Research Thesis T3100:

Advancing a Class Library to an Open-Source Project

Course of Studies: Angewandte Informatik

Duale Hochschule Baden-Württemberg Stuttgart/Campus Horb

by

Marco Schäfer

17.01.11

Edit Time	21.09.10 – 17.01.11
Student - ID, Course	8161805, TAI2008
Company	M&M Software GmbH, St.Georgen
Supervisor of DHBW	Prof. Dr.-Ing. Olaf Herden

Declaration of Academic Integrity

Hereby I declare that this thesis *Advancing a class library to an open-source project* has been written only by me. Furthermore, I confirm that no sources have been used in the preparation of this project other than those indicated in the project itself.

Monday, January 17, 2011 Lauterbach

Acknowledgement

I would like to thank *Prof. Dr.-Ing. Olaf Herden* from *DHBW Stuttgart/Campus Horb* for excellent mentoring and giving feedback during the whole edit time. Furthermore I thank my sister for proof-reading this document.

Abstract

This thesis *Advancing a class library to an open-source project* was developed by *Marco Schäfer*, student at *DHBW Stuttgart/Campus Horb*.
The author firstly introduces to the project by listing assumptions to the reader and the structure of this work.
Thereafter he describes the situation before the developed solution existed, the resultant problem and the goals of this thesis.
Chapter 3 analyses the existing visualization framework by giving an overview of the context and describing the packages of the framework in more detail.
Furthermore the requirements and expected behavior to the visualization framework are listed in chapter 4.
In Chapter 5 the architecture and design of the developed components is presented.
Finally achieved results, future tasks and the author's personal conclusion are pointed out.

Contents

1 Introduction

This chapter introduces the reader to the project by lining up some basic assumptions and giving an overview about the structure of the document.

1.1 Reader Assumptions

To understand the content of this project, it is beneficial for the reader to have some knowledge in the used technologies. It is not necessary that the reader knows all of them in detail but basic elements of the following technologies should be familiar.

- **Java Technology**
  Java is an up-to-date technology developed by the company *Sun Microsystems* to program platform independent software applications. It includes the object oriented programming language *Java* to write software, the *Java Development Kit* (JDK) which contains class libraries and a compiler and the *Java Runtime Environment* (JRE) to execute the programs.

- **Unified Modeling Language (UML)**
  The *Unified Modeling Language* is used for the diagrams describing the system. Equivalent to the Java Technology in implementation layer, UML is the foundation for system modeling. Readers should be familiar with most common UML-diagrams like Class-, Use-Case-, Sequence- or Deployment-models.

1.2 Structure of Document

- Chapter 2 gives a short overview of the project. It points out the current situation, the consequential problem and describes the expected goals.
- Chapter 3 analyzes the design and functionality of the existing visualization framework.
- Chapter 4 lists the requirements and expected behavior to the framework
- Chapter 5 covers the worked out architecture by giving an overview of the context and giving an overview to the *visualizationElements* package and describing its components in more detail.
- Chapter 6 completes the project by summing up achieved results, a short outlook about uncovered tasks and future extensions and finishes with a personal conclusion.

2 Project Overview

This chapter describes the current situation, points out the problem to be solved and lists the goals of this project.

2.1 Current Situation

To teach the function of several algorithms and data types, university lecturers at *DHBW Stuttgart/Campus Horb* use a framework to visualize the functions step by step. This framework was developed by *Björn Strobel* in year 2006 [1] and was continued and improved by *Fabian Hamm* 2010 [2].
The framework is used to show the function of abstract data types like queues stacks and lists, visualization of sort algorithms, operations on graphs and the function of recursion, backtracking and hashing.

2.2 Problem Description

The main problem of the framework is that if a lecturer wants to visualize the function of algorithms it won't be enough to just implement the algorithm and log the successive actions. He also has to implement the graphical elements to visualize the steps within the framework.
This leads into a high effort because he has to find and implement a realistic graphical visualization for the algorithm or data structure. Furthermore the Java class library has only few classes for graphical objects so it's hard to draw in source code.
Additionally it's hard to arrange the implemented components practically because the elements have to be placed by X- and Y-coordinates in the application window.

2.3 Goals of Thesis

The goal of the thesis is to develop several graphical elements to make step-by-step visualization of algorithms easier for the developer.
First of all the two antecessor theses have to be analyzed to get to know the existing framework.
Next step is to figure out the most qualified graphical elements to visualize algorithms and abstract data types.
These elements have to be designed and implemented with Java technology in this thesis.
If there's enough time functionality will be assured by Unit Tests.

3 Analysis of the Visualization Framework

This chapter introduces to the existing visualization framework developed in previous theses. It explains its design and functionality of the framework's components.

3.1 Overview

The existing framework to visualize algorithms step-by-step consists of two packages:

- **visualization**

  The *visualization* package contains abstract classes that are used to show steps of the algorithm graphically and textual. This package is described in more detail in chapter 3.2.

- **logging**

  The *logging* package provides abstract classes with functionality to log each step of an algorithm. This package is described in more detail in chapter 3.3.

To put the step-by-step visualization into practice a third package has to be linked as shown below in Figure 1: Visualization Framework Overview. This package must contain classes that extend the abstract classes existing in these two packages and adjust the functionality especially for its algorithm.

