universität
wien

MASTERARBEIT

Titel der Masterarbeit

NETLUKE

Play with Algorithms - Part II

verfasst von

Alexander Rotheneder, BSc

angestrebter akademischer Grad

Diplom-Ingenieur (Dipl.-Ing)

Wien, 2015

Studienkennzahl lt. Studienblatt:	A 066 926
Studienrichtung lt. Studienbuchblatt:	Masterstudium Wirtschaftsinformatik
Betreuer:	Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta

Eidesstattliche Erklärung

Hiermit erkläre ich eidesstattlich, die vorliegende Arbeit selbstständig und ohne Benutzung anderer, als der angegebenen Quellen und Hilfsmittel verfasst zu haben. Die aus fremden Quellen übernommenen direkten oder indirekten Gedanken, Konzepte und Zitate sind als solche kenntlich gemacht.

Die Arbeit wurde bisher weder in gleicher oder ähnlicher Form verfasst, noch einer anderen Prüfungsbehörde vorgelegt.

Wien, Juli 2015

Unterschrift:

_____

Alexander ROTHENEDER

About this thesis

The NetLuke project is a cooperative teamwork between the authors Georg Prenner and Alexander Rotheneder. The project itself was developed and is maintained by both authors. The thesis has been separated into two parts. The first part *NetLuke – Play With Algorithms* was written by Georg Prenner on an earlier date. This work represents the second part, *NetLuke – Play With Algorithms - Part II*, whose core chapters are written by me. Like my colleague, I chose to use "we" to underline the cooperative character of the NetLuke project throughout the development process. I have to remark that some chapters, like motivation or project goals may overlap with Georg Prenner's thesis. The content of part I is not part of this thesis, but is referenced frequently.

The first part of the thesis, *NetLuke – Play With Algorithms - Part I*, has been written in May, 2013. Since then NetLuke has been improved and many flaws have been corrected. In this part I am going to point to these improvements, explain the deficiencies of the original NetLuke and outline the development guide.

Contents

Chapter 1

Introduction

1.1 Motivation

> "Algorithmics is more than a branch of computer science. It is the core of computer science, and, in all fairness, can be said to be relevant to most of science, business, and technology."[18]

Nowadays society relies on the capabilities of modern computer systems. These machines seem to be capable of doing amazing things. They control ships, aircrafts and trains. Computers solve mathematical problems in nearly no time. Running companies without IT seems to be an impossible undertaking. But how do computers do that ? Regarding to David Harel the computer itself is a quite primitive machine:[...] "*A computer directly can execute only a small number of extremely trivial operations, like flipping, zeroing, or testing a bit.* "[...] [18] He points out that these bits represent the external stimuli from a keyboard for example. The value of the bits are also responsible for the reaction to an given input and the resulting output. But how to transform those trivial operations that can be directly executed by the machine, into a more complex task ? All this is defined by various processes that in turn are based on algorithms [18].

In modern software environments the underlying algorithms and data structures remain usually hidden even for software developers in some cases. However the fundamental knowledge of computer algorithms and data structures is very important to analyze established projects or to find new, more effective, solutions for existing problems in the field. Or as R. Sedgewick and K.Wayne explain it: "*The primary reason to learn about algorithms is that this discipline gives us the potential to reap huge savings, even to the point of enabling us to do tasks that would otherwise be impossible.* "[33] This shows the importance of a consolidated knowledge of data structures and algorithms. For today's computer science students it is crucial to know how these algorithms work and which one suits best for a given problem.

At the university students are being confronted with algorithms in many different courses. Amongst others, algorithms are part of Mathematics, Economics and Software Development. For students of computer science there is mostly one course dedicated to algorithms

and data structures. In those courses students are usually provided with a variety of different learning materials, which can be divided into three categories.

- textbooks and professional literature
- examples written in pseudo code
- PowerPoint slide-shows presenting the actual sequence of an algorithm

The first two categories represent the topic in a very static way. "Static" materials can be very useful to teach the basic theories of a specific algorithm or a data structure but when it comes to illustrating the actual steps,that are including in algorithm execution, those materials might be insufficient. As a result students may find it quite difficult to achieve a fundamental knowledge of algorithms just relying on these sources. Or as Hansen et. al describe it: "*COMPUTER SCIENCE STUDENTS generally find the analysis and design of algorithms to be a difficult subject. One reason for this may be that an algorithm describes a process that is both abstract and dynamic, while the methods typically used to teach algorithms are not.* " [...] [17]
PowerPoint slides or videos provide a more dynamic approach. These media are capable of presenting every single step of an algorithm and thus might lead to a better understanding of how an algorithm actually works. This considerations led to the film "sorting out sorting". The film, produced in 1981 by Ronald Baecker at the University of Toronto, shows nine internal sorting algorithms on different data sets and is about 30 minutes long [2]. The functional principles of every algorithm is shown and described by a narrator. The algorithm steps were presented slowly enough that the narrator could explain in detail what is happening and that the viewer has enough time to absorb it. The film is considered as a success and over 600 copies of it have been sold to other universities.
But is it really enough for a student to understand complex algorithms when they are being presented to him ? Even when someone explains every single step, like in "sorting out sorting" the learning keeps being a passive action. From our own experience we can tell that this might not be enough to gain substantiated knowledge. A possible solution might be the use of Algorithm Visualization (AV) software in teaching algorithms and data structures. In 1994 a study by Andrea W. Lawrence et. al has been published that examines the influence of using animations in teaching algorithms [25]. In their paper the authors examine whether there is a positive influence on the understanding of algorithms on students that have been given the possibility to use an AV software. The results were quite fascinating. They showed that the restricted access to an AV software on its own is not leading to an improved understanding of the subject. It is in fact the interaction that leads to amendments. Students who were allowed to use their own data sets showed increased capabilities of understanding the algorithms way of working. Another very important factor is to give students access to the AV software outside of the class room.
Another study from Hundhausen, Douglas and Stasko published in 2002 points out that the "*form of activity* " (the way the user interacts with the system) is the most important part when building a successful learning environment [6]. The authors alaso remark that content, just being presented to the students, have zero to little positive influence on the general understanding. A significant increase in understanding algorithms can only be achieved by an interactive AV system. Only learning environments that allow the user

to work with own data sets, to compare the output or the steps of algorithms, with the users own prediction are better than common learning materials. This claim is supported by a variety of other studies. For example from Bryne et. al in 1996 who point out that: " *It seems clear that the sheer use of algorithm animation does not automatically enhance learning*" [...] [5].
Putting all these findings together, a modern AV tool that is really capable of supporting the students in learning and understanding the principles of algorithms should therefore provide the following:

- it should give students the possibility to create their own examples, using their self created data sets with a seamless variation of difficulties
- it should allow the student to control the visualization step by step
- it should provide a self explaining graphical user interface that
  - allows the user an easy control of the system without much time needed for orientation (self explaining)
  - shows all information to the user, necessary to fully understand the principle of the algorithm, without overloading him with information
- it should provide a satisfying user experience (the user should enjoy working with the system)
- the system should provide an appealing graphical representation
- it should be available to the student outside of the class

When learning for exams we often used YouTube Videos or other AV systems to get a better understanding of the algorithms that we needed to understand. All these systems never fully complied to our expectation of a modern learning tool. (see chapter 2) During our bachelor work my colleague Georg Prenner and me for the first time came in touch with the "LUCAS" learning platform. (see Shikuta [32]) Since then our interest in this topic started to rise. When it came to choose our topic for the master thesis we had the chance to work on a new version of "LUCAS". With the help of our professor, Erich Shikuta, we developed and implemented a prototype of a web-based and modular AV system which is very close to our vision of a modern learning tool. Together with professor Shikuta we named the project NetLuke.

1.2 Problem Definition

Since the next section is extensively covered in the first part of this thesis, written by Georg Prenner, I will only give a brief overview of the problem definition. For a detailed exposure I reference to the equivalent sections in the first thesis. (see "NetLuke - Playing with Algorithms - Part I " [13])

1.2.1 Educational effectiveness

Many examples show that the idea of using AV system to provide support for students is not a new one [2]. However, as above mentioned studies show, not all of them really succeeded in giving the students a better understanding of algorithms. The question is, what defines a successful AV system. One very import keyword when it comes to designing a AV platform for the educational purpose is effectiveness. But how can effectiveness in this context be defined? From our point of view the AV system should provide a detailed step-by-step representation of the algorithm and the possibility to interact with the system anytime. The user should be able to control every step of the algorithm (eg: step-back, step-forward, reset etc.). From our point of view, this features combined with the given possibility to use self created data sets is the foundation for an effective AV system. Fine grained control also leads to the benefit that teachers can use the AV system to demonstrate specific operations of an algorithm directly without being dependent on external resources.

Many AV systems, we came across during our research, use highlighted pseudo code to demonstrate the principles of the algorithm. The educational effectiveness of such descriptive methods is questionable because they presume basic knowledge. In turn, a lack of understanding can lead to frustration and in abandoning the task of understanding the subject matter. But the use of an AV system should have the opposed effect. Hundhausen et. al put out that AV system are most successful if the technology "*[. . .] is used as a vehicle for actively engaging students in the process of learning algorithms.*" [6] An effective visualization is therefore a key factor when designing a successful AV system but becomes quite difficult to design when the complexity reaches a higher level. Richard K. Lowe outlines this problem in his study and proposes that animations within a visualization need to be highly expressive [27]. He describes two different types of problems that may arise: *overwhelming* and *underwhelming*.
The first problems arise when the information is being presented in such a way that the user is unable to analyze and absorb it effectively. For example, if the animation of an algorithm presents information very quickly, the user may be overwhelmed by the amount of information.
Underwhelming can be described as quite the opposite; The system fails to engage the learner to make him understand how the algorithm really works. Just watching an animation may lead to a false understanding of an algorithm. The user needs to do more than just observing [27]. With that in mind the process of designing an algorithm visualization can be described as a tightrope walk. It needs to make the user understand why a particular operation has been carried out without overburdening him with too much information.
Another very important factor is the use of **explanatory features**. We believe that the system should provide the possibility to save instances of the created examples including their history of interaction. It should highlight important steps using visual effects like transitions or color changes. While learning every student creates his own understanding of a specific algorithm [20]. Our believe is that the expedient use of animations can support this cognitive process by guiding the user's attention to the important parts of the algorithm execution.

1.2.2 User Experience

In order to fulfill its purpose, helping students to learn algorithms, an AV system must be easy accessible. A long training period would therefore be quite inefficient. For that reason the user interface (UI) should be simple, easy to understand and support the user in his tasks. Or as Galitz describes the topic: " *To be truly effective, good screen design requires an understanding of many things. Included are the characteristics of people: how we see, understand and think.* " [12] Doubtful buttons placed into overlapping menus lead to confusion and refusal towards the systems. A lot of tools, we came across during our research, were based on such an old and complex GUI. We get the impression that the GUI design played a minor part in the development. However, our believe is, that a visual appealing and structured presentation can significantly increase the level of motivation and commitment. A well designed UI is even more important in these days where the typical user has a higher expectation concerning the interface he has to work with.

1.3 Project Goals

The original NetLuke system has been developed around two years ago. In this time some minor and major issues occurred that we took care of. Besides the resolving of these issues we also implemented some new features and called the system NetLuke Version 2. Due to this reason I first give a summary of the original project goals followed by a description of the project goals of the new version. For a deeper look into the original project goals I once more refer to the thesis of Georg Prenner [13].

1.3.1 Original Project Goals

We wanted to build a modern and good looking AV system that effectively supports students in learning algorithms and data structures. The platform should be easy to use, easy to understand and it should motivate students to actively deal with the topic. Users should therefore be given the opportunity to practice the topic using self generated examples. We also wanted the user to have the possibility of saving/loading and exporting of images. Another target group are teachers who should be given an easy possibility to demonstrate their examples to students.
An additional goal was to design the system "developer-friendly". Netluke needs to be easy to extend by new topics. Therefore developers had to be given the possibility to focus on their key tasks: extending the system with new algorithms or data structures including the logic and the visualization. This means that NetLuke's core design needed a plugin development system that is as independent from the core system as possible. Or as Prenner puts it out: " *[...] A developer should not be concerned about placement of control items within the UI, or how visualization components interact with the business logic in the background - just how to user interfaces provided by the system.* " [13]. The systems itself should be robust against errors.

Another very important goal was to achieve the best possible compatibility with the diverse operating systems. For that reason we decided to create a web platform, which is platform independent by principle. This decision allowed us to aim for mobile environments such as mobile phone or tablet computers as well as for the "classic" PC. Targeting

multiple platforms has some major advantages. First of all we are able to to reach a higher number of users due to the growing distribution of mobile devices. In addition potential users do not have to struggle with bulky user interfaces any more because those designs simply do not work in the mobile world. With that in mind we had to put much effort in designing a modern design that is easy to understand. (see section 3.3.3 in Prenner's thesis [13]). Another really good reason for supporting mobile devices is the **touched based interaction** which emphasizes the game-like part of the NetLuke tool. A well designed, touch based, GUI may lead to fun factor necessary for a learning tool to be used frequently.

Although our decision of developing a web based platform certainly had some disadvantages (see Section 3.1 and Section 3.5 in Prenner's thesis [13]) it allowed us to implement an AV system that comes close to our vision of an ideal and modern learning tool. Our vision of a **user centered** and **non monolithic** approach is reflected by the design of Netluke. The System needs to be flexible, it needs to run on a wide range of existing computer platforms and it should give developers the opportunity to easily create plugins.

General Concepts

The general concepts that are necessary to meet our outlined goals are well described in Georg Prenner's thesis. Because of their importance to our work I cited those specific parts of Georg Prenner's thesis. [13, p.6-7]

- *Use of concepts which strengthen modularity and information hiding in order to enforce separation of concerns and to achieve fundamental reduction of system complexity.*
- *Layered & distributed architecture which follows design patterns similar to a model view controller (MVC) approach.*
- *Multi technological approach on top of the architectural concept which connects different programming environments with each other to enable browser based visualization and server based computation in a 2-Tier manner.*
- *Loosely coupled system components.*
- *Code reuse: Usage of existing code or at least fragments from related projects for plugin development.*
- *Comprehensive analysis of existing technologies and evaluation to facilitate the proposed server architecture. [. . .]*

1.3.2 Goals of Netluke Version 2

During the implementation of the first prototype implementation of the NetLuke system weh had to make some compromises in order to achieve our goal. It turned out that some of these decisions however did not fully accord with our initial vision. Due to this design flaws, which are discussed in detail in section 3.5, we decided to refurbish the

original prototype in order to create a project that matches this earlier described idea of a modern AV system.

Instead of altering the original project concepts, which are still valid, we wanted to slightly modify the way we applied this concepts on our project. That means we decided to modernize and extend the original implementation. The most significant modifications are made on the client side. With the goal in mind to ease the algorithm implementation process for module developers, the module side development process has been modified. The heart of the new NetLuke client is the so called "NL.js" library. It provides basic graphical building blocks that, when combined, allows module developers to create their algorithm visualization without the requirement of creating their own graphical structures. We also created a new User Interface, that is highly interconnected with the library, as it was our goal to provide the same user experience on every device category. The current UI of the NetLuke prototype provides different layouts as it distinguishes between mobile and desktop clients. Another goal of the new version was to provide a tool that allows developers to better debug the client side of NetLuke. This includes the client server communication as well as called JavaSript functions and JavaScript errors. The tool is named "NetLuke Logger" and is also part of the new system.

1.4 Thesis Structure

The first chapter described the generals ideas of our project. The importance of algorithms and the shortcomings of "classical" teaching methods is discussed in section 1.1. The next section of the first chapter, 1.2 the major concepts of algorithm visualization tool are presented. The project goals and the general concepts of the original NetLuke are discussed in section 1.3. This section also covers the ideas and goals behind the refurbished version of our AV system, that we called NetLuke 2.

In chapter 2 the theories of the field and previous AV tools are presented. The first section, 2.1, describes common theories of generating an effective visalization, while related projects are presented in section 2.2. Section 2.3, terminating the chapter, describes the advantages and disadvantages of the NetLuke system compared to other AV solutions.

Chapter 3 describes the NetLuke prototype implementation. Beginning with a technical overview, that is given in section 3.1 this chapter describes the server side (section 3.2) and the client side in detail (section3.3. The graphical JavaScript library, KineticJS, that was one of the "project enabling" technologies is presented in section 3.4. The last section of the third chapter, section 3.5, is all about the design flaws we uncovered in the first version of NetLuke.

Based upon the deficiencies, uncovered in the last chapter, chapter 4 describes NetLuke 2, the refurbished version of NetLuke. The new UI, that allows the same user experience on all device categories, is discussed in section 4.1. The new libraray, that can be seen as the heart of the new client is described in section 4.2, while section 4.3 is about our new debugging tool.

Chapter 5 focuses on algorithm development within the NetLuke system. The requirements for developers and the concepts of extending NetLuke are discussed in section 5.1.

Section 5.2 is all about the module system we created for developing algorithms. This section describes the construction of a module, the concept of the server side library and module communication using RPC.

Chapter 6 is a development guide that describes all necessary steps in order to create a new algorithm module for the NetLuke system. The guide is based on the "InsertionSort" sorting algorithm. The algorithm itself, together with the necessary preliminaries for the development, is described in section 6.1. Algorithm modules consist of a Java and a JavaScript dart. The Java part, which represents the server side of the module is described in section 6.2. The implementation of the JavaScript part, that contains the visualization of the algorithm, is demonstrated in section 6.3.

In chapter 7 the conclusion about the NetLuke project is presented whereas chapter 8 is about the current project status, unresolved issues and possible enhancements of NetLuke in the future.

Chapter 2

Related Work

This chapter gives an overview of the common theories in the field.(Section 2.1). It describes the key factors necessary for generating an effective visualization and why earlier AV solutions failed in their goal to help students to better understand algorithms. Important projects that influenced our wok are described in section 2.2. In the last section of this chapter, section 2.3,an evaluation of the NetLuke system is provided. The advantages and disadvantages of our solution compared to other tools are being listed.

2.1 Theories In The Field

After the film "Sorting out Sorting" has been released in 1981 the interest of universities to use more dynamic teaching materials started to rise. This trend led to the emergence of the first AV systems like "BALSA-II " in 1988.[1] Shortly after the first papers were released that recommended the usage of this new tools amongst other common learning materials like textbooks. An example is the article *"Teaching Algorithms"* by "A.Ricardo" and "Baeza Yates". The authors recommended to include AV tools like *Balsa* when teaching algorithms: "[...] *Among them we have to mention algorithm animation tools (e.g. Balsa) to ease understanding;*"[3].

Despite these recommendations and the rise of new AV systems many computer science lectors did not tend to use these learning materials when teaching algorithms. So what was the reason for this lack of interest in this "new" learning material? An answer to this question may give Hundhausen et.al in their paper "*A Meta-Study of AlgorithmVisualization Effectiveness*". Amongst other reasons the authors claimed that teachers may "*[...] feel that it is simply not educationally effective* " [6]. Many other studies as well claimed that there have been no significant improvements in the students performances when using AV tools in their learning environments [22, 25, 34]. As we wanted our AV platform to be **educationally effective** some investigation of those AV platforms was necessary in order to learn from the previous design flaws. So we needed to find out why those AV platforms failed in helping students to better understand the topic.

The major problem of those early AV systems was, that they were nothing more but a

[1] A detailed history of the field can be found in section 2.2 of Georg Prenner's thesis [13].

graphical debugger. Those systems just held images of the current data structures that were updated according to important changes in the program code[10]. Although the visualized output described the state of a program much better than just the textual representation, those systems were not able to explain the working principles of the underlying algorithms. According to R.Fleischer and L. Kučera the simple debugging of a program is not a good way to really understand an algorithm. In their paper "*Algorithm Animation for Teaching*" the two authors point out that: "[...] *an animation of an algorithm that only shows* ***what*** *is going on without giving an idea of* ***why*** *is usually just understandable just for observers that already know the algorithm's function and background ideas, and has therefore only a very limited value as a teaching tool*" [10]. Another reason for this bad success may lie within the lack of an "*extensible algorithm animation framework* ", that includes guidelines on how to build an educational effective AV System, as Jean Greylings puts it out [15].

2.1.1 Keyfactors For An Effective Visualization

With the goal in mind to create an educational effective AV system we began our research at a very basic level. We wanted to know in which situations animations are superior to static representations. This question was answered by the study "*Animation: can it facilitate?*". The authors claim that **interactivity** may be the key to overcome the drawbacks of animations [36]. Although this study examined the usage of animations for general purposes, the outcome is also valid for animations used in learning environments. In addition this perception gives an explanation for the ineffectiveness of the early AV systems, that just represented the data without allowing much Input from the User. It can be stated out that **interactivity** is one key factor for an effective AV system.

Together with the rising number of available AV systems more and more studies were published that try to explain the key factors for designing an effective solution. In 2008 a study by E. Tudoreanu and E. Kreamer was published in which the authors reveal that animation is more effective when the "*cognitive load*" to the users is decreased to a minimum [35]. This means that it is necessary to reduce the amount of unnecessary visual impressions and side-task to a minimum so that the user can user can fully concentrate on his main task; to understand the algorithm. Or as E. Tudoreanu and E. Kreamer put it out: "[...] *it is important to design program animation tools and environments that reduce the amount of data and the tasks reqired in a visulization session*"[35]. This principle is called **noise-reduction** and is another key factor towards an effective visualization.

In 2009 a study by Jean Grealyn has been published including a list of requirements for a "*pedagogically effective system* ". The author distinguishes between requirements that need to be fulfilled in order to "*[...] engage the students in an active learning process*" and additional requirements which "*[...] provide additional information*", when fulfilled [15]. The "main" requirements are described as follows:

- Allow the user to control the speed of the animation
- Allow the user to rewind the animation
- Allow the user to generate his/her own data sets for the algorithm animation

- Allow a stepwise execution of the algorithm animation
- Provide questions for the user to predict algorithm behaviour
- Allow students to construct their own animations

These requirements do not represent a new key factor, they rather give a detailed set of instructions on how the **interactivity** between the AV system and the user should be modelled.

"*A good animation needs a sophisticated conceptualization. The major questions in the design of a good animation are: What do show? & When to update?* "[10]. This statement is from R. Fleischer and L. Kučera who in turn quote P.A.Gloor and B.A.Myers. In "*Algorithm Animation for Teaching*" the authors provide a summary of the suggestions of many other studies of the features an AV system should provide. Due to the reason that the given summary is quite widespread it partly overlaps with the requirements suggested by Jean Grealyn [10, 15]. On the other hand these summary gives a good overview about the features an effective AV system should implement. The non-overlapping features are described in the numeration below [8, 10, 14, 29]:

- **Model**: The model is the representation for the abstract algorithm. In most cases it is useful to use the key data structure of the algorithm. (e.g.: for a sorting algorithm the model should be an array, a graph algorithm should be animated by representing the exploration of a tree structure etc.)
- **Functional structure**: The "Functional structure" conducts a detailed explanation of every step within algorithm execution that is represented to the user. It is necessary for the interacting user to know **why** something happened.
- **Correctness and runtime**: The animation should show the user **why** the algorithm is correct and efficient.
- **Text**: If the animation system is used outside of the class it should include text explanations as there is no teacher or lector available who could explain the actual processes.
- **Uniform representation**: An AV system might provide animations for many different type of algorithms. It is considered helpful to create a uniform design throughout all animations in order to reduce the time the students needs to adjust. A uniform layout includes the arrangements of the buttons, the representation of information on the screen and so on.
- **Simplicity**: The design as well as the control of the Algorithm animation should be kept as simple as possible. Basic operations like "*forward*", "*backward*" and "*run*" are sufficient in most situations. If the system allows the user to work with self-generated data-sets, the functionality should be easy accessible.
- **Fun**: A well designed AV systems creates interest in the topic. Boring animations and a long learning period should be avoided. The user should have fun when he uses the system as this automatically enhances the attention paid to the topic.

The visual sense can be considered as the most important one when it comes to studying (at least in this area). As an effective AV system comes hand in hand with an effective visualization it is necessary to consider the above features and key factors when designing a new AV tool.

2.2 Related Projects

We examined some, already existing, solutions like **AlViE** or **TRAKLA2** in order to be able to classify our design of the NetLuke project.[2]

	Direct Data Input	Pre-defined/Random examples	Custom data input	Dynamic Animations/Visualizations	Scripting/Configuration Capabilities	Pseudo Code View	Help View	References	History Mode	Visualization Export	Quiz mode & Questions	Debug System	Developer Guide	Web Access	Mobile OS Compatible
AiA	✓	✓	✓	✓[1]	–	✓	✓	–	✓	–	–	–	–	–	–
ANIMAL	–	✓	✓[1]	–	✓	✓	–	–	✓	–	–	–	✓[1]	–	–
LUCAS	✓	✓	✓	✓	–	–	✓	✓	✓	✓	–	–	✓[1]	✓[2]	–
HalVis	–	✓	✓[1]	✓[1]	–	✓	✓	– [2]	✓	– [2]	✓	–	–	–	–
AlViE	–	✓	✓	✓[1]	✓	✓	–	✓	✓	✓	–	–	–	–	–
JHAVÉ	–	✓	✓[1]	✓[1]	✓	✓	✓[1]	–	✓	–	✓	✓	✓[1]	✓	–
TRAKLA2	– [3]	✓	– [3]	✓[1]	–	✓	–	–	✓	–	✓	–	–	✓	–

Notes: 1: Partially available, 2: Unclear/Not evaluated due to missing information, 3: Included in feature list, but not available

Table 2.1: Features of different AV tools, taken from [13, p.25]

The results were quite fascinating as every AV system seems to follow a different approach. The **TRAKLA2** system for example requires an online registration and is quite difficul to use. Due to this reason we do not consider it to be the optimal solution for learning algorithms on your own. **ALViE** on the other hand comes with a lot of supported algorithms has a consistent User-Interface and provides a *visulization exporting tool*. A disadvantage of the platform is, that it requires XML input data to work on. This is not a big deal after a while but requires too much time in the first place. **LUCAS**, the predecessor of NetLuke, in turn enables the user to easily work with self generated data sets. The system incorporates many algorithms that are easy to control and well explained. The problem however is, that most animations vary in look & feel and that some implementations contain bugs.

[2]This topic is alreday extensivly covered in part I of this thesis written by Georg Prenner [13]. Only a short summary will be provided here due to this reason.

As we were interested in the featureset provided by the other platforms we compared them on the basis of 14 features we considered to be important. The above shown table (2.1) is taken from the first part of the thesis[13] and shows the results of this comparison.[3] As one can see none of the compared AV tools provide support for mobile devices, which is a bit surprising with the growing number of smartphones or tablets kept in mind. Some tools allowed to be used over a network and come with a special client application. While the stepwise execution is supported by all platforms, the support for exporting the visualization is only provided by **LUCAS** and **AlViE**. When it comes to direct data input the user has to choose between **LUCAS** and **AiA**. While all platforms provide to possibility to go "back" in algorithm execution, it is rarely possible to edit the underlying data structure after the initial start.

