Master's Programme in Computer, Communication and Information Sciences

Design of an Application to Collect Data and Create Animations from Visual Algorithm Simulation Exercises

Master Thesis

Giacomo Mariani

MASTER'S
THESIS

Aalto University
MASTER'S THESIS 2020

Design of an Application to Collect Data and Create Animations from Visual Algorithm Simulation Exercises

Master Thesis

Giacomo Mariani

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.
Otaniemi, 27 April 2020

Supervisor: D.Sc. (Tech) Lassi Haaranen
Advisor: D.Sc. (Tech) Ari Korhonen

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Author
Giacomo Mariani

Title
Design of an Application to Collect Data and Create Animations from Visual Algorithm Simulation Exercises

School School of Science

Master's programme Computer, Communication and Information Sciences

Major Computer Sciences **Code** SCI3042

Supervisor D.Sc. (Tech) Lassi Haaranen

Advisor D.Sc. (Tech) Ari Korhonen

Level Master's thesis **Date** 27 April 2020 **Pages** 65+12 **Language** English

Abstract

Visual Algorithm Simulation (VAS) exercises are commonly used in Computer Science education to help learners understand the logic behind the abstractions used in programming. These exercises also present problems common in the daily work of Computer Science graduates. Aalto University uses the JSAV library to create VAS exercises and evaluate the solutions submitted by students. The evaluation process counts the amount of correct steps given by the user during the exercise. However, because more detailed data is not collected, teachers currently can not recreate and analyse the submitted solutions more in depth.

This thesis presents the design, development and evaluation of an application prototype, which can be easily integrated in existing VAS exercises created with the JSAV library. The prototype is called Player Application, and it is designed as a service that can be easily integrated into other systems while still remaining independent. The Player Application consists of two main independent components: the Exercise Recorder and the Exercise Player. A third important contribution is the new JSON-based Algorithm Animation Language, which is designed to describe, structure and store the data collected from the VAS exercises.

The prototype was successfully tested in an online environment by importing the Exercise Recorder into existing exercises and replaying the submitted solutions in the Exercise Player. The tests showed that its design and architecture were valid. Next, the aim is to create a mature application, which can be used at Aalto University and other institutions, in addition the prototype still needs further development to support more VAS exercise types.

Keywords Algorithm Visualization, Algorithm Animation, Algorithm Simulation, JSAV, Visual Algorithm Simulation

Contents

Abbreviations

AA	Algorithm Animation.
AAL	Algorithm Animation Language.
AAS	Algorithm Animation System.
ADS	Algorithm and Data Structures.
AV	Algorithm Visualization.
CS	Computer Science.
DB	Database.
EP	Exercise Player.
ER	Exercise Recorder.
GPAL	General Purpose Animation Language.
GUI	Graphical User Interface.
JAAL	JSON-based Algorithm Animation Language.
JS	JavaScript
JSAV	JavaScript Algorithm Visualization library.
JSON	JavaScript Object Notation.
LMS	Learning Management System.
NPM	Node Package Manager.
PA	Player Application.
PV	Program Visualization.
SQuaRE	Systems and software Quality Requirements and Evaluation.
SV	Software Visualization.
SVG	Scalable Vector Graphics.
TSV	Taxonomy of Software Visualization.
UI	User Interface.
VAS	Visual Algorithm Simulation.
XAAL	eXtensible Algorithm Animation Language.

1. Introduction

The use of Algorithm Visualization (AV) is widespread in Computer Science (CS) education, where it is utilized to teach learners about Algorithms and Data Structures (ADS). Courses of ADS are an integral part of undergraduate CS curricula, since learning ADS concepts is necessary in order to understand the logic behind the abstractions used in computer programming. Furthermore, the problems, structures and algorithms taught in ADS courses recur commonly in the day-to-day work of CS graduates. Algorithm Visualization has therefore been used in various forms as an important teaching aid to enhance learners understanding of ADS concepts.

AV has also been a subject of research for more than thirty years, and its first applications into education date back to the early 1980's, when pixel displays started to allow more complex visualizations [47]. During the years many systems have been created with the scope to produce not only animations, but also Algorithm Visualization exercises that could be integrated in ADS courses. These AV software systems were implemented with different technologies and programming languages, with the aim to have them running under X or Microsoft Windows. Some examples of such AV software systems are BALSA [5], Zeus [4], and TANGO [52].

Starting from the late 1990's Java became the dominating language, and this made possible the creation of more complex and platform independent tools. Examples of Java-based AV tools include Matrix [31], JHAVÉ [43], and ANIMAL [49]. During this time, results from research about the impact of AV exercises on learning outcomes started to highlight the importance of enhancing learner engagement [17, 44]. These changes paved the way for the development of Visual Algorithm Simulation (VAS) exercises, which were implemented for example in Matrix and ANIMAL, with the aim of engaging the user into simulating the algorithm's behaviour.

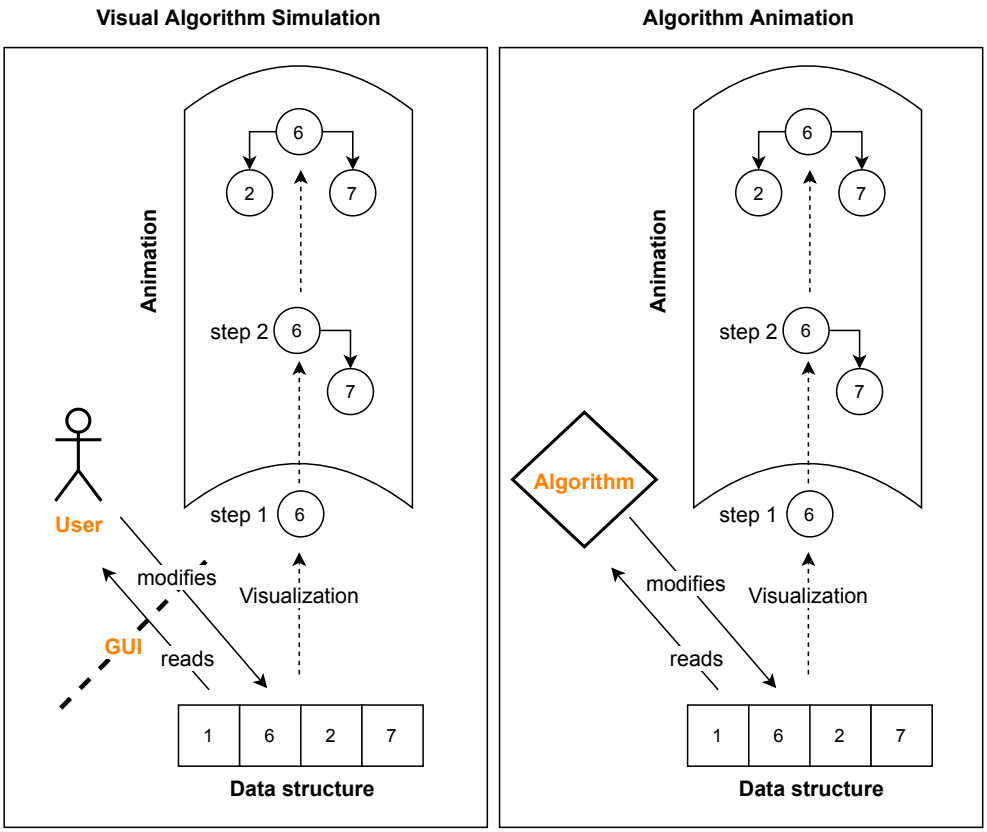


Figure 1.1. User doing a Visual Algorithm Simulation on the left, and pure Algorithm Animation on the right.

This evolution marked also the separation between the concepts of Visual Algorithm Simulation and Algorithm Animation (AA) within the AV field. In Section 2.2 we are going to look further into the concept of Algorithm Visualization, and the difference between Algorithm Animation and Visual Algorithm Simulation. For the moment, Figure 1.1 is an attempt to show the difference in practice between these two concepts: on the left VAS, where a user visualizes and modifies a data structure containing some values through a Graphical User Interface (GUI); on the right AA, where an algorithm directly reads and modifies a similar data structure. The effects of the actions of both the user and the algorithm on the data structure are visualized, and the sequence of the visualization steps creates an animation.

Support for Java on the Internet has however been decreasing in the last ten years, and this has brought to a demand for AV tools better supported on the web. Examples of such solutions are VisuAlgo [54], AlgoViz [1], and the JavaScript Algorithm Visualization library (JSAV) [24]. AlgoViz's aim is to create and maintain an online community for AV [2], with a collection of Algorithm Visualizations. VisuAlgo's goal is to offer a web-based AV tool, with a wide collection of AVs with integrated questions, and support

for JavaScript (JS) and HTML [15]. As for JSAV, its main difference from other modern AV software is that it is a JavaScript library. It is not aimed directly at learners, but rather at teachers and developers who want to create Algorithm Animations and Visual Algorithm Simulation exercises using JavaScript and HTML.

JSAV was created with the aim to add support for AVs in HTML5 and JavaScript, with also the possibility to create AV exercises using techniques involving learners on a deeper and more active way, such as simulating how an algorithm works. JSAV is currently used in ADS courses at Aalto University, University of Turku, Tampere University, and Virginia Tech. It has also been used to develop the OpenDSA interactive eTextbook [14]. One important feature of JSAV, which to our knowledge is missing from other JavaScript-based AV software, is the support not only for Algorithm Animation but also for Visual Algorithm Simulation exercises. In this type of exercises the learner can directly manipulate the data structure through the user interface (UI), with the aim of creating the visualization of the algorithm execution step-by-step, like shown in Figure 1.1. In VAS exercises JSAV offers various options to compare the learner solution with the model answer and to grade the submission. However, a standardized way of saving the steps of the submitted solution, and recreating them later as an animation, is currently missing. During a VAS exercise, the library automatically logs information on the user interaction with the visualization, which can potentially produce a large pool of data. This feature has been used, for example, in studies about students misconceptions. To our knowledge, up to now when the data collection and storage have been used, it has however been implemented at the single exercise level, with no standardized reusable approach [29, 33].

The focus of our research is at the conjunction of Visual Algorithm Simulation and Algorithm Animation, and more in specific about the recording of VAS exercise sessions and their transformation into animations. JSAV is also going to be at the center of our work, the reason being that we are interested not only in AV software, but in AV systems that offer the possibility to create both VAS exercises and Algorithm Animations. In this case JSAV is one of the few, if not only, modern open source solution at our disposal. Figure 1.2 shows the focus of our research: on the left side in green is the representation of a JSAV VAS exercise, while the part in red represents the recording, storing and replaying of the user interaction during the VAS exercise.

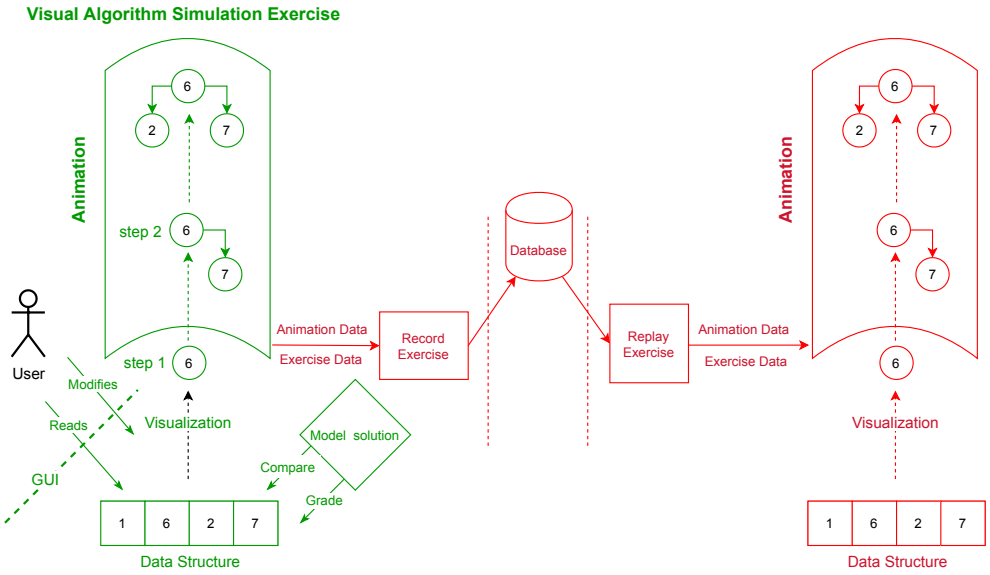


Figure 1.2. The focus of this thesis project. In green on the left is a JSAV VAS exercise, where the user interacts with a data structure through a Graphical User Interface (GUI). The part in red represents a system to record, store and replay user interaction during the JSAV VAS exercise.

The aim of our research is to design a prototype of a web application, which can be used in conjunction with JSAV Visual Algorithm Simulation exercises, to collect, store, and replay user interaction data in a standardized way. The application is called Player Application (PA), and it will be used as a tool to recreate the learners solutions as animations, starting from the collected data. Furthermore, the collected and anonymized data could also be valuable for researchers interested in *learning analytics*.

The design of such application is justified primarily by the need to offer learners the possibility to review their solution, for example in case of dispute on the received grade. Such feature could even be mandatory if the VAS exercises would be used in exams. In addition, the application would offer also other advantages, like the possibility to easily create AVs to be used as teaching material, and the collection of data for *learning analytics*.

To design the application we also needed to find an Algorithm Animation Language (AAL) able to describe in a satisfactory way the interaction between the user and the JSAV Visual Algorithm Simulation exercise. Therefore the research questions are:

1. Which solutions existing Algorithm Visualization software have used to record and replay Visual Algorithm Simulation exercises?

2. Which information should be saved in JSAV Visual Algorithm Simulation exercises submissions, in order to be able to replay them later as

animations?

3. How can we collect and replay JSAV Visual Algorithm Simulation exercises submissions?

This thesis is structured as follows. In the Chapter 2 we expose the background for our research, through a presentation of the concept of Algorithm Visualization and its history, and a review over existing solutions for Algorithm Animation and Algorithm Simulation. Chapter 3 is about the research methodology, which is based on Design Science. The results are presented in Chapter 4, which is structured according to the research methodology into three parts: the problem investigation in Section 4.1, the application design in Section 4.2, and the design validation in Section 3.4. Finally we discuss the outcomes of this thesis projects, their limitations, and the needs for future work in Chapter 5.

2. Background of Visual Algorithm Simulation

In this chapter we present existing knowledge, including research and solutions, relevant for the design of the Player Application. We start in Section 2.1 by explaining what Learning Management Systems (LMS) are, and why they are relevant for our research. After that, in Section 2.2 we introduce the concept of Algorithm Visualization, and its subdivision into Algorithm Animation and Visual Algorithm Simulation. In Section 2.3 we look into some existing, and recently developed Algorithm Animation Languages, which could be helpful to describe the Visual Algorithm Simulation exercises submission in our Player Application. In Section 2.4 we describe related work on collecting and replaying data from VAS exercises. In section 2.5 we present the JavaScript Algorithm Visualization library.

2.1 Learning Management Systems

Modern Learning Management Systems (LMS) are web-based information systems that are used in IT-supported learning [38], to organize and manage different tools needed to run a course, all within one integrated environment. LMS are also referred to as "learning platforms", "learning environments", "course management systems", "e-learning systems" [8, 10, 38], and more. One definition of LMS is that it "is a software application that automates the administration, tracking, and reporting of training events" [13]. Examples of tools and materials integrated within a LMS are course assignments, lecture material, discussion forums, submission and grading of assignments, chats, wikis and file sharing. Among the most used LMS world-wide are Moodle, Blackboard, Canvas and Brightspace D2L [16].

Figure 2.1 shows an example of how an exercise and its grading can be integrated in a LMS, and how the information is exchanged. This is

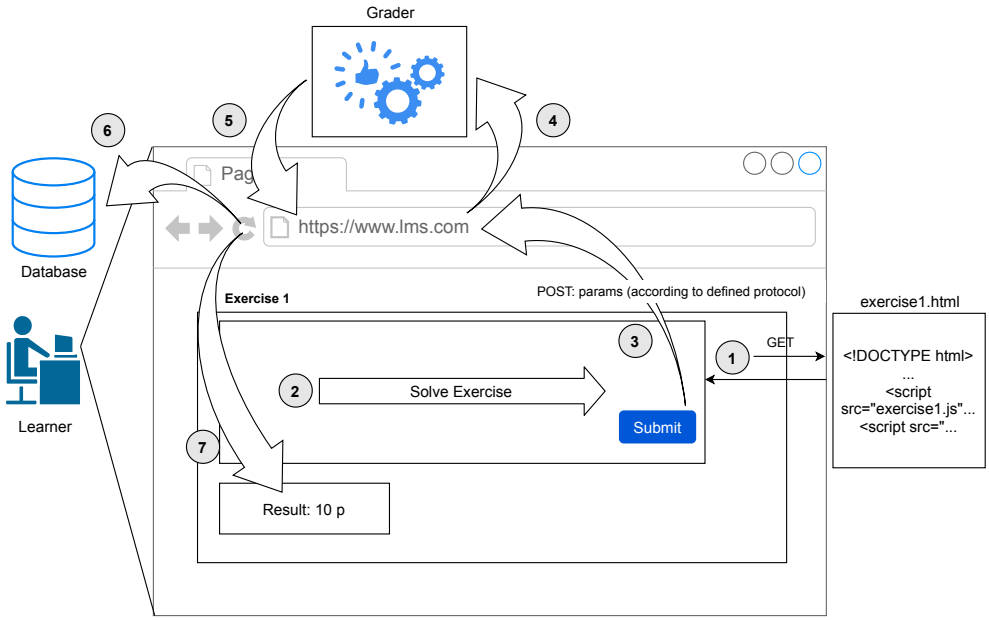


Figure 2.1. Example with integration of external content and services within a LMS: (1) Learner connects and LMS fetches external exercise; (2) learner solves and submits the exercise; (3) the exercise sends submission to the LMS; (4) LMS sends submission to external grader; (5) grader sends grading result to LMS; (6) LMS stores results in DB; (7) results are shown to the learner.

