Program Comprehension through Software Habitability

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

Abstract

The comprehensive understanding of a large software system is a daunting task because of the sheer size and complexity that such systems exhibit. In this context software visualization is a widely used approach, since well-conceived visual representations allow one to spot patterns. The large majority of visualizations use 2D representations, because they are easier to construct, navigate, and interact with. 3D representations usually exploit the 3rd dimension as an additional means to encode quantitative values, which is dismissed by many as a too small benefit in the light of the added complexity in terms of navigation and interaction.

We argue that a well-constructed, interactive, and easily navigable 3D visualization can greatly help in program comprehension tasks by supporting habitability. *Habitability transmits to a developer the notion that a software system is a physical space with strong orientation points. This can give developers the feeling of being "at home" in a system. We propose a 3D visualization of software systems hinging on the city metaphor. It is useful for program comprehension because it leads to clarity about the overall structure of a system. We apply our visualization technique on two large systems and discuss its benefits and drawbacks.*

1 Introduction

There's no place like home. Developers spend an important part of their time constructing complex software systems, an activity termed as programming, which is according to [22] "a kind of writing". Writing code is a human and mental activity. The more familiar we are with a program, the easier it is to understand the impact of any modification we may want to perform, *i.e.,* familiarity has an important influence on program comprehension strategies [18]. Familiarity is strongly related to *habitability*, which is what makes a place livable, like home. R. Gabriel [9] states that "habitability is the characteristic of source code that enables programmers, coders, bug-fixers, and people coming to the code later in life to understand its construction and intentions [...]".

We argue that habitability is a neglected but important concept of program comprehension. While Gabriel's position stems from the point of view of language design, we take the source code and its language as a given fact. How can we make an existing system habitable in the context of program comprehension and reverse engineering? In an ideal world we would want to obtain a mental model of a system like the one that a system expert has, who spends large amounts of time reading and writing code. This may take months or years, and since program comprehension is usually performed under time pressure, is not a viable option. A solution is proposed by the reengineering pattern "Read all the code in hour" [4], a technique to assess the state of a software system by means of a brief but intensive code review, which helps in familiarizing with the system code. Another program comprehension technique is visualization [16], which has been widely and successfully used by the program comprehension community [2, 7, 11, 12, 17, 19].

We claim that many of the proposed visualizations, despite their proven usefulness, fail at transmitting to the viewer a sense of habitability. In the case of 2D visualizations -the large majority- the viewer looks at a system like a picture, without any notion of physical space. Many of the 3D visualizations (such as [12, 14]) on the other hand fail at producing a notion of *locality* because 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 propose a 3D visualization technique which supports the concepts of habitability and locality by using as central *city* metaphor complemented by other decisions with respect to appropriate visual representations. The viewer perceives a system as a city with visual orientation points and a diminished but more realistic freedom of movement and interaction.

Using our technique we visualize two large software systems, namely ArgoUML and Azureus, and report on our findings.

