like source level information (names, labels, parameters) and metrics values. These properties depend on the element or relationship considered (fig. 17). The classes in the database layer implement a generic access to the tables. Basically they consists of ”builders” [9], that instantiate the objects representing entities and relationships.

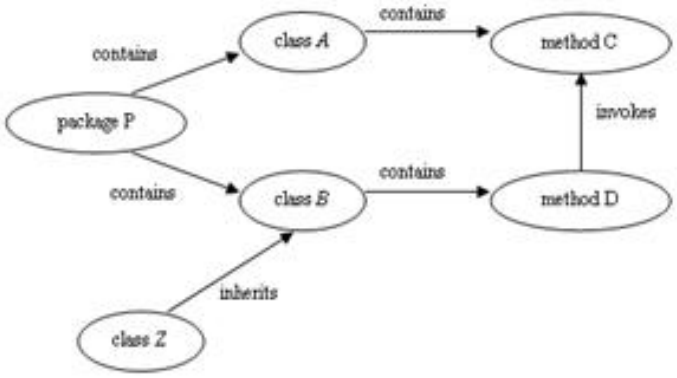


Fig. 17. Source code data structure

Model layer

The Model layer implements the object representation of the loaded entities and relationships in the EvoSpaces system. Each kind of entity or relationship is represented by its own class. Consequently, this layer contains two hierarchies of classes, one for the entities and one for the relationships, following the Famix metamodel for objectoriented programming languages [25]. For example, in figure 18, we show the Famix model for entities.

ModelView layer

This layer is the first to deal with visualization issues. It contains all the values and parameters used for the 3D rendering of the entities and relationships. Each object in the ModelView layer has a counterpart in the Model layer. Then, the ModelView layer contains two hierarchies of classes, one for the entities and one for the relationships that are similar to the hierarchies in the Model layer. However the entities and the relationships of the ModelView layer only contain displayable data. The ModelView layer works as a visual abstraction of the raw data stored in the model layer: it maps the data of the model layer to displayable elements. In particular, this is where: - The glyphs (graphical objects representing data through visual parameters [6]) are mapped to a given type of

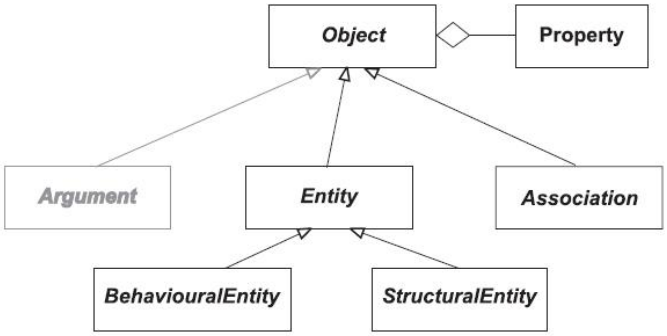


Fig. 18. The Famix modell for entities

entity or relationship - The values of the metrics are mapped to some visual scale (saturation of colors for example) and positions in the 3D space (layout) Moreover since we wish our system to let us experiment with different representations of the same set of entities and relationships, the same entity or relationship in the Model layer may be mapped to different visual objects in the ModelView layer. This makes it possible to maintain several concurrent views of the same set of software elements. However, at any given time, only one view will be displayed. This layer, which represents the largest part of our system, also contains the classes that control the interactions with the user. Any entity or relationship in this layer is associated with four objects: a shape, a color, a layout and a list of ”reactions” to user actions. The ”shape” defines how the element will look like in the 3D view (the glyph). It can be as simple as a fixed size geometric volume, like a cube or a cylinder, or be a much more sophisticated visual element, using transparency effects and textures. The dimensions of the glyphs are set proportionally to the value of one or more metrics. Since we adopted the City metaphor to represent software entities, the classes and files are represented as buildings and the relationships as solid pipes between the buildings. The visual features of the buildings (size and texture) are set according to the values of user-selected metrics. However, we soon realized that a linear mapping from metrics to visual features was not useful since users have a great difficulty to compare close visual feature, like a little difference in size or color. Therefore, we decided to present any visual feature in three, easy to distinguish, categories. Each category is mapped to a user-defined range of metric value. For example, the texture represents three different kind of building: house, apartment block, office building. Then we can map a given texture to a file according its number of lines of code (LOC). Since C++ header files (.h) are a special kind of file, we decided to map it as yet another texture: the city hall with columns. Another metric can be mapped to the height of the buildings, chosen among three categories: tall, medium and small. To enforce our City metaphor, we represent the functions and procedures inside the files or classes as stickmen of different colors (to distinguish function and procedure). In fact, these represent the ”workers” in the buildings. Each stickman is surrounded by yellow boxes (its resources) representing the local variables used by the procedures or functions. Once visualized, the user can interact with the displayed objects. Each visual

