universität
wien

MASTERARBEIT

Titel der Masterarbeit

NETLUKE

Play with Algorithms - Part I

verfasst von

Georg Prenner, BSc

angestrebter akademischer Grad
Diplom-Ingenieur (Dipl.-Ing)

Wien, 2013

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, ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel, verfasst zu haben. Die aus fremden Quellen übernommenen, direkten oder indirekt 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, Mai 2013

Georg PRENNER

About this thesis

The NetLuke project presented in this thesis is a cooperative teamwork between the authors Georg Prenner and Alexander Rotheneder. The project was developed and is actively maintained by both. Due to the extent of the project, the thesis is separated in two parts. Despite the fact that the core chapters of the first part of *NetLuke – Play With Algorithms* are written by myself, the authors decided to use active voice in this thesis. I chose to use "we" to underline the cooperative character throughout the development process.

The second part is written by Alexander Rotheneder at a later date, because of reasons not relevant to this thesis. The content of Part II is not covered in this work, but is referenced repeatedly. Upon request, the contents of both parts can be merged to a single document.

Contents

Chapter 1

Introduction

1.1 Motivation

> "Algorithms are the 'stuff' of computer science: they are central objects of study in many, if not most, areas of the field." [36]

Algorithms shape the world of computer science. Each and every solution to a computable problem is achieved by a more or less effective problem-solving procedure or as Christopher Hundhausen puts it: Algorithms are the " [...] building blocks of computer science " [14]. In the majority of cases, algorithmic calculations within programs, including their defined instructions, remain hidden for program users and sometimes even for software developers. However, algorithmic building blocks, designs and intentions matter. Without knowledge of fundamental algorithms and data structures, it is not possible to evaluate existing work in a precise manner, nor will one be able to design new innovative solutions. "[...] In order to use a computer properly, it is important to acquire a good understanding of the structural relationships present within data, and of the techniques for representing and manipulating such structure within a computer" [17]. Of course this statement from Donald E. Knuth in 1973 is not valid for the usage of computers today. Nonetheless, it certainly implies that a software engineer needs to know about data structures, how they work and also which particular algorithm suits best to find a solution for a given problem. In that sense, it is not surprising that academic courses cover algorithms and data structures. Their subject matters are mandatory and essential for students of computer science. Even students of business economics, geography, mathematics and various other fields find themselves confronted with one or the other algorithm. Albeit, considerable time and effort is required in learning complicated matters, such as dynamic algorithms and data structures.

Two groups of actors are involved in the **process of learning**: Teachers and students. The first group needs to rely on its ability to elaborate a subject matter as interesting, as precise and as effective as possible. In the optimal case, every student is focused on the discussed topic and is able to understand what the lecturer is unveiling. In reality students struggle to understand these topics. Acquiring knowledge about algorithms can be a tough quest. From our own personal experience, we can say, that it is difficult

to figure out the workings of procedures solely from lectures. Provided, but sometimes excessively **simplified examples rarely help** either. Even though textbooks give the opportunity to catch the meaning of an algorithm, neat examples always shorten the process of understanding. Therefore, we see exercising examples as "the way to go".

Imagine a student who is late to Computer Science 2 (CS2) class. Assume he did not understand a word the lecturer was saying. In preparation for his next exam he will study his course book, but is not able to fully comprehend the subject matter. Maybe this student attends a free online course on YouTube or suchlike platforms[1] with exercises and in-depth explanations which actually help him fill the learning deficit. Students are late on a daily basis, but what this story should tell us is, that there are many different ways of pursuing knowledge aside from textbooks or lectures in class. Yet another approach for learning computer science topics, is interaction with the "learning object" itself. With *NetLuke*, the authors of this thesis want to provide an **easily accessible environment**, which visualizes self created algorithmic **examples, which can be modified and extended within use**. The high level reason for developing NetLuke, was the intention to provide a comprehensive software platform that enhances students learning experience. Therefore, NetLuke should rather extend traditional "low tech" learning materials [14] instead of displacing them. Thus we looked at learning materials in computer science and investigated their value. Students are most likely confronted with common/popular materials like ...

- ... textbooks and professional/expert literature which both provide information about algorithm structure, theoretical background and application examples,
- ... slide shows which present sketches, drawings and other illustrations about structure, design and algorithmic execution,
- ... examples, exercises and sometimes pieces of pseudo code, which are either too simple in order to conceive examples with a higher degree of complexity, or too difficult to understand them in the first place.

In the case of highly dynamic topics, the descriptive representations listed above, **might not fit the needs** of every particular student. For instance, those who number among the visual - interactive learning type, or those who need to reiterate over a range of examples before fully comprising the subject matter, will have their difficulties. As numerous sources argue (see e.g. [20, 35, 41]) **animations and visualizations** of algorithms and data structures can help to **overcome the shortcomings** of static representations of topics. Our motivation directly stems from this viewpoint. The needs of students in their self study efforts are determining factors in our project, thus we want to address potential gaps in the learning process by ...

- ... giving students the possibility to create examples with a seamless variation in difficulty, according to their own needs.

[1] YouTube features several free high quality educational lectures in the fields of computer science and mathematics. Famous contributors are the MIT, UCBerkeley or Stanford University. See http://www.youtube.com/education?category=University for further information. Another exciting source is the "Harvard Open Learning Initiative" http://www.extension.harvard.edu/open-learning-initiative

- ... providing the ability to interact with algorithmic structures.
- ... visualizing algorithmic object examples in a precise and visual appealing way.
- ... creating a feature rich environment that allows step-by-step control over algorithms and data structures.
- ... adding comfort and a high level of flexibility in usage.
- ... adding a certain level of enjoyment to learning tasks.

Learning and teaching needs to be productive. The less time a lecturer is urged to explain a topic, the better. The less time a student spends on learning before gaining profound knowledge the better. Efficiency paired with a deeper level of comprehension leads to better results. Over the last decade web based eLearning platforms emerged (see e.g. [1, p. 113]) that helped students and lectures to organize work outside of classrooms. More recent developments try to support students by providing additional learning artifacts like advanced examples, quizzes, simulations or even virtual class rooms.

This modern **concept of eLearning can be adopted** and modified to create an environment that supports students in learning computer algorithms through visualization and direct interaction. Our concept goes beyond traditional learning software that is scattered around the web, because it does exhibit a much **higher level of flexibility** concerning algorithm content, usability and value for students and lecturers. With our perception of a new eLearning software in mind and the vision of our professor Erich Schikuta (see Schikuta [35]) of a comprehensive, modular and web based solution, we developed and implemented a prototype which comes very close to our vision of a modern and dynamic algorithm visualization (AV) system.

1.2 Problem Definition

The idea of having an AV system supporting learners is not new. However, most of the AV tools we came across in our research (see Section 2.3 and 2.4 in Chapter 2) either focused on the procedures of the algorithm (low level details) or lacked the ability to interact intuitively with them (static examples). Many AV solutions completely focus on the topics and forget that the body of knowledge is learned over time (learning curve). In addition to troublesome/insufficient educational quality, we observed technical problems leading to a degraded learning experience. In the process of analyzing AV solutions we could identify the following top level problems in current solutions:

1.2.1 Educational effectiveness

AV systems need to be effective, but what makes them effective? For instance, if the visualization aspects of the step-by-step operation is combined with an **interactive viewpoint**. We are convinced that a user should be able to specify the input data set, which composes the initial algorithmic problem. For us it is crucial that there is no narrow limit in creating own examples. Hundhausen et al. [14] and T. Muldner [25] support this claim. In our perception users must have control over each important step the algorithm runs through, because control over specific situations lead to a more in depth understanding. Yet another advantage of fine grained control is, that teachers have the

opportunity to demonstrate operations of algorithms or data structures directly without referring to external resources such as static images. Widely used descriptive approaches, like highlighted pseudo code are not considered (implicitly) educational effective, because they presuppose knowledge. Instead students should be [15] " [...] actively engaged in the process of learning algorithms" by the AV system.

Teaching algorithms and data structures effectively is particular difficult if the subject matter reaches a higher level of complexity. To overcome this problem, Lowe and Muldner [20, 25] point out and suggest, that animations within a given visualization need to be highly expressive (see next Section for further discussion). We observed that increases in problem complexity are accompanied by decreases in educational effectiveness. This is partly, due to missing theoretical frameworks [2] and deficits in implementations [38, p. 152].

1.2.2 Explanatory features

Some tools (see Section 2.4 for an evaluation of implementations) focus on the very basics if it comes to algorithm interaction. Take a binary search tree for example: Elements can be inserted or removed from the data structure while the representation of the tree changes in depth and/or width. What if you want to know how the tree looked like four steps ago before removing or inserting a specific element? Is it possible to save an instance of your example including its interaction history for further use or presentation? Are algorithm state changes emphasized by color or highlighted by visual effects like transitions and animations? A combination of these features is rather difficult to find in existing AV solutions. Our understanding is, that **cognitive processes can benefit from purposeful use of animations** of algorithm operations. Various sources confirm (see e.g. Greyling [10, p. 84], Lowe [20, p. 2] and Hansen [41, p. 5]) this claim. A user should exactly know "what is going on" during algorithm execution. Some tools choose to display operational steps in source- or pseudo code. As argued before, this concept implies that a learner already knows how a particular algorithm works in the first place. Therefore, pseudo code is useful to understand actual source code in a programming language. It helps to narrow down algorithmic steps in existing implementations. However, it does not help a student in learning new algorithmic concepts. It does not "paint the big picture". That is why we believe that additional explanatory features should not be abstract, instead clearly show what is going on by presenting visual representations.

1.2.3 User experience

In general, it is annoying if one needs to study tutorials or help sections of software, before starting to work with it in the first place. The problem is inherent to large software projects with complex user interfaces. AV tools should not be complicated and should allow users to start exploring algorithms right away. An eLearning tool needs to be easy to access, right from the start, in order to fulfill its purpose. A Graphical User Interface (further referred to as UI or GUI) must follow a logic that **supports and aids the user** during his learning tasks. A simple list of doubtful buttons scattered around on the screen creates confusion and hostility towards the system. In addition, existing solutions are predominantly based on standard Java toolkits with "rusty old GUIs that

do not fit into today's shiny world of smartphone interfaces". As a matter of fact, the current generation of users has higher expectations concerning UIs. Even though some of the reviewed tools exhibit a certain "academic appeal" (meaning the reduction of the interface to its bare minimum), we get the impression, that **UI design falls short** most of the time. A general problem in software development is, that its creators sometimes ignore the fact, that visual appealing presentations of problems lead to a higher level of motivation and commitment [42, p. 71]. "Looks" and a good structure in UI design do matter even for simple learning tools. A **user interface absolutely needs to be consistent** throughout presented topics. Although we do consider the relatively high age of most tools and their underlying programming frameworks, we get the impression that UI design did not have priority in their development process.

1.3 General Project Goals

Our vision is to create a feature rich, good looking, modern, and robust system environment which **supports interactive learning** of algorithms and data structures. Our targeted group of users divide into teachers, course instructors and lecturers (teachers). The second group consists of computer science students and other learners (users). The third group are system contributors, which we further refer to as "plugin or module developers". Each of those groups have individual requirements concerning the NetLuke system. Teachers should be able to use NetLuke for demonstration purposes. Users should be given the possibility to practice operations of computer algorithms and data structures with **self generated examples** varying in difficulty or extent. Users should have a maximum of comfort in using the application. This implies the existence of functionality that supports saving and loading of examples, exporting of images for other purposes like demonstrations, panning and zooming of the drawing area as well as robustness against errors.

Actual **Visualizations** of algorithm implementations and their corresponding control options need to be **intuitive, educationally effective and supportive**. Module developers need an environment that lets them focus on the core tasks e.g. extending NetLuke with additional topics. In the context of NetLuke, this means plugin development, which involves algorithm or data structure logic and visualization, surrounded by a plugin specific control system. The architectural design of NetLuke needs to incorporate plugin development which is as unattached/independent from the system as possible. For instance, 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 use interfaces provided by the system. Therefore, NetLuke should provide a relatively lightweight application programming interface (API) and libraries that facilitate creation of well implemented modules.

As stated before, the project aims to be feature rich and developer friendly, which raises architectural and technical questions, like: Which technologies suit our requirements (see Platform Requirements in Section 3.2) best? One very critical aspect in this regard is **compatibility with diverse operating system platforms**. We want to create a web platform, which is designed and built around its latest client technologies to support as many platforms as possible. Web browser based applications are platform independent by principle, which allows us to target desktop environments as well as mobile devices such

as tablet computers and smartphones. For us, it is absolutely necessary to go beyond the "classic" PC/Desktop paradigm, not only to reach a growing number of users which have the desire to be location independent, but to break the barrier of bulky user interfaces which do not work in the mobile world [42, p. 170]. With mobile devices in mind, we are forced to put considerable effort into interface design (see Section 3.3.3 for UI requirements) and implementation details. Another good reason for mobile device support is, that **touch based interaction** makes perfect sense for our type of application, as it supports the intuitive "game-like" interactive part of the tool. Moving to a web based solution certainly has its disadvantages (see Section 3.1 and Section 3.5 for discussion), but opens up new architectural and technological perspectives that support our wish for a modern kind of AV platform. Altogether, our vision of a **user centered, non monolithic approach** is reflected by NetLuke's architectural design (Chapter 5) and its set of features. System design needs to be as flexible as possible, support the whole range of existing computer platforms, and give future developers the opportunity to create plugins without too much effort.

The goal of this thesis is to present and describe the project implementation, by discussing its major ideas, concepts and technical solutions. Its purpose is to illustrate the development process from prototype design to implementation. Therefore, it focuses on the documentation of architecture and major components of NetLuke.

General Concepts

In order to meet the outlined goals, we do not only have to undertake a requirements analysis, but have to make use of high level concepts in software engineering. The following points represent determining factors in the design phase of NetLuke. Further in-depth analysis of the projects design and architecture will be undertaken in the Chapters 4 to 6. Project requirements are defined in Chapter 3.

1. Use of concepts (design patterns, programming techniques) which strengthen modularity and information hiding in order to enforce separation of concerns and to achieve fundamental reduction of system complexity.
2. Creation of a multi-layered & distributed architecture which follows design patterns similar to a model view controller (MVC) approach.
3. Multi-technological approach on top of the architectural concept which connects different programming environments with each other to enable web browser based visualization and server based computation in a 2-Tier manner.
4. Loosely coupled and interchangeable system components to allow easy extensibility and project maintenance.
5. Code reuse – Usage of existing code or at least fragments from related projects[2] for faster plugin development. This point also includes the creation of components which can be shared/re-used within NetLuke.

[2] As explained in Section 2.3 NetLuke is the successor of LUCAS which featured several algorithm implementations in Java. Logic which is encapsulated within those modules will be adopted for future NetLuke plugins.

6. Comprehensive investigation, analysis and evaluation of existing technologies[3] to facilitate the proposed architecture.

1.4 Project Structure

Because NetLuke is an aggregate software project, consisting of sub-projects, we decided to provide an overview over the involved[4] project pieces on the highest level. With the introduction of sub-projects, we want to facilitate stepwise and project independent development of components. This allows easier maintenance and separation of concerns throughout the development phase, while adding little complexity to the development process of modules (see Chapter 4). The following Eclipse projects were created during the development process:

- **NetLuke** – is the main project. It encompasses the server environment and the client components, which define NetLuke's overall feature set. Custom modules are deployed within this project. It requires an J2EE Application Server as the primary runtime environment. Chapter 5 discusses the server environment extensively.
- **NetLukeCore** – constitutes the core library (basic building blocks) needed for module development (see Section 4.2 for further discussion).
- **NetLukeAndroid** – is dedicated to the Android Web Application. It can be considered as a semi-browser-based runtime environment with special constraints. This project requires the Google ADT plugin[5] for Eclipse IDE. Section 6.6 covers the architecture of our Android application.
- **NetLukeTest** – acts as a manual and automatic testing environment for modules. Developers can analyze their implementations to discover bugs, to optimize components and to find logical errors in the source code. This project is covered in Part II of this thesis.
- **NetLukeUML** – features UML diagrams of modules and server components. Developers can use this project for basic to advanced modeling tasks and illustration purposes. This project requires additional Eclipse modeling plugins [6].
- **NetLukeProject** – represents the future project site, where source code and binary releases can be downloaded from. It will also feature a Quickstart user guide as well as a developer tutorial derived from Part II.
- **NetLukeStandalone** – This version of NetLuke does not require a separate J2EE Application Server like the normal version, instead it can be executed directly

[3] For instance: Server - Client communication and interface creation, visualization libraries, security frameworks and related modules

[4] All listed projects will be available for download through a project page hosted at the University of Vienna.

[5] http://developer.android.com/tools/sdk/eclipse-adt.html

[6] NetLukeUML requires the papyurs: (http://www.eclipse.org/papyrus/) plugin for Eclipse, modeling framework components (http://www.eclipse.org/modeling/) and the UML ObjectAID plugin: (http://www.objectaid.com/)

through a runnable JAR file (OS independent). We wanted to allow users inexperienced with configuration and deployment tasks in application servers to get started with a few clicks. However NetLukeStandalone is configurable regarding server port and log level.

- **TomcatEmbed** – is a required project for the standalone project. TomcatEmbed provides functionality and configuration options regarding the embedded version of Tomcat 7.

NetLuke ships with sample module implementations listed below. Their main purpose is to illustrate the feature set of NetLuke as well as capabilities current AV components offer. Again, detailed discussion is provided in Part II. The following modules are implemented and ready for use within the AV system:

- **HeapSort** – implements the heap sort algorithm based on the heap implementation in the NetLuke core library.
- **BubbleSort** – features a simple animated bubble sort algorithm.
- **BinaryTree** – like the heap sort module, this module uses a predefined binary tree data structure implementation of NetLuke core.
- **PrimAlgorithm** – this plugin was adopted and ported from the LUCAS project.
- **SelectionSort** – features an animated selection sort algorithm.

1.5 Thesis Structure

This Chapter covered the basic ideas behind NetLuke and what the project is all about. Teaching and understanding dynamic algorithms are difficult tasks which is why we are encouraged to contribute in this matter (Section 1.1). The following Section 1.2 outlined which problems are inherent in comparable projects but also in the field itself. We continued with primary concepts essential to our project goals (Section 1.3) and explained the structure of NetLuke (Section 1.4). We provide information regarding the most important terms and keywords used in this thesis in Section 1.6.

Chapter 2 introduces previous work and research undertaken in the field of Algorithm Visualization. Because principles, theories and philosophy in the field directly influenced our work, we provide a short introduction to key concepts of AV in Section 2.1. The evolution of the field is presented in Section 2.2. Over time many AV platforms emerged, each focusing on different aspects. We decided to illustrate three different approaches taken in actual AV system to point out individual advantages and disadvantages (Section 2.3). We compare and evaluate these examples (see Section 2.4) regarding technical and educational quality to stress the relevance of NetLuke.

Based on the analysis in Chapter 2, we introduce requirements in Chapter 3. To meet the project goals presented in Section 1.3, we distinguish between platform, developer and

system requirements. We start with considerations (Section 3.1) and difficulties which accompanied the requirement definition process. This leads directly to a threefold requirement structure beginning with the definition of platform requirements (Section 3.2). System requirements (Section 3.3) define the feature set of modules, the server system and the UI. While both types of requirements stem from the viewpoint of users, developer requirements (Section 3.4) incorporate the viewpoint of future module contributors. As requirements compete against each other, we analyze potential conflicts and solutions in Section 3.5.

Chapter 4 is dedicated to the architectural design of modules. We begin with an overview of what modules are and how they are developed (Section 4.1). We further discuss in detail our NetLuke core library – This major building block enables developers to create robust Java implementations – the business logic in our 2-Tier architecture (Section 4.2). Since NetLuke separates presentation from structure/behavior, we outline key functionality a Java module needs to incorporate in order to work in NetLuke. This includes data representation aspects as well as required module properties. We discuss this topic along with the assembly of server-side NetLuke modules in Section 4.3. This leads to the process of deployment and system integration of modules (Section 4.4). Module visualization and user control is subject to client-side (web browser) routines. An introduction to the architectural background of client-side JavaScript development is provided in the closing Section 4.5.

Chapter 5 focuses on the server realization of NetLuke. At the beginning, we will illustrate system components, which comprise the server-side application (Section 5.1), including component interaction. The next section covers the service layer of NetLuke, creating interfaces for JavaScript RPC and Web Service clients (see Section 5.2). We discuss how we decouple business logic from the service layer, making modules independent from the system. We continue with *NetLukeController* in Section 5.3, which handles/delegates RPC or Web Service calls to underlying modules. We explain how this component discovers and invokes implementing classes, methods and constructors during runtime. The component is central to NetLuke, as it manages module instances transparent to users and developers. The next Section 5.4 introduces the concept of module instance lifecycles and their shared state among system components. We continue with the user registration process (see Section 5.5) and how session and security is handled by NetLuke (Section 5.6). In the following Sections we focus on event handling through monitoring components (Section 5.7) and how the system can be configured (Section 5.8). The persistence layers' design and implementation is discussed in Section 5.9.

Chapter 6 focuses completely on the client architecture. Again, we begin with an overview over the components involved in Section 6.1. A detailed discussion about the User Interface covering desktop and mobile clients is presented in Section 6.2. The site structure explained in Section 6.3 provides information about where (web pages) users can interact with the system. Next (Section 6.4) is KineticJS the graphical library used in NetLuke. We explain why we decided to use it over other comparable solutions and why we consider KineticJS as optimal for NetLuke. In the following Section 6.5 we introduce our custom client side controlling component, which provides vital functionality to NetLuke. In the last Section 6.6 we briefly introduce the architecture of our Android application.

The discussion in Chapter 7 sums up our future plans and enhancements for the project. We investigate bugs, issues and errors within the system (Section 7.1) and continue with an analysis of the prototypes' architecture in Section 7.2. Upcoming and desired features for the platform are listed in Section 7.3 and 7.4.

We finish this thesis, with a Conclusion in Chapter 8. We summarize results and list our contributions to the field of AV.

1.6 Definition of Terms

Hundhausen et al. [15] describes **Software Visualization (SV)** as genuine systems which are " [...] designed to help members of programming teams do such things as improve system performance, comprehend the structure and evolution of software systems, and track down bugs". SV illustrates structure and behavior of software systems, which also applies to AV systems. According to [16] " [...] Structure is the visualization of static parts and relations of the system. Behaviour is the visualization of the program execution with real or abstract data".

According to J. Greyling [10, p. 329] **Algorithm Animation (AA)** " [...] is a form of technological support tool which encourages algorithm comprehension by visualizing algorithms in execution. Algorithm animation can potentially be utilized to support students while learning algorithms." In scientific literature [16] Algorithm Visualization is sometimes synonymously referred as AA, whereas we do not follow this terminology, instead we emphasize that animations maybe **part of an AV system**.

An **Algorithm Visualization (AV)** system comprises elements from Software Visualization (SV) and Algorithm Animation (AA). It is a subclass of SV [15, p. 2] with AA extensions. Together with the concept of **Information Visualization (IV)** [16, p. 9] and its goals: "[...] The goal of IV is to amplify cognition, that is, aid the understanding of some aspects of the data" AV systems are embedded into an educational context. Ideally a good AV system is capable of illustrating structure and dynamics which lay within an algorithm or a data structure in a manner, a learner can re-enact the results at hand. According to Moses [24] Algorithm visualizations can assist in the design of algorithms, in the debug process and of course in teaching algorithms to students and colleagues.

We use the term **NetLuke** to describe the overall Project itself, meaning all involved sub-projects and delivered artifacts. NetLuke is also the name of the J2EE web application project which provides all necessary functionality for client usage. We discuss its components thoroughly in Chapter 5. Section 1.4 explains the structure of NetLuke's sub-projects in detail.

The term **Desktop Paradigm** basically describes the typical PC setup consisting of a tower or laptop computer with medium-to-large sized screens. The term also relates to their **hardware capabilities** which expose a very high level of performance compared to mobile devices. In this thesis we consider desktop systems as relatively homogeneous not only in hardware, but also in software including operating systems. We use the term *paradigm* because inherent historical **significance of such systems is rapidly declining** [12, 26]. Consumer usage patterns heavily shift to mobile devices [23] which

creates a very heterogeneous environment for software developers. Therefore, we see a *paradigm shift* in device usage/development efforts towards mobile devices.

Mobile devices are either considered **mobile phones** or **tablet computers**, but (potentially) also any other devices with **touchscreen interaction capabilities**. Devices running with operating systems like Apple iOS, Google Android, or any other related OS are also considered *mobile* in this thesis. In addition to a desktop web browser version, NetLuke will be available as an web application (see Section 6.6 for further discussion) on iOS and Android devices.

A NetLuke **module** or a **plugin** is the aggregate of algorithm logic (core functionality) including necessary interfaces and the visualization components itself. In the last development phase, a plugin consists of two artifacts: (1) A Java Archive (JAR) file as well as a (2) JavaScript file which are both deployed within a compressed web-app (WAR) archive. See Figure 3.2 for an illustration of a plugin. The architecture of modules and how they are created is explained in Chapter 4.

Instances of module classes are called **algorithm instances** or just **instances**. An instance incorporates a visualization in JavaScript on the client and an according Java object on the server-side. We use the term instance synonymously for both. However, when we speak of an instance running on the server, it does not necessarily involve a visualization on the client, as instances (Java objects) are serialized in the database, not their visualization, which is created during runtime on the basis of an according Java instance. Therefore, instances are time-independent and self-contained. We explain the concept of instances in Sections 5.3 and 5.4.

The term **History Mode** describes the capability of an algorithm instance executed on the server, to go back and forth in execution. This behavior is controlled by a user on the client. Every NetLuke plugin needs to support this functionality.

The term **HTML5** does not only define the 5th version of the HTML markup language but also Cascading Style Sheets 3 (CSS3), JavaScript APIs and various other extensions [9]. In this thesis we will often speak of HTML5 **as unspecified** or **not fully standardized**. This refers to the work of W3C[7] and WHATWG [8]. The former specifies/standardizes existing implementations (snapshots) by the latter which considers HTML5 as a "Living Standard". Therefore, WHATWG continuously changes and extends the "Standard".

We use the term **platform** twofold. On the one hand we mean the platform the system offers to users i.e. the AV system itself. On the other hand we speak of platforms describing the software and hardware environment specific to devices.

[7] http://www.w3.org
[8] http://www.whatwg.org/

Chapter 2

Related Work

This chapter elaborates main ideas and theories which shape the field of Algorithm Visualization (Section 2.1). We discuss why educational effectiveness is important, what it means for us and why it is crucial for AV systems in creating educational value. The next Section 2.2 is dedicated to the history of AV systems illustrating conceptual progression in the field. In this Section we closely look at enhancements the field has experienced. We further discuss major AV concepts and concrete implementations in Section 2.3, which will be evaluated in Section 2.4 concerning quality, usefulness and educational value.

Discussion about related work is essential to our project. Valuable information regarding AV concepts, techniques, style and philosophy are derived from previous implementations and are further incorporated in our work.

2.1 The Field, its Ideas and Theories

The idea to AV systems is directly mapped to the difficulties in teaching and learning algorithms and data structures. Algorithms are often complex and dynamic, which defines the central issue: Understanding complex mechanisms/sequences in action can be very difficult. Students of computer science work with learning materials such as textbooks in hard copy or lecture presentations in digital form. Most resource types feature descriptive images, diagrams or models to illustrate and reduce algorithm complexity. Some of those materials are more likely to fail, if the subject matter is highly dynamic or even abstract to the student. They fail in the sense that a learner may not be able to understand the subject matter. For many supporters of AV aided learning, the logical conclusion is utilization of an interactive AV system, for instance one mentioned in Section 2.3. But the question remains, why is there no commonly used "standard system" available? Why are there so many different systems? Why are AV systems not established in academic lectures. In his study about Algorithm Visualization effectiveness [15], Hundhausen et al. concludes that AV systems are rarely adopted in computer science education.

> "[. . .] Despite its intuitive appeal as a pedagogical aid, algorithm visualization technology has failed to catch on in mainstream computer science education".

Aside from the inherent difficulties in creating AV solutions, this statement is particularly true, as there is discussion (for example in [2, 41]) about whether AV technology in general, is educationally effective, or not. Rejection of AV systems certainly implies that traditional teaching methods produce better results. As many studies have shown [15, 18, 24, 44] there is a significant chance that **AV technology is educationally effective**. Therefore it represents (at least) an alternative to static learning materials, if AV systems respect the needs of lectures and students alike. Before we outline what makes an AV technology educationally effective and pedagogical valuable we briefly discuss which theories go into the matter of learning effectiveness. Theoretical deliberations need to be taken into account. For instances, Bazik [2] points out that the lack of mainstream solutions is a result of missing theoretical foundations. Systematic research could provide a much needed normative framework for efficient creation of AV systems. Today two (meta) theories of effectiveness shape the theoretical groundwork of such systems: (a) *Epistemic Fidelity* and (b) *Cognitive Constructivism*.

(a) *Epistemic Fidelity* (EF) states that humans have certain symbolic models of real world objects in their minds. The concept further supports the claim, that students do have specific mental models of algorithm representations e.g. visualizations in their mindsets. In regard to AV systems EF describes a denotational semantic match between a learners mental model and the actual graphical representation of the algorithm in action. If the match exhibits a high level of fidelity, that is when the AV system is displaying what the student had in mind, the system contributes to a substantial gain in transmission of knowledge. The learning process is derived from an encode/decode pattern. The author of an AV system encodes his own knowledge structures in graphical representations which is later decoded by the student (referred as knowledge flow). Educational effectiveness is therefore achieved by a strong encoding of an experts mental model of an algorithm to a consistent graphical representation [2].

(b) *Cognitive Constructivism* (CC) assumes that people do not possess a prior mental model of knowledge. The theory basically states, that knowledge is constructed by interaction and engagement in the subject matter by mental processes. This constructed knowledge is individual and not absolute in contrast to EF theory. The assumptions lead to the conclusion that an AV system needs to motivate a learner in creating own mental images and understanding. Therefore an AV system must be able to support the creation of individual understandings of what is going on. The theory further suggests that learners should be able to define their own input sets of data and they should be motivated to make predictions of future algorithm states. Another key aspect is the ability to construct own visual representations of an algorithm or a data-structure to strengthen the link of a learner to an algorithmic representation. If an AV system is able to actively engage a student in creating knowledge during a learning process, then it can be considered as educational effective [15].

Both theories have a very different understanding of what knowledge truly is. As a result, the ways of achieving knowledge differ fundamentally. Epistemic Fidelity's observable behavior concept is characterized by a passive behavior of the learner, while Cognitive Constructivism is emphasizing the active involvement of learners. Hundhausen's studies clearly show that active learning is superior to passive learning. However, passive viewing

of algorithm visualizations does not lead to better results than conventional learning materials. *How* students use an AV system is in general much more important than *what* a student perceives.

"[...] In particular, our meta-study suggests that the most successful educational uses of AV technology are those in which the technology is used as a vehicle for actively engaging students in the process of learning algorithms" [15].

These results and assumptions correspond with our notion of an educational effective system that extends traditional learning methods. Rather than strictly following the cognitive constructivist proposals and suggestions, we keep the active viewpoint in mind. NetLuke is intentionally designed to motivate students in creating own examples and helping them to deduce knowledge from predictions about algorithmic operations. We also consider the EF concept as valuable, because NetLuke is intended to complement traditional learning methods, so we assume that students possess at least some knowledge about algorithms rather than being complete beginners. Hence, a student has an idea what the topic is about. As a result, we adopt concepts of EF in the sense that we definitely encode specific notions of representations, which need to be decoded. We assume that educational effectiveness is not solely defined by a theoretical approach but in practical use it is context dependent. In other words: The application domain and the intentions behind the use of AV matters. These considerations lead us to typical use cases for NetLuke. We want to:

- **Encourage students** in learning computer science fundamentals by strengthening their interest in computer science topics.
- **Help students** in gaining deep understanding of algorithms and data structures during their execution process. After learning with NetLuke, students should be able to pass exams, fulfill task assignments or simply follow the topics discussed in class.
- **Help teachers** and instructors in presenting topics effectively by actually showing what is going on while the algorithm executes. This enables teachers to give detailed information about algorithm characteristics.