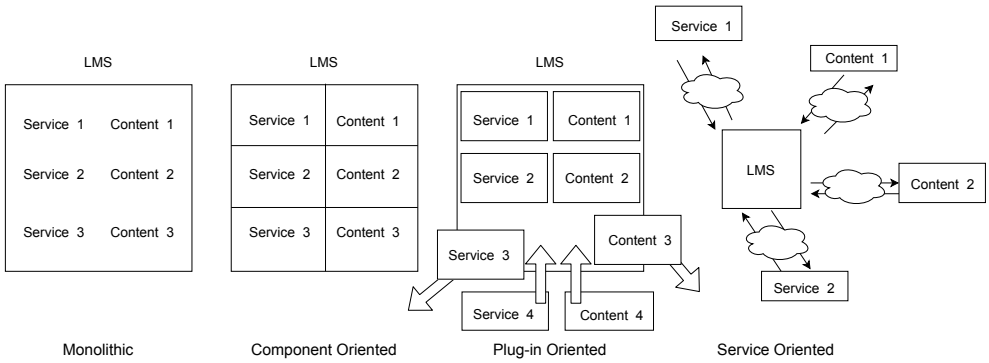


Figure 2.2. Types of architectures for LMSs. From the classic monolithic architecture to component oriented, plug-in oriented, and service-oriented. [28]

however just an example solution, since embedding of material and tools, and exchange of information can be realized in many different ways. One important aspect which will determine how the services and material will be embedded and information exchanged, is the architecture of the LMS. As shown in Figure 2.2, from the architectural point of view on one side there is the classic monolithic application, where all services and content are part of one big application, and extending the functionalities with external services requires a large amount of work. On the opposite side is the *service oriented architecture*, where each service works as a fully independent subsystem, and new functionalities can be added by integrating new services [28]. In between those two possibilities there are other architectural choices, where the services and contents can be more or less independent and integrated with various degrees of complexity.

A critical aspect in the integration of learning material and services, is how the information is structured and exchanged. For this purpose various *interoperability standards* have been defined, such as LTI [18], xAPI [58] and SCORM [50]. All these different specifications define how and which information is exchanged between the embedded content and the LMS, or between LMSs. While standardized solutions improve reusability and interoperability, they might have limitations when it comes to storing and interacting with the user profile information and work trace [48]. To overcome the limitations of standardized approaches, some institutions and work groups have developed nonstandardized solutions, in order to be able to handle, store and update the user information as needed.

Knowledge about LMSs is relevant for this thesis project because Aalto University, like many other universities as well, uses LMSs to embed and manage course exercises and materials within an integrated environment. The two main LMSs used for these purposes are Moodle [40] and A+ [28], with the latter being the one of choice for Algorithm and Data Structure courses.

2.2 Algorithm Visualization

The use of Algorithm Visualization (AV) is widespread in Computer Science education, where it is utilized to teach learners about Algorithms and Data Structures (ADS). Its use and impact on students learning outcome have been the subject of many studies since the 1990s, at first with conflicting results [6, 51]. Later however research has shown encouraging outcomes, especially regarding AV exercises demanding higher learner engagement [17, 45].

In order to understand the concept of Algorithm Visualization, it is important to first describe the concepts of *algorithm* and *data structure*. An algorithm can be defined broadly as a step-by-step procedure used to solve some problem or accomplish an end [39]. In the field of mathematics and computer science it is more strictly described as a "finite series of well defined, computer-implementable instructions to solve a specific set of computable problems" [37]. Computable problems are solved by processing data, which is organized into data structures in order to make the operation efficient. A data structure can be defined as a "way to store and organize data in order to facilitate access and modification" [9], and in computer science more precisely as a "collection of data values, the relationships

among them, and the functions or operations that can be applied to the data" [56].

With the term Algorithm Visualization (AV), we mean the use of visualization techniques to create a high-level description of an algorithm. The created description abstracts the data structures and operations without focusing on the particular implementation, which can be realized in different ways depending on the chosen technologies and programming language. AV is seen as falling under the umbrella concept of Software Visualization (SV), a concept that also includes Program Visualization (PV), which is distinct from the AV. A way to describe the difference between AV and PV is that if the visualization describes a general algorithm, then it is considered Algorithm Visualization, but if it describes the particular implementation of an algorithm, then it is considered Program Visualization [47].

Many studies in the last three decades have tried to determine the impact of various forms of AV on students learning outcomes. Multiple studies have shown that high level of learners engagement in the AV exercises is correlated to more positive learning outcomes, especially when the visualization exercises are carried out in groups [32, 35, 41]. Encouraging results have been reached even when AV exercises were not carried out in collaboration [30, 45].

The Engagement Taxonomy [44] was developed to define the levels of learner engagement during AV exercises, and was later extended in the Extended Engagement Taxonomy (EET) [42]. Table 2.1 shows the levels of learner engagement in the EET. Using the EET as a reference, we can call Algorithm Animation (AA) the visualizations that operate only on levels 1-4, and Visual Algorithm Simulation the visualizations that operate also on levels 5-7. Figure 2.3 shows the subdivision of SV and AV with the help of a Venn diagram.

Table 2.1. Extended Engagement Taxonomy [42].

0	No viewing	There is no visualization of the algorithm. Learners can review the code
1	Viewing	The visualization of the algorithm is shown but without interaction
2	Controlled Viewing	Learners have the possibility to control the animation, like speed, step sequence.
3	Entering Input	It is possible to change the input of the algorithm before execution.
4	Responding	Learners respond to questions during the visualization.
5	Changing	The visualization can be directly manipulated.
6	Modifying	The code can be modified before starting the animation.
7	Constructing	The learner constructs the visualization, for example with the help of a user interface.
8	Presenting	Learners present and explain the visualization.
9	Reviewing	Reviewing of the visualizations to give comments and feedback.

2.2.1 Algorithm Animation

In Algorithm Animation the different states of a data structure manipulated by an algorithm can be visualized, and the sequence of the states and their changes can be displayed as a visual animation. Furthermore, state history can be recorded to give users the possibility to step back and forth through the sequence [34].

In Algorithm Animation, both an event-driven and a data-driven approach can be used to create the visualization. The event-driven approach is based on the idea of identifying *interesting events* performed by the algorithm, and defining visualization actions to be triggered when the *interesting events* take place. The succession of visualization scenes creates an Algorithm Animation. The data-driven approach is instead based on the concepts of *state mapping* and *interesting data structures*. Here the assumption is that by observing the state changes happening to the data structures, it is possible to reconstruct actions taken by the algorithm. Finally by mapping the state changes to graphical events, the algorithm can be visualised. Furthermore the AA can be *live*, if the *interesting events* or *state mapping* result in a visualization at run time, or *post-mortem* if the visualization is created from logged data. [11]

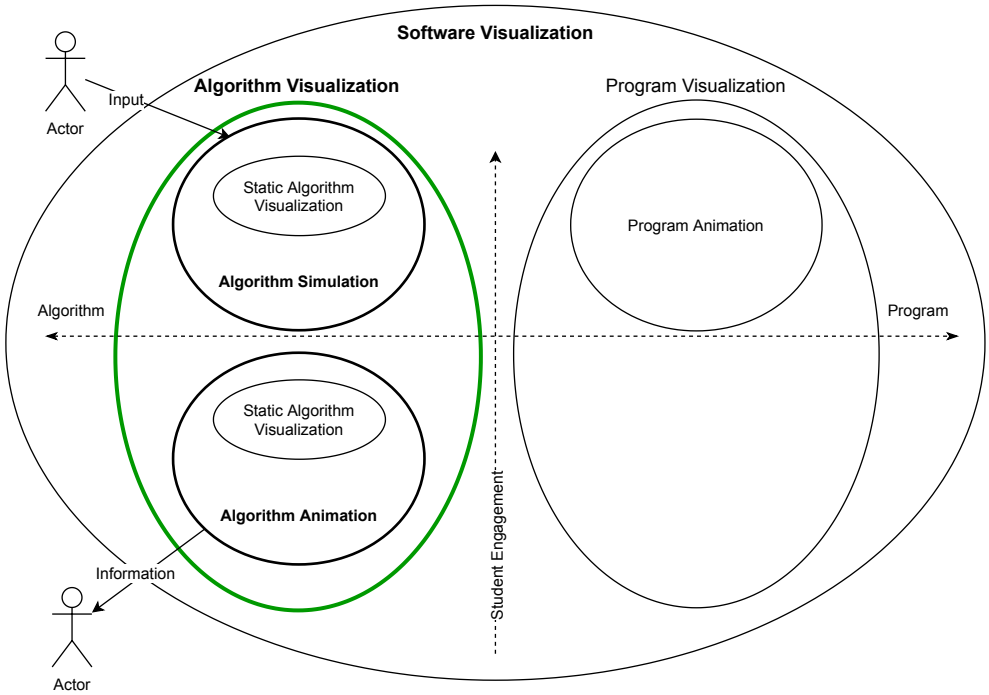


Figure 2.3. Subareas of Software Visualization and Algorithm Visualization.

2.2.2 Visual Algorithm Simulation

In Visual Algorithm Simulation the user can directly manipulate the data structure and construct the visualization using the user interface. Through the graphical tools offered by the UI, it is possible to change the data structures and therefore the animation [34].

The purpose of VAS exercises is to give the learner the possibility to explore different possible states of the data structure on which the algorithm is acting. Through the simulation, the learner can decide the state changes, in this way taking the role of the algorithm itself, with the final aim of recreating the algorithm execution step-by-step. VAS exercises enhance learners engagement, giving them the possibility to study the algorithm's behavior and its influence on the data structures, both under correct and incorrect execution. This process challenges them to find the correct solution. Visual Algorithm Simulation can be used to evaluate learners understanding of the algorithm, and to create Algorithm Animations for teaching purposes, which can also be embedded in online learning materials [14].

2.3 Algorithm Animation Languages

In order to collect, store, and replay the data arising from the user interaction with the Visual Algorithm Simulation exercise, we need a language able to describe the necessary metadata, data structures, events and visual elements required for the reconstruction of the VAS exercise as an Algorithm Animation. The language compatibility with JavaScript is also relevant, since JS is used to develop the VAS exercises in JSAV. Next we will explore two existing solutions which we consider potentially useful for our purposes.

2.3.1 The eXtensible Algorithm Animation Language

The eXtensible Algorithm Animation Language (XAAL) [22] was created to function as an intermediate language to exchange data among algorithm animation systems (AAS). XAAL was developed based on a survey of existing AASs and AALs [21], and the subsequent development of a Taxonomy of Algorithm Animation Languages [27]. The systems evaluated for the creation of XAAL were ALVIS, ANIMAL, DsCats, JAWAA, JSAMBA and

MatrixPro, while the AALs evaluated were AnimalScript, DsCats, JAWAA, Matrix ASCII, SALSA and Samba [22]. The language was designed to unify two existing different approaches for describing Algorithm Animations: one using mostly graphical primitives, and the other focusing on data structures.

The main features of XAAL are *graphical primitives*, *data structures* and *animation* [22]. A document is constructed using XML-schema modules and it is composed of four main parts: (i) metadata, (ii) definitions, (iii) initial state, and (iv) animation. The high level structure of a XAAL document is shown in Listing 2.1.

```
<xaal xmlns="http://www.cs.hut.fi/Research/SVG/XAAL">
  <metadata>..</metadata>
  <defs>..</defs>
  <initial>..</initial>
  <animation>..</animation>
</xaal>
```

Listing 2.1. Example of XAAL document structure [23].

XAAL offers the possibility to describe animations both by using only graphical primitives and by using data structures, in which case it is also possible to specify how the operation should be visualized using graphical primitives. In both cases the elements of the animation can be enriched with style and coordinates for positioning. Animations can be grouped into a sequence or as simultaneous execution, and timing of each operation within the animation can be specified.

The animations of graphical primitives can be described with a series of operations, which among others include show/hide, move, rotate, scale, group/ungroup, swap-id. In addition, the animations with data structures support the most common data structure operations which are create, remove, replace, swap, insert, delete and search. Listing 2.2 shows how an animation can be described using XAAL.

```
<xaal version="0.1" xmlns="http://www.cs.hut.fi/Research/SVG/XAAL"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.cs.hut.fi/Research/SVG/XAAL xaal.xsd">
    <initial>
        <array indexed="true">
            <index index="0"><key value="E" id="key1"/></index>
            <index index="1"><key value="X" id="key2"/></index>
            <index index="2"><key value="A" id="key3"/></index>
            <index index="3"><key value="M" id="key4"/></index>
            <index index="4"><key value="P" id="key5"/></index>
            <index index="5"><key value="L" id="key6"/></index>
            <index index="6"><key value="E" id="key7"/></index>
        </array>
        <tree id="tree" root="node1">
            <structure-property name="class" value="matrix.structures.CDT.probe.
            BinSearchTree"/>
            <node id="node1"><key value="E"/></node>
            <node id="node2"><key value="X"/></node>
            <edge from="node1" to="node2"/>
        </tree>
    </initial>
    <animation>
        <insert source="key3" target="tree"/>
        <insert source="key4" target="tree"/>
        <delete key="X" target="tree"/>
        <swap swap="key1" with="key2"/>
    </animation>
</xaal>
```

Listing 2.2. Example of describing an animation with XAAL [21].

2.3.2 General Purpose Animation Language

The General Purpose Animation Language (GPAL) [7] was developed as a tool for crating diagrams and graphs animations through scripting. It is based on two types of scripting, one to describe the visual element as Scalable Vector Graphics (SVG), and another one to describe the animation of the visual elements. The two scrips are converted with the aid of a parser into JavaScript and then rendered as an animation on the screen. Listing 2.3 shows an example of SVG scripting, while Listing 2.4 is an example of animation scripting using the elements defined in Listing 2.3.

```
// square(x,y,width,height,ID)
square(100,150,50,50,s1);
// circle(x,y,radius,ID)
circle(100,150,50,c1);
// triangle(x1,y1,x2,y2,x3,y3,ID)
triangle(306,292,356,442,256,442,t1);
// polygon(x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,ID)
polygon(605,170,655,200,585,250,605,250,605,170,p1);
```

Listing 2.3. Example of SVG scripting in GPAL [7].

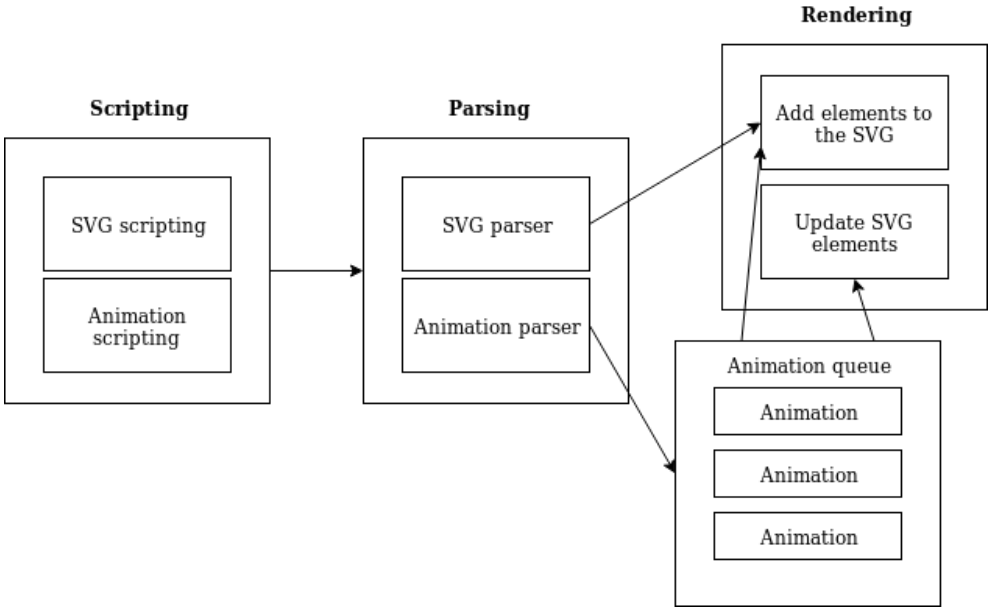


Figure 2.4. Architecture of GPAL [7].

```
function_main(){
    // Draw a new circle c2 on the right side of c1
    draw(c1,c2,circle,right);
    drawLeft(c1,c3,circle);
    // Draw a rectangle s2 on the top-right of s1
    drawDiagonal(s1,s2,rect,topright);
    drawArrow(s1,s2);
}
```

Listing 2.4. Example of animation scripting in GPAL [7].

