Visualizing Software Systems as Cities

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

Abstract

This paper presents a 3D visualization approach which gravitates around the city metaphor, i.e., an object-oriented software system is represented as a city that can be traversed and interacted with: the goal is to give the viewer a sense of locality to ease program comprehension.

The key point in conceiving a realistic software city is to map the information about the source code in meaningful ways in order to take the approach beyond beautiful pictures.

We investigated several concepts that contribute to the urban feeling, such as appropriate layouts, topology, and facilities to ease navigation and interaction.

We experimented our approach on a number of systems, and present our findings.

1 Introduction

One of the possible ways to categorize software visualizations is to make a distinction between 2D and 3D. 2D visualization has been extensively and successfully explored in the past [5, 7, 8, 10, 12, 14, 19, 20, 22]. This is not the case with 3D visualizations which are still struggling to be accepted, because of a number of reasons, such as navigation and interaction problems. Despite the proven usefulness of 2D visualizations, they do not allow the viewer to be immersed in a visualization, and the feeling is that we are looking at things from "outside". 3D visualizations [3, 4, 11, 15, 17] on the other hand provide the potential to create such an immersive experience, but there has been little research in software visualization on this aspect. Existing 3D visualizations of software (such as [15, 17]), while being visually appealing, fail at producing the immersive feeling, because they lack *locality*: the objects in the 3D space can be freely moved and the viewer is allowed too much freedom of movement, leading to disorientation – one of the main arguments against 3D visualizations. We argue that to provide locality, we need to enhance a 3D visualization by means of a well-established metaphor, as this allows the viewer to embed the represented elements into a familiar context, thus contrasting disorientation.

We propose a 3D visualization which depicts object-oriented software systems as habitable [21] cities that one can intuitively explore. The buildings in the city (representing software artifacts) are positioned according to well defined rules, and thus facilitate the establishing of visual orientation to gain familiarity with the system.

As we discuss in the section on related work, this idea is not new, but we claim that it has so far been used at the wrong levels of granularity. Choosing the right level of granularity is crucial to correctly support the city metaphor. Classes are the cornerstone of the object-oriented paradigm, and together with the packages they reside in, the primary orientation point for developers. Therefore we display classes as building and packages as city districts. We enrich this approach by also mapping source code metrics onto the size and type of buildings. Last, but not least, the overall layout and topology of the city is taken into account.

We implemented this central idea and many additional concepts into a tool named *CodeCity* which allows the user to visit these software cities and interact with the represented elements.

We exemplify our approach by visualizing three large software systems, namely the Java systems ArgoUML and Azureus, and the VisualWorks Smalltalk development environment, and report on our findings.

The contributions of the paper are:

- A 3D visualization approach which creates cities that look real, due to the combination of layouts, topologies, metric mappings applied at an appropriate level of granularity.
- A description of a fully working and freely available tool that scales up to very large systems.

Structure of the paper. In Section 2 we present in detail how we support the city metaphor by providing a number of examples. In Section 3 we discuss both the advantages and the drawbacks of our approach. We then provide a description of our tool in Section 4. Finally, we present related work in Section 5 and conclude in Section 6.

This article makes extensive use of color pictures. Please read it on-screen or as a color-printed paper version.

2 Visualizing Software Systems as Cities

Many existing 3D visualizations of software are undoubtedly appealing, but for a variety of reasons they fail to communicate relevant information about a system and thus fail to support program comprehension tasks. This is because they exploit the additional 3rd dimension as yet another visual means to communicate information, quickly leading to information overload, over-plotting, and navigation problems: Software is often depicted using a graph representation of nodes and edges which float in a 3D space and the user can "fly around" in that space to focus on interesting parts. We argue that this extended freedom is counter-intuitive and leads to disorientation and thus a negative perception of 3D visualizations.

We claim that to contrast the feeling of disorientation and frustration a good metaphor is needed, because it allows the viewer to embed the visual elements in a familiar context.

2.1 The City Metaphor

After a number of experiments -which are not the focus of this paper- to discover a good metaphor we settled on a city metaphor. Since we focus on object-oriented programs, the point is to depict the constructs that need to be understood in this context, such as packages, classes, methods, attributes, and all their explicit and implicit relationships. We represent classes as buildings located in city districts which in turn represent packages, because of the following reasons:

- A city, with its downtown area and its suburbs is a familiar notion with a clear concept of orientation.
- A city, especially a large one, is still an intrinsically complex construct and can only be incrementally explored, in the same way that the understanding of a complex system increases step by step. Using an all too simple visual metaphor (such as a large cube or sphere for the whole system or each package) does not do justice to the complexity of a software system, and leads to incorrect oversimplifications: We have to cope with the fact that software is complex.
- Classes and the packages they reside in are key elements of the object-oriented paradigm and thus, primary orientation point for developers. We currently do not display the class internals, because for a large-scale understanding it is not necessary. Apart from over-plotting problems, it is also contrary to the way one explores a city: A person does not start the exploration of a real city by looking into particular houses. It is however necessary for a fine-grained understanding and thus a topic of our future research.