As from our point of view, modern AV in general, is not just a subset of software visualization, but can be considered as a subset of the eLearning paradigm. AV solutions which focus on active learning succeed static concepts of presentation and allow new ways of dealing with topics. Nevertheless, NetLuke does not feature social or cooperative interaction practices, which is why it can be considered as a web based learning or training platform, particularly because it does not intend to replace conventional learning materials.

2.2 History of the Field

The field of algorithm visualization/animation for learning purposes is relatively old. The first popular, and at the time most comprehensive step in this direction, was undertaken

by Ronald M. Baecker at the University of Toronto, with his film *"Sorting Out Sorting"* in 1981. It showed the workings of several fundamental sorting algorithms in computer science. In 1984 BALSA-I (*Brown ALgorithm Simulator and Animator*), which was followed by BALSA-II in 1988, the first AV programs emerged, which featured an interactive algorithm animation system. They were able to display the execution of multiple algorithms instead of only one [10, 34, 38].

TANGO (*Transition-based Animation GeneratiOn*) was developed in the year 1990 at Brown University. The system featured an architectural framework for the design of animations. The first system following a client server paradigm was "Mocha". Many of those systems were extended and improved, but rarely ported to different software and hardware environments. They mostly remained platform dependent. In the beginning of the 1990s [38] the field received a broad level of attention. First of all the scientific community started to question the descriptive approach many AV systems were taking [25] by considering the theories mentioned in Section 2.1. Soon it became clear that learners need to be affected with the subject matters, for instance through animations of dynamic processes, questions and the mapping of symbolic patterns to operations of an algorithm. This period also marks the discovery of animations fulfilling a cognitive function, making them a valuable element in AV systems. Many studies were undertaken to provide theoretical groundwork for enhanced projects. The findings resulted in a new generation of AV systems. A prime example in this regard is HalVis[41] (*Hypermedia Algorithm Visualization system*), which contained many new features not seen before[1] and a new design philosophy. The software was directly derived from a study with thorough experiments undertaken at the University of Auburn. The intention of HalVis was to go beyond traditional text book like portrayal in respect to the ideas of Cognitive Constructivism. Another circumstance led to a growing interest in AV. The growing popularity of Java elevated the interest of young developers, to create feature rich AV systems – fast and easy.

Java opened up new possibilities such as cross platform development, object oriented programming styles, powerful graphical expressiveness through libraries and web integration through applets. Soon many different AV tools and applets emerged and spread all over the world wide web. Most of these applications concentrated on a small number of algorithms or data structures and heavily differed in feature scope and/or quality. As a result many AV tools deviated in usage and user experience in general - but in the end, Java revolutionized the development of AV solutions. Since then, a growing number of developers found previous solutions unsatisfactory, either because of their philosophy, purpose, feature set or their general learning effectiveness, which lead to a high level of fragmentation among AV systems.

As of now, in 2013, the number of active AV projects is constantly decreasing. The interest in the field started to decline after 2002 [38]. According to Shaffer et al. this effect is still in progress and cannot be explained by a saturation in AV topics, in the field itself, or by hiting a ceiling in terms of quality. Shaffer et al. further explain in their studies about the field, that most of the tools reviewed are of poor technical quality, or have little to no pedagogical value. A quarter of visualizations fall in between this valuation

[1] In general HalVis supported the paradigm of "enhanced - interactive" learning. Interactive Examples, Hyperlinks, Animations, Questions, etc.

scheme: Those solutions are potentially useful but severely limited in terms of quality. Even the better AV tools tend to have serious deficiencies. Roughly half of all algorithm visualization systems are actually animations, with no further content. In this case, users are relegated to being passive observers with no control over pacing (aside from animation speed), the data being processed, or the operations being conducted. [38, p. 3].

We explain this trend with the rapid changes in technology (fragmentation) combined with the variety of software platforms. This is accompanied by changing pedagogical priorities. On the one hand Java most certainly lost its appeal as the primary choice aside from C/C++, for academic projects. This assumption pertains users and developers alike, because languages like Ruby, Python, C#, Perl and even JavaScript receive justifiably attention, which is due to their high level of flexibility (see [40, 43]). On the other hand it seems like as if algorithms itself are "less important" in computer science education nowadays. As a result, scientific groups are less interested in developing new solutions from scratch [38]. We conclude that development activity is somehow dependent from code re-use abilities and the target platform/programming language. From a user point of view - nobody is interested in desktop exclusive versions of software that *"smell like 90s"*. We take those factors into consideration, by addressing the mentioned shortcomings (see Sections 3.3.1 and 3.3.3 for further discussion).

2.3 Selected Examples

To gain a feeling for already implemented solutions in relation to NetLuke, we examined some popular AV tools. The examples within this Section represent a subset of a broad range of reviewed solutions (see Table 2.1 for a list of tools). We selected AV implementations on the basis of differences in approach, philosophy and purpose, but also because of their relatively good overall quality. A comprehensive comparison of AV tools, including in-depth analysis can be found in Purvi Saraiya's master thesis: *"Effective Features of Algorithm Visualizations"* [34]. We restrain our own comparison to features in Table 2.2.

2.3.1 TRAKLA2

The first version of TRAKLA has been developed at the Helsinki University of Technology (HUT) by its department of computer science and engineering in the year 1993. Version 2 was released in 2007. The project is enhanced ever since. Like most other AV systems emerged in the late 1990s, TRAKLA visualized several popular algorithms and data structures which are subject matter in undergraduate computer science classes e.g. CS1. The underlying framework was designed around the idea, of having an integrated, multi-user, multi-role system which helps teachers in their lectures and students in their learning tasks [2]. TRAKLA's preliminary intention is to reduce the workload of lecturers/teachers by the introduction of an automated assessment and feedback system. It allows automatic grading and exercise creation by instructors [21]. The concept behind TRAKLA is quite exciting because it combines automatic assessment and feedback capabilities with self study exercises for students. All of which is possible beyond classroom borders. The architecture even supports collaboration features such as discussion boards, email integration and chat functions. The number of implemented algorithms and data

[2] A recent (2010) extensive study about TRAKLA2 can be found in [32].

structures is very high. TRAKLA has almost every topic at its disposal. You will find everything from basic search algorithms like bubble sort to more advanced topics like radix sort to search-trees, graph algorithms (Figure 2.3 depicts Prim's Algorithm), hashing and priority queues (Figure 2.1). Figure 2.2 illustrates a selection of basic algorithms to choose from. Although the developers claim that TRAKLA allows users to create their own examples and exercises – in most cases pre-defined data input sets seem to be the standard case.

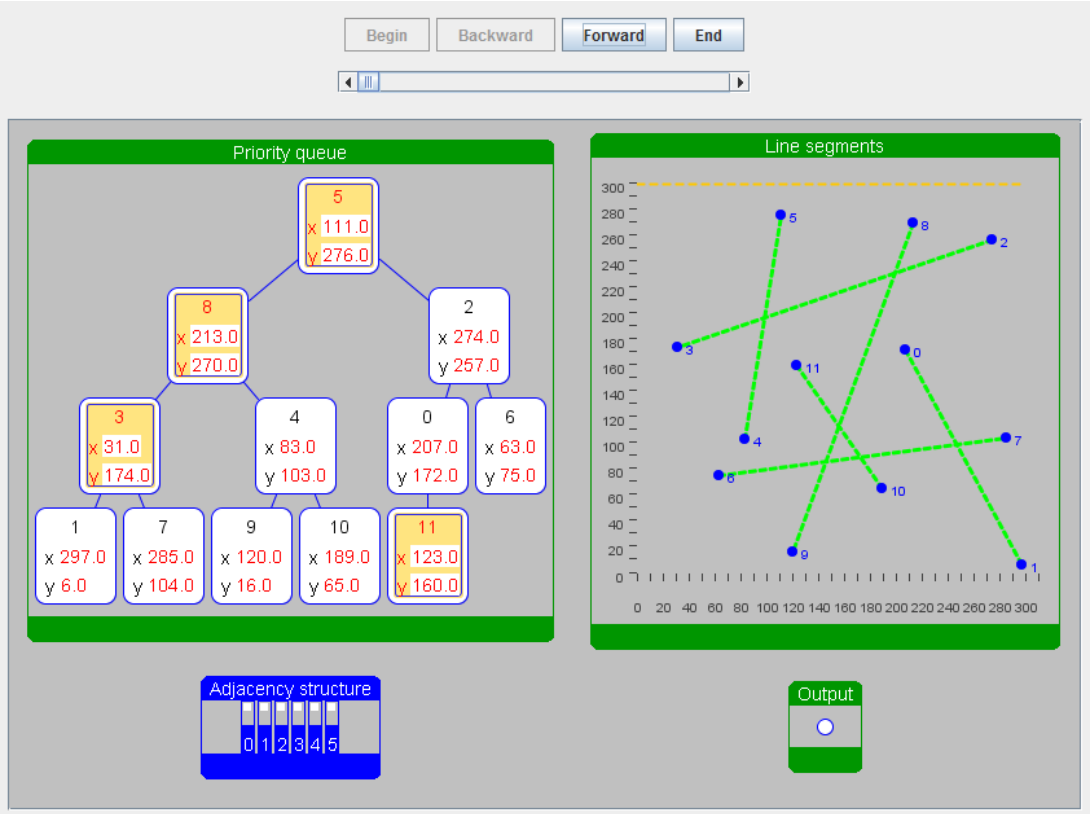

Figure 2.1: TRAKLA2 - Priority Queue Implementation

Another very positive aspect is the UI. Look & Feel of the system is consistent throughout every single algorithm. Views and information sections are well structured and do rarely change from one topic to the other. The high level of consistency combined with the well structured UI elements are particularly important to students as they help to gain control over the application. We got the impression that a user "feels safe" when using the tool. The built in drag and drop capabilities allow intuitive interaction with the topics at hand.

The rich feature set makes TRAKLA very useful in a teaching environment which relies on its capabilities. TRAKLA is meant to be embedded in a course system, because it is taking off work from teachers. It substitutes common functions and tasks by its automatic feedback and grading system. A teacher in an administrative role is able to manage courses within the system by preparing exercises or reviewing students progress with automatically generated statistical data. Although the visual simulation system is not explicitly comparable to NetLuke, it comprises several aspects of an effective learning system, because it deeply involves the learner in the topic. The front-end is web based (Applet) – hence it requires a Java VM. The Java applet environment is the real drawback of TRAKLA as it cannot be considered as a future proof environment outside of class-

1: Basic algorithms (DL 01 Jan. 2020 00:00:00 GMT+0200) (minimize)	Points	Submissions
1: Binary search	0 / 2	0
2: Interpolation search	0 / 2	0
3: Preorder	0 / 1	0
4: Preorder with stack	0 / 3	0
5: Inorder	0 / 1	0
6: Postorder	0 / 1	0
7: Levelorder	0 / 1	0
8: Postfix evaluation	0 / 1	0
9: Infix to Postfix	0 / 1	0
10: Dynamic programming	0 / 4	0
11: Refresher exercise: Basic Algorithms	0 / 0	0

Your points: 0/17

Figure 2.2: TRAKLA2 - Algorithm Selection

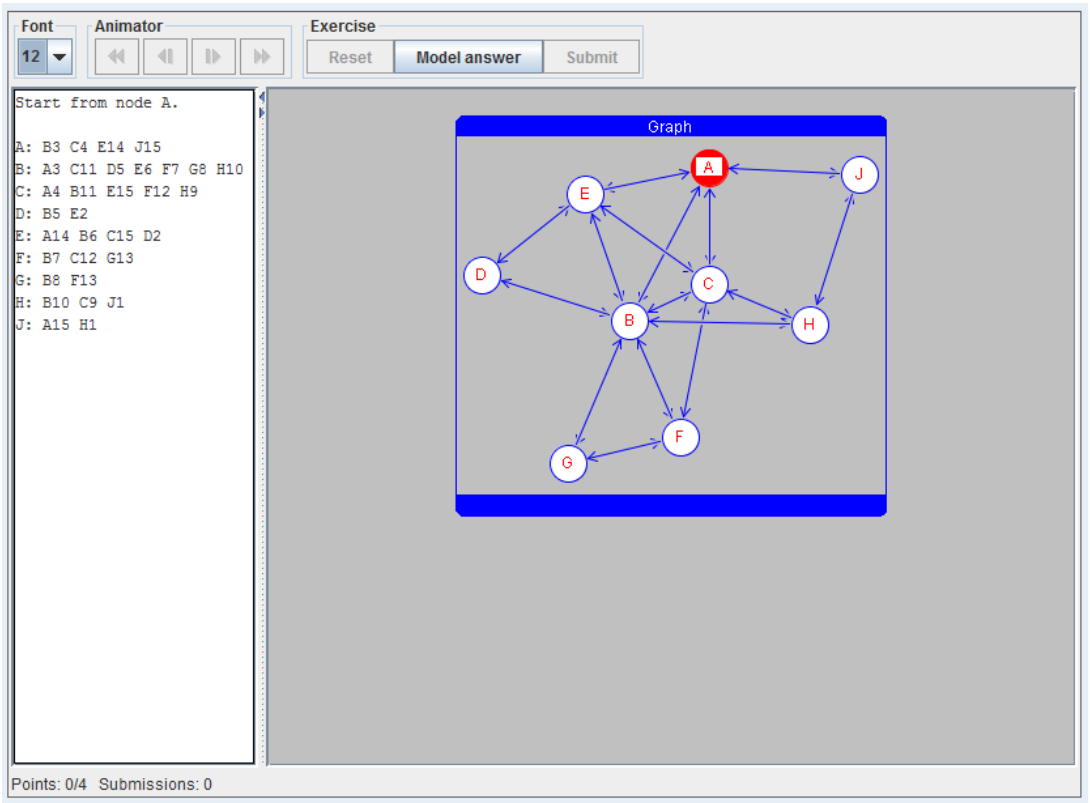

Figure 2.3: TRAKLA2 - Prim Algorithm Implementation

rooms. All in all TRAKLA seems to be a powerful AV tool, but its design philosophy is different from our notation of a relatively lightweight personal application that is intended to extend conventional learning materials.

2.3.2 LUCAS

LUCAS (*Learning and Understanding of the Characteristics of Algorithms and data Structures*) was designed and implemented by Babic & Deischler in 1999 at the University of Vienna under the supervision of Erich Schikuta [35]. The University of Vienna needed a tool to support students of undergraduate courses in their preparations for exams. The computer science department wanted a solution which is able to illustrate the execution flow of algorithms and data structures during course lectures. The basic requirements to the software are similar to other AV systems around, but nonetheless different in important aspects. What sets LUCAS apart from other tools is the high level of freedom a user has in creating examples. A user is able to define own data input sets for every implementation topic, thus makes the system ideal for novice users as examples can vary in difficulty. The GUI was intentionally designed to make LUCAS easy to use, which renders a user manual truly unnecessary. One special feature is LUCAS *"tape recorder"*. It allows users to return to every previous state of an algorithm or a data structure. Contrary to many other tools, it is possible to modify a previous state e.g. change of execution history. The tape recorder approach is similar to our "history mode".

"[...] Hereby the user can arbitrarily go forth and back in the execution of specific data structure methods or algorithm steps. The user is able to introspect all specific situations of a data structure or of an algorithm by going back in the trace of the executed steps. Therefore, it is possible, that the user can make new decisions at any point of the execution flow allowing the user to answer *"what would have been questions."* [35, p. 3].

As discussed before, LUCAS was developed to promote extensive self study capabilities for students having only rudimentary knowledge about algorithms. The concept of LUCAS is not to provide an "all in one" solution as TRAKLA does, but it extends/complements traditional textbook illustrations to overcome their lack of expressiveness. In the center of the idea stands the concept of dynamic visualization with a high level of freedom in creating examples, which makes it easy for students to explore algorithmic execution extensively. Like the majority of AV systems created in the late 1990s LUCAS is based on the Java programming language with a modular architecture including web based access through an applet. Java was the obvious choice, as LUCAS was planned as an extensible modular program. Many of the implemented algorithms were developed by students within the scope of their bachelor thesis. In 2010 we implemented Prim's Algorithm as illustrated in Figure 2.4 and Kruskal's algorithm. Letting students contribute to the project has certain advantages. It allows consistent development of new topics which fit in the lightweight framework of LUCAS. The philosophy was clearly: *"Implementations from students for students"*. The concept seems to work well, as many implementations are useful and "self study friendly". LUCAS takes a pass on formal descriptive explanation features in the main viewport. For instance: A pseudo code view is missing as well as a description view, which explains what operation is in action, because the designers do not seem to consider these features as necessary for the given use cases. LUCAS is the direct predecessor of NetLuke, hence the project title – "NetLuke". It stands for an enhanced and truly web based progression of LUCAS.

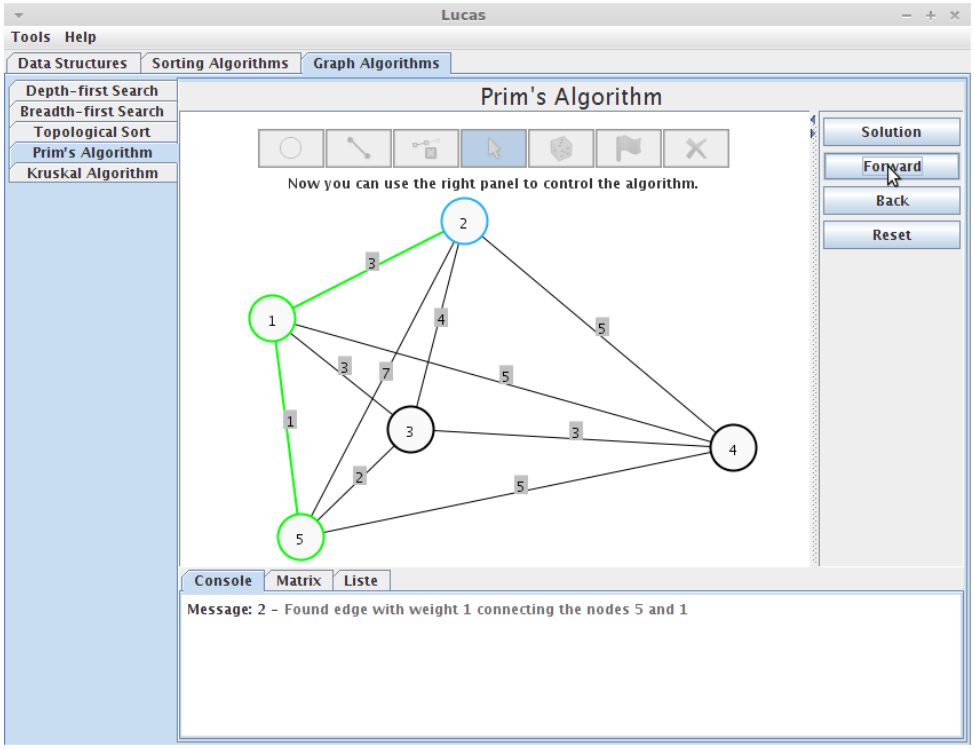

Figure 2.4: LUCAS - Prim Algorithm Implementation

2.3.3 AlViE

Yet another powerful AV system is AlviE. It has been developed by Giorgio Gambosi (University of Rome II), Roberto Grossi (University of Pisa), Carlo Nocentini (University of Florence) and Walter Verdese (University of Pisa). The project is currently maintained by Pilu Crescenzi. The idea behind AlviE is the integration of an AV tool into CS2 courses at the University of Florence [5]. Like many other tools, AlViE was implemented in Java and therefore requires an installed JVM. It is built upon the *JAZ* project [3] – its predecessor. Version 3.07 of AlviE, the latest release of the 3.x version tree, was released on February 1st in 2010. An advantage over other AV solutions is its supports for a wide range of algorithms (34). AlViE includes topic implementations reaching from simple calculations of Fibonacci numbers to the illustration of the Huffman code to algorithms which find the shortest path within a graph. For instance: The Dijkstra algorithm or the Bellman-Ford algorithm. However, the current version is 4.0.1 which has been released on September 6th, 2012. While the basic concept of the tool is the same as in the previous version, there has been a major redesign of the UI. The actual version only supports around 20 algorithms. This is due to some significant changes in the (visual) output system.

In contrast to LUCAS, AlviE's algorithm calculation routines are based on XML configuration files. They have to be created before the execution of the algorithm starts. Since version 4 it is possible to create input files from within the programs' UI. Therefore AlViE's data input method is very different from most other AV tools. For instance it is not possible to edit the XML file while the visualization is active/executing. Basically,

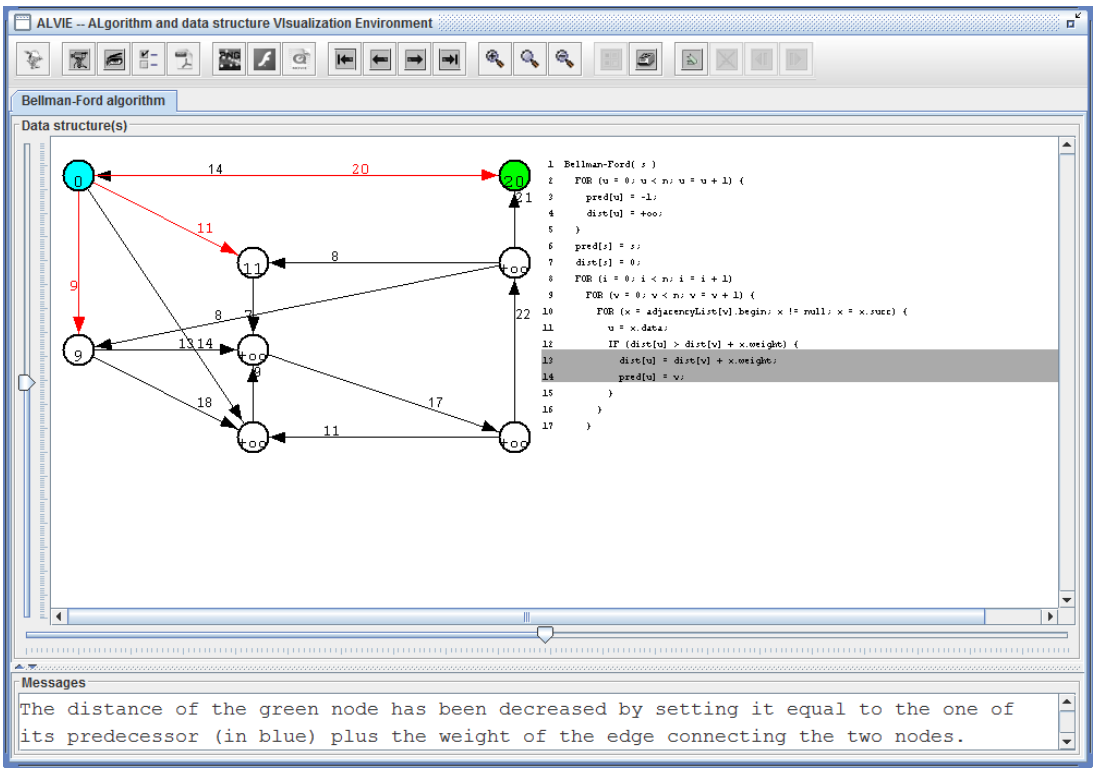

Figure 2.5: AlViE User Interface

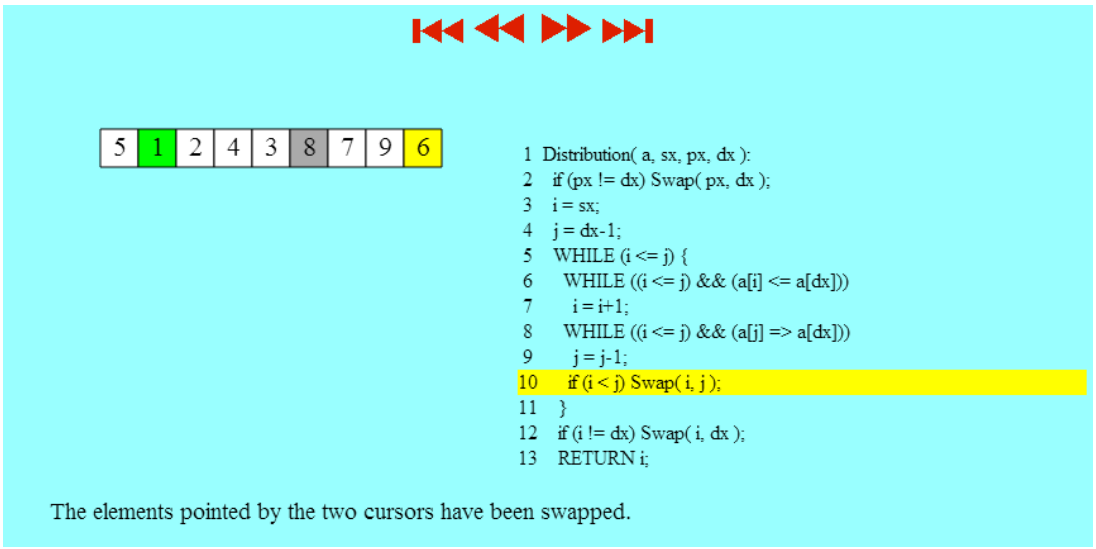

Figure 2.6: AlViE HTML5 Video Creator

a user has to create all input parameters before launching an algorithm example. The resulting visualization is comparable to a "movie". For instance: Inserting an element into a binary tree during runtime is not possible. One can argue about advantages and disadvantages of this approach. We see potential for high quality visual output, although we do not consider this "static" visualization approach effective in the first place (see Section 1.2 for further discussion).

Once the algorithm is launched and active the user has the option to navigate stepwise through the different stages of the algorithm (Backward, Forward, End, Beginning). Every algorithm is shown graphically and in highlighted pseudo code. This is shown in Figure 2.5. The combination of these features makes an understanding of even complex algorithms relatively easy but creating own examples sometimes difficult. What sets AlViE apart from other AV systems, is its capability to export visualizations into different output formats. In version 3.x for instance, it is possible to create quicktime- or flash movies. A screenshot function to export a picture of the current algorithm stage has also been implemented. Since version 4.0.x these functions were replaced by a *"HTML5 creator tool"*. This feature exports a standalone web site on the basis of the previously selected algorithm visualization (see Figure 2.6). AlViE also comes with useful descriptions of its supported algorithms which can be found in the included manual.

2.4 Evaluation

Skimming through ideas and implementations[3] gave us a good idea about the state of the field and which particular aspects needed attention when developing a new solution. We also wanted to ask ourselves which enhancements should be incorporated in NetLuke. Before we started developing we looked at a broad range of tools and projects, which are listed[4] in Table 2.1. We picked up several ideas from other projects than LUCAS, in order to enhance NetLuke and to avoid pitfalls. We are impressed by TRAKLA's large feature set, consistency and its overall refinement. HalVis showed us how students can benefit from AV systems, if they respect the needs of users. JHAVÉ [27] features a straightforward client/server architecture which makes installation and usage comfortable. Algorithms in Action [33] (AiA) has several advanced topics at its disposal such as recursive search algorithms (Splay Tree or Patricia Tree) and advanced sort algorithms. These are just some examples indicating the feature-manifoldness of AV solutions. To give an overview over these features, we decided to provide a comparison in a Matrix 2.2 and discussion in Section 2.4.2. This particular Section does not only intend to assess AV implementations in order to classify NetLuke or set it into proportion, but to critically analyze what specific aspects lead to problems for students and teachers. Therefore we look at the tools introduced in Section 2.3, including other AV systems as well.

[3]For a comprehensive list of AV tools, including descriptions, a repository of references and news updates visit http://algoviz.org

[4]The tools listed in Table 2.1 contain projects which could not be tested directly, because their sources could not be compiled (XTANGO, POLKA) unless source code modifications were applied. Other tools are not available for full analysis because their project sites are down or inexistent (HaLVis, Mocha).

Project	Year	Status	Innovation
AiA	2000	active	Visualizations of recursive algorithms
AlViE	1998	active	Allowed exporting of visualizations to Flash/HTML5 videos
ANIMAl	2000	inactive	Embedded scripting language and embedded animation creation system
HalVis	2002	inactive	Introduced a comprehensive question system which asks students about algorithmic execution details (allowed predictions)
LUCAS	2008	active	Straightforward, seamless and consistent creation of algorithm or data structure instances
JAWAA	1998	active	Extensive use of scripting technology to create algorithm animations
JHAVÉ	2010	active	Client/Server Environment
MatrixPRO	2004	active	Java Framework which supports algorithm simulation *ex tempore* with customizable animations
Mocha	1996	inactive	First system with a distributed client/server architecture
POLKA	1993	inactive	Featured a framework which allowed students to create animations within the environment
TRAKLA2	2007	active	High degree of AV software inclusion in course systems
XTANGO	1992	inactive	First Animation system on the X environment

Table 2.1: List of reviewed AV tools

2.4.1 Selected Example Projects

Two major aspects were considered during the evaluation of the AV systems in Section 2.3. First, technical- and second, educational quality concerning self study. From an educational viewpoint, as far as our notion goes, the tools we can partially recommend are LUCAS and to some extend AlVie 3/4.

Both tools feature a useful number of algorithm and data structure implementations which actually exhibit adequate to very high pedagogical value. Both systems encompass a rather good architecture and their animations help students in self study efforts. TRAKLA2 needs a registration at the online platform and is by far not as easy to use/deploy as LUCAS. Because of its conceptualization as an integrated AV system (integration within an academic course system), we do not see it as an optimal solution for self study purposes. AlViE 3 on the other hand is also a very interesting platform, particularly because of its included implementations and the consistent user interface. The visualization exporting tools may come in handy for presentation purposes. Feeding algorithms with XML data is not difficult after a while, but it requires (too much) time in the first place. We consider the need for creating XML files as the major handicap of AlViE.

As far as LUCAS is concerned, we see that it makes a difference in self study efforts, because it incorporates many easy to control and visually expressive topics. A user is actively engaged in learning. We also like the ability to create examples with self defined sets of data. Together with good animations in some topic implementations, we consider this feature as a key advantage over most other AV solutions. However, most topics vary in look & feel and some implementations contain bugs, mostly in the "tape recorder" part. The UI is simple, straightforward and easy to understand, but sometimes inconsistently structured. What LUCAS lacks, is a certain level of overall refinement, which seems legitimate given the fact that it is developed/extended by students. The project developed over a long period of time. This had a negative effect on the code quality, making LUCAS difficult to extend/maintain. From a technical point of view, it reached a dead-end, making a successor like NetLuke necessary.

2.4.2 Feature Comparison

Apart from included topics in AV solutions, we were interested in their functionality and the level of freedom a user experiences in creation of examples. We are further interested in the quality of visualizations and features which guide and help users in their learning tasks. We compared some of the (better) available solutions to get a general view about common featuresets in the field of AV. The analysis gives us indications about requirements for NetLuke. The results of our comparison are shown in Table 2.2. In the paragraph below you will find the description of the features we investigated.

Feature Description

1. **Direct Data Input** – Are users able to edit a given input set of data before/after loading a visualization example?

2. **Pre defined examples** – Is a user able to choose visualizations from existing examples or is random creation supported?

3. **Custom data input** – Are users able to freely input data regardless of extent or type which results in variation of example difficulty?

4. **Dynamic Animations/Visualizations** – Does the AV system allow custom placement e.g. drag & drop of objects, are operations supported by animations and suchlike?

5. **Scripting/Configuration Capabilities** – Is the system programmable by scripts or configuration files?

6. **Pseudo Code View** – Does the system provide a view with highlighted pseudo code throughout all topics?

7. **Help View** – Does the AV system provide additional textual guidance concerning operations or execution to the executed visualization?

8. **References** – Are additional references provided which allow in-depth analysis of topics?

