

Modelling Large Cities: Software Engineering

Nikolay Kazmin

*Computer Science
Jacobs University Bremen
Campus Ring 1
28759 Bremen
Germany*

Type: Guided Research Final Report

Date: May 12, 2008

Supervisor: Prof. L. Linsen

Executive Summary

There are a lot of existing Computer Aided Design (CAD) tools that help engineers and artists design rooms, buildings, or even whole residential areas, but there are very few that aid the design of an entire city. The proposed project aims to develop a tool that will enable users to model and then visualize large cities. The software engineering part of this project will have several major challenges. The first and most important will be the system design. Since the system will undergo numerous evolutions the design must be very extensible, it must allow easy integration of third party components into the system. The second challenge will be the graphical user interface; it must be very intuitive and should allow the user to very quickly get into the system and start using it right away. The last main challenge will be the definition of the file format that the system will use to save the modeled cities. It must be inherently extensible to allow for saving more complex data in the future.

1 Introduction

From the beginning it was clear that this project can have a lot of features and it would take many man-months to actually implement all the nice-to-haves. So we had two main tasks to complete during this guided research. The first one was to build the basic version of the system; the framework that will define the general behavior and provide the most valuable functionality. The second one was not so tangible, but much more important in the long term, we had to design the system in a way that extending it would be very easy and safe.

Therefore from the software engineering perspective the greatest challenge was extensibility. The design must be very robust, the data encapsulation must be meticulous and abstractions must be used whenever possible to allow for new components to be added. Designing for change is a very tricky business, there is always the danger from mistakes like the second-system-effect[1]. Over-designing can really ruin all the design efforts, it will lead to one "big pile"[1] of code that nobody really needs and uses and will add up to the complexity of the system so much that it can actually null all the efforts put in the design. But most importantly since we are not prophets we cannot anticipate all the possible extensions the system will have to endure, so we better be prepared to change the design when the new requirements come than to try to anticipate everything and over design the system. That's why when designing the City Modeling software I took the approach advocated by Kent Beck and the Extreme Programming[2] doctrine:

"You would introduce elements to the design only as they simplified existing code or made writing the next bit of code simpler."

2 Graphical User Interface

As soon as we decided that we are going to implement the system in Java we had to decide which graphical widget toolkit we are going to use. The two most widely adopted were Swing and SWT, and since JOGL integrated much better with Swing, that was our choice.

The main principle when building the graphical user interface was simplicity. We wanted the system to be as easy as possible to use so we decided to keep the user interface very clean and simple. In this section I will briefly describe the user experience in our application. I will omit the interactions with the modeling and the visualization canvases since it is more in the scope of Petar Dobrev's final report.

When the user starts the application there are only three buttons enabled, namely "New", "Open" and "Exit" 2.

So in the beginning the user has no other choice but to create a new city or open a previously created one. The open city dialog is pretty standard and I will not



Figure 1: Initial Buttons

discuss it here, I will directly go and explain the "New City Dialog"2.

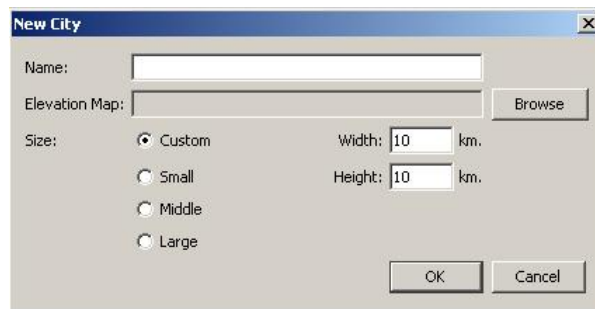


Figure 2: Initial Buttons

It appears when the user clicks on the new button in the main menu or just presses ctrl+N. The dialog asks for the most basic information required to create a city:

- *Name*: the name of the city, the dialog makes sure it is not empty.
- *Elevation Map*: the elevation map is optional, but if the user picks an elevation map then the dialog makes sure the picture is grayscale.
- *Size*: The size of the city in kilometers. The dialog provides some predefined city sizes as well as a custom size. The dialog also verifies the size of the dialog keeping it integer between 1 and 50.

