CodeCity

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

Abstract

The analysis of large-scale software is difficult in the absence of supporting tools, due to the sheer size and complexity of today's systems. We present CodeCity, a language-independent interactive 3D visualization tool for the analysis of large object-oriented software systems. Using a city metaphor, it depicts classes as buildings and packages as districts of a "software city". We start presenting the tool in the context of its integration in our tool chain and gradually increase the level of detail, by continuing with the system's architecture and ending with a peek into one of its main class hierarchies. We apply our tool on a set of Java, C++, and Smalltalk systems and discuss tool-related issues.

Key words: software visualization, city metaphor

1. Introduction

Software maintenance claims up to 90% of the overall costs of a system [9]. Half of the time dedicated to maintenance is actually spent on understanding the subject system, a difficult process in the absence of supporting tools. Among the many possible approaches, we focus on software visualization, because of its capability to synthesize large amounts of information.

We present CodeCity, a language-independent interactive 3D visualization tool, developed to support the analysis of large-scale object-oriented software systems. CodeCity revolves around the *city* metaphor [23, 24], i.e., it represents systems as cities, where classes are depicted as buildings, and packages as the districts of the city. First, we introduce the tool in Section 2 and then present our entire tool chain in Section 3. Further focusing solely on CodeCity, we look at its modular architecture in Section 4 and at flexibility in Section 5. We continue with a discussion on tool-related issues in Section 6 and lessons learned in Section 7. Finally we present related work in Section 8 and conclude in Section 9.

Email addresses: `richard.wettel@lu.unisi.ch` (Richard Wettel), `michele.lanza@unisi.ch` (Michele Lanza).

2. CODECITY in a Nutshell

CODECITY is a 3D visualization tool which supports software analysis tasks. In CODECITY, large-scale software systems are depicted as cities [24], a familiar metaphor which provides a natural environment for exploration. The buildings of the city representing classes reside in districts which represent the packages. The visual properties of the city artifacts map software metrics: the number of methods metric for the class is mapped on the building height, the number of attributes on the base size (both width and length). The nesting level of a package is mapped on the color saturation of the district, *i.e.,* deeply nested packages are colored with dark blue, while the shallow ones are light blue. The visualizations in CODECITY provide both a practical overview of the magnitude and structure of the analyzed system and the means to further explore any artifacts of interest.

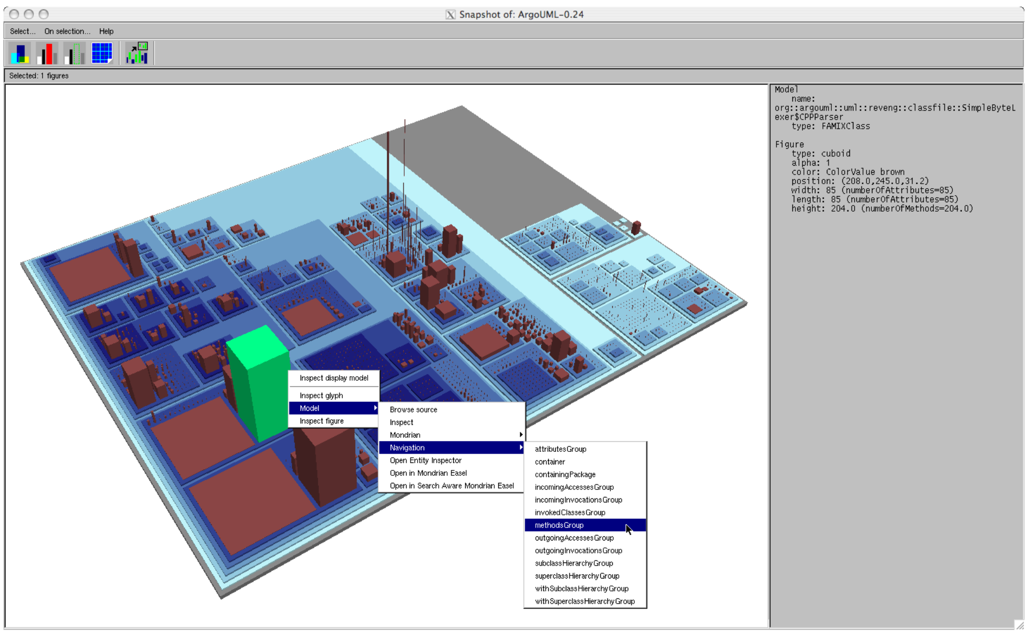

Fig. 1. ArgoUML, a 140 kLOC Java system, visualized in CODECITY

The CODECITY visualization window (see Figure 1) is made of the interactive visualization and an information panel, on the right side, which shows details on the focused element. The user can explore the city using the keyboard and the mouse. To allow for a good perspective over the city, CODECITY provides both basic movements such as panning and moving forward or backward, as well as orbiting movements on both the vertical and horizontal plane.

The interaction with the city artifacts is done either directly using contextual menus or indirectly using the query mechanism which provides both generic and pre-defined queries. There are two categories of such specialized queries: (1) absolute (*e.g.,* interfaces, root classes) and (2) relative to the current selection (*e.g.,* classes which invoke methods of the selected ones).

The visualization in Figure 1 shows us the structural organization of ArgoUML and reveals the outliers in terms of the mapped metrics. Next, we look at the same system from a software design perspective.