9. **Visualization Export** – Does the system offer functionality to export visualizations of examples as pictures, videos or in other representational forms?
10. **Quiz mode & Questions** – Does the system contain a quiz mode or does it impose questions during algorithm runtime?
11. **Debug System** – Are users able to view debug messages in case of errors?
12. **Developer Guide** – Does the project page, if one exists, provide source code with examples and/or guidance on how to implement additional topics?
13. **Web Access** – Is the system accessible over a Network/Internet?
14. **Mobile OS Compatible** – Is the system compatible and usable on a mobile device such as a tablet or phone?

AV Feature Matrix

	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	x	x	x	x[1]	–	x	x	–	x	–	–	–	–	–	–
ANIMAL	–	x	x[1]	–	x	x	–	–	x	–	–	–	x[1]	–	–
LUCAS	x	x	x	x	–	–	x	x	x	x	–	–	x[1]	x[2]	–
HalVis	–	x	x[1]	x[1]	–	x	x	– [2]	x	– [2]	x	–	–	–	–
AlViE	–	x	x	x[1]	x	x	–	x	x	x	–	–	–	–	–
JHAVÉ	–	x	x[1]	x[1]	x	x	x[1]	–	x	–	x	x	x[1]	x	–
TRAKLA2	– [3]	x	– [3]	x[1]	–	x	–	–	x	–	x	–	–	x	–

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

Table 2.2: AV Feature Matrix

The matrix shows the differences in functionality/features between the solutions. As expected, none of the tools allow learning on a mobile device. Some platforms can be used over a network (with an according client application) or can be launched from a web page (applet). Every solution offers the ability to go back and forth in execution (unexpected), but it is rarely possible to edit, for instance, a data structure during this

history mode. Pseudo code views are very popular among many AV systems whereas only AlViE and LUCAS allow exporting of visualizations to different mediums. This comes as a surprise, but it depicts the focus of developers of creating visualizations, overlooking the requirements of users. Nevertheless, we expected more systems to have some sort of scripting abilities or configuration options. A debugging window can only be found in JHAVÉ, which exhibits the most comprehensive set of features. Only LUCAS and Algorithms in Action [33] (AiA) allow easy and direct data input during runtime.

NetLuke will incorporate all of the mentioned features, except a pseudo-code view and scripting capabilities although both features may be added individually to a module implementation by its author. Further discussion is provided in Chapter 3.

2.4.3 Implementation Quality

By looking at the aggregate of AV tools comprising the field, we observed that their overall quality is far from being satisfactory. Most of the UIs we encountered in AV tools, other than the ones mentioned in Section 2.3 (mostly applets) are clunky and difficult to use. They often force a user to spend a serious amount of time to understand the control elements or the general intention behind them. Most of the applets that can be found by simple Google searches sometimes do not even work or do not allow users to create self-created examples. They just visualize algorithm operations in a static or predefined-automatic way. Muldner [25] argues that "[. . .] many other existing algorithm animation systems resemble visual debuggers in that they show the execution of the algorithm by code-stepping, and illustrate only the primitive code". It seems like that in most cases, these tools were developed with solely one single goal in mind: Display the behavior of the algorithm as fast as possible, but in a descriptive way. The results are often browser applets or old AWT (Abstract Window Toolkit) based Java programs with a very short feature list.

None of those programs matched our notion of a *"user centered approach"* as mentioned in Section 1.3. Some of the reviewed software implemented more than one algorithm, whilst others featured only one single algorithm. None of these programs consequently follow a platform/framework pattern that would make it easy to create collections of different algorithms. Therefore students may need to find a tool for each algorithm. We doubt that students benefit from "quick and dirty" implementations.

2.4.4 Summary

The field of AV offers a myriad of solutions for students and teachers, ranging from little applets with one algorithm to fully featured web based course systems. Applets may help students in self study efforts, but AV platforms such as TRAKLA, JHAVÉ or LUCAS offer a far better experience. In many aspects, most of the tools are old and out-dated – hence the overall platform compatibility, at least from today's perspective, is very poor as for instance applets do not work in most mobile browser (see Sections 3.1 and 3.5 for further discussion). Or they do not even start in a Java Runtime Environments (JRE) higher than 1.5. Deployment and installation of complex tools such as TRAKLA is difficult, rendering fast access to learning topics impossible. 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. The evaluation was also undertaken to answer the question, if the world needs yet another Algorithm Visualization system? Numerous reasons point in this direction, affirming the need for it.

Chapter 3

Requirements

This Chapter covers the essential project requirements, which have a direct effect on design decisions. Defining requirements is asking questions about NetLuke's low level functionality, system features and operational constraints. Requirements are derived from our outlined project goals (see Section 1.3 and 2.1) and of course influenced by related work introduced in Chapter 2. In order to support users in their learning tasks, the system must be highly expressive, educational effective and easy to use, while developers need to be able to implement modules extending the system. In the following sections we will discuss which functional and non-functional requirements determine our project architecture.

For a better understanding of our project implementation, we begin with an introduction to factors which influenced the definition of requirements 3.1. We discuss the effects of web technologies and competing platforms to provide orientation about the structure of requirements. We further define requirements in the Sections 3.2 – 3.4. We close this Chapter with an analysis about conflicts which arise through interdependent requirement definitions (Section 3.5.

3.1 Considerations

3.1.1 Platform Competition

NetLuke targets multiple client platforms, differing vastly from the ones in the *"pre smartphone era"*. With the advent of mobile "gadgets" and their operating systems e.g. Apple iOS, Google Android, Microsoft Windows Phone 7/8 came hardware which is considerably less powerful then what a modern PC offers. Mobile devices have "uncommon" screen sizes, different input methods, lower processing power and very limited memory capabilities. As we know, the emergence of these platforms, with their very own software environments "shacked" the world of the traditional desktop paradigm. Nowadays desktop and mobile platforms co-exist, but their ecosystems compete with one another. This situation reminded us of the famous browser war between Microsoft's Internet Explorer and Netscape Navigator during the late 1990s [9, p. 3] or the "desktop race" between Apple and Microsoft in the early 1990s. Both contentions had serious consequences for

(web) designers [30]. Now, this war evolved and re-emerged as a platform war [37] in between mobile platforms on the one hand and desktop vs. mobile on the other. What is particularly striking in this matter, is the rapid speed of development in the mobile/web sector, which makes it hard to project future proof designs. This circumstance affects NetLuke directly.

Heavy platform competition results in drawbacks for end users and developers alike, because modern platforms exhibit a high degree of heterogeneity in hard- as well as in software. This forces developers to adopt and redesign previous solutions. The web is getting more complex every day – again! A common "best practice" to reduce complexity falls under the term "responsive web design" [6, p. 102], a concept we utilize extensively (see Section 6.2.2 for further discussion). An alternative way of dealing with necessary adaption tasks is discussed in Section 3.1.2. The drawbacks for users become visible in slow, error prone and/or visually unattractive web applications. Although many frameworks [1] significantly enhance design, code quality and user experience, a lot of effort has to be put into workarounds for mobile devices [13]. What stands out, and is not likely to change, is the huge gap in performance and physical screen size of mobile devices compared to their desktop counterpart. This creates problems for certain UI features.

In the desktop era, it was easy to target the majority of platforms. One very common solution was Java with its operating system independent JVM, following the principle of "write once, run anywhere" (WORA). However, this principle does not apply anymore, because you will not find a JVM deployed on your mobile device, at least not for "ordinary – self created" Java applications. Applets do not run on iPhones and iPads, not even Java does natively. Google just offers limited – "to no" Applet support in their newer Android versions. To put it straight: Java on mobile platforms is a dead end, although Android Apps are Java based. A common solution to most of the mentioned problems above are HTML5 compatible web browsers – in our case "Mobile Web Applications" or just "web apps" (see Section 6.6).

3.1.2 Web Technologies

Eventually Rich Internet Applications (RIA) succeeded the limitations of modern platform heterogeneity by leveraging the abilities of modern web browsers/technologies [7]. Complex RIAs typically exhibit a desktop like application feeling with dynamic rendering, real time data communication and feature rich UIs. This "new" kind of web applications superseded the cumbersome Java Applets and static web pages. Continuous progress in network infrastructure and advancements made in HTML/CSS and JavaScript (JS) technologies/frameworks like *Asynchronous JavaScript and XML* (AJAX) enabled RIAs [2][46]. Today, RIAs are not bound to a desktop browser anymore, although many suffer from the problems mentioned in the previous Section. From our perspective, web apps are mobile RIA's. From a design perspective, their success is partly built upon many ideas/virtues

[1] The frameworks listed here (for a comprehensive guide visit http://www.markus-falk.com/mobile-frameworks-comparison-chart/) operate on different architectural layers, for instance directly on the client-side (within markup) or on lower level application development stages (server-side). Frameworks: http://jquerymobile.com/, http://phonegap.com/, http://www.sencha.com/, http://jqtouch.com/

[2] As of today, RIA's are mostly developed with JS/Ajax. The significance of Flash/Flex based implementations is declining, whereas another technology, Google Web Toolkit (GWT), gains popularity.

of the past which regained significance e.g. efficient use of limited hardware/network resources, space saving user interfaces and focus on information flow.

We referred to disadvantages of web based solutions in Section 1.3.Modern browsers support 3D animations (WebGL) and the full range of HTML5 features[3], but as a runtime environment, they are slow compared to environments which execute compiled code [19, p. 168]. Although Just in time Compilers (JIT) reduce the penalties of "delivered code" (JavaScript files are provided by the server and delivered to the JS compiler in contrast to compiled machine code), browser based applications sometimes suffer from poor performance or reduced responsiveness. Document Object Model (DOM) modifications are CPU and memory intensive tasks. Furthermore, standard implementations of JavaScript do not make use of multi core processors [8, p. 318]. These factors/problems are taken into account, but very difficult to solve. HTML5 and JavaScript offer a high level of potential system expressiveness, however its vendor specific browser implementations causes all kinds of problems ranging from low performance to layout issues on mobile devices.

> "[. . .] It is challenging to write nontrivial client-side JavaScript programs that run correctly on such a wide variety of platforms." [8, p.325].

The causes for this aggravating circumstance is a result of incomplete standardization and a growing number of complex heterogeneous environments [4]. Although HTML5 was not fully specified at the time of writing this thesis, we chose to rely on its capabilities to fulfill our goals. However, the imposed factors influenced our work (see Section 7.1).

3.1.3 Impacts

Different hardware devices, the vast range in processing capabilities, screen- sizes and resolutions, software platforms and on top different programming environments – all of these aspects related to *platforms* have an immediate effect on our projects' requirements. Therefore, all of the mentioned constraints, pros and cons of involved technologies need to be considered in the definition & design phase. Before we discuss project requirements itself, we want to give a better understanding why we distinguish between Platform- (Section 3.2), System- (Section 3.3) and Developer requirements (Section 3.4). The system overview presented in Figure 3.1 gives a preview of the system architecture, platforms involved and how components interact on a high level. The illustrated J2EE Server needs an Java Runtime Environment (JRE) installed on a server machine with any compatible operating system. The client needs a HTML5 capable web browser in order to display the visualizations. The client OS does not matter in this case. Server and client exchange XML messages to keep visualization and algorithm logic synchronized. Chapter 4 will illustrate the concept in detail. Although the depicted architecture looks very simple, each system unit has its own requirements to fulfill in order to realize the system.

[3] http://www.html5rocks.com/en/why

[4] A good example for an unsatisfactory web application is Facebook's "old" HTML5 based Android app, which was dropped/discontinued in favor of a native Google Android application. Visit http://techcrunch.com/2012/12/13/facebook-android-faster/ for a detailed description.

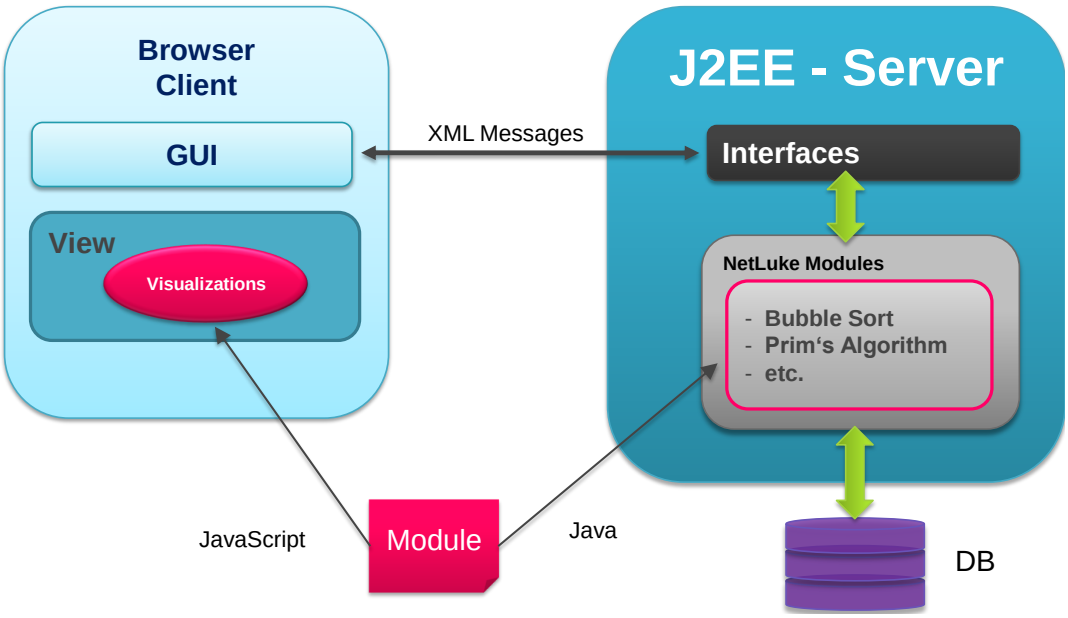

Figure 3.1: System Overview

3.2 Platform Requirements

By regarding the thoughts in Section 3.1 we decided to define non functional requirements which are owed/derived from determining factors in mobile and desktop client platforms. This also includes differences in usage interaction patterns. In our case, platforms "create" requirements for system components, therefore we subsume the mentioned considerations into the term "platform requirements". Since NetLuke utilizes an J2EE application server, we outlined several requirements, which are subject to a server environment. In order to meet the goals and concepts defined in Section 1.3, following platform requirements need to be fulfilled:

- True client platform independence in the sense that the front-end which comprises algorithm visualization and the UI, is compatible to every web browser which supports HTML5.
- Deference to very low overall hardware capabilities of mobile platforms despite heavy use of computational intensive HTML5 elements (canvas). NetLuke should work on middle-class mobile platforms [5] (2011) with a minimum resolution of 800x480. Desktop platforms however should exhibit a decent level of UI responsiveness utilizing enhanced hardware capabilities (2D/3D acceleration). Algorithm visualizations may use additional graphical effects on faster platforms (see Section 6.2).
- NetLuke shall be usable on devices with relatively low bandwidth interconnects such as standard 3G data connections. In order to provide consistent and predictable server response times the amount of transferred XML messages should be kept low.
- The system should respect the likelihood for client connection failures. It should be robust against connection instabilities.

[5] Apple iPhone 4 with iOS5 and Google Android 2.3

- The system should exhibit low server-side hardware requirements. Compatibility with major J2EE application servers such as JBoss, Tomcat or Jetty should be ensured.
- Ability to fully integrate NetLuke into an existing environment offering services like user authentication and management, database functionality and request handling (optional requirement).

3.3 System Requirements

This Section is divided into three cohesive parts. First off all, users need to be able to use the NetLuke system e.g. its algorithm visualizations provided by a module. Second, system usage needs to be simple.

Therefore (1) modules (Section 3.3.1) consisting of Java algorithm implementations and JavaScript visualization components need to fulfill a minimum set of requirements in order to exhibit educational value, as discussed in Section 1.2. The underlying (2) system environment (Section 3.3.2) which is not directly visible to the user, has to ensure seamless and comfortable use of modules. The (3) GUI (Section 3.3.3) itself needs to make all system and module functions accessible to the user throughout all device categories. We begin system requirement discussion with an introduction to modules.

3.3.1 Module Requirements

Module requirements represent core functional requirements, meaning algorithm visualization components and their server-side workings. Every module needs to meet requirements in order to operate within the environment flawlessly. Implementation details are not important to NetLuke as long as modules are not unnecessarily slow and incompatible with the system or web browsers. For a better understanding, Figure 3.2 illustrates artifacts comprising a module. The depicted JAR file contains Java class files which encapsulates an algorithm or a data structure implementation e.g. a binary tree. Methods of classes can be annotated with `@NetLuke` in order to be accessible from the JavaScript visualization routines. Chapters 4 and 5 will provide further explanations. XML messages contain arguments and return values for these methods. This leads to the following module requirements:

- Users can define their own set of input data for a given algorithm or a data structure, either manually or by random data generation to create examples. Minimum and maximum data boundaries to avoid unnecessary server load should be provided. These features need to be implemented by modules not the underlying system.
- Algorithm instances will be displayed on the main view as soon as a user has finished data input whereas data structure input operations (e.g. data insertion) need to be visualized right away.
- Users can reset any instance to its very first state (time of data input/first instance creation), at any given phase of the visualization/execution process to start over with learning tasks.

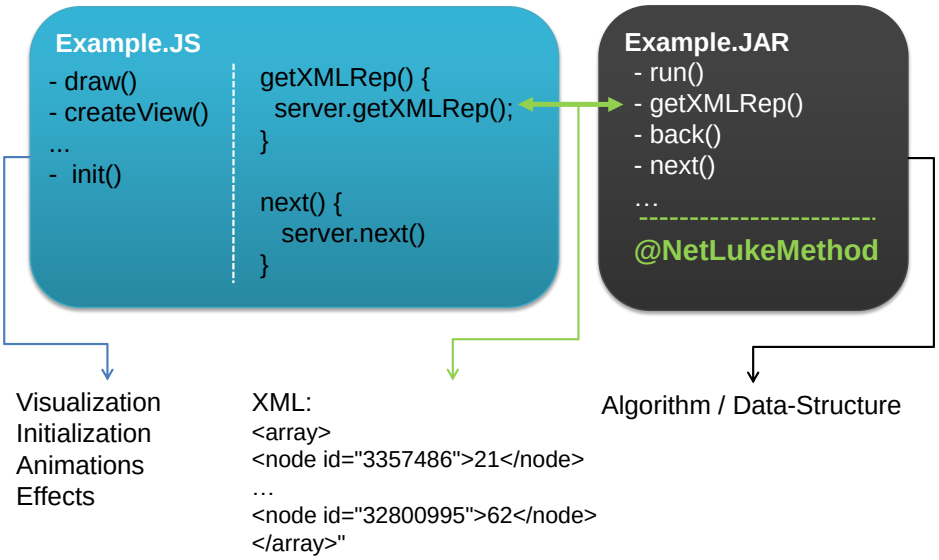

Figure 3.2: Module Overview

- Algorithms need to be able to execute without any steps in between and display results right away (results first). This behavior should resemble a playing movie with short pauses in between, or an animated slideshow.
- Algorithms are executed stepwise either forward or backwards. This requirement defines the history mode of modules (steps first).
- Data structure implementations need to provide minimum functionality and animation of the following procedures: (1) Insertion, (2) Deletion/Element removal and (3) Searching if applicable.
- Modules need to provide functionality to delete instances (see Section 1.6 for an explanation of the term instance) from the main view and memory (client and server) at any time.
- Modules need to be tolerant towards "mistyped" input e.g. characters instead of integer numbers (error tolerance).
- Modules need to be independent from the system environment and are solely connected with it through well defined interfaces using the intended components. This strong requirement defines, that modules are elf contained. Implementation details of the system remain transparent to modules.
- Modules have to make use of the panels and views provided by the system environment e.g. control panels for buttons which control module instances, views for visualization and menus for data input tasks.
- Visualizations in the main view need to make correct and extensive use of the embedded graphical library which is provided by the system environment (see Section 6.4). Graphical representations of elements e.g. graph or tree nodes need to focus on aesthetics. They need to provide a clear and unambiguous expression of the graphical concepts.
- Modules have to make correct use of additional textual output methods e.g. through provided text consoles (e.g. debug or algorithm console).

3.3.2 System Environment Requirements

First: The system environment, which is illustrated in Chapter 5 has to meet non functional requirements essential for users and developers. Performance-, robustness-, security- and reliability demands are in the center of consideration. The module requirements in Section 3.3.1 are dependent from the system environments capabilities. Second: The environment needs to provide functionality to allow developers easy deployment/integration of modules. This aspect presume low configuration efforts for module detection and easy module integration (UI, libraries). The system needs to provide interfaces (client and server-side) and components which help to incorporate modules in the system.

- Session management is transparent to the user. User sessions (not HTTP) are reusable. All instances are saved to the database upon session destroyal e.g. logout. If a user looses connectivity the last session will be restored. This requirement is particularly important for mobile devices with unstable interconnects.
- Avoidance of clear-text password saving in the database through high security encryption routines (one-way hashes plus salt).
- The system should be able to respond to events (event-driven web application) e.g. "Reverse AJAX" messaging to clients.
- Server-side algorithm logic encapsulated in modules must be accessible through JavaScript and Web Service (stateful service) clients. Business logic (Java modules) needs to be independent from clients.
- The environment needs to be able to recognize new modules semi- automatically and export its functionality through the service layer.
- The system shall provide configuration options for server port mapping, database back-end choices, logging capabilities, etc.
- Users can save or load multiple algorithm instances to the back-end database including preview images for easy recognition of instances.
- Instances can be exported as png image files with alpha channel support for future illustration purposes.
- The system shall not be limited to a single user, instead it should be capable to handle about $<$20 users concurrently (projected).

3.3.3 User Interface Requirements

We discussed that modern web applications need to work on very small screens with possibly small resolutions and on desktop computers with "gigantic" screens. One may think that smaller physical screens determine low resolutions. This assumption is misleading as next smartphone generations will feature about 4-5 inch size screens with resolutions up to 1920x1080 (Full HD resolution)[6]. High pixel densities would not be a big issue if a precise input method such as keyboard and mouse are used. Touch screens allow intuitive UI control with fingers and gestures, however, UIs need to be specifically designed and adjusted in order to support touch interaction properly [42, p. 47].

[6]In the time of the design phase, high end smartphones had a standard resolution of about 800x480 with a maximum of 326 ppi whereas tablets mostly featured a 1024x768 resolution with 132 ppi.

Therefore, the GUI needs to respect certain requirements to utilize an algorithm module in a consistent and easy way throughout different device categories. UI look & feel plays a key role in the design and is not considered as "nice to have" but instead absolutely necessary in order to support cognitive processes. On the one hand, we decided to give the GUI high priority in the design process, because it plays a key role in user experience. That is why we decided to define elementary user interface requirements to accommodate the obstacles which stem from touch displays, varying screen sizes and constrained processing capabilities. On the other hand we want to emphasize on the abstract separation of the user interface from the system environment to reach a better understanding on the deduction of UI design rules. As a result, following requirements were outlined:

- The UI needs to be straightforward and lightweight e.g. it should refrain from nested menus or overloading of control elements. Instead, it should make use of blocking dialogs and control element containers if possible.
- Consistent use of buttons, links and menus. Each control item must follow a specific design guideline which involves color schemes, structure, placement and presentation within dialogs.
- Dialogs and buttons should give visual feedback about their current state.
- Algorithm visualization must have visual priority over all other UI components (e.g. avoidance of overlapping UI elements).
- The user should be presented with the same structured UI layout throughout different types of algorithm visualizations.
- NetLuke must be usable on devices with very high or low resolutions and therefore adapt its visible elements accordingly without loosing functionality.
- Desktop and mobile look & feel need to be as close as possible whereas only the visual structure is forced to adopt. This can be subsumed under the term responsive design with CSS directives and JavaScript routines.
- The mobile GUI needs to respect common usage patterns and mental models of mobile designs beyond touch interaction (hidden control elements, pre defined hardware/software buttons provided by mobile OS).
- The GUI needs to give the impression that NetLuke is an application rather than a web-site (like a RIA). Regarding the UI, this can be achieved by a fullscreen window/view throughout all device categories and extensive use of Cascading Style Sheets (CSS) on all visual UI elements.

3.4 Developer Requirements

Developers need to ask themselves: What should be visualized/animated? What can be visualized? What can be created? How powerful is the underlying system in displaying different kinds of algorithm types? These questions can be subsumed under the term "system expressiveness" introduced in Section 1.2. The term can be considered as a prerequisite for the requirements of educational effectiveness, as only a high level of expressiveness can guarantee a good solution. The system's ability to visualize certain topics is largely a technical one, where as future developers must be able to understand how the system is working. A deep understanding of implementation details is not necessary

though. Therefore developers will have their own requirements which needed attention in the design phase.

Before discussing which requirements the system needs to satisfy, we have to outline who we expect to contribute in the future. Which skills bring developers to the table and which basic knowledge is required? We expect that the majority of developers will be students from the University of Vienna. The assumption is based on previous practices as LUCAS was extended by students from our home University. We expect that NetLuke will be indeed promoted as the successor of LUCAS and therefore adopt most of its surrounding standards. LUCAS followed a "Java only" approach: It is entirely based on Java and its object oriented paradigm. Algorithm logic as well as visualization was encapsulated in mostly one single package. Judging from a quick overview over the projects source code, many implementations heavily mix presentation, control and algorithm core aspects. As pointed out NetLuke separates aspects, as front-end technology is very different from back-end technology. This imposes new challenges for inexperienced developers. We expect that module development for NetLuke will be considerably more difficult than it was for LUCAS. Aside from programming tasks, a major challenge is the conceptualization of interaction patterns which are necessary in terms of communication, as seen in Figure 3.2. Developers need to define precise interfaces (methods) which enable algorithm control from the UI. Exchanged messages either contain data representations, control commands or both. Chapter 4 discusses the concept in detail.

However, many students are capable of programming decent programs in Java, because Object Oriented languages such as C++ or Java are substance in computer science lectures. Lots of projects and tasks are solved with little to big Java programs. JavaScript will be an additional challenge for students, not only because it differs vastly from Java aside from syntax, but also because it is not thought in-depth in lectures. One can argue about the priorities of programming languages in computer science, but the fact of the matter is that JavaScript was not as important (powerful and explored) 10 years ago as it is now. For NetLuke average Java skills, but advanced JavaScript knowledge is required. Therefore we intended to keep complexity and overhead for developers as low as possible. The following requirements represent our thoughts concerning support for developers on a conceptual level:

- In order to support developers properly, the system needs to follow a framework like approach without being to restrictive regarding implementation details. We argue that module developers need freedom in their designs and implementations. Therefore we do not speak of a framework NetLuke is providing. Instead we rather conceptualized the embedment of modules in the system environment.
- Developers need an easy way of connecting their Java implementation with the HTML front-end 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, control flow, 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 will not have the ability to test their implementations equally on all platforms [7].

- Developers should be given the ability to test server-side implementations extensively and independent from the front-end, creating more robust modules from the beginning.
- NetLuke, and its libraries, have to provide some sort of debug functionality in order to support module developers in finding semantic and syntactic errors.
- Module authors require documentation, code examples and templates to get started. Comprehensive guidance will be presented in Part II of this thesis.

3.5 Conflicts

Many of the defined requirements, create conflicts, which need careful evaluation. Several aspects directly influence the architectural approach and its implementation, because some requirements interfere with each other. Problems arise through developer- and module concepts in combination with the concept of server-side algorithm execution. Another obstacle are client demands because of their inherent heterogeneity (web browsers). This Section is about conflicts as well as general problems and how we intent to solve them.

Mobile versus Desktop

This conflict line comprises several aspects and characteristics. In Section 3.2 we postulate platform independence through HTML5 capable web browsers. In Section 3.3.3 we claim that NetLuke should not feel like a web-page instead like a RIA application. We also want to achieve that modules and therefore developers are independent from the client platform. This set of requirements raises several problems:

- Small physical screen sizes are a big problem for NetLuke and many other web projects. Mobile interaction is very different from desktop usage. Not only do you use your fingers for pressing buttons, which in return, need to respect the lack of precision when tapping a button (adapt in dimensions), but also the screen size is very limited. As a result, the UI control elements need to be generally larger in size which results in a domination of the view through the UI elements. This leads to problems. For instance in data structure insert operations. If one wants to insert a value into a Binary Tree, the visualization cannot be blocked while the user is entering the value and confirms it with pressing a button. The UI needs to be different from traditional mobile interaction patterns in the sense that the effects of supplied data input must be visible to the user.

- The web application on mobile devices should feel "snappy" and responsive. This can be extremely difficult to achieve, because we expect mobile users to have drops in their 3G connections to lower speeds. UI response times are very likely to vary,

[7] Apple iOS applications can only be tested within the iOS developer tool Xcode, which requires an Apple OSX compatible system. Since OSX is not as widespread/compatible as Linux or Microsoft Windows, we expect that many developers wont be able to test their modules during development on this platform at all.

because slow response times during client server data exchange have an immediate effect on algorithm visualization. This is a big problem in the whole concept of "client presentation - server execution" model. Since a user controls an algorithm instance on a server, its state needs to be synchronized with its client-side visualization. Therefore, 9 out of 10 times, Ajax calls are not asynchronous. Hence communication results in UI "blocking".

- HTML5 Animations and graphical effects are beautiful and strengthen system expressiveness but are extremely hardware resource intensive. Thorough analysis of mobile platforms showed that older Android devices offer little to no hardware acceleration support [8], which results in poor rendering performance. Since we want to target as many platforms as possible we cannot simply presume/activate hardware acceleration in our Android implementation. Apple iOS performs considerably better, but overall JavaScript performance is still low. Since developers may not be able to test their implementations on mobile devices we expect performance issues especially if a module is using resources inefficiently.

- In contrast to desktop browsers, mobile browsers need to respect their hardware limitations much harder. Since HTML5 is not fully specified each browser vendor evaluates the degree of standard support. Many Google Android versions have different browser implementations, which sometimes contain bugs in the HTML5 canvas part we heavily rely on[9]. On top of that, hardware vendors modify the underlying Android OS with own extensions (e.g. HTC Sense, Samsung TouchWiz) which sometimes causes side effects in browser handling. Summarized: Mobile browser implementations differ vastly from their desktop counterparts, making HTML5/CSS3 UI development very difficult [4, p. 7].

Creating the exact same user experience on every device is extremely difficult to achieve. We expect that the current prototype will be unusable on some mobile or tablet devices, whereas others (vendor unspecific) will have less difficulties. We cannot even make tentative predictions in this matter. To reach a compromise we needed to prioritise a platform. We decided to follow a "Desktop first" approach. We know that NetLuke unfolds its full potential only on a desktop platform or a very fast tablet. First of all we expect users to have displays ranging from 15 inches to 22 inches with resolutions up to 1920x1080. In addition visualizations with transitions and animations run smoother on modern browsers like Google Chrome or Internet Explorer 9/10. UI elements on a desktop platform do not feature the mentioned contradiction between small sized screens and the need for bigger visual buttons. The compromise does **not** mean we abandon mobile devices. The sheer amount of mobile environments with unpredictable factors just creates the need to focus on one platform first.

[8] Android supports hardware acceleration since Android 3.0 aka. "Honeycomb". However, this tablet exclusive version of Android never reached popularity among tablet vendors. Newer Android 4.0 devices are able to support hardware accelerated rendering, but it is almost useless in the browser as it produces visual distortions or other incompatibilities (personal experience).

[9] At this point we cannot provide scientific sources as old bugs get fixed and new ones arise in every version of Android. A look at http://stackoverflow.com/ tagging 'HTML5', 'scrolling' and 'Android' will give an impression

One may argue that native applications on Android or iOS will not exhibit most of the discussed problems [4, p. 2]. This is true, however this would make it impossible for our type of developers to extend NetLuke with JavaScript modules (at least not following our concept). Web applications are necessary. Amongst various other reasons the authors do not have the resources to develop many different platform specific applications, at least not within the scope of our master thesis.