When the new city is created the modeling perspective is enabled. Much more actions are now available and the user can start modeling the city. The first thing that the user sees is that the toolbar buttons are now enabled³. The first four buttons serve to switch between different modeling areas, namely: commercial, residential, industrial and highways. The very last button in the toolbar is for switching the perspective, it is a toggle button that switches between the modeling and visualization perspectives.



Figure 3: Toolbar Buttons

Actions are also made available in the main menu once the city is created, the user can now save the city or export the city model as an image.

3 City File Format

Being able to save the city model was one of the main requirements for the system. In order to do that we had to define file format that will be meet our needs and be extensible to easily accommodate change. XML based format seemed like the perfect candidate; it is inherently extensible, it is human readable so it is easy to understand and import into a third-party systems, and in the end it supports everything needed to save the city.

```
<?xml version="1.0" encoding="UTF-8" ?>
<city name="Dream City">
  <elMapFileName />
  <size>
    <x>15</x>
    <y>15</y>
  </size>
  <Elements>
    <area type="Commercial">
      <point>
        <x>0.08085106313228607</x>
        <y>0.9413533806800842</y>
      </point>
      <point>
        <x>0.0765957459807396</x>
        <y>0.33834588527679443</y>
      </point>
      .
      .
      .
    </area>
  </Elements>
</city>
```

Figure 4: City File Format

What we see in the picture⁴ is a snippet from a saved city file. The file extension of the saved cities is .cmx which stands for city model extension. As shown in the picture we save the name of the city, the size and the elevation map. Then what follows are the city elements; for each of them we save the type of the element and all the points that comprises it. How exactly the city and the elements are saved will be explained in the design section of this report.

The JDOM[9] library was used to actually create and parse the .cmx files, JDOM supports DTD validation so one of the future builds of the system can incorporate a DTD validation of the .cmx files before loading.

4 The System Design

The design was definitely the most challenging part of the project. In this section I will first introduce all the packages that define the system and the dependencies between them. After that I will go through every package explaining the challenges I faced and the solutions I provided.

4.1 Packages

I provide a UML package dependency diagram 4.1 as explained by Martin Fowler[4].

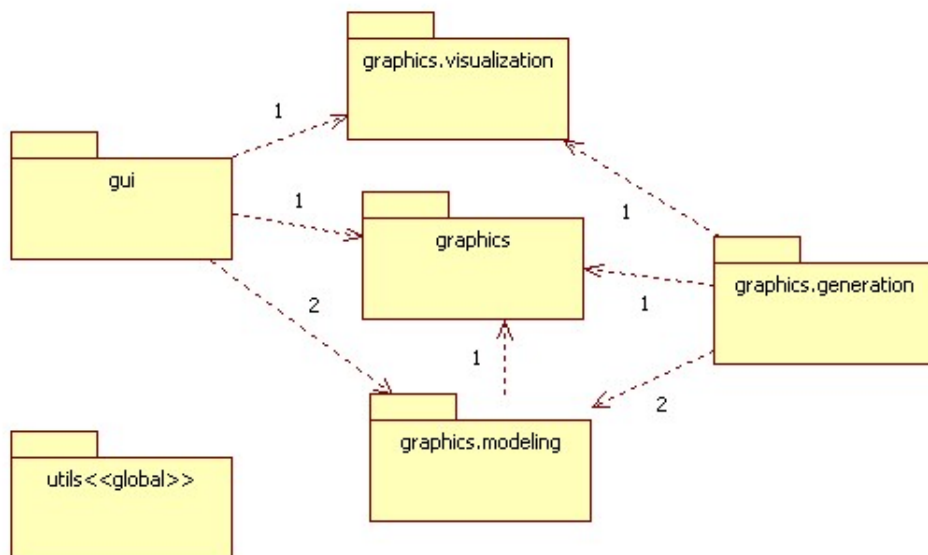


Figure 5: Package Dependency Diagram

The numbers above the dependencies are the number of classes whose interface define the dependency. For example if the number is 1 then the client package depends on only one class of the supplier package. This is not a standard notation but I included it to show that the packages are very loosely coupled. Another notable thing in the diagram is that there are no cyclic dependencies, this conforms to the best practices in software engineering.