The overall implementation quality of the AV tools we examined was not really satisfying. Many of the tools one can discover using a quick Google search are simple applets and are mostly outdated. Sometimes they don't even start when using a modern web browser or a modern Java Environment. The User Interface is often incomprehensible and most of the tools do not provide functionality to use "self-created" data sets. Instead these tools follow a rather descriptive approach: Just the actions of an executed algorithm are shown, but without any further explanation or the given possibility to interact with the system. In most cases applets only come with the support for one algorithm resulting in the need for students to find a tool for every algorithm. Although the overall experience is far better when using AV systems like **TRAKLA** or **AlViE**, even those tools lack regarding usability; Be it a complicated installation process(**TRAKLA**) or a complex data input(**AlViE**). Prenner sums it quite well by stating: "If one must compare academic AV systems to commercial educational software, it is evident that the majority of AV systems leap behind quite bad in many areas."[13]

2.3 Evaluation

In the design phase of the Netluke system we took care to involve the above presented **keyfactors** and **features** as well as the theories described in section 2.1 of the first part of this thesis[13].

As stated in section 1.3.1 we decided to create a web based solution as new technologies like HTML5 combined with diverse JavaScript libraries like Jquery seemed to be the ideal technical base frame for a modern GUI. Modern web based applications enable a good user experience and can be executed on nearly every device through the independence of the operating system.

2.3.1 Advantages of NetLuke

All functionality is provided in NetLuke's main window, that is available after a successful user login. It was very important to us that users recognize all available options at the first glance.[4] The first step for the user is to select an algorithm. Every algorithm comes with a short description and some useful references to additional information. When an

[3] A detailed explanation of the features can be found in the first part of this thesis[13]

[4] In NetLuke2 we implemented a new GUI due do reasons described in section 3.5

algorithm is selected, the user can decide whether to use his own data or to generate a random data set. Afterwards the algorithm instance is being created and the **model** of the algorithm is being shown on the drawing area. The user can drag the model (be it an array or nodes from a tree) freely around the drawing area. If the user is ready he can start the algorithm animation. It was very important for us to keep the algorithm control simple (**Simplicity**). When a sorting or a graph algorithm is chosen the user can conduct a step forward (NEXT), a step backward (BACK) or execute the animation until the algorithm is finished (RUN). Other algorithms, like trees, depend on a user input after every step. If such kind of algorithm is chosen we support the (BACK) and (NEXT) methods. A (RUN) method is not possible due to obvious reasons. Every step during algorithm execution comes with a textual description including information why this step has been executed (**Text & Correctness and runtime**). As all graphical elements that are used to visualize the various algorithms are based on the very same JavaScript library[5] a **uniform representation** throughout all algorithms is provided. A special feature which makes NetLuke superior to other AV systems is the possibility to store current algorithm instances (including their current execution state) and load them again at another point of time. The saved instance can also be loaded from another device. An additional feature we support is to continue a current session on another device without loosing the current instance. For example a user can begin the session on the desktop computer and finish it on his mobile phone. It is also possible to save images of interesting instances. This features combined with the easy handling of the systems increases the interest in the topic and makes the user having **fun** when using the NetLuke system.

Thanks to modern web technology (HTML5 combined with various Javascripts libraries) we are to able provide dynamic and highly expressive animations. As modern smartphones and tablets offer a high performance combined with fast JavaScript engines this animations are also available on mobile devices. A very important feature that NetLuke(2) shares which LUCAS, its predecessor, is the extensibility. This means that students or other interested developers can add new algorithms to the systems. For this purpose we implemented a module system, which is described in section 5.2[6]. As the intended use of NetLuke is to help students to better understand algorithms and data structures, its primary site of operation will be universities or other educational establishments. For us the extensibility of an AV system is crucial as developing new modules keeps the system up to date. For higher class students it may also be possible to make module development for NetLuke a part of selected courses.

2.3.2 Disadvantages of NetLuke

As already described NetLuke is a web application. In order to provide features as saving instances or to continue user sessions on another device a server-client architecture was necessary. While the user input and the graphical output is managed on the client, the instance data is stored on the server and delivered to the client side when necessary. This design decision comes with some disadvantages as well.

[5]The first version of NetLuke uses the KineticJS library while NetLuke2 is based on Konva.js, which is a fork of KineticJS. Chapter 3.4 will explain this topic in detail

[6]The chapters 5 and 6 of this thesis can be seen as a developer guide, describing the necesseray steps to create a new module for the NetLuke system

The NetLuke server side is based on Java (J2EE application) and requires an application server like Tomcat to be executed. Compared to other AV tools like AlViE, TRAKLA2 or LUCAS the effort for the initial start up is higher due to the time that is required setting up the application server. To minimize this effort we developed the **NetLuke-Standalone** package which comes with its own Tomcat instance and can therefore directly be executed[13]. Nevertheless we have to admit that a standalone application that comes with an installer is more comfortable regarding the first setup[7]. The client-server architecture results in another disadvantage when it comes to developing new algorithms for the system. A developer who is going to implement a new module for NetLuke has to provide fragments for the client side as well as for the server side. Due to the reason that the server side part of the module has to be written in Java while the client side is based on web technologies like HTML and JavaScript, the module development process can be rather challenging (See section 5.1). A detailed description of the system, it's components and the underlying technology will be given in chapter 3.

A feature we did not implement into the NetLuke system is an interactive Quiz mode that enables the user to guess the potential next step during algorithm execution. Although such a feature might have improved the overall educational effectiveness of our learning platform, we decided to cut out the "interactive quiz" feature. The additional client server communication that would be necessary and the extra time being spent when developing algorithms (every algorithm would have needed a separate implementation for the quiz mode) outweigh the potential benefit. We also do not explicitly provide a "Pseudo Code View". Although every developer can provide pseudo code of the algorithm in the textual description of every step this workaround can certainly not fully compensate this missing feature.

[7]It has to be noted that the NetLuke server is designed for a non stop operation and not meant for being started and stopped after every usage of the tool

Chapter 3

The Netluke System

This chapter aims to give a description of the first NetLuke systems, the architectural decisions we have made and the technology it is based on. An overview of the architecture, the NetLuke system is based on, is given in section 3.1. Section 3.2 will describe the server side, including topics like data persistence, remote procedure calls and session management. Section 3.3 is about the client side. The usage of the HTML5 canvas, its possibilities and restrictions and why we needed a graphical library (KineticJS/Konva) in order to provide an expressive visualization is also being described in this section. In the last section of this chapter (3.5) I will describe the deficiencies of the first NetLuke version which were the inducement for us to develop an improved version that we named NetLuke 2.

It is the purpose of this chapter to outline the components of NetLuke that are basically required for algorithm visualization tasks. A trivial understanding of the interplay between these components is necessary for developing new algorithms. Other components, even though equally important to the proper functionality of the system, will only be briefly mentioned and can be looked up in detail in the thesis of Georg Prenner [13].

3.1 Technical Overview

So far we explained that the NetLukes system is based upon a client-server architecture. This design decision was necessary in order ..

1. ... to fulfill the requirement to access NetLuke from multiple devices running different operating systems.
2. ... enable features like persisting and continuing user sessions.
3. ... to enable an installation free usage of NetLuke (on the client side) without much set-up time required.

This architecture is more complicated than a “normal client application“ as the whole application comprises two parts with different functions. Broadly speaking the server

side manages the algorithm calculation while the client side has to manage the graphical output and the user input.[1] Another task is necessary to manage the communication between client and server.

We decided to use the Java programming language for implementing the server side. Therefore the server side is a J2EE application that requires an application server to be executed. Although we developed the project on the Apache Tomcat server, every other application server like JBOSS or Jetty is able to execute it. When started the server loads all the algorithm server modules[2] and provides functionality for the communication with the client side. Every algorithm module has to provide certain methods that can be called from the client in order to offer **BACK** and **NEXT** functionality. The server also stores the user data and is responsible for the session handling. User data and instances are stored in a relational database. Whenever a session is ended or a user saves an instance the current algorithm state gets serialized and saved into the database.

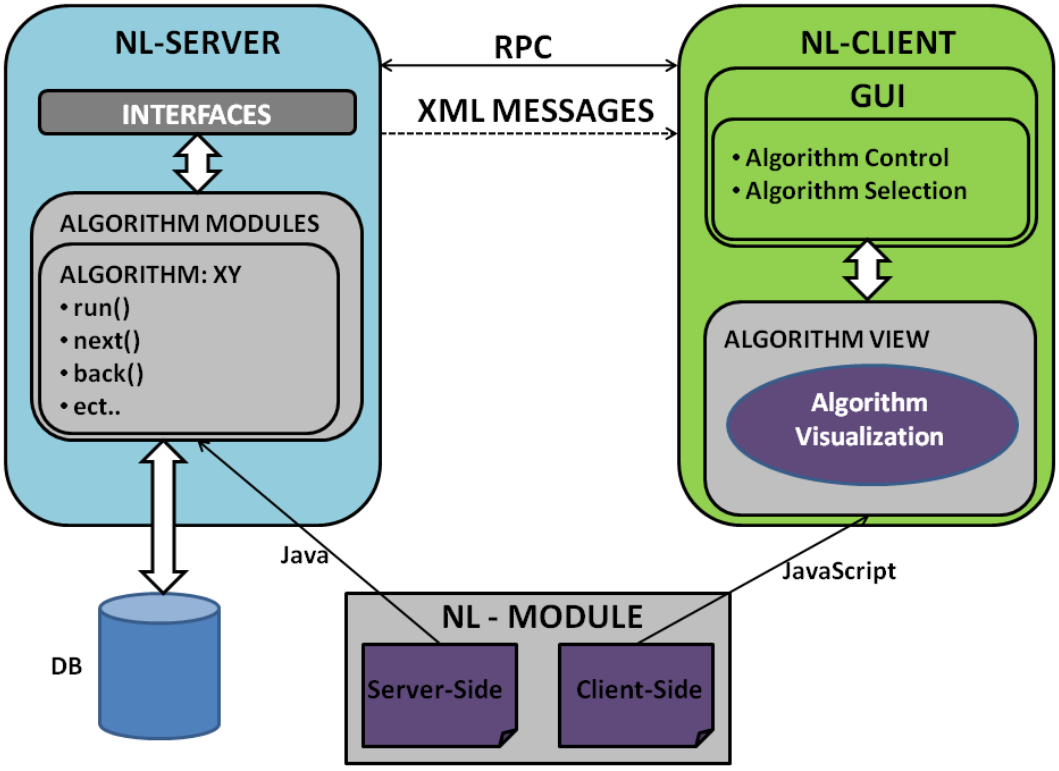

Figure 3.1: Overview of the NetLuke System

The client can be seen as the representation of the system to the user. It provides the user interface, processes the user input and visualizes the algorithm, as illustrated in figure 3.1. The client connects to the server using the provided communication interface, sends the user input and shows the response from the server to the user. Since the server needs to be aware of the client he is currently communicating with, we required a stateful connection. We decided to use RPC (Remote Procedure Calls) to achieve this. Although

[1] In NetLuke the client side can contain some algorithm logic as well. The actual realization depends on the developer's implementation of the particular algorithm

[2] Algorithm modules in NetLuke consist of a server and a client part. The server side modules are being loaded during the server startup

the NetLuke server also supports clients using stateful Web Service operations (JAX-WS) (see chapter 5 of first part of this thesis[13]) we recommend using RPC, as the Web Service functionality has not been fully tested. The following passages will therefore only apply to our client which is based on RPC. We decided to create a web application based on HTML5 and various JavaScript libraries in order to create a modern, good-looking and platform-independent client. The usage of an algorithm requires the successful deployment of the server module on the NetLuke server as well as the integration of the client part into the client. After a successful login (users are required to assign the algorithm instances) the user can select from all available algorithms. It is the client side module of an algorithm that defines which control elements are displayed and how the response from the server after an occurring event is presented to the user. Whenever a user performs a step forward/backward in algorithm execution the corresponding function on the server side is being called and the result is being transmitted back to the client. The server answer comprises instructions for the client embedded within XML messages. We choose the XML format as those messages can be directly interpreted within JavaScript. The server response gets then interpreted by the client and the output (the algorithm visualization) is presented on the GUI. For a proper functionality module developers have to follow some guidelines that will be presented in the chapters 5 & 6.

3.2 The Server Side

This section aims to explain the server side of the NetLuke system. All components that enable required functionality like module method discovery or method invocation are being described. A short insight in the NetLuke module concept, which is further illustrated in chapter 5 will also be given, as the module composition is highly interconnected with the mode of operation the server is based on.

The best way to describe the several components of the NetLuke system is to first outline the different tasks and then to describe how these tasks are being accomplished within the server. In our AV solution the server is responsible for ...

- ... loading and instantiating the server side part of algorithm modules.
- ... discovering and invoking the methods offered by the algorithm modules.
- ... providing a communication interface for the client in order to access the methods provided by the algorithm modules.
- ... module instance handling (saving & restoring instances)
- ... providing a connection to the database (persistence)
- ... the user management & session handling

3.2.1 Module Insight

A basic understanding of the NetLuke module concept,which will be described in detail later, is required to comprehend the functionality of the server. Every algorithm module

comprises the logic and implementation required for the visualization for one single algorithm. The server part of the module is a JAR file that encapsulates the algorithm logic. The algorithm logic itself is a Java Project that has to be based on our *netluke-core.jar*[3] library. The module developer also has to ensure to follow a specific package structure within the created Java project. The JAR file itself is being added as a library to the NetLuke server project. It is also required to add the new library to the *netluke.xml* configuration file as discussed in chapter 6.2.

The functional principle can be best described by using an example: Consider a user creating a new algorithm instance and afterwards using the **NEXT** button in order to initialize one step forward in algorithm visualization. The new instance is created by calling the **constructor** of the algorithm Java module. Each algorithm has to provide methods that allow a stepwise execution. These methods need to be accessible from the client side. The user action now calls the provided **next()** method of the previously created instance. The method gets executed, the results of the action stored to the instance and the results transmitted back to the client, where the visual output occurs. The question that now may arise is: Which components are responsible for the communication and how is the server capable of recognizing the loaded modules and the provided methods?

3.2.2 The NetLuke Service Layer

The Service Layer comprises the matching endpoints for Clients to connect to the NetLuke system. It is mostly detached from the underlying structure. Thus core components and modules are not aware of the Service Layer. The design of NetLuke allows multiple technologies to connect to the system, if a matching endpoint in the service layer is provided. In the first version of NetLuke we implemented matching endpoints for clients using ...

- ... RPC connections
- ... as stateful Web Service connection based on Java API for XML Web Services - Jax WS (see [13, p.66])

As for our client design only the method using RPC connections is relevant, I will only go into this method. For a deeper insight in the subject I refer to the chapters 5 & 6 of the first part of this thesis[13].

Finding an appropriate RPC solution was very challenging in the first place since NetLuke requires a stateful connection. Because the development of our own communication framework, based on manual AJAX calls, would have been an enormous effort, we decided to search for already existing projects. Finally we found a solution that perfectly suited our requirements. We decided to use DWR which stands for *DirectWebRemoting*. DWR is a Java Framework that enables the interaction between JavaScript on the browser side and Java on the server side. No additional plug ins on the browser side are required by DWR in order to work.

[3]The netluke-core library provides basic functionality for algorithm development and is further explained in section 5.2.1

> "DWR works by dynamically generating JavaScript based on Java classes. The code does some Ajax magic to make it feel like the execution is happening on the browser, but in reality the server is executing the code and DWR is marshalling the data back and forwards. This method of remoting functions from Java to JavaScript gives DWR users a feel much like conventional RPC mechanisms like RMI or SOAP, with the benefit that it runs over the web without requiring web-browser plug-ins." [28]

The only thing that is required for DWR is a J2EE application server with Servlet API 2.2 or later. A servlet is then used by DWR to create a single endpoint for a configured Java class. The behaviour of the class is exported to JavaScript. In NetLuke we created the *NetlukeDWR* class to serve as the module endpoint. For this to work the class has to be registered in the *dwr.xml* configuration file, as listing 3.1 illustrates.[4]

LISTING 3.1: dwr.xml DWR Configuration

```
1  <!DOCTYPE dwr PUBLIC
2          "-//GetAhead Limited//DTD Direct Web Remoting 3.0//EN"
3          "http://directwebremoting.org/schema/dwr30.dtd">
4  <dwr>
5    <allow>
6      <create creator="new" scope="session" javascript="remote">
7        <param name="class" value="at.ac.univie.netluke.service.
8                NetLukeDWR"/>
9      </create>
10      ....
11   </allow>
12 </dwr>
```

On the client side the **engine.js** and the **remote.js**, which is dynamically created, files have to be included. DWR generates a JavaScript representation of the NetLukeDWR class. On the client side this class can be accessed by using the **remote** object.

The *NetLukeDWR* class is designed to serve as an endpoint for module control functionality within NetLuke. It can be seen as "middleman " between the client and the server side. The client can only call functions provided by this class, which in turn are executed by the algorithm modules. In NetLuke every user gets assigned a unique session. (Stateful connection). Whenever a new user logs into the system a new session is created that is unique to this user until he logs out again (Stateful connection). A new instance of the *NetLukeDWR* class is being created for every session as shown on line 6 in listing 3.1. Since the client side is not aware of the available algorithms, another component is necessary in order to control and manage these modules. This component is the so called *NetLukeController* class.

A second class that uses DWR within NetLuke is the *MessageController*. This class is used to send reverse AJAX messages to clients (eg. logout information). There is one major difference between these two classes though: While the *NetLukeDWR* class is responsible for handling the *NetLukeController* objects for every user (see chapter 5.2.1), which requires an instance of this class for every created session, the *MessageController*

[4] See [13, p.65] for further details

class is created only once when the DWR servlet is being started (one object for every user).

3.2.3 The NetLuke Controller

The *NetLukeController* is "*the heart of NetLuke* " as Georg Prenner describes it in the first part of this thesis [13, p.67]. This component connects the service layer with the algorithm modules. The main task of the *NetLukeController* is to process requests to the modules from the generic interface provided by the service layer. As mentioned earlier, algorithm modules on the server side are deployed as libraries. Hence the system does not know anything about the implementation details (provided methods, constructor name, etc.). The *NetLukeController* discovers these classes and their functionality during runtime in order to make them available for users [5].

Let us reconsider the above used example: Whenever a user logs into the system and chooses to work with an algorithm on the client side, the according module on the server side is being activated. Therefore the client has to send the *setActiveAlgorithm(name)* command. The name property in this command has to match one identifier in the *netluke.xml* configuration file. Since this is the first RPC call from the client using this user session (hence the user just logged in) DWR automatically creates a new instance for this user by passing the username of the session owner to the *NetLukeController* constructor. Before the *setActiveAlgorithm(name)* method can be executed the NetlukeController (NC) has to perform several other tasks:

1. The NC performs a database read operation and checks if any previous instance of any algorithm exists for this specific user. If so, these previous instances get loaded and saved into a Java Hash Map that is exclusive for this instance of the NC. After this operation all previous instances (including data objects and algorithm attributes) are available to the user.

2. The NC consults the *NetLukeXMLProvider* component to get information about deployed algorithms. In order to provide this information the *NetLukeXMLProvider* reads the *netluke.xml* configuration file and returns the algorithm class names. Based on this result the NC creates Java hash maps that contain all the constructors and methods provided by the various modules. Since the methods and constructors of the modules are unknown to NC, the component takes advantage of Java Reflection in order to gather the required information.

3. Since the algorithm modules are now available to the controller he loads the constructors and the methods of the algorithm modules into the memory. We decided to use two Java Hash maps for this procedure. One hash map is used to store the constructors while the other one stores the algorithm methods. The *setActiveAlgorithm(name)* methods is executed to declare that the specific algorithm module is now ready for being invoked using RPC calls.

Now that the algorithm module is being loaded it can be used by the client. In general

[5] See [13] chapter 5.3 for further details

the first command is a constructor call in order to create a new instance of the algorithm. This is simply done by using the remote object provided by DWR:

- `remote.invokeIntegerConstructor('Algorithm xy',{params});`

The *invokeIntegerConstructor()* is provided by the *NetLukeDWR* component. It verifies whether or not the call is valid and then passes it on to the *NetLukeController*, since the service layer does not process requests. The NC then checks if a constructor with the name *"Algorithm xy"*, which accepts the given arguments, can be found among the loaded modules. If so the constructor is called and the returned object, which is type of *"Algorithm xy"*, becomes the currently active instance object for this session (user). In case the constructor call should fail, the server log will provide information that help to track down the issue. In both cases the client receives a response code from the server: **code: 200** to indicate that the operation was successful (the instance has been created), or **code: 404** to indicate that the constructor call has failed.

When the instance has been successfully created the user may want to perform different operations on this newly created algorithm. In order to this, the provided methods of the algorithm have to be called. Calling methods is quite similar to calling a constructor. Once again the remote object is used on the JavaScript side to call a method provided by the *NetLukeDWR* component:

- `remote.invokeMethod('method-name');`

The *NetLukeDWR* component extracts the given argument and dispatches the call to the NC which then searches for a method with the name *"method-name"* in the previously indexed methods that are implemented by *"Algorithm xy"*. There is one major difference between method and constructor calling however: The return value of invoked methods is a string, that is being directly transferred to the client and not a simple response code. This string is module specific and contains the data that is necessary for the client in order to visualize the algorithm animations. For example if the user invokes the *next* method, the return string contains the changeset [6] that is required to display the next step in the algorithm execution process.

3.2.4 The Server Side's Remaining Responsibilities

Now that we have illustrated how the different modules are loaded on the server and their methods are being provided to the client it is time to describe the other tasks of the NetLuke server. I will only give a brief overview to the remaining tasks. For a detailed description I refer to the first part of this thesis [13, p.67 - p.80].

Besides providing generic interfaces for module communication the server in NetLuke is also responsible for:

[6] Changesets will be described in chapter 5.2.3

- **Module Instance Handling:** Module activities in NetLuke can be divided into manual and automatic operations. Manual operations are user triggered and subject to the Netluke Controller. Automatic operations, like automatic instance serialization due to a connection loss, are carried out by other components that are unknown to the NC. NetLuke allows the reuse of instances independent from the original session. For example if a user logs into the system again, the previous instance will be restored without any user action required. All the necessary tasks are executed automatically. The system is also capable of restoring instances from sessions that have been invalidated. Hence a NC object is exclusive to one user we needed an object that stores all module instances that are currently loaded on the system (independent from the user). This is carried out by the *NLGlobalInstanceHolder* object. This object is a singleton, that is created during the startup of the server. When first called the NC reads all instances for a user from the database and saves them into the *NLGlobalInstanceHolder* object. The same task is executed when the NC calls a constructor. As a result the *NLGlobalInstanceHolder* holds references to all relevant algorithm instances on the system, independent of the session.

- **User Managament:**Since NetLuke is a multi-user platform a registration is required by everyone who wants to work with the system. During the registration process the user needs to enter a username, a password (two times) and a valid email address. The component within NetLuke that is responsible for the user registration is the *UserController*. It checks whether or not the chosen username is unique and if the entered password pair matches. A unique username is essential as it is used by the system as a key to associate created algorithm instances with users. After a successful registration, which is indicated by sending an appropriate response code to the client, the user is able to log in to the system and work with the provided algorithms. Registered user are assigned a 128 bit long immutable universally unique identifiers (UUID). This UUID is used to identify a user before he performs a database operation. Although certainly not necessary in the prototype, this approach guarantees a future proof design. In Java logic however, the user is just identified using the username. During the registration process the provided user credentials are stored in the database backend. The password is encrypted and stored with a randomly generated salt (*hash - salt* combination). By using this method we prevent that two identical passwords have the same hash value in the database.

- **Session Handling:** The state of created objects within NetLuke is closely linked to corresponding sessions, as mentioned earlier. Session creation in turn is linked with user authentication and security matters. Due to this reason we needed a component comprising functionality for user management (authentication and authorization), security and session handling. Session handling in NetLuke also means to track the events that are related to the session, be they on purpose (invoked by the user) or not, as this information is vital to the system in order to automatically invoke necessary methods. As we are not experts in the field of security of Java web platforms we decided to use a third party framework. The decision was soon made in favor of Apache Shiro. The framework comes with all necessary features to fulfill our requirements, as the following describing statement from the project webpage shows.

> Apache Shiro [...] is a powerful and easy-to-use Java security framework that performs authentication, authorization, cryptography, and session management and can be used to secure any application - from the command line applications, mobile applications to the largest web and enterprise applications.“[19]

Shiro's security functionality, that is based on a filtering system, can be configured on very fine levels. That was necessary as we had to define some pages (like the login page) where anonymous access is granted and no session is being created. A reason for the systems power is, that the filters can be configured using the configuration file (**shiro.ini**) or using Java properties. The main concepts the Shiro system is based upon are **Subjects**. A *Subject* is an object that describes a specific user and all data related with this user (session information, login credentials, etc.). Every web component within the context of shiro holds a reference to the Subject object. This is quite handy. For example when creating a new instance of the *NetLukeDWR* class that holds a reference of its owner. As the DWR service lays within the context of shiro the owner can easily be determined by using the provided subject.

NetLuke's user authentication functionality is also based on Shiro and its ability to create “authentication realms“. A *Realm* is used by Shiro to access application specific security data. In our case this data is the information stored in the database. A Shiro *Realm* adds an additional layer of abstraction on top of the provided Data Source (the database). In this way Shiro provides a standard interface to Data Sources. We created our own Realm, *NetLukeRealm*, which allows Shiro to manage to authentication process as configured in the *shiro.ini*. This happens completely transparent to the system.

Sessions created within Shiro are bound to the executing *Subject*. This is very helpful when it comes to track all user related objects and events. In order to detect events that require intervention from the system, the behaviour of the user and the components need to be monitored. These events can be separated into authentication related events and session related events. We implemented dedicated monitor classes for a proper detection of both types and attached them to Shiro using the *shiro.ini* configuration file. The *AuthMonitor* class detects **login** as well as **logout** events. If a user logs into the system who still has a valid session, including module instances, on another machine, the *AuthMonitor* invalidates the old session and transfers the objects to the new one. The *MessageController* class, which works as an observer of the AuthMonitor, then sends a message to the “old“ client machine informing it about the logout. If a user initiated logout event is captured however, the session is invalidated, and the module instance is being persisted. **Session** related events are monitored and detected by Shiro's own security manager. Though there is another component necessary in order to react to those events. This task is performed by the *SessionMonitor* class. If an expired session is being detected this component manages that the module instances, which were created within this session, are being persisted.

- **Persistance:** As stated out before NetLuke requires an underlying database back-end to save user credentials and to persist module instances throughout user sessions. In general the NetLuke system supports two main operation modes which affects the requirements imposed on the underlying database back-end: In the first mode of

operation NetLuke is deployed as a web application using a J2EE application server. Those environments will most likely already provide a database system like MySQL or MariaDB which should be used in this case. The other way to use NetLuke is as a standalone application. For this scenario we integrated an embedded version of Tomcat 7 into the package. In the second mode of operation the user most likely wants to start NetLuke right away without configuring any database systems in the first place. This requires NetLuke to automatically create and use a proper database back-end. Recapitulating the above said, we needed NetLuke to support various database systems. To fulfill this requirement the **persistance layer** has to be independent from the actual data source. Our decision to utilize the Java Naming and Directory Interface (JNDI) to discover provided data sources enabled us to achieve the required level of resource abstraction. For the actual data source, we decided to support database management systems (DBMS) that support the SQL 2003 standard.
Data sources for NetLuke need to be configured in the *context.xml* configuration file that is stored inside the "META-INF" folder of the project. Every datasource requires a "resource" entry in the **web.xml** configuration file in order to be found by the system. For both operation modes (standalone & web-application) we decided to use HyperSQL "HSQLDB" as the standard database back-end because it does not require an installation and can therefore easily be distributed with the underlying project as Suntae Kim and Bingu Shim point out [23]. The only steps that were necessary in order to get "HSQLDB" to run were to deploy the database library file, **hsqldb.jar**, into the **/lib** folder of the application server, to define an according resource entry in the **web.xml** and to define the data-source in the **context.xml** file. The prototype version of NetLuke comes with a bunch of predefined data sources, including mysql. In order for MySQL to work it was required to add the appropriate library (*mysql-connector.jar*) to the project. The below table 3.1, which is taken from the first part of this thesis[13], illustrates our predefined data sources. The default data source in our project however is **"hsql-temp"** as configured in the *DataSourceProvider* class.