Modules versus Web Technology

Feature rich web applications, and NetLuke is no exception in this case, heavily rely on JavaScript libraries which usually encapsulate functionality needed for specific tasks. JS libraries are fundamental building blocks for web driven software components. Sadly, and this is partly due to the nature of JS, every library exhibits its own style of message passing, arguments, object creation and configuration (API). JS libraries are not comparable to Java libraries in this case. Most of the JS libraries we use in our project, follow specific design philosophy and coding styles. Therefore, developers need to carefully study online documentations and tutorials if available. As a matter of fact HTML5 and JavaScript are not ideal languages in regard to our module based approach.

Complex web applications are often driven by a Content Management System (CMS) like Drupal, TYPO or Joomla which provide a framework for content publishing [39, p. 408]. They typically hide implementation details and enable direct content manipulation through an administration GUI. This is not planed for NetLuke. Even though NetLuke uses "classic" HTML technology and JavaScript/AJAX, we cannot "protect" developers in all cases from adoptions which have to be made either regarding UI layout or the potential expressiveness of algorithm visualizations. In Section 3.3.1 we stated that modules need to be "independent from the system environment". Client-side web technologies do not allow complete decoupling of components from others. First of all, following a strict understanding: Java is a high level programming language whereas JavaScript is still a scripting language. Second: JS does not support, or at least not very well, handy things like encapsulation, easy inheritance, method overloading, polymorphism or simple interface creation. Third: Java runs "on its own", where as JavaScript requires DOM/HTML and a web browser as a runtime environment.

Developer Viewpoint

In Section 3.4 we argued that developers "[...] should not be concerned about the target platform" and that they need an "[...] easy way of connecting their Java implementation with the HTML front-end through the interfaces/functionality provided by the system". We also claimed in Section 3.3.3 that "look & feel" of visualizations as well as the UI needs to be consistent in their presentation to the user.

The above requirements contradict, because our client-side library (see Section 6.5) can not provide enough abstraction between every platform, in the sense that a JavaScript module needs to respect the case of a mobile platform client. As we will see in Part II, this does not impose complex tasks for developers, but is accountable for some programming overhead. The problematic nature is due to factors mentioned in Section 3.1.1 and 3.1.2. However, we cannot completely rule out situations which lead to unpredicted UI errors,

unless a developer is aware and implements workarounds. Developers needs to adopt and need to make use of libraries and CSS files which enable "Responsive Web-Design" (RWD). What is RWD and how does it work?

The term comprises several techniques such as CSS3 media queries for layout and design adaptions, or browser feature detection through JavaScript. As soon as a client is detected and classified (its characteristics are known), the corresponding CSS stylesheets ensure correct, reasonable layout and styling. NetLuke is not different in this case, because UI adaption is realized by CSS stylesheets, therefore developers need to use correct class attributes for their HTML elements (mainly control buttons). If they do not, layout distortions may occur. In NetLuke, the concept of responsive web design through CSS stylesheets is not applicable for objects within `<canvas>`. Adaptions on visual features (blur effects, transitions, animations, etc.) need to be made programmatically in JavaScript, within the module implementation. Because of the problems concerning differences in mobile browser implementations, tweaks, hacks and workarounds have to be applied to overcome bugs in web browsers. This is very bothering for developers. We tried to keep those tasks to a minimum by adding most of the workarounds into a separate JS file (`loader.js`), which reduces the burden. It is simply more difficult to provide modules if multiple technologies and cross platform development is involved.

Chapter 4

Module Architecture

We begin our discussion of architectural concepts with the conceptualization of algorithm and data structure modules (further referred as just modules). Modules are the smallest self-contained components within NetLuke. We discuss system independent building blocks before we continue with server- and client architecture to illustrate the strategy we applied in the development process.

We begin this Chapter with an explanation about what a module is, and how it is crafted. Therefore we introduce the building process in Section 4.1. Modules consist of several components, one of which is a core library. We explain its encapsulated behavior, components and routines in Section 4.2. On the basis of core components developers create Java implementations, which represent NetLuke's business logic. We further explain the concept of JavaScript – Java interaction through RPC messaging, how exchanged messages look like as well as their creation process in Section 4.3. Next is discussion about how programming artifacts can be aggregated to deployable modules (Section 4.4). The closing Section 4.5 provides an introduction to client-side JavaScript components essential to modules.

4.1 Module Building Process

In the previous Chapter 3 we stated that modules on the deployment level (files), consist of two artifacts: A JAR file and a JS file (as seen in Figure 3.2). The Java Archive contains several components which comprise a deployable module on the server. It does not contain any JavaScript code. We call this part of a module *Java module*. The JavaScript file is developed after the Java Archive and is deployed within the NetLuke server project in the WEB-INF directory. Together with the higher level UI, it represents the front-end of a module. Both parts of a module communicate over RPC message passing.

A complete module is developed stepwise[1]. But how is the order of events when creating a new module? In general, a developer has to ...

1. ... create a new Eclipse Java Project and incorporate the *netluke-core.jar* file as the primary application library (Section 4.2). The Java implementation provides logic to the topic at hand e.g. and algorithm or a data structure, as well as methods to control its execution.

2. ... ensure a high level of module robustness by either using or adopting the test-cases within the NetLukeTest project. It is further required that a developer tests its implementation manually, before deploying.

3. ... deploy the module and configure the *netluke.conf* configuration file (see Section 4.4 for further reference) accordingly.

4. ... create the necessary JavaScript visualization routines and control components by using KineticJS (see Section 6.4 for an introduction to the graphical library) and our custom loader.js component (see Section 6.5).

5. ... integrate/embed visualization and control components into the client system environment (see Chapter 6).

6. ... ensure compatibility with all targeted client platforms, by manual testing the module on desktop, Android and iOS devices. Of course this is only applicable if a developer has access to required resources.

4.2 NetLuke Core

NetLuke is the designated successor of LUCAS. Hence we adopt, but also modify some of its ideas and architectural concepts. We reviewed the tool in Section 2.3.2 and evaluated it in Section 2.4.1. Since we implemented extensions for LUCAS two years ago, we are familiar with the source code and with most of the inherent problems to it. Despite the mixture of logic, control flow and visual presentation within its topic implementations, we were surprised by the sheer amount of redundant code. The tool is lacking an overall concept to reduce code complexity through abstraction – meaning components which can be re-used throughout all kinds of algorithm or data structure implementations. In other words – LUCAS was missing comprehensive libraries for integrative algorithm and data structure creation. As a matter of fact, many topics in the field are very distinct from each other. For instance: What is the component wise "intersection" between *quicksort* and a *binary search tree*? Even within the broad topic of data structures it is not easy to find similarities at first sight. So we asked – what does a *trie* has in common with a *double linked list* other than being a data structure?

Of course structures (the "skeleton") or determining functionality of topics can not be unified, but at least their computational elements can be. Algorithms and data structures alike work with data, which is either integer numbers, characters or strings. Representation of this data can be unified, because their abstract type is "just data". We decided

[1]Part II is dedicated to module development and provides direct instructions and guidance for developers.

to develop a small but usefull library which facilitates information hiding, code re-use, inheritance and reduction of complexity. We also wanted to help developers in creating new modules by providing a basic implementation structure. NetLuke's core library project provides those aggregable building blocks for algorithm and data structure module creation. The following paragraphs explain the contents of the library on class level.

NetLuke Core library consists of the following Java packages: **at.ac.univie.netluke.core.** (1) **component**, (2) **utils**, (3) **datastructure**, (4) **graph**, (5) **sort**, (6) **annotation** and (7) **test**. We begin with the *component* package as the classes within this package comprise most of the elementary building blocks helpful for module development. However the current state of the library does not cover all possible types of module topics[2], but can be seen as a first effort for development tasks.

(1) netluke.core.components

- The *Element* class is very basic. Its constructor takes either a char or an integer as an argument. Its main purpose is to wrap data e.g. for XML representation purposes (see Section 4.3.1 for a detailed discussion). This class is the most abstract building block, from which other classes such as the *Edge* component is derived.
- The abstract *Node* class can be indirectly used as an Element container for arrays and hashing algorithms or otherwise.
- *Edge* is a subclass of *Element*. It constitutes the same basic functionality as an *Element* together with additional start- and endnode attributes. It acts as a superclass for the *GraphEdge* class.
- *GraphEdge* is a subclass of *Edge*. This class is specifically designed for usage in graphs. Its constructor takes *GraphNode* objects as arguments. The *directed* attribute indicates if the edge is directed. This is needed in various graph algorithms such as Topological Sort or Depth/Breadth First Search.
- The *GraphNode* class is a concrete implementation of *Node*. It represents nodes in a graph with distinct attributes such as a name, if the node is visited, or if it is acting as a start node.
- A *TreeNode* is used in in tree and trie implementations. It allows saving of more than one *Element* object and implements functionality to access parent and child nodes.

Of course all of the provided components can be adopted, extended and modified by developers. However, API changes have serious effects on dependent components and should be avoided. Developers need to respect existing implementations which (heavily) depend on the library. To give more insight in this matter, Figure 4.1 illustrates the class diagram of the *component* and *utils* package including methods, dependencies and generalizations. Figure 4.2 maps the above core components to actual visualizations used in the Binary Tree/Prim algorithms modules. Note that the depicted elements styling is completely independent from the core libraries' element definition.

[2]Until now we focused on sorting algorithms, trees and graphs. Topics concerning hashing will follow with later releases by future developers.

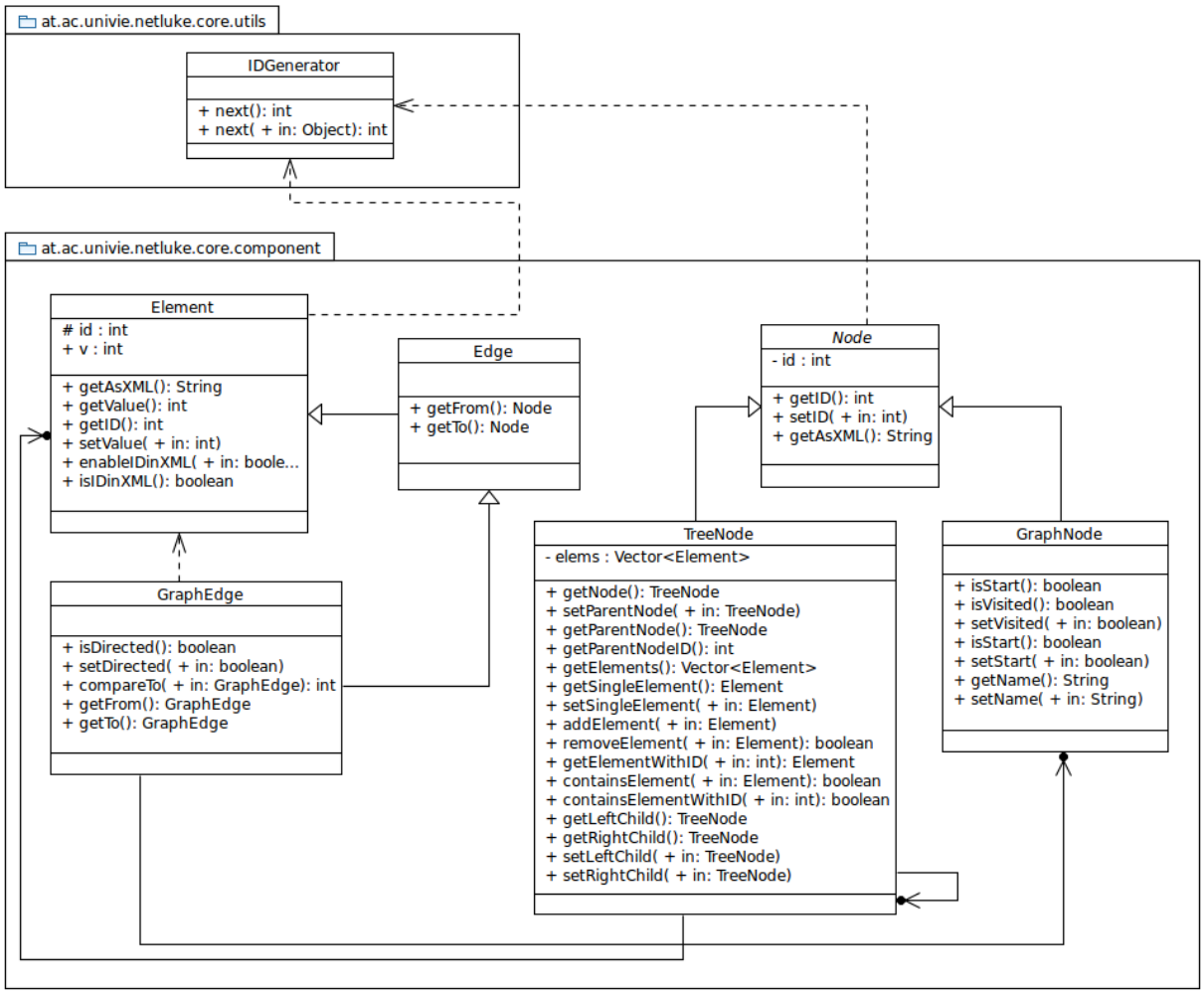

Figure 4.1: UML - NetLukeCore

(2) netluke.core.utils

- Methods of the *IDGenerator* class are statically accessed. The first ID generating method – *next()* returns an ID based on a simple integer, whereas an initial value gets incremented for each method call. The second method takes an Object as an argument and calculates the objects unique hash[3]. As we will see in the Sections 4.3 and 4.3.1 an ID is very useful to identifiy an Element, a Node or other components. Therefore we give any component an ID for this purpose. This ID does not need to be unique throughout the whole system, just within one algorithm instance.
- *DeepCopy* offers a *clone()* method for deep object copy tasks. This can be very useful when a module needs exact snapshots of its state e.g. tree structures or graphs.

[3]Generates an integer ID based on *System.identityHashCode(o)* for the provided object e.g. an Element, which we consider as unique enough for its purpose, as an Element or Node ID just needs to be unique within a given module instance and not throughout the system.

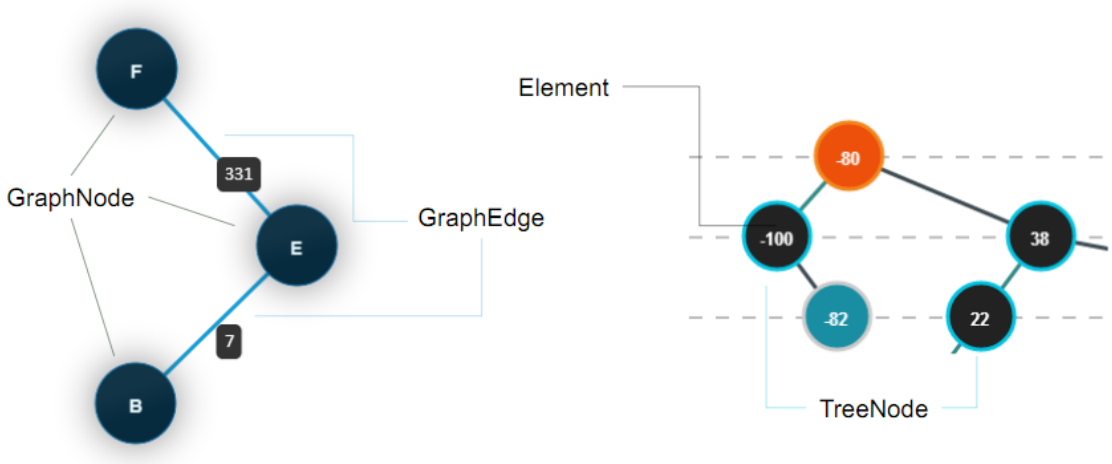

Figure 4.2: KineticJS - Component Visualization

(3) netluke.core.datastructure

- *DataStructureMethods* is a Java Interface which defines functionality every implementing data structure should provide.
- The *BinaryTree* Class implements a simple binary search tree with characteristic functions. It is used in the BinaryTree plugin
- The *Heap* Class implements a Heap which is used in the HeapSort plugin. It can be extended and used for a future (Priority) Queue implementation.

Basic data structures, listed in (3), are built upon core components. They can be found in the netluke.core.datastructure package. Since some data structures can be used among different topics, we implemented a *BinaryTree* and a *Min-Max-Heap* together with a Java Interface. It defines the basic range of functionality for all data structures. Related interfaces are used in (4) and (5). Figure 4.3 illustrates the *GraphMethods* interface together with components which comprise the Prim Algorithm module. The purpose of interfaces of this kind is to guide a developer in defining relevant functionality, because a module needs to provide certain characteristics in order to qualify for system integration.

(4) netluke.core.graph

At the time of writing this thesis, the package only features the *GraphMethods* interface similar to *DataStructureMethods* and *SortMethods* in (3). However, we encourage developers to contribute re-usable graph implementations for future graph algorithm modules. For instance a simple abstract graph class with concrete descendant sub-classes.

(5) netluke.core.sort

- The *SortMethods* interface defines basic functions e.g. `next()`, `back()`, `run()` etc. *next* resembles a step forward in algorithm execution, *back* a step back accordingly whereas *run* executes until a solution is found.

- The *SortAlgorithm* class is acting as a superclass for various in-place/array sorting algorithms. It offers value insertion, deletion, swapping and representation methods. It wraps a simple integer array. Sorting algorithms which are categorized as in-place/in-situ operate within their defined data structure e.g. an array. This kind of sorting algorithms usually require related to similar (convenient) operations on arrays.

(6) netluke.core.annotation

In the Requirements Section 3.3.1 in Figure 3.2, we mentioned a *@NetLukeMethod* Java annotation type. A developer needs to annotate methods and constructors in order to enable method and constructor discovery within the NetLuke system. NetLuke discovers modules through a configuration file – *netluke.xml*. We explain the underlying concept in Section 4.4. Information of methods and constructors encapsulated in modules is gathered during runtime by a special controller component (see Section 5.3 for further discussion). The introduction of an annotation concept is based on security and complexity concerns. Since NetLuke uses generic interfaces (see Section 5.2), module developers need to "bind" their methods onto the pre-designed endpoints. This binding is achieved by the *@NetLukeMethod* annotation. Only if a developer annotates a (strictly public) method/constructor it can interact with the service layer of NetLuke.

(7) netluke.core.test

This last package is all about testing core library components. Included test classes utilize the T2 automatic JUnit testing framework [31] to instantiate and test objects such as the *BinaryTree* or *Heap* implementation[4]. In essence, T2 calls constructors and methods in undefined random order, which can lead to unexpected method call sequences. This results in unforeseeable relationships/execution behavior and of course – errors. Since module development in a client-server environment is complex enough, especially for inexperienced developers, components need to be tested before they are used. Of course, manual testing of core components, for instance with the JUnit framework, is also done in the *netluke.core.test* package. Testing of actual modules is happening within the separate NetLukeTest project.

4.3 From Components to Modules

By using the components package and the according datastructures, a developer is able to create modules (1) faster, the module is more (2) robust and (3) much easier to integrate in the system environment. An integrated view of how a Java module is assembled, is provided in Figure 4.3 illustrating a sample Prim algorithm module. According to the listed developer requirements in Section 3.4 creation efforts should be as low as possible. By using the well known Eclipse IDE, and simple Java Projects in addition with core components, developers can implement and test implementations independent from the NetLuke server project (NetLuke). Like any other Java program, a module is structured by packages. Future contributors are advised to organize classes as suggested here.

[4]Part II of this thesis will discuss the T2 framework extensively.

The package structure of a Java module is as follows: The *at.ac.univie.netluke.plugin* package contains the main implementing source file – the business logic of a module, usually named after the algorithm itself e.g. `ExampleAlgorithm.java` and an according HTML file `ExampleAlgorithm.html` containing a description of the algorithm. Listing 4.1 illustrates how the structure (markup) of the description file **has to** look like. During system startup the description to the particular module is loaded automatically and integrated in the UI. Once again, this topic will be discussed in Part II of this thesis.

LISTING 4.1: HMTL - Algorithm Description File

```
<div id="ExampleAlgo">
  <h1>Example Algorithm Description</h1>
    <div>Description goes here</div>
    <!-- List some facts -->
    <ul>
      <li>Complexity: ...</li>
      <li>Working mode: ...</li>
    </ul>
    <div>
      <!-- Add Author information -->
      <div style="float:left;">
      <h4>Author</h4>
      Name: Example Author<br>
      E-Mail: example@email.com<br>
    </div>
    <div>
      <!-- Add optional images -->
      <a href="./images/sort/ExampleAlgo.png">
        <img src="./images/sort/ExampleAlgo.png"/>
      </a>
    </div>
  </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">
    <li>Reference link(s) go here </li>
  </ul>
</div>
```

Additional Java classes are grouped within the at.ac.univie.netluke.plugin.NAME package e.g. at.ac.univie.netluke.plugin.ExampleAlgorithm. By design, developers are forced to implement functionality to go back and forth in algorithm or data structure execution (History Mode). This functionality should be separated from the main implementation class to allow separation of concerns. If correctly implemented, these features can be tested with plain Java classes and custom methods before deployment. As soon as a Java module is properly tested, (manually in the test package and automatically in the NetLukeTest project) it is exported as a JAR file and deployed within the *lib* folder of NetLuke. Section 4.4 discusses the deployment process in detail. The next Section 4.3.1 covers a very important aspect within the module architecture, because we unfold how client server interaction, module communication and messaging works.

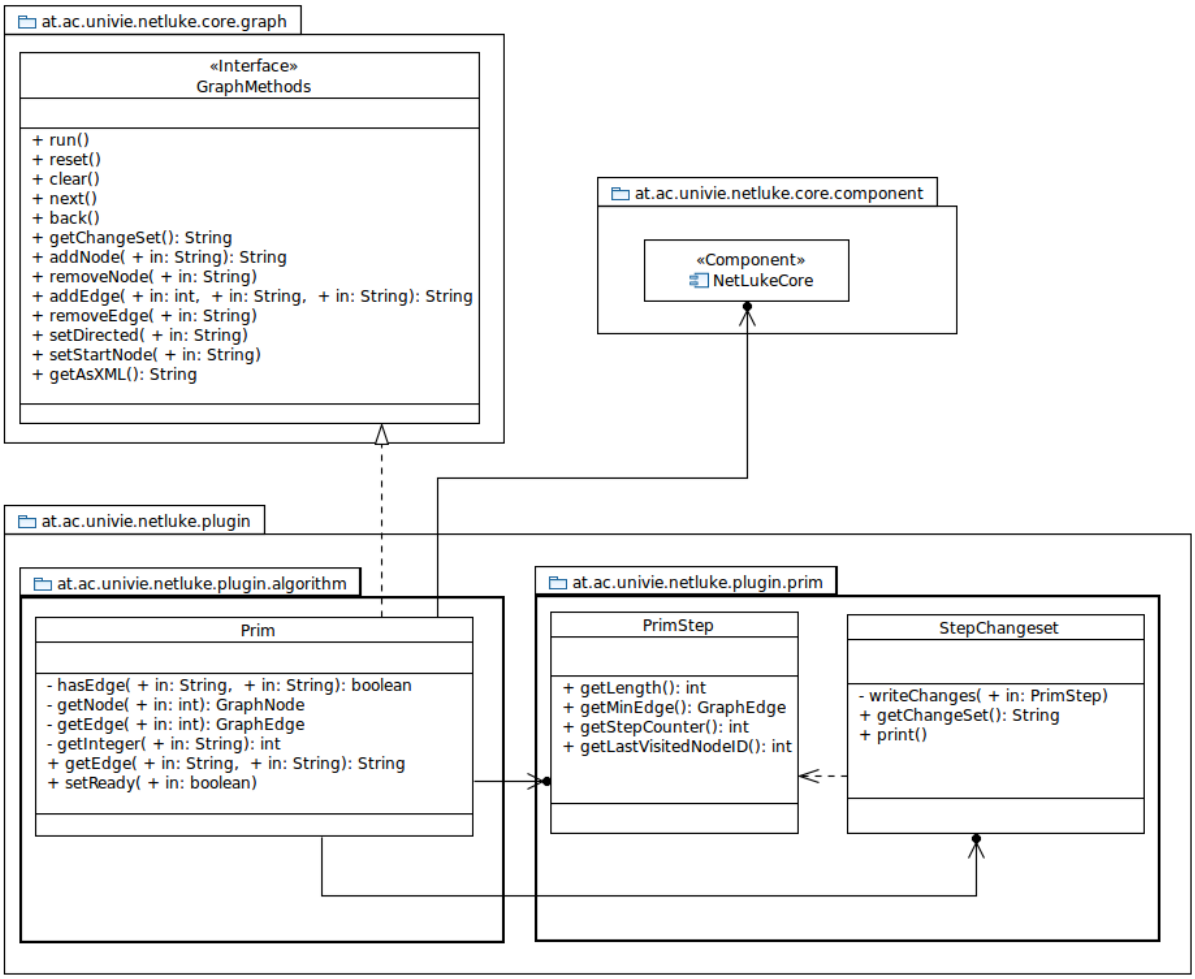

Figure 4.3: UML - Prim Algorithm Module

4.3.1 Module Connectivity

The concept of client visualization and server processing requires a communication technique and an appropriate message format. JavaScript components do not (and should not) know about the server-side processes involving objects, references and attributes. Therefore we introduce a message format to map object presentation on to business logic and vice versa. NetLuke uses Remote Procedure Calls (RPC) as the communication technique. Server components realizing RPC are explained in Section 5.2.1. The communication sequence is as follows:

1. The client calls a remote function provided by a Java module.

2. The server dynamically invokes the function and responds with a specific set of commands or data needed for visualization tasks.

3. A server response comprises client instructions usually embedded within XML messages – hence the *getAsXML()* methods in core components and business logic implementations.

4. As soon as a client receives a response, the message is parsed and interpreted according to a developers semantic conceptualization of the representation.

In order to relieve developers from complex (XML) message creation the core library facilitates the generation of XML messages by its *getAsXML()* (or analogically named) methods. Developers just need to call the methods accordingly to assemble representations of their own data. Listing 4.2 illustrates a short example of how XML messages are created. The `getArrayAsXML()` method (within the *SortingAlgorithm* class) assembles an XML array representation based on the elements' `getAsXML()` method. This XML representation is interpreted by client upon request.

LISTING 4.2: Example - Array XML representation creation

```
1  public String getArrayAsXML() {
2          StringBuffer xml = new StringBuffer();
3          xml.append("<array>\n");
4          for (int i=0; i < getArraySize(array);i++) {
5                  if (array[i] != null) {
6                          xml.append(array[i].getAsXML());
7                  }
8          }
9          xml.append("\n</array>");
10         return xml.toString();
11 }
```

As a matter of fact, NetLuke is not strictly bound to XML messages encoding information. JavaScript Object Notation (JSON) as well as custom data representation notations/encoding schemes may be used as well. However we still recommend using XML. It may lead to a shortened development process, because a lot of functionality is already provided for client-side XML message parsing. Already implemented modules use XML messages. In the design phase we decided on using XML markup as its hierarchical structure and syntax is well known among students – hence easy to understand, besides XML can be used in a variety of different programming environments such as C++, Java or Python which use Web Service connectivity. Since exchanged messages are small in size, the relatively high cost of transferring and parsing XML messages compared to – for instance JSON-formatted data – does not impose a problem [29, p. 3]. Yet another big advantage of XML markup is the inherent syntactical precision.

How do XML messages (1) look like in NetLuke? 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>". In case of an *Element* the resulting XML representation looks like this: <element $[optional : id = "int"]$>value</element>. When are messages (2) transferred? Basically whenever a client requests an algorithm instance to execute a step forward/backward or creates/-modifies data in the Java object. In this case a client will receive a complex XML message, containing information about the modified underlying data structure. What needs to be taken into consideration when working with RPC-AJAX requests (3)?

(1) XML Representation

A typical array representation, used in our bubble sort and selection sort modules, looks exactly like in the following Listing:

LISTING 4.3: XML Array representation

```
1 <array>
2  <element>1</element><element>0</element><element>9</element><element>3</element>
3 </array>
```

One may argue that the notation illustrated in Listing 4.3 is inefficient and unnecessary because of the `<element>` tags. A simple string containing the values separated by a delimiter would be sufficient in the case of an array. On the contrary, data structures can get complex and their representations need to be assembled fast and precise. Would a simple string of values be enough for the representation of splitted arrays in a quicksort instance? Maybe, but in this particular case a physical structure allows to deduce logical structures – hence XML is a human readable format, which can be a huge advantage at debugging.

After every single remote procedure call, a client visualization needs to be aware of the current state of an algorithm (the data stored in the object). This state is encapsulated within the servers response. However, in most cases it is not necessary to transmit a complete snapshot of a data structure representation for every step undertaken. In most cases, client presentation logic, just needs a minimum of information for visualization purposes. Therefore we chose to facilitate the concept of "changesets" similar to "revisions" in software versioning and revision control (SVN). Changesets basically indicate *what has changed to the preceding state* with very few instructions. If properly designed, they can be very expressive. Listing 4.4 below shows a typical changeset from the bubble sort module.

LISTING 4.4: XML Sorting Algorithm Changeset

```
1 <changeset>
2  <step>1</step>
3   <i>0</i>
4   <j>1</j>
5   <swapped>
6         <pos_a>1</pos_a> <pos_b>2</pos_b>
7         <element>-17</element> <element>75</element>
8   </swapped>
9 </changeset>
```

The XML message contains information about the position of the loop counter variables, if a swap occurred including positions and which elements (values) were involved in the exchange process. After receiving, the visualization system interprets the message and changes the visual representation accordingly. This example shows, that client processing tasks are reduced to message interpretation and rendering tasks only. We continue with a more complex example. The following Listing 4.5 shows a representation of a weighted, undirected graph used in the Prim module. Graph edges reference their node pairs with IDs. Nodes are independent from their connecting edges. Once again the XML structure is easy to assemble, readable, but appropriate for the task. Although the message is about 1kb in size, it just needs to be transferred when a client creates or (re-) loads an instance.

LISTING 4.5: XML Prim Algorithm representation

```
1  <prim>
2    <startnode>3221204</startnode>
3    <actualnode>null</actualnode>
4    <nodes>
5      <node id="3221204" start="true">A</node>
6      <node id="18027118">B</node>
7      <node id="14134009">C</node>
8      <node id="10665941">D</node>
9      <node id="13216403">E</node>
10   </nodes>
11
12   <edges>
13     <edge id="20251621" from="3221204" to="13216403"
14                 directed="false">2</edge>
15     <edge id="14127272" from="13216403" to="18027118"
16                 directed="false">10</edge>
17     <edge id="9851620" from="13216403" to="14134009"
18                 directed="false">11</edge>
19     <edge id="25928745" from="18027118" to="10665941"
20                 directed="false">33</edge>
21   </edges>
22 </prim>
```

Compared to the message in Listing 4.5, the according changeset is much smaller in size (Listing 4.6). It consists only of the needed components which changed from one execution step to the other. Notice the values in the lines 4 – 7 changed from edge weights to IDs. This information is sufficient for a complete visualization of steps in Prim, as IDs are used for referencing objects.

LISTING 4.6: Prim Algorithm changeset

```
1  <changeset>
2    <step>1</step>
3    <length>19</length>
4    <nodeFrom>17066018</nodeFrom>
5    <nodeTo>25803197</nodeTo>
6    <nodevisited>25803197</nodevisited>
7    <edge>32815697</edge>
8  </changeset>
```

The next code listing 4.7 features a binary tree representation in XML. Once again IDs are used to connect and reference nodes – in this case associated child and parent nodes. Trees can reach a high level of complexity, while representation efforts need to be low. Note: If a TreeNode comprises more than a single value, for instance in the case of a B+ or 2-3-4 tree, it contains more than one element tag, hence multiple values.