The main elements in the architecture of the GPAL are the SVG and animation scripting, the SVG and animation scripting parser, the animation queue, and the rendering, which includes rendering and updating of the SVG elements. The structure is shown in Figure 2.4

2.4 From Simulation to Animation

In this section we are going to look into related work about the subject of collecting and replaying data from Visual Algorithm Simulation exercises. The only existing solution we have found is MatrixPRO [26], which has also been incorporated in the TRAKLA2 system [36].

MatrixPro is a tool for crating VAS exercises where the submitted solution can be saved and replayed. In MatrixPro both the data structures and the visualization itself can be stored. Data structures can be saved either as serialized Java objects or as ASCII files, and later recreated from these formats. The visual elements can instead be exported to SVG or to TEXdraw format.

TRAKLA2 is a system for VAS exercises with automatic assessment,

which incorporates the MatrixPro tool. As such TRAKLA2 also allows saving the operations performed by the user during VAS exercises as serialized Java objects. The object is saved every time a grade operation occurs, and it includes the learner answer as a sequence of data structure states. This serialized Java object is then sent to a server via Remote Method Invocation (RMI) protocol. The data is sent to the server as log entries when the applet is initialized, the exercise is graded or reset, the model answer is opened or closed, the user has been idle over 60 seconds, and when a user operation ends the idle time. These log entries include a time stamp and identification data on the course, exercise, learner and performed operation.

2.5 JavaScript Algorithm Visualization Library

The JavaScript Algorithm Visualization Library [24] was developed to add support for AVs in HTML5 and JavaScript, with the possibility to create AV exercises using active learning techniques. The key features of JSAV are: (i) automated layout for traditional data structures like array, linked list, tree, binary trees and graphs, (ii) support for presentation slideshows, and (iii) support for VAS exercises on the *constructing* engagement level. The other types of exercises supported are on lower engagement level, and they are static images, animated slideshows and pop-up questions.

In VAS exercises the algorithm simulation is done mainly by clicking the visualization of the data structure or a button, and the actions can be assessed automatically by comparing the current state to a model answer. At the end of the exercise the model answer can be reviewed by the learner in the form of a slideshow. The feedback can be set in limited mode, in which case when requested by the user the number of correct steps up to that point is shown, or in continuous mode, where the feedback is given after each relevant operation. In continuous feedback mode the exercise can be set to either undo the incorrect step and allow the learner to try again, or to automatically correct the wrong step with the model answer, in which case points for the wrong step are not rewarded.

The author of the VAS exercises is required to generate the necessary input data and initialize the data structures handled by the algorithm. Event handlers have to be attached to the data structures in order to apply the changes when the user interacts with them. The author also has to write the model answer with annotated the steps to be graded. If during

the exercise the wrong steps are to be fixed according to the model answer, the author also has to provide a function which takes the state of the model answer and fixes with it the learner solution.

Extension and customization of JSAV can be achieved through the many events triggered when the user interacts with the visualization. Likewise, the library listens to some events on its container. JSAV objects have multiple class attributes, from general to more specific, which can be used to listen for events on the needed exercise elements and customize them.

3. Methodology

The research was carried out following the design-science research methodology [55]. In design-science research the aim is to create an innovative artefact with a defined purpose, which yields utility for a specified problem domain. The design process is an iterative and incremental activity, in which the artefact's utility, quality and efficacy has to be demonstrated through a validation process. The feedback from the validation phase is then used in the investigation and design phase. In our research an iteration had the duration of two weeks and was repeated over a period of six months. Each iteration was marked by a meeting with the Aalto University instructor responsible for the Algorithm and Data Structure courses, who represented the application stakeholders and impersonated different user roles. The application stakeholders represented by the instructor were teachers, students, developers and researchers. The aim of the bi-weekly meetings was to analyze the current situation, better understand the needs and expectations of the application users, validate the outcome of previous iterations, and set the goals for the next iteration. Through these series of iterations we incrementally analysed the context, designed the Player Application prototype, and validated it. Next we describe into more detail the *design cycle* and its phases.

3.1 Design Cycle

Design Science research is carried out according to the *design cycle*, which is part of the *engineering cycle*. The engineering cycle includes: (i) *problem investigation*, (ii) *artefact design*, (iii) *design validation*, (iv) *artefact implementation*, and (v) *implementation evaluation*. The *design cycle* represents the first three steps of the engineering cycle and the process is iterated until the artefact, in the form of a prototype, is ready for implementation.

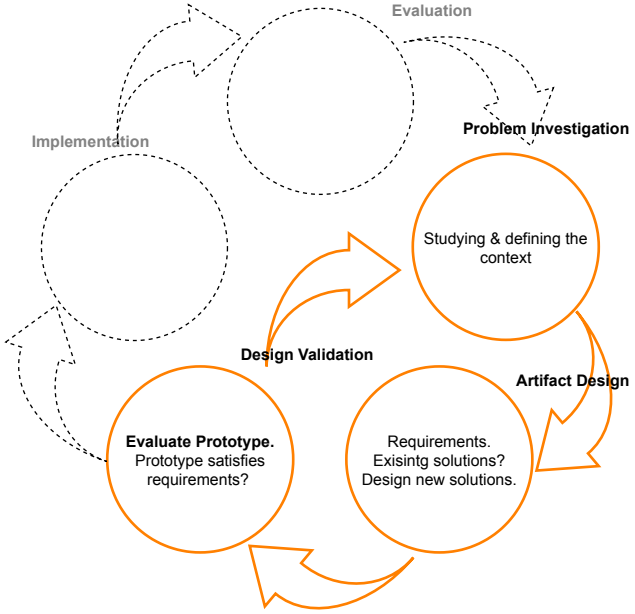


Figure 3.1. The design cycle as part of the engineering cycle.

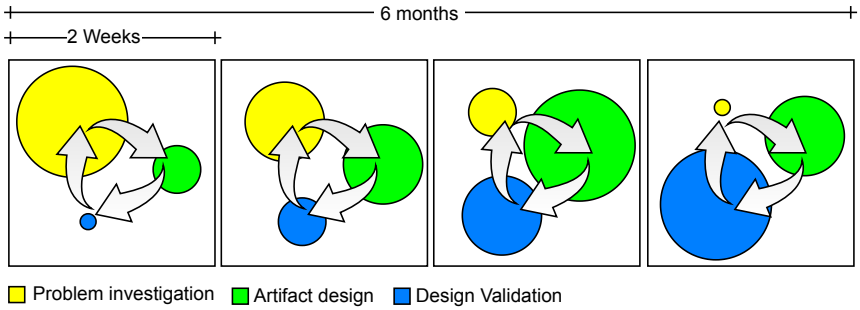


Figure 3.2. The phases of the design cycle within the cycle iterations. As we repeated the iterations, focus shifted from the problem investigation, to the artifact design and later to the design validation.

Figure 3.1 shows the *design cycle* as part of the *engineering cycle*. [57]

The goal for our iterations of the *design cycle* was to design a web application prototype, which can be used in conjunction with JSAV Visual Algorithm Simulation exercises, to collect, store, and replay user interaction data in a standardized way. As shown in Figure 3.2 the duration of each cycle was of two weeks, and it included the three phases in different proportions. The first iterations were dominated by the *problem investigation*, but later as we repeated them, the focus shifted toward the *artifact design* and later the *design validation*. Next we present the three phases of our *design cycle* from a methodological perspective.

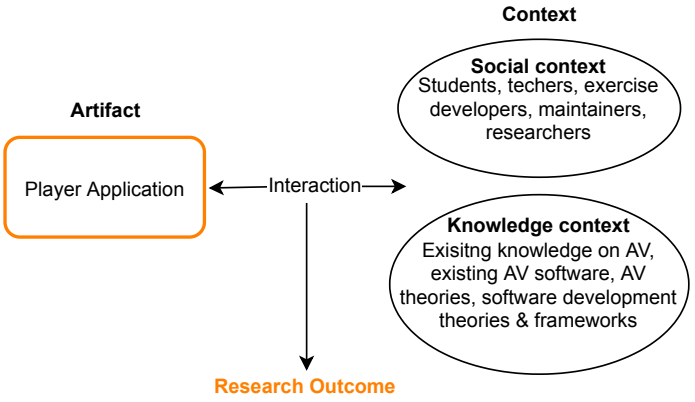


Figure 3.3. Research artefact and context.

3.2 Problem Investigation

The first step to reach our goal was the *problem investigation* phase. The aim of this phase is to investigate the context where the artifact is going to be used, along with the interaction between the two. In our research the artefact is represented by the Player Application, and its purpose is to recreate learners submissions of Visual Algorithm Simulation exercises as animations. The context can be divided into *social context*, representing the users, developers and maintainers of the application, and *knowledge context*, containing theories and existing knowledge about our research filed. This division is shown in Figure 3.3.

We studied the *social context* during the weekly meetings with the instructor representing the application stakeholders, who also impersonated the different user roles. At first these sessions were conducted in a more unstructured way, to understand the needs and expectations of different users. In later iterations to identify the use cases we opted for a more structured approach. The *knowledge context* was instead investigated at first with a broader approach during the background research for this theses, through a review of existing literature and technical solutions. Later we narrowed down the *knowledge context* to the JSAV library. The results of this phase are presented in Section 4.1.

3.3 Artifact Design

In the *artifact design* phase we designed and created the prototype for the Player Application. From the methodological point of view this phase can be divided into four areas: (i) *functional requirements*, (ii) *qualitative requirements*, (iii) *architecture*, and (iv) *code production*. These areas with

Table 3.1. Methodological choices in the *artifact design* phase.

Artifact Design Areas	Methodology
Functional requirements	Semi-structured interviews
Qualitative requirements	SQuaRE [19]
Architecture	Service oriented architecture paradigm
Code production	Test driven development

the respective methodological choices are exposed in Table 3.1.

The *functional requirements* were defined during the weekly sessions with the instructor representing the application stakeholders and impersonating the different user roles. The *qualitative requirements* instead were defined by applying the qualities exposed in the Systems and software Quality Requirements and Evaluation (SQuaRE) standard [19]. For practical reasons we describe these qualities more in depth in Section 3.4.

The design of the prototype *architecture* was based on the *service oriented architecture* paradigm, whereby the PA would work as an independent subsystem of a Learning Management System [28]. The core idea of this paradigm is that new services can be used to extend a system while still leaving these services fully independent. Following this approach the services can be created and run without restrictions on the technologies or environment used to implement them. This type of architectural choice was followed not only in the relation between the PA and the LMS, but also within the PA itself. In fact the main components of the PA were designed in order to work fully independently from each other.

In the *code production* of the prototype we followed a *test driven development* approach. We used an incremental process where we first wrote the minimum test necessary to have the existing code failing, and then we wrote the minimum code necessary to pass the test. This way incrementally we developed the necessary features.

3.4 Design Validation

The validation was carried out incrementally over the bi-weekly iterations. To establish an evaluation strategy to be used in the validation phase, we relied on the Framework for Evaluation in Design Science Research [53]. The framework presents a four-step process to guide the researcher through the choice and design of the evaluation strategy. These four steps are *(i)* determine the goals of the evaluation, *(ii)* choose the eval-

uation strategy or strategies, *(iii)* determine the properties to be evaluated, and *(iv)* design the individual evaluation episodes. Next we discuss the first three steps of the process, and we will present the fourth step, the design of the single evaluation episodes, in Chapter 3.4.

Goals of evaluation

The evaluation of the Player Application prototype was carried out in the validation phase of the *design cycle*, as shown in Figure 3.1. Its main goals were to establish the prototype's *efficacy* and *effectiveness*, and reduce uncertainty and risks. Projected to our research, *efficacy* is the extent to which the application prototype produces the desired results when the tests written by the developer are run, while *effectiveness* is the extent to which the prototype produces positive results when tested in its real context by real users.

Evaluation strategies

According to the Framework for Evaluation in Design Science Research, in order to choose the proper evaluation strategy, it is necessary to evaluate and prioritize the design risks [53]. If these risks are more socially and user-oriented, then a more human and effectiveness oriented evaluation approach is better. If instead the risks are more related to whether the used technology will work, then a technical and efficacy-oriented strategy is advisable. Technical and human risks are anyway likely to occur in different phases of the *design cycle*, with the technical ones being detectable already in the design-development phase, while the human ones will be more common when the application will be tested by users. For this reason in the earlier iterations of the *design cycle* the evaluation to validate the prototype was based mostly on unit and integration tests run on the existing code. Instead in later iterations we moved progressively to a more human-oriented approach, whereby the product of the design phase was tested not only by the developer, but also during the bi-weekly meetings by the instructor representing the application stakeholders. Finally we held two separate testing sessions during which the prototype was repeatedly tested by the instructor impersonating each time different user roles.

Properties to Be Evaluated

The SQuaRE standard [19] presents a series of high level qualities, which can be used to define software *qualitative requirements* and for their evaluation. In our thesis we used these qualities to define the Player Application *qualitative requirements*, and for the *design validation*. In this section we present these qualities as they are exposed in the SQuaRE standard, and in Section 4.2.1 we will present how we used them to define the *qualitative requirements* specific for the Player Application.

Functional suitability
The *functional suitability* is concerned with the degree to which the software meets users needs. This property is divided into: (i) *functional completeness*, (ii) *functional correctness*, and (iii) *functional appropriateness*.

Performance efficiency
The *performance efficiency* is about the amount of resources used by the software during use under the defined conditions. This property is divided into: (i) *time behaviour*, (ii) *resource utilization*, and (iii) *capacity*.

Compatibility
The *compatibility* property regards how well the software is capable of sharing information with other systems, and working in a shared environment. This property is divided into: (i) *co-existence*, (ii) *interoperability*.

Usability
The *usability* describes the degree to which the software can be used to reach the specified goals with efficacy, efficiency, and satisfaction. This property is divided into: (i) *appropriateness recognizability*, (ii) *learnability*, (iii) *operability*, (iv) *user error protection*, (v) *user interface aesthetics*, and (vi) *accessibility*.

Reliability
The *reliability* is the level to which the software can perform certain functions under certain conditions for a certain period of time. This property is divided into: (i) *maturity*, (ii) *availability*, (iii) *fault tolerance*, and (iv) *recoverability*.

Security

The *security* property defines how well the software protects data and information, while maintaining the appropriate level of access for the users and other systems. This property is divided into: (i) *confidentiality*, (ii) *integrity*, (iii) *non-repudiation*, (iv) *accountability*, and (v) *authenticity*.

Maintainability

The *maintainability* is about the level of effectiveness and efficiency with which the software can be updated and modified by developers. This property is divided into: (i) *modularity*, (ii) *reusability*, (iii) *analyzability*, (iv) *modifiability*, and (v) *testability*.

Portability

The *portability* describes the level of effectiveness and efficiency, with which the software can be transferred to another hardware or environment. This property is divided into: (i) *adaptability*, (ii) *installability*, and (iii) *replaceability*.

4. Results

4.1 Problem Investigation

In this section we present the outcomes of the first phase of the *design cycle*, the *problem investigation*. In this phase we looked at the context where the Player Application is going to be applied. As already described in Section 3.2 and as shown in Figure 3.3, the context can be divided into *social context* and *knowledge context*. We analyzed the *social context* by looking at the user roles and use cases, and the results are exposed in Section 4.1.1 and Section 4.1.2. The analysis of the *knowledge context* was instead conducted by performing an in depth analysis of the JSAV library, and the results are presented in Section 4.1.3. In a broader sense, the *knowledge context* was already covered when we carried out the literature review necessary to retrieve the background information presented in Chapter 2, so at this stage we focused more on the narrow context in which the PA is going to be employed.

4.1.1 Roles

To better understand users needs and expectations and have different point of views, we utilized user roles. The roles were defined during the bi-weekly meetings with the Aalto university instructor representing the application stakeholders. The roles we obtained are: learner, instructor, application developer, exercise developer, and researcher.

Learner
The learners are the persons completing the Visual Algorithm Simulation exercises. They do not have knowledge of how the exercise has been developed, and how the submission is recorder, stored, and replayed. The main

interest of the learner is to complete the exercise successfully, and have the possibility to later review the submission. Reviewing past submissions could be used as a way to enhance learning, understand own mistakes, share the created animation, receive feedback, or clear a dispute over the grading of the exercise.

Instructor

The instructor is the one giving the ADS course and employing the ready VAS exercises. The instructor is interested in being able to review learners submissions from VAS exercises, in order to evaluate their understanding, the exercise difficulty, and to give feedback. For the instructor it is also important that the Player Application creates a registry of all the submissions, which can be used in case of disputes over grading results. Furthermore, the instructor can use the application to create algorithm animations which can then be embedded in lecture material, or shared as a link.