element owns a list of potential actions that the user may perform on it. For example, the user could request to display the value of some metric for the object, to change its visual appearance, to display the related elements, to display the corresponding source file, etc. The list of possible actions is defined for each type of elements and is accessible through a contextual menu.

Rendering layer

All the mechanism responsible for the actual display of the views on the screen are located in the Rendering layer. The 3D rendering library used is JOGL [27], a binding of OpenGL for Java, which has been released by Sun for Windows, Solaris, Linux and Mac OS platforms. The rendering engine, which is responsible for the drawing of the 3D scene on the screen, uses the data stored in the ModelView objects. This engine also catches the actions of the user and executes the corresponding operations.

5.2 Interaction with the visual objects

Our investigations on the interactions modes with the tool went along three directions. First, we studied the way to display the information retrieved from the database. Second we investigated the ways to dynamically change the viewing parameters of the entities and relationships in order to find the best metaphors for the software elements in given situation. Third we investigated the navigation among the displayed software elements. As a result we implemented a context sensitive menu in the 3D space. As an example, all the metrics available for a given object can be displayed from its contextual menu. In this case, a new window opens with the list of metrics together with the values for the object. Since the kinds of relationship between the software elements are numerous, the user can select the one he wants to display in a preference window. It will be displayed as solid pipes between the associated elements. To represent the directionality of the selected relationship, we display a colored segment moving along the pipe from the origin to the destination. This gives the impression of the flow of information between the elements (figure 19). The relatiohship can also be used to navigate the City. In fact, by clicking on a relationship, the user can ask the system to move the camera to the element located at the origin or at the end of a relationship. The user can then navigate the City through the connected elements. finally, to set up and orient the camera in the 3D scene the user can use the buttons in a navigation panel or use their mouse and keyboard counterpart. Since the objects representing the files or classes can contain other objects (methods and variables), the user can zoom into the objects to display their contents. Then, the walls of the buildings are removed to display its ”workers” (stickmen).

5.3 Implementation of the research groups' ideas

City view

The buildings in the city are grouped together by packages. The buildings are sized depending on the value of two metrics selected by the user, the first for the kind of the

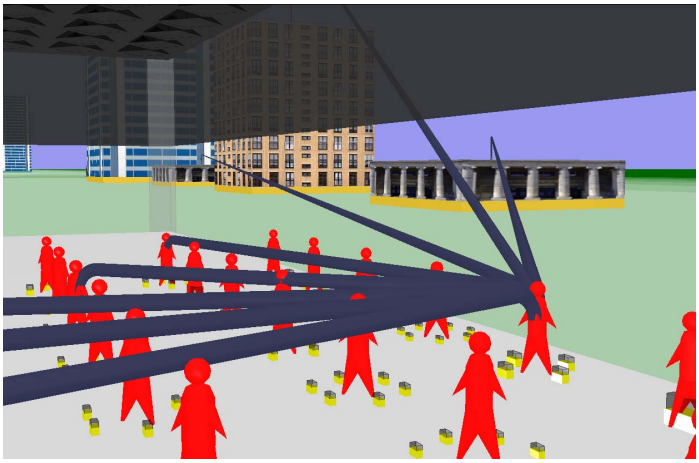


Fig. 19. The animated relations between entities

building and the second for its size. Figure 20 shows how the city in EvoSpaces looks like. Figure shows the panel that lets the user choose the two metrics to map to the kind and size of the building as well as the boundary of the metric value categories.

