Università
della
Svizzera
italiana

Facoltà
di scienze
informatiche

Bachelor Project

Citylyzer

A 3D Visualization Plug-in for Eclipse

Andrea Biaggi

supervised by
Richard Wettel
Prof. Dr. Michele Lanza

Abstract

Understanding the code of large software systems is hard just by reading them. For this reason we need tool support that helps in software understanding. Visualization is one of the means to deal with software analysis and comprehension. Citylyzer is a 3D visualization tool that uses the city metaphor of CodeCity [WL07a, WL07b] to represent the code: classes are buildings and packages are districts. The tool we provide helps the software engineer in reverse engineering of the software. This is done in an Eclipse Plug-in, one of the most popular Java IDEs. Citylyzer is useful because we have an immediate idea of the system analyzed and the analysis is done within the IDE.

Acknowledgments

We would like to thank Michele Lanza for giving us the opportunity to work on this bachelor project which we find very interesting and fulfilling. We thank Richard Wettel, Jacopo Malnati and Alejandro Garcia for the precious help during the whole project.

In the end we would like to thank all the people that supported us during all the bachelor study: parents, relatives, friends, professors and rugby mates.

Contents

Chapter 1

Introduction

High-level software systems are complex and difficult to analyze and understand just by reading the code. Reading is time consuming and does not give an overall idea of how a system is structured.

Reading the code takes a considerable amount of time, understanding what we have read takes more time than just reading it.

In order to better understand the code we need to reach a level of software abstraction, this can be reached by software visualization. Visualization is a means, with which we can see information about the system otherwise not easy to perceive. Visualization allows to better understand the code without reading it.

We go in the direction of 3-Dimensional visualization. 3D visualization could lead to a better utilization of the space, since we can display more things exploiting the third dimension and we think that we could have better interactions. We will try to use the concept of the city metaphor like Richard Wettel's CodeCity. This will be integrated in the Eclipse framework (`www.eclipse.org`) as a plug-in.

The choice of an Eclipse plug-in, in opposition to the standalone application CodeCity, is because the Eclipse IDE is widely used and because we can see the code representation in the same place where we write the code.

1.1 Goals

We propose a 3D visualization tool that helps the programmer to better understand the code. This tool will be integrated in the Eclipse IDE as a plug-in.

1.2 Structure of the Document

In Chapter 2 we describe the problem we want to solve.
In Chapter 3 we propose our solution to the problem.
In Chapter 4 we validate our solution.
In Chapter 5 conclusions about Citylyzer are drawn.
In Appendix A we explain implementation details of our solution.

Chapter 2

Problem & Related Work

2.1 Problem

Software engineers who have to extend and maintain software systems have to understand them first. Software systems are complex artifacts with relations between the elements inside the system.

Reverse engineering means recovering the original design starting from the code, it is useful for comprehension and analysis of software systems. Reverse engineering them is important for software industry, because software understanding is costly in terms of time, money and work.

In order to simplify software complexity and increase system understandability we need strong tool support. Visualization can be one means for achieving the goal of code comprehension.

Software is not tangible. Software visualization abstracts concepts of classes, packages, (relations), interactions to a concrete and understandable picture. This picture displays some proprieties of the software system. Visualization in general is needed to express and simplify an abstract concept or a real object in a way that the ”user” of that visualization can easily comprehend what he is seeing and he can easily spot patterns while looking at the visualization.

Our choice is to use 3D software visualization based on a city metaphor. A city metaphor is easy for the user to be utilized, because city structures are well known and can be understood by everyone. In our approach we exploit the third dimension. We use a third dimension order to compact our view and to represent more relations than in classical 2D software visualization.

Most of the applications provided for software visualization are standalone applications. We propose an Eclipse plug-in integrated in the IDE. so that the user can analyze the software while he is writing it.

2.2 Related work

In the following pages we present some 3D approaches and we focus on the ones that are related to our work.

2.2.1 3D visualizations

Information Pyramids

K. Andrews *et. al.* propose a 3D visualization of large hierarchies [AWP97], in their case they focus mainly on the filesystem. The hierarchy can be viewed as a tree, the root of the tree is mapped to a plateau, smaller plateaus arranged on top of the root represent the subtrees, obviously we can have more than two level plateaus, they are arranged in order to look similar to a pyramid. Leaves (representing files, documents, ...) are mapped to icons with different color depending on the file type and different height depending on file size (as shown in Figure 2.1).

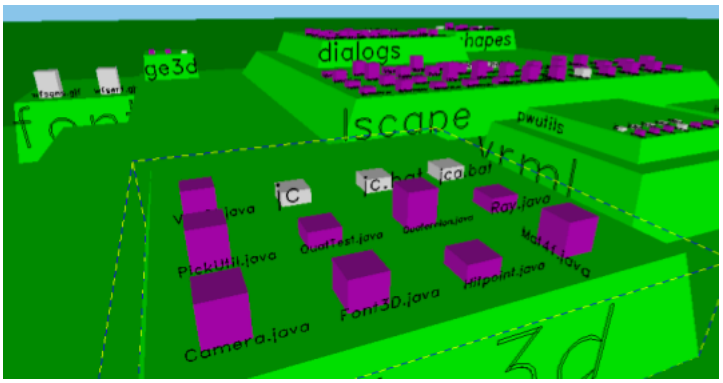

Figure 2.1: Detailed view of Information Pyramid applied to 3dexpl

This visualization can be applied to the object-oriented world by representing packages as plateaus and classes as icons, mapping for example the color to the number of lines and the height to the number of methods.

Software Landscapes

M.Balzer *et. al.* propose a 3D visualization using landscape metaphor [BNDL04]. It combines 3D images of landscape elements, customized layouts and hierarchical interconnection network. Landscapes are similar to physical environment (for example 2D surface like the earth with 3D elements such as buildings, mountains, ...). A typical aspect of the landscape metaphor is the hierarchy of abstraction levels.

In Software Landscape the hierarchy of packages is represented with nested spheres where the outermost one stands for the root package in the hierarchy. The spheres are positioned in circle inside a two-dimensional plane as shown in Figure 2.2.

In each sphere there is a plane on which classes are arranged. Each platform represent a landscape. A class is represented as a disc in the package landscape. In those discs, methods and attributes are placed inside as cuboids as you can see in Figure 2.3

For relations between classes the paper propose to use the hierarchical net (Figure 2.4). The relations are not direct but they are routed to the package and superpackage. The type and direction of the relation is represented by color brightness and gradient.