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

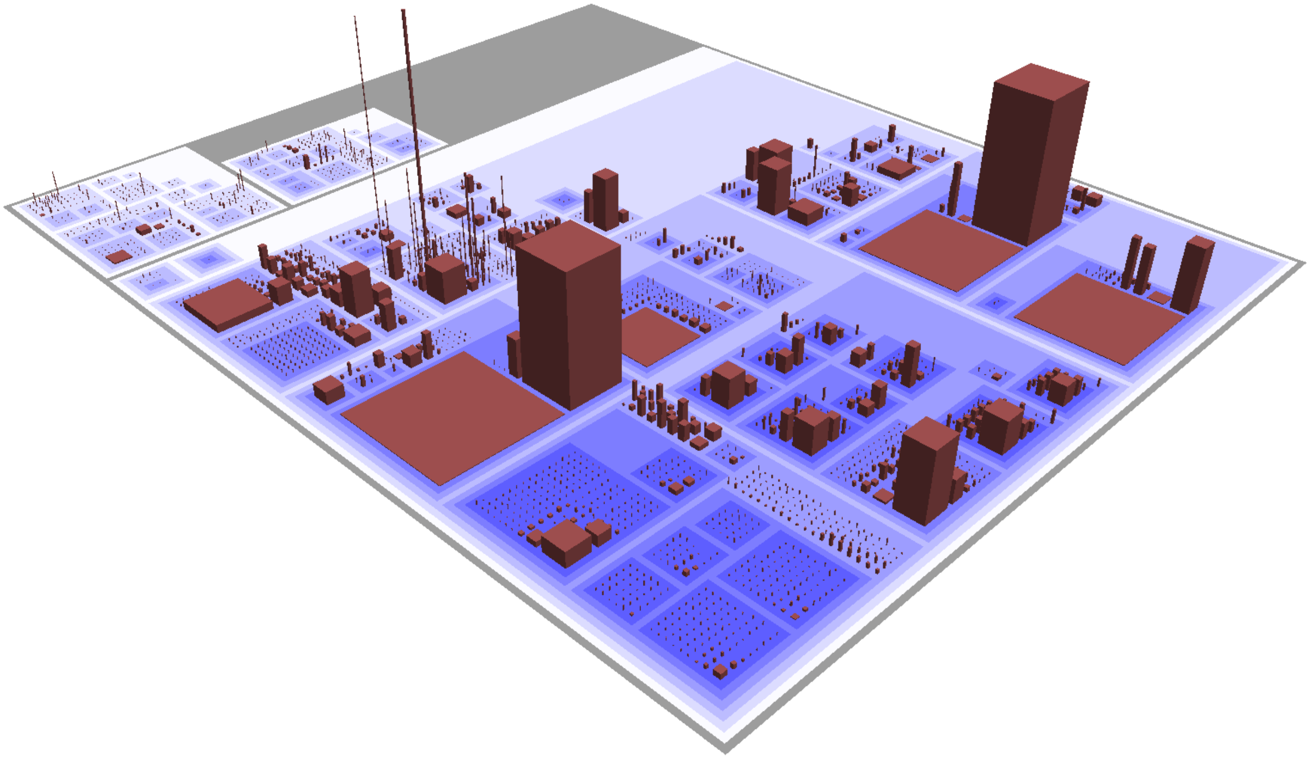


Figure 1. ArgoUML as a CodeCity

2 The City Metaphor

A good visual metaphor is the key to support habitability. Many 3D visualizations are undoubtedly appealing, but fail at communicating relevant information about the system and thus fail at supporting program comprehension tasks. We focus on object-oriented programs, where the constructs that need to be understood include packages, classes, methods, and attributes, and all their explicit and implicit relationships. After some experiments, which are not the focus of this paper, we settled on a *city* metaphor: classes are represented 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) does not do justice to the complexity of a software system, and leads to incorrect oversimplifications: Software is complex, there is no way around this.
- Classes are the cornerstone of the object-oriented paradigm, and together with the packages they reside in, the primary orientation point for developers. We do not display the class internals, because for a large-scale understanding it is not necessary. Apart from overplotting problems, it is also contrary to the way one explores a city: the person does not start by looking into particular houses.

System	**ArgoUML**	**Azureus**
Lines of code	136’000	274’000
Methods	14’221	24’644
Classes/Interfaces	2’522	4’737
Packages	143	457

Table 1. Systems under study

Throughout the remainder of this paper we apply our visualizations on two systems, ArgoUML (an open-source Java project to draw and generate UML diagrams) and Azureus (a widely used peer-to-peer Java application), with a special emphasis on ArgoUML (version 0.23.4). We give a rough outline of their size and complexity in Table 1.

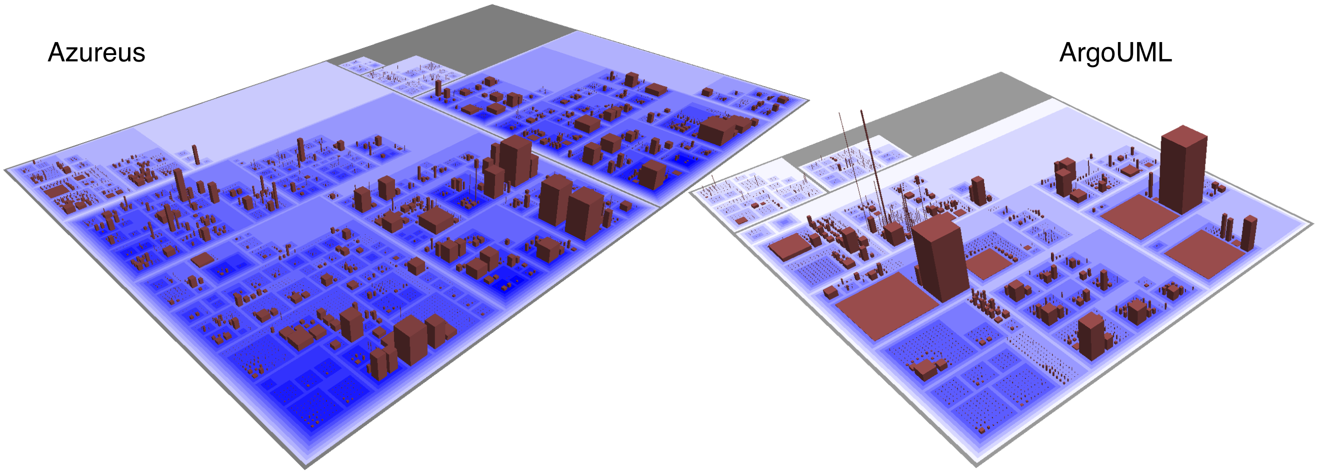


Figure 2. Comparing the code cities of Azureus and ArgoUML

2.1 Welcome to CodeCity

We see in Figure 1 ArgoUML, a 130+ kLOC Java system visualized as a CodeCity. The buildings represent classes and interfaces, placed in tiles representing the packages. The height of the buildings represent their number of methods (NOM), while the width and length represents the number of attributes (NOA). The increasingly stronger saturation of the tiles denotes the nesting level of the packages. On 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. They are not important at a first stage of the program comprehension process for exploring ArgoUML. The visualization allows us to easily spot some patterns such as the two massive buildings (potential god classes [15]), some antenna-shaped constructs, a number of classes looking like parking lots, and a large number of small houses. 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.

Another advantage is that using this visualization, systems can be compared to each other: In Figure 2 we see both ArgoUML and Azureus. We see that Azureus is larger than ArgoUML, but this information is already contained in the respective LOC-measurements. More important is that the visualization provides a sense of "OK, so this is what we are dealing with". Moreover, we see that although ArgoUML is smaller, its buildings have more exceptional shapes, while in Azureus there is only a small number of larger buildings. According to the more balanced proportions we see throughout the city of Azureus, its functionality seems to be more equally distributed among the classes.

