# Scripting 3D Visualizations with CODECITY

Richard Wettel
*REVEAL @ Faculty of Informatics - University of Lugano, Switzerland*

## Abstract

*Software visualization is a useful means to present and explore large amounts of information. However, constructing useful visualizations targeted at specific tasks is often a trial-and-error process. As a consequence, a visualization prototyping tools needs to be flexible to allow for the creation of new visualizations and also to provide an environment that grants access to its powerful mechanisms.*

*In this paper, we report on our experience with complementing the rich graphical user interface of* CODECITY, *a 3D visualization tool, with a scripting environment. The scripting language gives the programmers full access to the configurability of our system, without the need for them to be exposed to the real complexity of the application. Thanks to the scripting engine, adapting* CODECITY *to new types of data has become easy, as we illustrate with examples.*

## 1 Introduction

Software visualization provides useful assistance in tasks related to program comprehension and reverse engineering, because humans are good at spotting patterns. Software visualization in 3D allows for data exploration in environments closer to the ones we live in. In spite of the many useful visualizations available, researchers are always looking for new ones, either as improvements in terms of expressiveness, intuitiveness, or efficiency, over the existing ones, or just to visualize new types of data. To support this kind of exploration, a visualization tool needs to be flexible enough to accommodate unforeseen directions and to provide full access to its visualization creation mechanism.

Originally inspired by CodeCrawler's polymetric views [6], we developed CODECITY, a 3D software visualization tool based on a city metaphor. It provides assistance in exploring software systems, both for single version [13] and for multiple-version [15] analysis. From the very beginning, we strived for configurability, to allow its extension in order to support a wide range of tasks. To allow access to the configuration mechanism, we provided a graphical user interface (GUI). However, the consistent effort required to adapt the user interface after every extension of the system, made clear the need for a less constrained mechanism to be used when experimenting with visualizations. We extended our tool with scripting capabilities, which allows us to freely experiment and only adapt the GUI when the new visualizations are worth the effort.

## 2 Related Work

To our best knowledge, there is no 3D software visualization tool that provides scripting facilities. However, there are several approaches similar to ours with respect to the city metaphor we use. Knight *et al.*'s *Software World* [4] and Charter *et al.*'s *Component City* [2] use a city metaphor, however at a finer granularity which does not scale. Marcus *et al.* with *sv3d* [7] and Balzer *et al.* with *Software Landscapes* [1] use similar 3D metaphors to visualize software systems. Another city metaphor is proposed by Panas *et al.*'s in [10]. In *Verso*, Langelier *et al.* [5] use 3D visualizations to display structural information, by representing classes as boxes with metrics mapped on height, color and twist, and packages as borders around the classes placed using a tree layout or a sunburst layout.

In the context of customizability and scriptable visualizations, several 2D approaches have been proposed. Based on Müller *et al.*'s *Rigi* [9], a sequel of scriptability-related works have followed. In a first phase, Tilley *et al.* used the Tcl language to script 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 [12]. In a second phase, Tilley *et al.* proposed the customization of the user interface through scripting based on the Tk language [11]. A more recent work is Favre's $G^{SEE}$ *(Generic Software Exploratory Environment)* [3], a visualization tool which allows scripting the visualization using its own scripting language to accommodate the different data sources and forms. The work that inspired us in introducing scripting in our tool is *Mondrian* by Meyer *et al.* [8]. All the presented tools focus mainly on the scripting, while ours provides more specialized visualization and uses the scripting as a means to explore beyond its existing visualization capabilities.