Application Developer

The application developer is the one responsible for developing, maintaining and updating the application itself. The developer will use all the application features for testing purposes, and eventually will also produce animations to share test results and create documentation.

Exercise Developer

The exercise developer is the person developing new JSAV exercises or updating existing ones. He will want to be able to integrate the Player Application into JSAV exercises with as little effort as possible, without making the integration too invasive within the exercise logic. The main use of the application will be to test the new or updated exercises, and eventually produce animations in order to share test results, present new exercises, or create documentation.

Researcher

The researcher's main interest is in the data collected from the VAS exercises. For the researcher it is important that the data is retrievable from the database through an API, that it is well structured, understandable, and contains meaningful information. The researcher will also use the Player Application front end to browse through submissions, visually analyze the data, and create material to be used in presentations or publications.

Table 4.1. Use cases from roles point of view. Listed according to their importance for users.

Use Cases	Learner	Instructor	Application Developer	Exercise Developer	Researcher
Replay VAS as animation	- Review submitted solutions. - Use as learning aid to recall concepts. - Compare different solutions. - Receive feedback. - Present own animations.	- Use as teaching aid to show and explain solutions, and guide through solution steps. - Use as feedback tool.	- Aid in developing, maintaining and testing the Player application.	- Aid to develop and test VAS exercises.	- Browse through solutions in the database, select and visualize them.
Create animations	- Create own animations.	- Create and update teaching material.	- Create documentation.	- Create documentation.	- Create documentation for publications.
Embed animation in external HTML	- Create and present own material	- Share, create and merge teaching material.	- Embed into documentation.	- Embed into documentation.	- Embed into publications.
Save and load animation data locally	- Save data for later use. - Load previously saved animations. - Share the animation. - Use the as base to create new animations.	- Save data for later use. - Share the animation. - Use the as base to create new animations. - Create and test teaching material.	- Save, modify and reload the file to test the application under unexpected conditions.	- Aid to develop and debug VAS exercises.	*N/A*
Share animation via link	- Present own animations.	- Share teaching material.	*N/A*	- Share exercise testing results	*N/A*
Access submissions data in DB through API	*N/A*	- Collecting data for learning analytics	- Collecting data for application development	*N/A*	- Collecting data for research purposes.
Integrate Player Application into old exercises	*N/A*	*N/A*	- Needed for migrations	- Needed for migrations	*N/A*

4.1.2 Use cases

After obtaining the user roles, we organized two extra sessions together with the Aalto University instructor representing the application stakeholders, where we defined the use cases for the PA. In these sessions we utilized the user roles described in the previous section, to look at each use case from every user role's point of view. In this process we also defined the purposes for each role in using the application in each specific case. The outcome of this process is synthesized in Table 4.1. As can be seen from the cells marked with *N/A* in the table, not all roles are involved in all use cases. For example, somebody acting in the learner role will not be interested in accessing the submission data through the database, and it probably would not even be allowed to do it. Next we present the use cases for the Player Application, which are also exposed with more details in Table 4.1.

Replay VAS exercises as animation

The application is used to replay the Visual Algorithm Simulation exercises submissions as animations, either a slide show or an automatic animation. The played animation can either be from the exercise just submitted, in which case it can be automatically presented after submission, or it can be one of the submissions in the database. The person submitting the VAS exercise can be from any of the possible user roles, and therefore the

purpose of replaying the VAS exercise as animation can vary.

Create animations
The specificity of this use case, is that the user is submitting the VAS exercise not with the intent of solving the exercise, but in order to create an Algorithm Animation. Here the final aim is the animation itself, while in the previous cases the animation was a "side product" of carrying out the VAS exercise.

Embed an animation in external HTML
After obtaining an animation, the user might want to be able to embed it in an external HTML document. The purposes of such use vary depending on the user role, but for all it is essential that such a feature exists.

Save and load animation data locally
In this use case the user is saving into a local file the data which the application used to create the animation. The data can also be loaded from a local file into the application, and visualized as an animation. This case is especially relevant for learners and instructors, who can in this way save and load the animations locally, without the need to rely on the database access. The files can then be loaded when needed into the PA, or shared with other users. From developers point of view, this use case is relevant for testing purposes. An exercise developer could use the saved local file to debug an exercise, while the application developer could use it to debug the Player Application, or to create files with unexpected data to be loaded back into the application for testing purposes.

Share animation via link
Also this use case is concerned with sharing the animation produced by the PA. The sharing should happen via a provided link.

Access the submission data in the DB through API
For this use case the user is only interested in the raw data recorder by the Player Application, not in the animation itself. Furthermore, the focus is on being able to easily retrieve large quantity of data from the database through an API. The technical implementation is responsibility of the system storing and managing the data. The PA only provides the data.

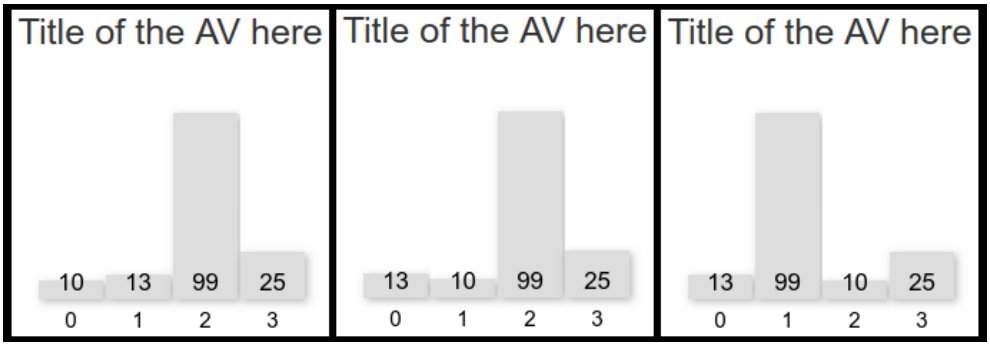


Figure 4.1. JSAV visualization with two swap steps called with the *setTimeout* function. In the leftmost picture is the visualization before the first swap, in the center is the visualization after the first swap, and on the right after the second swap. This visualizations are the result of executing the code presented in Listing 4.1.

Integrate Player Application into old exercises

This use case is specific for VAS exercise and Player Application developers. For them it is important that the PA can easily be integrated into already existing JSAV Visual Algorithm Simulation exercises in cases where they have to be reused and their submissions recorded.

4.1.3 JavaScript Algorithm Visualization Library

This section exposes the outcomes of the analysis we conducted of the JSAV library. In the analysis we focused on five main aspects: *(i)* how to create visualizations *(ii)* how to create slideshows, *(iii)* the data structure APIs, *(iv)* how to create Visual Algorithm Simulation exercises, and *(v)* JSAV events. Each of these five aspects are going to be presented separately.

```
1  var avOptions = {
2    title: 'Title of the AV here',
3  };
4  // Create JSAV visualization object
5  var av = new JSAV($("#jsavcontainer"), avOptions);
6  // Create the data structure
7  var arr = av.ds.array([10, 13, 99, 25], {indexed: true, layout: "bar"});
8  // This the initial state of the visualization
9  av.displayInit();
10 // The animation
11 setTimeout( () => arr.swap(0,1), 1000);
12 setTimeout( () => arr.swap(1,2), 2000);
```

Listing 4.1. Example of JSAV visualization.

Visualizations

To root element of JSAV is the visualization, which is created by a call to the JSAV constructor. Through the visualization object it is then possible to create data structures, UI elements, animations, exercises and event listeners. An example of how to create a JSAV visualization is shown in Listing 4.1 on line 5. An animation of swapping array elements can be for example realized by *(i)* creating an array, *(ii)* defining the state where the

visualization starts, and *(iii)* calling the swap method with a timeout. The procedure is shown in Listing 4.1 and the resulting visualizations created by JSAV are shown in Figure 4.1.

```
1  var avOptions = {
2    title: 'Title of the slideshow here',
3  };
4  // Create JSAV visualization object
5  var av = new JSAV($("#jsavcontainer"), avOptions);
6  // Create the data structure
7  var arr = av.ds.array([10, 13, 99, 25], {indexed: true, layout: "bar"});
8  // This the initial state
9  av.displayInit();
10 // Define the steps
11 av.step();
12 arr.swap(0,1);
13 av.step();
14 arr.swap(1,2);
15 // Start the slideshow after the definition
16 av.recorded();
```

Listing 4.2. Example of JSAV slideshow.

Slideshows

JSAV slideshows can be used to present data structures and any successive changes applied on them by the action of a given algorithm. To create a slideshow we need *(i)* a JSAV object, *(ii)* data structures and UI elements, *(iii)* an initial state, *(iv)* one or more steps, *(v)* actions that take place in each step, and *(vi)* a call to start the slideshow. Listing 4.2 is an example of code for a slideshow with two swap steps. Figure 4.2 shows the slideshow created from that same code.

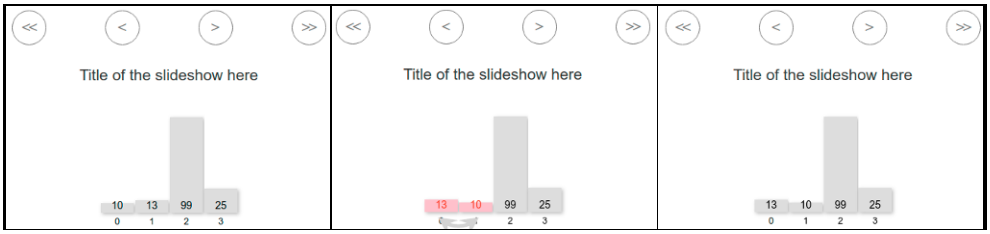


Figure 4.2. JSAV slideshow with a swaps step, resulting from the code presented in Listing 4.2.

Data Structures

JSAV natively provides support for the following data structures: *(i)* array, *(ii)* graph, *(iii)* linked list, *(iv)* matrix, *(v)* tree and binary tree. Each data structure type can be created using a specific method on a JSAV instance, which also takes an *options* object, like shown in Listing 4.2 on line 7. Each data structure has methods that can be used to interact with it. Descriptions of the native data structure objects can be found in Appendix A.1.

The data structures included in an exercise can be accessed through

the *initialStructures* element in the exercise object. This is shown in Appendix A.1.11, which represents the structure of a JSAV VAS exercise object.

On top of the native data structures, new ones can be created by extending the JSAV library [25]. For example in the OpenDSA projects [14] the JSAV native data structures have been extended with new one like *stack*, *array tree*, *circular queue*, *doubly linked list*, *binary heap*, and *red-black tree*. Analyzing these new data structures is not however in the scope of this theses.

```
var initialArray;
var barArray;

function initialize() {
// initialize click handler
if (typeof clickHandler === "undefined") {
  clickHandler = new ClickHandler(av, exercise, {
    selectedClass: "selected",
    effect: "swap"
  });
}
clickHandler.reset();

// initialize the array
initialArray = [];
for (var i = 0; i < arraySize; i++) {
  initialArray[i] = Math.floor(Math.random() * 100) + 10;
}
if (barArray) {
  clickHandler.remove(barArray);
  barArray.clear();
}
barArray = av.ds.array(initialArray, {indexed: true, layout: "bar"});
barArray.layout();
clickHandler.addArray(barArray);

return barArray;
}
```

Listing 4.3. JSAV initialization/rest function for a Visual Algorithm Simulation exercise.

Visual Algorithm Simulation Exercises

In a Visual Algorithm Simulation exercise the user can directly manipulate the data structure and modify the visualization using the UI provided by JSAV. This type of exercise requires a function describing the model answer, and a function to initialize and reset the exercise. The initialization/reset function has to return the data structures necessary for grading the exercise, and it can be also used to set the necessary click handlers. Listing 4.3 is an example of initialization/reset function. The model answer function also has to return the data structures used for grading, and it has to simulate the correct changes occurring in them, like shown in Listing 4.4. Finally the exercise is initialized through the JSAV object instance, passing the functions and options, like in Listing 4.5. Figure 4.3 shows a

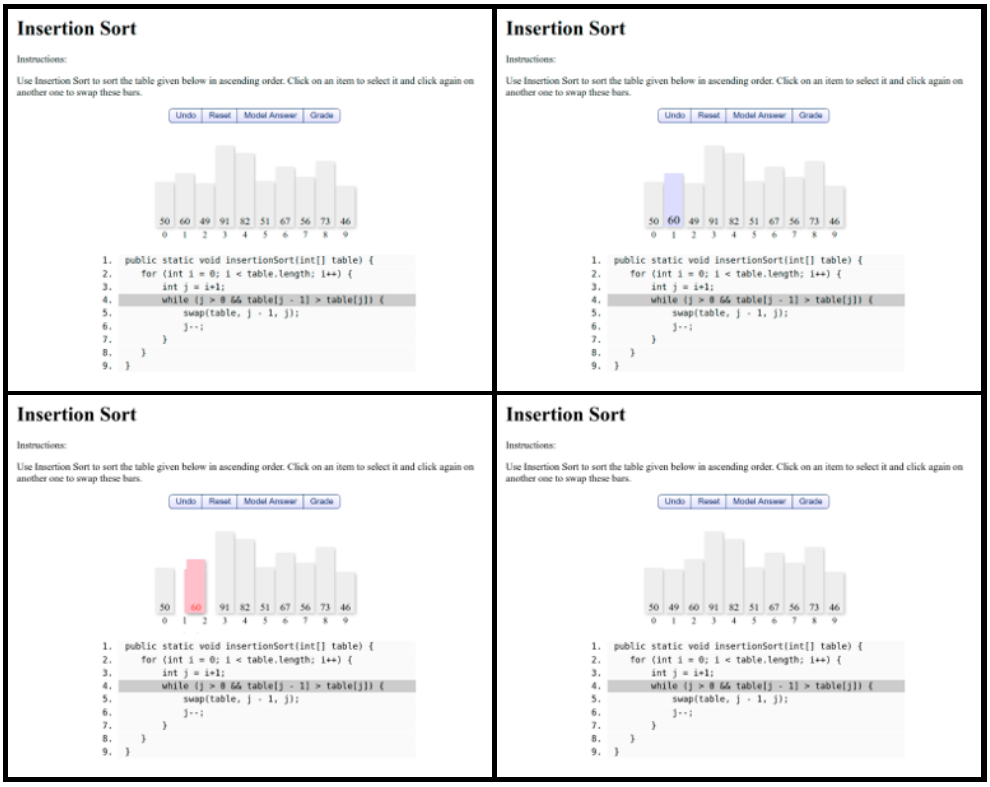


Figure 4.3. JSAV Visual Algorithm Simulation exercise. The user clicks on the array index 1 and then on index 2, which causes the values to swap.

JSAV VAS exercise where the learner has to simulate the insertion sort algorithm. The exercise shows the instructions, the data structure as a bar array, and the code sample. The learner has to click on the array bars to cause the values to swap. The images show the learner first clicking on the second array bar and then on the third array bar, which causes the two values to swap.

```
function modelSolution(jsav) {
    var modelArray = jsav.ds.array(initialArray, {indexed: true, layout: "bar"});

    jsav._undo = [];

    for (var i = 1; i < arraySize; i++) {
      var j = i;
      while (j > 0 && modelArray.value(j - 1) > modelArray.value(j)) {
        jsav.umsg('Shift "' + modelArray.value(j) + '" to the left.<br/> &
     nbsp;i: ' + i + '<br/>  j: ' + j);
        modelArray.swap(j, j - 1);
        jsav.stepOption("grade", true);
        jsav.step();
        j--;
      }
    }

    return modelArray;
}
```

Listing 4.4. JSAV model answer function for an exercise.

```
var avOptions = {
title: 'Title of the AV here',
};
// Create JSAV visualization object
var av = new JSAV($("#jsavcontainer"), avOptions);
var initialArray;
var barArray;

function initialize() {
    ...
}

function modelSolution(jsav) {
    ...
}

var exercise = av.exercise(modelSolution, initialize, {feedback: "atend"});
exercise.reset();
```

Listing 4.5. JSAV exercise initialization.

JSAV events

The JSAV library emits many types of log events arising from the user interaction. Some are triggered automatically while others are triggered if an event listener is set for that type of event on the data structure. For example the *jsav-init* log event is triggered automatically when a JSAV instance is initialized, while *jsav-array-click* is triggered upon clicking an array, but only if a click listener has been set on it. Table 4.2 shows the log events divided into categories according to how they are triggered, if by the visualization object, the exercise or by a data structure.

```
(function ($) {
  "use strict";
  var avOptions = {
    title: 'Title of the AV here',
    logEvent: ((eventData) => console.log('Event handler', eventData)),
  };
  // Create JSAV object
  var av = new JSAV($("#jsavcontainer"), avOptions);
  var arr = av.ds.array([10, 13, 99, 25], {indexed: true, layout: "bar"});
  // Attach click listener
  arr.click((index) => console.log('Click handler. Array index:', index));
}(jQuery));
```

Listing 4.6. Example of using JSAV events.