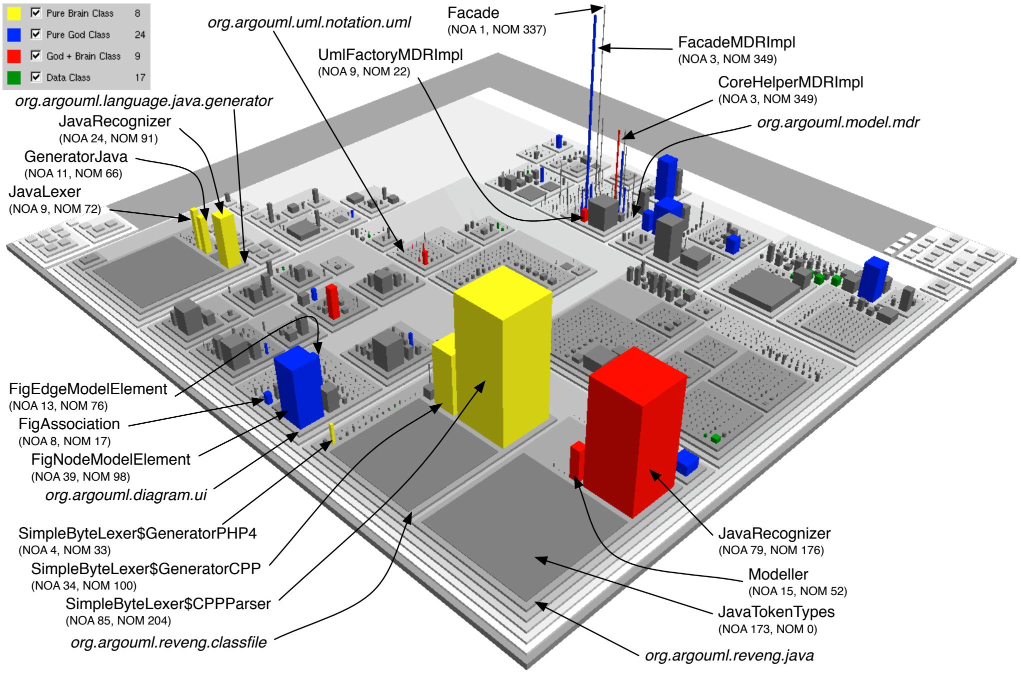

Fig. 2. ArgoUML's class-level disharmonies

The following scenario aims at characterizing a system from a design quality assessment point of view. We use the result of applying detection strategies to reveal design disharmonies [13] and represent it using the city metaphor to visually assess the degree of the various disharmonies and their distribution throughout the system (localization) [25]. Inspired by disease maps, we assign vivid colors to the design harmony breakers and shades of gray to the unaffected entities. This enables us to focus on the design disharmonies in the presence of a non-distracting context.

ArgoUML has 17 *Brain Classes* (yellow) and 33 *God Classes* (blue), 9 of which are affected by both disharmonies (red), and 17 *Data Classes* (green), distributed not all over the system, but rather sparsely, as Figure 2 shows. Some of the disharmonious classes are not surprising, given their high number of methods, such as the massive JavaRecognizer and CPPParser, which are fortunately generated classes that do not require manual maintenance. A less obvious example of harmony breakers are the 3 God Classes FigNodeModelElement (39 attributes, 98 methods), FigEdgeModelElement (13 attributes, 76 methods) and FigAssociation (8 attributes, 17 methods), which are core classes of the system and thus subjected to continuous maintenance. Another disharmonious agglomeration is a district characterized by a "forest" of very thin and very tall buildings (few attributes and many methods), representing package org.argouml.model.mdr. Out of its 35 classes, 8 are God Classes and 2 are God & Brain Classes. The largest affected class of this package, depicted by a building that literally touches the sky, is God Class FacadeMDRImpl (3 attributes, 349 methods). A disturbing case appears in package org.argouml.uml.notation.uml with one visible and three hardly visible God & Brain classes: NotationUtilityUML (NOA 6 , NOM 24, 1240 LOC), MessageNotationUML (NOA 2, NOM 29, 1538 LOC), AttributeNotationUML (NOA 2, NOM 8, 432 LOC), and OperationNotationUML (NOA 0, NOM 9, 450 LOC).

3. Toolset

In software development, as in real life, reinventing the wheel goes against progress. There are myriads of available successful tools in each niche, which allow us to build new tools on top of them. Indeed, CodeCity relies on a number of tools to perform its tasks (See Figure 3).