In short, our aim is to represent software systems as realistic cities that can be navigated and interacted with. This leads to a number of concepts that we developed and the corresponding research questions, which we discuss throughout this section by giving examples:

- *Displaying Software Metrics.* How can we depict semantic information about a software system, such as metrics?
- *Urban Layouts & Topology.* What are appropriate layouts to transmit an urban feeling?
- *Interaction & Navigation.* How can we interact with the city representation and which navigation facilities make sense in such a context?

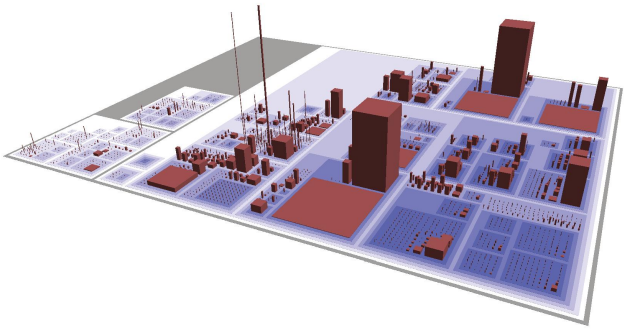

Figure 1. City of ArgoUML.

Example of a system overview. We see in Figure 1 the code city of ArgoUML, a 130+ kLOC Java system. The brown buildings represent classes and interfaces, placed in blue tiles representing the packages. The color saturation of the tiles is proportional to the nesting level of the corresponding packages. The height of the buildings represent their number of methods (NOM), while the width and length represents the number of attributes (NOA). On the left of the figure, at the far end of the city, we see two external suburbs, which represent the parts of the Java standard and the Java extended (javax) libraries that ArgoUML uses. The visualization allows us to easily spot some patterns. There are at least two massive buildings, which are potential god classes [18]. This city hosts a number of disproportional buildings, such as two antenna-shaped constructs, which represent classes with a large number of methods and very few attributes, as well as a number of buildings that look like parking lots, which represent classes with lots of attributes and very few or no methods (potential data classes). There are also a lot of small houses, which make up entire districts. The visualization is interactive and navigable using the keyboard, *i.e.,* it is easy to zoom in on details of the city or to focus on one specific district. We focus on this aspects in a latter part of the article.

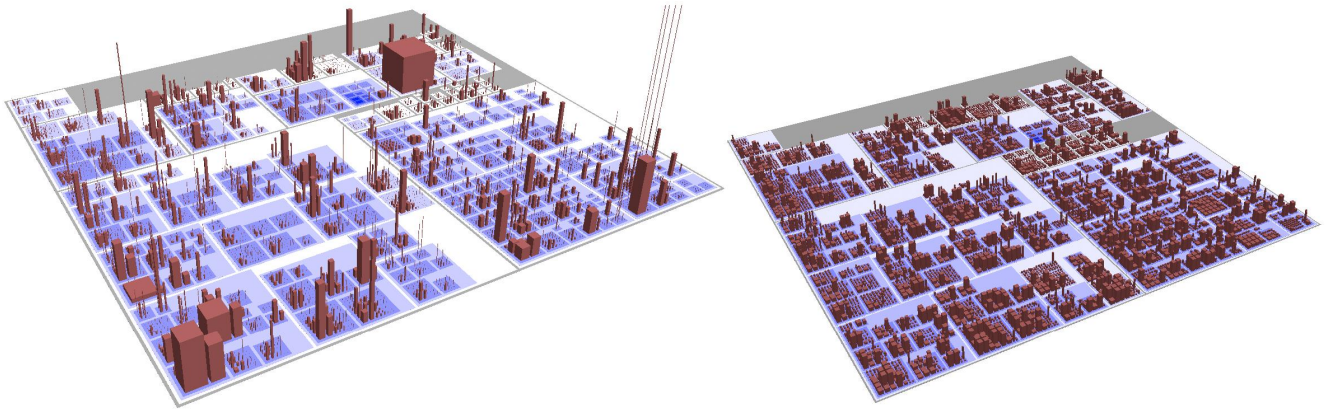

Figure 2. Linear (left) and boxplot-based (right) mappings applied to VisualWorks Smalltalk.