Each event creates an object containing the relevant data. All triggered events and their data can be caught by defining the *logEvent* option when initializing the JSAV instance. In fact we can pass a function to the *logEvent* option, and that function will be triggered every time a *jsav-log-event* happens. An example of the procedure is presented in Listing 4.6. Another option to catch all JSAV log events is to set an event listener on the *document* object, which listens for events of type *jsav-log-event*.

Each log event is an object containing the relevant data. By catching the right log events, we can therefore also reach the data relevant for our purposes. A more detailed list of JSAV log event objects can be found in

Table 4.2. Types of *jsav-log-event* triggered by JSAV.

Visualization	Exercise	Data Structures
jsav-backward	jsav-exercise-init	jsav-array-<Event Type>
jsav-begin	jsav-exercise-gradeable-step	jsav-edge-<Event Type>
jsav-end	jsav-exercise-grade	jsav-graphical-<Event Type>
jsav-forward	jsav-exercise-grade-change	jsav-matrix-<Event Type>
jsav-init	jsav-exercise-model-backward	**Event Types**
jsav-message	jsav-exercise-model-begin	click
jsav-object-move	jsav-exercise-model-close	dbclick
jsav-recorded	jsav-exercise-model-end	mousedown
jsav-speed	jsav-exercise-model-forward	mousemove
jsav-speed-change	jsav-exercise-model-init	mouseup
jsav-updaterelative	jsav-exercise-model-open	mouseenter
jsav-updatecounter	jsav-exercise-model-recorded	mouseleave
	jsav-exercise-reset	
	jsav-exercise-step-fixed	
	jsav-exercise-step-undone	
	jsav-exercise-undo	

Appendix A.1.12.

4.2 Application Design

In this chapter we present the results from the second phase of the *design cycle*: the *artifact design*. We are going to look at the solutions we have designed to reach our goal of having a web application prototype which can be used in conjunction with JSAV Visual Algorithm Simulation exercises, to collect, store, and replay user interaction data in a standardized way. We will start by presenting the application requirements, after which we go more into details with the proposed solution which includes: a new JSON-based Algorithm Animation Language (JAAL) and the Player Application prototype.

4.2.1 Requirements

In this section we present the *functional* and *qualitative requirements* for the Player Application. The requirements are defined so that the software's inherent properties will be developed to meet users needs. These properties can be classified into functional and quality properties [19]. The functional properties are domain-specific, and they determine what the software can do. The quality properties instead define how well that software performs. Since the functional properties are domain specific, we defined them at

Table 4.3. Functional requirements for the PA from the user point of view. Each requirement is considered according to the responsibilities of the PA main components and listed in order of importance.

Functional Requirements	**Player Application Exercise Recorder**	**Player Application Exercise Player**
All the actions that can be performed on a data structure are recorded and replayed in the animation.	- Records all actions from a JSAV VAS exercise. - Records the changes happening to the data structures due to an action.	- Shows all the actions taken in the JSAV VAS exercise. - Shows the changes to the data structure caused by each action.
The animation can be viewed step-by-step as a slideshow.	- Each action is saved as own step.	- Functionality to step through the animation one step at time.
The animation can be viewed using play/pause/stop buttons.	- Each action is saved as own step.	- Functionality to start and stop the animation.
All relevant information shown in the exercise is shown also in the animation.	- Save the relevant information to JAAL object.	- Show the relevant information from JAAL object.
Clicking the exercise control buttons is saved and shown in the animation.	- Record when the user clicks a control button. - Record the changes happening to the data structures when the button is clicked.	- Show the button click. - Show the changes to the data structure caused by the button click.
If the model answer is opened by the user, this is also shown in the animation.	- Record when the user opens the model answer. - Record the steps of the model answers viewed by the user.	- Show the opening of the model answer by the user. - Shows the steps viewed by the user.
The model answer is shown together with the animation, so that the user can compare it to the own solution.	- Record all the model answer steps.	- Show the model answer on side of the animation. - Functionality to step through the model answer.
It is possible to share animations.	*N/A*	- Offer a link that can be shared.
It is possible to export & import animations.	*N/A*	- Save animation data locally. - Load animation from local file.

first during the bi-weekly meetings with the instructor representing the application stakeholders. After testing the application prototype, the *functional requirements* were updated to better cover all the required functionalities. For the formulation of the *qualitative requirements* we used the properties from the SQuaRE standard [19] as reference. These properties were already presented in Section 3.4, and in this section we are going to present them again, this time in relation to the Player Application *qualitative requirements*.

Functional Requirements

In this section we present the Player Application *functional requirements* obtained from the meetings with the Aalto University instructor representing the domain experts and stakeholders. These requirements were first defined in January 2020, when the PA prototype design was not yet mature, and then updated in March and April 2020 after the testing sessions with the prototype. The *functional requirements* contain concepts like *Exercise Recorder* ER, *Exercise Player* EP, and JAAL which will be presented later

Table 4.4. Functional requirements for the PA from the point of view of the JAAL object. Each requirements is considered according to responsibilities of the PA ER and the LMS, and listed in order of importance.

Functional Requirements	**Player Application Exercise Recorder**	**Learning Management System**
All relevant GUI actions saved in JAAL.	- Listen for relevant actions. - Format the data to JAAL format. - Save the formatted data to JAAL object.	*N/A*
All relevant exercise data is recorded in JAAL.	- Save relevant exercise data. - Format the data to JAAL format. - Save the formatted data to JAAL object.	*N/A*
All saved submissions are available in the JAAL *format.*	- Send the JAAL object to the LMS.	- Save the received JAAL object to database.
The application can be connected to a LMS.	- Implements the protocol required by the LMS.	- Offer a protocol for connecting the PA.
The JAAL *object is sent to the LMS when the user clicks the grade and reset buttons, and reloads or closes the window.*	- Follow necessary events. - Send JAAL object to LMS upon events happening.	*N/A*
The model answer is saved in JAAL.	- Retrieve the model answer steps. - Format model answer steps to JAAL. - Save the formatted data to JAAL object.	*N/A*
The JAAL *object contains application version.*	- Implement versioning system. - Save version to JAAL object.	*N/A*
JAAL *objects are indexed.*	*N/A*	- Implement and manage indexing in database.
Saved JAAL *objects are accessible through API.*	*N/A*	- Implement API.

in this Chapter. These requirements are shown in Table 4.3 and in Table 4.4, which contain respectively the *functional requirements* from the user perspective, and from the JAAL data object perspective. In Table 4.4 we consider the respective responsibilities of the PA main components, and in Table 4.4 the responsibilities of the PA and the LMS. Even tough in Table 4.4 we show also the requirements that are fully under responsibility of the Learning Management System, we are not going to discuss them any further, since they are not in the scope of this thesis project. Next we will describe each *functional requirement* into more details.

All the actions that can be performed on a data structure are recorded and replayed in the animation

In a JSAV VAS exercise the user can use the UI to perform actions on the data structures, like swapping, copying, adding or deleting values. The PA must be able to record all these different actions and then replay them in the animation.

The animation can be viewed both step-by-step as a slideshow and using play/pause/stop buttons

The animation showing the steps from the submitted solution will be presented to the user after completing the exercise. The PA must offer the possibility to both control the animation using play, pause and stop buttons, and also as a slide show, with buttons to step through the animation one step at the time.

All relevant information shown in the exercise is shown also in the animation

Apart from the data structures, JASV presents the user with relevant information throughout the exercise, such as instructions, code samples and score. This type of information must be saved by the PA and included in the animation.

Clicking the exercise control buttons is saved and shown in the animation

The JSAV VAS exercises have four control buttons: undo, reset, model answer, and grade. Clicks on these control buttons and their effects on the data structures must be recorded by the PA and shown in the animation.

If the model answer is opened by the user, this is also shown in the animation

Clicking the model answer button in the VAS exercise will open a window containing the correct solution steps. The content of the model answer window and the steps viewed by the user must be saved and shown in the animation.

The model answer is shown together with the animation, so that the user can compare it to the own solution

The full model answer of the exercise must be recorded and shown at the same time when showing the animation, so that the user can compare the submitted solution steps to the model answer steps. The full model answer must be saved in all cases, even if the user did not click the model answer button.

It is possible to share animations

The PA must offer a way to share the animation via link, so that the link can be shared or embedded into any HTML document.

It is possible to export and import animations
The PA must include a functionality to export the animation to a file in the JAAL format, and another functionality to import an animation from a file containing data in the JAAL format.

All relevant GUI actions are saved in JAAL
The PA must record all relevant GUI actions during the VAS exercise, convert them to a format compatible with JAAL, and save them in the JAAL object.

All relevant exercise data is recorded in JAAL
Apart from GUI actions, all relevant exercise data, such as title, instructions, and score, must be formatted correctly and saved in the JAAL object.

All saved submissions are available in the JAAL *format*
All exercise submissions must be later available as JAAL objects. The PA is responsible for creating the JAAL object and sending it to the LMS, who is in turn responsible for saving it in a database.

The application can be connected to a LMS
It is important that the PA can be integrated in a LMS. For this reason there must be and agreement on the protocol to be used for exchanging the necessary data. The LMS is responsible for specifying the protocol, and the PA on its part must implement it.

The JAAL *object is sent to the LMS when the user clicks the grade and reset buttons, and reloads or closes the window*
The PA application must send the JAAL object to the LMS in three cases: *(i)* the user clicks the *grade* button, *(ii)* the user clicks the *reset* button, and *(iii)* the user closes the exercise window.

The model answer is saved in JAAL
The exercise model answer, its steps, and the state of the data structures in each step must be saved in the JAAL object.

The JAAL *object contains the application version*
The VAS exercise submission data must include the version of the PA that created the JAAL object.

Table 4.5. Qualitative properties from SQuaRE [19] with corresponding qualitative requirements for the Player Application.

Qualitative properties	**Qualitative Requirements**
	Functional Suitability
Functional completeness	The functionalities specified by the requirements are present.
Functional correctness	The functionalities specified by the requirements work with a satisfactory degree of precision.
Functional appropriateness	The functionalities specified by the requirements are presented in an appropriate and understandable way, avoiding unnecessary steps.
	Performance Efficiency
Time behaviour	The recording of the VAS exercise does not slow down the execution of the exercise steps. The creation of the animation does not require noticeable loading time.
Resource utilization	The resources used by the application have to be reasonable in relation to its functionalities.
Capacity	The amount of sessions is not limited. For each exercise session is possible to record one exercise. For each animation session is possible to play one animation at time.
	Compatibility
Co-existence	The application works in conjunction with JSAV and its required libraries. The application works when embedded in the Learning Management System.
Interoperability	The application is able to exchange the required information with the LMS
	Usability
Appropriateness recognizability	The application users find it useful when used according to the given use cases.
Learnability	It is easy to learn how to use the application.
Operability	The application is easy to use and control.
User error protection	The application protects the user from making errors when recording the exercise or playing the animation.
User interface aesthetics	The user interface enables pleasing and satisfying interaction for the user.
Accessibility	The application is usable also by people with diverse abilities.
	Reliability
Maturity	The application is reliable under normal conditions.
Availability	The application is available and accessible when required for use.
Fault tolerance	The application works as intended even when an error occurs.
Recoverability	The application recovers in case of failure or interruption without losing the data.
	Security
Confidentiality	The data is accessible only to those who are authorized to have access.
Integrity	The application prevents unauthorized access and modification of data.
Non-repudiation	The application records relevant user interaction.
Accountability	The recorded user interaction is traced to the user.
Authenticity	The identity of the user whose interaction has been recorded can be proven.
	Maintainability
Modularity	The application is designed of modules, such that change to one of them has minimal impact to other modules.
Reusability	The modules composing the application can be easily reused in other software.
Analysability	The application is designed and written in a way, that analyzing it is not too difficult. For example to asses the impact of some changes, or diagnose the cause of a failure.
Modifiability	The application can be effectively and efficiently modified without degrading its quality.
Testability	Test criteria are established for the application, and tests can be performed to verify if the criteria are met.
	Portability
Adaptability	The application can be adapted for use on different environments. For example different LMSs.
Installability	The application can be deployed in the needed environment efficiently and with effectiveness.
Replaceability	The application can easily be replaced with new versions of the same application or other similar applications.

Qualitative Requirements

The procedure followed to define the *qualitative requirements* is pictured in Figure 4.4. We used as a reference the qualitative properties from the SQuaRE standard [19], already presented in Section 3.4. In the process we then analyzed the Player Application with its context, use cases, and *functional requirements*, through the filter of these qualitative properties. As a result we obtained the requirements enforcing the qualitative properties. All the *qualitative requirements* are shown and described in Table 4.5.

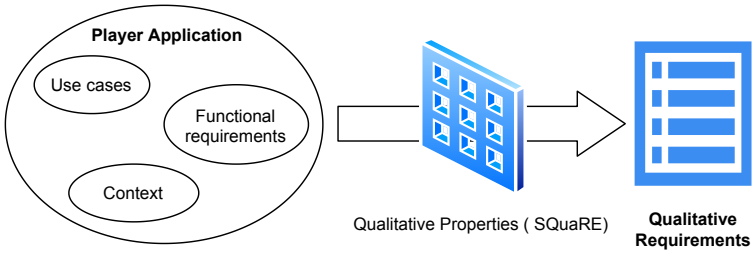


Figure 4.4. The process followed to define the qualitative requirements through the SQuaRE standard [19].

4.2.2 JSON-based Algorithm Animation Language

In this section we present the language we created to describe the interaction of the user with the Visual Algorithm Simulation exercise, and the animation created through it.

The Player Application is meant to primarily be used to record and replay JSAV Visual Algorithm Simulation exercises, but in theory it could crate animations based on data originating from any source, provided that the data respects the specified format. For this reason the language used to describe the animation had to be based on well supported technologies, not only in the context of JSAV, but also in a wider context. The most reasonable choice, when it comes to describing objects and events, is to base that description on the JavaScript Object Notation (JSON). To define our AAL we strongly relied on the structure used in XAAL. Because JSON and XAAL are the main models for our language, we called it JSON-based Algorithm Animation Language (JAAL).

In JAAL the main document is divided in four parts, the same defined in XAAL: *(i) metadata*, *(ii) definitions*, *(iii) initial state* and *(iv) animation*. The *metadata* section can include for example the student identification number and the application version. The *definition* part is used to specify certain options concerning the visualization, styling and the animation in general. The *initial state* section contains the initial data structures

and initial HTML of the exercise. The *animation* section includes the animation steps. The main structure of JAAL is shown in Listing 4.7

```
{
  "metadata": {} ,
  "definitions": {},
  "initialState": {},
  "animation": [],
}
```

Listing 4.7. Main structure of a JAAL.

Metadata

The *metadata* section can be used to include information which is not arising from the Visual Algorithm Simulation exercise itself, but is passed by the Learning Management System or the Player Application. Some examples are *user identification number*, *application version*, *timestamp*, and *keywords*. We did not strictly define any elements, since the information in the *metadata* section is not used by the Player Application but rather by the LMS or anybody utilizing the collected raw data.

Definitions

Currently we defined four types of elements which can be used in the *definitions* part: (i) *options*, (ii) *style*, (iii) *score*, and (iv) *modelSolution*. The *options* and *score* elements are composed of key-value pairs; the *style* element is a Cascading Style Sheets (CSS) definition in JSON format; the *modelSolution* is a string containing the function definition for the model answer. Listing 4.8 show how the *definitions* part can be used.

```
 "definitions": {
   "style": {
     "#jsavcontainer": {
       "height": "100%",
       "width": "100%"
     }
   }
   "score": {
     "total": 19,
     "correct": 0,
     "undo": 0,
     "fix": 0,
     "student": 1
   },
   "options": {
     "title": "Insertion Sort",
     "instructions": "Use Insertion Sort to sort the table given below in
     ascending order. Click on an item to select it and click again on another one
      to swap these bars."
   },
   "modelSolution": "function(e){var t=e.ds.array(a,{indexed:!0,layout:\"bar\"});
    e._undo=[];for(var n=1;n<10;n++)for(var r=n;r>0&&t.value(r-1)>t.value(r);)e.
    umsg('Shift \"'+t.value(r)+'\" to the left.'),t.swap(r,r-1),e.stepOption(\"
    grade\",!0),e.step(),r--;return t}"
 },
```

Listing 4.8. Definitions in JAAL.

Initial State

The data structures can be defined as objects. Each object describing a data structure contains the common elements *type*, *id*, *values* and *options*. The element *type* can be for example "array", "tree" or "list". The *id* will be used to uniquely identify the given data structure within the animation. The *options* element can be used for example to define if the visualization of an array should show the indexes, and to specify the orientation of the array visualization, as shown in Listing 4.9. On top of these common elements more can be specified, depending on the type of data structure. For example a tree or graph will have to specify also the nodes and their relations, like shown in Listing A.16. The *initialState* section can also be used to define the initial HTML data of the exercise. This information can be used by a web application to easily visualize the initial state of the exercise.

```
{
  "metadata": [],
  "definitions": {},
  "initialState":
  {
    dataStructures:
    [
      {
        "type": array,
        "id": "0d15981decd54d55aa6f679eb45e5205",
        "values": [0, 1, 2],
        "options": { "indexed": true, "orientation": "horizontal" }
      }
    ],
    animationDOM: "...."
  },

  "animation": [],
}
```

Listing 4.9. Representation of an array in JAAL.