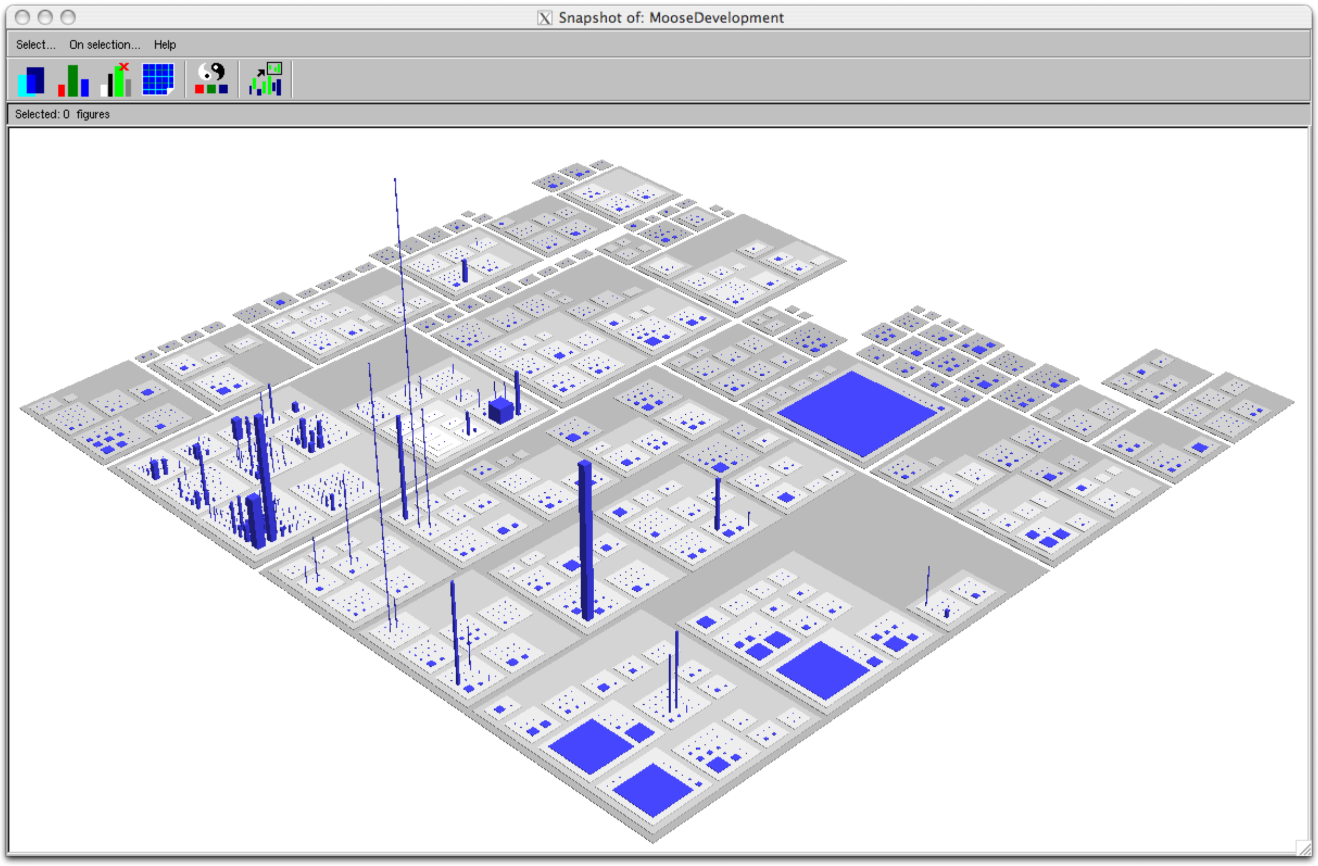


**Figure 1. Code city of MooseDevelopment**

## 3 Software Systems as Cities

We chose a city metaphor [14] for our visualizations because a city is an intuitive exploratory environment with a clear notion of locality.

Just like a city, a software system is a complex man-made product which cannot be oversimplified and must be incrementally explored.

In our code cities, the classes are represented as buildings and the packages as districts. Some of the visual properties of the city artifacts carry information about the software element they represent.

Figure 1 illustrates such a city in which the buildings color is blue, their height represents the number of methods metric of the class, and the base size the number of attribute metric. The color of the districts depicts the nesting level of the package they represent, according to a color scheme, *e.g.,* ranging from dark gray for root packages to light gray deeply nested packages.

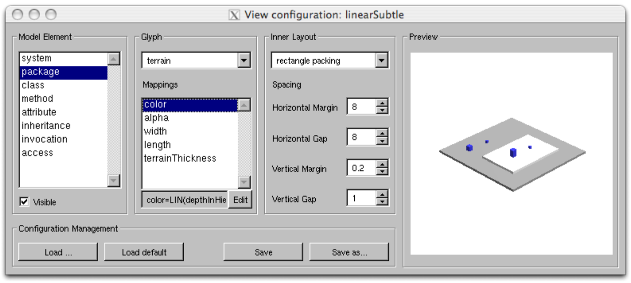
We support the creation of such fine-tuned visualizations through the view configuration mechanism, presented next.

## 4 View Configurations

A *view configuration* is a specification defining for each model element type (*e.g.,* class, package, method, inheritance):

1. the *visibility*, a boolean denoting whether it will be depicted or not,
2. the associated *glyph* type (what 3D construct is used for representation)
3. the *layout* to use when placing its components, and
4. the visual *mappers* associated with each property of the chosen glyph.

The user accesses the configuration mechanism by means of a view configuration user interface (See Figure 2), 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.