2.2 Displaying Software Metrics

We identified the following set of visual properties of the city artifacts that can carry semantic information about the system under investigation: dimensions, position, color, color saturation, and transparency.

In this respect we address a number of questions, such as: Which software metric to map? Which visual property is appropriate to depict a particular metric? How can we map a metric value on a visual property?

The decision on which metrics will get to influence the appearance of the city highly depends on the purpose of the visualization. In the examples presented throughout this paper, we settled on two simple metrics: the number of methods (NOM) and the number of attributes (NOA).

Intuitive mappings ease reasoning about a system. Since tall buildings are often associated with business, we mapped the NOM metric on the height to denote the amount of functionality of a class. We map the number of attributes on the size of the base of a building. Thus, a class with many methods but few attributes will be represented as a tall and thin building. In the complementary case, a class with a high number of attributes and few methods will appear like a platform (or a "parking lot"). A code city reveals these extremes, and also provides an impression of the magnitude of the system as well as of the distribution of system intelligence.

To illustrate how the mapping changes the city appearance, we chose a large software model depicted twice in Figure 2.

This model consists of more than 8,000 classes and is made of several software systems residing in the same Smalltalk namespace: Smalltalk itself, Jun (an OpenGL implementation), Moose (a reengineering environment), and our tool CodeCity.

A straightforward mapping between software metrics and visual properties of the city elements is the *linear mapping*, exemplified in Figure 2, left. The NOM metric of this system has a very wide value range (some Jun classes have well over 1'000 methods) and one consequence is that the overview of the system is compromised: The average-sized buildings appear insignificantly small in comparison to the extremely tall buildings (the latter do not fit in the figure). To see them entirely in the view would require to zoom out to the extent that the majority of the buildings are not recognizable anymore. This can be easily solved by normalizing the heights. A more serious problem is the extensive range of building sizes presented simultaneously to the viewer, which according to the *gestalt* principles [9] is more than one can grasp. To overcome all these drawbacks our cities exhibit only a reduced number of possible values for a visual property, based on a metric-based categorization. A building's height corresponds to one of 5 categories (very small, small, average, tall, very tall), according to the NOM value of the class that it represents. Choosing the boundaries between the categories can be done in two ways: The *boxplot-based mapping* computes the boundary values based on a boxplot technique, while the *threshold-based mapping* sets these boundaries according to empirically established thresholds taken from [13]. Boxplot-based mappings produce balanced cities, because the boundaries are relative to the values present in the analyzed system. The drawback is that they cannot be used to compare among cities, because a low value of a metric in one system may be high in another system. Threshold-based mapping overcomes this disadvantage, but requires well-defined thresholds for each metric. The right part of Figure 2 shows the same system with the boxplot-based mapping, looking more like a real city. For a description of the boxplot-based and threshold-based mapping, as well as for the chosen values for the categories, we refer the interested reader to [21].