2.2 The Buildings of CodeCity

However, the visualization presented so far is fundamentally flawed, because in a real city, despite the sometimes questionable mindset of architects, it is rare to have gigantic buildings or buildings like the antennas of ArgoUML. The linear mapping of the source code metrics, while being helpful, leads to an unrealistic feeling. The large variety of shapes also goes against one of the *gestalt* principles [8], which shows that humans efficiently distinguish at most 4-6 different shape sizes. If the city metaphor is to be fully used, we need a better type of representation for the buildings. Our goal is to represent a limited number of building types that better match the city metaphor.

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

Table 2. CodeCity building types

In Table 2 we list the types of buildings with their assigned sizes. The unit is "storey", *e.g.,* an apartment block is a six-storey building. In Figure 3 you see an example of the visual representation of the buildings. Since we map two different metrics, it is possible to have different combinations, such as a one-storey 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 found two ways of doing so, namely *boxplot-based* mapping and *threshold-based* mapping.

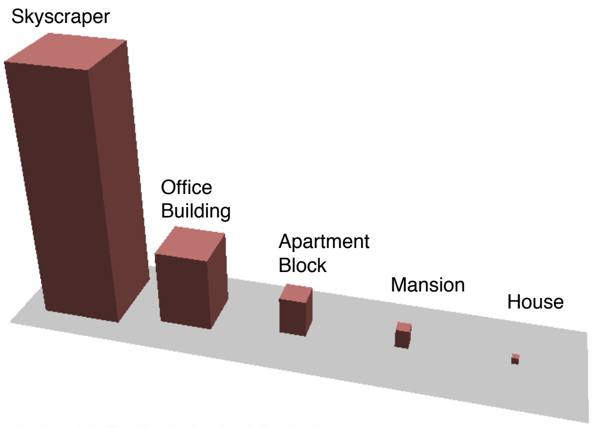


Figure 3. CodeCity building types

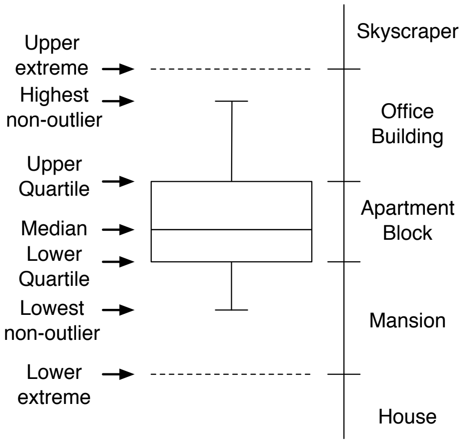
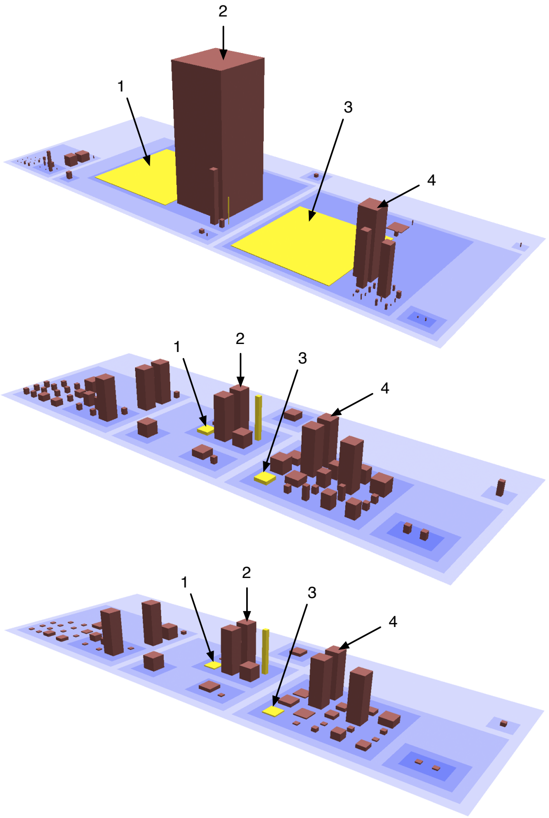