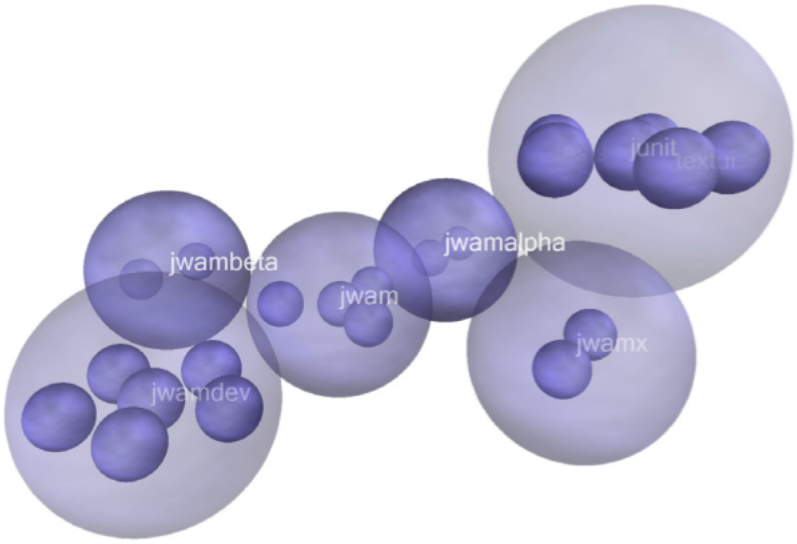

Figure 2.2: Package representation of "JWAM 1.6" with Software Landscape

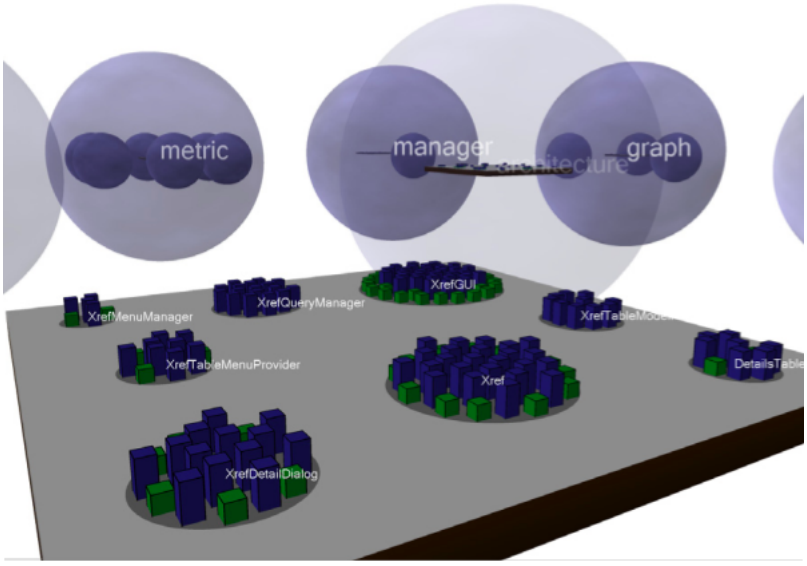

Figure 2.3: Close view of "SystemX" and class representations with Software Landscape

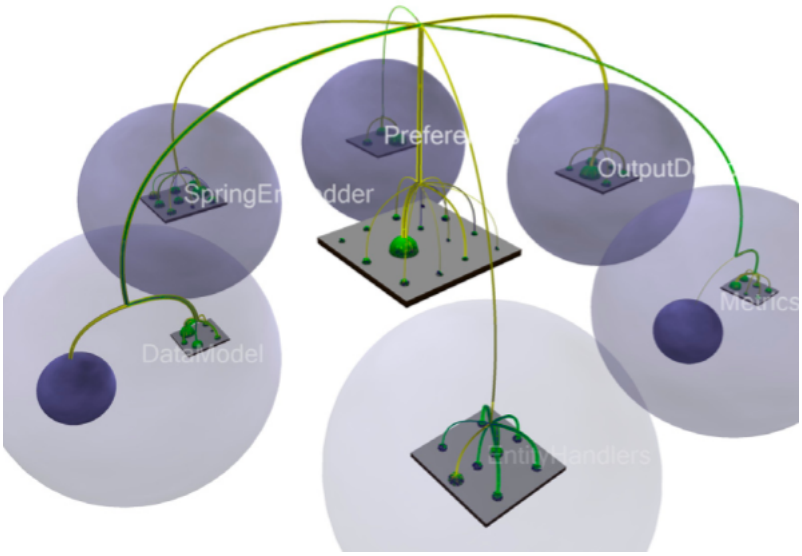

Figure 2.4: Relations visualization of "CrocoCosmos" using Hierarchical Net with Software Landscape

The landscape metaphor is a good tradeoff between information density and comprehensibility, but the representation is messy and it's easy to loose orientation in this system.

sv3D

A. Marcus *et. al.* present a framework for representing 3D visualization.[MFM03] This framework, called sv3D is a generic tool in which a user is free to provide the desired mapping of the visualization.

The tool has some set of predefined mappings.The default mapping of the tool maps each class in the system to a poly cylinder container in which there are many poly cylinders, each one is a line of code. Four others attributes can be mapped in the visualization in the following poly cylinder characteristics: color on the positive axis (for reference axis we use z, the one from top to bottom), color on the negative axis, height (positive axis) and depth (negative axis). Other propieties can be mapped to the poly cylinder shape and position in the container. An exemplification of it can be seen in Figure 2.5.

2.2.2 City metaphors

Software World

Software world, proposed by C. Knight *et. al.*, represent the entire software system by a world[CKTM02, KM00]. Source files are mapped to cities, in which there are one or more districts that represent classes. Inside the district there are buildings that stand for methods. An example of a class can be seen in Figure 2.6. If a building is dark it means that it is a private method, alternatively if it is bright it means that is a public

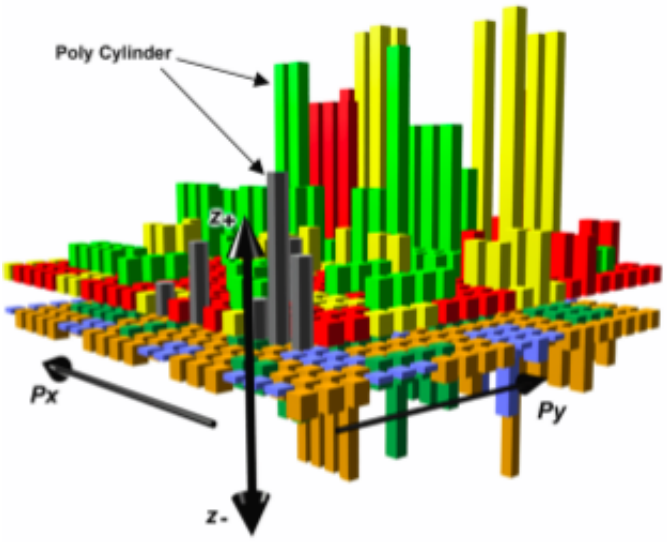

Figure 2.5: sv3D visualization example of poly cylinders and their proprieties

method. The height of the building maps to the number of lines of code and the input parameters of the methods are depicted with doors.

Figure 2.6: Class view with Software World visualization system

This approach has a fine level of granularity, applying this metaphor to large system would make it impossible to understand what the code is. Software world is not scalable.

A 3D Metaphor for Software Production Visualization

T. Panas *et. al.* propose the use of a city metaphor with high details [PBG03]. The city can be seen from the very inside of it (Figure 2.7) or we can have a top view of all the system (Figure 2.8).

Figure 2.7: Panas' 3D City metaphor street visualization

The metaphor is presented with buildings that represent java classes, each city is a package and trees, streets, lamps are just for decoration, no special meaning are given to them.

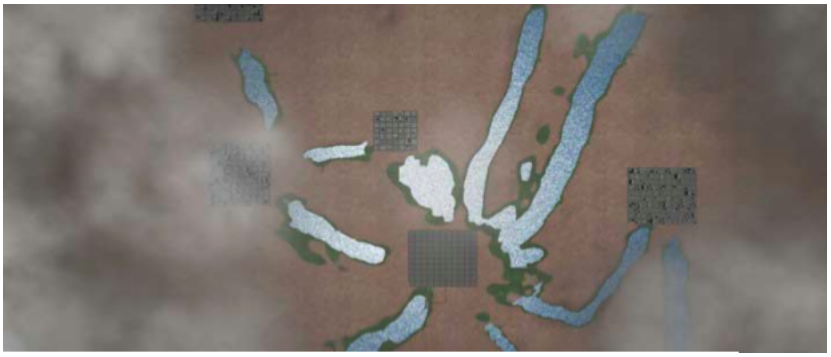

Figure 2.8: Panas' 3D City metaphor satellite visualization

The size of the building is mapped to the amount of lines of code of the class, the density of them in a particular region denotes the coupling between components. A building that looks old means that the source code of it needs to be refactored.