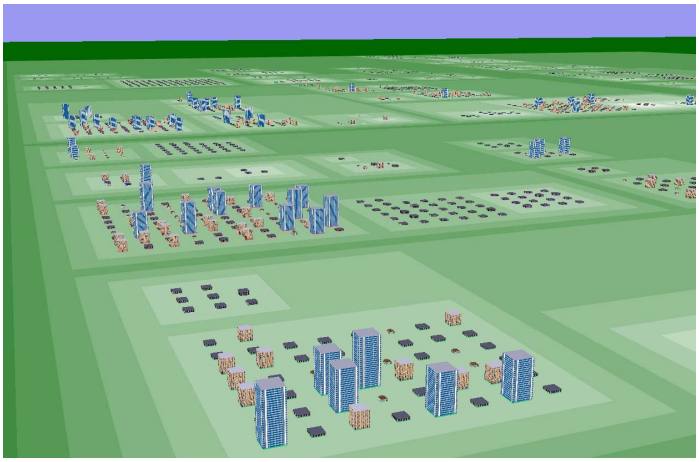


Fig. 20. The city in EvoSpaces

Archiats

To diplay the Archiat view we switch the representation of the files and classes from buildings to houses. But the elements are displayed at the same location in the 3D space

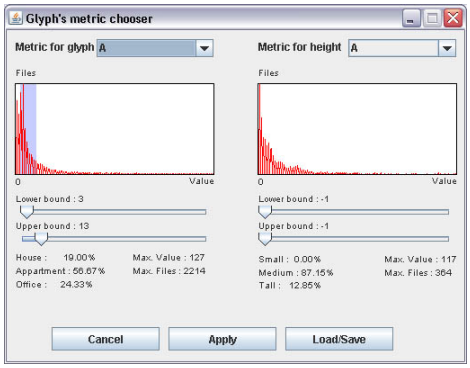


Fig. 21. The buildings' metric chooser

(figure 22). Since this representation can display four metrics at the same time, we designed a specific panel to select these metrics (figure 23). To implement the idea of the "normalizer", a scaling factor can be set for the metrics.

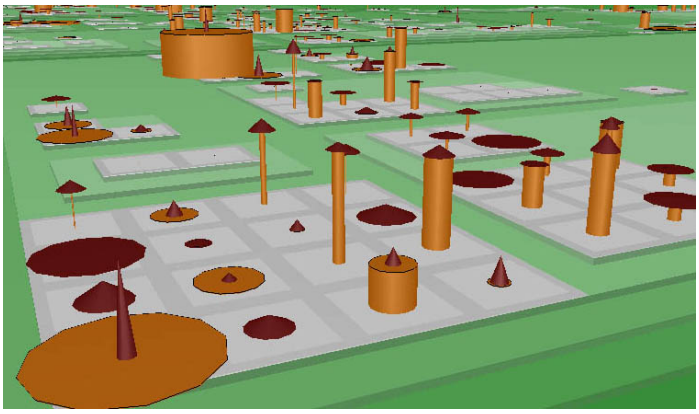


Fig. 22. Archiats in the EvoSpaces environnment

Night view

The night view allow us to display the execution trace in two different time scales: macroscopic and microscopic. Figure 24 present one frame (segment) of the macroscopic view and figure 25 presents the tool to set the threshold to eliminate noise in the dynamic display (i.e. omnipresent elements in the execution trace). In the same way as the other metrics, we group the number of calls to a class in three categories (high, medium and low). The limits between these categories are set with the same pannel.