Figure 4. Boxplot-based mapping

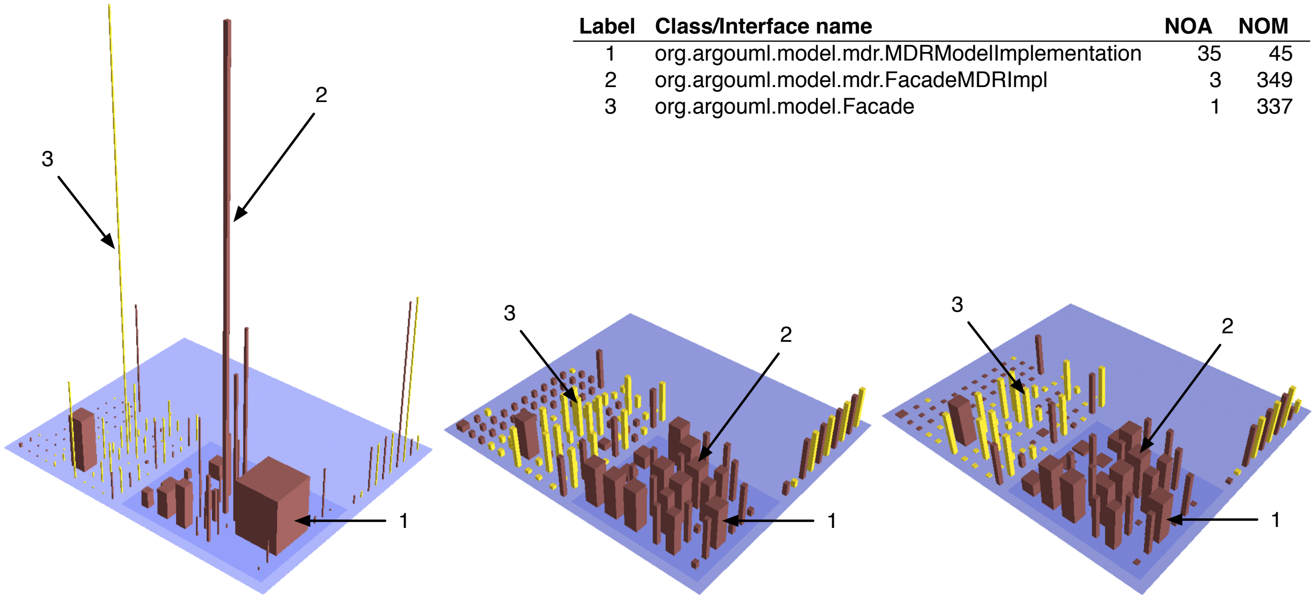
Boxplot-based Mapping. The boxplot is a widely used technique in statistics to reveal the center of the data, its spread, its distribution, and the presence of outliers. The construction of a boxplot requires obtaining the minimum and the maximum non-outlier value, and the quartiles (lower quartile Q1, median, upper quartile Q3) [20, 21]. We use the lower extreme limit, lower quartile, upper quartile, and upper extreme limit to split the population of the software artifacts with respect to one of the used metrics (*e.g.,* number of methods) into 5 groups (see Figure 4) which represent the building types. Two metric values within the same type range, even if they greatly differ, are mapped on the same building type. An important property of the boxplot is that the interquartile range (between Q1 and Q3) hosts the middle 50% of the data, thus insuring a well-balanced city in terms of buildings: at least half of them are apartment blocks.



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

Figure 5. Linear, boxplot-based, and threshold-based mapping applied to org.argouml.language

Threshold-based Mapping. The boxplot-based mapping has the advantage of creating balanced cities with few types of buildings. However, it makes a comparison of systems impossible, because the mapping thresholds for the building types depend on the metric values present in a system. A mapping to compare cities with few building types requires some "magic numbers" for the thresholds that hold across system boundaries. We took the values presented in [11] where the authors measured many widely different systems in terms of sizes, domain, and type (commercial vs. open-source). They produced threshold values for many software metrics. For Java systems the NOM threshold values are 4 (low), 7 (average), 10 (high), and 15 (very high). We use these thresholds as categorical boundaries for the building types for their height, and the boxplot-based mapping for the width and length.



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

Figure 6. Linear, boxplot-based, and threshold-based mapping applied to org.argouml.model

Comparison of Mappings. The results of applying the three presented mappings on the same system varies significantly. We use two examples taken from ArgoUML to discuss this issue (see Figure 5 and Figure 6). In each example, the interfaces are colored yellow and the classes brown, the most extreme-sized buildings are annotated. In both figures the mapped metrics are NOM for the height and NOA for length and width.

While with linear mapping one can visually compare the actual metric values by looking at the heights of the buildings, this apparent advantage disappears in front of the large number of different sizes that are hard to distinguish. Another drawback of linear mapping is that classes with very small metric values are so small in the visualization that it is difficult to spot them.