This metaphor shows also the interactions inside the program and how the relations between components are. Cars moving in the streets show a program run. Cars have different colors depending on which building they come from.

In the satellite view streets between cities are bidirectional calls, instead water are unidirectional calls (in the water we move by boats and not by cars). Clouds cover cities that are not in the current area of interest of the user.

With model we can see many other parameters: with the "Business Info" view the metaphor can represent the work distribution between users, classes that could be removed, parts that are highly modified, etc. with coloring, highlighting, etc..

The model presented by Panas *et. al.* lacks of package hierarchies, some details are useless (trees, lamps,...), many metrics are missing (number of attributes, methods)

and some features are difficult to spot in the city. The authors propose only static ideas, none of the part of the dynamic interaction between classes are shown. This system is just an idea, it is not yet implemented.

CodeCity

CodeCity is designed and implemented by Richard Wettel *et. al.* ([WL07a, WL07b]). This metaphor is our inspiration for the Eclipse Plugin. Details about it are explained in Chapter 3 because the same metaphor and mappings are used in ours Citylyzer. The classes are represented by buildings (width and length mapped to number of attributes and height mapped to number of methods) and districts, containing buildings or districts, are packages. The "default" view of CodeCity can be seen in Figure 2.9).

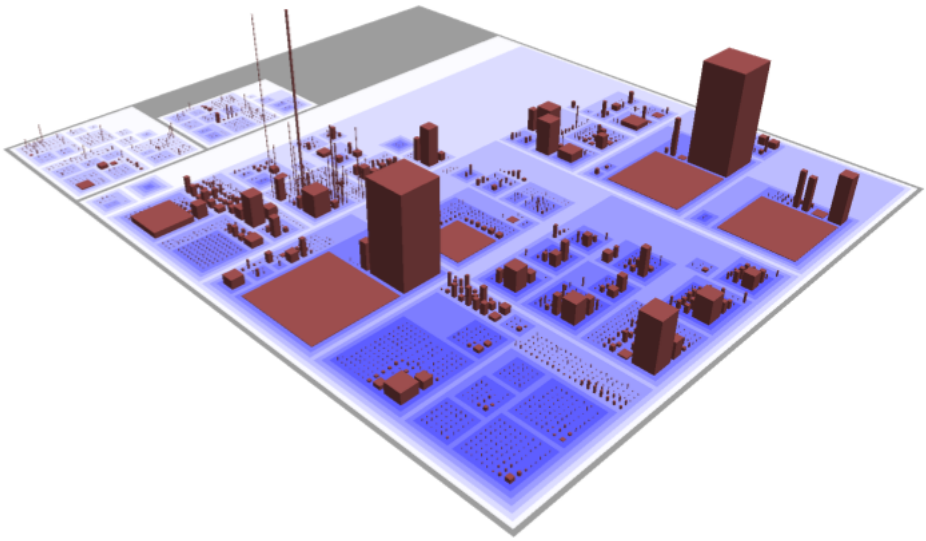

Figure 2.9: ArgoUML visualized with CodeCity

CodeCity has a lot of mappings that are not explained in our solution. The additional mappings can be used to see the system at different granularity levels.

Chapter 3

Solution

3.1 Main Idea

The choice of 3D is given by the fact that we can use the third dimension in order to allow the representation of larger amount of code than a 2-Dimensional visualization support. The third dimension permits to show relations that in a 2D world would not have been possible.

As we explained in Section 2.2.2 we take inspiration from Wettel's CodeCity. His 3D visualization is based on a city metaphor used to represent software systems. The main idea is to port his idea of CodeCity into an Eclipse plug-in.

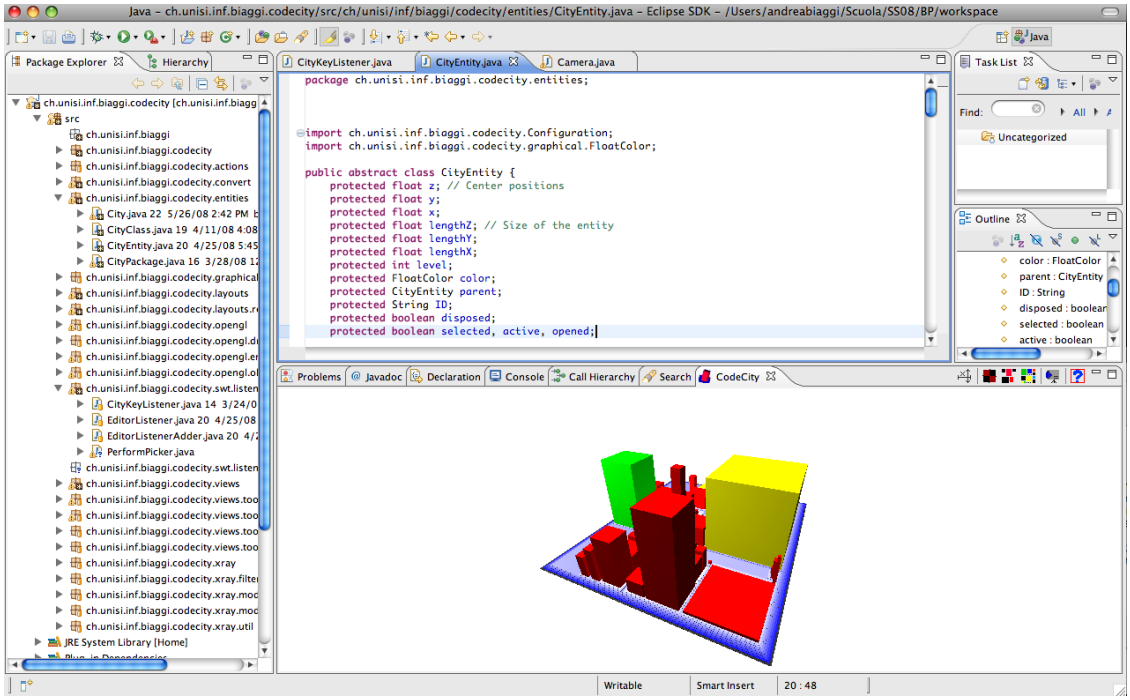

Figure 3.1: Citylyzer visualizing itself within the Eclipse IDE

A city is made of buildings (houses, skyscrapers, etc.), aggregations of buildings can

be seen as a districts, and aggregations of districts are bigger districts. Buildings and districts are the main component used in this metaphor. No other elements (i.e., trees) are present in our metaphor because they are not needed and they would add more confusion.