LISTING 4.7: XML BinaryTree representation

```
1  <tree>
2  <node id="4406154" parent="" left="" right="222068">
3    <element>−44</element>
4  </node>
5  <node id="222068" parent="4406154" left="25758823" right="18605439">
6    <element>88</element>
7  </node>
```

```
 8  <node id="25758823" parent="222068" left="" right="19399109">
 9    <element>−19</element>
10  </node>
11  <node id="18605439" parent="222068" left="" right="">
12    <element>95</element>
13  </node>
14  <node id="19399109" parent="25758823" left="" right="">
15    <element>67</element>
16  </node>
17  </tree>
```

(2) Remote Procedure Calls

Whenever a user requests an algorithm instance to conduct a step forward in execution, or whenever a value is inserted into any kind of data structure, the client transparently calls a remote Java method within a module. This simple remote method takes at least one argument, which is exactly the name of the method within the algorithm instance (the background processes of RPC are explained in Sections 5.2.3 and 5.3). We start with a simple, but very typical client side JavaScript remote procedure call, that looks like in the Listing 4.8 below:

LISTING 4.8: Example: Client Remote Procedure Call

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

The *remote* object in line 2 is the client-side RPC proxy (see Figure 5.2), whereas the `invokeMethod('argument')` is the actual remote service method to call. It maps the *'getAsXML'* parameter to a method within the module – in this case the `getAsXML()` method in the Prim module. At this point the question arises why methods of Java classes are not called directly through RPC? For instance like: `prim.getAsXML()`. Why do we need a `invokeMethod('name')`, which in return calls the module's method? NetLuke is designed to support all kinds of modules, not just the samples provided. Modules may be provided by different developers, who might preference different programming styles, concepts and approaches to problems. As a consequence we can not even roughly estimate a set of methods including their programmatic specification. By concept, modules and their implementation details are unknown to NetLuke. Therefore a `prim.getAsXML()` method is not practicable – **it is not abstract enough**. The applied concept of "dynamic binding" is supported by the fact, that a module has similar named methods e.g. `insert(int value)` or `insert(double value)`. Those methods can not be exported to the service layer (see Section 5.2 at the same time, because on the interface level, they are not distinguishable from each other by JavaScript. Instead we designed components (NetLukeDWR and NetLukeController) reducing the set of possible methods across modules to a set of generic methods. `invokeMethod('<name_of_method>')` in line 2 maps to any function call similar to `next(), back(), clear()` or `whateverFunction()`. This way one service method can be used for all argument free method declarations in all

modules. Invoking a constructor using `invokeConstructor('<name_of_constructor>')` works accordingly.

The pattern – `invokeIntegerMethod('<name_of_method_to_call>', <values>)` and for constructors – `invokeIntegerConstructor('<name_of_constructor>',<values>)` follows the same principle applied to any single parametrized method. The *values* argument can be an array (which can contain 0 - * values) or a single value. Multiple parameters require a different syntax, explained further below. Table 4.1 lists all remote service methods, which map to module methods.

Remote Method	**Example**
invokeMethod(String n)	next(), back()
invokeIntegerMethod(String n, int[] values)	insert(5), insert(1,3,9)
invokeDoubleMethod(String n, double[] values)	insert(0.1,5.2) value
invokeCharMethod(String n, char[] c)	remove('a')
invokeBooleanMethod(String n, boolean b)	setDirected(false)
invokeConstructor(String n)	BinaryTree()
invokeIntegerConstructor(String n, int[] values)	BubbleSort(1,9,-2)
invokeDoubleConstructor(String n, double[] values)	-
invokeCharConstructor(String n, char[] c)	-
invokeBooleanConstructor(String n, boolean[] b)	-
invokeXMLMethod(String xml)	addNode() (Prim)
invokeXMLConstructor(String xml)	-

Table 4.1: NetLuke RPC methods

Relevant, but at the same time somehow more difficult to use and comprehend, are the `invokeXMLMethod()` and `invokeXMLConstructor()` remote methods. A developer might need to call methods with multiple arguments: For instance a constructor with a type signature like `(int,int,int,String,boolean)`. The previously presented approach using the `invokeIntegerConstructor('name', values)` or `invokeConstructor('name')` is not applicable for this type of constructor signature. To overcome this conceptual problem, we provide functionality which dynamically searches these kind of methods or constructors based on a XML message created on the client-side. Example: The XML markup in Listing 4.9 is "translated" by the system (NetLukeController) into the following RPC method call to: `someMethod(int a, String[] str, boolean b)` with concrete values: `someMethod(4, {"String1", "String2"}, false)`. Note: "XML methods" can fully substitute "named methods" if preferred, but are slower in execution.

LISTING 4.9: invokeXMLMethod()

```
1  <method>
2    <name>someMethod</name>
3    <parameter type="int">
4      <value>4</value>
5    </parameter>
6    <parameter array="true" type="String">
```

```
7       <value>String1</value>
8       <value>String2</value>
9     </parameter>
10    <parameter type="boolean">
11      <value>false</value>
12    </parameter>
13  </method>
```

The same call pattern regarding "unusual" type signatures can be applied to any constructor with any type signature. Listing 4.10 shows an example of a more complex constructor: `someConstructor(int a, double[] ds, boolean b, String str)` with the values `someConstructor(91, {22.1, 21.098, 42.12, 99.9}, false, "Test")`.

LISTING 4.10: invokeXMLConstructor()

```
1  <constructor>
2          <name>someConstructor</name>
3          <parameter type="int">
4                  <value>91</value>
5          </parameter>
6          <parameter array="true" type="double">
7                  <value>22.1</value>   <value>21.098</value>
8                  <value>42.12</value> <value>99.9</value>
9          </parameter>
10         <parameter type="boolean">
11                 <value>false</value>
12         </parameter>
13         <parameter type="String">
14                 <value>Test</value>
15         </parameter>
16 </constructor>
```

(3) A word on efficiency

Not only should a module developer try to reduce the amount of remote procedure calls to a minimum, but keep an eye on the message payload. A high call rate, together with large amounts of data is relatively costly (time consuming). Our own tests showed that dynamic method invocation through RPC-AJAX requests to the underlying Java Reflection routines can sometimes result in relatively high response times. The completion of a remote procedure call (1. client generates message, 2. client sends data, 3. server calculates and sends a response, 4. client interprets message, 5. client modifies visualization) can take up to over 100ms if the connection is slow, whereas message passing consumes roughly two thirds of the time. The timeframe of 100ms is considered the upper limit for "blocking" calls in a web based UI, because 100ms "[...] is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result" [28]. Therefore the system gives visual feedback whenever the client requests data in form of a loading indicator. The term "blocking call" refers to the inability of a user to interact with the UI during the timeframe a call takes to finish.

In NetLuke this behavior is caused by RPC calls, which are unable to fully execute before the server responds, because execution (callback function) is postponed. This behavior is inherent to NetLuke's architecture as a visualization is strictly bound to the state of

an algorithm instance on the server. For instance: A user cannot insert a value into a data structure on the server if the client is waiting for a previous delete operation. If calls repeatedly exceed the threshold, the UI feels sluggish and unresponsive. Therefore, a plugin contributor needs to get a feeling for blocking RPC-AJAX requests, because their potential effects on user experience are devastating. If multiple messages are scheduled for transmission, they should be small in size. If a user is already waiting for the UI to load e.g. on page load, messages can be much bigger and should contain as much information as needed in order to keep the amount of subsequent requests as low as possible.

4.4 Module Deployment

So far we have explained what is needed to create a Java module, how JavaScript interacts with Java through RPC and we introduced the concept of changesets. Once a Java module is finished e.g. it provides intended functionality and is tested, then it can be exported to a JAR file. This file acts as a library within NetLuke. In this matter, two steps are necessary:

1. The JAR file needs to be copied in the `NetLuke\WebContent\WEB-INF\lib\` folder
2. The `netluke.xml` file in `NetLuke\WebContent\WEB-INF\config\` needs editing.

The main configuration file for modules is `netluke.xml`, illustrated in Listing 4.11. It is needed for class/module discovery by the Java Virtual Machines classloader. Basically the configuration file is splitted in two separate parts. First: Configuration and discovery of module implementations and second: Basic options regarding logging. Since this Section is about the deployment of modules we discuss logging options, starting with line 31, later in Chapter 5.

LISTING 4.11: netluke.xml Configuration File

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <netluke>
3    <!-- Sort Algorithms -->
4    <plugin-category type="Sorting">
5      <plugin identifier="BubbleSort">
6        <name>Bubble Sort</name>
7        <implementation>at.ac.univie.netluke.plugin.algorithm.BubbleSort</implementation>
8      </plugin>
9       ...
10   </plugin-category>
11
12   <!-- Graph Algorithms -->
13   <plugin-category type="Graph">
14     <plugin identifier="Prim">
15       <name>Prim's Algorithm</name>
16       <implementation>at.ac.univie.netluke.plugin.algorithm.Prim</implementation>
17     </plugin>
18   </plugin-category>
19
20   <!-- Tree Algorithms -->
21   <plugin-category type="Tree">
```

```
22      <plugin identifier="BinaryTree">
23        <name>Binary Tree</name>
24        <implementation>at.ac.univie.netluke.plugin.algorithm.BinaryTree</implementation>
25      </plugin>
26    </plugin-category>
27
28          <!-- CONFIG SECTION -->
29          <config>
30           <!-- Set the loglevel:
31           0: Only critical and fatal errors are displayed (INFO)
32           1: Warnings, critical and fatal errors are displayed (DEBUG)
33           2: Display all output, regardless of type (TRACE)
34           -->
35           <log-level>2</log-level>
36           <!--
37           stdout: Standard console output
38            file : write to netluke.log
39           -->
40           <log-destination>stdout</log-destination>
41           <!--
42           DataSource according to the JNDI entry in the Context.xml
43           -->
44                   <data-source>hsql-temp</data-source>
45          </config>
46  </netluke>
```

Let's assume a module developer wants to add a Kruskal graph implementation to the system. He would insert a fairly short XML markup between the lines 17 – 18 (within the graph category) in `netluke.xml`, similar to the code snippet in Listing 4.12. Information to the elements meaning is presented in the paragraph below.

LISTING 4.12: Sample Kruskal Algorithm configuration

```
1  <plugin identifier="Kruskal">
2    <name>Kruskal Algorithm</name>
3    <implementation>at.ac.univie.netluke.plugin.algorithm.Kruskal</implementation>
4  </plugin>
```

<plugin>

Is the start tag of the module configuration block. It has one single property – the *identifier* attribute. It defines a system wide identifier for a plugin/module. It is used in the underlying JSP/JS subsystem e.g. to discover/load a JS module and in server-side instance handling, discussed in Section 5.4. A module developer defines this value here and has to continuously use it at different stages of Java/JS module development. For instance in the JS Object containing the AV functions, in the module description file, (see the id attribute value in line 1 in Figure 4.1 in Section 4.3) and most important: The name of the JavaScript file implementing AV routines must be named exactly like the identifier value e.g. `BubbleSort.js` has the identifier "BubbleSort".

<name>

The value of this element serves for presentation purposes on the UI. The name appears in headings and captions for instance in the UI's "QuickSwitch" panel (see

Figure 6.3) or within the algorithm menu. The identifier attribute is inadequate for this purpose.

<implementation>
Adds the implementing Java class to the class path. The value is similar to the name of the package containing it, including its name (fully qualified name).

4.5 JavaScript

After configuration and deployment is completed, a module can be accessed through the provided interfaces (see Section 5.2). All of its previously configured (@NetLukeMethod annotation) methods can be used on the client-side. At this stage we recommend to start JavaScript implementation tasks. Although a programming guideline covering visualization and control-flow aspects is provided in Part II of this thesis, we want to educe the architectural background to client-side development briefly.

The client architecture was influence by two major questions:

1. What is the best concept to make it as easy as possible to write and integrate new modules?
2. Which client-side JavaScript frameworks and libraries are "easy to handle", but feature rich, compatible and robust?

Of course, both questions are heavily intertwined. Therefore, we had to provide and pre define a set of 3rd party frameworks, libraries, extensions (see Section 6.1) as well as NetLuke specific functionality (loader.js) to live up to the requirements. A very important architectural detail is the loader object implemented in *loader.js*. This component honors the first question. It acts twofold: As a controlling instance of visualizations and as a "helper" object for tasks a developer has to accomplish. We discuss this special component in Section 6.5. As a matter of fact a developer does not have as much freedom in client-side JS development as on the server-side. This is due to the runtime environment (browser) i.e. the web, its architecture and its technologies. We discussed this matter in Section 3.1.2 and 3.5. Therefore module developers have to follow a specific procedure in their coding tasks and use the template functions provided. This is also needed in order to connect the AV modules to the rest of the client system, which acts as a flexible framework. The essential steps within this procedure are:

1. Creation of a JS file exactly named after the identifier attribute value previously defined in `netluke.xml` e.g. `Kruskal.js`. It is placed (amongst other modules) within the `NetLuke\WebContent\WEB-INF\js\algo` folder of the NetLuke project.
2. Creation of a JS Object within this file. The object is named like the filename itself (as done in Java) e.g. `var Kruskal = {...}`. It will contain all functionality related to visualization tasks and mechanisms for user control. Most of the routines for AV must be implemented in the object itself, whereas some can be off loaded to

the JS loader object. The provided `Example.js` will act as a template file for this purpose.

3. Definition of an `init()` method, which initializes and connects the instance of the Object to the controlling object (loader). Within this function, a developer has to create UI elements specific for the module and attach according event listeners to them.

Concerning the second question we define/provide functionality which respects the type of application we implement – an AV system running in different platform environments, allowing future developers to integrate modules without knowing JS implementation details. In this regard we decided to provide needed components through (3rd party) JavaScript frameworks and libraries. In the design process we came to the conclusion that the system requires:

1. A single comprehensive JavaScript graphical library which basically allows creation of visualizations exhibiting a high level of visual expressiveness. It needed to be easy to use, extensible, feature rich, well documented, compatible with desktop and mobile browsers, actively maintained, supported and used by a community. After long internal discussion we decided to use KineticJS[5] elaborated in Section 6.4.

2. An extensible JavaScript library which handles DOM modifications in an sophisticated and consistent manner. We decided on the widely used jQuery framework [6] for this purpose.

3. Supporting components such as feature rich libraries for overlays and dialog creation, compatibility improving components as well as general purpose modules. Ideally those libraries should encapsulate a significant set of functionality, as more client libraries increase complexity and the likelihood for severe UI errors. The libraries are listed in Table 6.1.

[5]http://http://kineticjs.com/
[6]http://jquery.com/

Chapter 5

Server Architecture & Components

This Chapter focuses on the architectural design and conceptualization of NetLuke. As elaborated before NetLuke separates business logic from presentation tasks. In the field of AV this approach can be considered uncommon, but not entirely new, as Moses [24] describes a framework which visualizes distributed algorithms across AV client machines.

In this Chapter, we want to demonstrate which server components provide the services in question and how they interact. We also unfold which technologies leverage functionality. We begin with an system overview in Section 5.1. We continue with the service layer allowing JavaScript and Java to communicate. We also explain the potential role of Web Services in Section 5.2. The third Section 5.3 explains how the system handles instance creation and unfolds module specific method discovery and invocation, complemented by configuration options in Section 5.8. Section 5.4 discusses the lifecycle of instances according to system events (Section 5.7). Instances are related to users, therefore NetLuke offers basic user- and session management capabilities (Sections 5.5 and 5.6). The persistence layer, facilitating storage of credentials, instances and serialization tasks is discussed in Section 5.9.

5.1 System Overview

So far we explained the architecture of modules, how the development process of modules looks like, how a client calls remote methods and which steps are needed to deploy a module properly. In this Section we want to illustrate the components responsible for communication tasks, module method discovery and dynamic method invocation. First we want to give an overview over the component hierarchy of NetLuke. Figure 5.1 provides information[1] about the scope of components. So far we discussed the "Java Module" compartment with the core components in Section 4.2 and the resulting module in Section 4.3. Concepts of the JavaScript architecture introduced before in Section 4.5 match the bottom compartment. We will continue with the "JSP-Frontend" component later in the next Chapter. Now we unfold the components in the middle section of Figure 5.1.

[1]The component diagram in Figure 5.1 does not make use of interface declarations for better overview purposes.

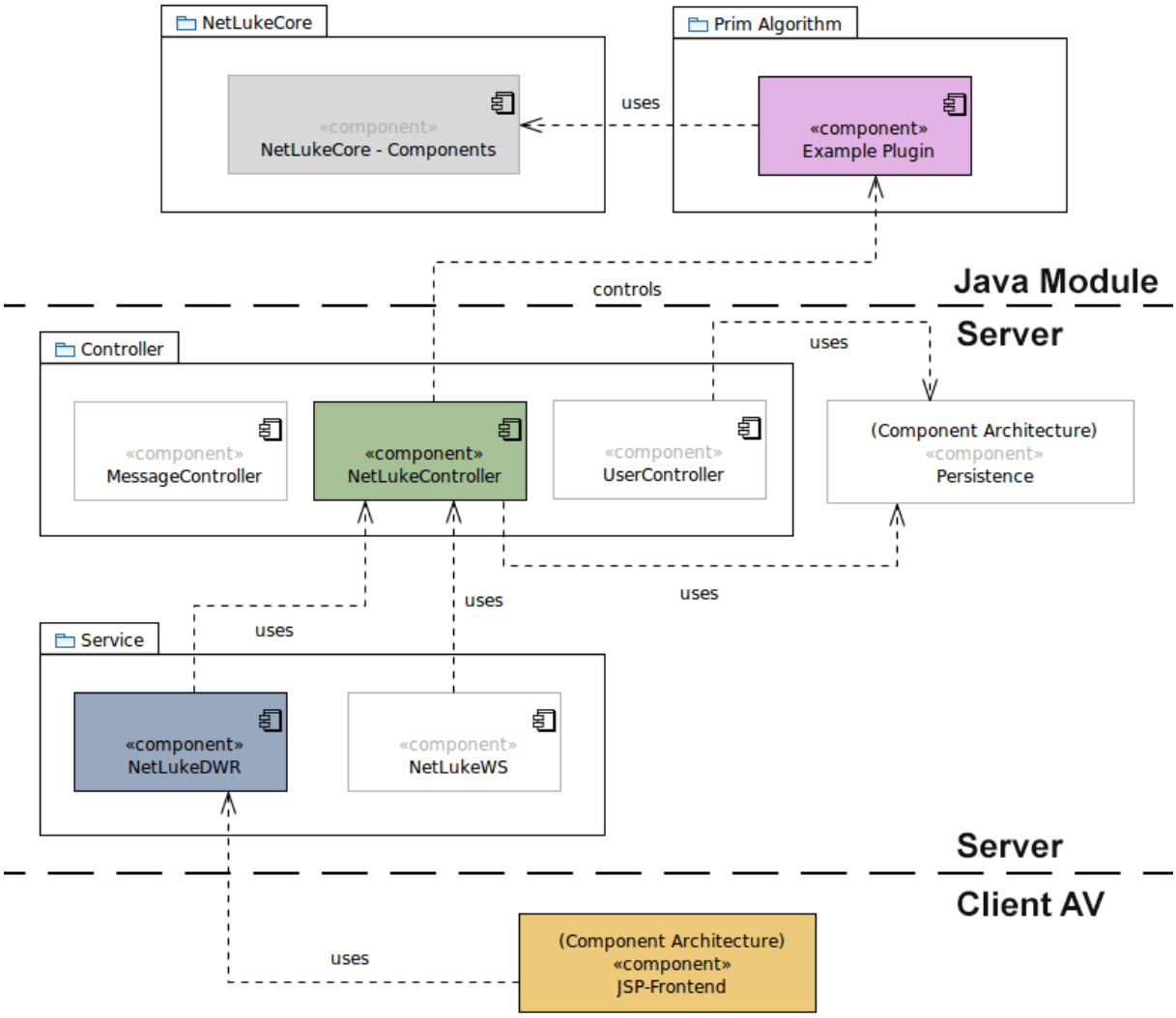

Figure 5.1: UML - NetLuke Component Diagram

5.2 Service Layer

We begin with the service endpoints NetLuke provides (Service package), because apprehension of this layer allows better overall understanding of the architecture itself. The current version of NetLuke was designed to support a variety of arbitrary clients using different kinds of technologies. However, the current prototypal implementation allows two types of clients to utilize modules.

1. HTML5/JavaScript web browser clients using RPC connections (Section 5.2.1).
2. Web service clients using stateful WS operations based on Java API for XML Web Services – JAX-WS (Section 5.2.5).

Other technologies are able to connect to NetLuke, if a matching endpoint in the service layer is provided. The underlying structure within the controller package in Figure 5.1 is mostly detached from the service layer, and as stated before, core components/modules

are not aware of the service layer. NetLuke Java modules can be used to write none distributed/network based Java applications for instance with Java Swing or Java FX. Standalone applications can work without network connectivity, when using Java modules directly. Therefore, NetLuke is not limited to a specific service/interface or client technology. If a developer decides to provide additional interface, he could create a custom one on the basis of: node.js[2], WebRTC[3], HTMl5 WebSockets[4] or Google Web Toolkit[5] or any other solution.

5.2.1 DWR Service

In Section 4.3.1 we discussed how clients call Java module methods through named requests. In the following Sections we explain how we managed to allow clients to access business logic. However, finding an appropriate RPC solution respecting our demands seemed challenging in the first place. We started by evaluating a possible design solely based on stateful Web Services (JAX-WS) representing the remote technology and simple AJAX calls to communicate with service endpoints. Since NetLuke requires stateful sessions (an algorithm instance needs to be aware of its owner including its state) and manual AJAX calls require considerable, unnecessary and at the same time unwanted programming effort, the outlined approach was not practical at all. It did not meet the requirements in any way. We abandoned the idea because of extra overhead and unforeseeable difficulties in the development process. Soon we realized that we needed the assistance of a "field-tested" framework, which facilitates our requirements to the last detail. The ability to connect JavaScript with Java was in the center of considerations. As mentioned in the requirements (see Section 3.3.2), we needed technology that is very easy to use. Therefore, a solid API is mandatory in order to make server-side method calls as trivial as possible. One guiding principle was: "The less code the better". Despite of these "comfort features", the factor of functionality, facilitating our low level non-functional requirements, was of even greater importance. Algorithm logic and its visualization components on the client need to maintain a specific state, which can not differ from each other (synchronous calls). A visualization needs to be precise. Therefore the UI needs to wait for the servers response to previous method calls, before changing its state.

We also wanted to provide functionality to send messages, initiated by server events (server errors, algorithm malfunctions, session information) to the web client. As pointed out before we wanted to give users the opportunity to react to those events. Therefore certain server components need to be able to invoke cient-side JS methods. In traditional "pre" HTML5 concepts this is not directly possible, as communication is stateless by principle. The design of HTTP/HTML does not encompass messages of this sort. Typically a client loads data from a server into the browser together with JS code. The result is often a static page e.g. "Web 1.0". This process does not envisage messages sent by the server without loading it in the first place. The needed feature is commonly referred as server-side or reverse AJAX, as it inverts the roles of client and server in the HTTP communication process.

[2]http://nodejs.org/
[3]http://www.webrtc.org/
[4]http://websocket.org/
[5]https://developers.google.com/web-toolkit/

We found a solution in *Directwebremoting*[6] or DWR "Easy Ajax for JAVA". It allowed us to meet our system- and developer requirements regarding client server RPC style communication aspects. In the following Sections, we explain what DWR is, how it works and why we use it to facilitate NetLuke's distributed architecture.

5.2.2 What is Direct Web Remoting?

DWR is a well documented, supported and actively maintained open source Java framework " [...] that enables Java on the server and JavaScript in a browser to interact and call each other as simply as possible" [22]. DWR requires a J2EE application server with Servlet API 2.2 or later, but does not need any web browser plugins or extensions on the client-side [7] for full functionality. DWR handles all of its processes in the background, keeping all communication details transparent to users and developers alike.

> "[...] DWR, which stands for Direct Web Remoting, is a combination Java/-JavaScript open source library that allows you to simply and easily build Ajax applications without getting into all the details of XMLHttpRequest coding. It allows you to call server-side code in a way that looks like it is running locally in the browser from the point of view of the client-side JavaScript that makes the calls. [46, p. 45]

The *XMLHttpRequest* (XHR) JS object, inherent to all AJAX based web applications and essential to NetLuke, is in the center of DWR. XHR requests and their responses are handled by provisioning flexible callback functions. The provided API is consistent and remarkably easy to use. What makes DWR so "powerful" is the ability to dynamically generate JavaScript functions out of Java classes based on its XML configuration file `dwr.xml`. The following quote from the project page [22] unfolds the meaning of this special feature.

> "[...] DWR will generate the JavaScript to allow web browsers to securely call into Java code almost as if it was running locally. It can marshal virtually any data including collections, POJOs, XML and binary data like images and PDF files. All that is required is a security policy that defines what is allowed."

The comprehensive featureset and its flexibility is just another reason why we chose this framework over other comparable solutions[8] as parametrized message passing to remote endpoints is indeed very easy to realize. Just a few lines of code are necessary to get a response from the server. Therefore, a developer can focus on core tasks such as programming visualizations and control structures. He is not bothered by XHR implementation details. We, the authors of NetLuke, on the other hand, can use DWR to focus on interface design while DWR handles all necessary processes involving marshaling of data, session based remote object creation and of course communication tasks. DWR does not

[6] http://directwebremoting.org/
[7] DWR supports Firefox 1.X+, IE6+, Opera 7.4.2+ and Safari 1.2+ [22]
[8] Google Web Toolkit (Vaadin), Apache Wicket, DOJO, etc.

impose any architectural changes to existing server-side code by concept. An application is "not built around DWR" in the sense that it creates dependencies to existing business logic with the (light) exception of reverse AJAX messaging.

5.2.3 How does DWR work?

DWR uses a standard servlet to create a single endpoint based for a configured java class. It exports its behavior to JS. Without DWR the AJAX/JAVA approach would require many different endpoints created by multiple servlets or just one complex single one. Development of suchlike would require considerable time and effort.

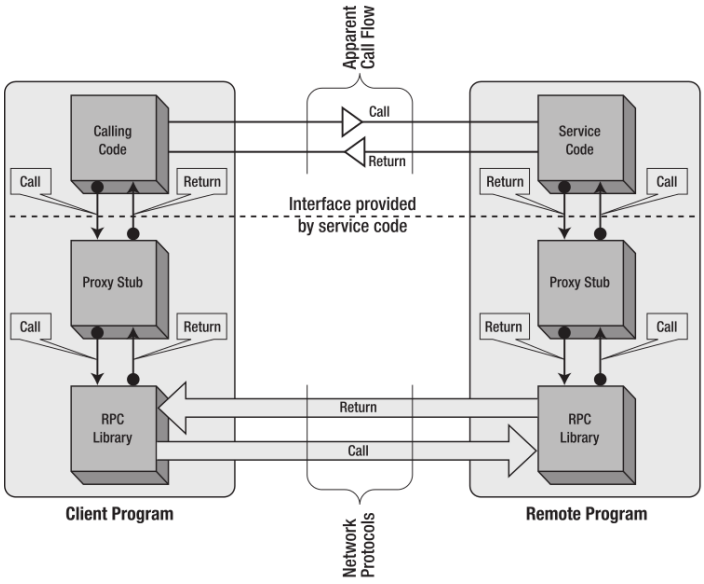

Figure 5.2: DWR/RPC Overview

In NetLuke the *NetLukeDWR* service class is configured as the endpoint for modules. The *MessageController* class is used for reverse AJAX messaging to clients. To make use of these classes, the client needs to include three "files" (see Listing 5.1) in order to allow JavaScript logic access remote functions. Only the first one is an actual included file [22]. In our case the `remote.js` and `Messages.js` "files" are generated JavaScript representations of the *NetLukeDWR* and *Messages* java classes dynamically created by DWR's servlet. The `engine.js` is the cient-side part of DWR [46] necessary for marshaling calls, data transmission and exception handling. Furthermore it exports an configurable *engine* Object, which can be used to tweak timeouts, error handling, filtering or other options related to DWR's behavior either globally or per-request. In NetLuke the *engine* Object is used for global error handling and data transmission indication i.e. a graphical loading indicator.

LISTING 5.1: DWR Client Stubs

```
<script ... src='/NetLuke/dwr/engine.js'></script>
<script ... src='/NetLuke/dwr/interface/remote.js'></script>
<script ... src='/NetLuke/dwr/interface/Messages.js'></script>
```

DWR is basically a Java RPC library for JavaScript because it behaves and acts according to the RPC standard model and involves components like stubs and proxies. In Section 4.3.1 we mentioned a "cient-side RPC proxy". We will now explain shortly the meaning of this. Figure 5.2 [46, p. 46] illustrates the standard RPC communication model, which can be easily mapped to the DWR model. Based on a short example derived from the Prim algorithm module, we will demonstrate RPC component mapping. For this purpose we revisit the JavaScript code from Listing 4.8 and the according server-side remote method in

Listing 4.2, both discussed in the previous Chapter. We explain model mapping directly through code comments within the Listings 5.2 to 5.4. We demonstrate which code fragments/components map the ones in the RPC model in Figure 5.2.

LISTING 5.2: Example: Remote Procedure Call

```
/*
 * The body of getGraph() represents the calling code used in
 * the Prim.js visualization implementation.
 */
getGraph : function() {
        /*
         * The remote object maps to the client-side proxy stub
         */
        remote.invokeMethod('getAsXML', {
                async: false,
                callback:function(str) {
                        //do something with str (server response)
                });
        });
}
```

LISTING 5.3: Example: XML message

```
/*
 * invokeMethod(name) represents the server-side proxy stub.
 * In NetLuke, the name parameter, identifies the actual
 * service code.
 */
public String invokeMethod(String name) {
        /*
         * The nc Object is actually a NetLukeController instance
         * instantiated by DWR servlet upon first request
         */
        return nc.method(name);
}
```

LISTING 5.4: Example: XML message creation in getArrayAsXML()

```
/*
 * getArrayAsXML() acts as the service code on the server-side.
 */
public String getArrayAsXML() {
        StringBuffer xml = new StringBuffer();
        // create an array representation and save it to xml
        ...
        /*
         * The returned String is available in the client-side callback function
         * as an argument.
         */
        return xml.toString();
}
```

5.2.4 DWR in NetLuke

Together with KineticJS, DWR can be considered as one of the "project enabling" frameworks/technologies. What made the decision for DWR relatively easy was the absolute

ease of Java integration, the high level of configurability, as well as security- and session features. Availability of these features is absolutely crucial to the project concerning defined requirements in Section 3.3.2. In the current implementation of NetLuke, DWR fulfills three essential purposes. First: NetLuke uses DWR extensively for RPC calls. Second: It is used to send messages e.g. logout notifications from the server to the client. Third: DWR handles the instantiation of *NetLukeController* objects (see Section 5.3), managing module instance execution. DWR allows the creation of objects based on a session. This ability is extremely important as a modules state needs to be maintained active throughout a users session. DWR keeps references to all created objects as long as the user sessions lasts. The Garbage Collector (GC) releases memory after a user session ends.

Currently (February 2013) Version 2.0.10 is the stable release. We decided to use Version 3 even though it is in Beta state, because, according to DWR's maintainer Joe Walker[9], its reverse AJAX logic is more robust and DWR3 supports Java "varargs". Dynamic creation of JavaScript objects out of Java classes relies on subtle manual configuration tasks. In NetLuke currently two classes export their functionality for remote usage. One is the *NetLukeDWR* service as well as the *Message* class for reverse AJAX functionality. The former represents the client JavaScript *remote* object used for RPC message passing to algorithm plugins. DWR's main configuration file is illustrated in Listing 5.5. Lines 6 – 9 induce DWR to allow remoting of the *NetLukeDWR* class. The creator attribute in Line 6 implies, that a new instance of this class is created upon request. The scope attribute tells DWR to uphold on the instance as long as the user session is valid. The javascript attribute renames the original endpoint to "remote" on the cient-side. The *MessageController* class is object to reverse AJAX messages and is configured like the *NetLukeDWR* class. The application scope means, that this class is available upon DWR servlet initialization until the DWR servlet is destroyed (one object for all users). The converter tags[10] are needed for data marshaling.

LISTING 5.5: dwr.xml Configuration File

```
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     <create creator="new" scope="application" javascript="Messages">
12       <param name="class" value="at.ac.univie.netluke.controller.
13               MessageController"/>
14       <exclude method="update"></exclude>
15     </create>
16
17     <convert converter="bean" match="at.ac.univie.netluke.controller.
18               MessageController"/>
19
```

[9] http://directwebremoting.org/dwr/downloads/changelog/dwr30.html

[10] More information regarding DWR server configuration can be found in the official documentation available under http://directwebremoting.org/dwr/documentation/server/index.html