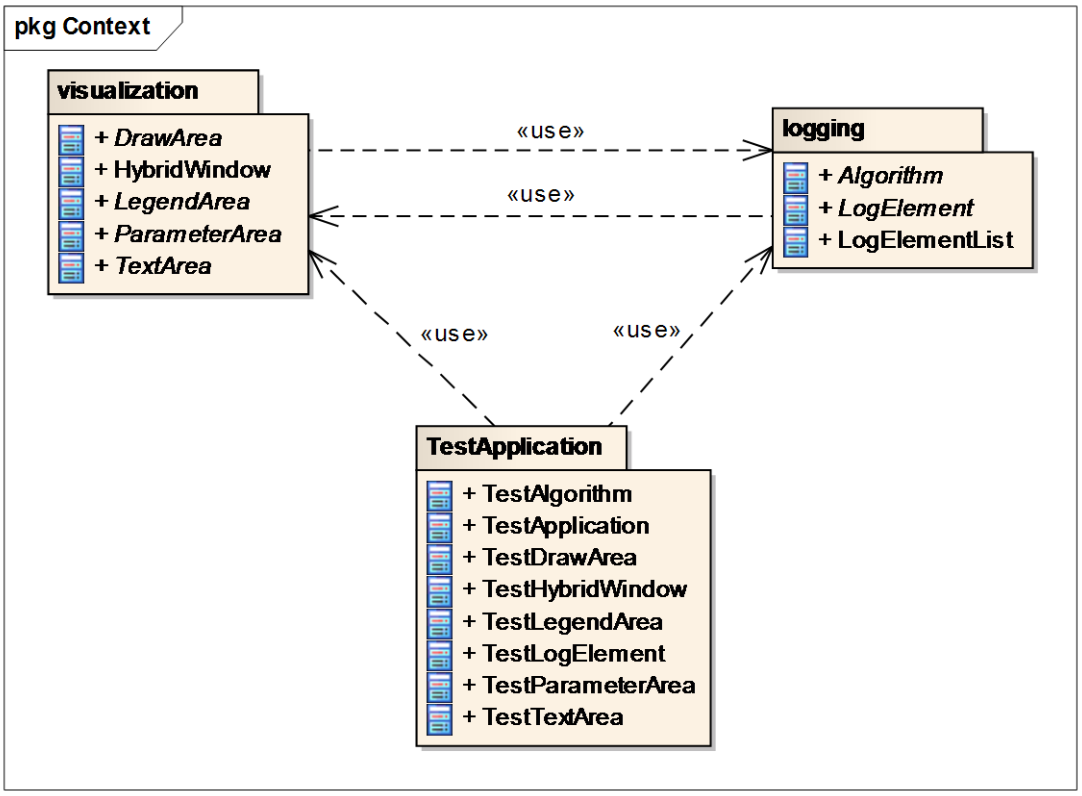

Figure 1: Visualization Framework Overview

3.2 Visualization Package

The visualization package contains the following abstract classes to visualize the function of an algorithm:

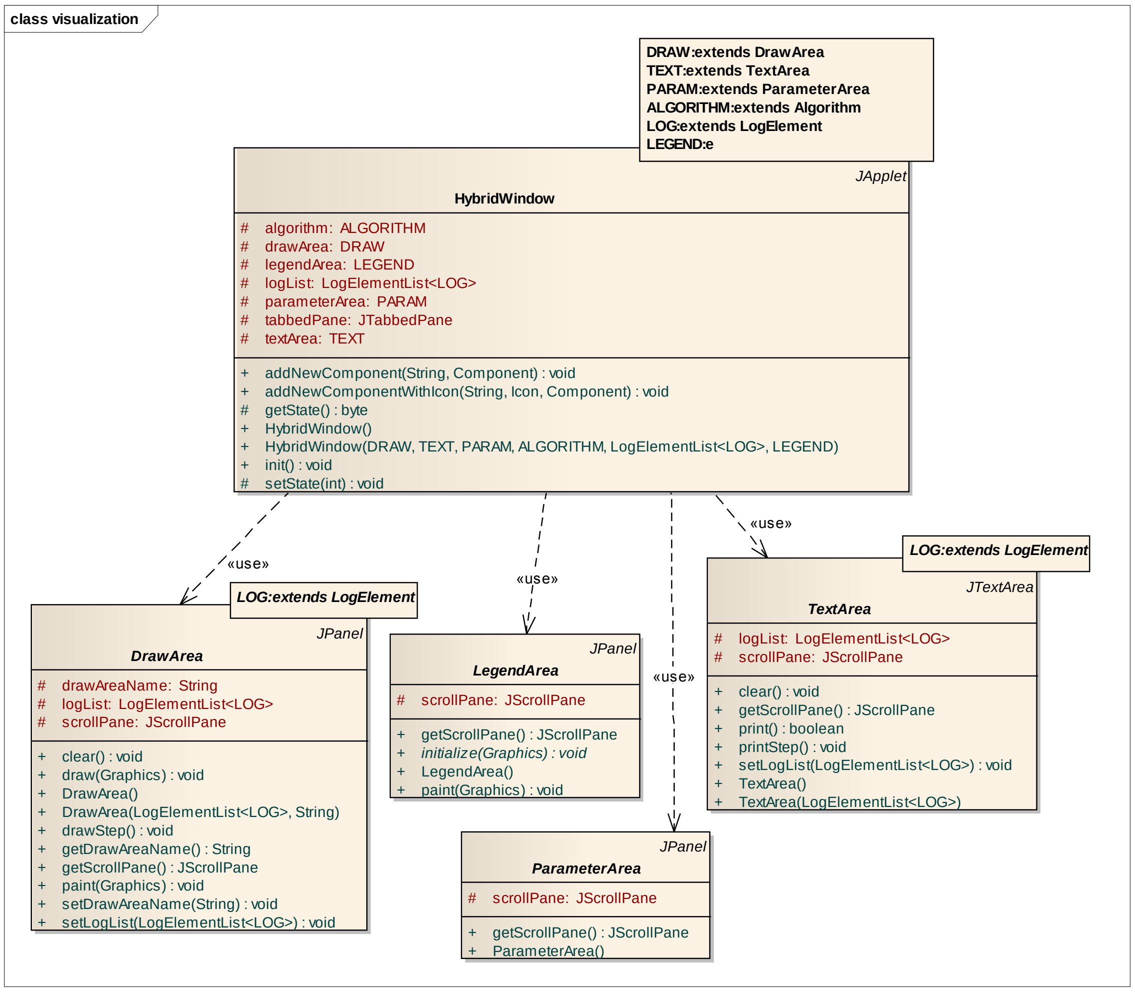
class visualization
DRAW:extends DrawArea
TEXT:extends TextArea
PARAM:extends ParameterArea
ALGORITHM:extends Algorithm
LOG:extends LogElement
LEGEND:e
JApplet
HybridWindow
# algorithm: ALGORITHM
# drawArea: DRAW
# legendArea: LEGEND
# logList: LogElementList<LOG>
# parameterArea: PARAM
# tabbedPane: JTabbedPane
# textArea: TEXT
+ addNewComponent(String, Component) : void
+ addNewComponentWithIcon(String, Icon, Component) : void
# getState() : byte
+ HybridWindow()
+ HybridWindow(DRAW, TEXT, PARAM, ALGORITHM, LogElementList<LOG>, LEGEND)
+ init() : void
# setState(int) : void
«use»
«use»
«use»
«use»
LOG:extends LogElement
JPanel
DrawArea
# drawAreaName: String
# logList: LogElementList<LOG>
# scrollPane: JScrollPane
+ clear() : void
+ draw(Graphics) : void
+ DrawArea()
+ DrawArea(LogElementList<LOG>, String)
+ drawStep() : void
+ getDrawAreaName() : String
+ getScrollPane() : JScrollPane
+ paint(Graphics) : void
+ setDrawAreaName(String) : void
+ setLogList(LogElementList<LOG>) : void
JPanel
LegendArea
# scrollPane: JScrollPane
+ getScrollPane() : JScrollPane
+ initialize(Graphics) : void
+ LegendArea()
+ paint(Graphics) : void
LOG:extends LogElement
JTextArea
TextArea
# logList: LogElementList<LOG>
# scrollPane: JScrollPane
+ clear() : void
+ getScrollPane() : JScrollPane
+ print() : boolean
+ printStep() : void
+ setLogList(LogElementList<LOG>) : void
+ TextArea()
+ TextArea(LogElementList<LOG>)
JPanel
ParameterArea
# scrollPane: JScrollPane
+ getScrollPane() : JScrollPane
+ ParameterArea()

Figure 2: Visualization Package

- **ParameterArea (1)**

  ParameterArea is an abstract class to set parameters for an algorithm. It extends *JPanel* class and is located on the top left of the *HybridWindow*. To make possible parameterization on the algorithm this class has to be extended by the application and some input elements (*TextBox*, *CheckBox*, *RadioButton*, etc.) have to be added.

- **LegendArea (2)**

  LegendArea is an abstract class to show a legend in the *HybridWindow* to describe the visualized elements. This leads into easier understanding of the visualized algorithm. It extends *JPanel* and is located on the bottom left of the *HybridWindow*. To show a legend in the application the class has to be extended by the application and with help of the *initialize* method the elements to show as legend have to be defined.

- **TextArea (3)**

  TextArea is an abstract class to visualize the steps of an algorithm textually. It extends the Java *JPanel* class and is located on the bottom of the *HybridWindow*. This class has to be extended and adjusted by the application. To show the steps textual the *print* method has to be overwritten.

- **DrawArea (4)**

  The abstract class DrawArea is the main part of the visualization of algorithms. It is the biggest part of the HybridWindow and is located in the middle. It extends the JPanel class, too. The DrawArea class has to be extended by the application and the draw method has to be overwritten to show the steps of an algorithm graphically.

- **HybridWindow**

  The class *HybridWindow* extends *JApplet* and is the container for *ParameterArea* (1), LegendArea (2) *TextArea* (3), and *DrawArea* (4). Additionally it contains elements to navigate through the steps of an algorithm (Next Step, Last Step etc.) (5). To use the *HybridWindow* as an applet this class has to be extended by the application and the constructor has to be overwritten. If it is used as an application it is not necessary to overwrite the constructor.