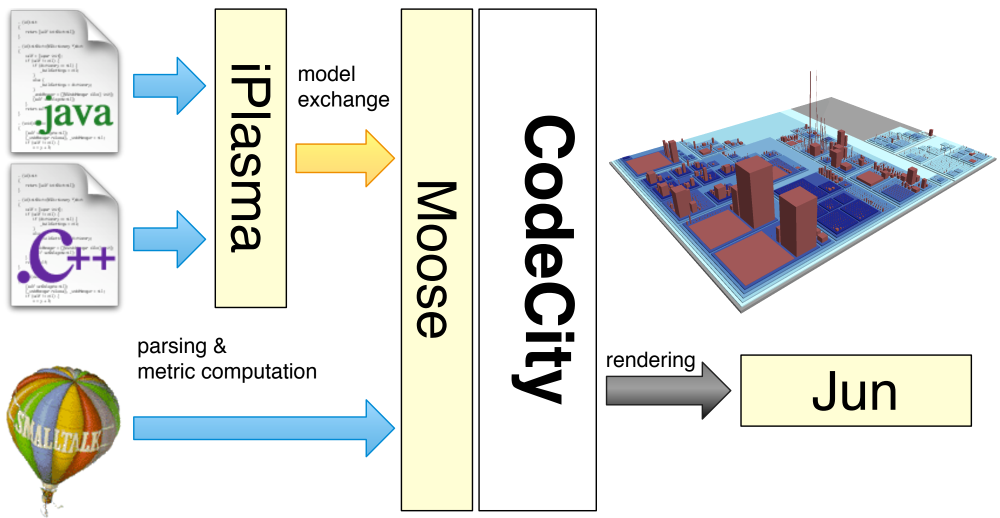

Fig. 3. Our complete tool chain

CODECITY is built on top of Moose [7], which provides an implementation of the language-independent FAMIX [5] meta-model. Thus, we benefit from the large amount of detailed, structured information provided by the FAMIX models of software systems and from an extensive set of software metrics computed by Moose. In the latest release of CODECITY, we also use the complementary 2D class blueprints [8] provided by Mondrian [17], a part of the Moose suite.

For Smalltalk code, we use the functionality of Moose from within CODECITY to create the FAMIX model of any system present in the Smalltalk image. For Java and C++ source code, we use iPlasma [16], a standalone tool which parses the code, computes the metrics, builds its own model, and exports the model using the MSE exchange format [1] before importing the model into Moose. From this point on, systems are handled uniformly, due to FAMIX's language-independence.

From the FAMIX model of any software system, CODECITY builds a visual model, applies mappings on its figures, and lays them out. Finally, it renders the 3D interactive cities using the OpenGL implementation of Jun [2].

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

4. The Architecture

The module-level architecture of CODECITY, depicted in Figure 4, is composed of four modules (*i.e.,*model, core, view management, and rendering) discussed in the following.

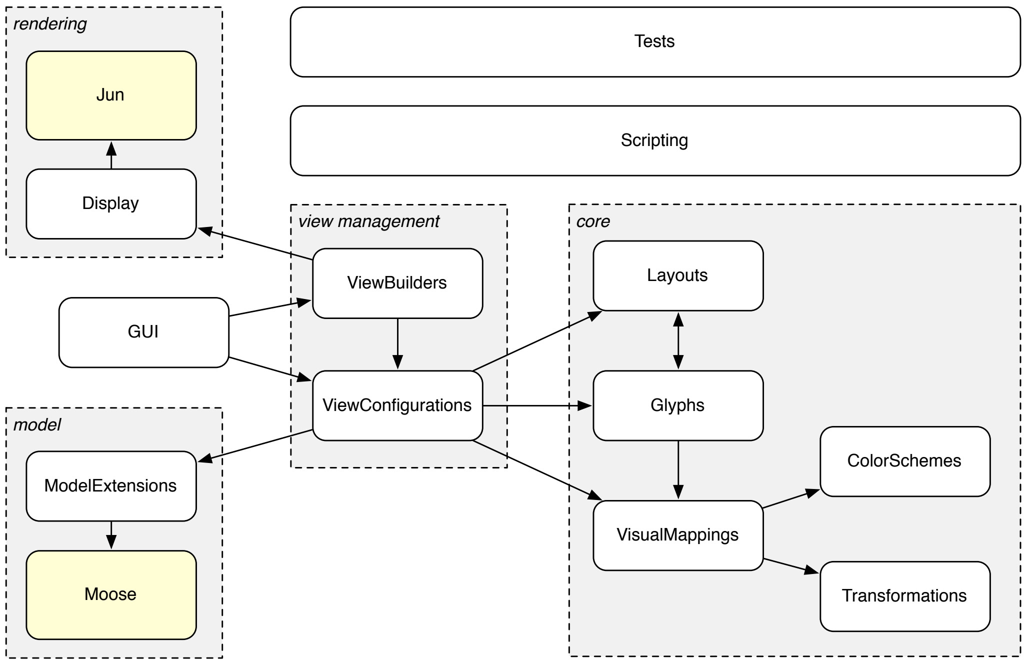

Fig. 4. CODECITY's module-level architecture

The **Model** module deals with modeling the software systems we visualize with CODECITY. We built our tool on top of the Moose framework, and extended the FAMIX meta-model as needed (in package ModelExtensions), using Smalltalk's class extension mechanism.

The **Core** module deals with the visual model, whose main components are figures (Glyphs), layouts (Layouts), and the mapping mechanism (VisualMappings). The latter, described in Section 5, provides the means to map model properties (*i.e.,* software metrics) onto glyph properties (*e.g.,* size, color), and involves the computational parts of the packages Transformations and ColorSchemes.

The **View management** module handles the configuration (ViewConfigurations), described in detail in Section 5, and the construction of visualizations (ViewBuilders) which, based on the current view configuration, takes a software model, builds the visual model of the system and calls the rendering engine to generate the visualization.

The **Rendering** module takes the visual model of a system and renders it on the screen. We currently use the Jun framework as an OpenGL implementation, on top of which we built the Display package, which serves as an interface between CODECITY and Jun, which deals with the visualization rendering, navigation, and interaction.