Animation

The *animation* element is an array containing the animation steps. Each step is an object containing the *type* of action performed, a *timestamp*, additional information like the clicked *index* number, *datastructureId* and *values* of the data structure involved in the action. Tree of the most common action types in JSAV VAS exercises would be "click", "gradeable-step", and "grade". Each step also contains the *animationDOM* element, which defines the inner HTML of the exercise element in the current step. This HTML data can be used by a web application to easily visualize the step. An example of the content for the *animation* element is shown in Listing A.17.

4.2.3 Application Prototype

In this section we describe the outcomes of the design process of the Player Application. First we expose the higher level design, which is connected to our theoretical base. After that we present the lower level technical design, which includes the application architecture. The working prototype can be tested on the web address `https://jsav-player-test-app.firebaseapp.com/`. The code with explanation on how to install an run it can be found from the git repository `https://github.com/MarianiGiacomo/jsav-player-application.git`.

Theoretical Design

The Player Application is at the core a software to create Algorithm Animations, therefore as part of the design process we analyzed it through the categories of the Taxonomy for Software Visualization (TSV) proposed by Price [47]. The aim of this phase was to give the application a strong abstract structure, on which to build the lower level architectural design. Table 4.6 shows how we applied the categories of the TSV to the PA. Next we describe the outcome of the analysis for each category.

Table 4.6. Categories of the Taxonomy of Software Visualization in the Player Application.

Category	How applies to Player Application
Scope	What is the range of exercises that can be handled by the PA?
Content	What subset of information from the exercise does the PA uses to construct the visualization?
Form	What are the parameters and limitations concerning the output of the PA?
Method	How is the visualization specified and how is the data source connected to the visualization?
Interaction	How does the user interact with the PA?
Effectiveness	How well does the PA communicate information to the user

Category - Scope

To determine the scope of the PA we answered the question "*what is the range of exercises that can be handled by the* PA*?*". The core of JSAV Visual Algorithm Simulation exercises is the data structure. Actions such as swapping or deleting values, on a technical level happen through the data structures APIs. For this reason to determine the scope of the Player Application, we have to define which data structures will be supported. JSAV core data structures are: array, tree, binary tree, linked list, and matrix. On top of core data structures, developers can also extend JSAV

and create new ones. As the scope of the Player Application, we define the core data structures of JSAV. Furthermore, we define that new data structures, in order to be eligible for support by the PA, will have to be implemented following the same principles as the core data structures. In specific, new data structures will have to emit the same types on JSAV events upon user interaction.

Category - Content
To determine the content of the application, we answered the question "*what subset of information from the exercise does the* PA *uses to construct the visualization?*". On a higher level, this information can be categorized into three groups: *(i)* exercise configuration settings, *(ii)* user interaction with the GUI, and *(iii)* data structures state. Data from category *(i)* is gathered from JSAV log events emitted upon exercise initialization and submission, and from the exercise object. Data from category *(ii)* is gathered only from those log events that JSAV emits upon interaction of the user with the visualization. Data from category *(iii)* is extracted from the JSAV exercise object. The exercise object structure is shown in Listing A.11. Table 4.7 shows more in details the content data divided into categories, and its source.

Category - Form
This category concerns the output of the Player Application, along with parameters and limitations governing it. The output of the Player Application is an animation, both in the form of user controlled slide show, and automatic animation. These two forms are produced based on the data gathered from the Visual Algorithm Animation exercise, and the subsequent JSON-based Algorithm Animation Language object. Alternatively a JAAL file can also be uploaded straight into the Player Application.

The parameters governing the animation, are those shown in the *Data* column of Table 4.7. On top of those parameters, there is an extra one for choosing the animation speed, which can be set from the Exercise Player part of the Player Application. The data structure visualization options are those present in the JSAV data structure objects in the *option* element. The options may vary according to the type of data structure.

Among the limitations of the Player Application, the two most important to be considered are:

1. The VAS exercise is recorded using the JSAV log events rising from

Table 4.7. Content data for the Player Application and its source, divided into categories.

Data	Source
Exercise object	Event: jsav-exercise-init
Configuration Settings	
Title	Event: jsav-init
Instructions	Event: jsav-init
Data structure visualization options	Exercise objects
Final score	Event: jsav-exercise-grade
User Interaction with the GUI	
Click on data structure	Event: jsav-<DS>-click
Open model answer	Event: jsav-exercise-model-open
Close model answer	Event: jsav-exercise-model-close
Click undo button	Event: jsav-exercise-undo
Click reset button	Event: jsav-exercise-reset
Click grade button	Event: jsav-exercise-grade-button
Idle time	Time between events
Data Structures State	
Indices	Exercise object
Values	Exercise object
Nodes	Exercise object
Edges	Exercise object

the user interaction with the visualization, the exercise, and the data structures. This means that every new data structure type will have to emit the same JSAV log events as the core data structures upon user interaction. On top of this, the support for the new data structures will have to be added to the Player Application.

2. The Player Application is only able to create animations if the input data is in the form specified by the JSON-based Algorithm Animation Language.

Category - Method

In the analysis of this category we answered the question "*how is the visualization specified and how is the data source connected to the visualization?*". The Player Application consists of two main independent processes: recording the VAS exercise, and replaying the exercise as animation. The first process transforms the interaction of the user with the VAS exercise into a JSON-based object, and sends it to the Learning Management System. The second independent process receives the object from the LMS, and

based on that creates the animation.

Category - Interaction
In this category the aim was to answer the question "*how does the user interact with the* PA*?*". As described in the previous category, the Player Application is divided into two major processes. The interaction happens only in the second process when the user replays the animation, since the recording is done automatically while the user interacts with the VAS exercise. The VAS exercise itself is not part of the PA. The user can interact with the animation by using the play, pause, stop, back and forward buttons. The other types of interactions are: selecting animation speed, importing a new animation from file, and exporting the existing animation.

Category - Effectiveness
To determine the effectiveness we had to answer the question "*how well does the* PA *communicate information to the user?*". At this point we could only analyse possible issues affecting the application effectiveness of which we were aware already in the design phase. These issues are:

- Integration with LMS: the Player Application is dependent on the LMS in regards to the saving, storing, and loading of the VAS exercise submissions data. For this reason the integration of the PA with the LMS is extremely important to guarantee effectiveness. Faulty integration can cause loss of effectiveness, even if the Player Application itself records and replays the VAS exercises correctly.

- New data structures: the JSAV library can be extended with new data structures, but these new data structures are not automatically supported by the Player Application. Therefore new data structures cause loss of effectiveness, at least until the support is added in the PA.

- JAAL: the Player Application creates the animations starting from a JAAL file. As long as the JAAL file is created by the PA itself during the recording process, we have control over its content and format. The JAAL file can also be imported from a local device, in which case there might arise problems which we can not yet foresee.

Architecture

On a higher architectural level the Player Application is divided into two main independent components: the Exercise Recorder (ER) and the Exercise Player (EP). As shown in Figure 4.5, both components act within their own process and do not need to be in contact or be in synchronization with the each other. The LMS loads the JSAV exercise, which then loads the ER. The ER records the interaction of the user with the exercise and sends the data to the LMS. In the other process the LMS loads the Exercise Player and passes the URL for fetching the animation data. The EP then fetches the animation data from the given URL and creates the animation.

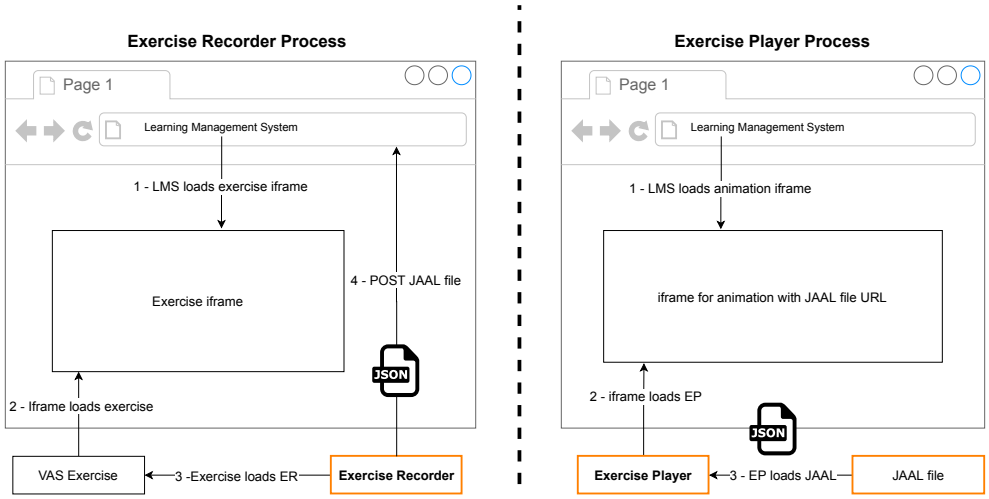


Figure 4.5. The two Player Application components within their own processes.

Exercise Recorder

The Exercise Recorder component is written in JavaScript and created using the Node Package Manager [46] (NPM). The whole component is divided into modules as shown in Figure 4.6. The different modules can be exported into a single bundle file to be imported into static pages using the <*script*> tag like shown in Figue 4.7.

The ER initializes automatically when imported into an HTML document,

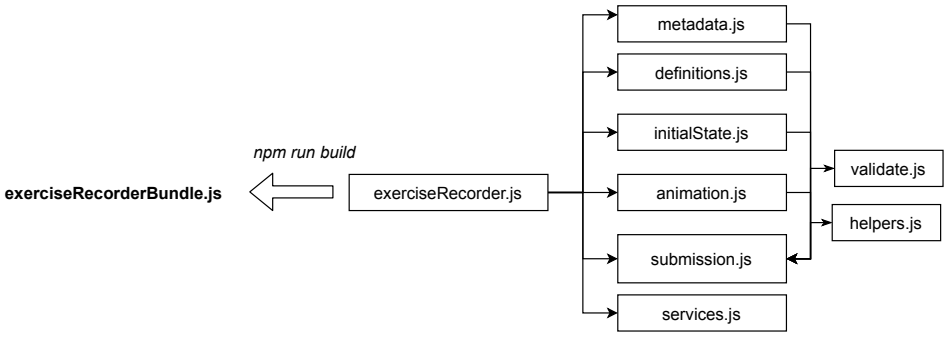


Figure 4.6. The Exercise Recorder component modules.

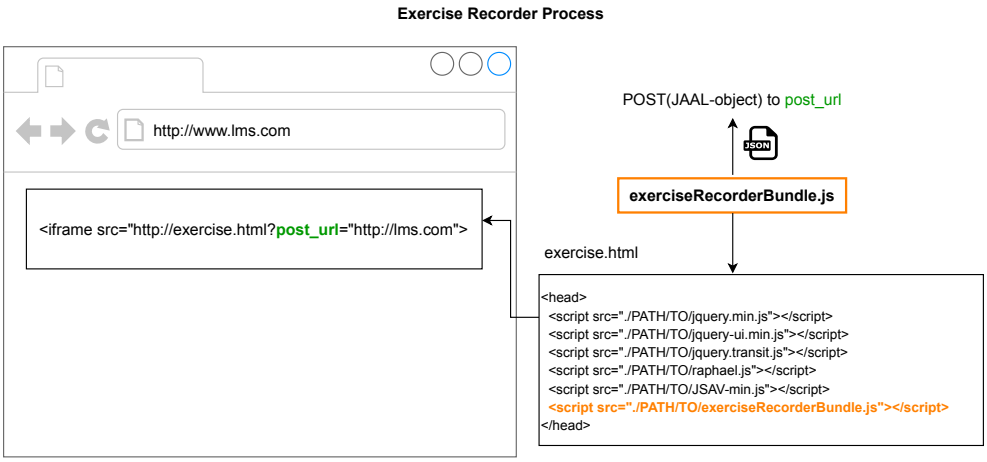


Figure 4.7. The Exercise Recorder bundle integrated into a JSAV exercise.

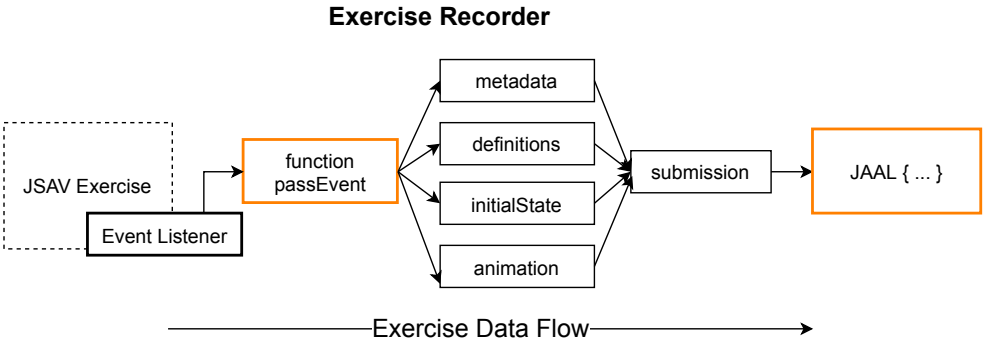


Figure 4.8. Handling and flow of JSAV event data and exercise data in the Exercise Recorder.

and it starts listening for all JSAV log events. For this reason it is important that the ER is imported in the *head* part of the HTML document, before the JSAV exercise is loaded. Upon initialization the ER will look for the *post_url* URL parameter, which should contain the URL where the recorded animation data has to be posted in the form of a JAAL object. This is shown in Figure 4.7. Instead of a URL, the *post_url* can also contain the string "window", in which case the recorded data will be posted to the window where the ER has been loaded.

Listing A.18 shows the content of the *passEvent()* function, which is the most important part of the ER. This function handles all the *jsav-events* and delivers them to the correct ER sub-modules, like shown in Figure 4.8. The most relevant data collected through the JSAV log events concerns the state of the exercise, which is recorded both by saving the data structures states, as well as coping the *innerHTML* of the exercise HTML element. The data is then formatted and stored in the JAAL object.

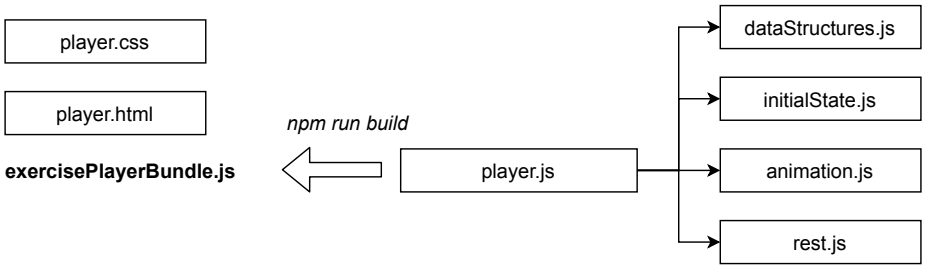


Figure 4.9. Architecture of the Exercise Player component.

Exercise Player

The Exercise Player component is written in JavaScript and created using NPM. The whole component is divided into modules, but it can be exported into a single bundle file, like shown in Figure 4.9. The bundle file has to be imported into the Exercise Player HTML document using a *<script>* tag like shown in Figure 4.10. This HTML document is what the LMS loads in order to use the EP. Figure 4.10 shows how the EP can be integrated into a LMS.

Since the EP bundle file uses certain HTML elements to construct the animation, it is important that it is imported in the EP HTML document after the *body* element. The content of the *player.html* file is shown in Listing A.19.

Figure 4.10. The Exercise Player component integrated into the LMS to replay the animation.

When the EP bundle file is loaded into the HTML document, it automatically looks for the URL parameter named *sumbission*, which should contain the URL to be used by the EP to fetch the JAAL object containing the animation data, like shown in Figure 4.10. For this reason the LMS has to provide this URL when loading the EP.

4.3 Prototype Validation

The *design validation* is the third step in the *design cycle*. In our six months long design process the importance of this step grew incrementally as we progressed through the iterations of the *design cycle*. The first validations of the Player Application prototype were done through unit and integration tests on the code of the prototype in November 2019. Later, in January and February 2020, as the prototype gained structure and usability, some features were tested in the bi-weekly meetings together with the instructor representing the Player Application stakeholders and users. Finally in March 2020 we produced a test application to simulate the use of the prototype in a real Learning Management System environment. After testing the prototype with the test application we evaluated how well it satisfied the requirements.