```
20        <convert converter="bean" match="java.util.Observable"/>
21        <convert converter="bean" match="java.lang.Object"/>
22      </allow>
23    </dwr>
```

5.2.5 Web Service

Aside from browser based connectivity, NetLuke allows to export its functionality over Web Service endpoints. A client using the service may be a Java client application or a .NET based application. We decided to support non web browser (platform specific) application types to allow more powerful native and hardware accelerated AV systems using NetLuke as the back-end. Suchlike AV systems may use frameworks like Java FX2[11] or use native visualization routines in .NET, QT, Android, OSX, etc. Of course suchlike applications brake the module approach as JavaScript visualizations need to be ported to a new underlying visualization system/library. Nonetheless we wanted to leave the door open for non web browser based clients.

We discussed the fact, that NetLuke requires a stateful mode of operation through out all of its services by concept. This is in particular cumbersome to Web Services – stateless by principle. We had several ideas combining a stateless Web Service facilitated by Apache Axis2[12] and the Spring Framework[13] with HTTP sessions, but it would have added an unwanted level of complexity to the project. Extensive configuration and programming tasks should be avoided. A RESTful approach would not make sense as well, because stateful operation does brake the REST principle[14]. A stateful REST service would require passing an authentication header along every request a user makes. This solution is certainly not "elegant" and therefore maculation. Therefore we decided to use the latest Java API for XML Web Services (JAX-WS 2.1.7 in 2012)[15] Reference Implementation (RI) to implement a "native" stateful service. It processes HTTP/SOAP requests over TCP protocol.

In NetLuke a JAX-WS based service implementation makes sense, because the creation efforts for stateful services are low compared to other solutions (annotations model, deployment descriptors are optional). It does not require a fully blown framework and is still easily configured by the `sun-jaxws.xml` file. Upon system startup, the system automatically creates a wsdl file under the custom URI http://<server-ip>:<configured-port>/NetLuke/ws?wsdl. By adding the listener class: *WSServletContextListener* to `web.xml`, clients can use the endpoint. Unfortunately utilizing JAX-WS imposes one problem, which renders our solutions sub-optimal: Clients need to support a "session maintain property"[16] in order to support stateful operation. In the optimal case, the client uses the JAX-WS libraries which support this mode by default. Developing a non Java based application that maintains sessions might be tricky. However, at the time of writing this thesis the Web Service layer is not yet finished. The current implementation

[11] http://www.oracle.com/technetwork/java/javafx/overview/index.html
[12] http://axis.apache.org/axis/
[13] http://http://www.springsource.org/spring-framework
[14] Representational State Transfer (REST), relies on HTTP response codes. Interaction is stateless between requests by principle
[15] http://jax-ws.java.net/
[16] http://docs.oracle.com/javaee/5/api/javax/xml/ws/BindingProvider.html

serves as a proof of concept ready to be extended for full service operation. Creating web service client prototypes visualizing data takes time and did not have priority in the development process.

5.3 NetLuke Controller

So far we covered how the service layer provides generic interfaces for modules. Now we explain how modules are managed and controlled. The component responsible is *NetLuke-Controller* (further referred as NC), which can be described as the "Heart of NetLuke". It connects modules with the service layer and handles requests to the persistence layer. In Figure 5.1 colored green, we can see that the controller interacts with the services and the module components directly. It processes requests to modules, controls algorithm instances and features extensive debugging capabilities. Since modules are deployed as extensions (libraries), the system is not aware of each method or constructor a specific (configured) class provides e.g. the system does not know anything about module implementation details. Therefore, a component is needed, that allows the system to discover classes and their inherent methods and constructors during runtime. NC performs essential tasks during system startup and runtime such as:

1. Making module classes available to clients (dynamic metadata discovery).
2. Indexing of module constructors, methods and attributes through Java Reflection for fast client access.
3. Mapping of named client method requests e.g. `invokeXMLMethod()` to method calls (dynamic calling).
4. Switching between active and inactive instances and global handling of module instances.
5. Providing debug information for module developers to minimize error probability in class and method discovery.
6. Handling of object (de-)serialization e.g. automatic loading and saving of instances.

On the basis of a scenario we will explain the important role of the controller within the system [17] – The process of creating an algorithm visualization initiated by an actual user. Our starting point is the creation of an algorithm instance on the client.

1. Instantiation

When a user chooses a topic on the UI, the underlying JavaScript routines "activate" the according module on the server-side. This is done by calling `setActiveAlgorithm(name)`, where as the name parameter matches the identifier given in `netluke.xml` as discussed in Sections 4.3.1 and 4.4. A user calls this method implicitly, not knowing that DWR creates a new instance of NC first by passing the username of the sessions owner to

[17] Detailed description of the NetLukeController class is provided via JavaDoc directly in the Class body.

its constructor, before `setActiveAlgorithm()` is called and executed. Hence, the first remote procedure call creates an instance of NC. The NC life cycle begins with a database read operation, checking if any previous instances of any module owned by the user (see Section 5.4 for further discussion) exists. If instances were found, they are reused and saved in a new Java hash map exclusive to NC. However, these instances are shared through a globally known data structure. If the user is new (has never created an instance before) the hash map is created but stays empty. We decided to use Java hash maps and concurrent hash maps for fast average $\mathcal{O}(1)$/thread safe access. At this point all previously created instances, containing all data objects and attributes are available to the user. During its initialization phase, NC creates several static and non-static data structures (mostly hash maps) containing information about deployed algorithm modules. The *NetLukeXMLProvider* (further discussed in Section 5.8) provides an initial set of algorithm names (identifiers) mapped to classes. Based on this information NC creates hash maps containing all constructors and methods found in these classes. This is done by facilitating the capabilities of Java Reflection as NC does not know about constructors and methods within modules. In that sense, NC facilitates flexibility. It allows to exchange modules and their implementation details without affecting the system or the service layer.

2. Method and Constructor Discovery

After NC object creation and initialization phase completes, the `setActiveAlgorithm()` method is executed. Its main purpose is to "declare", that one single module is active and is ready to be used. Active means that all necessary methods (classes) within a module are available in memory. This means that they are "prepared" for invocation through the Web Service or DWR RPC calls. Since method invocation through Java Reflection is commonly known a bit slower than traditional access [45, p.], we decided to use two hash maps containing methods in the first and constructors in the second one. We want to compensate slower execution by faster access times, hence storing methods before they are used to avoid searching overhead when an actual method is called. This way the overall RPC response time is kept to a minimum. The disadvantage is higher memory consumption. In general, NetLuke is memory intensive because of NC's intensive "buffering" activity.

3. Method and Constructor Access

Once a module – hence its implementing class, its constructors and methods are known to the controller, clients can use it. In Section 4.3.1 we educed how module methods and constructors get invoked by named requests. We explained the need for an additional abstraction layer between modules and the system. Now we unfold how constructors instantiate a module and how methods are called. We explain those vital mechanisms by means of two examples:

1. `remote.invokeIntegerConstructor('BubbleSort',{0,44,1,23,-7});`

2. `remote.invokeMethod('next');`

Lets assume a user has just registered to the system and wants to create a new instance of the BubbleSort algorithm including data. The implementing client module provides a JavaScript function which calls the `invokeIntegerConstructor()` service method provided by the NetLukeDWR service component. As soon as the RPC message is transmitted, it checks if the call is valid (contains integer values) and if the parameter is an array or not. Since the service layer does not process requests, it dispatches the call to NC by calling the `{Constructor(String name, String type, Object ... args))` method. On the basis of the provided arguments, this method checks if a constructor with the name BubbleSort exists that takes an integer array as an argument. If this check evaluates true, the constructor is called with `newInstance(args)` where as the returned object of type BubbleSort becomes the currently active instance object (instanceobj). If anything goes wrong in the process e.g. the check evaluates to false, enabling a higher debugging level in the configuration will output messages providing information which helps to track down the source of an error. In any case the client receives a response code informing about the error e.g. code 200 for successful instantiation and 404 for an invalid constructor call. The DWR service passes this code on to the client for subsequent processing. Calling a method is very similar to calling a constructor, with the exception that the invoked method returns a String and not necessarily a simple response code. By calling `invokeMethod('next')` on the client, a method with name 'next' of type void is searched in the previously indexed methods which are implemented by BubbleSort. The return value of `method.invoke(instanceobj)` is module specific (XML response message) and is directly returned to the client. Subsequent calls of parametrized module methods work the same way.

By using Java Reflection the number of methods exported to the service layer can be kept low. By "passing through" client requests to the according methods the system is completely independent from module implementations. Another advantage of this approach is robustness against errors within modules and erroneous client calls. Although usage of Java Reflection means loss of type security, NC does not only check for the name of methods but also their type signature, which reduces faulty invocations. This check is also needed to allow calls to method which have the same name, but different type signatures.

5.4 Module Instance Handling

The term "handling" comprises several distinct operations associated with module instances such as manual loading/saving, automatic serialization, or switching between running instances. While manual aspects, such as a user invoked events, are usually primarily subject to NC, automatic tasks are carried out by components which react to events unknown to NC. They are discussed later in Section 5.6 and 5.7. However, NetLuke allows clients to re-use instances independent from the session they were created in. User action for loading or storing instances upon login is not necessary as any tasks involved are carried out automatically, ensuring a high level of user experience in working with NetLuke. For example, when a user logs out of the system and logs back in, the visualization (except the positions of elements) is looking exactly like when he logged out, as the instance from the previous action is restored. With this kind of system behavior, even at unexpected events, we wanted to accommodate the requirement of unstable connections or otherwise failing clients. Automatic loading of previously created instances should help users to continue with learning tasks right were they stopped.

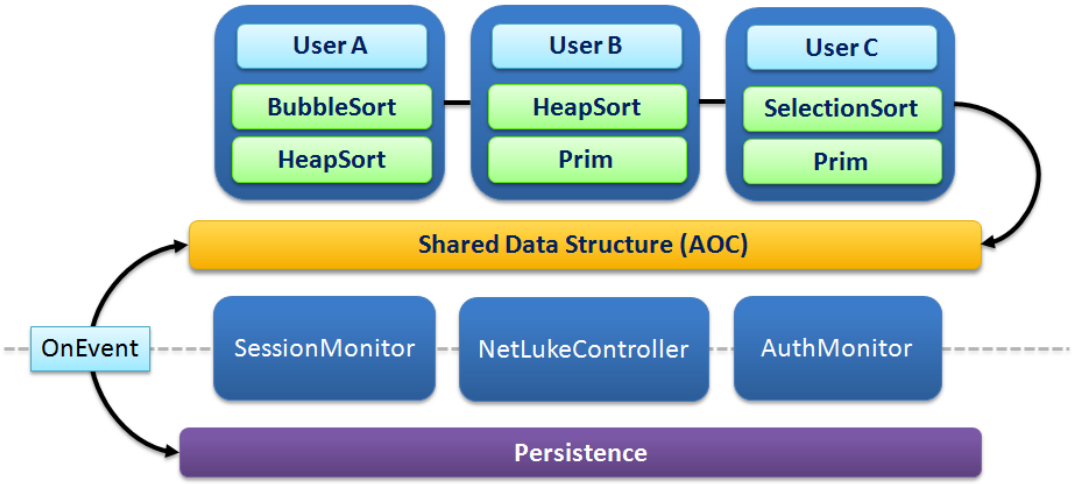

Figure 5.3: Shared Module Instances

NetLuke is capable of restoring instances from invalidated sessions. In the previous Section we heard that a NC object is exclusive to a user (one controller instance for one user). This implies that any associated instance may not be used by other system components due to missing references to NC. To overcome this problem, the system accesses a single object (Singleton), containing all module instances, which is shared among components. This object is realized by the *AlgorithmObjectContainer* (AOC) class. The object is first instantiated upon system startup. Upon first use of DWR, NC prompts the persistence layer for serialized instances to load them into AOC by calling `setInstances()`. Therefore AOC contains all references to the latest created instances independent from sessions. This way clients can access their instances fast, as they are stored in a Java *ConcurrentHashMap* (as different threads can access the data structure at the same time). An advantage of this concept is direct data access to all existing module instances, without having slow subsequent database reads. The drawback of this solution is high memory usage, as instances may contain multiple snapshots of itself due to the implementation of the history mode feature.

AOCs genuine purpose is to provide system wide information about a users set of module instances. Figure 5.3 illustrates the layered structure of components accessing AOC. A shared pool of data comes in very handy when sessions are not properly invalidated, connections break or simply when fast switching between instances is required (represents an event in Figure 5.3). The *ConcurrentHashMap* wrapped by AOC is shared among components such as NC, SessionMonitor and AuthMonitor and is indirectly referenced by persistence components.

5.5 User Management

NetLuke is designed as a multi-user platform, independent from the deployment method (standalone or application server). However, if deployed on an application server and connected to a client server DBMS, the tool can be accessed by many users concurrently. Even the standalone version is not limited to a single user. The multi-user approach distinguishes NetLuke from many other AV solutions in terms of accessibility, area of

operation and flexibility. Given the fact that NetLuke is currently in a "alpha to beta stage", user management is reduced to essential functionality, with future improvements in mind. We support registration of users and (not yet fully implemented) routines for user deletion and roles to associate permissions with users.

In the current prototype the *UserController* (UC) handles the registration process initiated by requests coming from `signup.jsp` (signup) or a Web Service client respectively. UC checks if a supplied username exists in the database, verifies that the provided password pair matches and if the supplied email address[18] is valid. UC's register method ensures, that usernames are unique, which is crucial, as usernames are used as keys for data structures (hash maps) in certain components (NC, AOC). Before a user is "created" it is checked if the password consists of at least four characters, or if any other error occurred during the process. Although user credentials are checked via JavaScript before submitting, we also need routines on the server, because UC does not know the source of the registration request. The result of the registration (successful/failed) is returned to the client in the form of appropriate HTTP response codes. After successful registration a user is able to login to the system and is authorized to use modules. Not accessible from the web UI at the moment are built in routines for user deletion along with all stored instance data existing in the persistence layer. UI control for this functionality (low priority), along with an "administrator interface" is scheduled for a future release.

User Identification

Although "oversized" and not indispensable for the current prototypal implementation, we decided on using immutable universally unique 128bit long identifiers (UUID) to "mark" users with an unique id. Currently we use UUIDs to identify users before attempting database operations. UUIDs exhibit certain advantages for instance over simple incremented counters [45, p. 761]. On the one hand we can rely on a more future proof design, for instance if NetLuke is integrated in a more complex environment (online E-Learning tool) with more sophisticated single sign on capabilities and database back ends. On the other hand UUIDs allow us to generate unique IDs within our application context while being independent from a specific data base technology. However, we do not work with UUIDs in Java logic for identification purposes of objects or sessions.

User Accounts

User creation involves storing credentials in the database back-end. Passwords should be stored in a non human readable encrypted format and never saved in clear text. This requirement is very important to us. With advanced deployment environments in mind, we decided to opt for a more sophisticated security solution. The process of user creation is as follows: If all checks in UC, concerning supplied registration data, evaluate to valid, the user password is encoded by an *Encrypter* class and stored in the DB as a password hash – salt[19] combination. This is a very secure way of storing credentials, because brute-

[18] E-Mail addresses are not relevant for any component at the moment, but may be needed for future purposes.

[19] A "salt" is added because users may choose the same password which results in an identical hash. Adding a random salt to the hashed password prevents duplicate hashes. For more information regarding hashing in Java visit: https://www.owasp.org/index.php/Hashing_Java

force attackers do not only have to guess the initial password, but also the random salt. The one-way hash (digest) is created by the provided password (String), a random salt based on `java.security.SecureRandom()` and a total of **1024** hash iterations. We utilize an SHA-256 based hashing algorithm, provided by the Apache Shiro security framework discussed in the next Section 5.6. It produces an encoded representation of the password. Together with salt (needed for the authentication process), the hashed password is stored in the DB. These measures were taken to actively secure NetLuke and reflect our concern for security on a low level basis.

5.6 Sessions & Security

Session-, authentication- and security management is very important to NetLuke. Handling of events related to sessions in particular, as we use information associated with sessions to track activity of users. We raised this subject in Section 5.4. Now we want to discuss this topic, together with security aspects in detail. For many tasks NetLuke heavily relies on the abilities of a security framework we introduce throughout the next paragraphs. In a previous Section, we mentioned, that the state of created objects (module instances) is closely tied to (session) events occurring during runtime. Events can occur ...

(a) ... unexpectedly – For instance when a mobile user looses connectivity to the server.

(b) ... on purpose – when a user closes the browser window without logging out first.

(c) ... within normal operation – when a user logs in or out.

All of these events are associated with component interaction (see Section 5.7). In order to provide an integrated component which facilitates authentication, authorization, session management and cryptographic functionality we decided to use a 3rd party security framework. We decided against integrated application server security solutions, like Java Authentication and Authorization Service (JAAS) [20], because JAAS is rather inflexible, complex (concepts, classes and API) and rather difficult to customize. Custom security solution development was definitely not up for discussion, as we are not experts in securing web applications.

5.6.1 Apache Shiro

We utilize the comprehensive Java security framework Apache Shiro[21] further referred as just "Shiro". Shiro is used for authentication, session management and may be used for future fine grained authorization aspects (roles). Shiro is very flexible in terms of application types (web application, Web Service, or standalone), configuration and deployment[22]. Shiro allowed us to separate security concerns and business logic with very little coding efforts. Shiro offers "Container Managed Security" configurable on a central level within

[20] http://docs.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html
[21] http://shiro.apache.org/
[22] For a detailed description of Apache Shiro visit: http://www.infoq.com/articles/apache-shiro

the configuration file (`shiro.ini`). It can also be configured declarative through Java properties. Shiro allows developers to adjust security constraints on very fine levels (filtering system), which is helpful for precise security directives. Securing a web application properly, is not a trivial task, especially not if there is a gradation in security needed, as some URIs need to be accessible by anonymous users or when session creation needs to be suppressed. For instance, we needed to exclude a servlets URI for mobile user authentication from session creation. This is in respect to Android application design. The registration page including the CSS style sheets should not require authentication. What made Apache Shiro the ideal solution for NetLuke, is its flexibility through customizable filter chaining, easy configuration and integration[23], but foremost its lightweight character where logic is "placed" right were it is needed. Moreover, Shiro does not interfere with architectural design, as functionality is located on deeper levels.

5.6.2 Subjects

One essential concept of Shiro are "Subjects". A reference to this object is available in every web component (JSP pages, service layer, monitors, etc.), within the context of Shiro. It describes a specific user e.g. the currently executing user, including all related data such as credentials or session data. "[. . .] It is Shiro's primary mechanism for single-user security functionality"[24] Having a this kind of information at our disposal saved many lines of code otherwise necessary. For instance: The DWR service holds a reference to its owner. This owner is determined by a subject inherent to the class, as it lays within context of Shiro. Yet another fundamental advantage of Shiro's concept is support for heterogeneous clients. For instance: a Subject is not necessarily determined by an HTTP Session object instead by a "Shiro Session", which is independent from a specific protocol.

5.6.3 Realms

Shiro supports authentication realms, specific Data Access Objects (DAOs) to communicate with a Java Data-Source (DS), in our case the configured database back-end (see Section 5.9). A major advantage of (JDBC) realms over custom data store access is the added layer of abstraction on top of the DS. Realms provide a standard interface to Data-Sources needed for customizable container-managed security. We implemented a custom JDBCRealm – *NetLukeRealm* – leveraging form-based authentication/digest authentication to NetLuke, discussed in the following Section. This component allows us to offload authentication logic associated with persistence queries to Shiro with very few configuration options in `shiro.ini`. Our *NetLukeRealm* class allows Shiro to manage authentication processes completely transparent to the rest of the system.

5.6.4 Authentication

In Section 5.5 we discussed user management and the registration process involving encryption of passwords and storing of user related data. The authentication process relies on Shiro‘s ability to process data provided by *NetLukeRealm*. Authentication of users does not require any programming efforts, as logons are handled transparently (container

[23]NetLuke utilizes the following Shiro libraries: *shiro-ehcache.jar*, *shiro-core.jar*, *shiro-web.jar*
[24]https://shiro.apache.org/static/1.2.1/apidocs/org/apache/shiro/subject/Subject.html

managed). This means that components accessible by clients e.g. JSP pages are completely independent from the definition of security constraints. A successful logon entitles users to access core client components (discussed in Section 6.1). Basically the authentication process is as follows. The `shiro.ini` configuration file tells Shiro to use `login.jsp` as the single point of authentication (form based), whereas a successful login event redirects users to the `index.jsp` (main page) and a failed one returns back to `login.jsp`. *NetLukeRealm* fetches the password hash and the additional salt in the background and hands it over to a credentials matcher, offered by Shiro. This component calculates the corresponding hash+salt combination (digest) and compares it to the credentials provided by the user. If the credentials match with the information in the data storage, a user is logged in and is further available for referencing by its "Subject". Associated events are discussed in Section 5.7.

5.6.5 Session Management

Shiro simplifies session management as session are specific to the currently executing Subject, not to a specific HTTP request. This allows us to retain session handling completely to Shiro making it easy to keep track of objects and events relevant to users. Particularly interesting for NetLuke is the ability to determine if a session with associated instances is still in use. Ideally these instances "stay alive" as long as a session is in use. Sessions are automatically kept alive even though a user is inactive because of the background reverse AJAX messaging done by DWR. However, in case there is no activity at all e.g. the client lost connection, the session is invalidated and automatically destroyed after 15 minutes. Sessions are checked in a five minute interval if they are valid e.g. still in use.

5.7 Monitoring

Monitoring behavior of users and components during runtime can be very helpful in detecting events which require action by the system. We discussed this matter in Section 5.4. Furthermore, monitoring components help at debugging tasks. Apache Shiro allows to implement custom monitoring[25] components for authentication, authorization and session events, while standard J2EE listeners provided by the servlet container, are used for initialization of components during startup.

5.7.1 Startup

In the current prototypal implementation we set logging (log4j) options on a declarative – programmatic level, before any other component is able to print debugging messages. This is done in the *StartupMonitor* class. It initializes the *NetLukeXMLProvider* class to read the logging options defined in the XML configuration file (see Section 5.8). As soon as the configuration is available to the system, we set the according options. Invocation of the *NetLukeXMLProvider* class also induces module class indexing, required by *NetLukeController*. The "startup monitor" is configured in the `web.xml` file as a listener class.

[25] We use the term "monitor" instead of "listener" to underline the involved reaction

5.7.2 Authentication

We capture successful authentication events as well as logout requests by the *AuthMonitor* class. It is attached to Shiro via the `shiro.ini` configuration file. If a user logs into the system, he may have an open session with associated module instances on a different client machine. This situation occurs if a user forgets to logout on the desktop, leaves the browser window open and switches to the mobile application on his phone. The *AuthMonitor* invalidates the previous desktop session and maps the old components created during this session to the new one (phone). The *MessageController* class is an observer of the *AuthMonitor* and sends a message to the desktop client informing about its invalidated session, by an UI blocking overlay. This way NetLuke allows seamless switching between instances. On the other hand, if a user logs out his session is invalidated, the events are captured by the monitor and induces the AOC to persist instances.

5.7.3 Sessions

Session related event monitoring is very similar to monitoring tasks in the above Sections. However, NetLuke needs to react to expired or invalidated sessions detected by Shiro's security manager. For this purpose we created a *SessionMonitor* class, which not only provides logging/debugging information, but handles the immediate persistence of instances a user has created during this session. Again, this event is handled via AOCs `persist()` method similar to the *AuthMonitor*. If a session expires, the executing client is notified via the observing *MessageController*, as explained in Section 5.2.4.

5.8 Configuration & Logging

The project is configurable via multiple files each (see Table 5.1). In Section 4.4 we illustrated `netluke.xml`. This file is the main configuration file specific to the project. Web application options[26] are subject to `web.xml`. Now we will explain how the information within `netluke.xml` is used in our system. Information encoded in the config file needs to be available on system startup, before any component can access the persistence layer, even before the logging subsystem is able to print/write messages as explained in the previous Section. We implemented the *NetLukeXMLProvider* (NXP) class to transfer XML markup into system parameters. Only one instance is required (Singleton). During its initialization phase, *NetLukeXMLProvider* reads `netluke.xml`, parses its values and attributes. If errors occur system startup is suspended. Its purpose during runtime is to provide data (classes, description, identifier) about deployed algorithm modules. This information is vital to NC. Furthermore it holds user defined logging options and sets up the persistence layer i.e. the global Data-Source (DS) discussed in Section 5.9. NXP acts as an "option/settings" distributor class among components.