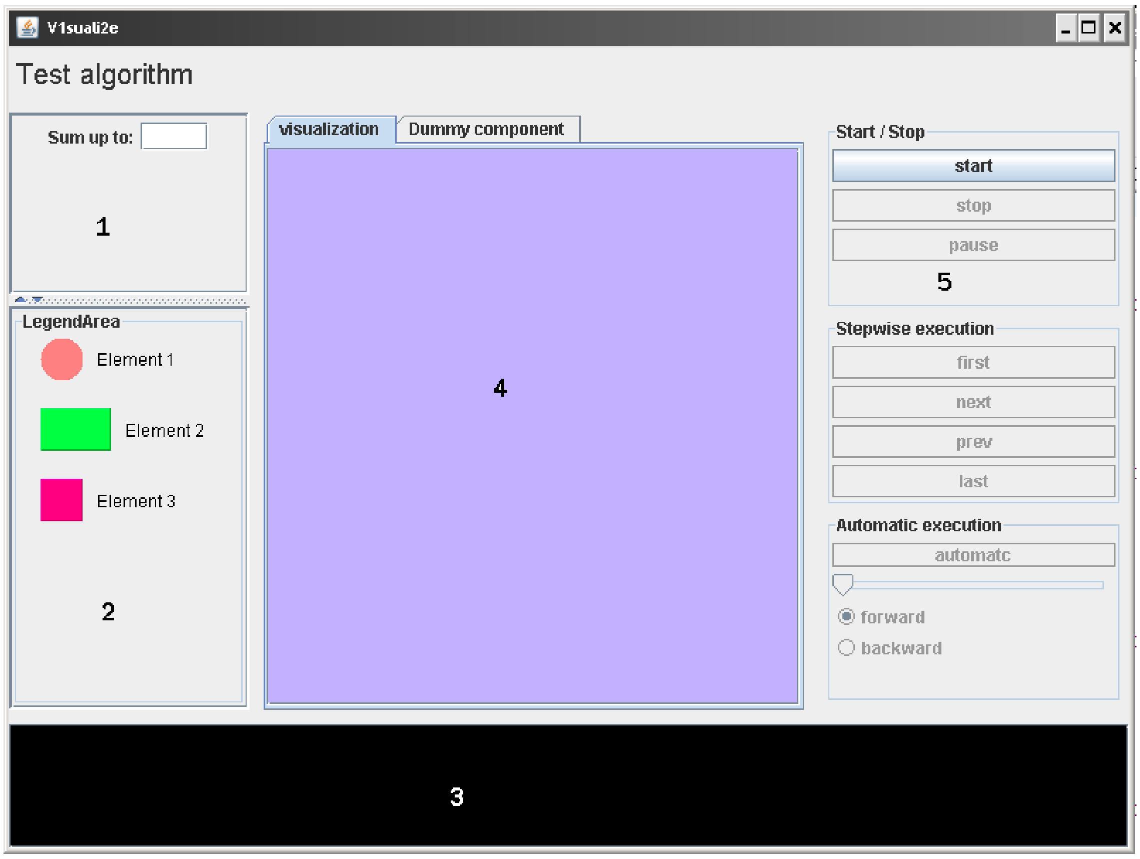

Figure 3: HybridWindow

3.3 Logging Package

The *logging* package contains the following abstract classes to log each step of an algorithm:

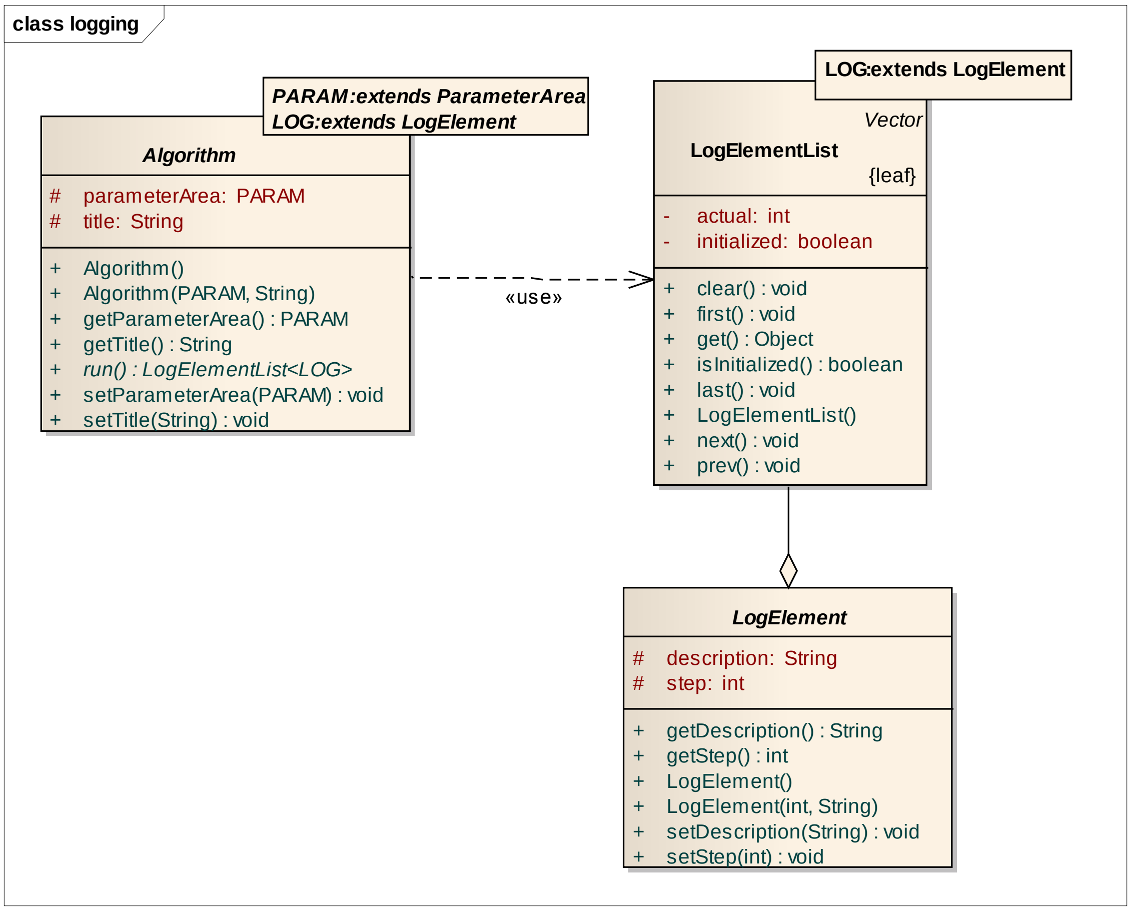

Figure 4: Logging Package

- **LogElement**

  The abstract class *LogElement* represents a step of an algorithm. The application has to extend this class by adding attributes required to save steps of a specific algorithm.

- **LogElementList**

  LogElementList is a final class which represents a container of *LogElements*. It stores the several *LogElements* of an algorithm and has some methods to easily navigate through the list.

- **Algorithm**

  The abstract class Algorithm represents the algorithm that is visualized with the framework. It has to be extended by the application and the *run* method must be implemented. In the *run* method the algorithm has to be implemented and the significant steps have to be logged by adding *LogElements* objects to a *LogElementList* instance. After the algorithm was finished this *LogElementList* instance is returned to visualize the steps with help of the visualization package (see chapter 3.2).

4 Requirements

Regarding to the analysis of the existing framework some requirements evolved.

4.1 Required Visualization Elements

In consideration of the fact that the visualization framework is used in computer science lectures, the elements it has to provide become clear.
First of all a table is needed to give an overview about values of variables or states of algorithms. But especially to visualize hashtables a table with two columns is needed.
The framework is also used to clarify the function of abstract data types queue and stack. To visualize a queue the elements are arranged in a row. So a table, with one row and columns as much as elements in the queue, is the most qualified to visualize a queue.
In contrast, the elements of a stack are arranged on top of each other. So a table with one column and rows as much as elements in the queue is the most practicable to visualize a stack.
A list is also a container of elements and so the best way to visualize it is similar to the way of visualizing a queue or stack. Therefor it was decided to draw lists as table with one row and columns as much as elements in the list.
Additionally the framework is used to visualize step-by-step the function of different sort algorithms like *Quicksort*, *Bubblesort*, *Mergesort*, *Heapsort* and several more. A bar chart is needed in the framework because is the standard graphic to visualize sort algorithms. But in some cases it is better to visualize the elements to sort not as bars but as dots. So a dot chart is also required in the visualization framework.
Furthermore another part of computer science is theory of graphs and algorithms on graphs like depth-/breadth-first search, prim's algorithm and a lot of more. To alleviate step-by step visualization of these algorithms, drawing a graph with appertaining edges and vertexes must be provided in the extended framework. Thereby it should be possible to differ between directed and undirected graphs as known in graph theory.
Additionally the framework is often used to teach and clarify the functionality of recursion and backtracking on the basis of the *N-Queens-Problem* or the *Knight's Tour*. Therefore it should be possible to picture a chess board.
Finally another usage of the framework is pathfinding with backtracking or graphs. So the visualizing framework should automatically generate and draw a

maze so that the user doesn’t have to be introduced in the complicated algorithms of maze generation.

There are some more graphical elements which clarify the usage of algorithms such as *The Towers of Hanoi* which are disregarded within this thesis because they have only one specific algorithm and are not used in more use cases as the elements named above.

4.2 Expected Behavior

The framework was developed to show the function of algorithms step-by-step. So for each step of the algorithm the visualization element has to be updated and redrawn. But the developer should not be forced to create a new visualization element for each step because this would need a lot of memory. Additionally in case of the maze, which is generated randomly, a different maze for each step would be created.
Furthermore the visualization elements should be adapted to the size of the *DrawArea* automatically because the *HybridWindow* can be resized by the user. But at the same time it should not become too small or even get hidden because of the window borders.

5 Architecture and Design

This chapter describes the architecture of the developed software. It starts by giving an overview over the developed system and describes the architecture of the new developed package. Finally it describes the several new developed components in detail.