3.2 Metaphor metrics

Classes are mapped into buildings, A building is represented by a square cuboid, where the square faces are oriented at the top, and at the bottom. The side of the square represents the number of attributes in the class represented by it, the height of the parallelepiped represent how many method the class have. For a better understanding we can see Figure 3.2

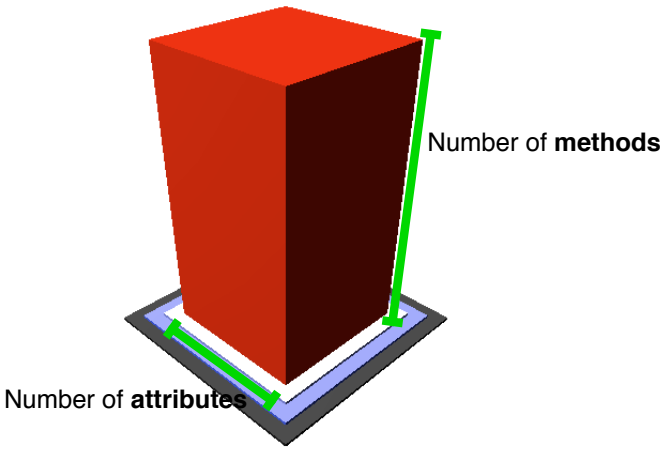

Figure 3.2: Building metrics

Packages are mapped to districts. Districts have no special meaning in the city, they are just districts containing buildings and/or other districts. Districts, although they have no special meaning, should group together buildings with similar semantical meaning. They are hierarchically disposed in order to reproduce the original package hierarchy in the software system (see Figure 3.3).

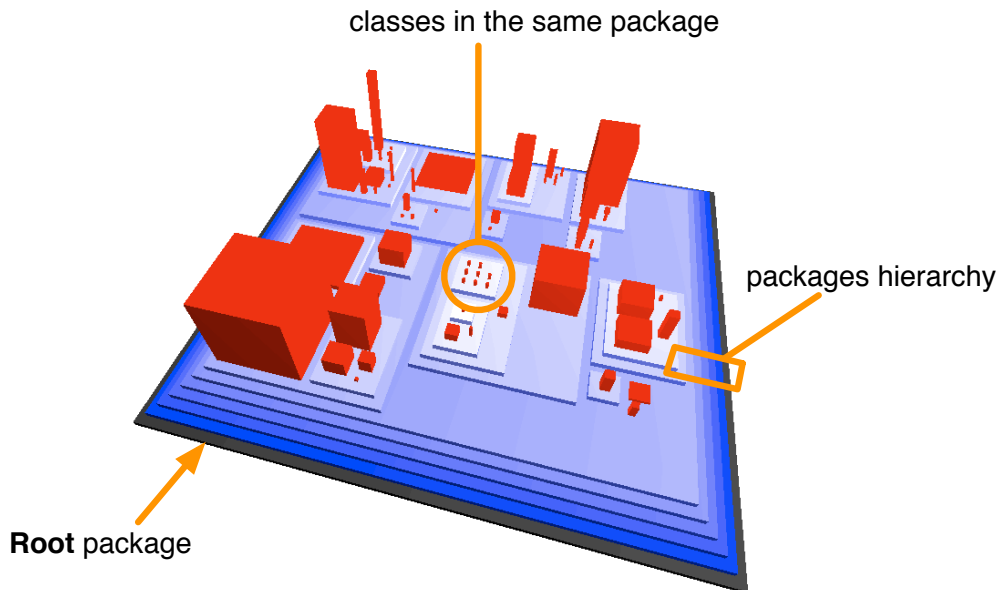

Figure 3.3: Explanation of the packages hierarchy

The root (dark grey district) represent the district container on which top all the city metaphor is presented to the user.

The colors of the buildings depends on the editor window of Eclipse. The standard color of a building is red, but if a class file is open his corespondent building becomes yellow. Instead if we are working on a class the building representing that class is colored in green.

3.3 User Interface

The view of Citylyzer has 6 buttons as shown in the Figure 3.4. Those buttons allow the interaction between the user and the system. Each action chosen by the user, through the a button, is performed in realtime.

Figure 3.4: Buttons on the Citylyzer view

The buttons have the following meaning:

1. Opens a dialog in which the user can choose the district thickness and the gap between the buildings, districts, etc..
2. Applies the chessboard layout to the city (this layout and the other layouts are explained in Section 3.5).
3. Applies the quadratic layout to the city.
4. Applies the rectangle packing layout to the city.
5. Places the camera in the default spot (45° rotation on the X and Y axis, distance from the center as the average of width and length, and looking to the center of the city).
6. Opens an help message that explains the keys for the city navigation (extended explanation of navigation can be found in Section 3.4.

3.4 Navigability

The navigability in the city is guaranteed by a camera that can move around. In our solution the camera responds to 8 key combinations, but other key combinations can be added easily to the code.

The four arrow keys rotate the camera around the city. We can imagine an hemisphere built on top of the root district in the city, having as center the center of the city and the radius the distance between the camera and the center. The camera moves only on the surface of this hemisphere when rotating.

The other four key combinations are created by pressing simultaneously the SHIFT key and (one of) the four arrow keys. SHIFT and the up arrow allows to zoom in, in opposition to SHIFT and down arrow that is the key combination for zooming out; both actions are made by come closer or move away from where the camera is looking at. SHIFT key with right or left allows to move the camera right or left.

This 8 key combination guarantee an almost full liberty of navigability in the city. Some camera movement are missing (i.e., move camera to the front or to the back) but they can be simulated using two or more of the basic combinations we have explained above.

3.5 Positioning algorithms

3.5.1 Chessboard layout

This is one of the simplest layout. Setting initial values:

n =numbers of elements
$s = \lceil \sqrt{n} \rceil$ side elements
Number of squares: $s \times s$
Size of a square: $max($ largest x among all elements, largest y among all elements$)$.
For each element E_j the algorithm executes the following steps:

1. If E_j is a container, execute the layout on it.

2. Take the square with coordinates: $x = j/s$ (int division), $y = j$ mod s.

3. Insert E_j in the middle of the square.

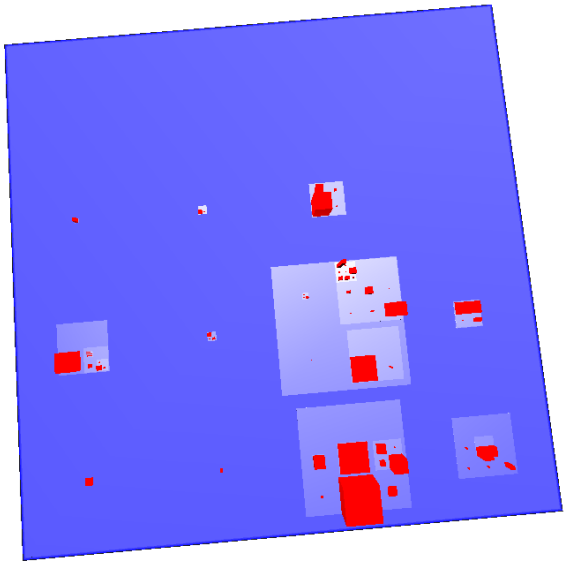

Figure 3.5: Application of Chessboard layout on Citylyzer itself

3.5.2 Quadratic layout

The goal of this layout is to pack the elements together with a simple algorithm. How the algorithm works is explained in the following pseudocode. Setting initial values:

n =numbers of elements
m = round$(\sqrt{n}) = \lfloor \sqrt{n} + 0.5 \rfloor$ number of elements per row
For each element E_j the algorithm executes the following steps:

1. If E_j is a container, execute the layout on it.

2. Place the element E_j

3. If $(j + 1)$ mod $m = 0$ go to new line for the place for positioning the next element. Else the place will be next the one placed now.