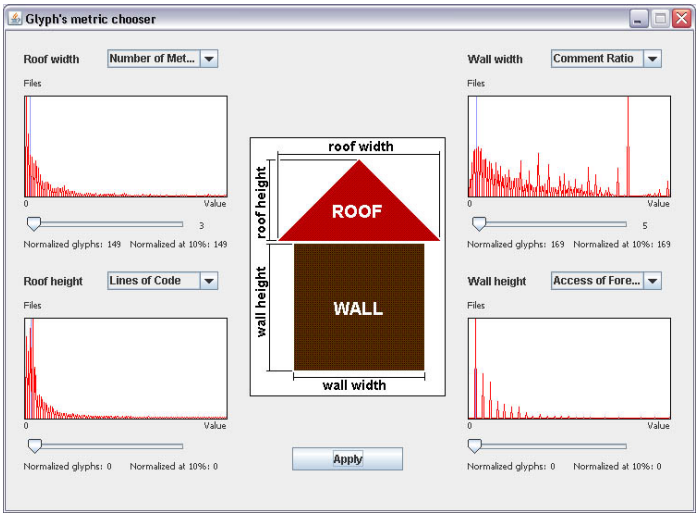


Fig. 23. The Archiats' normalizer

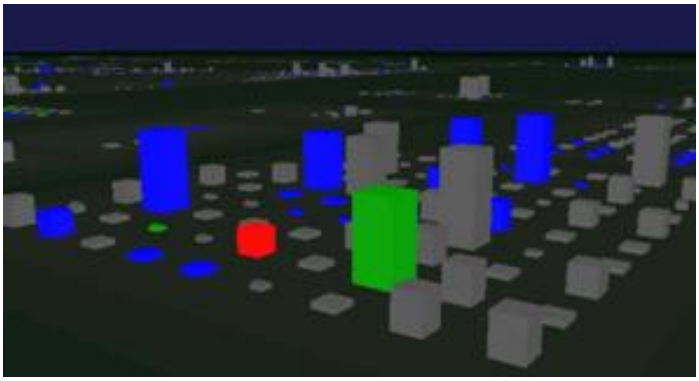


Fig. 24. The city by night, macroscopic view

To launch the microscopic view (figure 26), i.e. the display of individual calls in a single segment of the execution trace, one freezes the macroscopic view at a given segment. Then we display the calls in the execution trace in sequence as solid lines between the buildings. We can display up to ten calls at a time but with different brightness to identify the sequence of calls, the brighter the more recent in time. Self calls are displayed as rings on the elements.

6 Conclusions

With software systems becoming increasingly complex, software evolution has to be addressed in a systematic way by providing adequate means for dealing with this com-

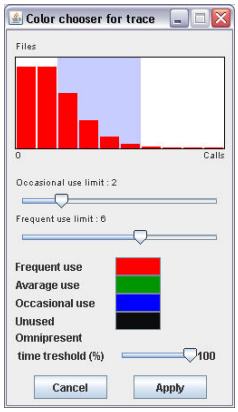


Fig. 25. The city by night's color and noise level chooser

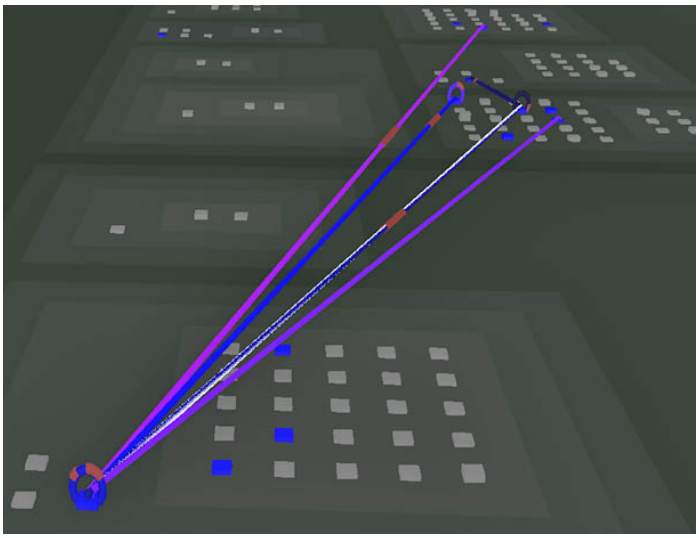


Fig. 26. The city by night, microscopic view

plexity. Multi-dimensional, multi-view visualization adopting techniques from the areas of architecting as well as multimedia and visualization will be one major technology for software engineers to get a grip on software complexity for development, maintenance and evolution of successful software systems.

Our project aimed at exploiting multi-dimensional navigation spaces to efficiently visualize evolving software systems to ease analysis and maintenance. One of the outcomes of the project is the EvoSpaces prototype implemented as plug-in to Eclipse, in which large software systems can be visualized and navigated in 3D.

This virtual world is named *software city* because we used the city metaphor to represent the virtual software entities. In a software city, the classes or files are represented as buildings displayed in districts representing the containing packages or directories. Then, the methods or functions are represented as stickmen when the inside of the buildings is displayed. The size and dimension of the buildings are set according to the value of user-selected metrics. The city gets a different shape depending on the selected metric. We can also easily filter information based on given metric values. Finally, the relationships between the classes or files are represented as solid animated pipes to show the flow of information. In this virtual world, not only the static relationships, but also the dynamic behavior can be visually represented thanks to a technique to display execution traces. The prototype has been architected so that extensions can easily be built and integrated.