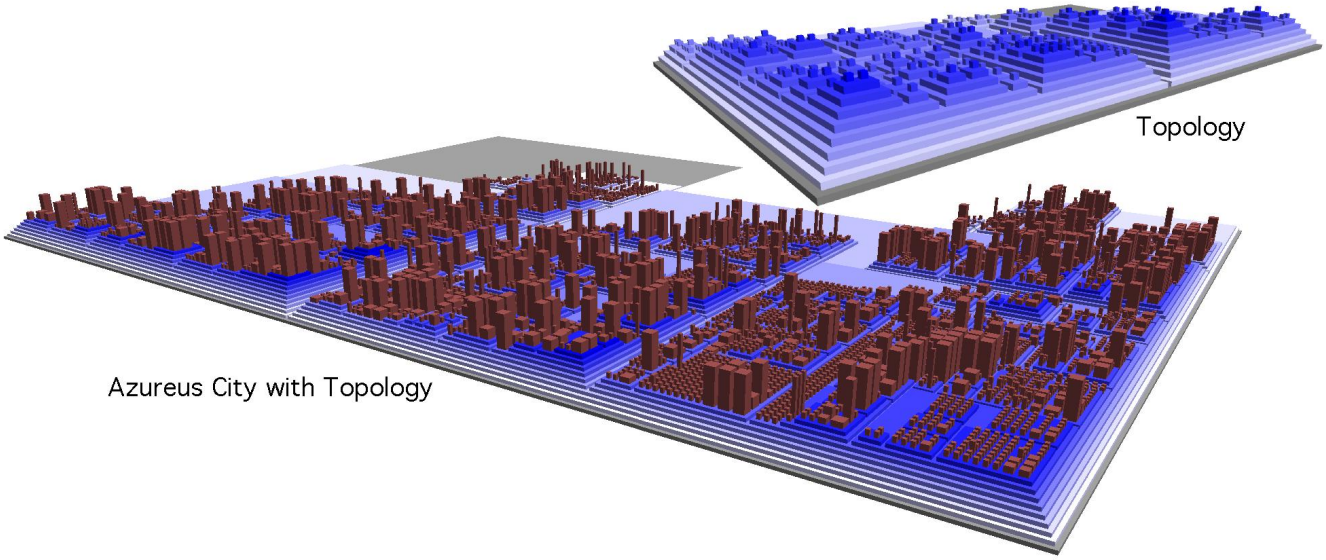

Figure 3. The city of Azureus and its package topology.

2.3 Urban Layouts & Topology

The layout of a real city is constrained by its physical evolution, i.e., the locations of the buildings have an inherent meaning. While such an observation sounds trivial in reality, in a software city there is no notion of "appropriate location" of a building, since software does not have a tangible physical presence [2].

We address this problem by placing buildings in districts, which can be themselves grouped recursively into larger districts, according to the package structure of the investigated system (see Figure 4). Due to package nesting, the layout is executed at every nesting level. Laying out buildings in a city governed by gravity (*i.e.,* buildings have their bases on the ground) boils down to a two-dimensional rectangle-packing problem, which takes a set of rectangles and places them into a rectangle of minimum area. We implemented a modified treemap algorithm. It starts with a rectangular space, large enough to host all the rectangles. We use a binary tree structure, whose root node is the total surface, to keep track of the space. Positioning a rectangle starts with finding the empty node whose size best fits the rectangle. Then, if necessary, we recursively split the chosen node until it reaches the rectangle's size. Within each district, the order in which the buildings are placed is given by their sizes (largest first). We extended the layout by introducing the notion of topology: We represent the nested packages as stacked platforms, thus placing the buildings at different altitudes. In the top right of Figure 3 we depict only the 450+ packages of Azureus, while the left part displays both the package topology and the classes.

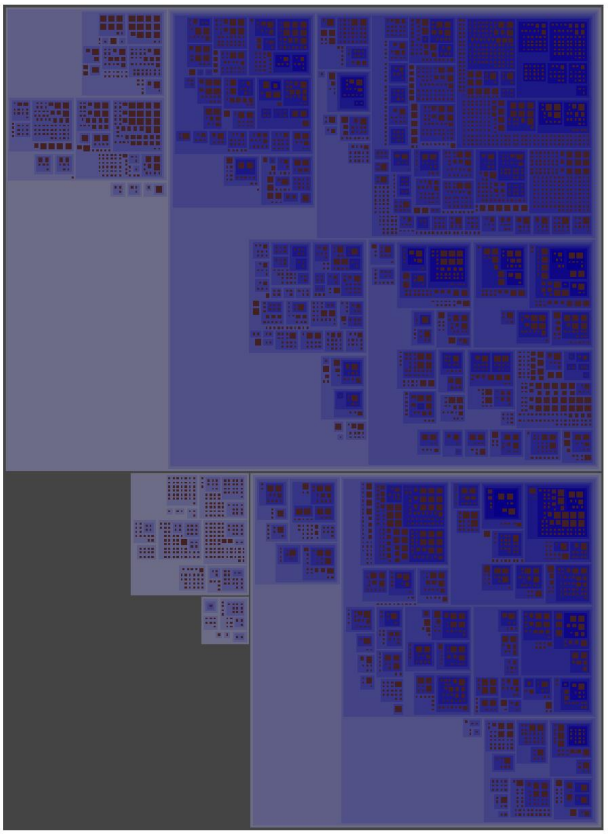

Figure 4. The city of Azureus layout.