4.2 The gui Package

The *gui* package consists of all the classes that provide graphical user interface functionality. There we can find the main frame, its element and all the dialogs. Each class in this package can be seen in the class diagram.

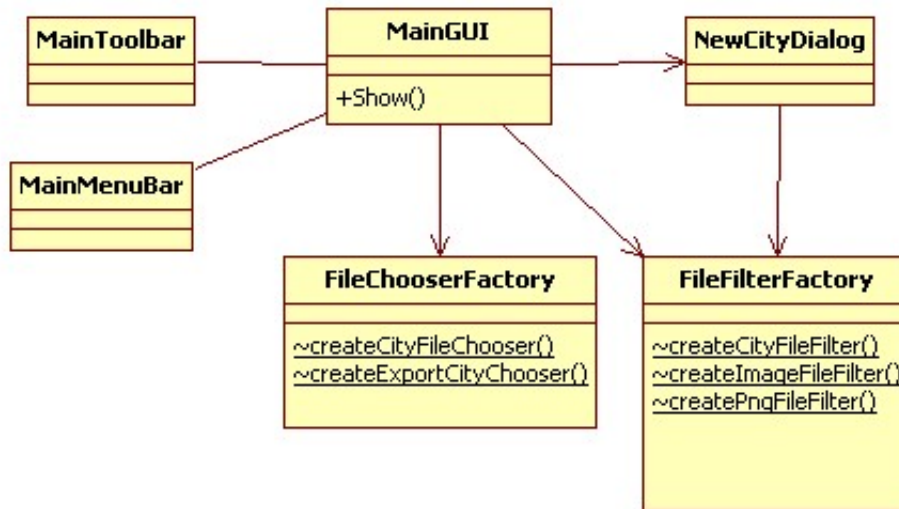


Figure 6: GUI Class Diagram

The design of the graphical user interface classes is pretty simple, no other package depends on the *gui* package so in future it can be freely changed if necessary. All the communication between the other packages and the *gui* takes place via a class in the *utils* package called *ConfigurationMananger*.

I have exported all the application logic of the main menu and the toolbar out of the *MainGUI* class into their own classes namely the *MainMenuBar* and the *MainToolBar*, but these are more or less trivial, the most interesting classes in this package are the two factory classes. They implement the "Factory Method" design pattern[3]. Both factories work very much in the same way, so I will explain only one of them in details. As a sidenote the difference is that one hides the file filter logic while the other hides the file chooser logic.

I have to use the *JFileChooser* dialogs provided by Swing many times, but I need some additional logic incorporated in it to meet my needs. I also don't want to hardcode that logic every time this dialog is shown. For example I don't want to write every time that the city file extension is *.cmx* or enumerate every time all the image extensions. If in the future we have multiple city formats or support more images, such approach will definitely make change very hard. That's why I use the factory classes which provide all the logic I want, but never leaks any implementational details outside of the factories. For instance the *FileChooserFactory*

has a *createCityFileChooser* factory method which returns a Swing object of type *JFileChooser*. As I said before the standart behavior of the *JFileChooser* does not provide the features I require, so the factory makes a private inner class that extends the *JFileChooser*, it then overrides all the necessary methods to provide the needed functionality and returns the object as a regular *JFileChooser*. No class outside the factory knows anything about the inner class or how it changes the behavior of the chooser. Another detail hidden by the factory is the file extension, the client just asks the factory to *createCityFileChooser* and never has to state anything about what files exactly can be selected by the chooser. Every detail about the city file is hidden there, which makes it very easy to change if we want to.