Name	**URL**
hsql-server	`jdbc:hsqldb:hsql://localhost:9001/`
hsql-mem	`jdbc:hsqldb:mem:netluke`
hsql-temp	`jdbc:hsqldb:file:${catalina.home}/temp/netlukedb/netluke`
hsql-file	`jdbc:hsqldb:file:${hsql.path}/${hsql.dbname}`
mysql	`jdbc:mysql:${mysql.addr}:${mysql.port}/${mysql.dbname}`

Table 3.1: Data Sources in context.xml, taken from[13, p.78]

This data source uses a database file that is located in the */temp/netlukedb/* folder of the application server. When using the tomcat application server, the temp folder is located inside the catalina home directory (*"CATALINA_HOME"*). The data source that is to be used can be changed on demand. When the system starts up the *DataSourceProvider* initializes the configured data source. If this operation should fail the data source is being switched to a fallback database located in the system memory (*hsql-mem*). In the next step the system verifies if a connection to the data source can be established. If it is the first start of NetLuke using a new database or the database does not contain any tables due to any other reason the

system executes the **init.sql** script, that creates the necessary database structure. Otherwise a consistency check on the database is being performed. After this the database is ready and can be used by the system.

A more detailed description of the *Resistance Layer*, the database schema that we created for the NetLuke system and information about how actual queries are performed can be found in chapter 5.9 of the first part of this thesis. [13]

3.3 The Client Side

This section is about the client side of the NetLuke system. The client is the representation of NetLuke to the user, as mentioned earlier in this chapter. Due to our architecture the client is much more than a simple web application that carries out obvious tasks like providing a user interface, rendering the algorithm visualizations or sending the user input to the server. In NetLuke the client is also a container for the client side modules (see chapter 5.2) and includes a client-side framework that provides basic functionality for client-module development. In the following a list of the various tasks of the client side is provided. Within NetLuke the client ...

- ... enables user to login to the system, create accounts or logout again.
- ... provides various control elements that enable the user to interact with the system.
- provides a drawing area where the actual algorithm visualization is being rendered.
- ... serves as a container for the client-side algorithm modules.
- ... offers a framework to developers which provides functionality for module development.
- ... sends user input to the server and shows the server response to the user.
- ... is able to distinguish between desktop and mobile devices and provides an adjusted layout.

3.3.1 The Client Structure

As we wanted NetLuke to be accessible from almost every device we decided to create a web application using various web technologies like HTML5 and JavaScript. Web applications usually consist of a static HTML structure combined with dynamic code elements which are responsible for the functionality. In NetLuke we have to distinguish between two types of dynamic behaviour:

- Dynamic content generated by JavaScript code. Examples for this category would be algorithm visualization tasks or switching the current algorithm instance.
- Dynamic components that are not related to JavaScript. This category enfolds all actions where a direct communication between the server and the client without the usage of DWR is necessary. This is the case in the first phase of the login/logout process for example.

As the server side is based on the Java technology we chose to use Java Server Pages (JSP) as the foundation for the client. "*JSPs are Web pages that have Jave code mixed in with HTML statements that define the page.* " That is how Java Server Pages are described by Robert W. Janson [21]. As JSP also enable the developer to mix Java statements with JavaScript code, which is embedded within the HTML syntax, our decision fell towards this technology.

It was our intention to keep the number of different pages within NetLuke as low as possible. This is because we wanted to give every user the feeling of working with a native application and a necessary page redraw will not stay unnoticed. We managed to reduce the number of different pages required for the NetLuke client to four. In the following list each single page, including a short description of its functionality, is being characterized:

1. **login.jsp**: The *login.jsp* page is the first page that is being displayed when the NetLuke client is used. On this side basic information about the project is presented and registered users can use the provided login mask to continue to the main page. Unregistered users have the possibility to create an account by following the link to the register page. As we consider a resolution of 1280x720 pixels as a minimum for modern clients an appropriate warning message is displayed if a user wants to use the system with a lower resolution. The result of a low resolution could be layouting issues as we designed the application only for devices with HD displays. We use a combination of Java & JavaScript routines to reliable detect the user's screen resolution.

2. **signup.jsp**: This page is used to handle signup requests from new users. It therefore provides a signup form combined with a JavaScript routine that verifies whether the input date matches the requirements (eg: password at least six characters long, email address includes an "@" sign, etc.). The data is then transmitted to the *register()* method of the **UserController** where the actual user registration occurs. (see chapter 3.2.4). The user gets informed about a successful registration and can then use the provided link to the *login.jps* page to begin using the system.

3. **index.jps**: This is the main page of the NetLuke system where the actual algorithm visualization takes place. The main purpose of the page is to provide the drawing area for the algorithms (stage) together with the necessary control elements. Depending on the user's device (desktop or mobile) the *index.jps* chooses an appropriate layout using various stylesheets. (**.css files**). As algorithm modules come within separated JavaScript files the page needs to re alter itself depending on the currently loaded algorithm. This means that control elements and the current visualization elements are depending on the algorithm. Whenever a new algorithm is loaded the page changes without needing to reload. The necessary actions are mainly carried out by the **loader.js**, which will be described in detail in the next section. Besides the **loader.js** this pages include many other JavaScript files that are responsible for communication tasks (AJAX & DWR) and styling work. Beside our own JavaScript functions we heavily rely on the commonly used jQuery JavaScript library[7] [9] .

[7] http://jquery.com/

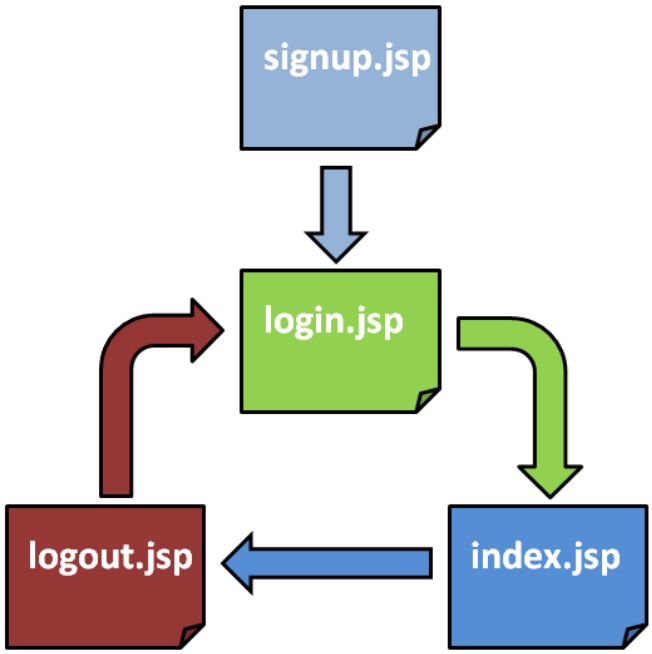

Figure 3.2: The Page Structure of NetLuke 1

4. **logout.jsp**: This page does not contain any actual content. When a user requests to logout he gets redirected to this page that is used to call several system functions. All module instances of the user are being persisted and the current user session is being invalidated by the **Shiro** framework. During this time an overlay is presented to the user who gets then redirected to the *login.jsp* page again.

Figure 3.2 illustrates the four different web pages within NetLuke. The figure also shows the path of a user when using the system.

3.3.2 The Client Library

Unfortunately, our architectural decision has certain negative aspects as well: As already mentioned, NetLuke requires the developer to provide a client side module together with the server side module when implementing a new algorithm for the system. (as shown in Figure 3.1). While our idea of providing an underlying library, which offers basic functionality, worked very well on the Java side (*netluke-core*), achieving a similar concept using JavaScript on the client side required considerable more effort. This problematic nature is also described by David Flanagan, author of "*Javascript: The Definitve Guide*", who points out that: "*Javascript does not define any language constructs for working with modules [...]*" [9, p.248]. Despite the severity of building a modular concept within a browser environment we achieved our goal of creating a library concept on the client side that offers a light weight framework and pre-defined functionality for developers. At this point I want to mention that at the time of implementing the first version of NetLuke, our (at some level) insufficient JavaScript skills may have prevented a better solution. That is one of the reasons that made us overthink the client concept and build an enhanced version of NetLuke. (see chapters 3.5 & 4)

Our main goal when designing the client side library was to support potential module developers and disburden them from unnecessary side tasks. We define "unnecessary tasks" as all activities that are not directly related to algorithm development tasks. In order to relief developers from jobs like creating control elements or designing menu layouts we decided that the system should provide all necessary UI elements. *"Therefore NetLuke has built in views, overlays, menus, modal dialogs, text consoles and panels for control elements like buttons or links."* This is how our decision is described by Georg Prenner in the first part of the thesis. [13, p.81] When designing the client side of a new algorithm module the developers can integrate these basic components in their implementation. This concept allows developers to reuse components throughout the system instead of forcing them to create a complete new control interface for every algorithm. This concept of "basic component provisioning" can be compared with the **netluke-core** library on the server side. As a lot of effort is necessary for the creation and the handling of these basic framework components we decided to use the popular jQuery library[8] and some other helping JavaScript libraries in order to ease the workflow.

All JavaScript libraries are located in the *Webcontent\js\main* folder whereas all library files related to jQuery are located within the *Webcontent\js* folder. A complete list of all helping modules can be found on page 82 of the first part of this thesis. [13]

The loader.js

The *loader.js* file can be considered to be the most important part of the client side framework. Every single algorithm implementation is depending on the loader's functionality. By acting as a container for algorithm modules the loader integrates those modules in the system. It can be described as an "*abstract layer on top of them*".[13, p.90] The loader declares "**global**" objects (like the drawing stage, access to the libraries, etc.) that are needed by the algorithm modules. In order for this to work the developers are required to interconnect their implementations with the loader.

Since all modules need to be bound to the loader this component is responsible for the instance handling on the client side. Whenever a user switches to another algorithm the corresponding modules gets loaded and initialized after the current module has been destroyed. It has to be stated that the loader.js is not directly aware of the published modules. This information is rather taken from the server and it is therefore necessary that the "***.js**" file on the client side is exactly named the same as the archive file on the server side. This can be a major source for JavaScript errors. A mistyped name or an incorrectly integrated loader may lead to errors thar are difficult to reveal.

The loader also provides functionality that is usable throughout all algorithm instances. It enables user to save/load the current algorithm, to zoom the drawing area (stage) or to export the current state of the algorithm as a picture. This is possible as all modules are required to use the *"global"* **stage** object that is initially defined in the loader. All these features work within every module without any necessary intervention from the module developer. The same applies for useful "helper functions" like parsing incoming XML messages or getting information about the user device. All these functions should reduce the workload of developers and help them to concentrate on the core tasks of module

[8] http://jquery.com/

development.

Besides the above mentioned functionality, the loader also carries out some layout related tasks. This is necessary for achieving a proper presentation on mobile devices. We were not able achieve this outcome by just using CSS. (see next chapter)

3.3.3 Layout

Based on the fact that the NetLuke client is a web application we have to distinguish between two major types of client devices:

- Desktop Computers: These devices are usually equipped with a large screen (21" or above) that is capable of running at a high resolution like 1650x1080 or 1900x200. The standard input devices are mouse & keyboard. Most modern PCs are equipped with a powerful hardware and therefore perform very well in modern JavaScript applications.
- Mobile devices: This category enfolds devices like mobile phones or tablet computers. While the resolution of this devices varies from 640x480 (SVGA) to 2560 x 1440 Pixels their typical screen size lays within 4" to about 10". The usual input is touch based (like a finger tip or a gesture).

For the first version of NetLuke we created two layouts of the User Interface; one for each device category.

The Desktop Layout

When NetLuke is accessed using a desktop computer the user gets presented an UI that is separated into three major parts. On the top of the page the "algorithm" menu is positioned. Amongst other options (help-function, about-section, etc.) this menu includes a "*Algorithm Control*" that is used for specific algorithm configuration and varies from module to module. The drawing area (stage), that spans over the majority of the screen, is positioned directly below the algorithm menu. On this stage the algorithm visualization is presented to the user. On the bottom of the page another menu bar is located. This menu comprises buttons for stage control as well as for basic algorithm control (play,next,back). Unnecessary control elements are hidden automatically. Figure 3.3 illustrates the Desktop Layout of the first NetLuke while running an instance of the *"InsertionSort"* algorithm. A more detailed description of the layout including an explanation of every menu element can be found on page 85 of the first part of this thesis [13].

The Mobile Layout

As a modern web application we wanted NetLuke to have support for mobile devices, as more and more people use smartphone and tablets. However, when our decision of supporting mobile devices had been made, we began to aks ourselves two questions: (1) How do we define and identify a mobile device? (2) How do we manage to support such a variety of different screen resolutions?

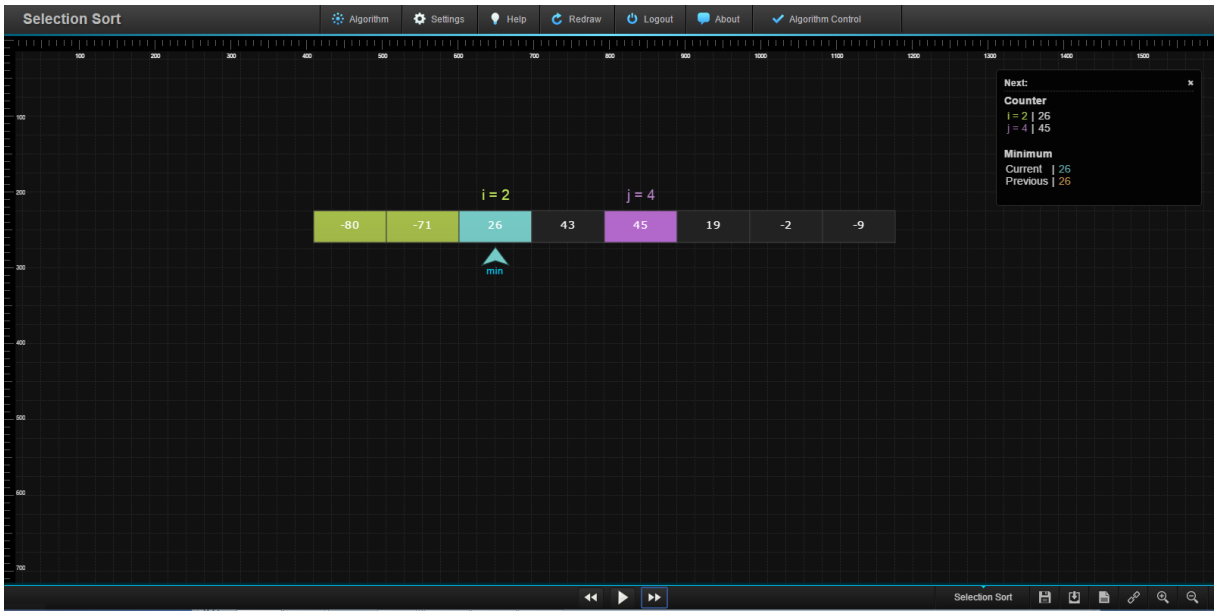

Figure 3.3: The Desktop GUI of NetLuke Version 1

From a developer's point of view the first question is not really easy to answer. After some discussion we decided that a connecting device has to fulfill two requirements in order to be categorized as "mobile ". (1) The device needs to support touch events and (2) the user agent[9] of the device needs to indicate towards a mobile operating system (IOS, Android, Symbian etc).

Now that our first problem was solved we still had to figure out how to handle the different screen resolutions. Luckily every modern mobile device sends the so called "Device Pixel Ratio"[10] along with the HTTP request header. This value allows web developers to design layouts that look (more ore less) the same on all mobile devices, independent from their screen size. With that in mind we created a layout for mobile devices (**mobile.css**) that is being loaded whenever our defined criteria are matched for a connecting device. It was our intention to create a design very similar to the one that is used on Desktop computers.

There is one major difference in these two layouts however. We discovered that small screens (like screens of smartphones) simple do not provide (physical) space to display both menu bars. This problem is independent of the provided resolution. When showing both menus on a smartphone display the buttons within them would be so tiny that an accurate control would simple be impossible. As we do not distinguish between smartphones and tablets we removed the upper menu bar from the mobile layout. This decision in turn changed the way of how mobile users access the NetLuke platform. Using the mobile browser is not possible hence NetLuke requires both menu bars in order to be controlled. Due to this reason we created the "NetLuke App" for iOS and Android. Both apps operate on the same principle: They are based on the **WebView** component of the particular operating system. **Webview** components allow to embed web content into a native application. The WebView is used to navigate to the **index.jsp** on which the

[9] "*The User- Agent header is a text string indicating the web browser which sent the request.*"[37]

[10] "*Device Pixel Ratio (dpr), is the ratio between physical pixels and density independent pixels on the device*"[16]

mobile layout is applied to. We then used a native menu overlay and attached it with our JavaScript functionality of the missing top menu bar. The login credentials are stored in the native part of the app. This makes the **index.jsp** page the only page visible in the mobile app. With this approach the NetLuke modules can be controlled using native menu fragments. Figure 3.4, that is taken from the first part of the thesis, illustrates the mobile layout with an open menu on an Android 2.3.7 device [13, p.87].

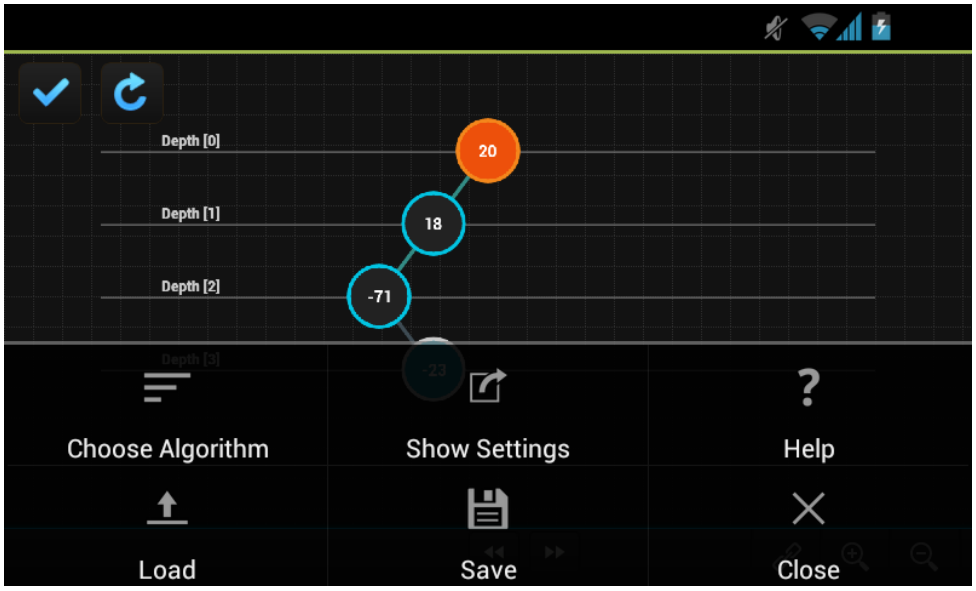

Figure 3.4: The Mobile Layout on Android with open Menu

3.4 KineticJS / Konva

Next to **DWR** the **KineticJS** framework can be seen as one of the technologies that enabled us to develop NetLuke in the first place. But what does it do and why is it so crucial for the project?
The graphical representation of algorithms on a web platform was quite challenging in the beginning. We needed a concept that enabled us to visualize various algorithms on the drawing area. We soon figured out that our initial idea of using pre-rendered graphics (bitmaps, or SVGs) is quite ineffective. Due to the tremendous amount of single graphics necessary in order to represent all possible algorithms we discarded this idea. At about the same time we discovered a really interesting feature of HTML5 that was supported by all major browsers as well as on mobile phones: the HTML5 **canvas**. "*The canvas element provides scripts with a resolution-dependent bitmap canvas, which can be used for rendering graphs, game graphics, art, or other visual images on the fly.*" That is how the canvas is described by the "*World Wide Web Consortium*" [30]. Or to describe it in a different way: The **canvas** provides a drawable region inside a webpage that allows developers to create graphics and animations with the usage of JavaScript. As the canvas matched all our requirements we decided to use it within NetLuke. However, we soon found out that **canvas** in its pure form is somehow insufficient to display complex algorithm visualizations. This is due to three main reasons:

1. The **canvas** only supports one geometric form; a simple rectangle. All other geometric forms have to be created by the developer by combining different paths

(points connected by lines) and rectangles. This results in an enormous effort when creating the foundation for a tree or a graph algorithm.

2. The animation system is quite complex. When implementing an animation the developer has to take care of every step: from detecting the actual visible elements til worrying about the timing.

3. The canvas inherently supports no layering architecture. This makes it quite complicated to organize more complex visualizations as the developer needs to implement his own structure for organizing the elements that are part of the visualization.

Although it seemed possible to implement the visualization by just using the pure **canvas** object combined with JavaScript, we looked for a better solution. And we soon found one: KineticJS.

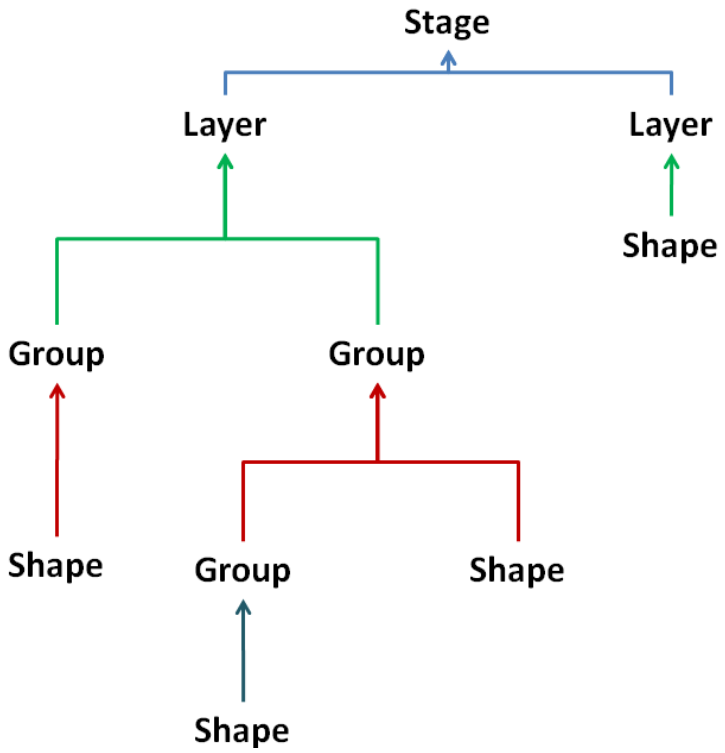

Figure 3.5: KineticJS object organization, taken from [24]

The author of the library, Eric Rowell, describes it as "*[. . .]an HTML5 Canvas Java-Script framework that extends the 2d context by enabling canvas interactivity for desktop and mobile applications.*"[31] The KineticJS library simplifies the usage of the HTML5 **canvas** by providing an object oriented API. It comes with a bunch of predefined geometric shapes like circles, triangles and text-elements. Every object is draggable and can be scaled, rotated or manipulated in any other way. The system also includes an animation API that reliefs developers from tasks like manually redrawing the stage after every animation step. The library also supports event listeners that can be attached to generated objects. But what makes KineticJS really powerful is the way it allows developers to organize the created objects. KineticJs creates a basic drawing stage. On this stage the developer can create various layers. Several shapes in turn can be put together in one group that is

drawn on a layer. It is also possible to draw shapes that are not member of a group, on a layer . Listing 3.5, that is taken from [24] and has been modified, illustrates the object organization within KineticJS.

The way the library organizes objects, the object oriented API and the possibility to easily create animations enabled us to create the complex visualizations within NetLuke. In December 2014 Eric Rowell stopped maintaining the KineticJS project. It has then been forked by Anton Lavrenov who renamed the project to **Konva**. As Konva is actively maintained all future releases of NetLuke (including NetLuke 2) will therefore use the Konva library.

3.5 Deficiencies Of NetLuke

Like every (or at least the most) other software project, NetLuke is suffering from some design flaws, bugs and other inconsistencies that may affect the user experience. Some deficiencies are the result of well calculated decisions while other happened due to the lack of better solutions at those time. Some flaws are just software bugs of course, design mistakes we did not notice during the development phase. The following section aims to describe the most important deficiencies that motivated us to develop an enhanced version of the NetLuke system.

3.5.1 Shortcomings concerning the layout

Although we both think that we managed to create a visual appealing GUI, the overall layout (presentation & controlling) is suffering from some issues. One problem lays within the differentiation between Desktop and mobile devices. While the GUI for PCs is really good and results in fun when using the system, the mobile layout is lacking a comparable user experience. This is the result of of our design decision. As we were not able to create a design that is fully functional on both device categories (fully responsive webdesign) we had to design two different layouts; a mobile and a desktop layout. The mobile layout is missing the upper menu bar as smartphone displays are not providing enough physical space to enable an accurate control when the full layout is shown. Due to the fact that we do not distinguish between tablet computers and smartphones both device categories need to access NetLuke via the NetLuke app. This might not have been necessary on tablets as there screen may provide enough space to display the full layout. As the app does not provide any possibility to register to NetLuke, the user requires a desktop computer in the first place. Another problem comes with the differences of the layouts. As iOS and Android are quite different concerning the way developers need to implements the menu structure for their apps, the two app version provide a quite varying user experience. Consider a user owning an Android smartphone, an iPad and a Windows desktop PC. In this worst case the user is confronted with three different layouts of the same application. Although the GUI is not that complex it may be enough to confuse a potential user who quickly wants to revisit his last algorithm instance on his tablet that initially has been created using the desktop GUI.

3.5.2 Deficiencies for Module Developers

Algorithm development for LUCAS, the predecessor of NetLuke, was quite difficult as this tool was missing an overall concept that defines the extensibility. As we wanted NetLuke to be attractive for developers it was our intention to design a good concept that provides an easy way to implement new algorithms for the system. We thought it would be a good idea to separate each single algorithm in a so called module. Every module should be based on provided libraries that offer basic functionality in order to reduce the complexity and to prevent redundant source code.(as described in chapter 5) Although our module concept works quite well we have to admit that there are some shortcomings as well, as the following enumeration illustrates.

Missing Library Functionality

Every algorithm within NetLuke consists of a client and a server part. Every part of the module in turn has to be based on a provided library that offers basic functionality; The server side, written in Java, has to be based on **netluke-core**, while the client side of the module, written in JavaScript, has to include the client side framework. Designing a working library for the client side was quite more challenging due to reasons already described. Additionally we faced the problem that the client side framework needed to be far more complex than it's counterpart on the server side in order to properly fulfill all envisaged tasks. In the last section I stated out that all UI elements are provided by the system.