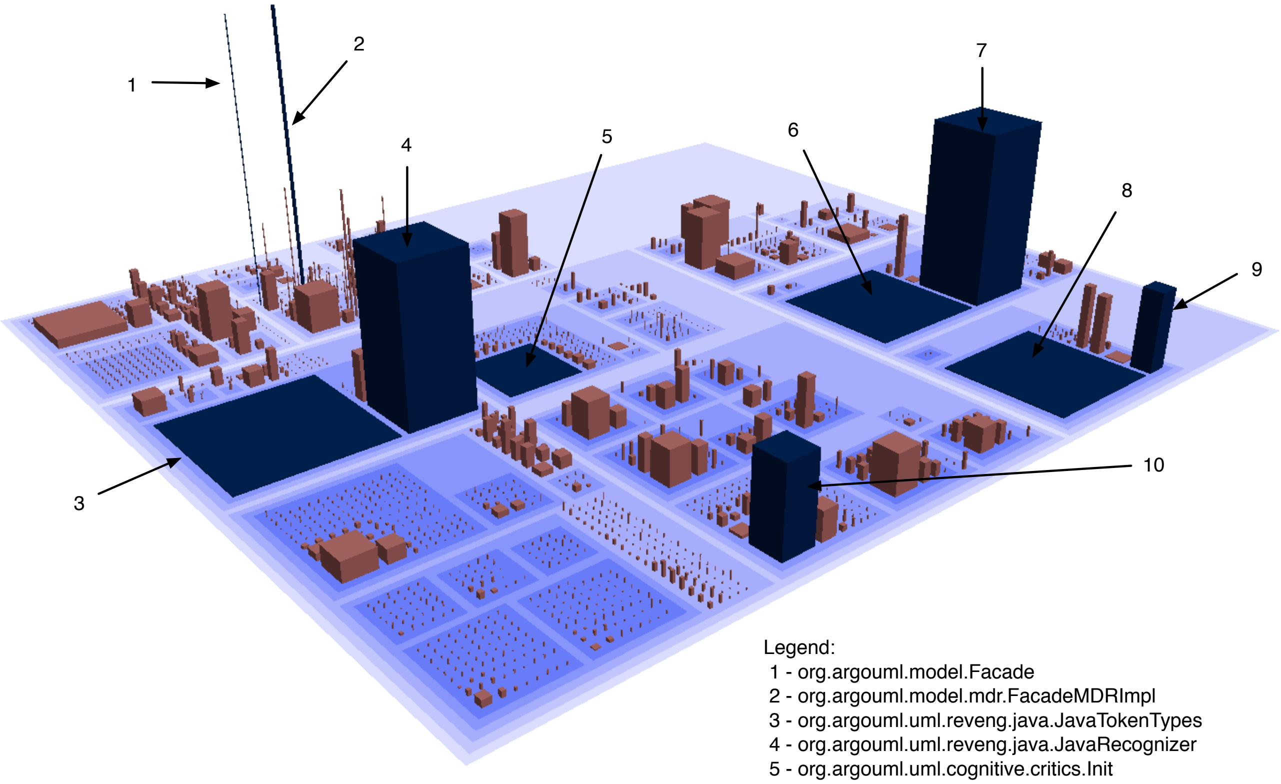
The two other mappings we propose are highly dependent on the 5 thresholds for each of the building types. A building with a moderate value for NOM and an high value for NOA should look moderately high and somewhat wider than the average buildings. With respect to proportions, a building with extremely high NOM and NOA should look gigantic, yet proportional (a massive skyscraper), while a building with extremely low NOA and NOM should resemble a small, yet visible, house.

In Figure 6 we see that with linear mapping, even on such a small subsystem it is already very difficult to have an overview and yet be able to see all the small entities. The best visibility is obtained with the boxplot-based mapping, where there is a visible balance in the clusters, due to the local context considered with the boxplot mapping. While the difference between the classes of the same category is not visible anymore, the very small buildings are still reasonably sized, so that they do not risk to become invisible in the overview. The threshold-based mapping (only on the height) is a good compromise, with a better overview than the linear mapping and yet still giving a sense of the distribution of the very large and very small classes. The same relation between the three mappings can be observed in Figure 5, representing the package org.argouml.model. It contains two very wide buildings and several tall buildings, including the most massive building in the system.

Summarizing, while the linear mapping is a representation of the real metric values, the boxplot-based and the threshold-based make the cities look more realistic, thus supporting habitability. The drawback of losing the real values is made up by the user interface of our tool (discussed later) which for each building we look at displays the actual metric values.

2.3 The CodeCity Layout

The final ingredient to the city metaphor is the layout. How can we obtain a city-like disposition of the classes in the system? We implemented a variety of layouts, such as simple grid or spiral layouts, not discussed here due to lack of space. We finally chose a rectangle-packing layout that takes into account the nesting of the packages, visualized as city districts containing buildings which represent the classes residing in them.



Legend:
1 - org.argouml.model.Facade
2 - org.argouml.model.mdr.FacadeMDRImpl
3 - org.argouml.uml.reveng.java.JavaTokenTypes
4 - org.argouml.uml.reveng.java.JavaRecognizer
5 - org.argouml.uml.cognitive.critics.Init
6 - org.argouml.language.cpp.reveng.STDCTokenTypes
7 - org.argouml.language.cpp.reveng.CPPParser
8 - org.argouml.language.java.generator.JavaTokenTypes
9 - org.argouml.language.java.generator.JavaRecognizer
10 - org.argouml.uml.diagram.ui.FigNodeModelElement

Figure 7. Overview of ArgoUML, with some tagged points of interest

3 A Walk through ArgoUML City

To exemplify the various concepts we introduced in this paper, we inspect and discuss the city of ArgoUML.

First impression are lasting impressions: The first view of a system influences the decision on where to start the investigation. It has to help us obtain a picture of the magnitude and structural complexity of the system. We use a rectangle packing algorithm for the layout of the classes and packages to improve the compactness of the first view. In Figure 7 you see the code city of ArgoUML, using the linear mapping technique. The annotated buildings represent the classes that we decided to look into after taking a cruise flight around the city.

There are several eye-striking buildings: the two tallest buildings are also very thin (buildings 1 and 2), there is a number of other tall buildings (4, 7, 9, 10) and there are some very wide, but flat buildings (3, 5, 6, 8) looking like parking lots. This view points out the mentioned outliers which represent a good starting point for our investigation.

The two antenna-like skyscrapers, representing a class and an interface with an enormous number of methods and very few attributes, are related: The class called org.argouml.model.mdr.FacadeMDRImpl (349 methods, 3 attributes) implements the interface org.argouml.model.Facade (337 methods, 1 attribute). Strangely, it is the only one implementing this enormous interface, which makes us assume that either in earlier versions of ArgoUML there were other classes implementing the interface, which eventually disappeared, or that designers' intend to provide support for polymorphism, or that it was just a bad design decision. To be sure which of these assumptions is true, we would need to look at evolutionary information about the system.