4.3 The graphics.modeling Package

The *graphics.modeling* package consists of all the classes that provide modeling functionality. These classes provide all the application logic while the user is in the modeling perspective. You can see a class diagram 4.3 of this package on the figure, for simplicity I have omitted most of the class methods.

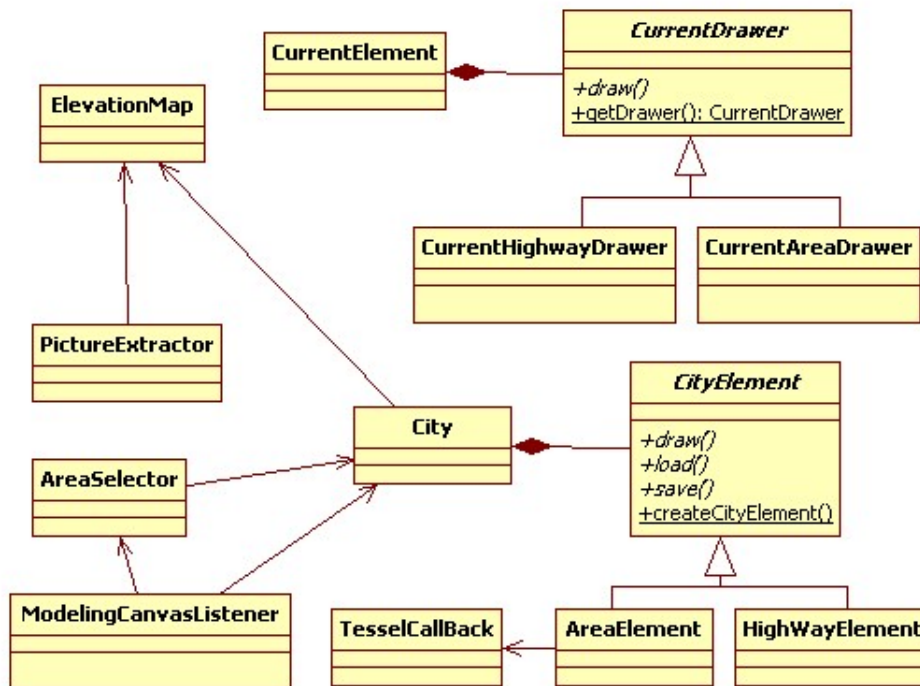


Figure 7: Modeling Class Diagram

As can be seen from the diagram the class *City* is composed from *CityElement*

objects. The class *CityElement* is an abstract class and defines an interface that every city element must have; the city elements must be responsible for their own drawing, saving and loading. Currently there are two concrete classes that extend the *CityElement* class; namely the *AreaElement* and the *HighwayElement*. The good thing about them is that no class except the parent knows that they exist. All the city elements are created via a factory method in the *CityElement* class, which makes it extremely easy to add new city elements to the hierarchy. The *City* class only works with the general interface provided by the *CityElement*. As a code example I have provided the draw method in the *City* class, note that it does not have a clue what concrete elements are being drawn, it only knows of the abstract *CityElement* class^{4.3}.

```
public void draw(GL gl)
{
    for (int i = 0; i < this.elements.size(); i++) {
        CityElement elem = this.elements.elementAt(i);
        if (elem != null)
            elem.draw(gl);
    }
}
```

Figure 8: City.draw()

Another interesting thing in the class diagram is the *CurrentElement*. Whenever the user starts drawing a new element on the modeling canvas a current element is created to store the information provided by the user. The difficulty comes from the fact that the user can start drawing a residential area for example and then decide on the fly that actually he wants to draw a commercial area. The bad thing about it is that residential and commercial areas are drawn differently on the canvas. So to allow the user change the drawing object on the fly without having to create a new *CurrentElement* I decided to abstract away the drawing. Therefore every time the user changes the type of the modeled object only the drawer of the current element is changed. This idea is very close to the State Pattern[3]: "Allow an object to alter its behavior when its internal state changes. The object will appear to change its class". Note also that the *CurrentDrawer* is an abstract class, the concrete drawer objects are again created via the factory method *getDrawer()*. This way we are free to add new current drawers when new modeling elements are added.