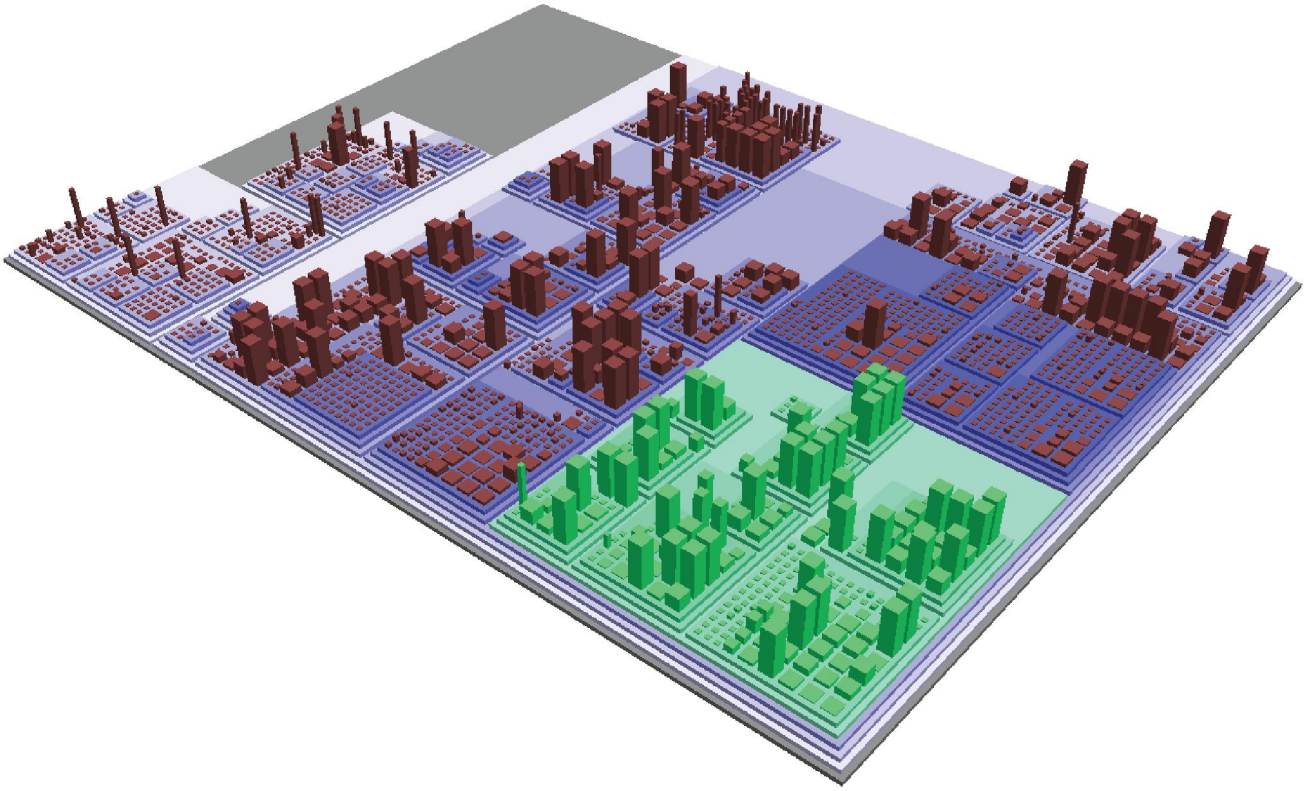

Figure 5. Selection of a district in ArgoUML.

2.4 Interaction & Navigation

Being able to navigate and interact with a visual representation is a critical feature, since static pictures are limited with respect to expressiveness. We support the following types of interactions:

- *Selection.* In CodeCity one can select any artifact or group of artifacts and interact with them. The lower part of Figure 5 shows the selection of the ArgoUML city district org.argouml.uml.diagram, appearing in a light green color. The selection can be done manually by clicking on elements, or automatically with a query engine described later in this section. With the current selection one can perform operations, such as adding to or removing from the selection, clearing the selection, and inverting the selection.

- *Spawning.* Spawning complementary views, i.e., isolating elements is useful whenever we need to focus on a particular part of the system. We can make a selection of artifacts in the city and spawn a new view that contains only the selected elements, allowing us to continue the exploration in detail. Figure 6 shows an isolated district.

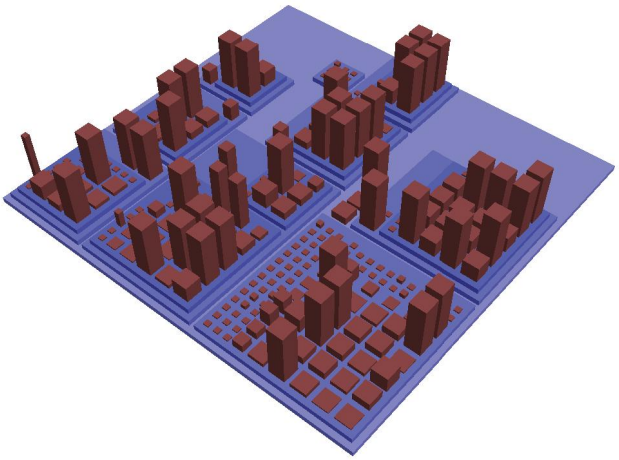

Figure 6. Spawning a part of the city.