Changing org.argouml.model.Facade is a difficult and dangerous operation: In Figure 8 we tagged the potentially affected classes using a dark color, and we see that they are spread all over the system. For this figure we chose a combined mapping (threshold-based for the height and boxplot-based for the width and length), because even very small

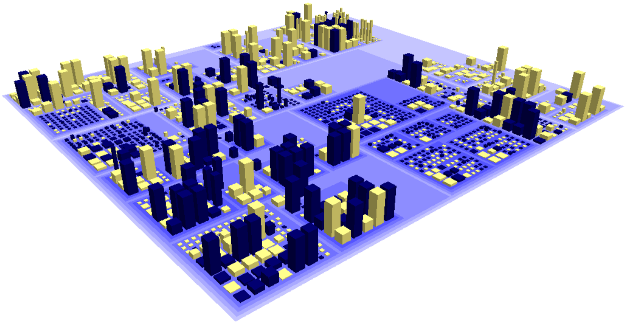


Figure 8. Change impact for interface org.argouml.model.Facade

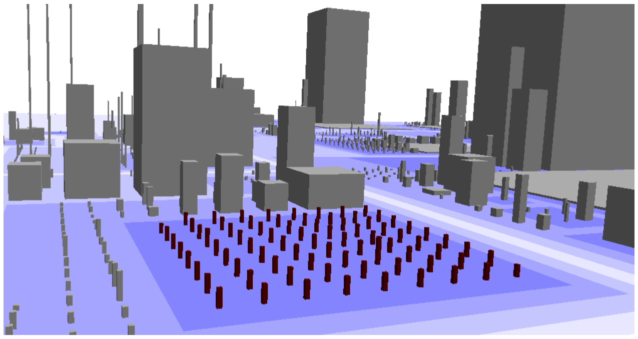


Figure 9. ArgoUML PackageRules **suburb**

buildings are easy to spot.

The buildings that look like parking lots, have many attributes and few methods. We are interested in the classes that use all that data. One of these buildings is an interface called org.argouml.uml.reveng.java.JavaTokenTypes, with 173 attributes and no methods. The only class that accesses the interface's attributes is one called org.argouml.uml.reveng.java.JavaRecognizer, in the same package. This huge class not only uses the interface's 173 attributes, but also it's own 79 attributes within the 176 methods it provides. We assume that the developers wanted to isolate everything that has to do with parsing Java code in as few classes as possible.

The next flat building represents the class org.argouml.uml.cognitive.critics.Init with 91 attributes accessed exclusively internally by the only method this class provides, namely init(). This initialization class is less problematic, since it is well encapsulated.

Moving over to the next flat building, we discover another token-related interface (org.argouml.language.cpp.reveng.STDCTokenTypes), having 152 attributes and no method. Similarly to the Java parsing part, this interface's attributes are exclusive data for another huge class, called org.argouml.language.cpp.reveng.CPPParser, with 204 methods and 85 attributes. The same observations we made in the case of the Java parsing pair also apply here.

With some expectations of finding a third parsing couple, we turn our attention to the last large and flat building and, to our surprise, we learn that the interface is called org.argouml.language.java.generator.JavaTokenTypes (containing 146 attributes and no methods) and its unique data accessor is called org.argouml.language.java.generator.JavaRecognizer (24 attributes and 91 methods). A possible reason for the presence of the class JavaRecognizer and of the interface JavaTokenTypes in two different packages may be the incremental migration of a hierarchy of classes from one package to another, with both source and destination co-habiting in the same version of the system. To be sure of this, we would have to look into the system's history.

The last massive building in Figure 7 is the abstract class org.argouml.uml.diagram.ui.FigNodeModelElement, which provides common data for the numerous node types. Since the nodes are core elements of any UML diagrams, there is a great amount of functionality in this class.

The last part of the city we want to investigate is the suburban area, such as the package org.argouml.ui.explorer.rules depicted in Figure 9, an entire district composed exclusively of small houses. The root of the main hierarchy implemented in this package is the interface PerspectiveRule[1] implemented by the abstract class AbstractPerspectiveRule, the superclass of 72 classes, all located in this package.

The only class of ArgoUML implementing the interface PerspectiveRule is the class AbstractPerspectiveRule. This is odd, because if there was no common functionality, the interface would suffice, while in the opposite case the abstract class could take over the contract from the interface. Given that the abstract class overrides Object's toString() method (concrete functionality is not possible in an interface) and superfluously declares as abstract methods two of the three methods already declared in the interface, the obvious solution is to declare the remaining method from the interface as an abstract method in the abstract class and remove the interface PerspectiveRule. The vast majority of the 72 subclasses of AbstractPerspectiveRule just provides an implementation for the three abstract methods in their superclass. At a second glance in the code, in spite of the low complexity, it seems that there are a lot of design problems revealed by this package. First, there are some NOP implementation of methods (return null), which makes us question the structure of this class hierarchy, since some of the

[1] We omit the preceding qualifier org.argouml.ui.explorer.rules in the class names mentioned up to the end of this section.

subclasses need that method, while others do not. Second, there is a lot of type checking in these methods, both direct (*i.e.,* the instanceof operator) and indirect, by using the methods of the notorious org.argouml.model.Facade. On a closer look at org.argouml.model.Facade's code, we see that it provides dozens of methods that do only type checking, with names such as: isAStereotype, isAState, isAAssociationRole, isANode, *etc.* It looks as if there has been a convention to keep type checking code (which can be easily replaced using overriding methods and polymorphism) in one place only, in spite of the few direct uses of instanceof introduced by programmers probably not fully aware of this convention.