4.4 The graphics.generation Package

The *graphics.generation* package consists of all the classes that perform the roads and buildings generation in our city. It is the middle ground between the modeling package and the visualization package. Seen from a package perspective the gen-

eration package just takes a *City* object from the modeling, generates the roads and the buildings and creates the *DrawingMap* object which is then used intensely in the visualization package. Class diagram of the *graphics.generation* can be seen on the figure4.4.

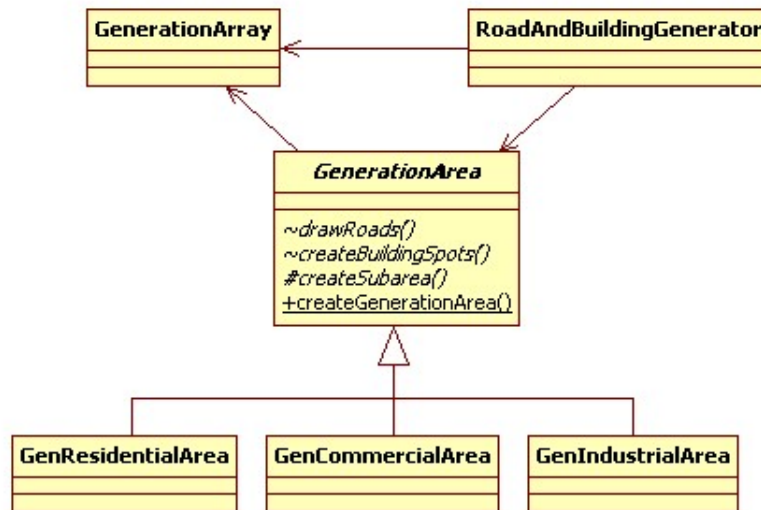


Figure 9: Generation Class Diagram

The main class in the diagram is the *GenerationArea*, it is an abstract class that defines the general interface required by all the generation areas. It was very important for the project to make the roads and buildings in each area unique. To achieve that I created a subclass of *GenerationArea* for each area type that we have. In this way I made each area responsible for generating its own roads and putting its own buildings. I also encapsulated these subclasses with a factory method in the *GenerationArea* so that no class except the parent knows that these subclasses exists. This approach provides for very easy and seamless integration of additional areas in the future. All that has to be done is create a subclass of the *GenerationArea*, implement the abstract methods for the new area and add the new class in the factory, no other changes in the existing code are required.

Another notable thing in the diagram is the protected abstract method *createSubarea()*. The problem there was that the general algorithm for extracting sub areas was the same for all types of areas. The only difference was that different areas created their subareas in a different way, different information needed to be passed to the constructors so the code could not be generalized. So I pulled up the implementation of the general algorithm in the *GenerationArea*, but I abstracted the creation of the sub area. So that every concrete area can use the general algorithm but can override the creation of its subareas to meet its needs. The approach is famous as the Template Design Pattern[3].

```

protected Vector<GenerationArea> extractSubAreas() {
    Vector<GenerationArea> retVal = new Vector<GenerationArea>();
    for (int i = this.getMinBound().x; i < this.getMaxBound().x; i++) {
        for (int j = this.getMinBound().y; j < this.getMaxBound().y; j++) {
            if (this.genArray.getValueAt(i, j) == this.index) {
                if (this.cityImage.isAreaType(i, j)) {
                    int[] pixel = this.cityImage.getPixel(i, j);
                    GenerationArea area = this.createSubarea(pixel);
                    area.setIndex(this.genArray.getNewAreaNumber());
                    area.setSize(0);
                    area.seedFill(i, j, this.index);
                    retVal.add(area);
                }
            }
        }
    }
    return retVal;
}

```

Figure 10: Extract Areas template method

The code snippet is from the class *GenerationArea*, it provides the general algorithm for extracting subareas, but gives the concrete classes the freedom to change how their sub areas are created(the highlighted line).