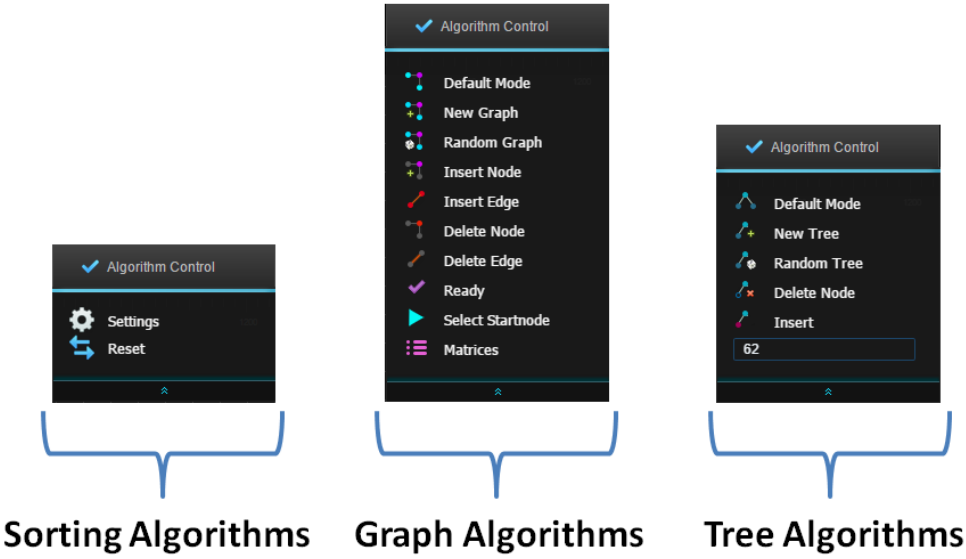

Figure 3.6: Algorithm Specific Menus

Although this statement is correct it can be misinterpreted. We decided that algorithm specific functionality has to be provided by the developer inside the client part of the algorithm. Due to the amount of elements necessary in order to provide control elements for every specific algorithm, we discarded our initial plan that these elements should be provided by the library as well. The result is that the system provides the space and handles the correct positioning, but the developer is responsible for the functionality provided in the algorithm specific menu. This applies for the "array creation wizard" when

using sorting algorithms, or the tree menu entry when using a tree algorithm. Figure 3.6 illustrates the different algorithm specific menus. This means that we were not able to strictly decouple the algorithm logic from the control mechanics, as the developer needs to write some control functionality besides the same code responsible for the algorithm functionality.

Code Redundancy

When the server side library and the client side framework are being compared one may notice another major difference regarding the provided functionality. Every server side implementation of an algorithm module has to be based on our provided library (**netluke-core**). The purpose of this concept is to offer basic functionality that is required for the algorithm development. (This will be further explained in chapter 5) This approach comes with a bunch of advantages for the module developer. As basic data structures like arrays or nodes, most common algorithm are based on, are already defined, the developer is not required to implement them every time he starts developing a new module. This means that algorithm implementations of the same kind (all sorting algorithms, all graph algorithms, etc.) share the same basic ground structure. Or to express it in a more precise way: All sorting algorithms are based on the very same array definition,all graph algorithms depend on the very same element definition of "graph-node" or "graph-edge". (all within **netluke-core**). This principle of "define once - use everywhere" effectively prevents code redundancy on the server side. Within the module implementations only additional features that are not provided by the library need to be defined.

On the client side however, the situation is slightly different. The visualization within the client side is based on the KineticJS library. (see chapters 3.3, 3.4 and 6) Although the client side of the module is interconnected with the provided framework, that offers UI elements and other functionality, it is the developers task to design and create the actual visualization of the algorithm. This implies that all graphical elements that are necessary for the visual representation of a particular algorithm need to be defined in the specific algorithm module. (in the same ***.js** file) This design concept comes in hand with a major drawback: As it is not possible to use functionality defined in other algorithms, the developer has to define graphical elements for a particular module, that has already been implemented for another algorithm. A good example for this problematic nature are the sorting algorithms that come with the NetLuke prototype. (BubbleSort & SelectionSort) Although these algorithms are quite different regarding their working principle, a graphical representation for both modules could be based one the same graphical base structures: A graphical representation of the array, including a textual representation of the values, two counters and a basic animation that shows the swap of a value pair inside the array. The concept of NetLuke however results in the need of those basic graphical elements to be defined twice: one time for every sorting algorithm module. Or to express it in a more general way: As it is not possible to reuse graphical functionality within another module, the same definition of graphical elements may appear in various algorithms. This in turn leads to a lot of code redundancy. This again results in two problems: (1) The source code of algorithms is unnecessarily bloated, which makes it more difficult to understand for other developers. (2) Various graphical elements, necessary for the algorithm visualization may be designed multiple times. Every implementation of these building blocks is different. This makes it complicated for other developers to find the

“best solution“.

Complexity

Another problem may be caused by the complexity of the system. Potential module developers need to be able to understand and write two different programming languages: Java and JavaScript. In order to fully understand the system we also recommend to have some experience concerning web development and client-server architectures. The most challenging part for new developers however, is to understand the concept of building two parts of an algorithm module that need to communicate with each other for our concept to work. Although we tried to ease the development process by providing helping functionality like changesets (see section 5.2.3) or automatic xml-message encoding on the client side, the module communication part stays quite challenging. This is mostly caused by missing debugging functionality on the client side that leads to developers searching the “JavaScript console“ for the reason of the detected misbehaviour. Although there are some really good tools available in order to debug JavaScript code within the webbroswer (like Firebug[11] or the JavaScript console that comes with the Google Chrome browser) we were confronted with a bunch of situations were the error was at a quite different location as expected. From our point of view, JavaScript is still more complicated to debug than “non-script“ programming languages like Java or C++. This behaviour combined with the complexity of our system makes it quite challenging to develop for NetLuke, especially for inexperienced developers.

[11] http://getfirebug.com/whatisfirebug

Chapter 4

NetLuke Version 2

In this section the enhanced NetLuke system will be discussed. Based on the uncovered design flaws, described in the last chapter, we decided to create an improved version of our AV system. Version 2 of NetLuke enfolds a complete new user interface that is presented in Section 4.1. The next section is all about our new JavaScript library. The major aim of this library was simplify the module development process within NetLuke. It can be seen as an advancement of the *loader.js* and is presented in section 4.2. Section 4.3 is all about the *"Logger"*. The "Logger" is a little tool that helps developers to better analyze the performed tasks on the client side, without the need of an external JavaScript debugger.

4.1 The New User Interface

One of the first decisions we made during the design phase of the new NetLuke was to create a new User Interface (UI). Although the old UI was not bad regarding the "look & feel" it was a big downside that mobile users are required to use the "NetLuke App" to connect to the system. We also disliked the concept of different layouts for desktop and mobile devices. (See section 3.5.1 for further details) Therefore we needed to create a new UI that could not only be displayed on all devices but also provides the same user experience on every device category.

Our idea was to create a fully responsive web application. That means that the website is able to adjust the design on its own depending on the users device. This concept is called responsiveness. These design changes however should not break the original "control flow". As a self creation of such a concept seemed hardly be manageable, we started to look for a framework/technology that would satisfy our needs. And once a again we found a suitable solution: The **ionic** framework comes with all the features we need.

"Ionic is a powerful HTML5 native app development framework that helps you build native-feeling mobile apps all with web technologies like HTML, CSS, and Javascript. ". That is how the framework is described on the project's website [7]. The tool itself is a framework that uses a bunch a JavaScript files, CSS components and custom icons in order to provide a full responsive layout for developers. Although it is actually made for building mobile web application it is also suitable for our demands. **Ionic** itself is based on the popular *Angular.JS* library that is now maintained by Google. *Angular.JS* introduces the "Model-

View-Controller“ pattern into the JavaScript world. According to Adam Freeman, author of the book “Pro AngularJS“, this concept allows developers to create complex applications faster and easier. [11]

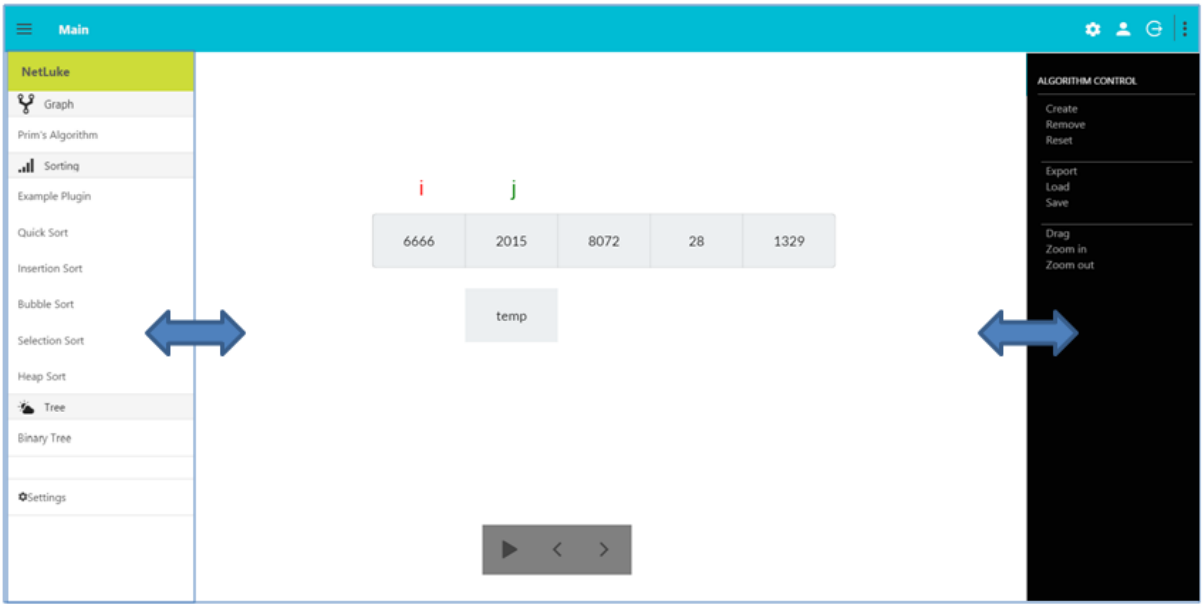

Figure 4.1: New UI IN NetLuke2

Using the **ionic** framework we managed to design a complete new UI that is fully responsive. The new UI is shown in figure 4.1. It features a brighter theme than the original NetLuke. By default all side menus are hidden. This concept allows us to use the full space for the algorithm visualization. The only visibile UI components in this state are the upper menu bar and the algorithm control menu. On desktop devices the new UI supports mouse and keyboard control while on mobile devices the user can control the system using touch based input. The left menu panel, that slides into the screen when the left menu icon on the upper menu bar is clicked, features the algorithm selection menu. After the algorithm selection is finished the menu disappears automatically. The right menu panel contains algorithm specific options like “create“, “reset“ or “remove“. While some menu entries are defined by the module developer (see section 6.3 for further information) other entries are defined by the system. These predefined entries are “load“, “save“ and various options to manipulate the stage. The two menu panels are also shown in figure 4.1. The upper menu bar contains entries for accessing the global settings, for entering the user customization menu and for logging out of NetLuke.

The responsiveness of the new UI is illustrated in figure 4.2. The right side of the figure shows the new UI on a desktop device whereas the left side is taken from a smartphone running in landscape mode. The smartphones is an emulated Nexus 4 device running a physical resolution of 1280x720 pixels and providing a CSS pixel ratio of 2. Here one can see that the user is presented the same layout although some components vary in size in order to increase the usability. In contrary to the old layout that comes with previous versions of NetLuke, all UI elements are shown when using a mobile browser to connect with NetLuke. The usage of the “NetLuke“ app is no longer required on mobile devices.

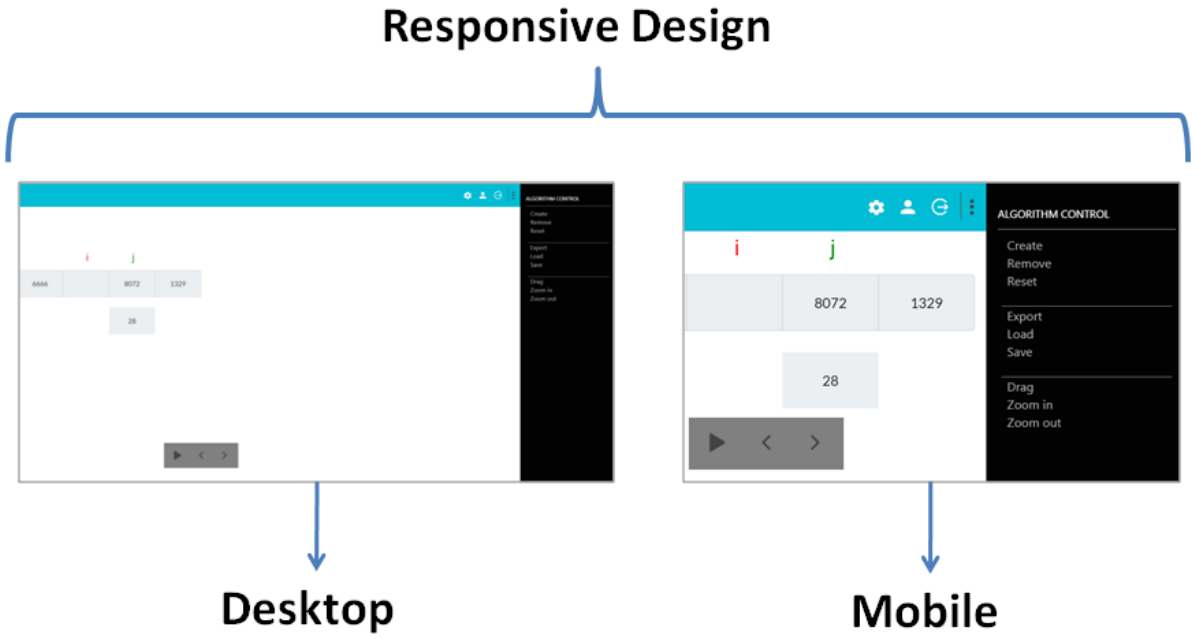

Figure 4.2: NetLuke2: Repsonsive UI

4.2 The NetLuke Libraray

Another disadvantage in the original NetLuke prototype emerges from the client development concept. In contrary to the concept for developing modules for the server side, we do not provide basic implementation structures the developer can build his implementation upon. This is in so far problematic as graphical representation of algorithms of the same category lead to code redundancy as explained in section 3.5.2. That is why we decided to implement a whole new library for the client side that provides graphical building blocks for future algorithm implementations.

The heart of the new library is the so called **NL.js** component. It is the successor of the **loader.js** used in the old NetLuke system. It generates a new namespace on the Java Side, called *"NL"* that is used for all module implementation and library components. It provides the same functionality as the old framework and has been extended by a lot of useful features. The new library now provides a new way of calling server side functions that is based on a configuration object. As remote procedure calls are now wrapped by the library we support better error handling and an extended message parsing as discussed in the next chapter. The **NL.js** library is highly interconnected with the new user interface. As our development concept requires all algorithm implementations to be based on the new library, this interconnection allows some real amazing things: Developers can now use predefined functions to define the menu entries, that are being shown on the UI when the specific module is loaded. It is no longer required to define algorithmic specific menu entries within the source file of the algorithm. Another feature that comes with the new library is the "Creation Wizard". This tool is directly embedded in the new UI and allows the creation of new algorithm instances of any kind. The developer just needs to import the necessary part of the library and provide the name and type (sorting algorithm, tree algorithm, etc.) of his implementation. The "Creation Wizard" is then automatically "generated" and included in the algorithm menu. This component

then provides an interactive menu to generate the necessary values for the algorithm construction and afterwards calls the constructor on the Java side. This reduces the complexity and the lines of code of the JavaScript file representing the client side of the algorithm module.

The most important feature of the new library however is the provison of graphical building blocks for the client side module implementation. Like **netluke-core** we wanted to create a repository of fundamental objects that are necessary for the algorithm visualization. Creating a working concept however was not as easy. The library needed to be "complete". This means that all objects provided by the library needed to be suitable for all potential algorithm implementations. As developers should not modify or extend library functions within their algorithm implementations we needed to cover all potential cases.

Like in **netluke-core**, the library of the server side, we decided to create small building blocks that can then be used by developers to create the algorithm visualization. Every graphical element is based on the **Konva** framework. All elements provide properties and methods. The included methods are used in the algorithm implementation to manipulate the component and to access their properties. The library also takes care of the animation of the various components. The following section describes the components that are necessary for creating a visualization for a sorting algorithm.

- NL.Element.NLCell

This is the most basic graphical structure when designing a sorting algorithm. It allows the representation of exactly one value. The **NLCell** element uses a rectangle, that is based on the "Konva.rect" element and a text field, that uses the "Konva.text" method to fulfill its tasks. This elements provide operations to:

- set the text that is afterwards displayed in the visual representation.
- get the saved text from the **cell**.
- change the background color of the element. By default the color is grey, which is defined in the **NL.Element.NLStyles** object.
- hide/show the text saved in the **cell**.
- get the "TextObject". This is necessary for some operations that need to manipulate the text field.

- NL.Element.NLArray

The "NLArray" can be seen as a container for "NLCell" objects. The constructor of this function requires an array of values. These values are the basis on which the "Cell" elements are automatically being created. The **NLArray** element can be seen as the major component when designing an algorithm visualization for a sorting algorithm. This elements provides various functions for moving and swapping the "Cells" inside the array. That is the reason why developers need to create an "NL.Array" element when creating a new instance instead of various single "Cells". Some algorithms like the "SelectionSort"

can be visualized without the requirement of any other components. The basic operations of this element are[1]:

- swapping cells inside the array.
- moving a cell to another position inside the array.
- retrieving the information about a cell from the array.
- manipulating the properties of the cell objects stored in the array.

- NL.Element.NLCounter

The **NLCounter** element is used to visualize the counter objects being used in various sorting algorithms. This element is based on the "Konva.text" object and requires an "NLArray" upon creation. The "Counter" object is drawn above the array and requires a name that is displayed. The position of this element refers to the array object it is connected with. That means a counter having the position "0" is positioned above the centre of the first cell object of the array element. It provides methods to change its position, to change its colour and to show or hide the "Counter" element.

-NL.Element.NLTempCell

The **NLTemCell** is an adaption of the **NLCell** element. It is used to store temporary values. This is needed in Sorting Algorithms like the InsertionSort where the "temp" value needs to be saved in order for another value to take its place in the array. Like the "Counter" this element needs to be connected with an "Array" object. This is necessary for the provided functions to work. The "TempCell" is drawn below the "Array" and it's position argument also refers to the array. When the "TempCell" has a position of "0" it is drawn exactly below the first "Cell" of the "Array". The operations provided by this element are:

- Move to a specific position within the array index.
- Get a value from a specific array position.
- Set the saved value to a specific array position.
- Change the stored value with one from the array at a specific position.
- Hide/Show the saved value.

- NL.Element.NLStyles

The **NLStyles** object serves as a utility class. It contains the default values and parameters of all graphical building blocks within the library. All objects require a configuration object when being created. This object contains the definition of the object properties. As developers should not always be required to define the whole configuration object upon element creation, we decided to create the **NLStyles** object. The developers just needs

[1] A complete list of the provided methods can be found on the project site

to declare the properties that are divergent from the default values. All other values are automatically taken from the “Styles“ object and inserted by the library.

With the new concept we achieved a tremendous decrease in the complexity of module development. It was our primary task that developers should concentrate on the visualization and the algorithm functionality instead of designing graphical elements. Our new library effectively prevents code redundancy as all algorithm visualizations are based on the very same building blocks. If however a module is going to be implemented that is currently not supported we highly recommend to extend the library instead of using the old design concept.[2] This way other developers can also use the new included features and a break in the design philosophy is averted.

I want to state that we are also working on the support of Graph and Tree algorithms. At the time when writing this thesis however the implementation of the according elements was not finished. Due to this reason I decided only to present the completed elements in the section above. A JavaScript documentation will be available on the Project's Web Page.

4.3 The NetLuke Logger

With the “Logger“ we implemented a tool that should help module developers to find errors or other design flaws. The initial idea behind this utility is that the communication between the client and the server side is hard to debug.

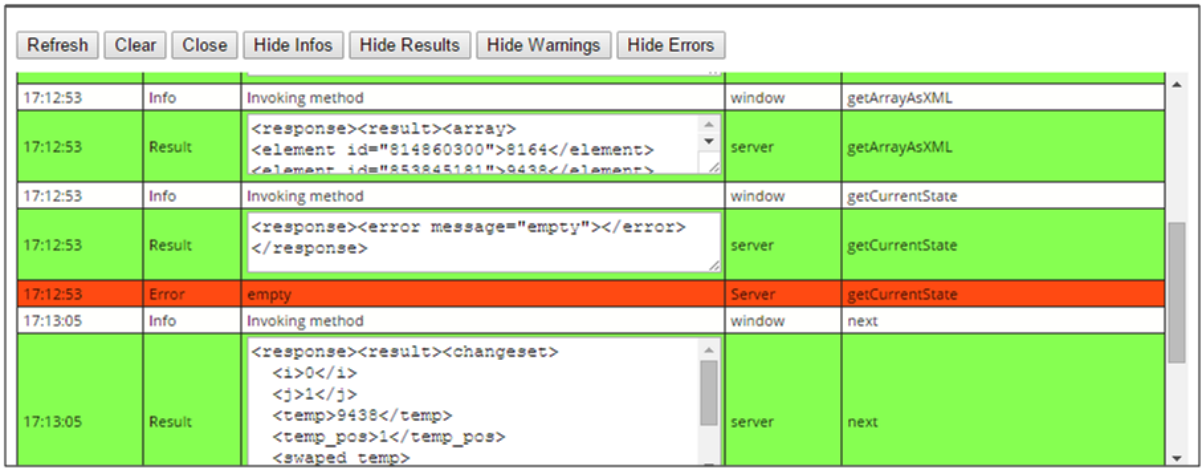

Figure 4.3: NetLuke Logger

The “Logger“ allows the analyze the various function calls and visualizes the XML messages. It is a good way for developers to verify if the server responds with the expected changeset. Errors and other problems are highlighted automatically. This tool is shown inside of a “POP-UP“. Therefore developers may need to disable their PoP-Up-Blocker or add an exception in order to use it. The “Logger“ can be enabled in the personal settings that can be accessed using the top menu bar. It is illustrated in figure 4.3.

[2] An according guide will be provided on the web site of the project

Chapter 5

Algorithm Developement Principles

This chapter focuses on the main principles behind algorithm development for the NetLuke platform. It provides a detailed description of the module design and the developer requirements.

The first section of this chapter (5.1) is about general ideas and concepts and explains in detail what skills are required in order to successfully implement new modules for the platform. The next section 5.2 is all about the NetLuke module concept. In this section the composition of modules within the NetLuke environment as well as the module communication using RPC are explained. This includes the usage of the new features that come with the new library as it changes the process of calling remote Java methods.

5.1 Development: Concepts and Requirements

When designing the NetLuke System we put a lot of effort in making the system easy extensible by other developers. Those developers may ask themselves questions like: How powerful is the system that I am currently working on? How can a specific algorithm be displayed? What can be visualized? This questions can be subsumed under the term "system expressiveness" which is considered as a prerequisite for the educational effectiveness as Prenners points out [13, p. 32]. While future developers need to know how the system is basically working, a deep understanding of the implementation details is certainly not necessary. Because of this reason developers have their own requirements that we payed attention during the design phase.

Before we were able to elaborate our concepts for future developers we had to outline who we expect these future developers to be. Which skills are required and what basic knowledge is necessary? NetLuke is the successor of LUCAS, which has been developed by students of the University of Vienna [32]. Due to this fact we were quite sure that the majority of developers will be students as well.
LUCAS was entirely based on the JAVA programming language and its object oriented paradigm. The logic as well as the visualization was often encapsulated in one single package. In many implementations there is no separation between presentation, control and algorithm core logic.
NetLuke is quite difficult as it uses different technologies for the back end and the front

end. The algorithm logic is implemented in Java, while the visualization is written in JavaScript. This design involves new challenges for developers. In our opinion module development for NetLuke is more difficult than for LUCAS. This is because of two reasons: On the one hand developers need to be able to program JavaScript. On the other hand it is necessary to keep the communication between the back end and the front end in mind when designing an algorithm implementation for NetLuke.

Object oriented programming languages like C++ or Java are often substance of lectures in computer science classes. Many projects at the university are developed using this kind of languages. Therefore many computer science students are capable of writing programs in Java. JavaScript may be an additional challenge for students. Compared with Java, JavaScript however is quite different concerning the syntax and the areas it is commonly used. The main reason for this challenge may indeed be the fact that it is not subject matter of any lectures. Prenner points out, that one can argue about the priorities of the different programming languages in computer science, but the importance of JavaScript has grown in the last 10 years [13]. That is definitely true, when thinking of modern web applications (wep apps) or websites that would not be possible without the use of JavaScript.

Another challenge for developers is the, previous mentioned, communication between the back end and the front end. Although this is not true for every case, it can be said that the algorithm logic (the stepwise execution) is executed on the back end while the visualization takes place on the front end. These two parts of the system have to communicate in order to achieve a synchronized state. We provide basic methods and data structures that help the developers to understand the general structure of an algorithm implementation in NetLuke. Otherwise the developers are not restricted in creating their own designs and implementations within these boundaries. As a result developers have to take care of their own interaction patterns which are necessary for the communication. They determine on their own which messages are being sent between the front end and the back end to keep a synchronized state. These messages either contain data, control commands or both. The sections 6.2 and 6.3 will provide a possible implementation.

Besides giving developers the freedom to design their own implementations we wanted some development issues the developer should not be bothered with. *The following part is taken from the thesis of Georg Prenner [13]*

- *Developers need an easy way of connecting their Java implementation with the HTML frontend through the interfaces/functionality provided by the system environment.*
- *Developers should* ***not be bothered with UI elements*** *and their placement within panels, instead they should be able to concentrate on client side visualization tasks and server side algorithm logic.*
- *Developers should* ***not be concerned about the target platform*** *e.g. mobile or desktop. There is one client implementation artifact for all devices because we expect that module developers won't have the ability to test their implementations equally on all platforms*

- *Developers should be given the ability to test server side implementations extensively and independent from the frontend.*
- *NetLuke, and its libraries, have to provide some sort of debug functionality in order to support module developers in finding semantic and syntactic errors.*

When developing a web application for multiple platforms (desktop and mobile) many technical issues have to be taken care of. For instance the developers have to consider multiple screen resolutions or design layouts for desktop and mobile browsers. UI elements have to be positioned correctly and so on. When developing for NetLuke, developers are not bothered with this kind of development issues. The system takes care of it. NetLuke provides a predefined UI that is suitable for mobile and desktop clients (including various screen resolutions). NetLuke offers a variety of control elements the developers can integrate into their different algorithm visualizations. As the positioning of the control elements is done by the system the developer does not need to take care of it. In section 6.3 an example for adding control elements for a specific algorithm will be provided. The UI itself is not meant to be manipulated by the developers. (Including colors, position of elements or size of items). As the above mentioned points out the developer can focus on his key tasks: to implement the server side algorithm logic and the client side visualization.