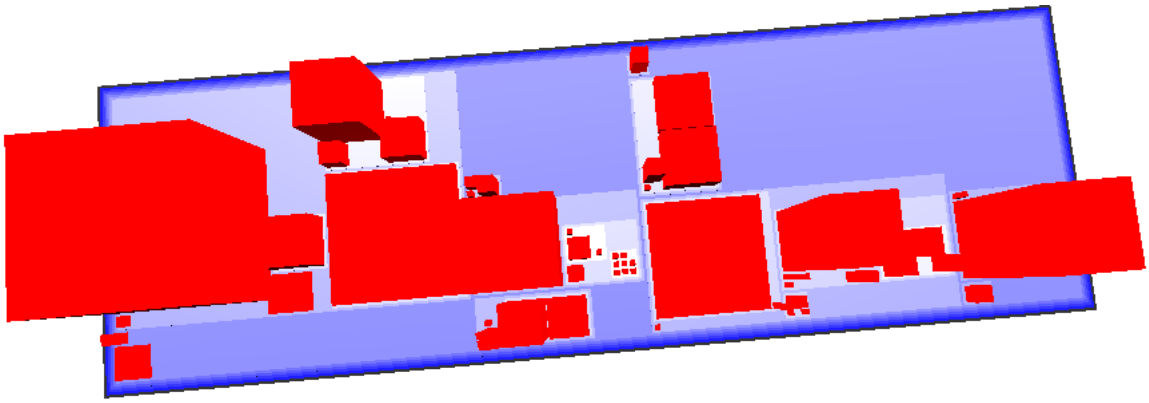

Figure 3.6: Application of Quadratic layout on Citylyzer

3.5.3 Rectangle packing layout

This layout seems to be the best one in order to pack more objects in a single place. The algorithm is the following (in high level).

We have a tree structure in order to position the elements. The nodes can be either a node with two pointers to nodes, or a node containing the element to place. Each node has a position, width and length.
For each element E_j the algorithm executes the following steps:

1. If E_j is a container, execute the layout on it

2. Select the candidate nodes (leaf nodes that are free and size at least large as E_j)

3. Choose from candidate nodes the node N in which to place E_j. There are two criteria for the choice.
   First precedence goes to nodes which, by placing E_j do not cause the expansion of covered area: $Covered.size_{before} = Covered.size_{after}$ Among those node we select the one who's rectangle is closer to E_j.
   If is not possible to find a node. We expand in the way that the covered area is closer to a square. We choose the node where the function
   $f = |Covered.size.x_{after} - Covered.size.y_{after}|$ is minimal.

4. If the node N is exactly the size of E_j, then N becomes the target node N_j. Otherwise split N to isolate the amount space needed by E_j into a single node N_j, which can be done in 2 cuts. We chose the direction of the first cut so that the space remaining in the other node is maximal.

5. Translate E_j to the position given by N_j

6. If expansion occurred: $Covered.size := Covered.size + expansion$

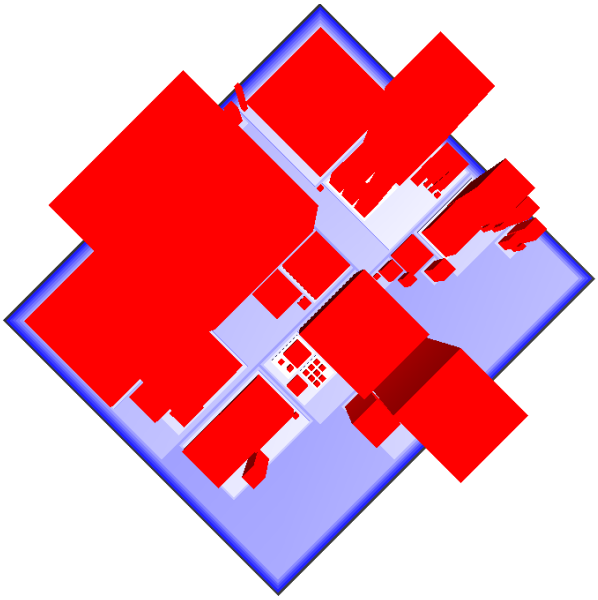

Figure 3.7: Application of Rectangle packing layout on Citylyzer

3.6 The plug-in

Citylyzer can be downloaded from `http://atelier.inf.unisi.ch/~biaggia/citylyzer/`. The plug-in requires other plugins in order to work, they are X-Ray and OpenGL SWT Bindings. How to download those plugins is explained in details in the website of Citylyzer.

The plug-in is created for the third version of Eclipse. Due to a problem with the OpenGL bindings (created on year 2005) the plug-in does not work under Intel based Macintosh OS X.

Chapter 4

Validation

4.1 Analyzing with Citylyzer

We use our tool to analyze itself and trying to see what we can understand just by looking at the visualization. We want to spot anomalies in the code, classes that have many responsibilities.

The goal of this task is to see the usefulness of our Eclipse plug-in and the correctness of it.

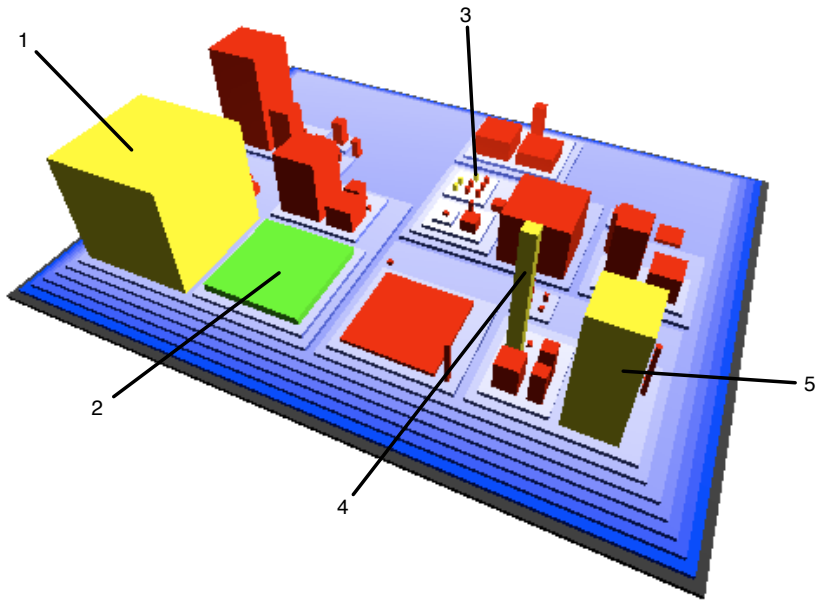

Figure 4.1: Citylyzer visualizing itself in order to validate the plug-in

	Class	Package	methods	attributes
1	Camera	**x**.codecity.opengl	26	28
2	Light	**x**.codecity.opengl.environment	1	24
3	ResetCameraButton	**x**.codecity.views.toolbar.actions	1	1
4	DependencyBuilder	**x**.codecity.xray.model.core	33	3
5	ClassRepresentation	**x**.codecity.xray.model	28	10

x stands for ch.unisi.inf.biaggi

We can compare the visual data given by Figure 4.1 and the data of the table above. It is easy to see that the metrics are mapped to the right values. For example from the Light class and DependencyBuilder class we can see that the attributes are mapped to the width and length of the building, instead the number of methods are mapped to the height of the building.

4.2 Application on large systems

We apply our Citylyzer on 2 large system in order to assess if our tool is scalable or not. We apply it on ArgoUML (Figure 4.2) and on Azureus (Figure 4.3). Then we analyze some characteristics of ArgoUML that can be discovered by our tool.