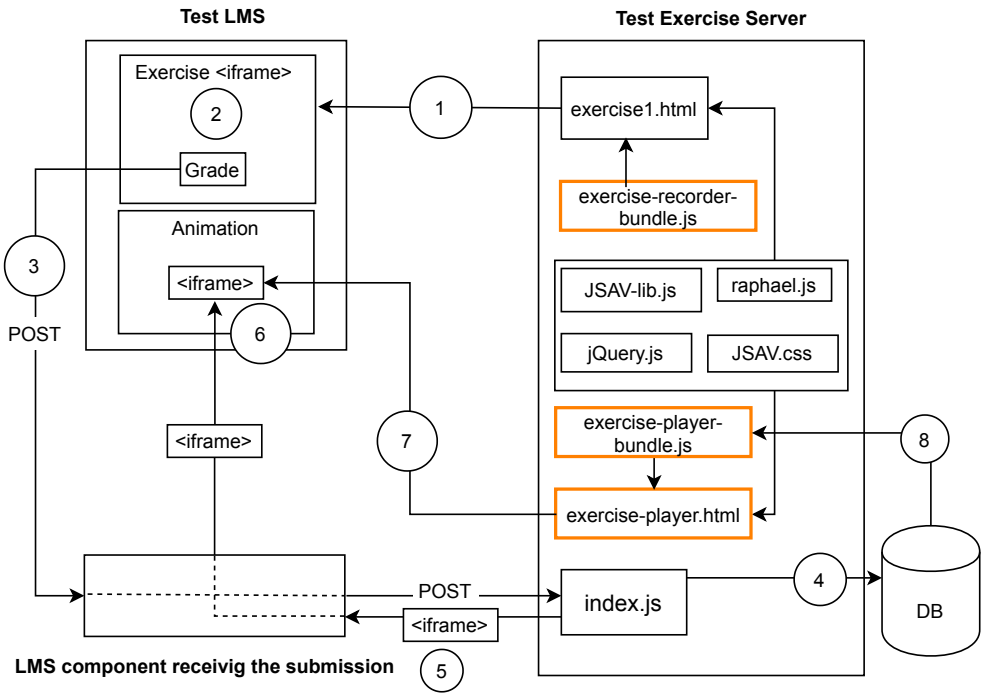
```
describe("jsav-exercise-grade", () => {
  afterEach(() => {
    submission.reset();
    helpers.resetAllData();
  });

  test("Saves grade to submission definitions", () => {
      const exerciseGradeEvent = helpers.exerciseGradeEvent;
      recorder.passEvent(exerciseGradeEvent);
      const expectedScore = { ...exerciseGradeEvent.score };
      const receivedScore = submission.state().definitions.score;
      expect(receivedScore).toMatchObject(expectedScore);
  })
})
```

Listing 4.10. Jest integration test to check that the final grade is saved when the *jsav-exercise-grade* event is detected.

4.3.1 Code Testing

To test the code we used the Jest [20] testing framework. At the beginning we wrote unit tests for single functions, then as the functions started to interact to create functionalities and the application grew, we begun writing integration tests. When implementing a new functionality, we first wrote the test with minimum amount checks to have the code failing it. Then we implemented the functionality with just enough code to pass the tests. We repeated this iterations for each functionality, until the test included all the required checks and the code passed all the test. Figure 4.10 is the example of an integration test to check that the final grade is saved when the *jsav-exercise-grade* event is detected.



1) The iframe in the LMS fetches the exercise from the exercise server with a GET request, passing as query parameter the URL for posting the exercise submission.
2) The LMS loads the exercise into the iframe.
3) When the exercise grade button is clicked, the Exercise Recorder sends a POST request to the URL given by the LMS and the LMS forwards the POST request to the exercise server.
4) The exercise server stores the submission data in the database.
5) The exercise server responds to the POST request by sending an iframe with pre-filled *src* attribute for loading the Exercise Player. The URL in the *src* attribute also contains the query parameter *submission*, which has the URL for loading the animation data.
6) The iframe sent by the exercise server is loaded in the LMS.
7) The loaded iframe fetches the Exercise Player.
8) The Exercise PLayer fetches the animation data from the URL given in the *submission* query parameter and creates the animation.

Figure 4.11. Architecture of the test application.

4.3.2 Test Application

To validate the Player Application prototype we created a test application containing one Visual Algorithm Simulation exercise. With this application we simulated a real Learning Management System environment, and integrated the prototype into it. Figure 4.11 shows the architecture of the test application and how the prototype, represented by the *exercise-recorder-bundle.js*, *exercise-player-bundle.js* and *exercise-player.html* files integrates into it. The application can be used at the address `https://jsav-player-test-app.firebaseapp.com/`, and the code can be found from the Player Application git repository `https://github.com/MarianiGiacomo/jsav-player-application.git`. Figure 4.13 is a screenshot of the test application showing the VAS exercise in the background and the EP in the foreground window.

We organized two testing sessions to evaluate the Player Application prototype with the test application. In each testing session the Aalto

Table 4.8. Evaluation of the PA prototype for functional requirements satisfaction.

Functional Requirements	Missing	Partial	Ready
User Point of View			
All the actions that can be performed on a data structure are recorder and replayed in the animation.		X	
The animation can be viewed step-by-step as a slideshow.			X
The animation can be viewed using play/pause/stop buttons.			X
All relevant information shown in the exercise is shown also in the animation.			X
Clicking the exercise control buttons is saved and shown in the animation.		X	
If the model answer is opened by the user, this is also shown in the animation.			X
The model answer is shown together with the animation, so that the user can compare it to the own solution.	X		
It is possible to share animations.		X	
It is possible to export & import animations.	X		
JAAL point of view			
All relevant GUI actions saved in JAAL.		X	
All relevant exercise data is recorded in JAAL.			X
All saved submissions are available in the JAAL format.		X	
The application can be connected to a LMS.	X		
The JAAL object is sent to the LMS when the user clicks the grade and reset buttons, and reloads or closes the window.		X	
The model answer is saved in JAAL.	X		
The JAAL object contains application version.	X		

University instructor representing the stakeholders impersonated the different application users, and used the application accordingly. After the first testing session, based on the feedback received we did some improvements to the prototype. The second testing session was also recorded, in order to analyze it later to better evaluate the prototype.

4.3.3 Satisfaction of Requirements

The results of the final evaluation of the Player Application prototype are shown in Table 4.8 for the *functional requirements*, and in Table 4.9 for the *qualitative requirements*. To establish how well the PA prototype satisfies the *functional* and *qualitative requirements*, after the two testing sessions we evaluated the degree to which each requirement was satisfied on a scale of *missing*, *partial* and *ready*. The evaluation of all the *functional requirements* and most of the *qualitative requirements* was done by the Aalto University instructor representing the application stakeholders. The evaluation of the *qualitative requirements* in the areas of application *maintainability* and *portability* was instead done by this master thesis supervisor.

As can be seen in the Tables 4.8 and 4.9, some of the requirements are not yet satisfied. The reason is that the prototype is not a mature

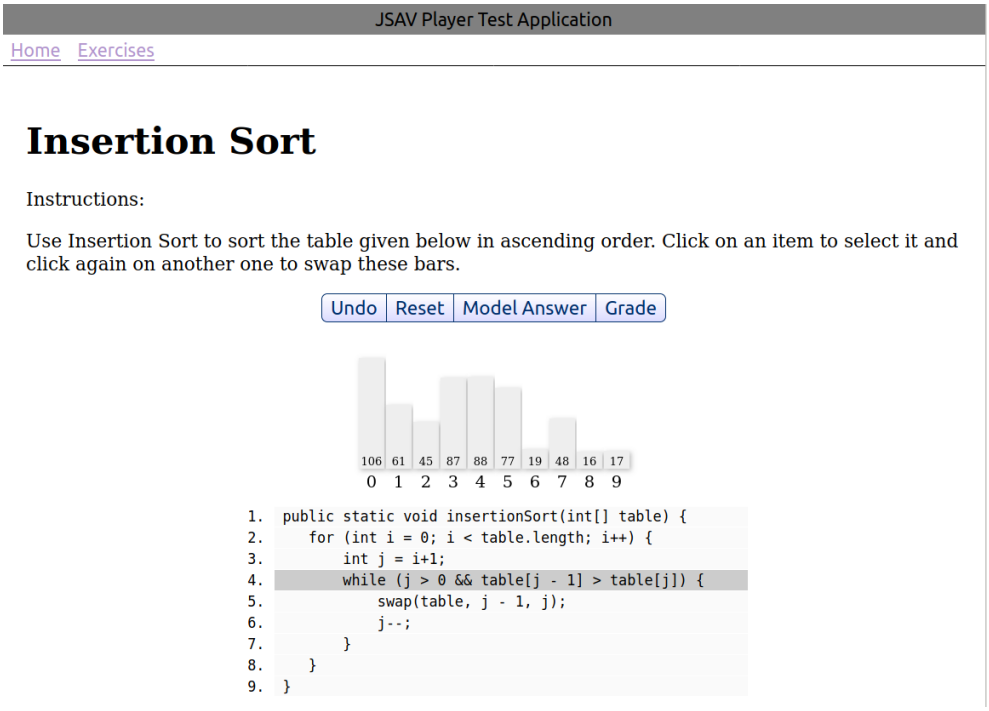


Figure 4.12. Screenshot of the test application showing the exercise before submitting it for grading.

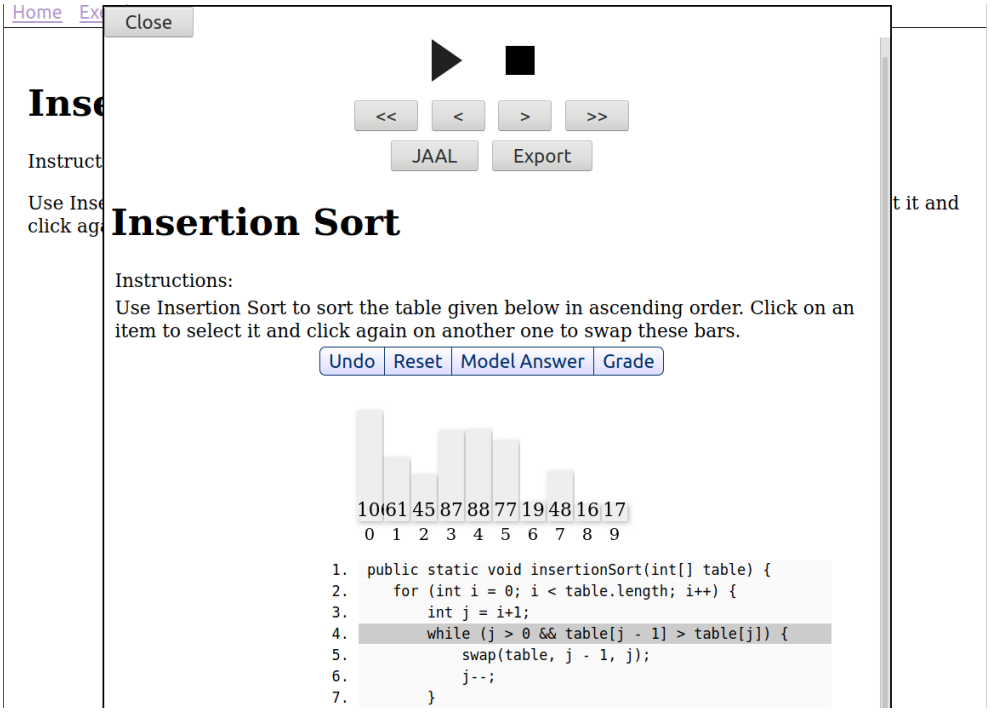


Figure 4.13. Screenshot of the test application showing the animation after submitting the exercise for grading.

application, but rather a proof of concept that the current design is valid, and it can be used to develop a complete Player Application to be used at Aalto University and other institutions. A more in depth analysis about the unsatisfied requirements and how to fulfill them in the future development of the Player Application is discussed in Section 5.2, along with the limitations of the current prototype.

Table 4.9. Evaluation of the Player Application prototype for qualitative requirements satisfaction.

Qualitative Requirements	Missing	Partial	Ready
Functional Suitability			
The functionalities specified by the requirements are present.		X	
The functionalities specified by the requirements work with a satisfactory degree of precision.			X
The functionalities specified by the requirements are presented in an appropriate and understandable way, avoiding unnecessary steps.			X
Performance Efficiency			
The recording of the VAS exercise does not slow down the execution of the exercise steps.			X
The creation of the animation does not require noticeable loading time.			X
The resources used by the application have to be reasonable in relation to its functionalities.			X
The amount of sessions is not limited.			X
For each exercise session is possible to record one exercise.			X
For each animation session is possible to play one animation at time.			X
Compatibility			
The application works in conjunction with JSAV and its required libraries.		X	
The application works when embedded in the Learning Management System.		X	
The application is able to exchange the required information with the LMS.		X	
Usability			
The application users find it useful when used according to the given use cases.		X	
It is easy to learn how to use the application.		X	
The application is easy to use and control.		X	
The application protects the user from making errors when recording the exercise or playing the animation.		X	
The user interface enables pleasing and satisfying interaction for the user.		X	
The application is usable also by people with diverse abilities.		X	
Reliability			
The application is reliable under normal conditions.			X
The application is available and accessible when required for use.		X	
The application works as intended even when an error occurs.		X	
The application recovers in case of failure or interruption without losing he data.	*N/A*		
Security			
The data is accessible only to those who are authorized to have access.			
The application prevents unauthorized access and modification of data.			
The application records relevant user interaction.	*LMS*		
The recorded user interaction is traced to the user.			
The identity of the user whose interaction has been recorded can be proven.			
Maintainability			
The application is designed of modules, such that change to one of them has minimal impact to other modules.			X
The modules composing the application can be easily reused in other software.		X	
The application is designed and written in a way, that analyzing it is not too difficult. For example to asses the impact of some changes, or diagnose the cause of a failure.		X	
The application can be effectively and efficiently modified without degrading its quality.		X	
Test criteria are established for the application, and tests can be performed to verify if the criteria are met.		X	
Portability			
The application can be adapted for use on different environments. For example different LMSs.			X
The application can be deployed in the needed environment efficiently and with effectiveness.			X
The application can easily be replaced with new versions of the same application or other similar applications.			X

In Figure 4.12 we can see the test application with the VAS exercise before submission, and Figure 4.13 shows the Exercise Player open in the foreground after the submission. As can be seen from the images, the same data structure which was in the exercise is also presented in the EP. The play and stop buttons can be used to play the animation automatically, while the arrow buttons are used to view the animation as a slide show and to step through it. Figure 4.13 shows two functionalities which were not yet implemented when we conducted the evaluation tests: the JAAL button, which can be used to view the content of JAAL object, and the export button, which opens a modal window containing an HTML *iframe* element as a string. The *iframe* can be directly added to any HTML document to embed the animation into it.

5. Discussion and Conclusions

In this Chapter we analyse the outcomes of the thesis project. We start in Section 5.1 by answering the research questions we proposed at the beginning of this thesis, as well by looking at the aim of the thesis project and how well we have achieved it. In Section 5.2 we analyse the limitations of this thesis project and of its main outcome the PA prototype, and we present the plans for future work. Finally, in Section 5.3 we discuss the contributions of this thesis project.

5.1 Discussion

The first research question we proposed at the beginning of this thesis was "*which solutions existing Algorithm Visualization software have used to record and replay Visual Algorithm Simulation exercises?*". To answer this question we looked at existing software, and the MatrixPro [26] tool incorporated in the TRAKLA2 [36] system is the only one we found that has VAS exercises and also offers the possibility to record and replay them. In MatrixPro the data structures are saved as serialized Java objects or ASCII files, and the visual elements can be exported to SVG or TEXdraw format. The object is saved every time a grade operation occurs, and it includes the learner answer as a sequence of data structures states. The data is sent to the server when the applet is initialized, the exercise is graded or reset, the model answer is opened or closed, the user has been idle over 60 seconds, and when a user operation ends the idle time. These log entries include a time stamp and identification data on the course, exercise, learner and perform operation.

The second question was "*which information should be saved in JSAV Visual Algorithm Simulation exercises submissions, in order to be able to replay them later as animations?*". To answer this question we analyzed

two languages used to describe Algorithm Animations: the eXtensible Algorithm Animation Language (XAAL) [22], and the General Purpose Animation Language (GPAL) [7]. From the analyses we saw that the data structures state, as well as graphical elements, should be saved in each step of the exercise submission, along with visualization options. Based on this information we designed the JSON-based Algorithm Animation Language, a new language to describe and share Algorithm Animations created from VAS exercises, which can easily be utilized by modern web applications.

The last question was "*how can we collect and replay JSAV Visual Algorithm Simulation exercises submissions?*". To find an answer we analyzed in depth the JSAV library, along with the visualizations and exercises which can be created with it. We understood that in order to save the relevant data, the PA has to listen for the events emitted by JSAV, and use the objects exposed through them to reach and save the relevant information. The outcome was the design of the Player Application with its two components: the Exercise Recorder and the Exercise Player.

Finally, we discuss the aim of this thesis project, which was to design the prototype for a web application, that can be used in conjunction with JSAV Visual Algorithm Simulation exercises, to collect, store, and replay user interaction data in a standardized way. We reached such an aim by *(i)* defining the JSON-based Algorithm Animation language, *(ii)* designing and developing the Player Application prototype, and *(iii)* evaluating the prototype in a test application which resembles a Learning Management System environment. The PA prototype currently has the following working features:

- Records and replays all actions performed on arrays in a JSAV Visual Algorithm Simulation exercise.

- Lets the learner review the submitted solution step-by-step or using the play/pause and stop buttons.

- Shows in the animation all the information presented to the learner during the exercise.