5.1 Visualization Framework Overview

Figure 5 shows how the new package *visualizationElements* was integrated to the existing context.

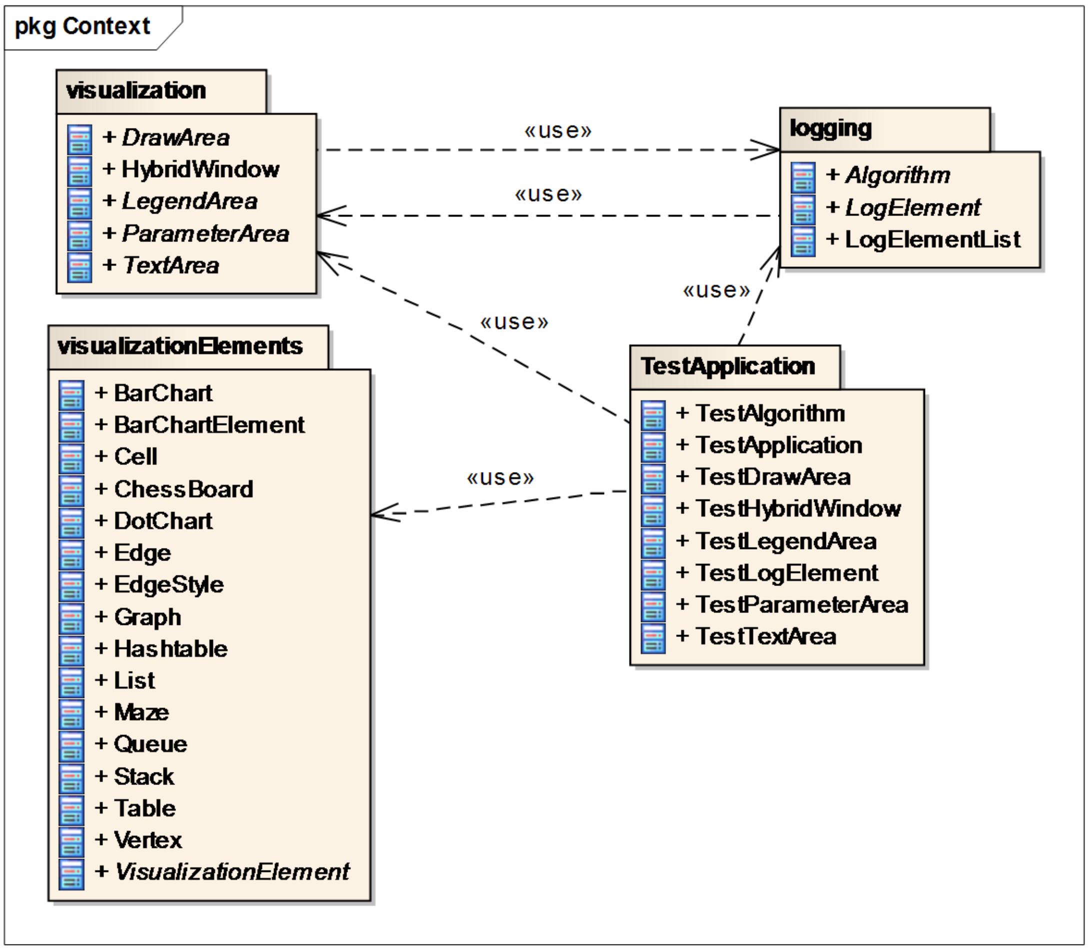

Figure 5: Extended Vizualization Framework Overview

The application still uses functionality of the logging package to log each step of the algorithm and visualizes the logged steps with help of the visualization package.
The developer can now use elements of the visualizationElements package within the application to visualize the steps easier than before. He does not have to create graphics by himself.
Using the provided functionality is quite easy. The only thing a developer has to do is to instantiate the chosen visualization element within the extended *DrawArea* of the application and to call the element's *draw* method within the *DrawArea*'s *draw* method.

5.2 VisualizationElements Overview

Figure 6 shows the architectural structure of package visualizationElements.