As for the scientific dissemination, our approaches and prototypes have been presented in specialized international conferences and have attracted the interest of the research community (see our publications in the references section). For future work, we will extend our research to a multi-user environment where people collaborate to solve problems. This implies to develop research in two directions: (1) collaborative maintenance of software systems; and (2) metaphors and models to ease teamwork on the same system.

Acknowledgement

We are grateful to the Hasler Foundation Switzerland, who has generously supported this project.

References

1. T. A. Hamou-Lhadj. Compression techniques to simplify the analysis of large execution traces. In *Proc. of the IEEE Workshop on Program Comprehension (IWPC)*, 2002.
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. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.
4. S. Boccuzzo and H. C. Gall. Cocoviz: Supported cognitive software visualization. In *Proc. Working Conf. on Reverse Eng.*, 2007.
5. S. Boccuzzo and H. C. Gall. Cocoviz: Towards cognitive software visualization. In *Proc. IEEE Int'l Workshop on Visualizing Softw. for Understanding and Analysis*, 2007.
6. C. M. C. and E. S. G. Information rich glyphs for software management data. In *Proc. of IEEE Computer Graphics and Applications*, 1998.
7. 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.
8. P. Dugerdil and S. Alam. Evospaces: 3d visualization of software architecture. In *19th International Conference on Software Engineering and Knowledge Engineering (SEKE 2007), Boston*, 2007.

9. G. E., H. R., J. R., and V. J. *Design Patterns. Elements of Reusable Object Oriented Software*. Addison-Wesley Inc, 1995.
10. L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.
11. E. Fanea, S. Carpendale, and T. Isenberg. An interactive 3d integration of parallel coordinates and star glyphs. *IEEE Symp. on Info. Visualization*, pages 149–156, 2005.
12. S. Few. *Show me the numbers: Designing Tables and Graphs to Enlighten*. Analytics Press, 2004.
13. M. H. Halstead. *Elements of software science, operating and programming system series*. Elsevier, 7, 1977.
14. A. Hamou-Lhadj and T.Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behavior of a software system. In *Proc. of the IEEE Int. Conf. on Program Comprehension (ICPC06)*, 2006.
15. C. Knight and M. C. Munro. Virtual but visible software. In *International Conference on Information Visualisation*, pages 198–205, 2000.
16. 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.
17. M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Trans. on Softw. Eng.*, 29(9):782–795, 2003.
18. M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.
19. 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.
20. T. J. McCabe. A complexity measure. *IEEE Trans. on Softw. Eng.*, 2(4), 1976.
21. P.Dugerdil. Using trace sampling techniques to identify dynamic clusters of classes. In *Proc. of the IBM CAS Software and Systems Engineering Symposium (CASCON)*, 2007.
22. M. Pinzger. *ArchView - Analyzing Evolutionary Aspects of Complex Software Systems*. Vienna University of Technology, 2005.
23. M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proc. ACM Symp. on Softw. Visualization*, pages 67–75, 2005.
24. A. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.
25. D. S., T. S., and D. S. Famix 2.1 the famoos information exchange model. Technical report, University of Bern, July 2001.
26. B. Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.*, 11(1):92–99, 1992.
27. Sun. Java binding for opengl (jogl). `https://jogl.dev.java.net`.
28. T.Ball. The concept of dynamic analysis. In *Proc. 7th European Software Engineering Conf. (ESEC99)*, 1999.
29. M. Triola. *Elementary Statistics*. Addison-Wesley, 2006.
30. J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.
31. J. V. W. De Pauw, D. Lorenz and M. Wegman. Execution patterns in object-oriented visualization. In *Proc. of the USENIX Conf. on Object-Oriented Technologies and Systems (COOTS)*, 1998.
32. 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.
33. 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.

34. R. Wettel and M. Lanza. Codecity: 3d visualization of large-scale software. In *ICSE Companion '08: Companion of the 30th International Conference on Software Engineering*, pages 921–922. ACM, 2008.
35. M. Zelkowitz, A. Shaw, and J. Gannon. *Principles of Software Engineering and Design*. Prentice Hall, 1979.