The client side visualization takes place in the drawing area. Technically this drawing area is a HTML5 canvas element. This elements uses all the available screen size subtracted by the size necessary to display the UI elements. The calculation of the size is done automatically by the system. Developing the client side can be splitted into subtasks. At the beginning the developer has to decide which control elements he wants to be added to the UI for his specific visualization. The next task is the implementation of the visualization itself. In NetLuke we decided to use the KineticJs JavaScript libray to extend the native HTML5 capabilities as mentioned in section 3.4. Each algorithm visualization on the client side is represented as a JavaScript Object. As Netluke represents an interactive learning tool, these objects (visualizations)) should provide functions like starting/stopping the algorithm execution or going backward/forward step by step. These functions have to be connected with the previously added UI elements. Another very important task is the design of the communication between the back end and the front end. For example a user clicking the “step next “ button on the UI should result in two actions:

- the algorithm execution on the back end should be triggered
- the animation on the front end should display the result of this algorithm execution.

Section 6.3 will provide a detailed example on how to create an algorithm visualization in NetLuke.
To sum things up we wanted to create a developement concept for NetLuke that gives the developers the freedom to create their own visualizations based on their own ideas. Therefore the manipulation of the canvas is not restricted. On the other hand we did not want to bother future developers with issues concerning UI design and Webdesign issues like positioning buttons or designing menus for multiple platforms.

5.1.1 Developer Requirements

In our opinion developing for Netluke can be quite challenging. NetLuke is based on two different programming languages, which are both necessary in order to implement an extension. The client-server communication is based on DWR and includes the exchange of XML or JSON messages. The visualization is based on modern web-technologies like HTML5 combined with certain JavaScript frameworks like KineticJS/Konva. Potential developers should at least understand the principles behind these technologies. The following list represents the miminum requirements a developer should comply with when starting developing for this system.

- basic Java skills as the back end algorithm implementation is written in Java
- advanced JavaScript skills as the algorithm visualization in the front end is implemented in JavaScript
- basic knowledge of HTML5 as the drawing area is based on a HTML5 canvas element
- basic knowlegde of Jquery
- basic knowldge of XML or JSON
- advanced skills in the use of a modern Java developement IDE like eclipse

5.2 The NetLuke Module System

"Modules are the smallest self-contained components within NetLuke" [13]. Each module in NetLuke contains the algorithm logic and the visualization of one single algorithm. So every time a developer wants to extend NetLuke with a new algorithm a new module has to be added to the system. Basically a module consists of two artifacts: A JAR file and a JavaScript file. Figure 5.1 illustrates a module. The JAR contains several components and represents the back-end logic. It is deployed directly in the NetLuke project. The JavaScript file represents the front-end logic and is deployed to the WEB-INF directory within the NetLuke server project. The JavaScript file should be developed after the the implementation of the back-end logic. The artifacts of a module communicate over RPC messages.

Building a module for NetLuke should be done stepwise. In Section 4.1 of his thesis, Georg Prenner provides a list of steps that are necessary to create a new module [13]. Due to some changes in NetLuke2 I will include this list in an adopted form. In general module developement in NetLuke is done by..

1. ... creating a new Java Project [1] and including the *netluke-core.jar* file as the primary application library. *netluke-core* is a library that provides basic data structures and methods that help developers implementing their algorithms in NetLuke. (See section 5.2.1)

[1] We recommend using the Eclipse IDE when developing for NetLuke as the System itselfs has been developed using Eclipse and the devlopemend guide refers to this IDE

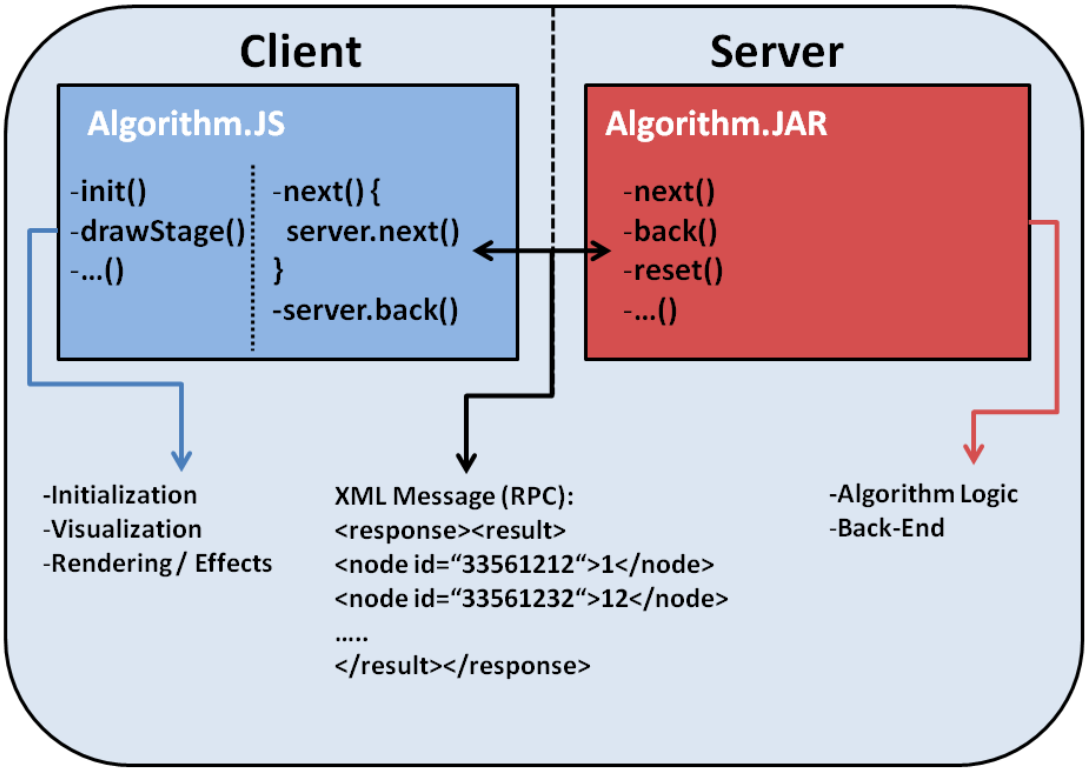

Figure 5.1: Module Overview

2. ... implementing and testing the server-side module. (JAR file). Testing ensures a high level of module robustness and can either be done by using or adopting the test-cases withing the **NetLuke** project. (In the original NetLuke there has been a seperated project for testing purposes called NetLukeTest. The test cases are now integrated within the NetLuke system) We further recommend that a developer tests the implementation manually before the deployment.

3. ... deploying the JAR file and editing the *netluke.conf* configuration file

4. ... implementing the JavaScript visualization using our new designed library (NL.js)

5. ... integrating visualization and control components into the client system environment (see Chapter 6 of Prenner's thesis [13] for a detailed description)

6. ... ensuring the compatibility with all targeted platforms by manual testing

5.2.1 NetLuke Core

When implementing extensions for LUCAS, the predecessor of NetLuke, as part of our Bachelor thesis we uncovered some major issues. One of the biggest problems in our opinion is, that the tool is suffering from a heavy amount of redundant code. That is because there is no overall concept of abstraction that can lead to the re-use of components throughout different algorithm implementations. Or as Prenner puts it out: "LUCAS was missing comprehensive libraries for integrative algorithm and data structure creation" [13]. For NetLuke we wanted to provide such a library in order to reduce the amount

of redundant code and to give future developers a concept they can base their implementations on. Although every algorithm or data structure is unique and can therefore not be unified, they all work with data. This data may be integer values, characters or strings. Due to the reason that the abstract type of all these mentioned data types is "data" their representation can be unified. Based on that concept we developed the *netlukecore* library. The main goals were inheritance, code re-use, information hiding and an overall reduction of complexity. The library represents the underlying building block of every single algorithm implementation in NetLuke. It also provides a basic implementation structure for developers. With this concept we also wanted the different modules (algorithm implementations) in NetLuke to follow the same structural principles. The following paragraphs give an overview of the library.

As we focused on sorting algorithms, trees and graphs, we implemented only support for these topics into the library. Developers are welcome to extend the library for their own purposes. (For example when implementing hashing). We do not recommend changes of the already existing API as existing implementations are dependent on it.
The library itself is a JAR file that currently consists of the following packages:

- at.ac.univie.netluke.core.components

The classes within this package represent the most fundamental building blocks for module development. They are the abstract representation of data within NetLuke.

- The *Element* class: The Element class is the most basic one. Its purpose is to store one single value. Every Element has a unique ID (generated when the constructor is called) and can wrap it's containing value into the XML or Json representation(containing the value and the elements id). The constructor accepts either a char or an integer as argument.
- The *Node* class: The Node class is abstract. It acts as a base class for the concrete implementations of *GrapheNode* and *TreeNode*.
- The *GraphNode* class: The GraphNode class is derived from *Node*. Unlike *Node* it is not abstract but a concrete implementation. It is especially designed for the usage in graphs and has therefore attributes like name, if it has been visited or if it is acting as a start node. The constructor accepts a string or an integer value. Like the *Element* it can wrap its containing values in XML or JSON format.
- The *TreeNode* class: The TreeNode class is also derived from the *Node* class. It has been designed for the use in tree implementations. It contains an ArrayList of *Elements* allowing it to save more than one *Element* object. It implements functionality to acces parent and child (Tree)nodes. The constructor ecpects either an Element, a list of Elements and a Treenode (representing the parent) or two Treenodes (representing parent and child).
- The *Edge* class: The Edge is a subclass of *Element*. It provides the same functionality as an Element extended with the possibility to save attributes for start- and endnodes. The constructor expects an integer value, representing the value of the edge, and two node elements.
- The *GraphEdge* class: The GraphEdge is a subclass of *Edge*. Especially designed for the usage in graphs it provides a "directed" attribute indicating whether the edge

is directed or not. This is necessary in implementations of various graph algorithms like Depth/Breadth First Search. The constructor expects a GraphNode element as argument.

- at.ac.univie.netluke.core.datastructure

Classes withing this package represent useful data structures, developers can include in their algorithm implementations. The *DataStructureMethods* interface is designed to provide a basic range of functionality every data structure implementation should provide. It can be stated that the classifcation according to the names of the packages is not mandatory. This means that classes provided in the *at.ac.univie.netluke.core.datastructure* package can also be implemented in sort algorithms, when considered useful. This does not apply for the interfaces as every category (*DataStructureMethods, GraphMethods, SortMethods*) provide its own, special, interface. Some of the classes within this package (*DataStructureMethods, Heap, BinaryTree*) are built upon core components.
At this moment the package contains the following classes:

- The *DataStructureMethods* class: This class represents a Java Interface that defines basic functionality every data structure that is implemented within the NetLuke environment should provide.
- The *Heap* class: The *Heap* class is the implementation of a simple Heap. It provides characteristic functions and is the underlying basement of the HeapSort plugin.
- The *BinaryTree* class: The *BinaryTree* is used in the Binary Tree plugin. It is a simple implementation of a binary search tree.
- The *Boundary* class: The *Boundary* can be considered as a simple helper class. Its purpose is to store two Integer values, representing the upper and the lower boundary. The use of this class might be useful when implementing algorithms that split the actual sorting array. (eg. Quicksort) The class also offers the functionality to export the containing data into the XML or Json representation.
- The *Stack* class: This class is the implementation of a Stack. It can be very helpful when implementing algorithms for NetLuke that use recursion like Quicksort. Due to the requirements of NetLuke to implement a stepwise execution and to go back during algorithm execution a recursive implementation is often not possible. The *Stack* class helps the developer to save data that would normally be saved in the native Java Recursion Stack, which is not accessible. The class offers methods to add, to remove and to get an object from the stack. In addition the *Stack* provides a Copy Constructor. This is necessary in order to gain an exact copy of the Stack as "normal" copies of complex structures in Java using the "=" operator only generates references to the original object [1]. That is not what we wanted because the original object is often a subject of change during algorithm execution, for example.

- at.ac.univie.netluke.core.utils

Classes within this package represent utilities that are considered useful when developing for NetLuke. Currently there are two utility classes in NetLuke core (*IDGenerator* and *DeepyCopy*). Developers are welcome to add more useful utilities to this package.

- The *IDGenerator* class: When designing NetLuke we thought it would be very useful for data structure objects like elements, nodes or node-extending data structures (TreeNode, GraphNode) to have IDs which allow an identification of these components. The *IDGenerator* is the result of these considerations. It provides two methods that are statically accessed. Both methods are named "next()" but differ in the arguments they are expecting. The *next(object)* method generates a unique hash of the object given as argument based on the *System.identityHashCode()* method. This method is called automatically in the constructors of the mentioned data structure objects. (So when generated an element already has a unique ID) That ID has to be unique just within an algorithm instance, not within the whole system.
  The other *next()* method requires no argument and returns an ID based on an integer value that is increased everytime the method is being called. This method can be used by developers to add IDs to their objects that contain previous mentioned data structure objects.
- The *DeepCopy* class: This class provides the possibility to make an exact copy of complex objects. Sometimes this can be necessary, for example when you need the exact snapshot of the actual state of a graph. *DeepCoopy* uses serialization in order to gain a complete new object, not just an object holding refrences to the old one. It has to be stated that in some cases (very large/complex objects) the copy operation might fail and result in a "Stack Overflow". That happened to us when we tried to make a clone of a large *Stack* object. In such cases the use of a "Copy Constructor" may help. Intensive testing, even with large amounts of input data, is required when using this method, as it can be a potential source of errors [1].

- at.ac.univie.netluke.core.graph

Currently this package only contains the *GraphMethods* interface. This interface defines functions every graph implementation within the NetLuke environment should provide. It can be seen as a proposal from the NetLuke developers which functions should at least be facilitated even in the most rudimental implementation of a graph algorithm. The *GraphMethods* interface is quite similar to the interfaces for data structures (*DataStructureMethods*) and sorting methods (*SortMethods*). Developers are very welcome to provide additional implementations like re-usable graph implementations for this package.

- at.ac.univie.netluke.core.sort

Classes within this package represent the basement of sorting algorithm implementations within NetLuke. At the time of writing this thesis the package includes the *SortMethods* interface and the *SortAlgorithm* class.

- The *SortMethods* class: As mentioned above the *SortMethods* interface is quite similar to the interfaces provided in the datastructure or in the graph package. Basic functionality, every implementation of a sorting algorithm should provide, is defined in this interface. Examples for methods in this case would be: "**next()**", "**run()**" and "**back()**". The "next()" method describes a step forward in algorithm execution wheras the "back()" method describes a backward step. The "run() " starts the algorithm execution until the algorithm has finished. (in most cases this

means that the data has been sorted)

- The *SortAlgorithm* class: The **abstract** *SortAlgorithm* class represents a superclass for miscellaneous in-place sorting algorithms. It is build around an *Element* array and provides various functions to manipulate data within this array. In place/in-situ sorting algorithms work within their given data structure. We wanted to integrate the basic functionality for this data manipulation into this superclass. Therefore *SortingAlgorithm* provides methods for value insertion, value swapping (on the array), deleting methods and functions to represent the data saved in the array. As most sorting algorithms are based on such operations, the provided functionality should be a helpful sub-construction for developers when implementing sorting algorithms. Especially needed methods/functionality may have to be added in the sub-class (the actual implementation of the algorithm).

- at.ac.univie.netluke.core.annotation

Methods in the NetLuke system that shall be accessible from the client side (using DWR) need a special annotation in order to get found by the DWR servlet. Developers need to annotate methods and constructors, they want to access from the client, with the *@NetLukeMethod* annotation type. Only using this annotation the (**public**) method/-constructor can interact with the Service Layer of NetLuke. The *NetLukeMethod* class within this package describes this special Java annotation type. The module detection itself is being discussed in a later section.

5.2.2 NetLuke Module Composition

As already mentioned algorithm implementations withing the NetLuke environment are separated into modules, that contain a JAR and a JavaScript file. It has also been stated out that the *netlukecore.jar* library represents the fundamental API for every algorithm implementation and is therefore an integral part of every JAR file. But how exactly are these JAR files, that represent the back end logic of every algorithm, assembled? As every other Java program, a module is separated into packages. Future developers are highly advised to organize their own implementations in the suggested structure presented in this thesis. In general the project is named after the algorithm or data structure[2] itself and contains three packages: The **at.ac.univie.netluke.plugin.algorithm** package contains the main source file, that represents the business logic of a module. The main source file has to be named after the algorithm that is going to be implemented.(**InsertionSort.java**)

The **at.ac.univie.netluke.plugin.insertionsort** package contains additional classes that are needed for providing the full functionality (helper classes). In NetLuke developers need to implement functionality that enables the user to go back and forth in algorithm execution. Implementation of this functionality should be separated from the main business logic. Classes that are needed to provide this functionality should be moved into this package. Figure 5.2 illustrates the buildup of a module based on the InsertionSort algorithm.
Before exporting the project as a JAR file and deploying it to the NetLuke system the

[2]due to the reason that the later sections include a development guide referring to the InsertionSort, all examples will use "InsertionSort" as algorithm name

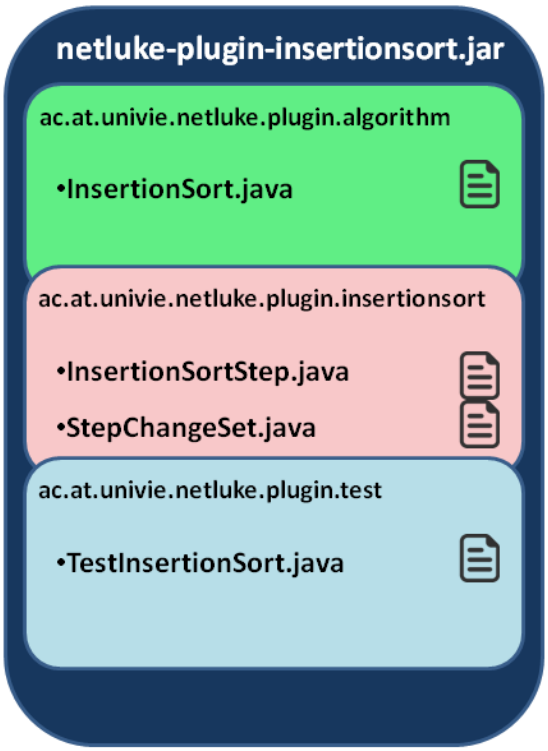

Figure 5.2: Module Structure

written code should be extensively tested. This can be done by writing a simple Java class that calls the methods the implementation provides. This testing class should be moved into the **at.ac.univie.netluke.plugin.test** package. After manual testing the project should be exported as JAR file and included into the NetLuke test project for further testing. A detailed guide on development and testing the back end logic will be given in section 6.2.
The client side (the JavaScript part) consists of just one JavaScript file that has to to deployed to the **WebContent/app/algo** folder into the NetLuke server project. A detailed guide on how to implement the client side will be given in section 6.3.

5.2.3 NetLuke Module Communication

Due to our decision to create a concept that is based on a server side algorithm execution and a client side visualization we needed to find a communication technique. (On the one hand the client has to know what happens on the server side in order to display the results of this execution. On the other hand the client has to tell the server which exact execution to start after an user input has occurred) As the JavaScript side is not aware of the processes happening on the server side, including objects and attributes, we created a message format that is able to [...] " *map object presentation on to business logic and vice versa* ", as Prenner puts it out [13]. As mentioned in chapter 3, NetLuke uses Remote Procedure Calls (RPC) provided by a DWR servlet as communication technique between client and server. For further information about the underlying server components that realize RPC functionality I refer to Section 5.2.1 of the thesis of Georg Prenner [13].
Basically the communication that is necessary to call a function on the server and to receive the results on the client side consists of four steps:

1. A remote function (provided by a Java module and annotated with the *@NetLukeMethod*) is called by the client.

2. The called function is dynamically invoked by the server.

3. The output of the server side execution contains data (client instructions or information blocks) needed for the visualization on the client side. These data is usually embedded within XML or JSON messages and being sent to the client[3].

4. The client receives the server response and the message gets parsed. According to the client side implementation the information included in the server message is being used to generate the visualization.

LISTING 5.1: getArrayAsXML - Method

```
1  private StringBuffer getArrayAsXML(final Element[] array) {
2          final StringBuffer xml = new StringBuffer();
3                  if (array == null) {
4                          return xml.append("<array></array>");
5                  }
6                  xml.append("<array>\n");
7                  for (int i=0; i < array.length;i++) {
8                          if (array[i] != null) {
9                                  xml.append(array[i].getAsXML());
10                         }
11                 }
12         xml.append("\n</array>");
13         return xml;
14         }
```

Although we improved the functionality of NetLuke regarding the usage of JSON as messaging format we still recommend using XML. (This applies especially for beginners) That is due to a simple reason:
One of our goals was to relieve developers from complex message creation. As a result the core library *netlukecore.jar* provides certain functionality that includes the generation of XML messages out of data stored within core components (or implementations built upon core components). When building their implementations upon this core library, developers just need to call this provided methods to generate XML representations of their own data. At the time when writing this thesis not all core components have been redesigned in order to provide the same functionality for the JSON format. An example is the *SortAlgorithm* class within the *at.ac.univie.netluke.core.sort* package. While the XML builder allows to create an XML representation of the initial array as well as of the actual array, the JSON builder only allows to create a representation of the actual array. When using JSON (or self defined messaging formats) missing functionality has to be added by the developer. Because of this reason the following as well as the guide provided in the next sections will focus on the XML format.
But how to get an XML representation of given data within NetLuke? A good example is the **getArrayasXML(Element[ ])** method within the **SortingAlgorithm** class. Every

[3]The NetLuke system is not strictly bound to these message formats. Developers are welcome to use their own messaging format if it better suits their needs in special situations. However we recommend using XML or JSON as the underlying architecture already provides functionality that is designed to wrap data into these formats.

algorithm based on this class provides this method, which either accepts the initial array or the current array as parameter. This method assembles an XML representation of a given array using the **getasXML()** method provided by the **Element** class. Listening 5.1 illustrates the method, while listening 5.2 illustrates the element's **getasXML()** and **setXML()** method.

LISTING 5.2: getAsXML/setXML - Methods

```
1  public String getAsXML() {
2                   if (this.xml == null) {
3                           setXML();
4                   }
5                   return xml.toString();
6           }
7
8  private void setXML() {
9  this.xml = new StringBuffer("<element id=\""+ this.id+"\">"+this.value+"</element>");
10          }
```

The next question that may arise is how do XML messages in NetLuke look like? Prenner describes this quite well in his thesis. I will quote his description as the structures of XML messages have not changed with the new version and therefore his delineation is still valid: " *We chose to follow a simple pattern of message composition very similar to the XML-RPC data type. The basic pattern resembles a [key] = [value] pattern. For instance "<type[attrid]>value</type>".* " [13, p.49] Figure 5.3 illustrates the resulting XML representation in case of the above mentioned **SortingAlgorithm** class when the **getArrayasXML(Element[ ])** method is called. (For this example a little array with the values 5,1,2 has been used)

LISTING 5.3: XML Array representation

```
1  <array>
2  <element id="25411885">5</element>
3  <element id="16743523">1</element>
4  <element id="20940721">2</element>
5  </array>
```

Figure 5.4 shows the resulting JSON string when calling the corresponding **getAsJson()** method.

LISTING 5.4: JSON Array representation

```
1  {"sort_array":[{"value":5,"id":25411885},{"value":1,"id":16743523},{"
       value":2,"id":20940721}],"sorted":false}
```

As stated before the client and the server side have to be kept in a synchronized state. The information needed to achieve this is encapsulated in the XML messages (or any other messaging format) that are being sent from the server. (After every call from the client side) In most cases however it would be not convenient to transmit a full snapshot representing the current state of the data structure to the client. This is due to two main reasons:

1. Always sending a complete representation of the data struture would require a (more or less) complete implementation of the algorithm on the client side to interpret the

data for the visualization. That is not our intention as the client side logic's primary purpose should be visualization tasks. Although, it can sometimes be necessary to implement parts of the algorithms logic on the client side as well. (Especially when implementing very complex algorithms)

2. The amount of data being sent should be kept at a minimum.[4] On the one hand large XML messages need more time to be interpreted (especially on slower devices such as smartphones) while on the other hand it may take more time until they are being transferred (especially when working with a slow connection). Although this reason is not as important as the first one it should not be ignored.

To circumvent this problem we chose to use the concept of **changesets**. Changesets are based on the principle to only contain the information that is necessary to determine "*what has changed*". They can be compared with "incremental backups " where only files that have been changed are being transferred.

Changesets in NetLuke

Changesets in NetLuke just hold the information that is necessary to determine the difference between the current and the previous state in algorithm execution. Changesets depend on the developer's implementation and differ from implementation to implementation. They need to be properly designed in order to contain the information needed for the visualizations tasks on the client side. Listing 5.5 illustrates a simple changesets from the BubbleSort module.

LISTING 5.5: XML Changeset BubbleSort

```
1  <changeset>
2   <step>4</step>
3    <i>1</i>
4    <j>3</j>
5    <swapped>
6          <pos_a>3</pos_a> <pos_b>5</pos_b>
7          <element id="25411885">5</element>
8          <element id="20940721">2</element>
9    </swapped>
10 </changeset>
```

This changesets contains information about the step number, the value of the counter variables and that a swap occurred (including positions and information about the elements that were involved). A corresponding implementation on the client side assumed, the system interprets the changeset and changes the visualization according to this information. Basically it can be said that a full snapshot of the data structure should only be transferred when a new algorithm instance is created, otherwise the use of changesets should be sufficient. Although this is a very simple example the principle should stay

[4] On Page 51 of his thesis, Prenner gives a great example on how much smaller an efficient changeset can be compared to a full snapshot of the datastructure

the same throughout all algorithm implementations:[5] Client implementation should be reduced to message interpretation and visualization [13]. In the next chapter a guide on how to implement changeset functionality will be provided.

Remote Procedure Calls in NetLuke

Whenever an action occurs on the client side (for example a user requesting an algorithm instance to execute or inserting/deleting a value in a data structure) that is by design dependent on a server response, a remote Java method within the specific module is transparently called by the client. Since NetLuke2 the way of calling a remote Java method has changed. The `remote.invokeMethod()` is now part of the new NetLuke JavaScript library (**NL.js**) as mentioned in Section 4.2. Instead of directly accessing this method (with necessary parameters) we now have to create a JavaScript object containing the parameters in it's imbedding variables. The JavaScript object is called `NL.Request()`. Listing 5.7 illustrates the use of the new library wheras listing 5.6 shows the use of the old method.