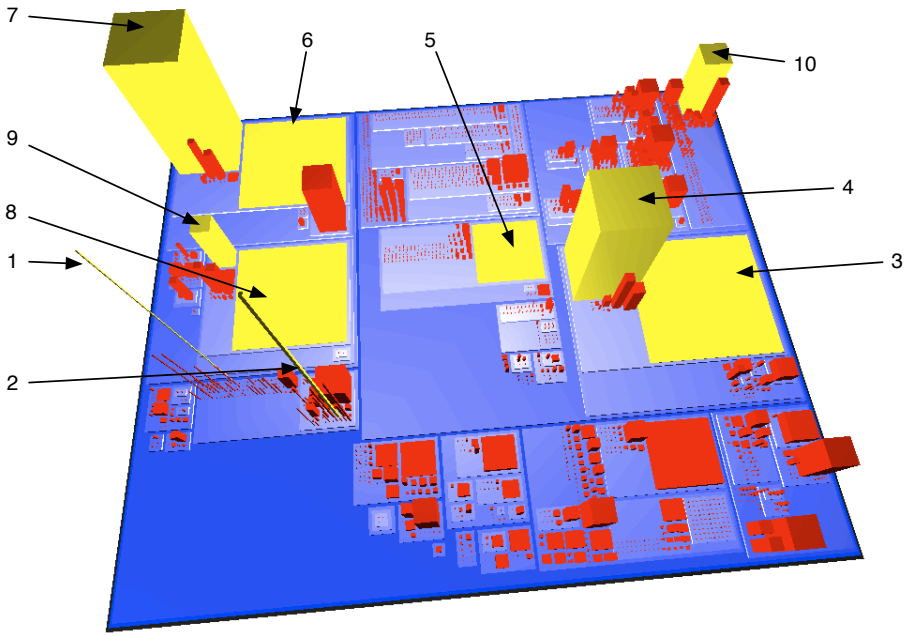

Figure 4.2: Analysis of ArgoUML with some points of interests.

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

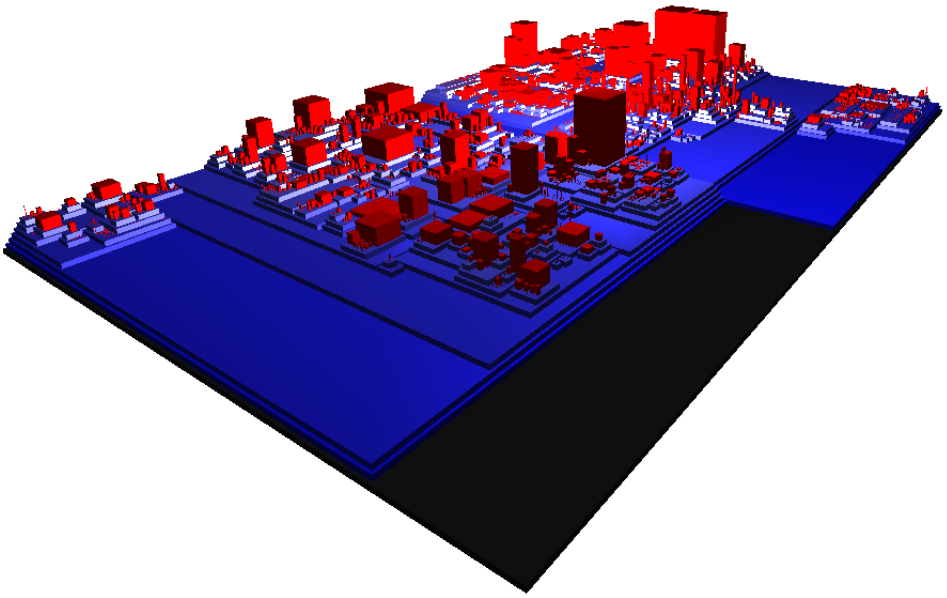

Figure 4.3: Screenshot of our tool analyzing Azureus.

4.3 Analysis and conclusions of validation

We can clearly assess that out tool has the propriety of scalability. In fact we had applied it on two of the largest systems in the Java world. The understandability and structure of the data is still the same as the one of small systems. It does not matter how much large a software system is, our tool can be applied to it without problems and it can handle all the amount of code.

From the use of Citylyzer we can easily spot classes with high responsibilities, classes with high number of attributes and give a special meaning to them. In the next page we explain what we can easily understand from ArgoUML.

Interfaces and classes that implement those interfaces have high buildings and usually thin. You can see an example in Figure 4.2 where the interface labelled with 1 and

the class (number 2) have an high number of methods (337 for the interface and 349 for the class) and a low number of attributes (1 for the interface and 3 for the class). This interface should be modified and reduced to multiple interfaces but it is hard to change because it has too much responsibility.

Large but flat buildings usually express ”configuration” classes. Those classes have attributes storing types and constants. A clear examples could be in Figure 4.2 where 3, 6 and 8 are building representing classes that contain attributes needed by the classes represented by respectively 4, 7 and 9. Another example of configuration classes is class 5 where it contains all the attributes and settings needed to initialize ArgoUML.

Chapter 5

Conclusions

5.1 Reached goals

We have built Citylyzer, a 3D visualization plug-in for Eclipse IDE. The tool works in Eclipse with some minor problem (does not work under Macintosh Intels).

Our tool has the following proprieties: scalability (Chapter 4), navigability (Section 3.4) and orientation. Orientation is given by having the navigation always starting in the same place, and we can return back to the starting point anytime, anywhere.

The interaction propriety does not work as prefixed. It is only one-way. That means that from the code we can interact with the model, but we cannot from the model go to the code or the details of the code. This problem will be faced in the future.

5.2 Problems encountered

We encountered a lot of problems in the 3D displaying part. If we did not had those problems, Citylyzer would be much more functional with more features that for now are put in the section about the future work. Other small problems occurred during the path, but they were not so crucial in terms of the outcome.

5.3 Future work

The plug-in needs to be improved in the near future to adapt to users needs. We have in mind some upgrades for the next versions.

Packages and classes have to be recognizable in the view, for this feature we have two main solution to choose from. The first is to label packages and classes as done in Information Pyramids (Section 2.2.1). The second solution, more user friendly than the previous one, is to allow the selection of the buildings, and when a building is selected informations are displayed on the screen or in a dialog. This can only be done by changing the underlying OpenGL implementation with GEF3D [Eng] or with another OpenGL implementation that supports what we need.

More metrics could be displayed in the city, for example we can use building transparency and/or building darkness/brightness in order to map other values (i.e., lines of code).

We can add a customization panel in which the user can specify the colors of the elements, which metric is mapped to which building attribute.

The final and interesting future work could be to integrate our Citylyzer with Jacopo Malnati's X-ray and other visualization plug-ins in order to provide to the user a high variety of tools for visualization.

Appendix A

Implementation

A.1 UML Diagrams

From the Figure A.1 we can see a simplified UML Class diagram of the classes that models the city. Getters, setters, not important methods, not important attributes and Layout subclasses have been removed in order to ease the understanding of how the city is modeled by us.