The following Table 5.1 introduces configuration files and their purpose. Usually a module developer does not get in contact with any of the files except `netluke.xml`. All other files however, are interesting for system administrators as Data-Sources are setup in `context.xml`, whereas `dwr.xml` specifies DWR options. Deployment descriptor files

[26] We decided to skip in depth explanation of configuration files other then netluke.xml as they can be found in the according online documentation.

(`web.xml` and `jax-ws.xml`) are used to map URLs to servlets/JSP pages (see Section 6.3) and describe the Web Service endpoint. As pointed out before, Apache Shiro is configured through `shiro.ini`[27].

File	Purpose
netluke.xml	- Definition of general module categories (topics). - Definition of actual modules (identification, implementing class) as explained in Section 4.4 and Listing 4.11. - Definition of the system-wide log level (INFO, DEBUG, TRACE) and targets (stdout, file).
context.xml	- Contains pre-defined JNDI/JDBC data source directives for HSQL (server, memory, file) and MySQL data sources. View Section 5.9 for more information.
dwr.xml	- Contains instructions/options for class remoting, their scope and security constraints (illustrated in Listing 5.5).
web.xml	- Definition of custom servlets e.g. mobilemm – for mobile authentication requests. - Definition of 3rd party servlets (DWR, Apache Shiro). - Definition of filters (DWRSessionFilter, Shiro). - Definition of listeners or "Monitors" e.g. *StartUpMonitor*, Shiro, JAX-WS - Web Service endpoint configuration - URI mapping for JSP pages e.g. index.jsp to home - Resource References for Data Sources in context.xml. - Definition of DWR servlet security constraints.
sun-jaxws.xml	- Defines the implementing Web Service class and its URI (endpoint).
shiro.ini	- Sets up Apache Shiro including filter chains, realms and listener classes (monitors)

Table 5.1: Configuration Files

Logging

NetLuke "borrows" its logging implementation from the Apache Commons Logging[28] library. Since we distinguish between three log-levels: (0) INFO, (1) DEBUG and (2)

[27] http://shiro.apache.org/configuration.html
[28] http://commons.apache.org/proper/commons-logging/

TRACE, we use checks like `log.isDebugEnabled()` whether to print a debugging message or not. "Critical components" like NC, AOC, or NXP are subject to extensive logging. For instance, a developer is notified about the available methods and constructors NC has found or whether if a method was called with the correct number/type of parameter or not. We also print SQL queries in debugging mode to track down errors related to the persistence layer.

NetLuke supports printing to standard-out (stdout) or writing to a file (`netluke.log`) located in the log folder within an application servers root directory. Module developers are recommended to set the level to (1) DEBUG rather then (2) TRACE. The (0) INFO level is intended for normal operation mode and is enabled by default in the standalone version.

5.9 Persistence Layer

NetLuke is deployable as a web application (WAR) inside an J2EE application server but also as a standalone application using an embedded version of Tomcat 7 (see Section 1.4). This twofold deployment strategy has implications on the underlying database back-end. At the time of writing this thesis, we can not estimate if NetLuke is most likely to be hosted on a private desktop system, or if it is integrated in an hosted environment. Dedicated servers will probably use a client server database system (DBMS) such as MySQL or Microsoft SQL Server. In the former case, a user wants to start the application right away, without any configuration hassle. This means that an installation and setup of a client server DBMS system is unnecessary. NetLuke should automatically create and use an appropriate database back-end. In the latter case, NetLuke should be able to use more advanced DBMS. Therefore, the persistence layer needs to be independent from the Data-Source. From a design perspective, the Java Naming and Directory Interface (JNDI) mechanism discovering Data-Sources facilitate our requirements concerning resource abstraction. For the actual data storage, we decided to use a file based relational DBMS in the default configuration, which supports SQL standard 2003.

In the planning phase SQLite[29] seemed to be the obvious choice. SQLite is known for its simplicity and performance. However it requires a binary installation on the client machine hosting NetLukeStandalone. This fact rendered our requirement for an "out of the box" usage of NetLuke impossible. Research pointed us in the direction of HyperSQL 2.0 or "HSQLDB"[30]. This relational database system is implemented in Java. "[...] HyperSQL can provide database access within the user's application process, within an application server, or as a separate server process." [11]. This means that we can use a HyperSQL for file based JDBC compatible database access as well as for server mode through JNDI. HyperSQL does not require any specific configuration, but its library (*hsqldb.jar*) needs to be deployed within the application servers `lib/` folder. We support MySQL server connectivity as well with the appropriate library (`mysql-connector.jar`).

[29] https://www.sqlite.org/
[30] http://hsqldb.org/

5.9.1 JNDI Data Sources

We support more than one DB back-end system. For this purpose we prepared several pre defined data sources (resources) in the web applications context.xml as a template. System administrators should be able to switch to other back-ends than the default one. The mapping of resource names to actual JDBC URL paths is listed in Table 5.2. The according resource references are located in the `web.xml` file. Our default data source, for both projects – NetLukeStandalone and NetLuke, is named "hsql-temp". The name relates to a db file located in the temporary folder (`temp/`) of an application server e.g. Tomcat's home directory (CATALINA_HOME). The database folder itself, containing stored data, is `netlukedb/`. Upon system startup, the *DataSourceProvider* (DSP) class, independent from the defined DS, initializes the configured DS. If it is not available, it switches to a fall back db located in system memory (hsql-mem)[31]. It then checks if the DS exists and if a connection can be established. If the db does not contain any tables (upon first execution of NetLuke), the `init.sql` file creates the initial database structure containing all necessary tables. Otherwise the db is checked for consistency and is initialized accordingly. After this process is finished the database back-end is ready for use. Furthermore, the database is portable. New versions of NetLuke can use the existing database as long as the db schema (see Section 5.9.3) does not change. Instructions of how to port the db will be provided in the according release packages.

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.address}:${mysql.port}/${mysql.dbname}`

Table 5.2: Data Sources in context.xml

5.9.2 Queries

The actual Java logic comprising the persistence layer is rather straightforward, as read and write operations are conducted directly by using plain SQL queries. The main components in this regard are *DataSourceProvider* (DSP) and *DataSourceQuery* (DSQ). DSP establishes, pools and provides connection to the selected DS. DSQ contains the queries and helper methods. Before queries are executed, DSQ requests an connection to the DS from DSP. DB requests originating from a user are checked before execution. We identify and verify first if a user is authorized to conduct database operations. DSQ checks if a username has a corresponding UUID. This requires a preliminary data base read operation before subsequent data base operations can be executed.

In general we wanted to avoid data base operations during client activity. Response times would suffer dramatically if instances would be saved to the db after every state

[31] The system prints debugging messages, warning about data loss upon system shutdown in fallback mode

change or element position change. Therefore the majority of db operations occur upon authentication or session invalidation, such as persistence of all module instances stored in AOC 5.4.

5.9.3 Database Schema

NetLuke uses four tables as illustrated in Figure 5.4. (1) USERS: Contains all relevant user data; (2) SAVES: This table is used for storing module instances (serialized) together with related information upon users request; (3) USER_OBJECTS: Contains all instances (latest) a user has created over time and (4) USER_SETTINGS stores user specific UI settings. We use the built-in HSQLDB id type as the PRIMARY_KEY in all of the tables where as the UUID foreign key is used for cross-referencing between tables. It is created during user account creation.

(1) We do not persist session related information, instead we use Shiro's cookies for session tracking and previously discussed one-way hashes (digest) to store credentials. Currently unused are the "role" fields discussed in Section 5.5. We decided to add this field to indicate roles of users e.g. "simple" or administrator in future releases. However, we do not know if this gets implemented in future releases because of low priority. A role system, based on Apache Shiro would allow fine grained authorization to sensible functions. For instance: A possible future config page could provide access to global settings or administrative functions (delete or add user, management of saved instances, etc.).

(2) The SAVES table provides necessary fields for saving and loading instances upon user request. A user provides a save name, together with an optional description. The instance itself (snapshot) is serialized and stored within the object_value field. A save may contain a picture of the visualization together with a preview. Actual client visualizations (the state of the visual representation) are not persisted e.g. positions of nodes or other visual elements (db operations with RPC are expensive).

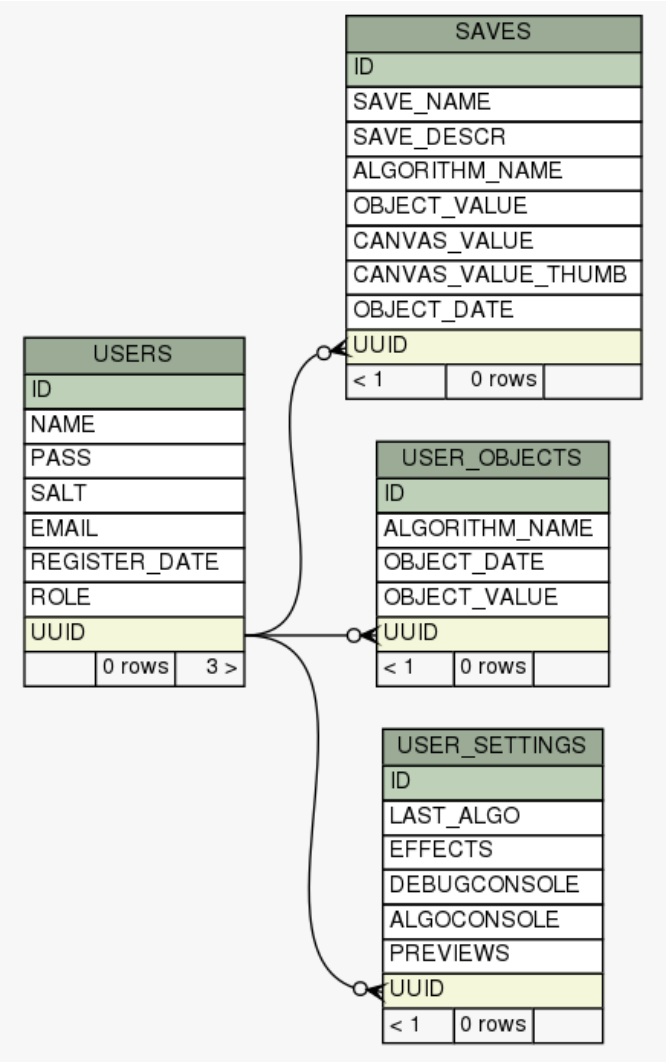

Figure 5.4: Database Schema

(3) Automatic persistence of instances as discussed in Section 5.7 and restoring of module instances 5.3 in NC, relies on the USER_OBJECTS table. The object_value field holds the last used instance of a certain topic. Therefore multiple instances are associated with one UUID.

(4) The settings table holds user specific settings related to UI and visualization behavior e.g. if advanced effects (module implementation specific) in visualizations are enabled, or if the debug console is activated. These settings are discussed in the following Chapter 6 in Section 6.2.1.

Chapter 6

Client Architecture

In this Chapter we briefly elaborate on client architecture, the UI and client-side components facilitating AV. We will explain which components interact to create a supportive environment for JavaScript AV module development. We also discuss how the featureset of NetLuke is usable through the UI. This includes AV control, features such as canvas panning and zooming, exporting of visualizations, loading or saving of instances and other interesting concepts introduced in previous Chapters.

In the first Section 6.1 we discuss client design concepts and components. We continue discussion with the underlying page structure (see Section 6.3) and move on to the graphical library in 6.4, which is central to visualization aspects. Client features are handled by a custom library, named loader.js explained in Section 6.5. We close this Chapter with an introduction to the Android web client application in Section 6.6.

6.1 Component Overview

In Chapter 3, we outlined several requirements regarding user experience on the one hand, and the demand for a developer friendly programming environment on the other hand. A major problem concerning these requirements is the fact, that client-side programming in JavaScript is not as precisely and unambiguously possible as it is in Java (see Section 3.5). Java is an Object Oriented Programming language (OOP) running in a JVM. JavaScript is a scripting language interpreted (mostly) by browser engines. Our architectural approach on module development works very well in OOP like Java. Browser environments however, require considerable programming efforts to realize a modular concept [8, p. 248].

Therefore, NetLuke should provide a client-side framework including pre defined functionality fulfilling functional and non-functional requirements. In Section 3.4 we argued that UI components should be provided by the system. Therefore NetLuke has built in views, overlays, menus, modal dialogs, text consoles and panels for control elements like buttons or links. These components need to be used by module developers, just like if creating Java modules with *NetLukeCore*. By following the concept of basic component provisioning, we create a "light framework" for algorithm module integration, helping developers to focus on core tasks such as algorithm visualization programming and control routines. We heavily rely on jQuery and its plugins for component realization and event

handling. This however creates a very high level of coupling to jQuery. This circumstance is always undesirable, but in a HTML/JavaScript environment difficult to avoid. Since jQuery is very widespread, well supported and documented it evolved to the quasi "standard JavaScript library" [8, p. 523], alleviating the problems of strong dependency.

Since we use many jQuery plugins among other custom JavaScript libraries, we provide a list in Table 6.1. All jQuery related files are located in the `WebContent\js` folder, whereas others reside in `WebContent\js\main`. **Important:** Version updates on the listed components are very critical, because of the high level of dependency among them. Exchanging libraries entails extensive testing efforts. Note: The listed jQuery plugins are not structurally altered, instead we modified look & feel with custom CSS style sheets.

Module/Component	**Purpose within NetLuke**
iscroll-min.1.js	Allows touch based scrolling of `<div>` elements on Android devices with version greater than 4.0 (used to fix an Android issue with Overflow:scroll/display:hidden CSS properties applied to `<div>`).
jquery-1.7.2.min.js	DOM manipulation, DOM element selection, event handling. jQuery is used in NetLuke to simplify client-side scripting dramatically.
jquery-impromptu.js	This library is used for UI blocking dialogs very useful when creating subsequent dialogs for data input (used for complex dialogs).
jquery.blockUI.js	General purpose blocking overlay creator, with direct CSS styling options (used for simple overlays).
jquery.jgrowl.js	Extensively utilized to deliver messages, warning and notifications to users e.g. informing about algorithm execution steps, or hints if errors occur.
jquery.prettyPhoto.js	This library is used for HTML canvas image export directly to the screen, displaying a preview and a fullscreen image.
jquery.select	Styling of HTML `<select>` element (Quickswitch)
jquery.ui	jQuery UI (reduced functionality) is currently used for drag & drop of console windows.
jquery.validate.js	Realtime validation of user input during the signup process on register.jsp (/signup)
view.js	Sets the appropriate viewport on mobile devices before CSS stylesheet import
loader.js	Custom library containing JavaScript UI components, containers and instance handling routines (Section 6.5)
kinetic-v4.4.1.min.js	Advanced HTML5 canvas framework used in all algorithm visualizations. (Section 6.4).

Table 6.1: JavaScript Modules & Components

6.2 User Interface

This Section introduces the User Interface of NetLuke predominantly on the basis of figures and according element descriptions. We introduce the desktop UI separated from the mobile UI to underline the differences in features and usage.

6.2.1 Desktop

Desktop users are presented with a fullscreen UI, consisting of a top menu, a large view containing the visualizations (further referred as stage) in the middle view and a bottom menu with "history mode" control buttons. An illustration of the UI presented to the user is depicted in Figure 6.3. Figure 6.1 shows the top menu. Its functions and control elements are explained in the paragraph below:

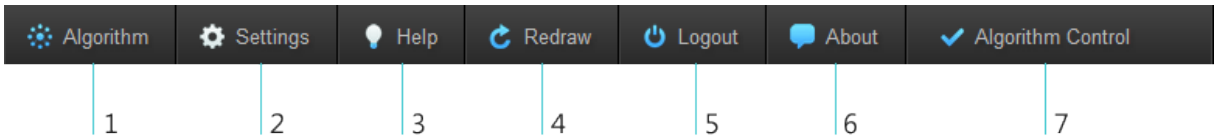

Figure 6.1: Desktop UI - Top Menu

1. The "Algorithm" button opens the algorithm menu including module specific descriptions. The menu offers a selection of deployed modules, allowing users to choose/start an algorithm instance of their choice. The current active instance is highlighted in green color. The description on the right view contains, additional information regarding the module. For example a general description of the topic, runtime complexity and a changelog of the algorithm implementation. The description/information is part of a plugin and must be provided by its author, as discussed in Section 4.3 and illustrated in Listing 4.1.

2. NetLuke offers several options to customize visual effects of module visualizations or to modify certain UI features. A user may also activate settings in order to receive more textual feedback regarding the algorithms state. The **algorithm console**, if incorporated by a developer, provides textual information about algorithmic steps and execution. Its counterpart, the **debug console**, provides additional textual information about network and JavaScript errors. This can be very useful when developing a new plugin. The **effects setting**, can brighten up learning experience with graphical effects such as transitions, blurry shadows and gradients. This option should only be enabled on a fast computer and a browser with good hardware acceleration support such as Google Chrome, Internet Explorer 9/10 or Mozilla Firefox 10+. It is disabled/unavailable on mobile devices by default.

3. This button opens a dialog, which explains the particular items and buttons within the UI (help page).

4. Clicking this button redraws the stage (drawing area). If errors in the rendering of shapes occur, triggering the button can restore a failed rendered stage, However, this function does not refresh the page or loads data from the network.

5. This button triggers a dialog asking the current user if he wants to logout, redirecting back to /login. In the logout process all running instances will be saved for future use, however, graphical representation (coordinates of shapes) will not be saved. Therefore, positions of shapes can vary from session to session.

6. The button opens an overly with information about the current version of NetLuke including a changelog, a list of bugs and issues.

7. This button slides down a module implementation specific list of control elements. They are used to control e.g. insert, delete, modify structures of the current instance. Developers must provide the control elements within this container. The container div element is named "scroller" internally.

Figure 6.2: Desktop UI - Bottom Menu

The bottom menu illustrated in Figure 6.2 controls algorithm execution and comprises functionality for image scaling and manual instance handling.

8 The back (in execution history) button restores a previous state of an algorithm instance. First use enables the "history mode".

9 The play button only exists in certain sorting or graph algorithms, and is the equivalent to the play button of a video player. It runs the algorithm through in steps (equivalent to next) in a user defined time, until the algorithm is finished.

10 When clicking the next button, the algorithm changes to its next possible execution state.

11 The Algorithm "QuickSwitch" select field allows fast switching through deployed algorithms directly instead of using the algorithm menu button in 1.

12 The save button calls a dialog for manual instance saving. A user has to provide a **valid savename** (>3 characters) and an optional **description**. Duplicate savenames are overwritten by default. The current stage (visualization) will be exported to a preview image.

13 Opens a list with all stored instances, their description and a preview image. Users can load or delete an instance. In addition a user can view previews of saves.

14 The export trigger, exports the current stage to an image (transparent png) for further illustration purposes. The image can be downloaded directly.

15 This trigger allows panning of the stage. It drags the stage with all containing shapes. This function is very useful with zooming, in particular when your screen size is limited.

16 & 17 Magnifies or shrinks the current stage in 5% steps.

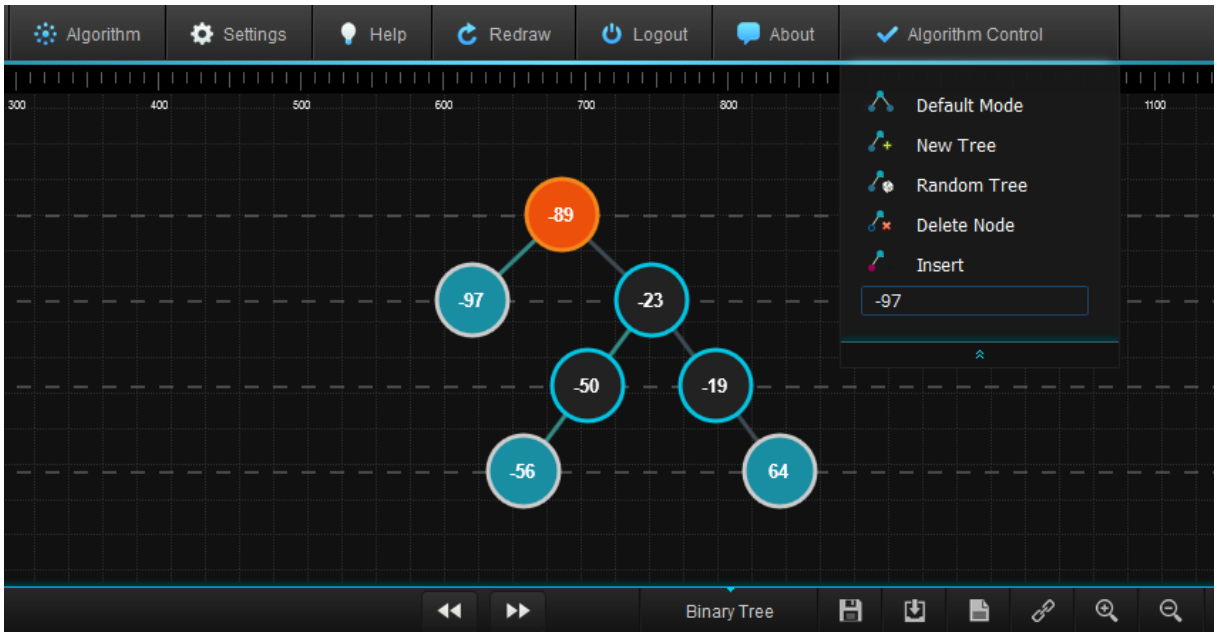

Figure 6.3: Desktop UI - Main View

6.2.2 Mobile UI

Mobile devices are sometimes very different to desktop computers and occasionally very similar. We discussed this paradox circumstance in Section 3.1.2, 3.3.2 and 3.5. As a matter of fact mobile clients require a massive amount of UI adaptions concerning layout, styling and sometimes features compared to a desktop UI [13, p. 118]. One of our requirements stated (see Section 3.3.3), that look & feel of UIs should be similar. In order to achieve this, we implemented JS routines to classify a client, made adoptions to the CSS stylesheet and adjusted the DOM structure.

The first question that arises is: What is a desktop device and what defines a mobile one? The answer to this questions depends on the viewpoint. Judging from physical hardware indicators (screen dimensions, processor, weight) the differentiation is rather easy. From a web developers point of view it is very difficult to determine the relevant software capabilities of a client, as boundaries between devices are blurred. Precise classification of devices is complicated, but heavily required in making the UI user friendly. For instance: How to distinguish a mobile high-end phone from a modern tablet computer, including the OS? For example, mobile devices (phones and tablets) feature touch screen resolutions ranging from 800-x-480 to 2560-x-1600. They have multi-core processors and up to 2 Gigabytes of memory. Even, their browsers claim to support latest HTML5 technologies (which they do not, or inadequate)[1]. The essential problem is, that every software platform needs its own critical adaptions to overcome layout issues and error handling. JavaScript performance of mobile devices is still multiple times lower compared to a desktop platform [4, p. 3].

However, we "define" the distinction on the basis of two attributes: (1) does the client support touch based interaction over a touchscreen? (2) is the client identifying itself

[1]For further information visit http://mobilehtml5.org/

throughout requests (user agent)? In the current prototype we do not distinguish between tablets and smart phones. In terms of styling and layouting, we use a stripped down and adapted CSS style sheets for mobile clients (`mobile.css`) derived from the desktop platform (`desktop.css`). We do not distinguish between iOS or Android devices in terms of look & feel. Instead we opted for a coherent user experience throughout all device categories. Hence, the mobile UI is very similar to the Desktop UI, as you can see in the Figures from 6.4 to 6.6.

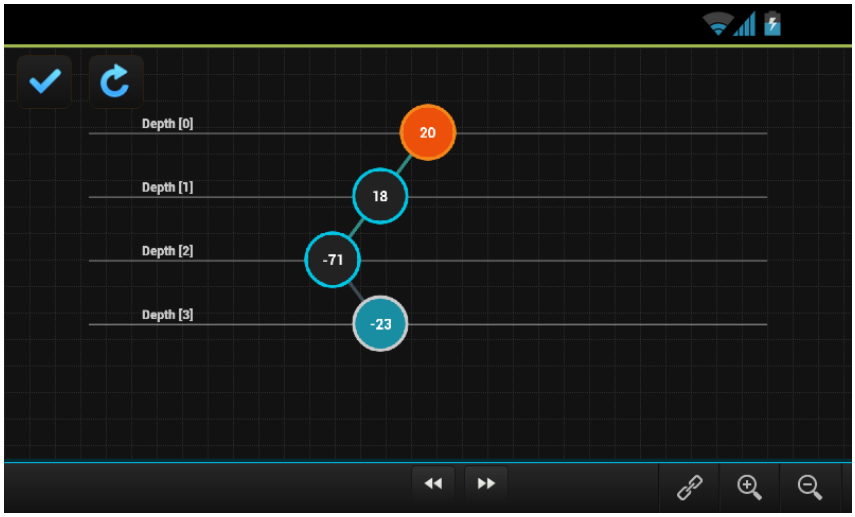

Figure 6.4: Android UI - Main View

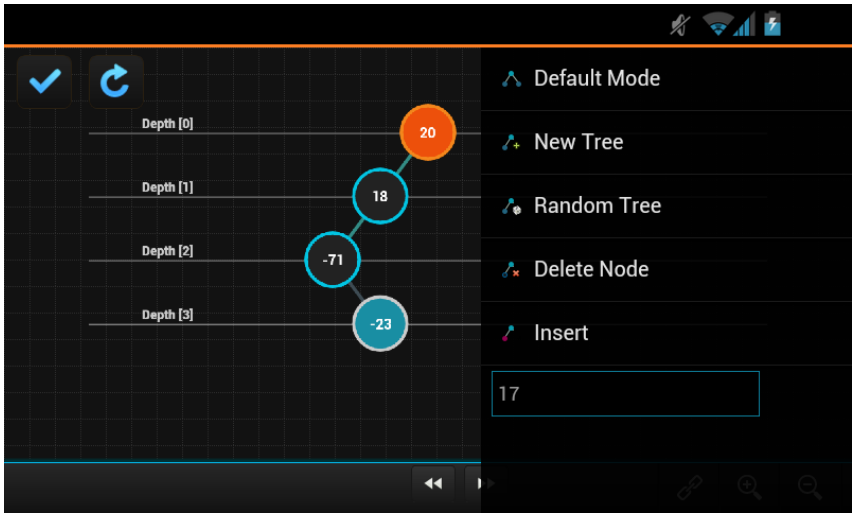

Figure 6.5: Android UI - Algorithm Control Menu

A major difference in using NetLuke on mobile devices is the way a user access the platform. Users do not point their browsers to an URL, instead they install the "NetLuke App". In the case of Android we use its *WebView* component [2]. On iOS we use *UIWebView* [3]. Both components, further referred as "WebViews", allow embedding of web content into a native application. This approach has several advantages in regard to our JavaScript module approach, as web content needs to be developed and deployed only once. A

[2]https://developer.android.com/reference/android/webkit/WebView.html
[3]https://developer.apple.com/library/ios/# documentation/uikit/reference/UIWebView_Class/

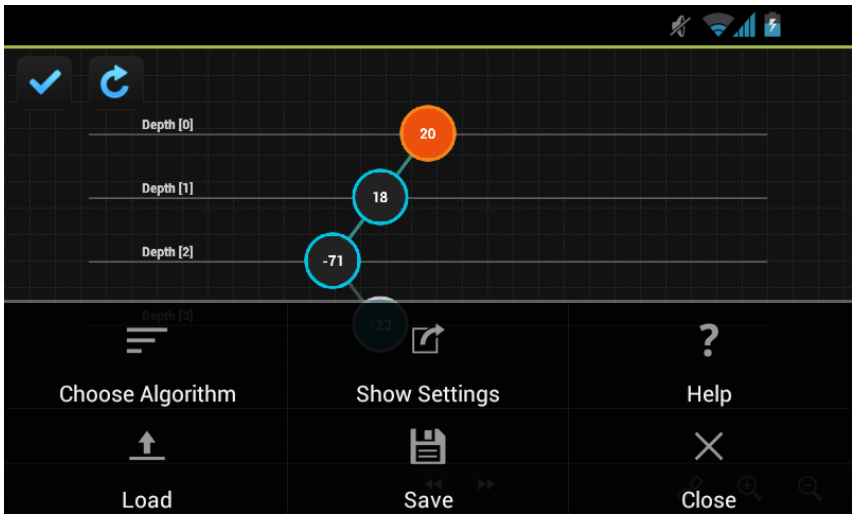

Figure 6.6: Android UI - Bottom Control Menu

major problem of "WebViews" are compatibility[4] and performance issues[5]. We discuss our mobile approach in Section 6.6 on the basis of our Android application.

6.3 Site Structure

Web applications typically require a mixture of static web markup with dynamic page elements. We also needed the ability to incorporate highly dynamic software components, not related to JavaScript, into our web pages. We decided to use Java Server Pages (JSP) over its (designated) successor Java Server Faces (JSF), mainly because JSP is sufficient for our type of application. Due to the nature of NetLuke, static content within pages is reduced to a minimum. The JSP pages act as a skeleton for dynamic content generated by JavaScript modules (`loader.js`) and Java alike. Of course in a web based environment JavaScript code needs to be embedded in HTML. The same principle is applied in NetLuke for non algorithm module scripts (see Section 6.5 for further discussion). We discussed our intention for an "application like" feeling, extending user experience over multiple dynamic web pages. Therefore, we decided to reduce the amount of pages on the basis of their intended purpose.

login.jsp [*/login*]

This page represents the start page of NetLuke for desktop clients. It provides basic information about the project itself and allows users to log in to the system. On mobile devices, this page is incorporated in the application. The link to `register.jsp` (/signup) allows clients to create a user account. Embedded Java/JavaScript routines check the clients physical screen resolution. If the resolution is below 1280-x-720 (HD) the page prints a message warning about possible UI layouting issues, as HD resolution is considered the minimum required resolution for NetLuke.

[4]Open issues on the Android platform are listed on https://code.google.com/p/android/issues/list. Entering the search terms "canvas" or "WebView" will illustrate the problems with the platform.
[5]http://www.codefessions.com/2012/03/how-fast-is-html5-canvas-part-2.html

index.jsp [*/home*]

The main page of NetLuke provides the UI together with actual visualizations. It incorporates the JavaScript modules listed in Table 6.1 in respect to the identified client platform type e.g. desktop or mobile. It provides the main view with its canvas element (stage) for visualization routines as well as every control component the system offers. JavaScript functions within `index.jsp` handle DWR errors, reverse AJAX messaging and client platform identification. After the page structure is loaded, all other subsequent content creation and manipulation is done by `loader.js`. Algorithm modules are not integrated in single jsp files, but in JavaScript files. Therefore visualizations and control elements are dynamically created, loaded and destroyed within `index.jsp` without page reloads. The reason for this behavior is due to performance and structural reasons. Less pages reduce load times and create a more application like user experience due to missing page loads. Developers are forced to use the provided UI components and do not have to create a new page, just one single JS file.

logout.jsp [*/logout*]

A users request to the logout page results in an immediate logout invocation of Shiro's management subsystem, resulting in automatic persistence of running module instances. After logout on a desktop platform, the user gets redirected to `login.jsp`. On a mobile device, the *WebView* closes and returns a user to the login/start view.

signup.jsp [*/signup*]

This page is used to handle registration of new users to the system. The page incorporates a signup form and a client-side validation routine (jQuqery.validate) to validate user data before submitting. After submit the request is dispatched to the UserController's `register()` method (see Section 5.5). The user is notified whether the registration was successful or not.

6.4 KineticJS

In this Section we introduce a very influential library in our project, which is the "HTML5 Canvas Framework" *KineticJS*[6]. All algorithm visualizations are derived and dependent from this critical component in our client architecture. Before we introduce KineticJS and its concepts, we explain why we require a library in the first place, instead of using the native pixel based HTML5 canvas[7] with its 2D rendering context object. For this purpose we look at the JavaScript code fragments in Listing 6.1 and 6.2. Both draw a green rectangle similar to the one in Figure 6.7. Although both methods are relatively compact and resemble each other in syntax, the KineticJS vari-

Figure 6.7: Simple Rect

[6]http://kineticjs.com/

[7]We also considered object-model based SVG for rendering graphics. However large SVG graphics with many containing elements are more resource intensive. In addition, SVG is incompatible with Android 2.x devices. We also consider HTML5 canvas to be more widely used and supported.

ant is far better to read. A closer look reveals that, the rectangle created by KineticJS is draggable, the other is not. In Listing 6.1 we can see the stage and layer objects, which are not directly available in the plain canvas example. In fact, those objects together with groups, create a very flexible environment for developers. Shapes can be grouped together and drawn on a layer, whereas many independent layers can co-exist in one stage. Every Kinetic object supports scaling, rotating, moving and many more visual effects. Event listeners can be attached to all objects, enabling interactivity with mouse/keyboard and touch devices. In summary, it can be stated that: KineticJS offers every feature needed to create highly expressive AV modules. KineticJS allows us and developers to create fascinating visualizations. Their extent is only limited by effort and time.

LISTING 6.1: KineticJS Rect Example