The other main class in this package is the *RoadAndBuildingGenerator*. It is the interface class of this package and is the only class used outside of the package. It provides some initializations on the *GenerationArray*, the *GenerationAreas* and the *DrawingMap*, before giving away control to the concrete area classes and their roads and buildings generation.

The class *GenerationArray* was added a little bit later in the system. We realized that we have to read and write pixels to the *CityImage* too often during generation which introduced very big performance overhead; that's why we decided to create the class *GenerationArray*. Its purpose was to completely replace the *CityImage* during the generation phase to reduce the overhead provided by raster manipulations. *GenerationArray* is internally implemented by a two dimensional array of integer, every type of object is defined by a constant in that array, for example the water is marked by -3. Of course these implementational details are invisible to the outside users of the *GenerationArray*. They can communicate with the class only by its public interface which completely encapsulates the internal structure, examples are the methods *markWater(int i, int j)* and *isWater(int i, int j)*. The client classes have no idea that the *GenerationArray* is internally a 2D array and water is marked with -3 in it.

4.5 The graphics.vizualization Package

The *graphics.visualization* package incorporates all the logic required for the final visualization of the city. It is pretty much self contained since it does not depend on any other packages. In this package we can find the *VisCanvasListener* class which implements all the visualization and user interaction with the displayed city. The most interesting class in this package is the *DrawingMap*. The underlying implementation of this class is a two dimensional array of type *MapElement*. The array is filled in the generation phase via a interface that encapsulates the *DrawingMap* internals very well:

- public void addRoad(Vector<Point2D> roadToBeDrawn, double width)
- public void addBuilding(Vector<Point2D> buildingLot, ModelType type)
- public void addWater(int i, int j)

The generation classes have no idea about the internal structure, they just point out what should be drawn and where. The array in the *DrawingMap* is filled with subclasses of the abstract class *MapElement*. This hierarchy was introduced to generalize the drawing of the *DrawingMap* into the screen so that the *VisCanvasListener* can draw without knowing the exact objects that are drawn. In this case I also distributed responsibility, the *DrawingMap* subclasses are responsible for loading their textures and drawing themselves on the screen. This technique provides for high extensibility; new drawing elements can be added seamlessly. How exactly the whole process works can be seen on the sequence diagram4.5.

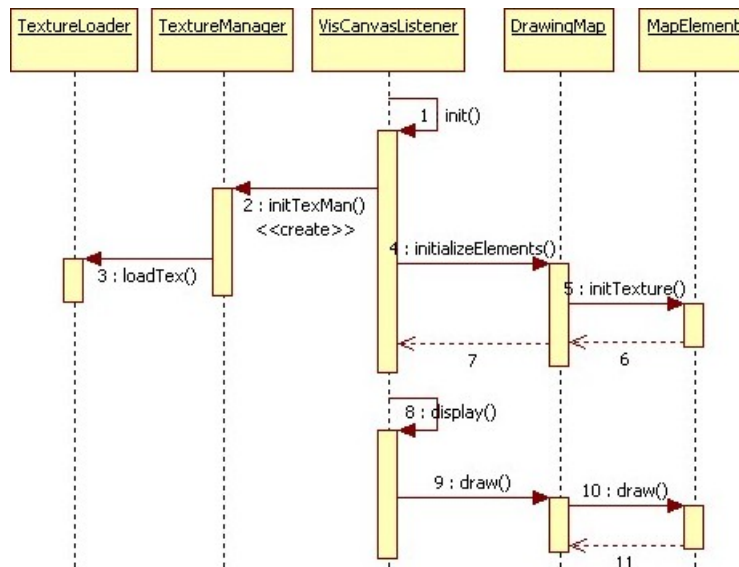


Figure 11: Visualization Sequence Diagram

First the *VisCanvasListener* is initialized. It then tells the *TextureManager* to load the textures required to visualize the city. After that the *DrawingMap* is informed that the textures are loaded and that the *MapElements* can initialize their texture handles. The next step is that the canvas is actually displayed. It calculates its view box and then tells the *DrawingMap* to draw every element within certain bounds. The diagram also shows the *TetureManager* and *TextureLoader* classes. They are responsible for loading the texture files and then handling the texture IDs. All the classes and the relationships between them can be seen in the class diagram below4.5.