5. Tunable Visualizations

Measuring complex "organisms" such as software systems involves tens, if not hundreds, of different metrics. On the one hand, the amount of information a view is able to display is limited, and displaying too many details can overwhelm the viewer. As a result, one needs to carefully select the software characteristics of interest, according to the task at hand. On the other hand, there are further interesting, higher-level aspects to software, such as the ones of software evolution or the quality of the design. To provide assistance for various tasks, we strived for extensive configurability, by means of the tunable parameters, assembled as view configurations.

A *view configuration* is a specification defining for each model element type:

(i) *visibility*, *i.e.,* whether to provide a visual representation for it,
(ii) the associated *glyph* type,
(iii) the *layout* to use when placing the representations of the contained software artifacts (*e.g.,* the layout specified for packages will be used to place the sub-packages and classes), and
(iv) the visual *mappers* associated with each property of the chosen glyph.

The understanding of our tool's flexibility starts with the visual mappers, presented next.

5.1. *The Mapping Mechanism*

The mapper class hierarchy, described in Figure 5, started with realizing that the one-to-one straightforward mapping of properties is not always desired [23], and has grown ever since.

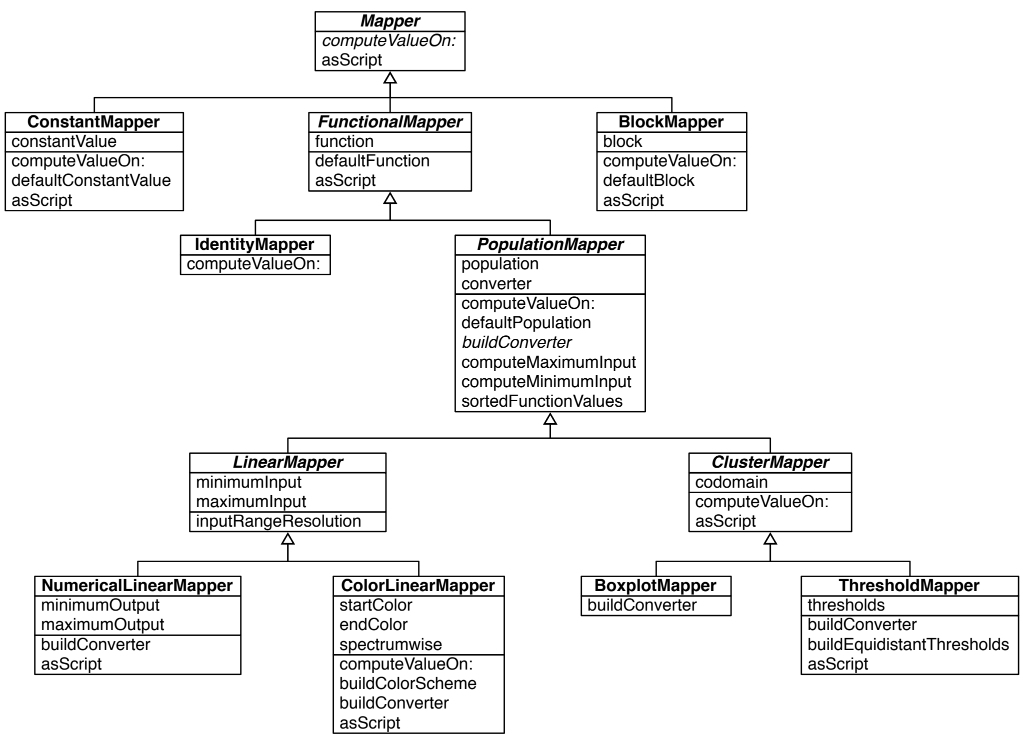

Fig. 5. Class diagram of CODECITY's mappers

The abstract Mapper class is the root class of the hierarchy and defines the essential method computeValueOn: which takes as parameter a model element, (*e.g.,* a class or a package) and returns a value mappable on a figure property. The asScript method converts a Mapper object into a script, *i.e.,* a string whose evaluation returns a Mapper object equivalent to the original one. It is used to store the view configuration (containing mappers, among others) as scripts. Instead of exporting the configurations as files, we save them as class-side methods in class ConfigurationRepository, which puts them under the Smalltalk version control mechanism (*i.e.,* Store).

The ConstantMapper returns a constant value regardless of its input. It is used to map the same value for all the elements of a particular type, *e.g.,* the brown color of class figures (Figure 6a).

Next, the BlockMapper is a generic mapper which requires some Smalltalk programming skills [1]. We use it to deal with cases for which there is yet no direct support. As an example, Figure 6b presents a mapper that makes classes opaque and interfaces semi-transparent [2].