- *Tagging.* During the exploration of a city, we may need to tag a set of buildings because we want to remember them or because we deem them as less relevant. We can assign a particular color to a selection

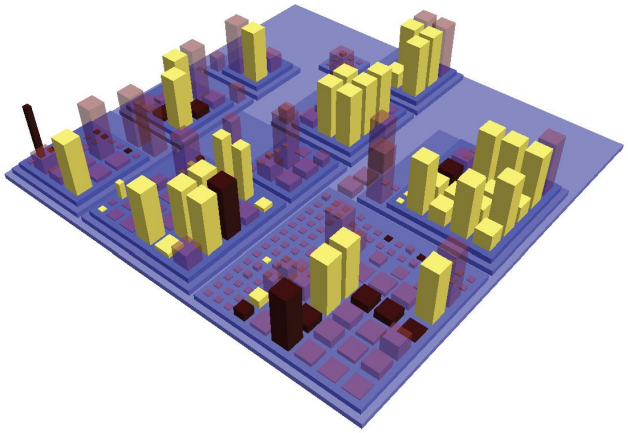

Figure 7. Color and transparency tagging.

or also use transparency. Figure 7 illustrates how one can combine these effects to emphasize particular elements, while making the rest less obvious. The two class hierarchies in the package, whose roots are FigNodeModelElement and FigEdgeModelElement both defined in package org.argouml.uml.diagram.ui, are colored in yellow and red, respectively, while the other classes' transparency is set to 60%.

- *Filtering.* Manual selection of elements can be cumbersome, therefore we provide a means to perform automated searches by indicating a set of criteria, which enable searching for artifacts that match a particular name (such as *UML*), or type (all classes of the GUI district), or category (all root classes), or artifacts related to the current selection (all sub- or superclasses, all classes related through invocations or accesses, *etc.*). We implemented a query engine to automatically search for the elements matching the query.

- *Navigation.* As opposed to some 3D visualization systems, where the viewer can rotate or move elements, our approach bears more similarity to video games, where the player is placed *within* an environment with only limited capabilities. We make a distinction between *vertical* navigation (the user gets to orbit and fly around the city, move forward or backward, change the altitude, etc.) and *horizontal* navigation. The latter corresponds to "driving around" the city, in the midst of the buildings, but with limited movement capabilities, given by the physical constraints of the world: it is impossible to pass through buildings or go below the ground (no God-mode in our approach). These two of types of navigation are *modal*, i.e., the user can only be in one or the other navigation mode.

3 Discussion

After using CodeCity for a certain time, we noticed a few aspects we want to discuss:

- *Scalability.* Because we settled our initial level of granularity to the class level, our approach scales up well in terms of the size of the system that we can display as a code city. However, in cities representing very large systems the interactivity and navigability can be substantially slowed down. Performance optimization is mandatory to increase realism. We are currently considering the use of level-of-detail (LOD) techniques to improve scalability.

- *Navigation & Interactivity.* CodeCity provides various navigations possibilities: moving back or forward, hovering left or right, orbiting around the city, changing altitude. Because of the rich semantics put inside a code city, one can interact with anything in the city. Moreover, a high number of selection queries are available, as well as the possibility to change the color and transparency of the selected items, which enables visual tagging.

- *Completeness.* The classes and the package structure provides a fair amount of information for an overview of the system. One of the things missing in our city metaphor is a proper representation of lower-level artifacts, such as methods or attributes, and depiction of the relationships between classes, such as inheritances, method invocations and attribute accesses. While we can display the relationships as directed edges, it leads to visual occlusion and decreased realism. Representing each relationship as a link is therefore not a feasible solution. For now, relationships can be visualized on demand, and are accessible as selection queries or through a direct inspection of the source entity represented by the building.