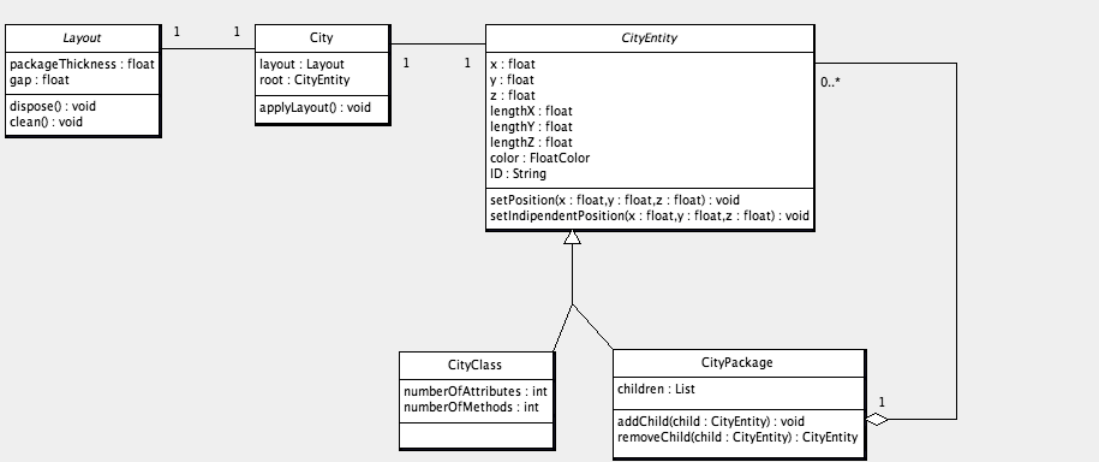

Figure A.1: Simplified UML Class Diagram of the model of the city

This UML diagram gives the main idea how the core is structured. There's the main component that is the City that contains a tree of CityEntities. The tree represents the hierarchical structure of packages and sub-packages. The Layouts inheriting from Layout contain the behavior on how the entities in the city should be placed.

Other classes that have high importance in our model are the Drawer that has a city as parameter. The Drawer is graphical implementation dependent. It is necessary in order to draw the entire city on the Eclipse view.

From the choice of the graphical implementation many other classes have to exist (i.e., the Camera for navigability). Other classes can be created for handling interaction

between plug-ins, models, framework, etc..

A.2 Plug-in Interactions

Citylyzer plug-in interacts with the Eclipse IDE, Jacopo Malnati's X-Ray plug-in and a 3-D visualization tool/plugin in order to display the citylyzed system. Our choice of 3D visualization plug-in is explained in Section A.3. The interactions, at high level, between all the plugins work as shown in Figure A.2.

In the next pages we explain in details some of those interactions in order to see what happens behind the scenes.

A.2.1 X-Ray model creation

Malnati's X-Ray create a model of the code starting from the source files [Mal07]. The source files are analyzed and important values are gathered from it. From the source files X-Ray creates a model in which classes are organized in a hierarchy based on inheritance. After the tree is built metrics are applied to his representation and dependencies are computed. Figure A.3 shows how this is done.

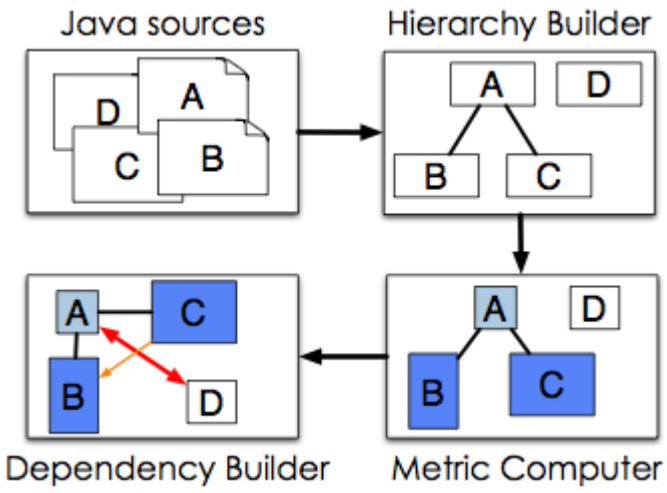

Figure A.3: How X-Ray gathers informations

For our solution we do not need all the metrics given by X-Ray (i.e., we do not use the lines of code) and we do not exploit the dependencies.

A.2.2 City creation

After we have the X-Ray model, we create our city representation of the classes. For this we have to convert his model to ours. We interface with his model through his effective model interface.

First from the X-Ray model we can get the list of all the packages, since they are not hierarchically organized we have to structure them. How is done can be read in

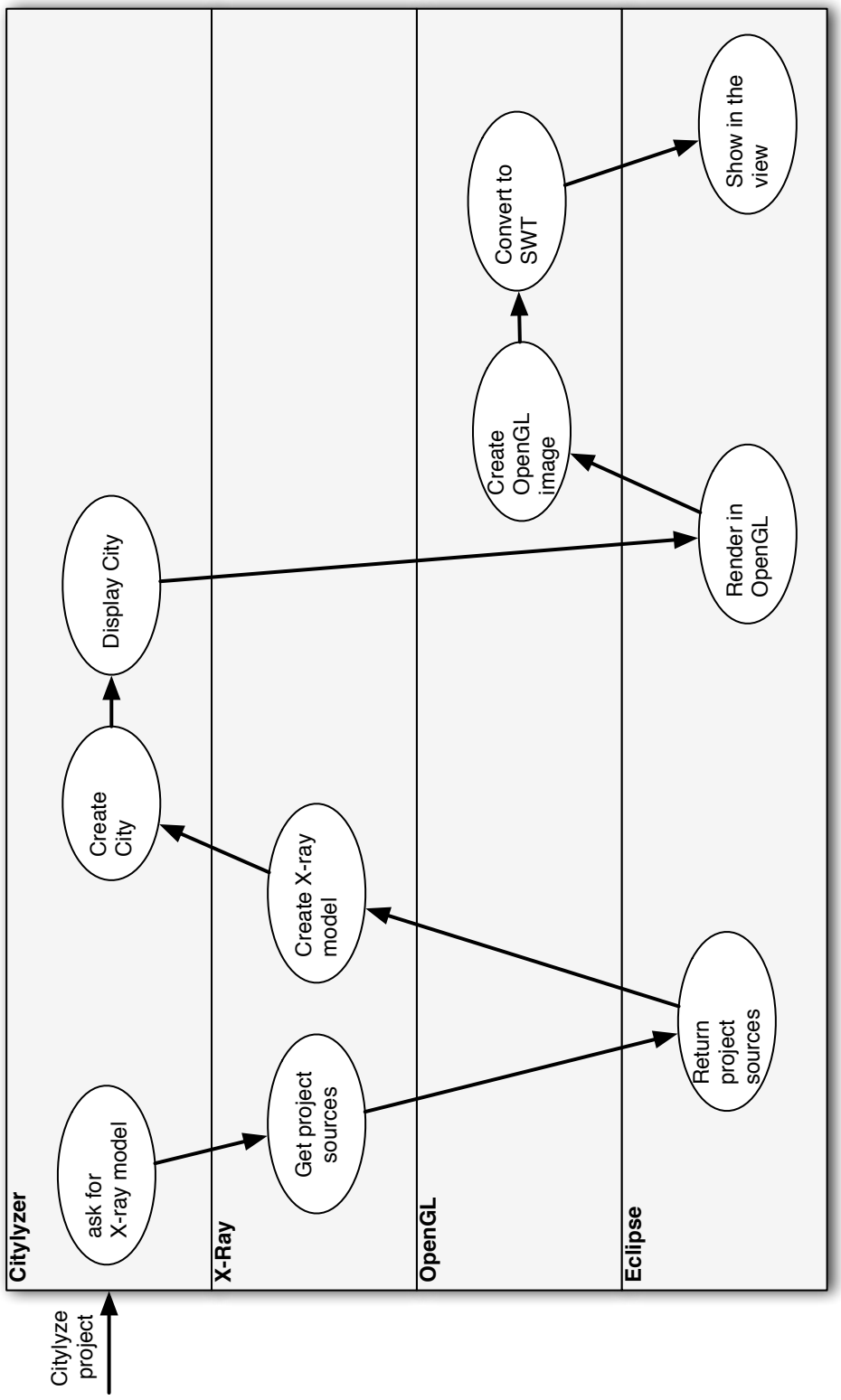

Figure A.2: Interaction beween Citylyzer, Eclipse, X-Ray and 3-D visualization

Section A.2.3. From this hierarchical X-Ray model of packages we create our districts and sub-districts hierarchy, just by mapping each level to a district.