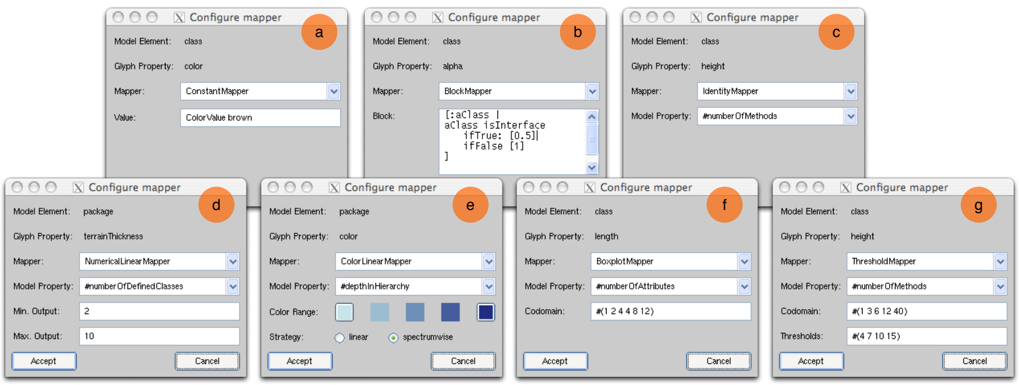

Fig. 6. User interface examples for the mappers

The abstract class FunctionalMapper is root to the core subtree, which defines the attribute function. This attribute describes the method that needs to be called on the input parameter (*i.e.,* the model element) to obtain the value of a property of interest of the model, *e.g.,* a software metric.

The functional mapper which provides the most faithful metric representation is the IdentityMapper, based on the identity function: $f(x) = x$. An example of mapping the number of methods of a class on the building's height is shown in Figure 6c. The implementation advantage of the identity mapper is that the output value of the mapper only depends on the input model element.

However, to perform more complex mappings, we need to consider the metric values of all the elements for a particular model type (*i.e.,* a population). The super-class of these mappers, the abstract class PopulationMapper also defines a converter for its the computational tasks.

The first population mapper is the LinearMapper, whose minimumInput and maximumInput depend on the population. The first concrete linear mapper is NumericalLinearMapper, which requires the boundaries for the output range to be specified before performing its computations. An example of such mapper is mapping the package's number of defined classes on the terrain's thickness within a given range (*i.e.,* 2..10), presented in Figure 6d.

For colors we defined the ColorLinearMapper, which uses a color scheme defined by the startColor and endColor attributes to linearly convert the value of the input property to a corresponding

[1] A block in Smalltalk is a self-contained piece of code which can be evaluated/executed.
[2] Classes and interfaces are modeled by the same type in FAMIX.

color within the given range. The direction inside the color spectrum is given by the attribute spectrumwise. Figure 6e presents our usual color mapper for the packagess: The depth in hierarchy (nesting level) of the package is mapped on a color range of blue shades. The lowest value is mapped on the light blue and the highest value on the dark blue, as reflected by the startColor and endColor color pickers. For every other metric value the mapper produces an intermediate color of the spectrum, such as the three equidistant color samples placed between the color pickers.

The root of the second hierarchy of population mappers is ClusterMapper, which specifies the sequence of possible outputs (*i.e.,*codomain), depending on the cluster of input values which the model element belongs to.

The first concrete cluster mapper is BoxplotMapper, which uses the boxplot technique to compute the boundaries between the input ranges corresponding to the *extremely low*, *low*, *average*, *high*, and *extremely high* categories. The boxplot mapper in Figure 6f maps a value $l \in \{1, 2, 4, 8, 12\}$ on the building's length, according to the cluster to which its number of attributes metric value belongs. Each output value corresponds to a building category described in [23].

The second cluster mapper, ThresholdMapper, allows the user to manually introduce the boundaries (*i.e.,* called here thresholds), based on statistical data on the typical values of software metrics, such as the ones described in [14]. The threshold mapper in Figure 6g maps on the building's height a value $h \in \{1, 3, 6, 12, 40\}$, according to the cluster to which the number of methods metric value of its class belongs. This example's input clusters, corresponding to the number of methods metric for Java classes, are: $[0, 4)$, $[4, 7)$, $[7, 10)$, $[10, 15)$, and $[15, \infty)$.

5.2. *Configuring a View*

View configuration tuning is visually done from the GUI (See Figure 7), which provides widgets for the modification of every view configuration parameter. The preview panel reflects the current view configuration applied on a dummy model, and allows to quickly understand the effect of each configuration parameter on the view. The configuration management capabilities allow saving a potentially useful configuration under a given name and description, and provide access to the saved configurations, for direct use or as base for building new configurations.