4 Tool Implementation

We implemented the visualizations presented in this paper in a tool called CodeCity (Figure 8), written in Smalltalk and built on top of the Moose reengineering framework [6], which makes it language-independent. CodeCity provides flexibility in configuring the views and supports all the three types of metric mappings we have presented. CodeCity provides full interaction with any element of the city (such as coloring, making it transparent, eliding, *etc.*). We provide a highly-flexible query mechanism to search for elements. Right-clicking any of the items brings up a context menu to perform a variety tasks, such as inspecting the model entity, accessing the represented source code, *etc.*

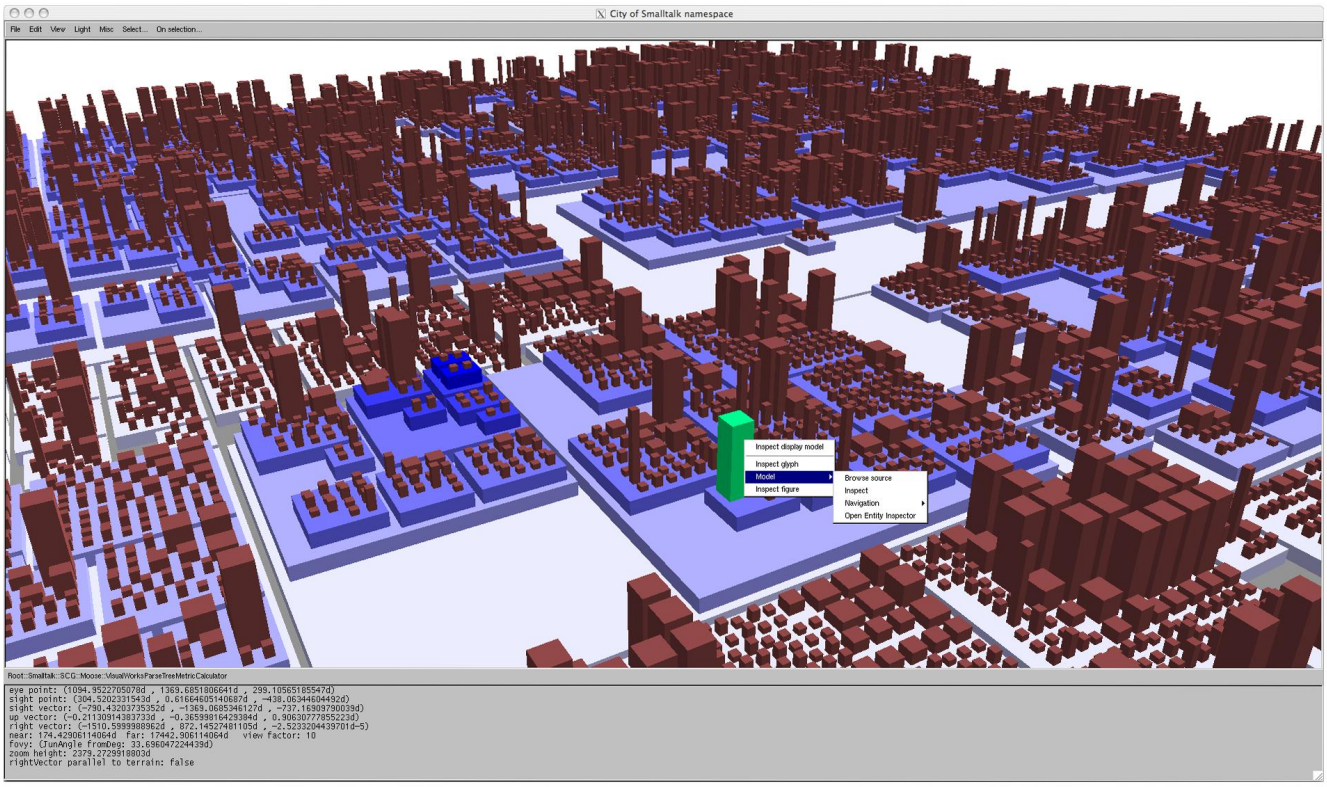

Figure 8. The CodeCity User Interface.

5 Related Work

A number of researchers have used 3D visualizations of software. While it is interesting related work, many used general 3D visualizations to represent software without any particular metaphor. Because of that we omit their discussion and concentrate on work similar to ours.

Knight and Munro [11] proposed a city representation in which a Java class is represented as a district, with methods represented as buildings. However, the authors do not discuss scalability, and their language-specific approach is not largely applicable. The visual mapping is not well chosen, leading to unrealistic cities with thousands of districts. Moreover, the authors do not exploit package information to lay out the components.

In [4], the same authors increased the granularity and applied this idea for the representation of the components in a software system and mapped semantic information (number of contained components) on the type of the building.