View configuration: linearSubtle

Glyph: terrain

color=LIN(depthInHie

Inner Layout: rectangle packing

Spacing

Horizontal Margin 8

Horizontal Gap 8

Vertical Margin 0.2

Vertical Gap 1

[✓] Visible

**Figure 2. View configuration user interface**

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. In the early versions of CodeCity, the view configurations saved by the user were serialized to files, which were not stored by Smalltalk's versioning system. Moreover, a renaming of a class which was part of the configuration (*e.g.,* any of the mappers, glyphs, layouts) would break the saved configurations.

To address these issues, we made a first step towards scripting, by exporting the configurations as scripts, *i.e.,* source code which evaluated produces valid configuration objects. These scripts are saved as class-side methods in a class serving as view configuration repository, and thus are easily tracked with Store. Because every object which is part of the view configuration mechanism knows how to generate its own building script, a view configuration script is just a composition of basic object building scripts.

Besides the view configuration mechanism, there is another important part in constructing a visualization: the builder. The role of the builder is to assemble a visualization based on the data to visualize (*i.e.,* a Moose model of a system) and on a view configuration which describes what to see and how to see it. The builder in our case incorporates knowledge about how the model is organized (*e.g.,* packages can contain packages and classes, classes contain methods and attributes), which allows them to build and layout the elements in the right order. This kind of information can also be obtained by reading the annotations of the meta-model, if such annotations are available.

## 5 Scripting visualizations

Scripting provides a mechanism for prototyping new visualizations and experiment their feasibility before fully embedding them in the tool's user interface. The scripting language we devised allows shortcutting the builder's part of the builder, manually specifying the view configuration, and visualizing any kind of data. The only downside is that scripting ad-hoc visualizations in CodeCity requires knowledge about both the meta-model and the basic constructs of the scripting language.

Similar to Mondrian's Easel [8], our scripting interface (See Figure 3) is made of three parts: a variable list (bottom left), a code editor for the script (bottom right), and the visualization generated by running the script (top). The example script shows the classes with more than 15 methods in MooseDevelopment as terrain glyphs and their methods built on top of them using a layout called progressive bricks. The methods are colored in dark red and the ones of the meta-classes are semi-transparent. The buildings representing classes are connected by blue edges representing inheritance relationships.

## 6 Conclusions

We presented an experience report on extending an existing 3D visualization tool with scripting facilities. Although not able to fully replace all the tool's parts, the scripting is an efficient alternative to the view configuration and builder mechanisms and allows experimenting with new visualizations before integrating them.