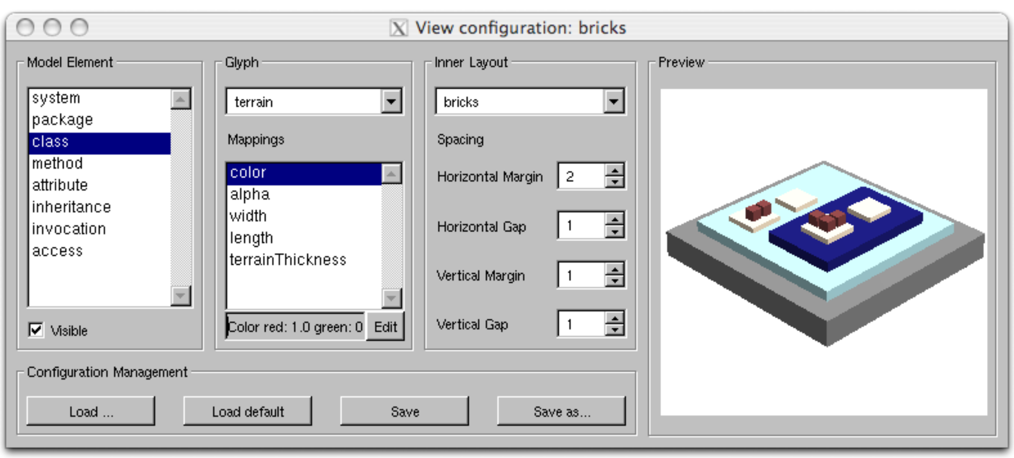

Fig. 7. User interface to the view configuration

5.3. *Scripting a View*

In spite of the flexibility provided by the GUI, it does not fully grant access to CODECITY's core, and can only visualize software system models, based on view configurations. However, by replacing the view construction part, we could visualize any type of structured information. Therefore, we implemented a basic scripting language inspired by Mondrian [17] to build ad-hoc visualizations. This allows us to experiment the feasibility of a new visualization before fully embedding it in the GUI. Although in the meanwhile the view construction part of the tool could be written using scripts, for performance reasons we kept it as before, *i.e.,* using domain knowledge to optimize the building. An example of a simple script is presented in Figure 8.

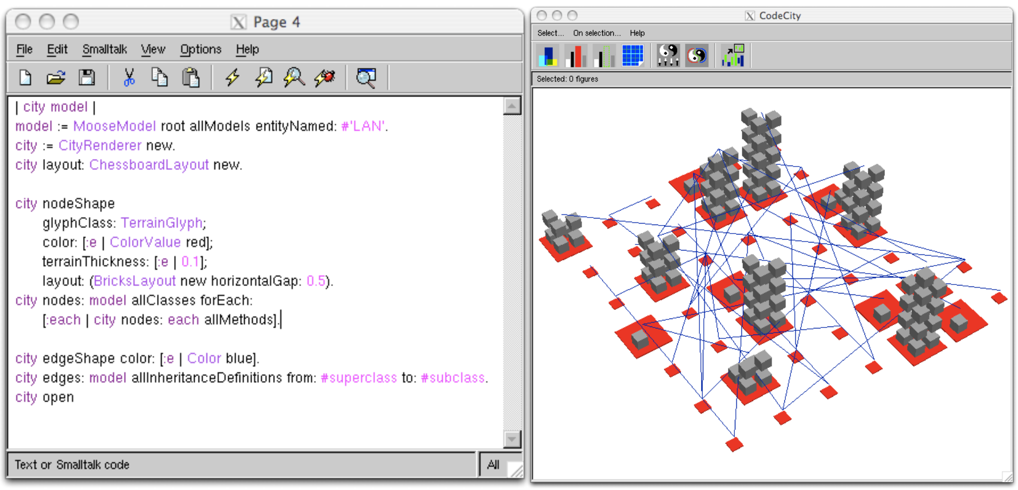

Fig. 8. Scripting example (left) and the produced output (right)

6. Discussion

Generality. We initially conceptualized CODECITY as a 3D visualization tool for object-oriented software systems, based on a city metaphor. We extended CODECITY to visualize both the evolution of software and disharmony-related information [26], however not the object of this paper. The configurability of our approach also allows us to step away from the city metaphor, by defining new types of glyphs and layout strategies. Moreover, the scripting takes it from a visualization tool for object-oriented software to a general information visualization tool.

Completeness. We are still looking for meaningful ways of representing the relationships among software entities (*e.g.,* inheritance, invocation, access), either explicitly as links between the artifacts or implicitly, *e.g.,* as distance between them in layouts.

Language Independence. Our visualizations are built using models of the software systems, rather than source code. Due to the language-independence of the FAMIX meta-model for object-oriented software we rely on, with CODECITY we can visualize software written in several programming languages, including Java, C++, or Smalltalk.

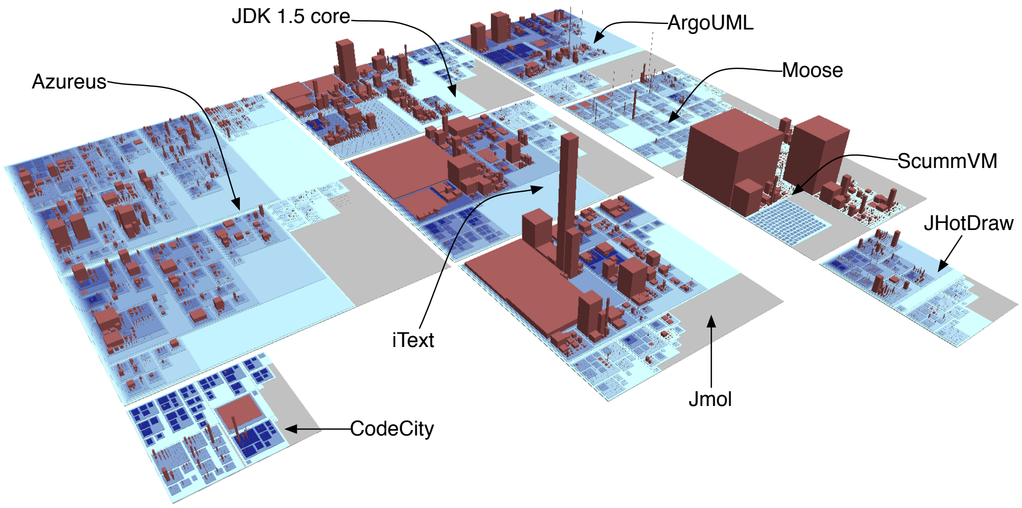

Fig. 9. Towards the 1 million LOC territory

Scalability. We applied CODECITY on several open-source software systems, some of which are presented in Table 1. To prove the scalability of our tool, we present a scripted visualization which brings together the worlds of Java, C++, and Smalltalk by depicting all the systems presented in Table 1, comprising ≈ 1 million (> 920k) lines of source code. This visualization allows us to visually compare the systems from the perspective of structure and magnitude.