Each package representation of X-Ray contains a list of classes. From each class we take the metrics we need in order to build our city, and we create buildings from them. The metrics taken into account are shown in Figure A.4.

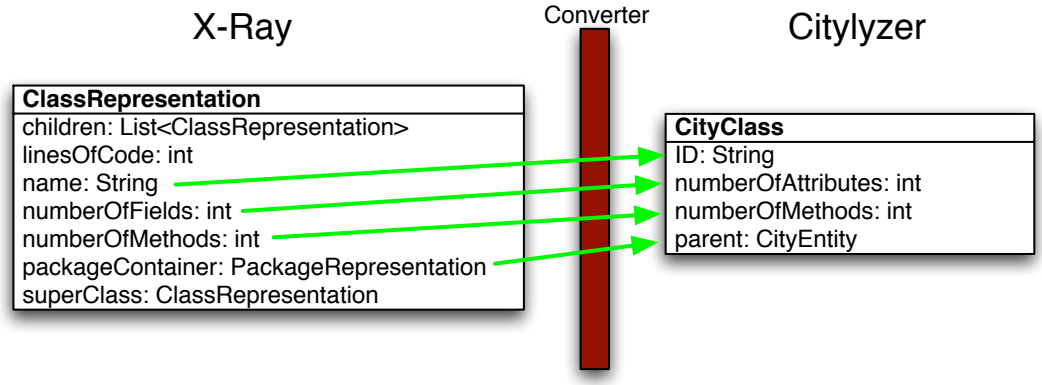

Figure A.4: Conversion from X-Ray class to Citylyzer class

A.2.3 Package hierarchy

From X-Ray plug-in we can retrieve only a packages list of what is in the system. The packages are not hierarchically structured. For that reason we had to find an algorithm to build the hierarchical representation of the packages.

In doing that operation we need an additional data structure that helps in reconstructing the package hierarchy of the system analyzed. This data structure is a tree in which nodes are labeled by strings and they can have from 0 to many children.

The algorithm works as follows:
Initial settings: currentNode ← root (root has an empty string label)
For each package in the package list:

1. Split the package name in tokens (the split char is ".")
2. for each token
   (a) tempNode ← currentNode.children.nodeWithLabel(token)
   (b) if tempNode ≠ `null` currentNode ← tempNode
   else currentNode ← new Node with label token
3. currentNode.addClassesRepresentations(package.getClassesRepresentation())
4. currentNode ← root

return root

An example of this algorithm working can be seen in Figure A.5 where as input we have the following list of packages (inserted in the tree in the same order as they appear in the list):

- `org.biaggi.model.class`
- `org.biaggi.view`
- `org.biaggi.model.package`

After the execution of the algorithm we have the packages organized in a hierarchical way.
Note: only the steps where a creation of a node occurs is drawn

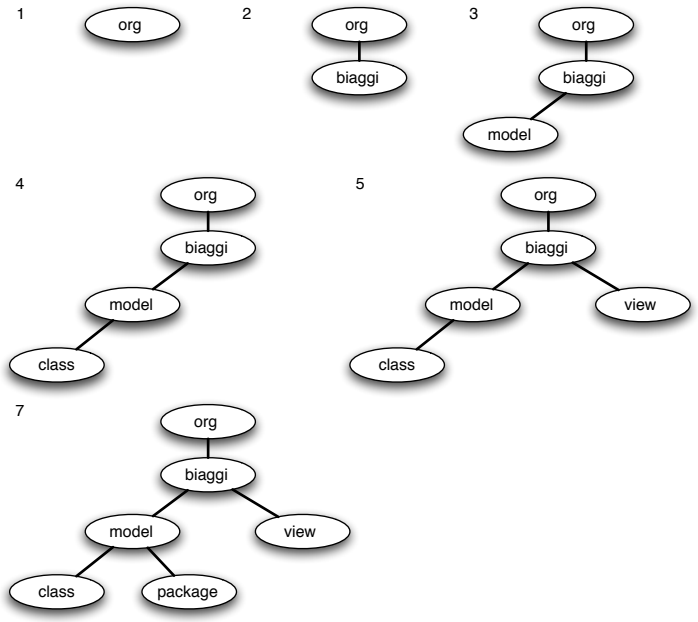

Figure A.5: Example of the hierarchy algorithm

A.3 OpenGL implementations

We have tried different openGL implementations in order to make it work in Eclipse IDE under Mac OS X. The only solution we have found working "properly" was a experimental deprecated OpenGL binding for SWT. `http://www.eclipse.org/swt/opengl/`

The big problem about this plug-in was the absence of documentation because some methods implemented by it differs from the standard openGL calls. In our specific case we could not use the openGL standard call `glSelectBuffer(int arg0, int[] arg1)` because in our implementation the method presents itself as `glSelectBuffer(int arg0, int arg1)`.

We also had the news that GEF3D will be released shortly. We had the bad luck to work on this plug-in before the release of it. For sure it would have been a good option for the visualization of the model in the Eclipse view.

List of Figures

Bibliography

[AWP97] Keith Andrews, Josef Wolte, and Michael Pichler. Information pyramids: A new approach to visualising large hierarchies. In *Proceedings of VIS 1997 (IEEE Visualization Conference)*, pages 49–52. IEEE CS, October 1997.

[BNDL04] Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software landscapes: Visualizing the structure of large software systems. In Oliver Deussen, Charles D. Hansen, Daniel A. Keim, and Dietmar Saupe, editors, *VisSym 2004, Symposium on Visualization, Konstanz, Germany, May 19-21, 2004*, pages 261–266. Eurographics Association, 2004.

[CKTM02] Stuart M. Charters, Claire Knight, Nigel Thomas, and Malcolm Munro. Visualisation for informed decision making; from code to components. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 765–772, New York, NY, USA, 2002. ACM.

[Eng] Lehrgebiet Software Engineering. Gef3d. `http://www.gef3d.org`.

[KM00] Claire Knight and Malcolm C. Munro. Virtual but visible software. In *IV*, pages 198–205, 2000.

[Mal07] Jacopo Malnati. X-ray - an eclipse plug-in for software visualization. Bachelor's thesis, University of Lugano, June 2007.

[MFM03] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3d representations for software visualization. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff, New York, NY, USA, 2003. ACM.

[PBG03] Thomas Panas, Rebecca Berrigan, and John Grundy. A 3d metaphor for software production visualization. In *IV '03: Proceedings of the Seventh International Conference on Information Visualization*, page 314, Washington, DC, USA, 2003. IEEE Computer Society.

[WL07a] Richard Wettel and Michele Lanza. Program comprehension through software habitability. In *Proceedings of ICPC 2007 (15th International Conference on Program Comprehension)*, pages 231–240, 2007.

[WL07b] Richard Wettel and Michele 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.