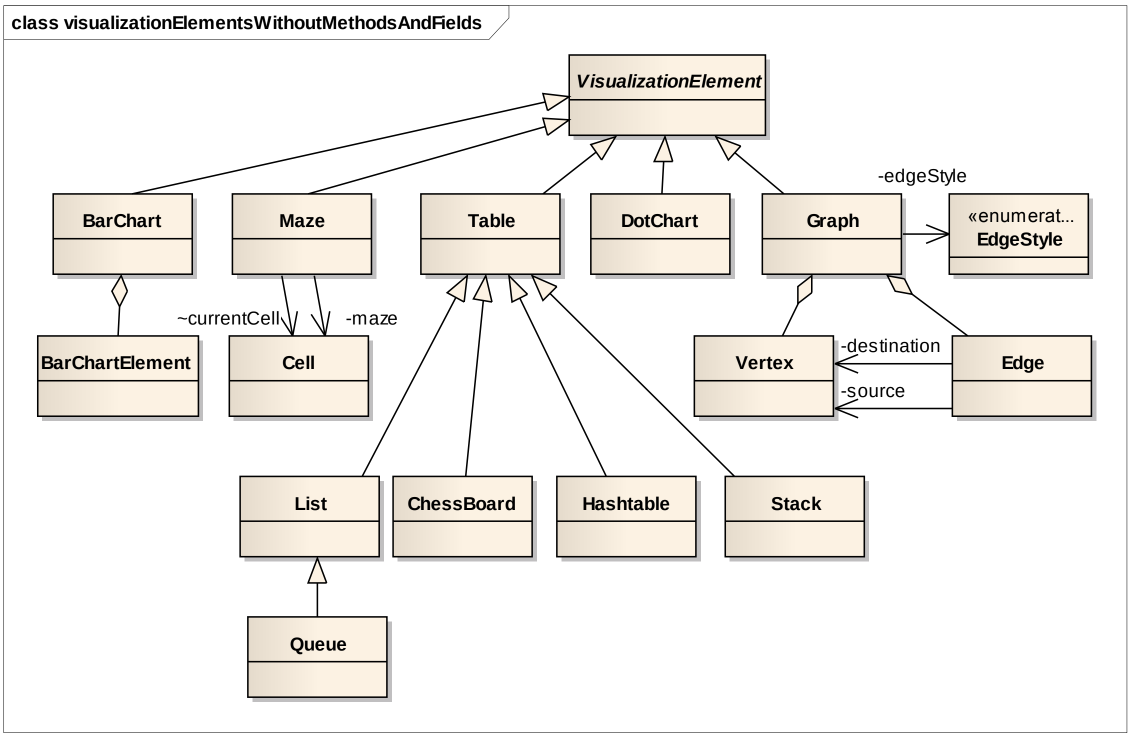

Figure 6: VisualizationElements Package

The package has a hierarchal structure. Each class in package *visualizationElement* which represents a visualization element extends the class *VisualizationElement*. Furthermore classes which are consequentially similar and differ only in little functionality and design, build up an inheritance, too.
Additionally elements which consist of multiple other elements build up an aggregation.

5.3 VisualizationElements Components

This chapter describes each component of the *visualizatonElements* package in detail:

- **VisualizationElement**
  The abstract class *VisualizationElement* is the base class for all visualization elements. It provides the *draw* method which every element needs to draw itself.

- **Table**
  To visualize a table the *Table* class has to be instantiated and the values have to be passed into the constructor as two-dimensional array. The constructor is overloaded so that the column captions can be passed in as one-dimensional string array optional. The *draw* method draws a grid and highlights the column captions.

- **Hasthtable**
  The *Hashtable* class is just a table with two columns, one for the hash value and one for the corresponding object. The captions of the *Table* is set to "*Hash value*" and "*Object*" automatically.

- **Stack**
  The *Stack* class is implemented as extension of *Table* inheritance with one column. Its constructor is overloaded. The values can be inserted either as *Vector* or as one-dimensional array of any type. Internally a table with one column is created within the constructor. To point out that the represented graphic element is a last-in-first-out memory (LIFO) a bolster is drawn around the values with an open border at the top. That highlights that only the last element that was pushed on the stack can be popped out.

- **List**
  The class *List* is implemented as *Table* extension with one row. Its constructor is overloaded. The values can be inserted either as *Vector* or as one-dimensional array of any type. Within the constructor a *Table* inheritance with one row is created internally.

- **Queue**
  The class *Queue* is implemented as implementation inheritance of *List*. Its constructor is overloaded, too. The values can be inserted either as *Vector* or as one-dimensional array of any type. The *Queue* is drawn in the same way as a *Table* instance. But it has a bolster around with open borders on the left and right. This clarifies that elements which are inserted first must be removed first and newly inserted elements are appended at the end as a first-in-first-out memory works (FIFO).

- **BarChart**
  The *BarChart* class either takes a *Vector* or an array of *BarChartElements* that represnent the values of the *BarChart*. Additionally it takes the height and width of the *DrawArea* to adjust the size of the *BarChart*.
  The *draw* method arranges successively *BarChartElements*. It adjusts its size to the size of the window automatically.

- **BarChartElement**
  Elements to present within a *BarChart* are implemented in the *BarChartElement* class. It gets passed the numeric value and optionally the *Color* into its overloaded constructor. If no *Color* instance was passed into the constructor the black is set as default value for the color property. It is drawn as a rectangle with the previously set color. Its height depends on the value it represents. And the width depends on the number of values the *BarChart* visualizes and the window size.

- **DotChart**
  The class *DotChart* has a similar function than the *BarChart* class. It takes a *Vector* or one-dimensional array of integers and the height and width of the *DrawArea* in its constructor. Dependent on the height and the integer values each *Vector* element is drawn as little dot. The space between each dot depends on the width of the *DrawArea* and the number of elements to draw.

- **Graph**
  The *Graph* class consists of a *Vector* of *Vertex* instances and a *Vector* of *Edge* instances which are also passed into the constructor as a flag whether the Graph instance represents a directed or undirected graph and an *EdgeStyle* value to decide in which style the edges are drawn. The draw method draws each *Vertex* and each *Edge* dependent on the “directed”-flag either as normal line or as arrow.