System	Language	kLOC	Packages	Classes	Build time (s)
ArgoUML	Java	137	144	2'542	58
Azureus	Java	274	457	4'737	260
iText	Java	80	149	1'250	50
JDK 1.5 core	Java	160	137	4'715	80
JHotDraw	Java	30	72	998	8.5
Jmol	Java	85	105	1'032	36
CodeCity	Smalltalk	18	129	291	1.5
Moose	Smalltalk	32	278	961	3
ScummVM	C++	105	329	1'331	95

Table 1
The systems analyzed with CodeCity and the time performance of the tool

Performance. Table 1 shows the time needed to build and render the views for the presented systems, using a view configuration with class-level granularity of the representation (only packages and classes are represented). We measured it on an Apple MacBook Pro machine, with 2.4 GHz Intel Core 2 Duo processor and 4 GB of RAM, running on Mac OS X 10.5.3 under X11. Increasing the granularity to a finer-grained level or choosing to represent relationships would increase the time needed for building the visualizations (*i.e.,* the most time-costly part). The performance in rendering, navigation, and interaction is fairly good, since it depends on the number of polygons and we use simple figures (*e.g.,* cuboids) with a low amount of polygons (as opposed to spheres). The movement remains fluid at a screen resolution of 1440 x 990, even in the case of Azureus, which is represented by over 5,000 figures. However, this changes in the case of a finer level of granularity, which increases the number of figures.

7. Lessons Learned

Relying on Moose. Building CODECITY on top of the Moose framework provided the big advantage of not having to start from scratch. Moose, with its support for system modeling (both for single version and for evolution), metrics computation, and disharmony detection, enabled us to focus solely on the visualization part. Moreover, the Moose community actively maintain the framework. We rarely felt the need to extend Moose and then mostly with new metrics, such as the following example:

```
depthInHierarchy
  <property: #DIH
    longName: 'Depth in Hierarchy'
    description: 'The level of depth in the package hierarchy'>

  ^self lookUpPropertyNamed: #DIH
      computedAs:
        [self packagedIn isNil
          ifTrue: [1]
          ifFalse: [1 + self packagedIn depthInHierarchy]]
```

Using Reflection. Using reflection is an early investment which pays off in the long term. The mere addition of new glyphs, layouts, or mapper types to CODECITY's core enriches the user interface without any modification of the GUI classes. Indeed, fast prototyping is one of Smalltalk's strengths.

OpenGL. Jun was our default choice because it was the only OpenGL implementation available in VisualWorks Smalltalk. In spite of its lack of documentation, Jun provides many examples, which provide a starting point in working with it. The availability of the source code allowed us to debug the code and was the ultimate help in learning about the various aspects that come up in 3D graphics, such as *view point*, *sight point*, or *up vector*. However, many times the discoveries we made, which were not obvious due to the lack of documentation, were sources of frustration, such as the fact that Jun does not use the hardware graphic acceleration.

Unit Testing. CODECITY relies on over 270 unit tests progressively added along with the code they test over the 2 years of development, which provides us with increased confidence to perform any major modification of CODECITY's architecture. Our tests cover the core part of the system (glyphs, layouts, mappers, transformations) and the configuration management (the view configuration and partially the builders). The effort of writing tests was almost uniformly distributed over the entire lifetime of the system and it felt like healthy programming practice rather than an effort. Running the tests takes just a few seconds. The existence of the tests provide reassurance throughout the development process, since it removes some of the doubts one might have after a heavyweight change in the system's core. Moreover, the discovery of a bug most of the times results in a new test case, which makes it stronger. The GUI part is left untested, partly because it is difficult and partly because it is the part which gets constant attention and it is easy to observe wrong behavior.

8. Related Work

More than a decade ago, Reiss presented a framework for 3D visualization [20]. Since then, many 3D approaches have been proposed. Knight et al.'s *Software World* [11] and Charter et al.'s *Component City* [4] use a city metaphor, while Marcus et al.'s *sv3d* [15] and Balzer et al.'s *Software Landscapes* [3] use a similar 3D metaphor to visualize single versions of software systems. Ducasse et al. [6] used the city metaphor of the SimCity game to express challenges behind software evolution, which remained an idea. Another metaphor idea without implementation is proposed in Panas et al.'s [19] description of such a 3D city. Langelier et al.'s *Verso* [12] uses 3D visualizations to display structural information, by representing classes as boxes with metrics mapped on height, color and twist, and packages as borders around the classes placed using a tree layout or a sunburst layout.