LISTING 5.6: Remote Procedure Calls in NetLuke 1

```
1 getArraySize :function() {
2         remote.invokeMethod('getArraySize',{
3                 async:false;
4                 callback:function(str){
5                         //do something with result(str)
6                 });
7         });
8 }
```

LISTING 5.7: Remote Procedure Calls in NetLuke 2

```
1 getArraySize :function() {
2         var request = new NL.Request();
3         request.method = 'getArraySize';
4         request.onSuccess = function(xml) {
5                 //do somethewing with result(xml)
6         }
7
8         NL.Connection.send(request);
9 }
```

In line 2 a new variable called *request* is being created. The object type of *request* is *NL.Request()*. The *method* variable contains the name of the remote Java method that has to be called on the server side whereas *onSuccess* gets assigned to an anonymous function that will be called after a successful execution on the server side. The anonymous function expects the server answer as an argument. Within this function the visualization tasks are initialized. With `NL.Connection.send(request);` the library is told to execute the function stored in the *method* variable. The *NL.Request()* object provides more variables

[5] Sometimes it can be necessary to implement parts of the algorithm logic on the client side as well. Although this should be avoided it is better to have more logic on the client side than to work with changesets that are so complex that they are not debuggable any more or lead to unexpected errors

which are necessary when calling a constructor for example. Listing 5.8 illustrates invoking a constructor when using the new library. The *invoke* variable, which is normally set to "method " needs to be set to "constructor " as shown in line 3. The *parameters* variable itself contains two variables: *data* which holds the actual values the constructor is expecting and *type* which defines the data type of the values stored within *data*. The *data* variable can be set to "int", "double", "boolean", "char" or "String".

LISTING 5.8: Remote Constructor Call in NetLuke 2

```
1  .....
2  var request = new NL.Request();
3          request.invoke = 'constructor';
4          request.method = 'InsertionSort';
5          request.parameters = {
6                          data :ARR_VALUES,
7                          type :'int'
8          };
9          request.onSuccess = function(response) {
10                                 InsertionSort.createArray();
11         };
12         NL.Connection.send(request);
```

A question that may arise at this point is: Why did we introduce this new method of calling remote Java methods? One can argue that the new approach is more complicated, requires more lines of code and the use of a new (needless) JavaScript Object. There are two main reasons for this design change:

1. Better **Error Handling**: Debugging JavaScript code can sometimes be really challenging, especially when it comes to finding errors within anonymous functions. Sometimes an error occurs and even the "Java Script " console or modern Web Tools integrated into modern web browser won't be able to tell the developer the location of the error. Since we wanted to prevent future developers from endless "debugging" sessions we introduced this new method. Regarding remote method calls the new library can be seen as an additional "debugging " layer: It wraps the calls from the client before calling the original `remote.invokeMethod()` function. This enabled us to analyze the method call before actually sending it. Eventually occurring errors are being revealed by the library and the developer is informed by an appropriate error message. For example if the someone tries to call a constructor with "**Sting** " instead of "**String**" he receives an error saying: "*Datatype "Sting" unknown, cannot send request!*" The library also analyzes the server response and informs the developer if the server answer contains unexpected content. For example a Java method returning malformed XML will result in an error saying: "*Malformed XML response: "*

2. Improved **Readability**: The other reason for this design change is that we wanted the JavaScript code to become better readable. Although the original method would have required less lines of code there is one big disadvantage: The initial method definition is very nested as it requires the implementation of the callback function within the *remote.invokeMethod()* function body. The new approach on the contrary

enables the developer to define the callback function afterwards (or whenever whe wants). The separation of the original parameters into variables leads to a more structured code. For example the new code structure enables developers to find out far more quickly which bracet belongs to which function.

It has to be stated that the library uses the old functionality in order to call remote java methods. The use of the new method is obligated as direct access to the old functionality is not allowed in NetLuke 2. For a better understanding of the RPC principles within NetLuke I refer to chapter 5 of Georg Prenners thesis [13].

Chapter 6

Algorithm Developement in NetLuke

6.1 Requirements & Preliminaries

To be able to implement new algorithms for NetLuke developers must at first fulfill the technical requirements (1). In the second step the Eclipse IDE is configured and the NetLuke Project is imported (2). Before starting with the actual implementation the developer has to make a concept of the future design. This includes a detailed analysis of the algorithm that is going to be implemented.(3)

(1) Technical Requirements

In order to implement new modules for the NetLuke environment, developers need to have the following components installed on their development workstations:

- **Java JDK:** NetLuke's core logic as well as the module server side logic is written in Java. Therefore the current JAVA JDK (Java Development Kit) is necessary for development. When writing this thesis the current Java version was 1.7.[1] The Java JDK is available in two version: 32 and 64 bit. If developers want to use the 64 bit version they have to make sure to be working on a 64 bit platform and to have a 64 bit version of the IDE (Eclipse) installed on their PC.
- **Java IDE:** We recommend using the "ECLIPSE" IDE [2] as a development tool as we used it as well when creating NetLuke. In addition the development guide later in this chapter will have many references to this tool. We suggest using "**Eclipse IDE for Java EE Developers** " as it comes with many useful tools that are necessary when creating a web project. In Eclipse developers should install certain extensions like "GIT" from the included marketplace in order to download the NetLuke project from the server of the university.
- **Application Server:** As NetLuke is a Web Project it requires an underlying application server in order to be executed. When starting with the development we chose to use the **Apache Tomcat** [3] server. The version we used when writing this

[1] The current JAVA JDK can be found at "https://www.oracle.com/java/index.html"
[2] The current version of Eclipse can be found at "https://www.eclipse.org/downloads/"
[3] The current version of Apache Tomcat can be found at "http://tomcat.apache.org/"

thesis was **Tomcat 7.0.53**, but any version from the 7th generation should work. Working with applications server in *Eclipse* is quite easy as the IDE provides functionality to integrate server runtimes into the project environment. NetLuke requires some libraries that have to be placed inside the **lib** folder of the Tomcat installation directory. These libraries are necessary to provide functionality like database communication or security functions. These library files are available on the GIT directory[4] of the NetLuke project. Table 6.1 illustrates the necessary libraries.

Library File	**Function**
jaxb-api.jar	Providing a Java API for XML Webservices
jaxb-impl.jar	
jaxb-xjc.jar	
jaxws-api.jar	
jaxws-rt.jar	
jaxws-tools.jar	
stax-ex.jar	
streambuffer.jar	
mysql-connector.jar	Provide functionality to connect to a MYSQL database
hsqldb.jar	Provide functionality to connect to a HSQL Database

Table 6.1: Tomcat Library Files

(2) Setting up the Development Environment

After extracting/installing the required components mentioned in (1) the first thing to do is to add the *Apache Tomcat 7 Server runtime* to the **Eclipse** workspace. This is simply done by clicking `File -> New -> Other)` in the upper toolbar of the IDE. In the subsequently showing wizard the `(Server)` tab has to be selected. After clicking next the developer has to choose `Tomcat v7.0 Server` from the list and add a new **server runtime environment**. In the last step the location of the server runtime has to be set to the directory where the downloaded tomcat installation (with the already included library files) is saved.
The next step consists of importing the **NetLuke** and the **NetLukeCore** project. The **NetLukeCore** project is necessary to generate the **netluke-cor.jar** file. The easiest way to import these projects is using the GIT repository. In his book, *"Versionskontrolle mit Git"*, Jon Loeliger explains the usage of the GIT repository very detailed and quite understandable [26]. Summarizing a local clone of the GIT repository has to be created from which the two projects can be imported. The Eclipse marketplace provides miscellaneous plugins that allow to connect to a repository and import the projects from within the IDE itself.
After the import has finished there should be three projects in the *Project Explorer* window showing up: **NetLuke**, **NetLukeCore** and **Servers**. If that is the case the next step consists of verifying the correct setup of the development environment. In order

[4]The GIT directory can be accessed using: "http://donatello.wst.univie.ac.at/repositories/145/Data/"

to do this the correct execution of the **NetLuke** project on the local application server instance has to be controlled. To execute the **NetLuke** project the developer has to perform a right click on the according project and select `Run As -> Run on Server`. In the following selection window the local tomcat instance has to be selected. If all libraries are correctly included, as mentioned above, the server should start without any errors and display the welcome page of **NetLuke** in the built in Web browser of Eclipse[5]. In order to perform a test of the database[6] functionality a new user account within NetLuke should be generated and an initial login with this user should be performed. If no errors occur during these steps the development system is considered to be ready.

(3) Algorithm Analyzation

Before actually starting with the implementation of a new module we consider it quite useful to perform an detailed analysis of the algorithm. The reason for this is, that developers should have a detailed concept of their future implementation. When beginning with the actual coding a developer should be able to answer following (exemplary)[7] questions:

- How does the algorithm really work? Which state has to be achieved in order to say the algorithm has finished?
- What do I want do happen when a User performs a "next" step?
- What do I want do happen when a step backward is triggered?
- How should the algorithm visualization look like? Which elements are necessray? What visual effects do I want do implement?
- How should the changeset be built?
- How do I construct the Client - Server communication?
- How much logic do I want do implement on the client side?

In this thesis, I decided to provide a development guide for NetLuke on the basis of the **InsertionSort** sorting algorithm.

6.1.1 The InsertionSort

"Insertion Sort is the method most card players use to sort their cards. They keep the cards dealt so far in sorted order, and as each new card arrives they insert it into its proper relative position. To sort the array x[n] into increasing order, start with the first element as the sorted subarray x[0..0] and then insert the elements x[1], . . . , x[n-1] " [. . .] [4] That is the way how Jon Lois Bentley, author of the book " Programming Pearls ", describes this sorting algorithm. Following this description the principle behind **InsertionSort** can be described as follows: The algorithm sorts a given array using insertion. The left side of the array (beginning with J=1) is considered to be already sorted. When "J" (representing the index of the values inside the array) now gets increased the element on its position

[5]The standard port of a new Tomcat instance is 8080. This can be changed in the "server.xml" located in the server project

[6]NetLuke comes with its own database which is generated during the first startup

[7]This example just aims to give developers an impression of what to take care of during this process. It is not meant to describe the whole scope of the design phase

gets placed on its (temporally) position on the left side of the array. During this step all elements that are larger than the current element are moved one position to the right inside the (sorted) array. When "J" has reached the end of the array all elements should be at their correct position and the array is sorted. Listing 6.1 illustrates the algorithm using pseudocode.

LISTING 6.1: InsertionSort - Pseudocode

```
1  InsertionSort(Array[]):=
2          {
3           For j:=2 to n
4                  Temp:=Array[j]
5
6           i:=j-1;
7                  while(i>0) and (Array[i]>Temp) do
8                          Array[i+1]:=Array[i];
9                          i:=i-1;
10
11                 Array[i+1]:=Temp;
12          }
```

Taking a deeper look at the pseudo code may be very helpful as it may result in a better understanding of the algorithm itself. That gained knowledge can be very useful when working on the actual implementation. Refering to this example, developers now may see that a usual implementation is (1) built with two loops and (2) requires a "temp" element. In addition one can see that the "temp" element is assigned in the outer loop (3). As **NetLuke** requires a stepwise execution of the algorithms, this can be considered to be very important: Dependent on the implementation, the "temp" value may be valid for more than one step. In that case it might be a good decision to show the "temp" value to the user during the visualization on the client side. This decision in turn has influence on the implementation of the changeset and the JavaScript part.

6.2 The Java Part

This section aims to give a detailed explanation of server side module generation within NetLuke. It covers all the necessary steps to provide a detailed insight in this topic. As this can be done best by using an example I decided to describe the important steps of the implementation process.

6.2.1 Preparatory Work

In the first step a new Java Project in Eclipse has to be created. The project has to be named "InsertionSort"(or accordingly after the algorithm that will be implemented). Within this project three Java packages have to be created:

- (1)**at.ac.univie.netluke.plugin.insertionsort**
- (2)**at.ac.univie.netluke.plugin.algorithm**
- (3)**at.ac.univie.netluke.plugin.test**

In the next step the developer has to import the **netluke-cor.jar** library, which can be found on the GIT repository.[8] This can be achieved by performing a "right-click" on the project and selecting `Properties -> Java Build Path -> Libraries -> Add Jar)`. In the current version it is also required to add the gson library **gson-2.2.4.jar** which is required for serialization. After completing these steps the **NetLuke** project has to be configured in order to accept and load a new module. Therefore the *netluke.xml* config file inside the NetLuke project has to be edited as shown in listing 6.2. The netluke.xml is located at "*Webcontent\WEB-INF\config\netluke.xml*" of the NetLuke project.

LISTING 6.2: netluke.xml - config file

```
1 <!-- Sort Algorithms -->
2 <plugin-category type="Sorting">
3 .
4 .
5  <plugin identifier="InsertionSort">
6   <name>Insertion Sort</name>
7   <implementation>at.ac.univie.netluke.plugin.algorithm.InsertionSort
       </implementation>
8  </plugin>
```

As shown in the listing above a new **plugin** within the **sorting** category has to be created. This tells the system where to find the main class of the new module and which category the algorithm belongs to. As the user can later see the algorithm category this information is necessary. *When building modules for the first NetLuke it would be necessary to also create a describing HTML file for the algorithm in this step. As the client side has changed, this file is now accessed in a different way and therefore part of the client side module. (see chapter 6.3)*

6.2.2 Implementation

The main challenge for developers when implementing for **NetLuke** may lie within the requirement of providing a stepwise execution of the algorithms.[9] When doing research for our project we noticed that most algorithm implementations are constructed using loops. Loops are impractical when a stepwise execution is required: One execution would lead to the algorithm to run until it has finished. The problem every developer is confronted with now, is to rebuild an algorithm toward a stepwise execution without loosing its working principles.[10] This can be done by splitting up the algorithms logic into a **next()** and a **back()** function and saving all the necessary data between two steps during algorithm execution. The result of every step has to be transmitted to the client to achieve a proper visualization. But how can this "splitting up" of the algorithm logic be achieved and what data should be sent to the client side ?

[8] As the project needs to be exported as JAR at the end, we recommend to copy the library file into the project folder

[9] I chose to demonstrate the algorithm implementation within NetLuke using a sorting algorithm as most implementations of these kind of algorithms are not designed to support a stepwise execution. This allowed me to underline the importance of this requirement of the NetLuke system

[10] This especially relates to sorting algorithms because other algorithms like trees or hashing have a stepwise behaviour by default, as these algorithms require new input after every step.

6.2.3 The Main Java Class

The core of every server side module within **NetLuke** is the main class of the algorithm. The first step of the implementation consists of creating a new Java class inside the **ac.at.univie.netluke.plugin** package. Once again the class has to be named after the algorithm. In order to work properly this class has to provide specific methods that can be called from the client side. The developer is free to add as many additional methods as he wants while these basic methods are included. Listing 6.3 illustrates a boilerplate for the "InsertionSort" algorithm including the necessary methods. As InsertionSort is a sorting algorithm the building blocks provided by the **netluke-core.jar** library can be used. By extending the *SortAlgorithm* class the developer is already provided with an array of the type *Element* and some basic operations to manipulate this array. Methods that need to be called from the client need the *@NetLukeMethod* annotation, as mentioned in section 5.2.1. By calling *super()*, as shown on line 9 of the listing, the values that are delivered to the constructor are being transformed into objects of the type *Element* and stored into the provided array.

LISTING 6.3: InsertionSort.java - Biolerplate

```
1  package ac.at.univie.netluke.plugin.algorithm;
2  public class InsertionSort extends SortAlgorithm implements
       Serializable {
3
4          private static final long serialVersionUID = 745663658742 99L;
5          @NetLukeMethod
6          public InsertionSort(final int[] values) {
7                  super(values);
8          }
9          @NetLukeMethod
10         public NLResponseContent back() {
11                 //todo back-method()
12         }
13         @NetLukeMethod
14         public NLResponseContent next() {
15                 //todo next method()
16         }
17         @NetLukeMethod
18         public NLResponseContent getArray() {
19                 return new NLResult(super.getArrayAsXML());
20         }
21         @NetLukeMethod
22         public NLResponseContent getInitialArray() {
23                 return new NLResult(super.getInitialArrayAsXML());
24         }
25         @NetLukeMethod
26         public NLResponseContent getArraySize() {
27                 //todo get-arraySize()
28         }
29         @NetLukeMethod
```

```
30          public void run() {
31                  //todo run-method()
32          }
33          @NetLukeMethod
34          public void reset() {
35                  //todo reset-method()
36          }
37  }
```

The purpose of the *next()* method is to conduct one step forward in algorithm execution whereas *back()* has to trigger an equivalent step backward. It depends on the algorithm and on the developers decision what exactly is done in one step. My colleague and I decided not to create any restricting rules in order to give the developers the full freedom when designing their algorithm module concepts. When taking a look at the exemplary "InsertionSort" implementation, illustrated in listing 6.4 one can see that it is built using two loops and two counter variables. In order to create an InsertionSort module for the NetLuke System it is necessary to rebuild this code.

LISTING 6.4: InsertionSort - Examplary Implementation

```
1  Insertionsort (int data[],int n) {
2     int tmp,i,j;
3     for (j=1; j<n; j++) {
4         i =j - 1;
5         tmp = data[j];
6         while ( (i>=0) && (tmp < data[i]) ) {
7             data[i+1] = data[i];
8             i--;
9         }
10        data[i+1] = tmp;
11    }
12 }
```

Developers need to create a *next()* function that, when executed, conducts exactly one step forward in algorithm execution. Applied on the InsertionSort, this means that the "loops" have to be replaced with "*if . . . else* " constructs. (Basically it can be said that this applies for all occuring loops). The developer can now decide whether he replaces both loops(1) or he just replaces the outer loop(2). This is a serious decision. When replacing . . .

1. . . . both loops every step is executed on the server. With an efficient design, nearly no algorithm logic on the client side is necessary. This way requires more communication between the client and the server as every algorithm step that has to be visualized on the client side needs information from the server. (More changesets are being transferred)

2. . . . just the outer loop the developers has to implement some of the algorithm logic on the client side as well. When this method is chosen, every step that occurs in

the inner loop has to be reproduced on the client side as well. Synchronization is only done after a step occurring in the outer loop. This requires less "client-server" communication but more logic on the client side.

When designing the InsertionSort module I chose to replace both loops in order to keep the client logic as simple as possible.

LISTING 6.5: InsertionSort - Rebuilt for NetLuke

```
1  public boolean next() {
2  if (this.firstrun) {
3    step.setFirstrun(true);
4    this.j=1;
5    this.i=j-1;
6    this.temp=this.sort_array[j].getValue();
7    this.tempE=this.sort_array[j];
8    this.firstrun=false;
9   }
10  if (!this.sorted) {
11    if (this.i>=0 && this.temp < this.sort_array[i].getValue()) {
12      //If construct representing the inner loop
13          this.sort_array[this.i +1] = this.sort_array[this.i];
14      this.i--;
15     }
16    else { //Else .. representing the Outer Loop
17          this.sort_array[i+1]=this.tempE;
18          this.j++;
19        if (j == this.sort_array.length) {
20            //Algorithm has finished
21            this.sorted=true;
22            return true;
23           }
24          else {//Algorithm not finished
25                this.i=this.j-1;
26                this.temp=this.sort_array[j].getValue();
27                this.tempE=this.sort_array[j];
28           }
29         }
30   return true;
31  }
32 return false;
33 }
```

During this step it is also necessary to make the counter variables "global", as they will be needed in the *back()* function as well. When rebuilding the code, the developer needs to take care not to change the principle of the algorithm. The actions performed in the rebuilt code (swapping elements, increasing the counters) have to stay the same as before. Listing 6.5 illustrates a possible implementation of a *next()* function of the InsertionSort without using loops. This sample code already uses the functionality provided by the

netluke-core.jar library. As a result the sort array, the algorithm is based on, uses the *Element* data type. This requires the *getValue()* function to be used in order to compare the values of two different elements as shown on line "3". At the beginning it is necessary to define the counter variables ("i" & "j") as well as the temp element and the temp value. While the condition of the inner loop could be taken over without any troubles (as one can see on line "7"), replacing the outer loop was a bit more complex. In order to find out, whether the algorithm has finished or not, another "*if ... else* "construct was necessary. It also required some testing to find out when to increase the counters. The result is a *next()* function, implying the logic of the InsertionSort algorithm, that can be used in the NetLuke environment.

I want to point out that there is no general set of instructions on how to rebuild an algorithm in order to achieve a stepwise execution as every algorithm is different regarding the working principles it is based on. This example does not aim on giving a detailed tutorial on how to redesign the code of an algorithm in order to fit these requirements, it rather should show possibilities of how this target can successfully be achieved. There are many sources that provide implementation examples for various algorithms, but hardly any of these sources offer illustrations for an implementation of a stepwise execution. That is the reason why the redesign of the algorithm can be quite difficult, especially when implementing a sorting algorithm.

Beside conducting one step forward in algorithm execution, the *next()* function also has to deliver the necessary information in order to visualize the step on the client side. This can be achieved using **changesets** as mentioned in section 5.2.3. Before designing the changesets for the *next()* function it can be helpful to first design the *back()* function as both may share the same logic. This function conducts a backward step on the server and deliver the necessary information to the client for an appropriate visualization. Applied on the InsertionSort the *back()* function has to...

- ...set the counters to the values before the last "next" step
- ...invert all changes on the sorting array that occured during the last "next" step
- ...set the temporary element to the value before the last "next" step
- ...send all the necessary information to the client in order to visualize the backward step (sending the **changeset**)

This can only be achieved when saving some of the actions performed during the "next" step. At this point one can see that the information that is stored during a "next" step, to provide a "back" functionality, can also be used for visualization purposes on the client side. A question that may arise at this point is how to save this information and how can these **changesets** be created?

6.2.4 Saving Algorithm Steps

To gain the ability of saving every step during algorithm execution the developer needs to create a new data structure that can hold this information.[11] Therefore a new java

[11] Although the principle of saving the steps in order to provide a "back" functionality stays the same throughout all algorithms, the actual implementation varies from module to module. The purpose of

class inside the **at.ac.univie.netluke.plugin.insertionsort** package has to be created. The class should be named **InsertionSortStep.java** or accordingly after the algorithm that is gonna be implemented. The data structure should contain variables that keep the necessary information in order to restore the last algorithm state and provide getter and setter methods to access these variables. It depends on the developers deliberations and on the algorithm complexity what to actually save. As the back logic requires the information stored in this data structure it has to be well thought out. For the implementation of the InsertionSort I decided to create a "heavyweight"(saving many variables) step. That is because I wanted to restore the last state without getting the required logic too complex. Table 6.2 illustrates the created variables.

Variable Name	Type	Function
i&j	Integer	Save the actual counter value
temp_pos	Integer	Save position of temp element
temp_value	Integer	Save value of temp element
swap_pos	Integer	Save position of swap
outer_swap _pos	Integer	Save position of swap in outer loop
swapped	Boolean	Indicator whether a swap in the inner loop occurred
swap_temp	Boolean	Indicator whether a swap in the outer loop occurred
firsturn	Boolean	Indicator whether it is the first start
sorted	Boolean	Indicator whether the algorithm has finished
swap	Element	Save of the swap Element
outer_swap	Element	Save of the outer swap Element

Table 6.2: InsertionSortStep Variables

Every single instance of the **InsertionSortStep** class is capable of saving one algorithm step. As most algorithms require more steps until their execution has finished, saving one "step" object is not sufficient. Every execution of the "next" function should create a new step object that needs to be stored. Another data structure for saving the previous "steps" is required. We chose to use the ArrayList, as it can be used without further preparations and comes with all the necessary functionality. The ArrayList is part of the *java.util** package. A new (global) ArrayList with the datatype **InsertionSortStep** has to be created within the **InsertionSort.java** class. We decided to call this data structure "steps" throughout all algorithm implementations, als listing 6.6 illustrates.

LISTING 6.6: ArrayList for holding steps

```
1 private ArrayList<InsertionSortStep> steps=new ArrayList<
    InsertionSortStep>();
```

Afterwards it is necessary to modify the "next" function to that effect that a new "step" object is generated every time the function gets called. Additionally it is necessary to set

this section is to show that principle on the basis of an actual implementation and not to give a detailed instruction

the variables of the generated “step“ to be able to reconstruct the actual step. At the end of the “next“ function the new generated step object has to be added to the ArrayList. The necessary changes for the InsertionSort are illustrated in listing 6.7.

LISTING 6.7: InsertionSort - Saving Changes

```
1  public boolean next() {
2  if (!this.sorted) {
3   final InsertionSortStep step=new InsertionSortStep(this.i, this.j);
4   if (this.firstrun) {
5    step.setFirstrun(true);
6    step.setTemp(this.tempE);
7    }
8    if (!this.firstrun) {
9     step.setTemp_value(this.temp);
10    step.setTemp(this.tempE);
11    }
12    if (this.i>=0 && this.temp < this.sort_array[i].getValue()) {
13    step.setSwaped(true);
14    step.setSwap_pos(this.i +1);
15    step.setSwap_Element(this.sort_array[this.i + 1]);
16    }
17    else {
18         step.setSwap_temp(true);
19     step.setOuter_swap(this.sort_array[this.i+1]);
20     step.setOuter_swap_pos(this.i+1);
21         if (j == this.sort_array.length) {
22           this.steps.add(step);
23         }
24         else {
25     }
26   }
27  this.steps.add(step);
28  return true;
29  }
30  return false;
31  }
```

6.2.5 Implementing Back Functionality

At this stage of the development process the developer can begin with implementing the back functionality as all requirements are fulfilled. When executed, the "back" function has to get the information from the last "step" object, stored in the "steps" ArrayList, restore the algorithm to the previous state and afterwards delete the "step" object from the data structure. Generally it can be said that implementing the back functionality is not that complex when the step object contains sufficient data. My implementation of the *back* function for the InsertionSort is quite simple and performs following actions:

- Check if there is a step object in the ArrayList. If there is none, no previous step happened
- Set the counters (i & j) of the sort instance to the values stored in the step object.
- Restore the Temp Element.
- Restore the temp value.
- Restore the *sorted* indicator (boolean).
- Restore the *firstrun* indicator (boolean).
- Reverse a potentially occurring swap.
- Reverse a potentially occurring swap of the temp element.
- Delete the *step* object from the *steps* ArrayList.

Extensive testing of the *back()* and *next()* function is highly recommended as they may be a potential source of errors that can be hard to find in a later development stage. As the back function should also deliver a **changeset** to the client, a listing will be provided later in this section when the function is completely implemented.

6.2.6 Changeset Implementation

Changesets incorporate the information necessary to visualize the algorithm changes on the client side. They need to be generated and transmitted to the client after every execution of the *back()* or *next()* function. Within the NetLuke environment, the developer is not strictly bound to a specific messaging format. However we recommend to use XML or JSON messages as JavaScript (especially JQuery) makes it very easy to parse messages that are delivered using these formats. For the InsertionSort I decided to use XML, as some methods provided by the **netluke-core.jar** library are not yet upgraded in order to return JSON messages. Although this is not a big problem, a developer who decides to use the JSON format may have to write some more logic on his own.[12] As mentioned before there are no restrictions for the developer regarding the design of the changeset. Depending on the design and the current algorithm the developer may have to implement two **changesets**, each one used by either the *back()* or the *next()* function. On the other hand there are situations where one **changeset** is sufficient for both functions.

[12]The *getArrayAsXML()* function within the **at.ac.univie.netluke.core.sort** package for example has no JSON counterpart yet.

Changeset functionality is provided in a seperate class that has to be created inside the **at.ac.univie.netluke.plugin.insertionsort** package. We decided to call this classes *StepChangeSet* throughout all algorithm implementations. The purpose of this class is to get a (XML) changeset of the actual changes. To achieve this goal we decided to follow the now shown approach: (1) The constructor of the class expects the current sorting array (the type of which is Element) and the current step. (2) The constructor automatically calls a function that analysis the step and extracts the necessary information. This information is then parsed to the XML format. (3) The class provides functionality to access this XML message. Listing C.1, which can be found in the appendix, illustrates the implementation of the **StepChangeSet** class.