- **Vertex**
  Vertexes of a *Graph* are represented as instances of the *Vertex* class. They are located within the *DrawArea* by the X- and Y- coordinates which are passed into their constructor. Additionally a marking and the color of the vertex are set on construction of the *Vertex*. A *Vertex* is drawn as filled circle in the set *Color* and named by the marking.

- **Edge**
  The *Edge* class represents a connection between two instances of type *Vertex*. Therefore it needs two vertexes, one source vertex and one destination vertex. As the *Vertex, Edge* it gets a marking and *Color* injected, too. If the *Graph* is directed they are drawn as arrows otherwise as normal lines. Additionally they are drawn depending on the *EdgeStyle* angled or direct.

- **EdgeStyle**
  The *EdgeStyle* enum consists of the two values *Directed* and *Angeled*. Depending on the *EdgeStyle* an *Edge* is drawn from source to destination directly or with connected horizontal and vertical lines (Angeled).

- **Chessboard**
  The *Chessboard* is a kind of *Table* where the cells of the table are filled with color and there are as much rows as tables which are named with letters and numbers as known from chessboards. A two-dimensional array of type *Boolean* is passed into its constructor which represents the chessboard's visited cells. The visited cells are shown as letter within the cell. The board is sized depending on the *DrawArea*'s size automatically.

- **Maze**
  The *Maze* class creates and visualizes a maze with the passed in height and width. Internally it is organized as two-dimensional array of type *Cell*.

- **Cell**
  A *Cell* represents a cell of a maze. It is located by the number of the row and column within the maze. Four *Boolean* values represent the walls in each direction. The walls of a cell can be set or "removed" either by getters and setters or by special remove wall methods called.

Some of the developed visualization elements are redundant because they could also be represented by other elements. For example *List* and *Queue* are nearly the same but it was decided to implement both. The reason for this is that these elements are often required and it’s easier for the developer to choose the required element.

6 Results

This chapter shows the results and open aspects of the thesis. Finally the author describes his personal conclusion about this project.

6.1 Achieved Results

The developed system fulfills the main expectations according to the elaborated requirements and behavior.
All of the required elements which are worked out in chapter 4.1 are realized as Java classes.
With help of the *visualizationElements* package a developer does not have to create own elements to visualize the algorithms. This makes developing much easier because drawing with Java was very exhausting. Not only to visualize the values adequately but also to locate them clearly arranged within the *DrawArea* is much easier now.

6.2 Open Aspects / Future Extensions

There are still some open aspects to do in second part of the thesis.
First the whole visualization framework should be tested with unit tests and system tests.
Afterwards it should be published as open source project on a qualified portal.
Additionally the projects popularity must be improved to bring new developers on board which will continuing developing on the visualization framework.

6.3 Personal Conclusion

From my personal point of view, the achieved results of the project are satisfying, but not perfect.
The main goals were fulfilled. All functionalities based on the requirements analysis and are working quite well under the expected behavior.
The architecture of the system has a clear design and is easy to extend so that changes and additional functionality can easy be implemented.
In the company I am working I do not use Java technology. So it was hard for me to reach the achieved results and sometimes implementing was dragging. But during this thesis I improved my *Java* knowledge very much.
The project was heavily focused on practical work and not so much research was done. This real world project helped me a lot in developing my software engineering skills. I am confident that this was a really good practice for me as software developer.

A Appendix

A.1 Glossary

Hashtable	In a Hashtable stores the memory addresses of objects that are calculated by a hash function.
List	List is a collection of objects. The objects are referenced by pointers. So objects can be accessed randomly.
Queue	Queue is a collection of objects. The objects can only be read in the same order as they were stored.
Stack	Stack is a collection of objects where the objects can only be read the other way round as they were stored.

A.2 References

Literature:

[1] Björn Strobel: Java Bibliothek zur schrittweisen Ausführung von Algorithmen, BA Horb, 2006

[2] Fabian Hamm : Weiterentwicklung einer Klassenbibliothek zum Open Source Projekt, DHBW Stuttgart / Campus Horb, 2010

[3] Till Rathmann: Prozedurale Labyrinth-Generierung, Universität Erlangen-Nürnberg, 14.11.2005

Internet:

[4] Animering af algoritmer:
http://www.akira.ruc.dk/~keld/algoritmik_e99/Applets/
Date: 15.10.2010

[5] Everything you want to know about mazes but you are afraid to ask
http://www.ii.uni.wroc.pl/~wzychla/maze_en.html
Date: 15.10.2010

[6] Relevant Algorithm Animations/Visualizations (in Java)
http://www.ansatt.hig.no/frodeh/algmet/animate.html
Date: 17.10.2010

[7] Inside Technique : Backtrack Recursion and the Mysterious Maze : Maze Generation:
http://www.siteexperts.com/tips/functions/ts20/page3.asp
Date: 17.10.2010

[8] Think Labyrinth: Maze Algorithms:
http://www.astrolog.org/labyrnth/algrithm.htm
Date: 17.10.2010

A.3 Figures

A.4 Abbreviations

FIFO	First-In-First-Out
JDK	Java Development Kit
JRE	Java Runtime Environment
LIFO	Last-In-First-Out
UML	Unified Modeling Language