In the context of customizability and scriptable visualizations, several approaches have been proposed. Based on Müller et al.'s *Rigi* [18], a sequel of scriptability-related works have followed. In a first phase, Tilley et al. worked on scripting using the Tcl language of not only the visual representation, but also of other aspects which allow users to tailor a wide range of reverse-engineering tasks, such as parsing, information extraction and organization [22]. In a second phase, Tilley et al. proposed the customization of the user interface through scripting based on the Tk language [21]. A more recent work is Favre's G^{SEE} *(Generic Software Exploratory Environment)* [10], a visualization tool which allows scripting the visualization using its own scripting language to accommodate the different data sources and forms. Another recent work in the field is *Mondrian* by Meyer et al. [17], which was the inspiration of introducing scripting in our tool. While all the presented tools focus mainly on the scripting part, our approach provides more specialized visualization and uses the scripting as a means to explore beyond the existing visualization and to extend its capabilities.

9. Conclusion

We presented CODECITY, a 3D visualization tool for object-oriented software system analysis. The tool revolves around the city metaphor, in which classes are displayed as buildings and packages as districts. To accomplish its tasks, CODECITY relies on other tools for code parsing, modeling, and rendering. Along with the module architecture of the system, we described the way these modules connect to each other. We then presented the mapping mechanism and its role in CODECITY's configurability and two perspectives of tailoring our visualization: the user's (*i.e.,* configuration) and the programmer's (*i.e.,* scripting). We discussed tool-related issues and lessons learned while developing our system. As a proof of scalability and language-independece we visualized several Java, C++, and Smalltalk in one view, totaling nearly 1 million LOC.

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). We want to thank Tudor Gîrba for his support in the implementation of the scripting part.

References

[1] The MSE format.
URL http://smallwiki.unibe.ch/fame/mse

[2] A. Aoki, K. Hayashi, K. Kishida, K. Nakakoji, Y. Nishinaka, B. Reeves, A. Takashima, Y. Yamamoto, A case study of the evolution of jun: an object-oriented open-source 3d multimedia library, in: Proceedings of International Conference on Software Engineering (ICSE), 2001.

[3] M. Balzer, A. Noack, O. Deussen, C. Lewerentz, Software landscapes: Visualizing the structure of large software systems., in: VisSym 2004, Symposium on Visualization, Eurographics Association, 2004.

[4] S. M. Charters, C. Knight, N. Thomas, M. Munro, Visualisation for informed decision making; from code to components, in: International Conference on Software Engineering and Knowledge Engineering (SEKE '02), ACM Press, 2002.

[5] S. Demeyer, S. Tichelaar, S. Ducasse, FAMIX 2.1 — The FAMOOS Information Exchange Model, Tech. rep., University of Bern (2001).

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

[7] S. Ducasse, T. Gîrba, O. Nierstrasz, Moose: an agile reengineering environment, in: Proceedings of ESEC/FSE 2005, 2005.

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

[9] L. Erlikh, Leveraging legacy system dollars for e-business, IT Professional 2 (3) (2000) 17–23.

[10] J.-M. Favre, Gsee: A generic software exploration environment, in: In Proceedings of the 9th International Workshop on Program Comprehension (IWPC 2001), IEEE Computer Society, 2001.

[11] C. Knight, M. C. Munro, Virtual but visible software, in: International Conference on Information Visualisation, 2000.

[12] G. Langelier, H. A. Sahraoui, P. Poulin, Visualization-based analysis of quality for large-scale software systems, in: ASE, 2005.

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

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

[15] A. Marcus, L. Feng, J. I. Maletic, 3d representations for software visualization, in: Proceedings of the ACM Symposium on Software Visualization, IEEE, 2003.

[16] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, R. Wettel, iPlasma: An integrated platform for quality assessment of object-oriented design., in: ICSM (Industrial and Tool Volume), 2005.

[17] M. Meyer, T. Gîrba, M. Lungu, Mondrian: An agile visualization framework, in: ACM Symposium on Software Visualization (SoftVis 2006), ACM Press, 2006.

[18] H. Muller, K. Klashinsky, Rigi: a system for programming-in-the-large, Proceedings of the 10th International Conference on Software Engineering (ICSE '88) (1988) 80–86.

[19] T. Panas, R. Berrigan, J. Grundy, A 3d metaphor for software production visualization, International Conference on Information Visualization (2003) 314.

[20] S. P. Reiss, An engine for the 3d visualization of program information, Journal of Visual Languages and Computing 6 (3) (1995) 299–323.

[21] S. Tilley, Domain-retargetable reverse engineering. ii. personalized user interfaces, in: Proceedings of 10th IEEE International Conference on Software Maintenance (ICSM'94), IEEE Computer Society Press, 1994.

[22] S. Tilley, H. Muller, M. Whitney, K. Wong, Domain-retargetable reverse engineering, in: Proceedings of 9th IEEE International Conference on Software Maintenance (ICSM'93), IEEE Computer Society Press, 1993.

[23] R. Wettel, M. Lanza, Program comprehension through software habitability, in: Proceedings of ICPC 2007 (15th International Conference on Program Comprehension), 2007.

[24] R. Wettel, M. Lanza, Visualizing software systems as cities, in: Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis), 2007.

[25] R. Wettel, M. Lanza, Visually localizing design problems with disharmony maps, in: Proceedings of Softvis 2008 (4th International ACM Symposium on Software Visualization), ACM Press, 2008.

[26] R. Wettel, M. Lanza, Visually localizing design problems with disharmony maps, in: In Proceedings of Softvis 2008 (4th International ACM Symposium on Software Visualization), 2008.