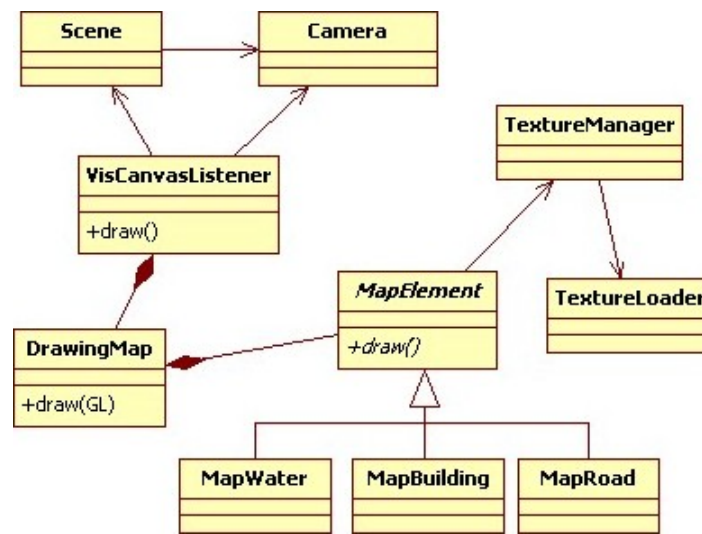


Figure 12: Visualization Class Diagram

The *Scene* and *Camera* classes are used exclusively by the *VisCanvasListener*. The *Camera* class encapsulates all the user interaction with the displayed city, this way it is the only place we have to change if we want to somehow modify the user experience. It is also possible to abstract the class's interface and then extend concrete camera classes to provide different camera modes. The *Scene* on the other hand is responsible for the rendering of the 3D city model. It is the class that sets up the environment of the city, handles the mini map and provides LoD functionality.

4.6 The utils Package

The *utils* package provides different utility functions for the system. It is a so called "global" package, since it provides functionality for all other packages. Such packages can usually endanger the future evolution of the project, but taking into account that the *utils*'s interface is highly unlikely to change this does not pose a

threat to the extensibility of the system.

Since the classes in this package are not related I will not provide a class diagram but will enumerate the classes providing a brief explanation for the most important ones.

- *ConfigurationManager* : except for providing some transformations functionality for the *ModelType* users, its main value comes from the fact that it serves as a mediator between the canvases and the GUI. It implements the Singleton Pattern[3] and it holds the whole configuration information that is set in the GUI(for example the toolbar button that is currently pressed). The approach decouples the display logic in the canvases from the graphical user interface, and makes us free to alter the user experience in whichever way we want.
- *FileUtilities* : provides some file utility methods.
- *GeometryUtils* : provides geometry functionality like computing intersections, computing orthogonal vectors, etc.
- *ModelType* : this is not a class it is an enum type and it holds all the modeling types that incorporate the city. It should be the first point of change when adding new modeling elements to the city.
- *DrawingPoint* : mostly a container class, provides a few method for transformations and measuring distance. Used mainly in the modeling package and a little bit in the generation.

5 Statistics and Tool Support

We have used Eclipse[5] as an integrated development environment. It's IntelliSense and refactoring capabilities proved once again to be very helpful and saved us a lot of time.

Since more than one developer is working on the project we needed a version control tool to keep the code synchronized. We choose Subversion[6] and its Eclipse plug-in which integrated with our IDE seamlessly. In the final days we had more than 270 commits to the SVN server.