When implementing the InsertionSort I decided to use the same changeset for both functions. The only way this could be achieved was to generate the changesets during the *next()* function and save them in another data structure. I once again decided to use the ArrayList as illustrated in listing 6.8.

LISTING 6.8: ArrayList for holding changesets

```
1 private ArrayList<StepChangeSet> changesets = new ArrayList<
    StepChangeSet>();
```

This approach required to modify the *next()* function in a way to create a changeset of the current step and save it to the **changesets** ArrayList. The **step** and the **changeset** have the same index in their ArrayList as they are generated at the same execution period of the *next()* function. This changeset also represents the return value of the function. When the modification of the *next()* function is done, the developer can finish the implementation of the *back()* function as the required changeset, that should be send to the client, is now available. Listing 6.9 illustrates the implementation of the *back()* function for the InsertionSort.

LISTING 6.9: InsertionSort - Back Function

```
1  private String backSort() {
2    if (this.steps != null && !this.steps.isEmpty() && this.steps.size
       () >= 1) {
3        final InsertionSortStep step = this.steps.get(this.steps.size
           () - 1);
4        this.i = step.getI();
5        this.j = step.getJ();
6        this.tempE = step.getTemp();
7         if (step.isSwaped()) {
8          this.sort_array[step.getSwap_pos()]=step.getSwap_Element();
9         }
10        if (step.isSwap_temp()) {
11         this.sort_array[step.getOuter_swap_pos()]=step.getOuter_swap
              ();
12        }
13        this.sorted=false;
14        this.temp=step.getTemp_value();
15        this.firstrun=step.isFirstrun();
16        this.steps.remove(this.steps.size() - 1);
```

```
17          final String result = this.changesets.get(changesets.size() -
                1).getChangeSet();
18          this.changesets.remove(this.changesets.size() - 1);
19          return result;
20      }
21      return null;
22  }
```

Every time the *back()* function is executed, one changeset from the ArrayList will be returned to the client and deleted from the data structure. When following this approach, the *next()* and the *back()* function return the same changeset for the same step, as one can easily see. This decision comes with some stumbling blocks. In order to deal with this (minor) problem, some client logic is necessary. This will be described in section 6.3.

6.2.7 Utilization Of New Features

When developing NetLuke 2 we also refurbished the **netluke-core.jar** library to achieve a better error handling during client-server communication. During this refactoring process we created the **NLResponseContent** interface. As **NLResponseContent** is only an interface it is not possible for a function to return a value of this type. Therefore we created the **NLResult**, the **NLResultType** and the **NLErrorType** classes which all "implement" this interface. While the first class can be seen as a wrapper that can be used to send results like "strings" or XML messages, the other classes represent lists of response types and errors that can be used by the developer instead of writing his own response messages. When the **NLResult** return type is used, the data inside the message will be automatically checked and in case of an error an appropriate messaged will be displayed in the client log. For every method that sends data to the client, it is now mandatory to have a return type implementing the **NLResponseContent** interface, as we implemented some features to improve the error handling when this data type is used. The benefit of this new feature will be demonstrated in section 6.3.

In order to fulfill these requirements another modification of the *next()* and *back()* function is necessary. There are two possible ways to implement these modifications. Either the return type of the two functions can be changed or two new functions with the new return type can be generated. The new functions have to call the existing functions. For the purpose of a better demonstration I decided to create new functions. As the new functions inherit the name of the existing functions it was necessary to rename the old ones. Therefore I renamed the existing functions to *nextSort()* and *backSort()*. The new *back()* function is quite simple. It just calls the existing function and checks whether it returns a value or not. If that is the case, a new **NLResult** instance with the return value from the existing function is created and returned. If there is no return value a new **NLErrorType.Empty()** is generated and returned. The new *back()* as well as the renamed *backsort()* functions are illustrated in listing C.2 which is located in the appendix.

The new *next()* function is slightly more complicated as the existing function has "boolean" as return type. This means that the existing function is not able to return the actual changeset. This problem can be solved by testing if the original function returns "true"

when called within the new *next()* function. If that is the case a new **NLResult** from the changeset, that has been created within the *NextSort()* function, is being generated and returned. Otherwise a new **NLResulType.False()**, which is also based on the **NLResponseContent**, will be created and returned. The final *next()* and *nextSort()* functions are illustrated in listing lst. This listing also includes the necessary modifications in order to create a changeset everytime the function is being executed, as mentioned in section 6.2.6. After these changes are implemented the client side is able to process a stepwise back and forward execution and submit the information necessary for the visualization to the client using the generated **changesets**. But when taking a look at the boilerplate presented in listing 6.3 one can see that there are still some functions left that need to be implemented.

6.2.8 Implementing Absent Functions

After implementing the *next()* and the *back()* function there are the following functions left (annotated with the @NetLukeMethod) that the developer needs to take care of:

1. **public NLResponseContent getInitialArray()**: The purpose of this method is to return the initial sorting array without any changes. As this function is already provided by the library, there is no need for the developer to implement it. The initial array can be accessed by simple calling the provided method as shown on line number 23 in listing 6.3.

2. **public NLResponseContent getArray()**: When this method is called it should return the current sort array. This function is also part of the library. Once again a call of the provided method is sufficient. (This is illustrated on line number 19 in listing 6.3)

3. **public NLResponseContent getArraySize()**: This function returns the size of the sorting array. This is very important for the visualization as the client needs to know this value in advance. The reason for this will be described in section 6.3. The implementation of this function is quite simple as listing 6.10 demonstrates. If a sorting array exists within the current algorithm instance, a new **NLResult** containing the size of the array will be returned. Otherwise the function returns an appropriate error message.

4. **public void run()**: This function is for testing purposes only. It is used to repeatedly call the *next()* function until the algorithm has reached the "finished" state. The *run()* method on the client will be implemented using only the *next()* method as it needs a changeset after every step in order to visualize the algorithm execution.

5. **public void reset()**: The *reset()* function restores the algorithm to it's initial state. Therefore the counters have to be reset, the current sorting array has to be overwritten with the initial sorting array and the two ArrayLists holding the **steps** and the **changesets** have to be cleared. The *reset()* function for the InsertionSort also has to set the *firstrun* indicator to "true". This function is illustrated in listing 6.11.

LISTING 6.10: InsertionSort - getArraySize - Function

```
1 public NLResponseContent getArraySize() {
2   if (this.sort_array == null) {
3         return new NLError("The sort array is null and has no size");
4   } else {
5         return new NLResult( String.valueOf(this.sort_array.length));
6         }
7 }
```

LISTING 6.11: InsertionSort - Reset - Function

```
1 public void reset() {
2  this.sorted = false;
3  this.firstrun = true;
4  if (this.changesets != null) {
5     this.changesets.clear();
6  }
7  if (this.steps != null) {
8         this.steps.clear();
9  }
10 this.j=1;
11 this.i=j-1;
12 this.sort_array = (this.initial_array != null) ? this.initial_array.
       clone():null;
13 }
```

6.2.9 Deploying The Java Module

After all necessary functionality has been implemented and if the project does not contain any errors the java module is ready to be deployed to the NetLuke server. Therefore the current (InsertionSort) project has to be exported as a "jar" archive. This can be done directly within Eclipse. The archive should be called **netluke-plugin-insertionsort.jar**. The new generated archive has be placed in the *Webcontent\WEB-INF\lib* folder inside the **NetLuke** project. If not already done, it is necessary to edit the *netluke.xml* as mentioned in section 6.2.1. Afterwards the NetLuke server can be started by performing a right click on the **NetLuke** project and selecting `Run As -> Run on Server`. If the *netluke.xml* has been edited correctly the new module should be found by the server and loaded automatically. This can be verified by taking a look at the console messages written by the NetLuke system. If the plugin has been loaded successfully an appropiate message, as shown in figure 6.1 will be displayed.

If the deployment of the new module has been successful the developer can use the built-in testing methods of the NetLuke system and subsequently begin with the implementation of the client side. At this point I want to point out that the server side most likely has to be refurbished sometimes when implementing the client.[13]

[13] Although we recommend extensive testing before implementing the client side, some combinations of backward and forward execution, that cannot be tested in advance, may lead to an unexpected behavior. This may require some redesign of the server side.

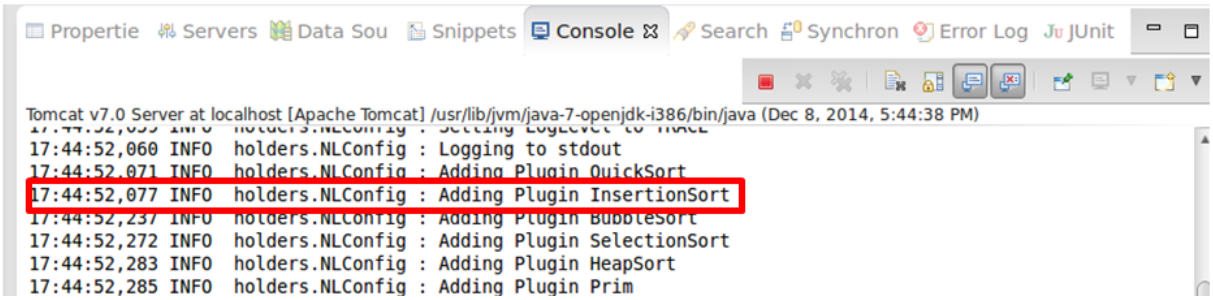

Figure 6.1: NetLuke - Console Log

6.3 The JavaScript Part

This section is all about the client side of the algorithm module. It is recommended to finish the server side implementation before beginning the work on the client side. This section continues the work on the "InsertionSort" algorithm and aims to describe all necessary steps for creating a fully functional module visualization.

6.3.1 Preparations

As the client side of the module is written in JavaScript the developer should use a compatible editor. Although it is also possible to use Eclipse for developing JavaScript we recommend to use a more specialized programming environment. We used a Beta version of an editor called "**Sublime Text**"[14] which we made good experience with. (Beta version 3). However, any editor that supports syntax highlighting should do it. An editor that supports automatic error detection would be even better, as JavaScript can sometimes be tricky to debug.

In the first step of the client developing process the developer needs to create a new folder right within *Webcontent\app\algo*. The folder has to be named after the algorithm that is gonna be implemented. In our case the name of the folder is **InsertionSort**. The next step consists of creating a new JavaScript file that is equally named. (**InsertionSort.js**). It is very important to follow these naming guideline for the algorithm implementation file to be found by the system.

In the last step of the preparation the module developer needs to create a description of the algorithm. This description is a simple HTML file that is loaded together with the algorithm module on system start up. The purpose of this file is to store basic information about the algorithms working principle, its complexity and the general runtime. Also the last changes in the implementation and the name of the developer can be documented. This description file is shown to the user whenever the algorithm gets selected on the "algorithm selection" menu. Once again this file has to be named after the algorithm. **InsertionSort.hml** It is stored together with the JavaScript file and is required to follow a given structure, as illustrated in listing 6.12.

[14] http://www.sublimetext.com/

LISTING 6.12: InsertionSort.hml - Algorithm Description File

```
<div id="InsertionSort">
        <h1>Insertion Sort Description</h1>
        <h2>Description</h2>
        <div>
        <--Description goes here-->
        </div>
        <ul>
                <li>Complexity:</li>
                <li>Working mode:</li>
                <li>Sorts:</li>
        </ul>
        <div>
         <div style="float:left;">
                <h4>Author</h4>
                Name:<br>
         </div>
         <div>
                <--Image goes here-->
        </div>
        <h4>Changelog</h4>
        <table>
         <tr><th>Version</th><th>Changes</th></tr>
         <tr><td>0.1</td><td>Java implementation</td></tr>
        </table>
        <h4>References</h4>
        <ul class="reference">
        </ul>
</div>
```

6.3.2 Base Structure

The tasks of the client side consists of calling the provided server functions and create the visualization based on the server's response. The whole logic on the client side module has to be located in the JavaScript file. Since NetLuke 2 we introduced following concept for client side development: Every algorithm implementation is part of the **NL** namespace that we established on the client side with the new library. Every algorithm implementation presents an object that is placed inside an array named **Plugins** that is part of the library. Due to this approach the client side library has access to all available algorithm implementations.

Listening 6.13 illustrates a boilerplate for the client side. In this listening one can see the usage of the namespace (lines 1 & 31) and the object definition (line 3). The example also demonstrates the configuration object within the plugin definition (line 4) which includes the definition for the response format that should be used (XML) and entries for the side menu. After that all necessary functions that are required for the visualization and control of the algorithm are shown. (lines 14 - 29).

LISTING 6.13: JavaScript - Boilerplate

```
(function(NL) {
 'use.strict';
  NL.Plugins.InsertionSort = {
   configuration :{
    responseFormat :'xml',
     menuEntries :{
      creation :{ },
      reset :{ }
         },
   },

 init :function() {
 },
 initializeView :function() {
 },
 getArray :function() {
 },
 drawArray :function(array) {
 },
 play :function() {
 },
 next :function() {
 },
 back :function() {
 },
 reset :function() {
 },
};
})(NL);
```

After the ground structure of the implementation is finished, the developer should define the elements that are necessary for the algorithm visualization. The *InsertionSort* is a sorting algorithm that works on the following principle: The value inside the array that is pointed on by the outside counter (J) gets saved into a temporary variable. The inner counter (I) that is always smaller than J points to the position "J-1" at the beginning and is decremented after every run. If the value inside the array indicated by "I" is smaller than the temp variable it gets stored at the temps variable original place in the array. If the run is finished the temp variable gets stored at the empty place in the array and is now considered to be sorted.

In order to visualize this behaviour we need a (Konva) layer, a (Konva) stage, a graphical representation of the array, two counters and a place to store the temp value. As all required elements are provided by the library the developer is just required to include them in his implementation. Before this can be done, it is necessary to declare variables for these elements that are accessible throughout the algorithm (global declaration). This is illustrated in listing 6.14. The actual initialization of the elements occurs in the *init()* method, that is the first function that is called when the algorithm module is loaded.

This method verifies if there is already an existing instance of this algorithm for the user and initializes the stage and the layer. If an existing instance is found, it gets loaded, otherwise the user should be redirected to the "creation menu". The implementation of the **init** function is demonstrated in listing 6.15 while the **initializeView** function, that is called by the **init** function is illustrated in listing 6.16.

LISTING 6.14: JavaScript - Declearation of global Elements

```
1 NL.Plugins.InsertionSort = {
2  stage :{},
3  arrayLayer :{},
4  arr :{},
5  counterJ :{},
6  counterI :{},
7  swapBox :{},
```

LISTING 6.15: JavaScript - Init Function

```
1 init :function() {
2  var self = this;
3  this.initializeView();
4  var request = new NL.Request({
5      async :true,
6      method :'isInstantiated',
7        parameters :[{
8          data :'InsertionSort'
9        }],
10     onSuccess :function(xml) {
11      if(xml.text() === 'false') {
12        NL.Menu.triggerEntry('creation');
13          } else {
14       self.drawArray(self.getArray());
15           self.getCurrentStatus();
16     }
17    }
18  })
19 NL.Connection.send(request);
20 }
```

LISTING 6.16: JavaScript - InitializeView Function

```
1 initializeView :function() {
2  this.stage = NL.Element.Stage.get();
3  this.arrayLayer = new Konva.Layer();
4  this.stage.add(this.arrayLayer);
5 }
```

When taking a deeper look at the **init** function one can see two interesting aspects: (1) In line number 4 a new **NL.Request()** is defined. Since NetLuke 2 this way is used to call methods provided by the server side. In this case a request is defined that calls the *"isInstantiated "* function. A parameter, the name of the algorithm, is also being dispatched.

The "*onSuccess*" function is being called when the answer from the server arrives. (2) If there is no existing instance for the current user, a menu within the *NL* namespace named **creation** is being triggered. The *NL.MENU* is our new concept of allowing developers to customize the algorithm specific menu entries. If an existing instance is found however, the system should draw the array and restore the last state. Therefore it is necessary to call the **drawArray()** function followed by the **getCurrentChanges()** function.

Menu Creation

As a result of the unified UI developers can now customize the menu entries. This includes the displayed name of the menu button, the function that is called and some other functionality. Basic functionality requires a *Creation* button and a *Reset* button. Developers are welcome to add more entries if required. (like remove, etc.). Menu entries are simply added as objects inside the global object entry. (See line 6-9 in listing 6.13). While the *Reset* entry just has to call a function that is defined inside the algorithm module, the *Creation* entry is more advanced. As our library provides a creation wizard for arrays (and other algorithms) some more arguments are required as shown in listing 6.17.

LISTING 6.17: JavaScript - Menu Entries

```
1  menuEntries :{
2    creation :{
3     entryName :'Create',
4     type :'array',
5     name :'InsertionSort',
6     dataType :'integer',
7     maxValues :10,
8     minValues :4,
9     allowNegative :true,
10    maxValueLimit :9999,
11    minValueLimit :-999,
12    closeRightMenuOnClick :true,
13    closeRightMenuOnClose :true,
14    constructorCallback :function(xml) {
15          var self = NL.Plugins.InsertionSort;
16          NL.Element.Stage.reset();
17          self.initializeView();
18               if (xml.text() === 'ok') {
19          self.drawArray(self.getArray());
20        }
21    }
22
23   },
24   reset :{
25    entryName :'Reset',
26     entryFunc :function() {
27      NL.Plugins.InsertionSort.reset();
28        }
29    },
30 }
```

By providing the library the necessary information like the data structure, min value, max, value and the name of the algorithm the developer is relieved from the task of creating a creation menu by himself, as this is done automatically. When a user wants to create a new algorithm instance an appropriate menu appears that let him decide whether to enter self selected values or to generate random ones. As the library knows the name of the algorithm, the constructor on the server side is automatically called with the generated values.

Drawing the Algorithm

After the instance has been generated (or if the system finds an existing instance for the user) it needs to be presented to the user. As the instance creation happens on the server side, (the name of the constructor and the necessary parameters are just being invoked by the client) it is necessary to gather the information about the current instance from the server before it can be visualized. This is illustrated in listing 6.15 on line "14" and in listing 6.17 on line "19". The function that enquires the information from the server is called **getArray()**. As one can see in listing 6.18, that shows the implementation of the **getArray()** function, we use the same principle as before: We generate a new *NL.Request* object in order to remotely call a function that is provided from our server side module that has been implemented before. The function creates a JavaScript array that is then filled with the values from the array that is stored on the server. By following this approach we managed to create an identical copy of the data structure saved on the Java side.

LISTING 6.18: JavaScript - getArray Function

```
1  getArray :function() {
2    var array = [];
3    var request = new NL.Request({
4     invoke :'method',
5     method :'getArrayAsXML',
6     async :false,
7     onSuccess :function(xml) {
8      xml.find("element").each(function() {
9        array.push(parseInt($(this).text()));
10         });
11
12     }
13   }).send();
14  return array;
15 }
```

This time however we told the system to wait for the server result before continuing with any other operations. This is done by setting the **async** flag to "false". This is necessary as the result of this function is used as the input parameter for the **drawArray** function. This flag is used to make the AJAX calls synchronous. They are asynchronous by default, which enables the JavaScript environment, that is single threaded by design, to continue executing the source code after the position of the asynchronous call. In this case however we need the system to wait for the response, in order not to execuce the following function

(**drawArray**) without the required input parameters. After the server answer arrives, the JavaScript array is filled and returned. In our implementation the **getArray** function is directly called by the **drawArray** function. This function is used to initialize the graphical element representation, the counters and the "**NL.TempCell**" (our graphical representation of the temp element) and assign it to the according variables.

LISTING 6.19: JavaScript - drawArray Function

```
1  drawArray :function(array) {
2   this.arr = new NL.Element.NLArray(array, {
3    layer :this.arrayLayer
4   });
5   var arra = this.arr;
6
7   self.counterI = new NL.Element.NLCounter(arra, "i", {
8    position :0,
9    fill :'red',
10   visible :true
11  });
12
13  self.swapBox = new NL.Element.NLTempCell(arra,{
14   position :1,
15   });
16  var stage = this.stage;
17  stage.draw();
18 },
```

In listing 6.19 which shows a snippet of the **drawArray** function, the most important parts are illustrated. Here one can see that we use the *NL.Element.NLArray*[15] element for the graphical visualization of the underlying data structure (an array). The array that holds the values is required as input parameter. A second input parameter expects a configuration object. This configuration object is used to tell the array the layer it should be drawn on. The **NL.Element.NLArray** element then automatically creates the visualization based on the input array without any further intervention required from the developer.[16] The **NL.Element.NLCounter** is used in order to represent the counters of the algorithm. As every counter element depends on an array element it is required to specify this base element upon counter creation. The constructing function also expects a name for the counter and a configuration object. We called the first counter "I". As illustrated in the snippet, the starting position of the counter is "0", it has a red colour and is visible after the creation. In this regard the position is based on the particular cells of the array. Position "0" means that the counter is plotted above the first cell of the previously defined array. The second counter, named "J", has a starting position of "1" (plotted over the second cell) and is coloured green.
Another element that is required for the visualization of the "InsertionSort" is the

[15]More information regarding the different elements provided by the new library can be found in chapter 4 and on the project page.

[16]The different methods provided by the library in order to manipulate the various objects responsible for the visualization are presented in chapter 4. A detailed documentation can be found on the project site.

NL.Element.NLTempCell. This element is able to graphically represent one value and is plotted below the array. As it is able to store a number from the array and to put a value back into the array it is also required to define the underlying array upon element construction. When this element is defined it is also necessary to provide a config object as second input parameter. Once again this configuration object must contain the position parameter that is referring to the array element.

After all required elements are defined it is necessary to make them visible. In this implementation all elements are on the same layer. Implementations of other algorithms may require additional layers for achieving a satisfying visualization. Every graphical object however lays on the same stage and in order to make newly added elements visible it is necessary to redraw this stage. This can be seen on line "17" in listing 6.19.

Implementing Algorithm Control

For the algorithm control it is necessary to implement the **next** and the **back** function which are responsible for calling their counterparts on the serverside and visualize their response subsequently. The **next** function once again uses a **NL.Request** object to perform a function call on the server side. The server function then returns the changeset that we defined when implementing the Java part of the module. This changeset is used as input parameter for another function, called **applyCurrenChanges**, that is responsible for the actual changes within the visualization. The **next** function is illustrated in listing 6.20 while the **applyCurrentChanges** function can be found in the appendix (C.3).

LISTING 6.20: JavaScript - next Function

```
1  next :function() {
2   var self = this;
3   NL.Player.disable('nextEntry');
4   if (!self.isAnimating) {
5    new NL.Request({
6     method :'next',
7     async :false,
8     onSuccess :function(xml) {
9      self.applyCurrentChanges(xml);
10    }
11   }).send();
12  }
13  return self.nextDone;
14  }
```

As one can see the **next** method verifies if a specific variable is false before the task is executed. This *isAnimating* flag is necessary for the correct timing of the **play** method, that will be discussed later in this section. Another important aspect can be seen on line number "3" in the listing. This line disables the *"next"* button on the algorithm control menu. The function itself returns whether the algorithm execution is already finished or not.

The **applyCurrentChanges** function is a more complicated one. In the beginning the

function checks whether the changeset from the server contains a *"result"* entry. If not, the algorithm is considered to be finished, the flag is set accordingly and the function returns. Otherwise the changeset is being parsed to JavaScript variables. The main functionality however is located in the **todo** function that is defined inside the **applyCurrentChanges** function. This function itself includes six other functions that are responsible for applying the changes to the graphical representation of the "InsertionSort"algorithm. For this construct to work, the **todo** function is being called after its definition. This function then verifies whether there is no current animation running that could interfere the current task. (*"isAnimaiting"* has to be false). Afterwards the system begins to apply the changes on the visualization. As JavaScript is single threaded it is required to wait for every animation to be finished before starting a new one. All our library functions support so called *call-back* functions that are executed whenever the called function is finished. This can be seen when taking a look at function **f1**. This function places the graphical representation of the temp element to the position that has been extracted from the changeset. In the configuration object we tell the system that the animation should take 0.4 seconds. (line number 27 in listing C.3). After the function is finished the next function, named **f2** should be called. With this concept we allow developers to design even complex animations without worrying about the timing. Depending on the changeset the functions are being applied one after another and the animation of one step is shown to the user.

We designed the animation of the "InsertionSort" as follows: The temp element (indicated by the counter "J") is taken from the array and moved to the swapBox. If the element indicated by the second counter ("I") is greater then the temp element, it moves to the position "I+1" inside the array. The counter I is decremented by 1. Whenever "I" has reached the left end of the array ("I" = -1) or the element at position "I"is smaller than the temp element, the value stored in the temp array moves back to the array. It then gets positioned at index "I -1". Afterwards the counter "J" gets increased by 1 and "I" is assigned "J-1".

Besides the **next** function, which illustrates exactly one step within algorithm execution, the user can also push the "play" button that starts a complete run of the algorithm. The "play" button calls the **play** function that in turn repeatedly calls the **next** function until the algorithm has finished.

LISTING 6.21: JavaScript - play Function

```
1  play :function() {
2   NL.Player.disable('playEntry');
3   var self = this;
4   var intervalID = setInterval(function() {
5    if (self.next()) {
6       clearInterval(intervalID);
7           NL.Player.enable('playEntry');
8    }
9   },100);
10 }
```

As one can see in listing 6.21 we used the *setInterval()* function within the **play** method.

This function, that comes with JavaScript, allows developers to recurrently call another function after a specific amount of time. We use this function to call the **next** method every 100 milliseconds. Contrary to a loop this function is quite efficient regarding CPU time and allows the JavaScript environment to continue with another task. As the **next** function will not execute while the "*isAnimating*" flag is set, we decided to use such a short interval to guarantee a smooth animation. Once the algorithm has reached the "finished" state the interval function is being cleared.

Implementing Back Functionality

When pressing the "back" button the user should see the last step being reversed. Concerning the implementation the **back** function is quite similar to the **next** function. This method also calls the according function on the server side and uses the changeset, that is being returned, as an input parameter for another function that applies these changes to the graphical representation. In addition this method sets the "*nextDone*" flag to "false", in case it is being called after the algorithm has reached the finish state. In addition it sets another flag, called "lastback", that is used in our implementation to allow a correct sequence of animation steps when various back/forward combinations are applied by the user. Due to it's size I once again decided to move the listing of the **applyBackChanges** function in the appendix (listing C.4), while an illustration of the **back** method is positioned right here. (listing 6.22)

LISTING 6.22: JavaScript - back Function

```
1  back :function() {
2   var self = this;
3   self.lastback=true;
4   self.nextDone=false;
5   NL.Player.disable('backEntry');
6    new NL.Request({
7     method :'back',
8     async :false,
9     onSuccess :function(xml) {
10     self.applyBackChanges(xml);
11    }
12   }).send();
13 }
```

As the listing of the **applyBackChanges** function shows, it is not as complicated as the function responsible for representing the forward steps. We decided to make the "back" animation less complicated and as a result we required a less complicated structure for the timing. We used this concept on the "InsertionSort" for demonstration purposes. Every developer is free to decide whether the "back" animation should be as complicated as the one representing a step forward.