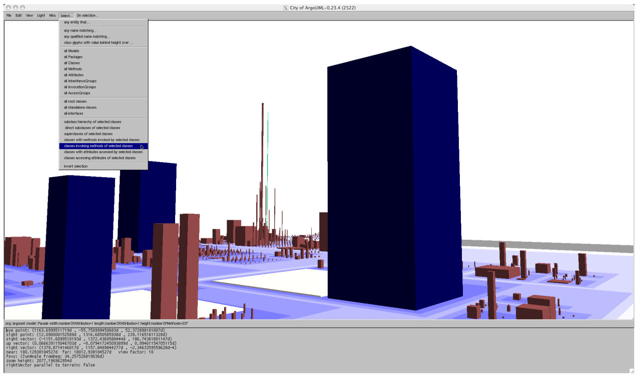
We stop here our tour through ArgoUML and reflect on our findings in the next section.

4 Discussion

Before we discuss the advantages and disadvantages of our approach, we briefly present our tool support, which is a crucial part of the approach.

4.1 Tool Implementation

We implemented the visualizations presented in this paper in a tool called CodeCity (Figure 10), written in Smalltalk and built on top of the Moose reengineering framework [5, 6], which makes it language-independent, *i.e.,* Moose currently supports the modeling of Java, C++, Smalltalk, and Python programs.

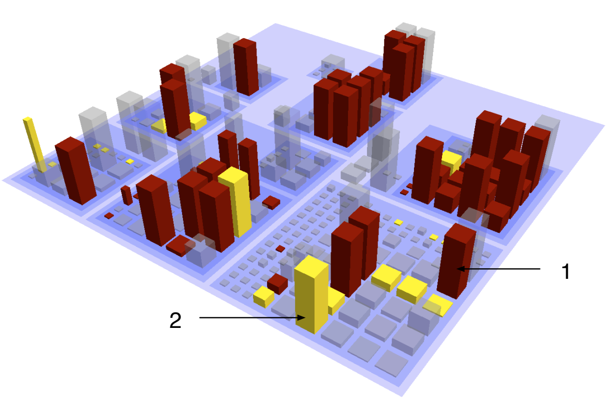

City of ArgoUML-0.23.4 (2522)
File
Edit
View
Light
Misc
Select...
On selection...
any entity that...
any name matching...
any qualified name matching...
class glyphs with value behind height over ...
all Models
all Packages
all Classes
all Methods
all Attributes
all InheritanceGroups
all InvocationGroups
all AccessGroups
all root classes
all standalone classes
all interfaces
subclass hierarchy of selected classes
direct subclasses of selected classes
superclasses of selected classes
classes with methods invoked by selected classes
classes invoking methods of selected classes
classes with attributes accessed by selected classes
classes accessing attributes of selected classes
invert selection

```
org::argouml::model::Facade width:numberOfAttributes=1 length:numberOfAttributes=1 height:numberOfMethods=337
eye point: (1163.6999511719d , -55.750999450683d , 52.372898101807d)
sight point: (12.090000152589d , 1316.6850585938d , 239.11651611328d)
sight vector: (-1151.6099510193d , 1372.4360580444d , 186.74361801147d)
up vector: (0.066639115046703d , -0.079417245093899d , 0.99461154705115d)
right vector: (1379.8714146017d , 1157.8489844277d , -2.3463359559628d-4)
near: 180.12938104527d  far: 18012.938104527d   view factor: 10
fovy: (JunAngle fromDeg: 34.257526819036d)
zoom height: 2077.196962954d
rightVector parallel to terrain: false
```

Figure 10. The CodeCity User Interface

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 popup context menu, which allows one to perform a variety tasks, such as inspecting the model entity, accessing the represented source code, *etc.*

While investigating a part of the system, the user is able to visually mark the territories already explored, by changing visual properties of the buildings and districts, such as color and transparency. To illustrate this, we will look closer into one of the core packages of ArgoUML, called org.argouml.uml.diagram, which contains the figures needed to draw the UML diagrams. To avoid the possible occlusions from other classes that do not make the object of our interest, we spawn a new view containing this district only (see Figure 11).