During the final days of development, I decided to use a profiler in order to find where are the bottlenecks in the program so that we can optimize them. I tried some open source profilers which didn't work, so finally I used a commercial product: YourKit[7]. It might be very expensive, but it's definitely worth its money, everything in it just worked out of the box, it even integrated automatically with Eclipse. The first time we ran the profiler, it showed many interesting things⁵.

It showed that too much time is going in calling `Stack.push()` and `Stack.pop()`, which was normal from one side since we were using a lot of recursive-like meth-

Name	Time (ms)	%
javafx.media.opengl.glu.GLU.gluBuild2DMipmaps(int, int, int, int, int, int, Buffer)	10 134	27%
graphics.generation.GenerationArea.seedFill(int, int, int)	8 452	22%
javafx.imageio.ImageIO.read(File)	5 659	15%
java.util.Stack.pop()	4 816	13%
graphics.generation.GenerationArea.extractIntersectionPoints()	2 353	6%
java.util.Stack.push(Object)	1 942	5%
java.awt.image.Raster.getPixel(int, int, int[])	1 001	3%
java.lang.ClassLoader.loadClassInternal(String)	911	2%
java.lang.Thread.run()	901	2%
utils.GeometryUtils.computeOrthogonalSlope(Point2D, Point2D)	761	2%
javafx.swing.UIManager.<clinit>()	570	1%

Figure 13: Profiler

ods. Still the performance penalty was too big and needed to be optimized. After we knew where to optimize, the rest was easy, we moved the variable validity checks before the stack push and the results were very good. We were able to greatly improve the time spent on push and pop. After the optimizations the system spent around 80% less time in pop and around 70% less time in push5.

Name	New Time (ms)	Old Time (ms)	Time Diff (ms)
java.util.Stack.pop()	961	4 816	-3 855
java.util.Stack.push(Object)	590	1 942	-1 351

Figure 14: After Optimizations

And in the end I will provide some statistics[8] about the actual code that we have developed for the system5.

Filename:	Source:	Comment:	Both:	Blank: /	Total:
C:\Documents and Settings\Nikolay Kazmin\M...	219	21	0	41	281
C:\Documents and Settings\Nikolay Kazmin\M...	187	2	2	47	238
C:\Documents and Settings\Nikolay Kazmin\M...	329	44	0	51	424
C:\Documents and Settings\Nikolay Kazmin\M...	230	20	0	61	311
C:\Documents and Settings\Nikolay Kazmin\M...	291	5	0	78	374
C:\Documents and Settings\Nikolay Kazmin\M...	396	8	1	81	486
C:\Documents and Settings\Nikolay Kazmin\M...	345	36	10	93	484
[Totals] (# of files = 49):	4666	401	24	1173	6484
Total Files Processed:	45				
Percentage of Comments:	6.6% (425/6484)				

Figure 15: Code Statistics

6 Conclusion and Future Work

In my opinion we were able to meet the initial requirements. We managed to build a working version of the product that provides the basic functionality. There are still many bells and whistles that are not implemented, but we have provided many extension points where this can be done. The different software components that build up the system are very loosely coupled which provides for easier evolution. The design is simple where possible and non-trivial where necessary to allow for extensions.

We leave the code base well documented, javadoc can be found near almost every non trivial method, while the higher level design decisions are explained in the final report.

The performance of the system is also very much acceptable, the city can be rendered in real time even on older computers.

The future development of the system will mostly be focused on making the city more attractive. Currently the system allocates the building lots so what's left to be done is actually create a model manager and add many different 3d models into the system. Of course it will be great if some animations can be added, like cars, people to make the city more "alive".

References

- [1] Frederick P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition)* .
- [2] Kent Beck, *Extreme Programming Explained: Embrace Change*.
- [3] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*
- [4] Martin Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)*
- [5] <http://www.eclipse.org/>
- [6] <http://subversion.tigris.org/>
- [7] <http://www.yourkit.com/>
- [8] <http://www.geronesoftware.com/>
- [9] <http://www.jdom.org/>