In [16] the authors also propose a city metaphor. In their case, the city represents a package and contains, for increased realism, non-source elements, such as trees, streets, and street lamps. In this metaphor, the program run would be represented as cars originated from different components, leaving traces to determine their origin and destination. Unfortunately, this paper presents only the ideas, supported by static rendered images without allowing interaction.

The 3D approach proposed by Marcus *et al.* [15] gravitates around poly cylinders, grouped together in floating containers. Each poly cylinder represents a line of code and they are grouped in containers representing files, which makes it not very appropriate for systems with thousands of classes and hundreds of thousand of LOC. The interaction provided by this approach is more on placing the elements in the scene (moving, rotating, scaling). Other interactions such as queries and extraction were only mentioned as future work. Moreover, in a view where the user can manipulate the sizes of the elements, one cannot visually compare them.

At a higher level of abstraction, Balzer *et al.* [3] proposed the idea of representing software systems as landscapes, which is a concept we want to look into and possibly merge with our approach.

Andrews *et al.* [1] produced 3D visualizations of file systems with a similar layout and topology. Files are depicted by equal-size cubes, colored according to a characteristic of the file, such as file type or age.

6 Conclusions

In this paper we presented a 3D software visualization approach based on a city metaphor. The classes are represented as buildings of the city and the packages as its districts. We investigated several factors that concur to the realistic aspect of the city, such as an appropriate level of granularity, mapping between the software domain and the urban domain, and layouts and topology. We looked into ways of mapping software metrics to visual properties and came up with a mapping technique that eases the reasoning about the visualized system, by limiting the number of different sizes presented at one time. The two variations of this mapping have their advantages and disadvantages. The boxplot-based mapping produces well balanced cities, but is not useful for comparisons among systems. The threshold-based mapping overcomes this drawback, due to the absolute values used, but is conditioned by the existence and reliability of thresholds. The interaction features our approach supports are selection, isolation, tagging with color and transparency, querying the system, and inspecting elements. Navigation within and around the city is realized as vertical and horizontal navigation. We implemented all the concepts presented in this paper in a tool called CodeCity. CodeCity scales up to industrial-size software systems, such as ArgoUML (over 2,500 classes), Azureus (over 4,500 classes) or VisualWorks (over 8,000 classes).

Our future work is dedicated to further increase the realism of the cities and to provide more interaction features. We would also like to find proper representations for the class internals (methods and attributes) and the relationships.

Acknowledgments. We acknowledge the financial support of the Hasler Foundation for the project "EvoSpaces - Multi-dimensional navigation spaces for software evolution" (MMI 1976). We also thank Romain Robbes for proof-reading.

References

[1] K. Andrews, J. Wolte, and M. Pichler. Information pyramids: A new approach to visualising large hierarchies. In *Proceedings of VIS 1997 (IEEE Visualization Conference)*, pages 49–52. IEEE CS, Oct. 1997.

[2] T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[3] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz. Software landscapes: Visualizing the structure of large software systems. In *VisSym 2004, Symposium on Visualization, Konstanz, Germany, May 19-21, 2004*, pages 261–266. Eurographics Association, 2004.

[4] 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.

[5] M. D'Ambros and M. Lanza. Reverse engineering with logical coupling. In *Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering)*, pages 189 – 198, 2006.

[6] 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.

[7] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, Jan. 2005.

[8] R. Falke, R. Klein, R. Koschke, and J. Quante. The dominance tree in visualizing software dependencies. In *VISSOFT*, pages 83–88, 2005.

[9] S. Few. *Show me the numbers: Designing Tables and Graphs to Enlighten*. Analytics Press, 2004.

[10] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, Nov. 1997.

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

[12] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.

[13] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[14] 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 xxx–xxx, Los Alamitos CA, 2007. IEEE Computer Society Press.

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

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

[17] T. Panas, R. Lincke, and W. Löwe. Online-configuration of software visualization with Vizz3D. In *Proceedings of ACM Symposium on Software Visualization (SOFTVIS 2005)*, pages 173–182, 2005.

[18] A. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.

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

[20] L. Voinea, A. Telea, and J. J. van Wijk. CVSscan: visualization of code evolution. In *Proceedings of 2005 ACM Symposium on Software Visualization (Softviz 2005)*, pages 47–56, St. Louis, Missouri, USA, May 2005.

[21] R. Wettel and M. Lanza. Program comprehension through software habitability. In *Proceedings of 15th International Conference on Program Comprehension (ICPC 2007)*. IEEE Computer Society, 2007.

[22] M. Wilhelm and S. Diehl. Dependencyviewer - a tool for visualizing package design quality metrics. In *VISSOFT*, 2005.