Label Legend:
1 - org.argouml.uml.diagram.ui.FigNodeModelElement
(superclass of the red hierachy)
2 - org.argouml.uml.diagram.ui.FigEdgeModelElement
(superclass of the yellow hierachy)

Figure 11. Tagged district of package org.argouml.uml.diagram

In this new local view, the highest buildings (containing the largest number of methods) are ui.FigNodeModelElement[2] (NOM=94) and ui.FigEdgeModelElement (NOM=73). We are interested in the subclass hierarchies of these two classes, we colorize each subclass hierarchy in a different color. By clicking on the highest building, we select ui.FigNodeModelElement. From CodeCity's menu, we choose the "Select subclass hierarchy of selected classes" and on this new selection we choose from the menu "On selection..." the item "Modify base color" and change the color to a dark red. Similarly we do the same for the edges, with a yellow color. If we are less interested in the classes that are not part of these two hierarchies, we can select the two superclasses and choose to select the subclass hierarchies for the current

[2] We omit the preceding qualifier org.argouml.uml.diagram in the class names mentioned in this section.

selection. This triggers an aggregated query having as a result the selection of the classes from any of the two hierarchies. Then we make a selection inversion, since we want to change the appearance of all the other classes in the view. On this new selection we choose to "Modify alpha" and bring the value closer to 0, such as 0.4 (*i.e.,* 60% transparency). The end result of these interactions with the diagram district of the ArgoUML city can be seen in Figure 11. To further amplify our lack of interest on these classes, we could also modify their color to a neutral one such as gray. The color tagging of groups of building can be used as described here, but it is also useful in a city-wide view to denote the parts of the city that we have already explored.

4.2 Reflections

Habitability & Locality. While the linear mapping best reflects the actual values of the chosen metrics for the software artifacts in a software system, it hampers habitability due to the extreme proportions that some of the buildings may have. For this reason, we looked into two other mappings that increase the habitability. The boxplot-based mapping, on the one hand, cannot be used to compare systems. The threshold-based mapping overcomes this disadvantage, and is also able to produce fairly habitable cities. The price we pay in this case is the difficulty of finding reliable thresholds for the categories. Another factor that influences the aspect and proportions of the buildings in the city are the dimensions assigned to each of the categories. After experimenting with different values, we chose values close to the ones in a real city, so that the representation of an average class in terms of the chosen metrics resembles an average building in a city.

Scalability. Because we settled our level of granularity to the class level, our approach scales well in terms of the size of the system that we can display as a code city.

Navigation & Interactivity. CodeCity provides various keyboard- and mouse-based navigation possibilities: moving forward or backward, hovering left or right, orbiting around the city, changing altitude. We still miss a navigation mode that enables walking on the ground (or driving in a car), so that the city immersion is even more realistic. We can interact with any item in the city. Moreover, selection queries are available, as well as the color-tagging of any item.

Completeness. The classes and the package structure provides an overview of the system. At the current stage we do not directly represent lower-level artifacts, such as methods and attributes, that would actually reside within the buildings. We also do not currently directly represent relationships such as inheritance and method invocations. While we have this information at disposition, and we can already represent them as edges connecting the buildings, they quickly lead to over-plotting problems. An appropriate representation of the relationships and the lower-level artifacts is part of our future work.

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 to emphasize habitability. Because of that we omit their discussion and concentrate on the approaches with a greater similarity to our work.

Knight and Munro [10] 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 [3], 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 [13] Panas *et al.* 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 Marcus *et al.* [12] 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.* [1] proposed the idea of representing software systems as landscapes, which is a concept we want to look into and possibly merge with our approach.

6 Conclusions

We have presented a program comprehension approach using 3D visualizations based on the metaphor of a city. We emphasize the concept of habitability and locality by providing a navigable and interactive environment in which we can freely move. The concept of habitability is enforced by making the class buildings in the city have realistic proportions. Navigation is allowed in all three dimensions, but with a clear notion of the ground. Interacting with the buildings in the city is easy and allows us to quickly access the underlying source code. The concept of *color-tagging* allows us to mark already visited places or places that we may want to come back to. We illustrated how we comprehend a system by walking through the ArgoUML CodeCity.

The main contribution of this paper is the construction of a 3D environment which by means of appropriate representation techniques, such as the boxplot-based mapping, allows us to obtain a visual world that hinges on the city metaphor. The city metaphor triggers the concepts of locality and habitability, *i.e.,* developers can talk about a certain part of the city as an actual physical place. We are convinced that this notion eases communication between people beyond what is possible with UML. We elaborated various interaction techniques, such as tagging, that allow to progressively uncover and inspect new parts of the city.

As part of our future work we plan to perform a comparative evaluation with visual techniques such as UML diagrams and other 2D visualizations.

Acknowledgments. We gratefully acknowledge the financial support of the Hasler Foundation for the project "EvoSpaces - Multi-dimensional navigation spaces for software evolution" (Hasler Foundation MMI Project No. 1976).

References

[1] 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.

[2] D. Beyer and C. Lewerentz. CrocoPat: A tool for efficient pattern recognition in large object-oriented programs. Technical Report I-04/2003, Institute of Computer Science, Brandenburgische Technische Universität Cottbus, Jan. 2003.

[3] 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.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[5] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[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] J.-M. Favre. Gsee: a generic software exploration environment. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 233–244. IEEE, May 2001.

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

[9] R. P. Gabriel. *Patterns of Software*. Oxford University Press, 1996.

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

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

[12] 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.

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

[14] 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.

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

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

[17] M.-A. D. Storey, C. Best, and J. Michaud. Shrimp views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*, 2001.

[18] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration. *Journal of Software Systems*, 44:171–185, 1999.

[19] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 12–21. IEEE Computer Society, 1997.

[20] M. Triola. *Elementary Statistics*. Addison-Wesley, 2006.

[21] J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[22] G. M. Weinberg. *The Psychology of Computer Programming*. Dorset House, silver anniversary edition edition, 1998.