**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

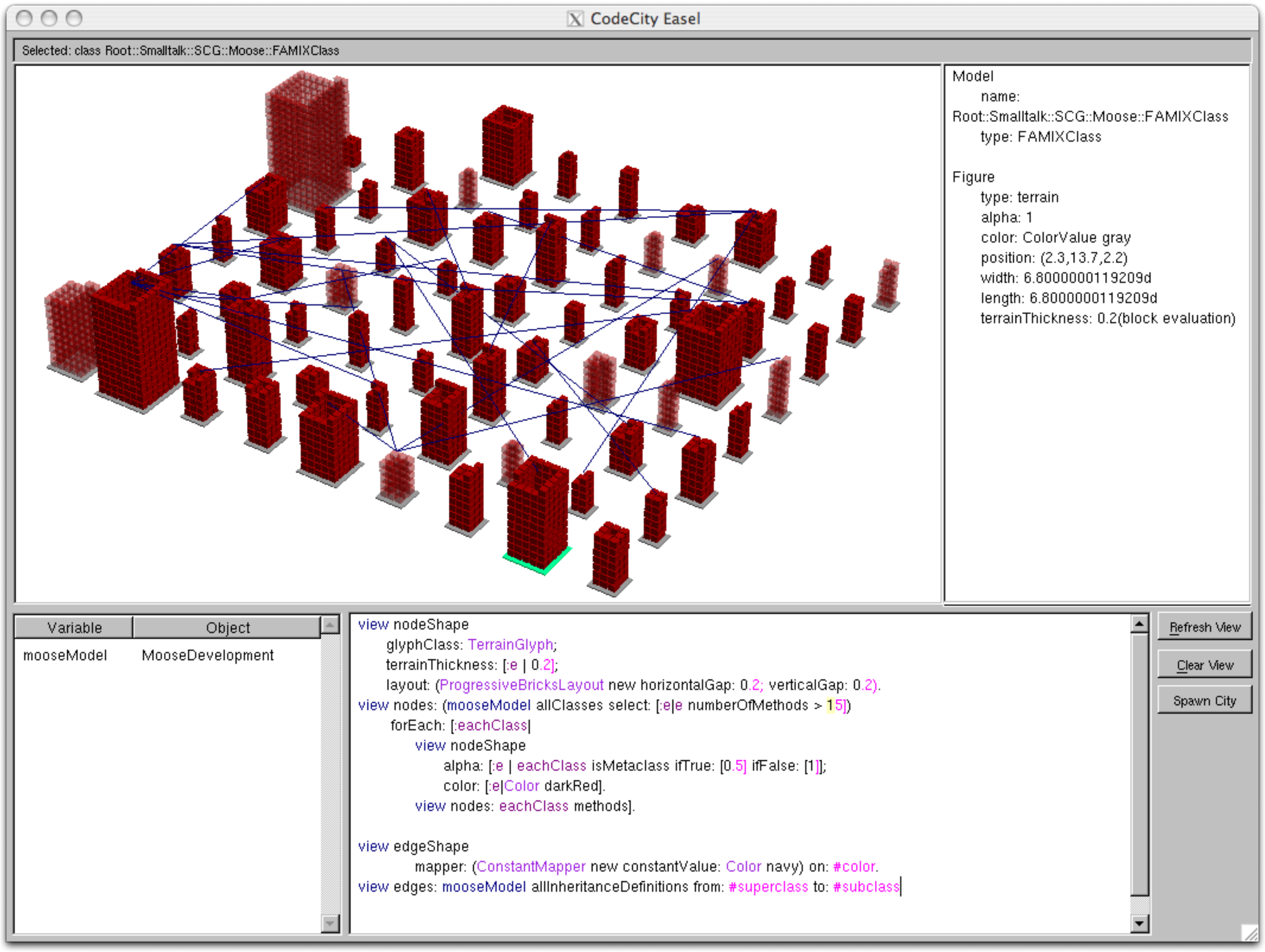
[1] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz. Software landscapes: Visualizing the structure of large software systems. In *VisSym 2004, Symposium on Visualization*, pages 261–266. Eurographics Association, 2004.

[2] 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.

[3] J.-M. Favre. Gsee: A generic software exploration environment. In *In Proceedings of the 9th International Workshop on Program Comprehension (IWPC 2001)*, pages 233–244. IEEE Computer Society, 2001.

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

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



Selected: class Root::Smalltalk::SCG::Moose::FAMIXClass

```
Model
     name:
Root::Smalltalk::SCG::Moose::FAMIXClass
     type: FAMIXClass

Figure
     type: terrain
     alpha: 1
     color: ColorValue gray
     position: (2.3,13.7,2.2)
     width: 6.8000000119209d
     length: 6.8000000119209d
     terrainThickness: 0.2(block evaluation)
```

| Variable | Object |
|---|---|
| mooseModel | MooseDevelopment |

```
view nodeShape
    glyphClass: TerrainGlyph;
    terrainThickness: [:e | 0.2];
    layout: (ProgressiveBricksLayout new horizontalGap: 0.2; verticalGap: 0.2).
view nodes: (mooseModel allClasses select: [:e|e numberOfMethods > 15])
     forEach: [:eachClass|
         view nodeShape
             alpha: [:e | eachClass isMetaclass ifTrue: [0.5] ifFalse: [1]];
             color: [:e|Color darkRed].
         view nodes: eachClass methods].

view edgeShape
         mapper: (ConstantMapper new constantValue: Color navy) on: #color.
view edges: mooseModel allInheritanceDefinitions from: #superclass to: #subclass
```

**Figure 3.** CODECITY**'s scripting interface**

[6] 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.

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

[8] M. Meyer, T. Gîrba, and M. Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis 2006)*, pages 135–144. ACM Press, 2006.

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

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

[11] S. Tilley. Domain-retargetable reverse engineering. ii. personalized user interfaces. In *Proceedings of 10th IEEE International Conference on Software Maintenance (ICSM'94)*, pages 336–342. IEEE Computer Society Press, 1994.

[12] S. Tilley, H. Muller, M. Whitney, and K. Wong. Domain-retargetable reverse engineering. In *Proceedings of 9th IEEE International Conference on Software Maintenance (ICSM'93)*, pages 142–151. IEEE Computer Society Press, 1993.

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

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

[15] R. Wettel and M. Lanza. Visual exploration of large-scale system evolution. In *Proceedings of WCRE 2008 (15th Working Conference on Reverse Engineering)*, pages xxx–xxx. IEEE CS Press, 2008.