```
<script>
  var stage = new Kinetic.Stage({...});

  var layer = new Kinetic.Layer();

  var rect = new Kinetic.Rect({
        x: 239, y: 75, width: 100, height: 50, fill : 'green',
        stroke: 'black', strokeWidth: 4, draggable: true
  });

  // add the shape to the layer
  layer.add(rect);

  // add the layer to the stage
  stage.add(layer);
</script>
```

LISTING 6.2: Standard HTML5 Canvas Rect Example

```
<canvas id="myCanvas" width="578" height="200"></canvas>
  <script>
    var canvas = document.getElementById('myCanvas');
    var context = canvas.getContext('2d');

    context.beginPath();
    context.rect(188, 50, 200, 100);
    context.fillStyle = 'green';
    context.fill();
    context.lineWidth = 7;
    context.strokeStyle = 'black';
    context.stroke();
  </script>
```

KineticJS simplifies drawing to an HTML5 canvas tremendously, while offering a powerful object oriented API[8] capable to create any kind of aggregated shape including attached event listeners/handlers. KineticJS reduces coding efforts extremely, by extending and wrapping the 2D context of the canvas element, allowing developers to create very complex interactive and animated graphics, with very few lines of code. Another positive facet of KineticJS is its design philosophy striving for simplicity. Of course KineticJS has competitors – it is not the only canvas Javascript library out there. Projects like EaselJS[9],

[8]For a list of features visit https://github.com/ericdrowell/KineticJS/wiki
[9]http://www.createjs.com/#!/EaselJS

InfoViz[10] or FabricJS[11] are very popular. However, there are some very good arguments to use KineticJS over other solutions apart from features. One aspect is its high level of performance. Another advantage is its great accessibility for designers[12].

The author of KineticJS (Eric Drowell) offers lots of example code[13], a comprehensive set of tutorials, a well written documentation and of course support through Github[14]. Accessibility is a particularly critical aspect in Netluke, because future developers need to have a chance to learn how to create complex shapes through tutorials and documentation. It does not make sense to use a library, which is the most advanced in technical terms, but too difficult to comprehend by novice developers. Beginner-friendliness is most certainly something that applies to KineticJS. Another major advantage over other frameworks is its official support for mobile devices out of the box. Attaching mobile, or a combination of mobile/desktop events to shapes is very easy – thus making the framework the ideal choice for NetLuke.

Somehow, problematic are the fast development cycles of KineticJS, because API changes occur from time to time. Therefore, updating the library requires careful studying of the changelog[15]. We further discuss this subject in Section 7.2.

6.5 loader.js

The `loader.js` script, further referred as just "loader"[16] is the controlling/managing component on the client-side. The User Interface itself, its behavior and AV module implementations alike, are dependent from its functionality. The loader acts as a framework for the whole client application. Its tasks, features and functions are listed below. The inherent relevance of this component becomes clear when looking at the scenario illustrated in Figure 6.8.

Tasks and Features

- The loader attaches/integrates AV modules into the system, by creating an abstract layer on top of them. The loader is acting as a container for modules. This allows its routines to access objects and attributes within AV modules the loader is not aware of during "compile time". However, developers need to "bind" their implementations to the loader, and follow the "conventions" in Part II of the thesis, in order to allow access. For instance: In line 2 of Listing 6.1 we can see the stage object, which represents the main drawing area. Every module has one `loader.stage` object containing all related KineticJS objects (groups, shapes, layers). This way functions can act globally on modules, independent from each other.

[10] http://philogb.github.io/jit/
[11] http://fabricjs.com/
[12] http://stackoverflow.com/questions/8938969/current-state-of-javascript-canvas-libraries2012
[13] http://www.html5canvastutorials.com/kineticjs/
[14] https://github.com/ericdrowell/KineticJS/
[15] https://github.com/ericdrowell/KineticJS/wiki/Change-Log
[16] The loader.js file is only accessible upon a valid login, and therefore only subject to the main UI in index.jsp.

- Another core task is client-side instance handling. The loader initializes, instantiates and destroys a corresponding module automatically if a user switches to another topic (module). Designing functionality, which dynamically loads and destroys (to avoid memory leaks) JS plugins/modules unknown to the system, is not trivial. If module implementations are not correctly bound to the loader, this functionality is a major source for JS errors. Therefore developers need to stick to the design guidelines in Part II.

- Since all modules use the global `loader.stage` object, the loader is able to provide common features across all AV modules such as saving/loading, stage panning, zooming, image exporting and much more. Module developers do not have to be concerned about any of these features in their own implementations.

- The loader provides DOM injection of necessary AV and system control elements including event listeners/handlers. All UI menus and dialogs, apart from module specific ones, are created by the loader e.g. help dialog, about, settings, etc.

- The loader also covers aspects of responsive web design. As a matter of fact, our application is way to complex to just use a mobile optimized CSS style sheet for solving mobile UI problems (see Section 7.1). Therefore the loader has to incorporate workarounds for mobile devices before modules are loaded. Some workarounds have to be applied once (viewport setting, stage sizes, device-pixel ratio, etc.) while others have to be applied each time a module is loaded. Currently we distinguish between mobile and desktop platforms directly. In future releases we will first check if a client supports touch interaction and then classify a client through evaluation of the user agent. By this measurement, we hope to achieve a higher level of compatibility, by introducing graduation in client classification, to respect blurring of boundaries concerning device categories.

- Provisioning of many "helper functions" for faster module programming. We pointed this out many times before, but developers should concentrate on visualizations. Therefore, the loader provides generic functionality to modules. To name a few: Parsing of XML data, layouting functions, device classification, random number generation, etc. The loader also provides information about system wide variables/attributes such as stage size, screen size or the current zooming factor.

Loader Usage Scenario

The scenario in Figure 6.8 illustrates tasks the loader performs during typical usage of the platform. As pointed out before, the loader is creating the UI during the loading process, in the sense that event listeners are attached to buttons, views get initialized and page element containers get prepared for usage. This process includes all necessary adaptions, which have to be made in relation to the device (mobile, desktop, os, etc.) type. After UI creation has finished, the loader is either loading the last used instance or clears all JavaScript objects and control elements from a previous instance. Subsequent activity occurs when a user is interacting with the system e.g with AV implementations.

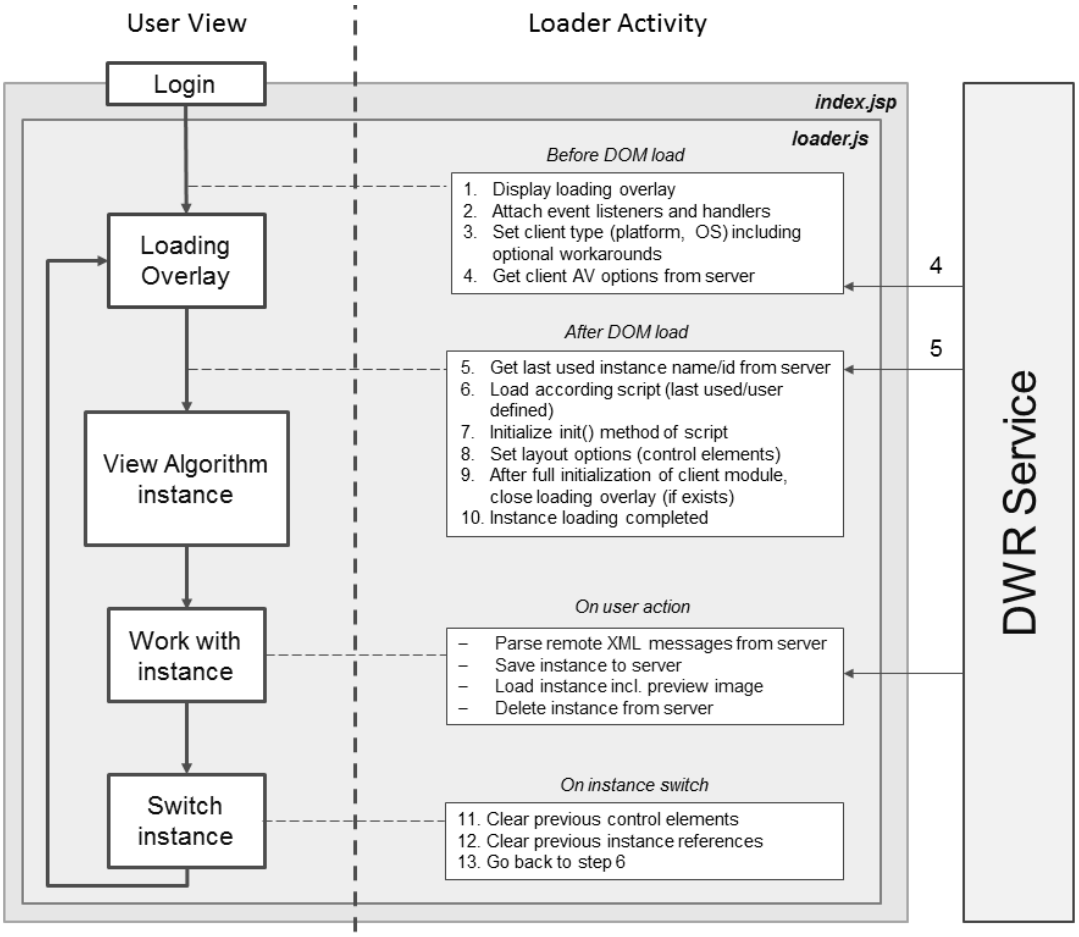

Figure 6.8: Loader Usage Scenario

6.6 Android

Since NetLuke is a multi-platform application we also want to introduce the mobile User Interface. At this point, we decided to introduce the Google Android UI, instead of the Apple iOS', as the UI prototype is already more advanced than the one for iOS.

In general there are two different approaches when developing a mobile application. A developer can either decide to build a native application for the preferred platform or he has the opportunity of building a so called "web application" or "mobile web app" [4]. Native apps have all the advantages of the specific platform including toolkits, device-emulators and, in case of Google Android development, a plugin for Eclipse which helps designing an application. The big disadvantage of native application is, that they only run on the specific platform. So if a developer decides to port an app to another platform, like iOS for instance, he has to start from scratch. This is where web applications come into play. These applications are built with platform independent technologies like HTML, CSS and JavaScript and run inside the browser of a mobile device. Due to the standardization of these technologies those applications run on all mobile devices and do not need to be ported. In practical use, workarounds have to be applied in order to ensure cross-platform compatibility.

6.6.1 Android WebView

During the design phase of NetLuke for Android we decided to take an hybrid approach between a native app and a web app. One big reason for this decision was, that we wanted the skeletal structure of our app to be native. That means that basic functionality like storing username/password combinations for future logins, or the implementation of the control menu is Android specific. Core functionality like displaying, algorithm visualizations, on the contrary, is specific to the web application part. Android, like most other mobile operating systems, offers a so called *WebView*, which incorporates web content in an Android specific view component. When connecting to a NetLuke server, the previously entered username/password combination is passed over to the *WebView*, which establishes a connection to NetLuke. As soon as a client is logged in, the main view, as illustrated in the last picture of Figure 6.9, is loaded. By using Android's *WebView* we also had the possibility to use the native *MenuBar* component of the mobile operating system to interact with the web content. Some control functions like changing the current algorithm or load/save are implemented using this *MenuBar* object. When tapping a specific button within this bar, a JavaScript command is being inflated, invoking NetLuke to perform an action (like opening the Algorithm selection menu). Therefore we can directly call JavaScript functions without binding them to Android buttons programmatically in the web code. This helps in separating concerns, but creates a high level of dependency to the loader.

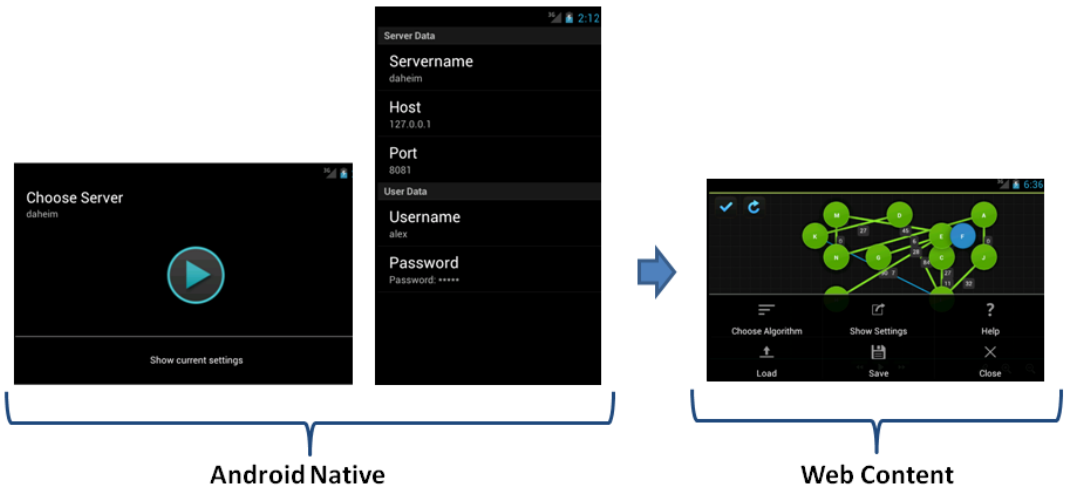

Figure 6.9: Android - Architecture

The connection process to NetLuke is divided into two parts. First, the username and password combination is being submitted and checked by the "mobilemm" servlet. In this stage of the connection process, we do not create a session for the client. Second, if the servlet returns the HTTP status code "200", the application uses the same user credentials to connect to the regular login site. At this point the client is associated with a regular session. If login fails, the user is notified.

6.6.2 Android Development

When developing the android application we ran into several problems. On big issue is the fragmentation of the operating system itself. Older versions of Android like Froyo (2.2) are still widespread and fairly common. Therefore we decided to support this devices. Another major issue is the rendering engine used by the *WebView* itself. While Google released a mobile version of its popular browser Chrome with a fast JavaScript engine (V8), the *WebView* is still based on the technology of the previous Android stock browsers. Due to a poor JavaScript performance of this engine the NetLuke application is sometimes suffering from little interruptions and other performance issues; especially on slower devices. Google announced, that the WebView engine will be replaced in Android 5 “Key Lime Pie” by the Chrome rendering engine, which should solve the performance issues. Another problem, that we weren't able to solve yet, is a bug related to KineticJS library. When first dragging a group of shapes the initial state is not going to be refreshed/cleared which leads to "ghosting"[17]. This means that there is a duplicate of this shape on the drawing stage which allows no subsequent manipulation.

[17]This bug is not exclusive to KineticJS. It seems like that Android's WebView implementation has problems clearing the canvas upon first user interaction. We opened a bug report concerning this issue on https://github.com/ericdrowell/KineticJS/issues/130, further information can be found in Google Android list of issues: https://code.google.com/p/android/issues/detail?id=35474

Chapter 7

Future Work

This Chapter summarizes our thoughts on the current state and future of NetLuke. We decided to conduct an analysis of the current system by taking a look at unresolved issues in Section 7.1. We also critically question our architecture/concept in Section 7.2. First we discuss UI issues, concepts to enhance, as well as components which may need redesign. We also look at at aspects of NetLuke we are very happy with, such as our concept of dynamic module instances, the separation of presentation and business logic in general as well as with the desktop UI. We continue with existing and future module implementations in Section 7.3. The following Section 7.4 covers documentation and modalities for project access.

7.1 Unresolved Issues

Our project needs to be classified as something in between **proof of concept** and **beta release**, although we speak of the application as a prototype. NetLuke is, although not completely finished, ready to use as a learning platform for students. Developers may add modules as well. This classification is also based on the current implementation progress, but is also derived from unresolved issues that plague NetLuke. We encounter many problems in the mobile UI part and complications in the way modules are incorporated in the client system. Readers of this thesis should be aware of the fact, that both authors are not experts in the field of UI development in general as well as in HTML5/CSS3/JS. Therefore we do not know every framework, technique or workaround which may lead to better UI implementations or module integration. We begin with the mobile UI, as we encountered many difficulties during its development.

Mobile UI

In this context we have to differentiate between Google Android and Apple iOS. In general, we had far more serious problems with Google's platform then with Apples'. We are aware of the fact, that Android runs on many different devices in multiple versions, resulting in a fragmented market share. Apple has virtually one version[1] of their iOS(6) installed on their devices. Resolutions on this devices are also very homogeneous. In particular we

[1]Apple's update strategy is much more efficient, resulting in a far less fragmented OS environment.

are disappointed by the incredible problems Google has with their inconsistent *WebView* implementations concerning resolution-dependent page elements such as <canvas>. Our application faced different problems throughout every Android version from 2.2 to 4.2. Layouting, positioning of page elements and performance degradation are in the center of concerns. Even with the limited amount of hardware devices[2] we had available for testing purposes, we encountered layouting problems in the menu structure, which sometimes resulted in inaccessible menu options. On other occasions the browsers HTML5 canvas implementation was too slow to interact with visual elements. For example, drag & drop did not work at all, because of poor frame rates. The problems are too manifold to explain them all in detail. However, we regularly update a bug report, found in the applications **About** section.

In newer Versions of Android (4.2.X) Google just recently switched to the mobile Chrome browser[3], which renders the native mobile browser we know from Android 2.1 through 4.1.2 obsolete – hence the adaptions and workarounds. This step is necessary, as Androids *WebView* uses a stripped down code base of their standard browser. HTML5/CSS3 performance for element transitions, transforms and animation is very poor compared to iOS devices. It seems like that Android *WebView* was not designed for highly dynamic content. We get this impression, because the supported features are not backed up by solid implementations. Therefore we highly appreciate Google's decision to use Google Chrome in future releases as JavaScript code execution and compatibility with popular HTML5 concepts will see enhancements. The situation on iOS devices is different, but not perfect. We do have to make iOS specific adaptions to the client-side code (impact of the devicePixelRatio attribute and viewport attribute). However, once a workaround is in action for a particular iOS device, it is not limited to a single device, where as a workaround for Android may cause errors on other devices. This behavior is very frustrating as workarounds lead to fragmented/unreadable code.

Therefore, a high level of user experience (a key requirement), could not be reached on Android devices. Experience varies from **decent** to complete inability of usage. We also have to admit that designing a mobile UI is much more difficult than we expected. Our approach of modifying an existing UI (desktop) did not work out as expected. We started developing and planning the mobile application back in 2011 where the mobile sector was relatively manageable in terms of hardware capabilities and platforms. Since then many new hardware devices emerged. On the other hand new JavaScript/HTML5 frameworks gained popularity, which overcome the obstacles of mobile development or at least try to. Interaction patterns changed rapidly as new ways of mobile web UIs found their way into the web. However, poor canvas rendering performance on most Android devices is still very likely to be seen, as our own very recent tests with a Google Nexus 7, Galaxy Nexus or a Google Nexus 4 showed.

Desktop UI

In contrast to the mobile UI, the desktop implementation offers a decent user experience, with minor layouting issues in the bottom control menu and the dialogs. We are

[2]We tested NetLuke on the integrated Android Virtual Device emulator, as well as on Samsung Galaxy S devices with Android Versions ranging from 2.2 to 4.2.

[3]https://www.google.com/intl/us/chrome/browser/mobile/

very satisfied with its performance (loading, memory usage, rendering performance) and robustness across all major browsers. The UI is simple, easy to work with and predominantly consistent in look & feel. Moreover, it does not contain severe bugs, which result in unexpected behavior. As a result, many requirements defined in Section 3.3.3 could be met. However, we do believe that the UI needs a redesign in certain areas. We are not happy with presentation characteristics, in particular with aesthetics. For instance: The algorithm menu looks blunt, the overall theme is unfinished and the icon set needs to be unified. This is partly because CSS markup is inconsistent. We use many defined CSS rules only in one area of the page, instead of using a more coherent concept e.g. site-wide consistency of element styling. The CSS files itself needs optimization regarding cascading rules and inheritance. This point is also particularly important for future developers, as they need to know which CSS to use in creating HTML control structures.

Server Components

Altogether we are very pleased with the server implementation. We did not encounter any serious errors leading to crashes or corrupt instances, during normal operation. However, we can not exclude errors, instead expect them to be found sooner or later. Some central components (NC, AOC) may suffer from memory leaks, serious bugs, others may exhibit "light errors". The testing process is still ongoing. Debugging web applications running in a servlet container requires good tool support and time, but in the end we hope to track down many errors unknown to us right now. We plan to use VisualVM[4] for performance profiling, together with JUnit tests for further intense testing of all components. What we have seen so far is that memory consumption is very high. This needs to be addressed. We expect that memory will constrain the applications performance severely. This may be seen when more then 10 users use the application concurrently. Performance degrades (response times) which implies that NetLuke does not scale very well at the moment. Another big "construction site" is the persistence layer of NetLuke. Its main component (DataSourceQuery) is inefficient. For example: DSQ has many duplicate code parts, some operations require a delete operation before DB writing can be conducted (instance restore). The methods containing the queries are most likely not thread safe. The service layer needs work, the Web Service implementation is still unfinished and remains untested. A Web Service client prototype is also missing.

7.2 Architecture

Given the fact that both Authors did not have any experience in JavaScript, Android or iOS development, the client architecture and implementation is most likely not state of the art. This will become very visible at maintenance tasks and is also a reason for the UI problems we face. For instance, we do a lot of HTML content creation directly in JavaScript with concatenated strings representing HTML. This however, results in unstructured code, degraded performance and this practice further violates separation of concerns. The source for this problem is located in the `loader.js` script discussed in Section 6.5. Instead of creating a sophisticated lightweight framework for developers, we packed functions in this component, which are difficult to comprehend and inflexible. The

[4] http://visualvm.java.net/

script needs redesign[5] because it heavily mixes content, structure and styling amongst workarounds for mobile devices. Its overall complexity is very undesirable, as current JS modules are closely tied to the loader. Module dependent functionality within the *loader* object should be off-loaded to a separate object. This would allow separation of concerns permitting future developers to add general purpose functionality shared by other modules. This would lead to a better structured design reducing overall programming tasks.

The problem of code maintenance and change is amplified by jQuery, its plugins and KineticJS. Client-side components within `loader.js`, as well as algorithm modules, are very dependent to their enabling 3rd party JavaScript plugins. For instance: A simple API change in a newer version of KineticJS can break all deployed modules, making changes to all of them necessary. This is very likely to happen, as KineticJS is developed at a rapid pace, incorporating new features as well as bug fixes (see Section 6.4). A developer has to be aware for the changes made to the KineticJS library. If an update is scheduled it is absolutely necessary to check the KineticJS changelog [6], before actually updating the library. We started developing with KineticJS v3.x. – the current version is 4.4.3 (April 2013). Since then many new features and API changes made their way into this library. Its fast pace of progression was one determining reason for utilizing the library, as client-side programming concepts evolve and browser support for dynamic content gains significance.

Although the combination of requirements imposes architectural trade-offs, we consider, under the given circumstances, our prototypal implementation as a success. The comprehension of what a module is, how its development process takes place and how it is integrated into the system is the key for quality results. Indeed, the surrounding environment is doing the rest.

7.3 Modules & Features

Creation of a large quantity of modules did not have priority in our work, because the overall concept, its architecture and implementation was in the center of concern. Although module development for NetLuke requires skills and a training period, we are confident that future developers will create more modules. NetLuke is planed as a comprehensive, modern AV platform with a large amount of topics. In order to become a relevant AV platform a significant number of module implementations are required. What is certainly missing are complex sorting algorithms such as radix sort, quicksort or bucketsort. Hashing algorithms are missing, as well as queues and priority queue implementations and B-Trees or tries. For this purpose, *NetLuke Core* the library discussed in Section 4.2, needs to be extended. In general, we expect that some developers will exploit the huge potential NetLuke has to offer. KineticJS is one key factor in this regard, because it offers everything needed to create impressive visualizations. From a technological point of view, there are many interesting technologies to investigate facilitating new features. Interesting would be to integrate web workers, which allow parallel execution of JS code particularly interesting on mobile devices. A feature we miss is client-side saving of element positions.

[5]Redesign of *loader.js* is scheduled for future releases.
[6]https://github.com/ericdrowell/KineticJS/wiki/Change-Log

At the moment loaded elements are positioned randomly on the screen. We may implement accurate positioning by using the concepts of W3C WebStorage[7]. Other features, enhancements and more stuff we like to see in future releases:

- The ability to use a secure communication protocol (HTTPS).
- A "pause" button to halt the execution of an algorithm instance.
- The Web Service needs to be fully implemented, including a client prototype.
- A new, better looking loading indicator in the left bottom of the screen.
- New mobile UI interaction elements (buttons, which adopt to bigger mobile screens and sliding menus.
- JavaDOC for all server methods and components are currently missing.
- A general code cleanup, more efficient use of concepts i.e. design patterns.
- Full UI keyboard control (for faster data input).
- Enhanced remote interfaces (support of String data type).
- JavaScript tool-tips on buttons and menus.
- Additional visual effects supporting visualizations of algorithms e.g. fading on delete of shapes, etc.

7.4 Documentation

Even though this thesis discusses ideas, concepts, design and system architecture of NetLuke, many components and processes are not covered in full detail. This is due to the limited scope of this thesis where we had to focus on core aspects.

We plan to provide further documentation in different forms. On the one hand, we will updated the source code itself with more comments as well as extensive JavaDoc annotations. We plan to integrate a Java API documentation for NetLukeCore first, before we document other projects. By this measures we want to provide future developers with more information regarding the inner mechanisms of NetLuke, resulting in a shortened module development process. On the other hand, NetLuke will be hosted on a project site at the University of Vienna. The project itself, including all sub-projects, will be available for download. Furthermore we will provide a quick start guide for contributors on how to get started with development tasks. The guide is planned as a short version of both parts of the thesis, giving architectural insight and direct step-by-step instructions.

[7] http://dev.w3.org/html5/webstorage/

Chapter 8

Conclusion

The modern concept of Algorithm Visualization is about supporting teachers and learners of computer science in their specific tasks, by allowing interaction with topics in an explorative way. Their visualizations need to concentrate on the dynamics that lay within algorithms and data structures, while explicitly highlighting the particular operations a system user is conducting. System usage needs to be easy, consistent and not limited to a specific operating system.

The NetLuke project was motivated by the fact, that existing AV software does not follow our vision of a modern concept, as most of the solutions are not directly aimed at supporting learners or teachers, instead often resemble static representations. In this thesis we introduced the NetLuke project making four major contributions to the field:

1. A multi-user AV platform accessible through standard web browsers, or Web Service clients, making the system independent from the client operating system. NetLuke supports desktop computers, mobile devices and other touch based computers. The platform separates business logic implemented in Java, from JavaScript presentation, by facilitating the concepts of DWR. The platform is very flexible in terms of integrated topics, functionality and modification.

2. Invention of a module based system, which allows developers to extend the platform with their own visualizations, by using standard programming languages. The idea behind modules, is to provide solutions "from learners – to learners". We allow module developers to reuse code from NetLuke's predecessor and offer a core library for faster and more robust development of modules. The framework character of NetLuke enables contributors to focus on developing modules, rather then concentrating on UI integration tasks.

3. NetLuke's modules feature highly expressive visualizations, allowing users to freely create and examine algorithms and data structures. Modules aim to be intuitive, educationally effective and supportive. They incorporate latest web technologies making them more future proof and visually attractive than Java applets.

4. A new philosophy of designing AV learning platforms, characterized by the shift from statically determined visualizations of algorithmic behavior towards interaction with

structures and control over dynamic behavior. The idea is to resemble interaction patterns in games and map them to interactive learning of algorithms and data structures.

Bibliography

[1] Nicolas Apostolopoulos, Harriet Hoffmann, Veronika Mansmann, and Andreas Schwill, editors. *E-Learning 2009: Lernen im digitalen Zeitalter*, number 51 in Medien in der Wissenschaft, Münster, 2009. Gesellschaft für Medien in der Wissenschaft e.V., Waxmann Verlag GmbH.

[2] John Bazik, Roberto Tamassia, Steven P. Reiss, and Andries van Dam. *Software Visualization in Teaching at Brown University*, chapter 25, page 382–398. MIT Press, 1998.

[3] Giancarlo Bongiovanni, Pierluigi Crescenzi, and Gabriella Rago. Jaz: Java algorithm visualizer. a multi-platform collaborative tool for teaching and testing graph algorithms. In *In Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualization*, page 73–80, 1998.

[4] Andre Charland and Brian LeRoux. Mobile application development: Web vs. native. *Queue*, 9(4):20:20–20:28, April 2011.

[5] Pierluigi Crescenzi and Carlo Nocentini. Fully integrating algorithm visualization into a cs2 course: A two-year experience. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE 2007)*, pages 296–300, Dundee, Scotland, 2007. ACM, ACM.

[6] D. Cruse and L. Jordan. *HTML5 Multimedia Development Cookbook*. Packt Publishing, Limited, 2011.

[7] J. Farrell and G.S. Nezlek. Rich internet applications the next stage of application development. In *Information Technology Interfaces, 2007. ITI 2007. 29th International Conference on*, pages 413–418, june 2007.

[8] David Flanagan. *JavaScript : The Definitive Guide*. O'Reilly, Beijing; Sebastopol, CA, 2011.

[9] A. Freeman. *The Definitive Guide to HTML5*. Apressus Series. Apress, 2011.

[10] Jean Greyling. Developing a set of requirements for algorithm animation systems. *International Journal of Computing and ICT Research, Special Issue Vol. 3, No. 1*, page 83–89, October 2009.

[11] HSQL Development Group. Hypersql user guide, jan 2013.

[12] Stacey Higginbotham. Mobile computing is killing the desktop pc, January 2009.

[13] Steven Hoober and Eric Berkman. *Designing Mobile Interfaces*. O'Reilly Media, 1 edition, December 2011.

[14] 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.

[15] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13:259–290, June 2002.

[16] Ville Karavirta. *Facilitating Algorithm Animation Creation and Adoption in Education*. dissertation, Helsinki University of Technology, December 2007.

[17] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.

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

[19] Bruce Lawson and Remy Sharp. *Introducing HTML5 (2nd Edition)*. New Riders Press, 2 edition, October 2011.

[20] 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.

[21] Lauri Malmi, Ville Karavirta, Ari Korhonen, Jussi Nikander, Otto Seppälä, and Panu Silvasti. Visual Algorithm Simulation Exercise System with Automatic Assessment: TRAKLA2. *Informatics in Education*, 3:267–288, 09/2004 2004.

[22] David Marginian and Joe Walker. Direct web remoting, December 2012.

[23] Kim Mateus. Web usage prediction: When mobile and desktop collide, March 2011.

[24] Yoram Moses, Zvi Polunsky, Ayellet Tal, and Leonid Ulitsky. Algorithm visualization for distributed environments. In *Proceedings of the 1998 IEEE Symposium on Information Visualization*, INFOVIS '98, pages 71–78, Washington, DC, USA, 1998. IEEE Computer Society.

[25] T. Mudner and E. Shakshuki. A new approach to learning algorithms. In *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, volume 1, pages 141 – 145 Vol.1, april 2004.

[26] David Nagel. Desktop pcs to see further decline in 2012, August 2012.

[27] Thomas L. Naps. Jhavé;: Supporting algorithm visualization. *IEEE Computer Graphics and Applications*, 25(5):49–55, 2005.

[28] Jakob Nielsen. Response times: The 3 important limits, 1993.

[29] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: A case study. In *CAINE'09*, pages 157–162, 2009.

[30] B. Phillips. Designers: The browser war casualties. *Computer*, 31(10):14 –16, 21, oct 1998.

[31] Wishnu Prasetya. *T2 : Automated Testing Tool for Java User Manual*. University of Utrecht, 2007.

[32] T. Rajala, T. Salakoski, E. Kaila, and M.-J. Laakso. How does collaboration affect algorithm learning? a case study using trakla2 algorithm visualization tool. In *Education Technology and Computer (ICETC), 2010 2nd International Conference on*, volume 3, pages V3–504 –V3–508, june 2010.

[33] Guido Rössling, Markus Schüer, and Bernd Freisleben. The animal algorithm animation tool. *SIGCSE Bull.*, 32(3):37–40, July 2000.

[34] Purvi Saraiya. Effective features of algorithm visualizations. Master's thesis, Virginia State University, United States, Virginia, Blacksburg, July 2002.

[35] 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.

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

[37] Alpha Seeking. Understanding google's position in the platform war, December 2012.

[38] Clifford A. Shaffer, Matthew Cooper, and Stephen H. Edwards. Algorithm visualization: a report on the state of the field. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, SIGCSE '07, pages 150–154, New York, NY, USA, 2007. ACM.

[39] L. Sikos. *Web Standards: Mastering HTML5, CSS3, and XML*. Apressus Series. Apress, 2011.

[40] Tiobe Software. Tiobe programming community index for april 2013, 2013.

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

[42] Debbie Stone, Caroline Jarrett, Mark Woodroffe, and Shailey Minocha. *User Interface Design and Evaluation*. Morgan Kaufmann, Amsterdam, 2005.

[43] Tecosystems. Not dead yet: The rise and fall and rise of java, 2011.

[44] M.E. Tudoreanu, R. Wu, A. Hamilton-Taylor, and E. Kraemer. Empirical evidence that algorithm animation promotes understanding of distributed algorithms. In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*, pages 236 – 243, September 2002.

[45] Christian Ullenboom. *Java ist auch eine Insel*. Galileo Computing, tenth edition, 2012.

[46] Frank Zammetti. *Practical DWR 2 Projects (Practical Projects)*. Apress, 2008.

List of Figures

List of Tables

Listings

Appendix A

Abstract

A.1 English

Algorithms play a major role in computer science and therefore in academic training of undergraduate students. Without knowledge of fundamental algorithms, it is not possible to evaluate existing work in a precise manner, nor will one be able to design new innovative solutions. Learning and teaching algorithms is difficult as static learning materials such as textbooks with images or sketches are unable to explain the dynamics which lay within algorithmic execution. Algorithm Visualization (AV) software addresses these shortcomings with explicit visualizations of dynamic algorithm behavior. The purpose of AV systems is to support teachers in their lectures and students in self-study efforts. However, existing AV systems either lack technical refinement, such as platform incompatibilities or do not exhibit enough educational effectiveness. Some of them follow a specific design philosophy which does not suit the needs of users.

Therefore my colleague Alexander Rotheneder and I want to contribute in this field by introducing NetLuke - a new and modern web based AV solution, which aims to be educational effective, pedagogical valuable, feature rich and easy to use. NetLuke was designed to work on mobile devices and desktop systems alike. Thanks to latest HTML5 technologies the overall platform compatibility and look & feel are considered better than in most other solutions. This first part of the thesis focuses on the ideas, the architectural design philosophy and the implementation efforts which define the application. On the basis of consequently defined requirements, which respect the needs of different user groups, I want to demonstrate which considerations affected design decisions as well as the development process itself. NetLuke is considered as an advanced prototype implementation which will be extended by future student developers. This thesis will provide ideas and implications for future work as well as information regarding extensibility.

Keywords — Algorithm visualization, data structures, animation, education, HTML5, Java, JavaScript, Android, iOS, KineticJS

A.2 Deutsch

Algorithmen und Daten-Strukturen sind die zentralen Elemente in der Informatik und daher ein wesentlicher Bestandteil in der Ausbildung von Studierenden des Fachs. Das Wissen über diese fundamentalen Themen ist unabdingbar für das Verständnis von computergestützten Problemlösungsstrategien. Ebenso ist deren Kenntnis Voraussetzung für das Entwickeln von eigenständigen innovativen Lösungsansätzen. Leider stellt sich des Öfteren heraus, dass die angewandten Methoden in der Lehre, sowie die involvierten statischen Lernmaterialen, wie Abbildungen oder Texte, die inherente Komplexität und die Dynamik von Algorithmen nur unzureichend vermitteln können. Die software gestützte Visualisierung von Algorithmen (AV) kann hier Abhilfe schaffen, indem dynamische Inhalte sowie das Verhalten explizit, während der Laufzeit dargestellt werden. AV Software hilft somit Lehrenden Inhalte zielgerichteter zu vermitteln. Studenten hingegen, wird die direkte Interaktion mit Inhalten außerhalb der Klassenräume ermöglicht. Unglücklicherweise zeigt ein großteil der exisitierende AV Lösungen markante Schwächen in der technischen Umsetzung, im Konzept oder in der Fähigkeit einen relevanten pädagogischen Mehrwert gegenüber klassischen Lernmaterialien zu erzeugen.

Aus diesem Grund haben mein Kollege Alexander Rotheneder und Ich eine neuartige, moderne und vorallem effektive AV Software mit dem Titel “NetLuke” entwickelt, die speziell in technischer Hinsicht neue Wege beschreitet. NetLuke ist grundlegend für die Verwendung auf multiplen Platformen, wie Desktop Systemen, Tablet Computern und Smartphones ausgerichtet, wobei auf neueste HTML5 Technologien zurückgegriffen wird. Dabei stehen die Benutzer der Plattform, sowie die Ansprüche zukünftiger Entwickler im Vordergrund. Dieser Teil der Masterarbeit befasst sich mit Anforderungen, Konzepten, der Architektur, sowie Design-Entscheidungen die dem Projekt zugrunde liegen. Zudem erkläre ich die technische Umsetzung des aktuell vorliegenden Prototypen anhand von Fragestellungen die in der Designphase bestimmend waren. Ein Ausblick auf die Zukunft und die eigene Beurteilung des Projekts bildet den Abschluss dieser Arbeit.

Schlüsselbegriffe — Algorithmen, Daten Strukturen, Visualisierung, Ausbildung, Animation, HTML5, Java, JavaScript, Android, iOS, KineticJS

Appendix B

Curriculum Vitae

Georg Prenner

Geboren am	7. Jänner 1985
Geboren in	1130 Wien
Nationalität	Österreich

Ausbildung

Bundesrealgymnasium Fichtnergasse

Matura	2003

Universität Wien

Diplomstudium Politikwissenschaft	2003 - 2013
Bakkalaureatsstudium Wirtschaftsinformatik	2005 - 2011
Masterstudium Wirtschaftsinformatik	2011 - 2013

Publikationen

NetLuke: Teaching Algorithms and Data Structures in Mobile Environments
Georg Prenner and Alexander Rotheneder and Erich Schikuta, submitted to: Proceedings of the 8th European Conference on Technology Enhanced Learning, EC-TEL 2013