Implementing the Algorithm Control Menu

In the last step it is necessary to connect the functions that are responsible for the algorithm execution with GUI elements. If no appropriate GUI elements are being provided

a user won't be able to start the algorithm animation. At this point I want to admit that the steps performed in this section should be performed earlier during the development process. For a better overview however, I decided to make it the last point of the development guide. [17]

Many previously defined functions refer to a component named **NLPlayer**. This element is part of the new library and shows the new algorithm control menu. It has to be defined inside the main configuration object of the algorithm implementation.

LISTING 6.23: JavaScript - Algorithm Control Menu

```
playerEntries :{
 show :true,
  playEntry: {
   entryName: "Play",
   entryFunc :function() {
    NL.Plugins.InsertionSort.play()
    },
   show :true,
   enabled :true
  },
  nextEntry: {
   entryName: "Next",
   entryFunc :function() {
    NL.Plugins.InsertionSort.next()
    },
   show :true,
   enabled :true
  },
  backEntry: {
   entryName: "Back",
   entryFunc :function() {
    NL.Plugins.InsertionSort.back()
    },
   show :true,
   enabled :true
  }
```

As one can see in listing 6.23 the developer just needs to define which elements should be part of the menu. Every entry is then connected with a function that is being called when the according menu button is pressed by the user. It is also possible (and recommended) to disable the menu entries during the animation. This prohibits that a user calls "next" again before the function has even finished.

Starting the Module

When all implementations steps are complete the algorithm module is ready for launch.

[17] As some other functions are dependent on the elements that are being defined in this section, the algorithm control structure should be initialized at an earlier state of the development process.

If no mistake has been made the newly implemented algorithm appears in the "Algorithm Selection Menu" that is located in the main menu of the new client. After the algorithm selection a new instance needs to be generated. This is done by using the new "creation wizard". If the implementation is free of errors, the user should now see a graphical representation of the new algorithm including a menu to control the algorithm execution. This is shown below in figure 6.2.

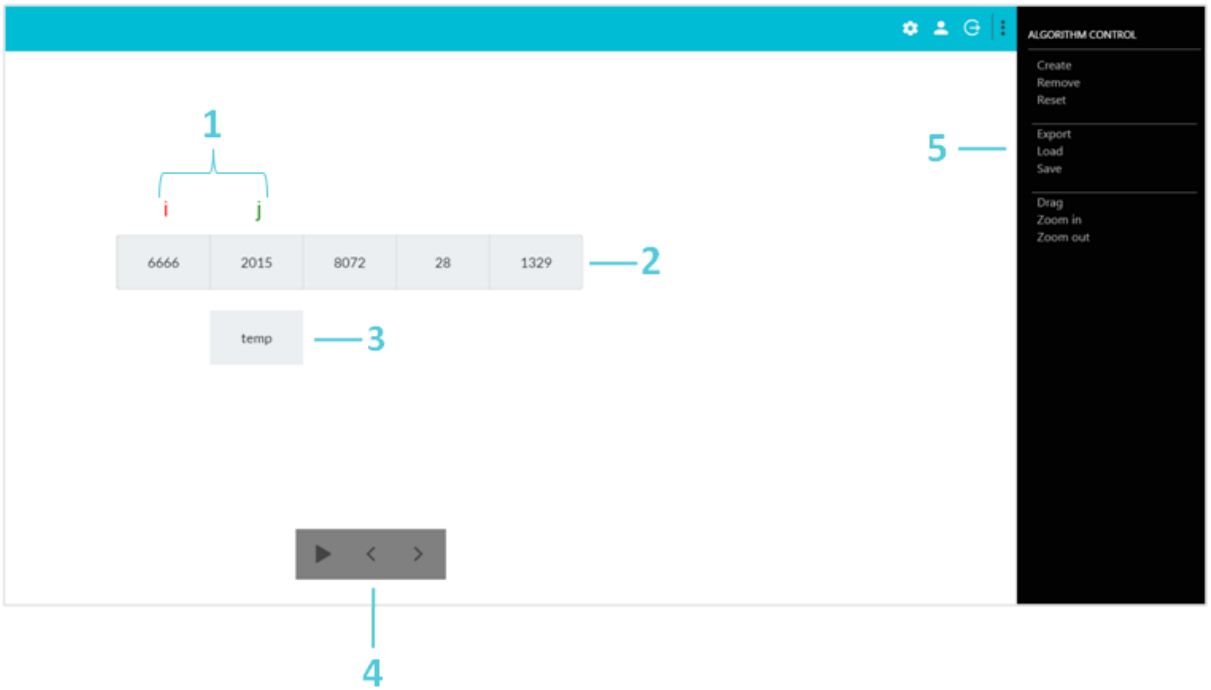

Figure 6.2: NetLuke2 - Visualization

The annotations in the figure point to the various elements of the library that we used in this implementation. (1) is an indicator to the **NL.Element.NLCounter** element while (2) shows the **NL.Element.NLArray** object. (3) points to the **NLTempCell** while (4) and (5) are references to the different menus being used in NetLuke 2.

Chapter 7

Conclusion

The NetLuke project was driven by our motivation to create a modern AV system that supports both, students and teachers in the complicated field of teaching and understanding algorithms and data structures. As the topic of algorithms is a dynamic field a new approach of teaching is required, that takes account of this fact.

Our tool allows a direct interaction with the subject matter. The user interaction combined with an expressive visualization, that is the result of dynamically highlighting the most important aspects during algorithm execution, can be seen as the key features of our learning software. As good learning materials should also create fun and raise the interest in the topic, we wanted our solution to be modern, good looking and easy to use. We took these requirements into account when designing the NetLuke system. We created a web platform that can be accessed from mobile devices, such as smartphones and tablets and from desktop computers. The client- server principle allows data persistence and nice features like continuing sessions on another device without loosing the state of the algorithm.

As we wanted our tool to be easy extensible we created the module concept. The idea behind the module concept was to allow developers to reuse code from the old LUCAS system for the Java side while using JavaScript for creating the visualization on the client side. We implemented a core library that provides basic functionality and data structures in order to ease the development process.

Although the concept of the first NetLuke prototype was well thought out, we uncovered some minor and major deficiencies that we needed to take care of. The result of our deliberations is an enhanced version of NetLuke, that features following improvements:

1. A new JavaScript library, called “NL.js“, that contains graphical building blocks and basic functionality for the client side.
2. A new User Interface that is interconnected with the library and provides the same user experience on all device categories. The new UI removes the restriction of being required to use the “NetLuke App“ when using a mobile device.
3. New debugging options for developers. The “NetLuke Logger “ helps developers to

find errors in their implementation more quickly.

The new library on the client side involves a bunch of advantages: (1) It significantly reduces the lines of code necessary for implementing the client side of a module. Algorithm implementations in NetLuke 2 feature a unique design, as all visualizations are based on the very same graphical building blocks (2). The interconnection of the library with the UI enables developers to directly integrate "Algorithm creation functions" in the main menu (3). The provisioning of basic building blocks effectively prevents code redundancy (4).

With the enhancements made in NetLuke 2 we managed to come a step closer to our vision of a perfect learning platform. Although we have to admit that the system is not ready to be used by students on a daily basis, as the library is currently only supporting sorting algorithms, we managed to introduce a new concept that is capable of reducing the amount of time necessary to create an algorithm visualization to a minimum. The idea of a graphical library that is able to illustrate the execution of every kind of algorithm could be very useful in other projects as well.

Chapter 8

Future Work

This chapter can be seen as a summary of our thoughts concerning the future of the NetLuke Project. Although many enhancements and improvements have been made to the system, it is not yet finished. Section 8.1 will therefore contain a list of steps that are required in order to make the system ready to be used by students as a learning platform on a daily basis. Section 8.2 is about further enhancements that could be implemented without much effort.

8.1 Unresolved Issues

The introduction of the new client side library simplifies module development and helps developers to save time. In the current implementation stage however, the library only supports sorting algorithms. In order to make the system ready to be used by students it is required to add support for the remaining kinds of algorithms as well.

Documentation

As the concept of the library, that is highly interconnected with the User Interface is not trivial, we plan to provide a detailed documentation for future developers. This documentation, that will contain information about the used JavaScript techniques, how to create "Konva " objects and the connection with UI, will be published on the project page[1]. This documentation will also include the "development guide" that is part of this thesis and shows how to use the provided components in order to create an appealing algorithm visualization.

Extending the Library

As we concentrated our effort in designing the architecture and generating the library concept, it was not our highest priority to implement support for various different algorithms in the library. However this library extension is necessary in order to support graph, tree and heap algorithms as well. Although not trivial, we are confident that this can and will be done by future developers based on our documentation. Extending the library consists

[1]The Project page is located at http://www.wst.univie.ac.at/workgroups/netluke/index.php

of various steps. Initially the developer needs to create the graphical elements required for the algorithm visualization using the KonvaJs framework. Afterwards the functionality for the graphical objects need to be implemented. This step is crucial as the library needs to be "complete". This means that one must be able to create a visualization for every algorithm of the same kind with just using the provided functionality. This support is crucial for potential module developers as the new concept only allows the usage of graphical elements that are provided by the library.

Creating Modules

Together with the extension of the library we hope that future developer will begin to increase the amount of supported algorithms. To become a relevant part of the students examination preparation planning, the system needs to support a variety of different algorithms. This comprises sorting algorithms like "quicksort" or "radix sort" as well as graph algorithms like "Dijkstra's algorithm" or "Kruskal's algorithm".

UI

The new UI is based on IONIC and Angular.JS. It features a responsive layout and is therefore capable of providing the same user experience on every device category. Although all menus and functionality are implemented, we did not invest much time in order to create an attractive theme. Future developers may create a more expressive theme using other colours than the provided white and blue, that is the IONIC standard theme.

8.2 Future Enhancements

App Creation

With NetLuke 2 we removed the requirement of an App to use the platform when using a mobile device. Some people however prefer the usage of a dedicated app over a browser. As the client side is now based on the IONIC framework a new NetLuke app can be implemented without much effort. IONIC has been originally developed in order to create mobile hybrid apps. Together with a "wrapper " like Apache Corodva it is quite easy to create a native app for IOS and Android based on the already existing web project. Apache Cordova is a framework that allows to generate native apps out of HTML5 code. The framework allows the usage of native device features via JavaScript [38]. An app created that way would be a suitable replacement for the original NetLuke app.

Premium Accounts

Currently the database is only used for storing the login credentials and the saved instances. A future version of NetLuke could feature an account system that allows the usage of different account types. It would be possible to create some "public algorithms" that are free to use by everyone and "premium algorithms" that can only be viewed when having a premium account. Students and employees of the university could automatically be granted premium access while external people would have to pay a little fee to get access to more complicated modules.

Bibliography

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Fourth Edition*, volume 4. Addison-wesley Reading, 2005.

[2] Ronald Baecker. Sorting out sorting: A case study of software visualization for teaching computer science. *Software Visualization: Programming as a Multimedia Experience*, 1:369–381, 1998.

[3] Ricardo A. Baeza-Yates. Teaching algorithms. *SIGACT News*, 26(4):51–59, December 1995.

[4] Jon Louis Bentley. *Programming pearls*. Addison-Wesley Professional, 2000.

[5] Stasko John T Byrne M. Dwyer, Catrambone Richard. Do algorithm animations aid learning? *Technical Report GIT-GVU-96-18*, 1996.

[6] CHRISTOPHER D. HUNDHAUSEN and SARAH A. DOUGLAS and JOHN T. STASKO. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259 – 290, 2002.

[7] Drifty Co. Ionic documentation overview, January 2015.

[8] Nils Faltin. Algorithmen lernen mit interaktiven visualisierungen. *INFOS*, 1:87–96, 2001.

[9] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, 2011.

[10] Rudolf Fleischer and Luděk Kučera. Algorithm animation for teaching. In *Software Visualization*, pages 113–128. Springer, 2002.

[11] Adam Freeman. *Pro AngularJS*. Apress, 2014.

[12] Wilbert O Galitz. *The essential guide to user interface design: an introduction to GUI design principles and techniques*. John Wiley & Sons, 2007.

[13] Prenner Georg. Netluke - Play with Algorithms - Part I. Master's thesis, University of Vienna, Austria, Vienna, Universitaetsring 1, May 2013.

[14] Peter Gloor. Animated algorithms. In *Elements of Hypermedia Design: Techniques for Navigation & Visualization in Cyberspace*, pages 235–241. Springer, 1997.

[15] Jean Greyling. Developing a set of requirements for algorithm animation systems. *International Journal of Computing and ICT Research*, page 83, 2009.

[16] Ilya Grigorik. Http client hints. *Network Working Group,Internet-Draft*, 2014.

[17] Steven Hansen, N. Hari Narayanan, and Mary Hegarty. Designing educationally effective algorithm visualizations. *J. Vis. Lang. Comput.*, 13(3):291–317, 2002.

[18] D. Harel and Y.A. Feldman. *Algorithmics: The Spirit of Computing*. Addison Wesley, 2004.

[19] Les Hazlewood. Application security with apache shiro, March 2011.

[20] Christopher Hundhausen and Sarah Douglas. Using visualizations to learn algorithms: Should students construct their own, or view an expert's? In *Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)*, VL '00, pages 21–, Washington, DC, USA, 2000. IEEE Computer Society.

[21] RobertW Janson. *Beginning Java with WebSphere*. Apress, 2013.

[22] COLLEEN KEHOE, JOHN STASKO, and ASHLEY TAYLOR. Rethinking the evaluation of algorithm animations as learning aids: an observational study. *International Journal of Human-Computer Studies*, 54(2):265 – 284, 2001.

[23] Suntae Kim and Bingu Shim. Mdt: Tool support for measuring logical coupling. In *Proceedings of ICCA*, pages 24–27, 2013.

[24] Anton Lavrenov. Konva - javascript 2d canvas framework, May 2015.

[25] Andrea W Lawrence, Albert M Badre, and John T Stasko. Empirically evaluating the use of animations to teach algorithms. In *Visual Languages, 1994. Proceedings., IEEE Symposium on*, pages 48–54. IEEE, 1994.

[26] Jon Loeliger. *Versionskontrolle mit Git*. O'Reilly Germany, 2009.

[27] Richard K. Lowe. Animation and learning: Value for money? In D. Jonas-Dwyer & R. Phillips R. Atkinson, C. McBeath, editor, *Beyond the comfort zone: Proceedings of the 21st ASCILITE Conference*, ASCILITE '04, pages 558–561, Perth, Australia, 2004. AJET.

[28] David Marginian and Mike Wilson. Direct web remoting, December 2014.

[29] Paul Mulholland and Marc Eisenstadt. Using software to teach computer programming: Past, present and future. *Software Visualization: Programming as a multimedia experience*, pages 399–408, 1998.

[30] W3C Recommendation. Html5 -a vocabulary and associated apis for html and xhtml, October 2014.

[31] Eric Rowell. Kineticjs, January 2012.

[32] Erich Schikuta and Sylvia Schikuta. Lucas - an interactive visualization system supporting teaching of algorithms and data structures. In George Siemens and Catherine Fulford, editors, *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2009*, pages 3776–3781, Honolulu, HI, USA, June 2009. AACE.

[33] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[34] John Stasko, Albert Badre, and Clayton Lewis. Do algorithm animations assist learning?: An empirical study and analysis. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, pages 61–66, 1993.

[35] M. Eduard Tudoreanu and Eileen Kraemer. Balanced cognitive load significantly improves the effectiveness of algorithm animation as a problem-solving tool. *Journal of Visual Languages & Computing*, 19(5):598 – 616, 2008.

[36] BARBARA TVERSKY, JULIE BAUER MORRISON, and MIREILLE BETRANCOURT. Animation: can it facilitate? *International Journal of Human-Computer Studies*, 57(4):247 – 262, 2002.

[37] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: Beyond layers. In *Proceedings of the 2007 International Conference on Dynamic Languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007*, ICDL '07, pages 143–156, New York, NY, USA, 2007. ACM.

[38] J.M. Wargo. *Apache Cordova 4 Programming*. Mobile Programming. Addison Wesley, 2015.

List of Figures

List of Tables

Listings

Appendix A

Abstract

A.1 English

Obtaining a fundamental knowledge of algorithms and data structures is necessary for nowadays students of computer science in order to analyze and refurbish existing projects or to find new, innovative solutions. "Classic" learning materials like textbooks often fail to explain the dynamic behavior that comes with algorithm execution. For this reason my colleague Georg Prenner and I developed a so called “Algorithm Visualization“ (AV) tool which is capable of dynamically showing every step within algorithm execution. We wanted to create a modern, effective and "easy to use" learning platform for students, which is educational effective. The project was called NetLuke and was supervised by Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta. Major aims were the usage of modern HTML5 technology, the easy extensibility by other developers and a well-designed user-interface in order to make the system convenient to use. In the first part of the thesis (NETLUKE Play With Algorithms - Part I) written by Georg Prenner, the technical concept, the general ideas and the implementation of the first prototype are described.

This is the second part of the thesis called “NETLUKE Play With Algorithms - Part II“. This part begins with a theoretical chapter that aims to answer the question of the educational effectiveness and describes the mistakes made by previous AV solutions. This chapter is followed by a description of our prototype implementation and the flaws and deficiencies we uncovered. Since the first part of this thesis has been released over two years ago we decided to refurbish the original design based on new technologies and advances that occurred during this period. I will describe this new design, including our ideas and concepts. A chapter will focus on the theories and ideas behind algorithm development for NetLuke 2 as this is much easier than before. This chapter is followed by a development guide based on the “InsertionSort“ algorithm. Finally I will provide ideas for future work on the NetLuke AV system.

A.2 Deutsch

Die Aneignung eines fundierten Wissens über Algorithmen und Datenstrukturen ist essentiell für Studenten der heutigen Computerwissenschaften. Ohne ein solches Wissen ist es nicht möglich bestehende Projekte zu analysieren oder neue, innovative Ansätze zu entwickeln. Aus diesem Grund haben mein Kollege Georg Prenner und Ich ein Tool entwickelt welches in der Lages ist die einzelnen Schritte innerhalb der Ausführung von Algorithmen dynamisch darzustellen. Es war dabei unser Ziel eine moderne, effektive und visuell ansprechende Lernplattform für Studenten zu entwickeln. Das Projekt, welches unter der Leitung von Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta stand, wurde NetLuke getauft. Das System sollte auf moderner HTML5 Technologie aufbauen, ein gut designtes Benutzerinterface besitzen und leicht zu erweitern sein. Der erste Teil dieser Arbeit (NETLUKE Play With Algorithms - Part I), geschrieben von Georg Prenner, behandelt das technische Konzept, die generellen Ideen und die Implementierung des ersten Prototypen.

Dies ist der zweite Teil der Arbeit, "NETLUKE Play With Algorithms Part II " genannt. Er beginnt mit einem theoretischen Teil welcher der Frage nach der "Erzieherischen Effektivität" nachgeht. Weiters werden die Fehler früherer Anwendungen untersucht. Im nachfolgenden Kapitel wird auf die Implementierung unseres Prototypen und auf die Fehler die wir seitdem entdeckt haben, eingegangen. Aufgrund der Tatsache dass der erste Teil dieser Arbeit vor über 2 Jahren veröffentlich wurde, entschlossen wir uns dazu das Design basierend auf neuen Technologien und Entwicklungen zu überarbeiten. Dieses neue Design wird von mir beschrieben. Ebenso werde ich auf unsere Ideen und Konzepte zur Neuentwicklung eingehen. Ein anderes Kapitel beschreibt unser Konzept zur Entwicklung neuer Algorithmen für das System. Dies wurde in der neuen Version stark vereinfacht. Daran anschliessend folgt eine Anleitung für Entwickler welche auf dem "InsertionSort" Algorithmus basiert. Abgeschlossen wird die Arbeit durch meine Ideen bezüglich zukünftiger Entwicklungsmöglichkeiten rund um die NetLuke Plattform.

Appendix B

Curriculum Vitae

Alexander Rotheneder

Geboren am	2. September 1986
Geboren in	3400 Klosterneuburg
Nationalität	Österreich

Ausbildung

Bundesrealgymnasium Klosterneuburg

Matura	2005

Universität Wien

Bakkalaureatsstudium Wirtschaftsinformatik	2005 - 2013
Masterstudium Wirtschaftsinformatik	2013 - 2015

Publikationen

NetLuke: Web-Based Teaching of Algorithm and Data Structure Concepts Harnessing Mobile Environments
Georg Prenner and Alexander Rotheneder and Erich Schikuta, submitted to: Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services, iiWAS '14

Appendix C

Source Code Listings

C.1 StepChangeset Class

LISTING C.1: StepChangeset - Class

```
1  public class StepChangeSet implements Serializable {
2
3   private static final long serialVersionUID = 880848300948163743 1L;
4   private InsertionSortStep step;
5   private final StringBuilder XML_CHANGE_SET = new StringBuilder();
6   public StepChangeSet() {}
7
8   public StepChangeSet(final Element[] array, final InsertionSortStep
        step) {
9          XML_CHANGE_SET.append("<changeset>\n");
10         writeChanges(array,step);
11         this.step = step;
12  }
13
14  public void writeChanges(Element[] array, InsertionSortStep step) {
15  this.XML_CHANGE_SET
16  .append(" <i>"+step.getI()+"</i>\n")
17  .append(" <j>"+step.getJ()+"</j>\n")
18  .append(" <temp>"+step.getTemp().getValue()+"</temp>\n")
19  .append(" <temp_pos>"+step.getJ()+"</temp_pos>\n");
20
21         if (step.isSwaped()) {
22         this.XML_CHANGE_SET
23         .append(" <swaped>\n")
24         .append(" <swaped_pos>"+step.getSwap_pos()+"</swaped_pos>")
25         .append(step.getSwap_Element().getAsXML())
26         .append(" </swaped>\n");
27         }
28
29         if (step.isSwap_temp()) {
```

```
30          this.XML_CHANGE_SET
31          .append(" <swaped_temp>\n")
32          .append(" <outer_swap_pos>"+step.getOuter_swap_pos()+"</
               outer_swap_pos>")
33          .append(step.getOuter_swap().getAsXML())
34          .append(" </swaped_temp>\n");
35          }
36          this.XML_CHANGE_SET.append("</changeset>\n");
37
38  }
39
40   public String getChangeSet() {
41          return this.XML_CHANGE_SET.toString();
42   }
43
44   public void print() {
45          System.out.println(this.XML_CHANGE_SET.toString());
46   }
47
48   public InsertionSortStep getStep() {
49          return this.step;
50   }
51  }
```

C.2 Back & BackSort

LISTING C.2: Back & BackSort - Functions

```
@NetLukeMethod
public NLResponseContent back() {
final String result=this.backSort();
 if (result!=null) {
  return new NLResult(result);
 }
 else {
  return NLErrorType.EMPTY.get();
 }
}

private String backSort() {
 if (this.steps != null && !this.steps.isEmpty() && this.steps.size()
     >= 1) {
  final InsertionSortStep step = this.steps.get(this.steps.size() - 1
     );
  this.i = step.getI();
  this.j = step.getJ();
  this.tempE = step.getTemp();
   if (step.isSwaped()) {
        this.sort_array[step.getSwap_pos()]=step.getSwap_Element();
   }
   if (step.isSwap_temp()) {
        this.sort_array[step.getOuter_swap_pos()]=step.getOuter_swap()
           ;
   }
  this.sorted=false;
  this.temp=step.getTemp_value();
  this.firstrun=step.isFirstrun();
  this.steps.remove(this.steps.size() - 1);
  final String result = this.changesets.get(changesets.size() -
  1).getChangeSet();
  this.changesets.remove(this.changesets.size() - 1);
  return result;
 }
return null;
}
```

C.3 The applyCurrentChanges Function

LISTING C.3: JavaScript - applyCurrentChanges Function

```
applyCurrentChanges :function(changeset) {
 var self = this;
 var checker = $(changeset).find("result").text();
  if (checker ==="false") {
      console.log("ENDE");
      self.nextDone = true;
  }
  else {
      self.lastouter=false;
      var i = parseFloat(changeset.find("i").text()),
      j = parseFloat(changeset.find("j").text()),
      temp = parseFloat($(changeset).find("temp").text()),
      temp_pos= parseFloat($(changeset).find("temp_pos").text()),
      swap_pos=parseFloat($(changeset).find("swaped_pos").text()),
      outer_swap_pos=parseFloat($(changeset).find("outer_swap_pos").
         text()),
      swap_value = parseFloat($(changeset).find("element").text());

      var tempcelltext = self.arr.getCell(temp_pos).getText();
      if (self.newrun) {
           self.arr.setCellText(temp_pos,temp);
      }

      var todo = function() {
       function f1() {
        self.swapBox.setPosition(temp_pos, {
         duration :0.3,
         onFinish :f2
                 });
                }

                function f2() {
         self.swapBox.showText();
         self.counterI.setPosition(i,{});
         self.counterJ.setPosition(j,{});
          if (self.newrun) {
              self.swapBox.takefromArray(temp_pos,{
                duration: 1,
                onFinish :f3
                           });
               self.currentTemp = temp;
               self.newrun = false;
          }
                  else {f3.call();
                  }
```

```
45          }
46
47          function f3() {
48           self.arr.setCellText(temp_pos,tempcelltext);
49            if (!isNaN(swap_pos)) {
50                          f4.call();
51                  }
52
53            if (!isNaN(outer_swap_pos)) {
54                f5.call();
55                  }
56          }
57
58          function f4() {
59                   self.arr.moveCell(i,i+1,{
60                     duration :1,
61                         onFinish :f6
62                         });
63           self.arr.showCell(i+1);
64                  self.arr.hideCell(i);
65          }
66
67                 function f5() {
68           var a = self.swapBox.getText();
69           self.arr.setCellText(i+1,a);
70           self.arr.showCell(i+1);
71           self.swapBox.hideText();
72           self.lastouter=true;
73           self.newrun = true;
74           f6.call();
75                 }
76
77                 function f6() {
78            self.lastback=false;
79            self.isAnimating=false;
80            NL.Player.enable('nextEntry');
81            NL.Player.show('nextEntry');
82            NL.Player.enable('backEntry');
83                 }
84
85          if (!self.isAnimating) {
86           self.isAnimating = true;
87           f1.call();
88      }
89
90          todo();
91   }
92  }
```

C.4 The applyBackChanges Function

LISTING C.4: JavaScript - applyBackChanges Function

```
applyBackChanges :function(changeset) {
 var self = this;
 // Get variables from changeset
 // Same code as in applyCurrentChanges()
 var temp_text = self.swapBox.getText();
 self.swapBox.setPosition(temp_pos,{});
 self.swapBox.setText(temp);
 self.swapBox.showText();
 self.counterI.setPosition(i,{});
 self.counterJ.setPosition(j,{});

  if (self.arr.getCell(i+2) != undefined) {
     self.arr.showCell(i+2);
  }

 if (!isNaN(swap_pos)) {
     self.arr.moveCell(i+1,i,{
     duration :0.5,
     onFinish :function () {
        NL.Player.enable('backEntry');
        NL.Player.enable('nextEntry');
         if (parseFloat(temp_text) != temp) {
             self.arr.setCellText(j+1,temp_text);
         }
          }
     });
    self.arr.showCell(i);
    self.arr.hideCell(i+1);
 }

 if (!isNaN(outer_swap_pos)) {
      if (self.arr.getCell(j+1)!=undefined) {
           self.arr.showCell(j+1);
      }
      self.swapBox.showText();
      NL.Player.enable('backEntry');
      NL.Player.enable('nextEntry');
 }
}
```