- Records and shows also when the model answer is opened by the learner during the exercise.

- Saves all submissions into a JAAL object which is posted to the URL specified by the Learning Management System upon grading the exercise.

- Shows the content of the JAAL object with the data collected from the VAS exercise submission.

- Offers the application user an *iframe* which can be used to embed the animation in an HTML document.

The prototype we created was useful to test the design, but has limitations which are presented into more details in the next section. The outcomes of this thesis project will be used to developed a fully working application, which will be integrated in the Visual Algorithm Simulation exercises of the Algorithm and Data Structures course at Aalto Unviersity in the autumn 2020.

5.2 Limitations and Future Work

There are limitations concerning the methodology we have used to carry out this thesis project. Perhaps the most relevant is the fact that one instructor represented all the Player Application stakeholders. This path was chosen due to time restrictions, but it limited the amount and quality of information used to define the use cases and the application requirements. The limitation also concerns the evaluation of the prototype, since it was done only by the instructor and the application developer, and not through testing by real users. Because of this, there is a risk that the designed prototype does not fully respond to needs and expectations of end users, that the evaluation process considered features that are not relevant, and that the outcomes would be different if the evaluation was to be done by real users.

A second limitation regards the methodologies we have used for this projects: Design Science for the PA design, and *test driven development* for PA development. Perhaps other methodologies such as Agile Software Development [3] or Development and Operations (DevOps) [12] should have been considered as well. In fact, with their focus on continuous delivery and integration during development, they might have helped to produce a prototype fully integrated in the Learning Management System. One of this two methodologies could still be used in the development of the

complete PA.

Next we look at the limitations of the main outcome of this thesis: the PA prototype. As shown in Table 4.8, some of the *functional requirements* are still missing. At the moment not all relevant GUI actions are recorded, and not all saved submissions are available as animations. The reason for this, is that the ER currently does not have support for data structures other than arrays. The Exercise Player is also still missing a complete import and export functionality, since it does not offer download or import of the JAAL object. Another important feature missing from the PA is the recording and replaying of the model answer.

The testing sessions highlighted the need for a number of improvements over the existing design:

- Possibility to look simultaneously at the exercise model answer and the animation in the Exercise Player. If possible, the wrong steps in the given solution should be highlighted.

- Higher default, minimum and maximum speed for the animation in the Exercise Player.

- Improving the user interface for the play, pause and stop buttons. These buttons could be shown when the user is not in the learner role.

- Removing the import button from the Exercise Player. It could instead be included when the EP is used as stand alone.

- Removing from the animation the steps of the learner looking at the model answer when solving the exercise. These steps could be shown when the user is other than a learner.

- Reduce the duration of the micro animations, like the one simulating a click over a data structure element. Even better would be to make the duration of the micro animations proportional to the chosen animation speed.

- Exercise result should be shown more clearly in the EP.

- Generally there is a need to customise the Exercise Player depending on

the user role. This could be achieved either by developing this feature according to the LMS protocol, or by implementing certain features only on the standalone version of the EP.

The prototype which is deployed in the test application is still being developed, and some of the issues presented above were already addressed after the testing and evaluation. For example, as can be seen in Figure 4.13, the import button is no longer shown by default, and neither are the model answer steps viewed by the user during the exercise.

Some limitations are present also in the satisfaction of the *qualitative requirements*. The lacks present in the *functional suitability* are due to the fact that not all *functional requirements* are yet fulfilled, and this has been addressed above. The other *qualitative requirements* areas which still need improvement are the *usability*, *reliability*, *security*, and *portability*. We will analyse this limitations one category at the time:

- *Usability*: the possibility of the Player Application protecting the user from making errors are very limited, since its role is barely recording the GUI actions. One feature which can be added is asking for confirmation when the user closes the current window. A more pleasant UI will instead be developed in the final version of the application. As for the usability for people with diverse abilities, this is something which requires special attention and will be addressed in the future development.

- *Reliability*: Currently, if the VAS exercise includes JSAV data structures or actions not yet supported, the prototype would easily end up in an error state, and would not be able to record all necessary data. These conditions will gradually decrease as the application is developed further.

- *Security*: The Player Application security is highly dependent on the LMS where it is embedded, and on the exercise server. The data received by the LMS is not exposed in global variables, and it is posted to the URL given by the LMS. Since the PA is designed to be used within a LMSs, it does not have any access control.

- *Maintainability*: The PA maintainability will have to be increased by adding more documentation and end-to-end testing on top of the existing integration tests.

- *Portability*: Since many different LMS exist and with different protocols, the amount of work required to adapt the PA to a new LMS will depend also on the LMS itself. As long as the LMS can load the Exercise Recorder and Exercise Player into an iframe, and pass them the necessary query parameters, no change will be required on the PA itself.

On top of the limitations just presented, there are two more which arose already in Section 4.2.3 where discussed the architecture of the PA:

- New data structure types not included in the JSAV core data structures will have to emit the same JSAV log events as the core data structures, and the support for the new data structures will have to be added to the PA.

- The Player Application is only able to create animations, if the input data is in the form specified by the JSON-based Algorithm Animation Language.

In this thesis project we followed the *Design Cycle*, which represents the first three of the five phases of the *Engineering Cycle*. Future work will be focused on the *implementation* and *evaluation* of the ready Player Application, which are two phases of the *Engineering Cycle* that were not covered in this work, like shown in Figure 5.1. Future work will therefore focus at first on developing a fully working Player Application, integrating it in the A+ Learning Management System, evaluating it through testing with real users, and later on analyzing the data collected by the Exercise Recorder and stored in JAAL format in the database.

5.3 Conclusions

The main contribution of this thesis project is the creation of a prototype application to record and replay Visual Algorithm Simulation exercises created with the JavaScript Algorithm Animation library. The application prototype is a proof of concept that such an application can be created following the proposed design. The code for the Player Application prototype with instruction on how to use it is available as a repository at `https://github.com/MarianiGiacomo/jsav-player-application.git`. The same repository also contains the code for the test application front-end (the test

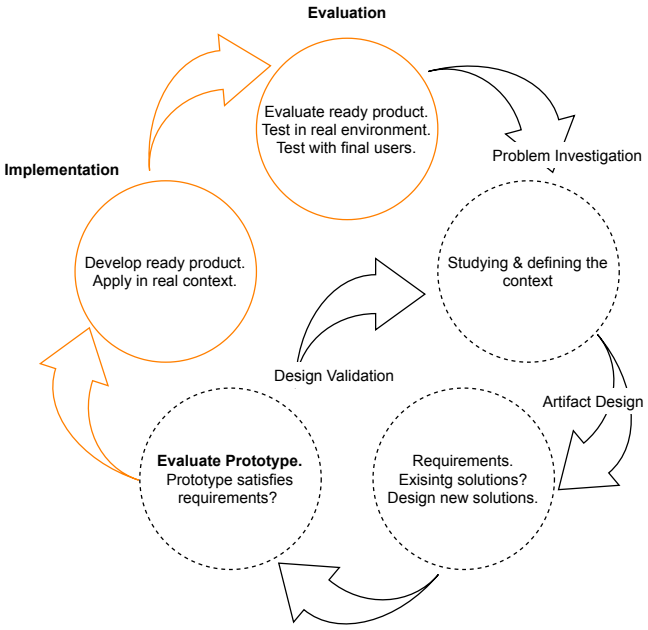


Figure 5.1. The engineering cycle. In future work we will implement the last two phases of the engineering cycle.

LMS) and back-end (the test exercise server). The Player Application main components are the Exercise Recorder and the Exercise Player, and they can also be used independently to only record and store a VAS exercise animation into a JAAL object, or to create an animation from a JAAL object.

A second contribution of this thesis is the creation of the JSON-based Algorithm Animation Language, which can be used to describe Algorithm Animations. A JAAL file can also be created by any application or by other means, without the use of the Exercise Recorder, and the Exercise Player can be used to visualize it.

A third contribution is the documentation we have created about JSAV, which we have presented in Sections 2.5 and 4.1.3, and can be found in the Appendix. This documentation can be used by anybody who wishes to learn how to create algorithm visualizations and exercises with JSAV, or who wants to develop an application based on it.

A last contribution is the testing application, which simulates an exercise server and a simple Learning Management System using a protocol similar to the A+ grader service. This application can be used as a testing ground for services and exercises that need to be integrated into the A+ LMS, or as a tool for teaching about LMSs. The exercise server could be further developed to work as a service offering JSAV VAS exercises with the Exercise Recorder integrated in them, the Exercise Recorder itself, and the Exercise Player also as a standalone application.

References

[1] AlgoViz. AlgoViz.net. URL http://www.algoviz.net/. Accessed: Dec 2019.

[2] A. J. D. Alon. *The AlgoViz Project: Building an Algorithm Visualization Web Community*. PhD thesis, Virginia Tech, 2010.

[3] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. Manifesto for agile software development. 2001.

[4] M. H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *Proceedings 1991 IEEE Workshop on Visual Languages*, pages 4–9. IEEE, 1991.

[5] M. H. Brown and R. Sedgewick. A system for algorithm animation. *SIGGRAPH Comput. Graph.*, 18(3):177–186, Jan. 1984.

[6] M. D. Byrne, R. Catrambone, and J. T. Stasko. Do algorithm animations aid learning? *GVU Center Technical Reports*, 18, 1996.

[7] V. K. Calpakkam. General Purpose Animation Language, 2019. URL https://lib.dr.iastate.edu/creativecomponents/148.

[8] H. Coates, R. James, and G. Baldwin. A critical examination of the effects of learning management systems on university teaching and learning. *Tertiary education and management*, 11:19–36, 2005.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[10] C. Dalsgaard. Social software: E-learning beyond learning management systems. *European Journal of Open, Distance and e-learning*, 9(2), 2006.

[11] C. Demetrescu, I. Finocchi, and J. T. Stasko. Specifying algorithm visualizations: Interesting events or state mapping? In *Software Visualization*, pages 16–30. Springer, 2002.

[12] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.

[13] R. K. Ellis. Field guide to learning management systems. *ASTD learning circuits*, pages 1–8, 2009.

[14] E. Fouh, V. Karavirta, D. A. Breakiron, S. Hamouda, S. Hall, T. L. Naps, and C. A. Shaffer. Design and architecture of an interactive eTextbook–The OpenDSA system. *Science of Computer Programming*, 88:22–40, 2014.

[15] S. Halim. Visualgo–visualising data structures and algorithms through animation. *Olympiads in Informatics*, page 243, 2015.

[16] P. Hill. Academic LMS Market Share: A view across four global regions, Jun 2017. URL https://eliterate.us/academic-lms-market-share-view-across-four-global-regions. Accessed: Feb 2020.

[17] C. D. Hundhausen. Integrating algorithm visualization technology into an undergraduate algorithms course: ethnographic studies of a social constructivist approach. *Computers & Education*, 39(3):237–260, 2002.

[18] IMS GLOBAL. Learning tools interoperability core specification 1.3, Apr 2019. URL https://www.imsglobal.org/spec/lti/v1p3/. Accessed: Feb 2020.

[19] ISO/IEC 25010:2011. Systems and software engineering–systems and software quality requirements and evaluation (SQuaRE)–system and software quality models. *International Organization for Standardization*, 34:2910, 2011.

[20] Jest. Jestjs.io HomePage. URL https://www.jestjs.io. Accessed: March 2020.

[21] V. Karavirta. XAAL-extensible algorithm animation language. *Master's thesis, Department of Computer Science and Engineering, Helsinki University of Technology*, 2005.

[22] V. Karavirta. Integrating algorithm visualization systems. *Electronic Notes in Theoretical Computer Science*, 178:79–87, 2007.

[23] V. Karavirta. XAAL Documentation, Sep 2009. URL http://xaal.org/documentation/index.html. Accessed: Dec 2019.

[24] V. Karavirta and C. A. Shaffer. JSAV: the JavaScript algorithm visualization library. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 159–164. ACM, 2013.

[25] V. Karavirta and C. A. Shaffer. Creating engaging online learning material with the jsav javascript algorithm visualization library. *IEEE Transactions on Learning Technologies*, 9(2):171–183, 2015.

[26] V. Karavirta, A. Korhonen, L. Malmi, and K. Stalnacke. MatrixPro-A tool for on-the-fly demonstration of data structures and algorithms. In *Proceedings of the Third Program Visualization Workshop*, pages 26–33. Citeseer, 2004.

[27] V. Karavirta, A. Korhonen, and L. Malmi. Taxonomy of algorithm animation languages. In *Proceedings of the 2006 ACM symposium on Software visualization*, pages 77–85. ACM, 2006.

[28] V. Karavirta, P. Ihantola, and T. Koskinen. Service-oriented approach to improve interoperability of e-learning systems. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*, pages 341–345. IEEE, 2013.

[29] V. Karavirta, A. Korhonen, and O. Seppälä. Misconceptions in visual algorithm simulation revisited: On UI's effect on student performance, attitudes and misconceptions. In *2013 Learning and Teaching in Computing and Engineering*, pages 62–69. IEEE, 2013.

[30] C. Kehoe, J. Stasko, and A. Taylor. Rethinking the evaluation of algorithm animations as learning aids: an observational study. *International Journal of Human-Computer Studies*, 54(2):265–284, 2001.

[31] A. Korhonen and L. Malmi. Matrix: Concept Animation and Algorithm Simulation System. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '02, pages 109–114, New York, NY, USA, 2002. ACM.

[32] A. Korhonen, M.-J. Laakso, and N. Myller. How does algorithm visualization affect collaboration? Video analysis of engagement and discussions. *Webist*, 2009.

[33] A. Korhonen, O. Seppälä, and J. Sorva. Automatic recognition of misconceptions in visual algorithm simulation exercises. In *2015 ieee frontiers in education conference (fie)*, pages 1–5. IEEE, 2015.

[34] A. Korhonen et al. *Visual algorithm simulation*. Helsinki University of Technology, 2003.

[35] M.-J. Laakso, N. Myller, and A. Korhonen. Comparing learning performance of students using algorithm visualizations collaboratively on different engagement levels. *Journal of Educational Technology & Society*, 12(2): 267–282, 2009.

[36] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in education*, 3(2):267, 2004.

[37] Math Vault Glossary of Higher Mathematical Jargon. Algorithm. URL `https://mathvault.ca/math-glossary/#algo`. Accessed: Nov 2019.

[38] T. J. McGill and J. E. Klobas. A task–technology fit view of learning management system impact. *Computers & Education*, 52(2):496–508, 2009.

[39] Merriam-Webster Dictionary. Definition of Algorithm. URL `https://www.merriam-webster.com/dictionary/algorithm`. Accessed: Nov 2019.

[40] Moodle LMS. Moodle.com HomePage. URL `https://moodle.com/lms`. Accessed: Feb 2020.

[41] N. Myller, M. Laakso, and A. Korhonen. Analyzing engagement taxonomy in collaborative algorithm visualization. *ACM SIGCSE Bulletin*, 39(3):251–255, 2007.

[42] N. Myller, R. Bednarik, E. Sutinen, and M. Ben-Ari. Extending the engagement taxonomy: Software visualization and collaborative learning. *ACM Transactions on Computing Education (TOCE)*, 9(1):7, 2009.

[43] T. L. Naps, J. R. Eagan, and L. L. Norton. JHAVÉ & Mdash;an Environment to Actively Engage Students in Web-based Algorithm Visualizations. *SIGCSE Bull.*, 32(1):109–113, Mar. 2000.

[44] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, et al. Exploring the role of visualization and engagement in computer science education. *ACM Sigcse Bulletin*, 35(2):131–152, 2002.

[45] J. Nikander, J. Helminen, and A. Korhonen. Experiences on using TRAKLA2 to teach spatial data algorithms. *Electronic Notes in Theoretical Computer Science*, 224:77–88, 2009.

[46] NPM. Npmjs.com HomePage. URL `https://www.npmjs.com/`. Accessed: March 2020.

[47] B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages & Computing*, 4(3):211–266, 1993.

[48] M. Rey-López, P. Brusilovsky, M. Meccawy, R. Díaz-Redondo, A. Fernández-Vilas, and H. Ashman. Resolving the problem of intelligent learning content in learning management systems. *International Journal on E-Learning*, 7 (3):363–381, 2008.

[49] G. Rößling and B. Freisleben. ANIMAL: A system for supporting multiple roles in algorithm animation. *Journal of visual Languages and computing*, 13, 2002.

[50] SCORM. SCORM.com HomePage. URL `https://scorm.com`. Accessed: Feb 2020.

[51] J. Stasko, A. Badre, and C. Lewis. Do algorithm animations assist learning?: an empirical study and analysis. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 61–66. ACM, 1993.

[52] J. T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, 23(9):27–39, 1990.

[53] J. Venable, J. Pries-Heje, and R. Baskerville. Feds: a framework for evaluation in design science research. *European journal of information systems*, 25 (1):77–89, 2016.

[54] VisuAlgo. Visualgo.net. URL `https://visualgo.net/`. Accessed: Dec 2019.

[55] R. H. Von Alan, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS quarterly*, 28(1):75–105, 2004.

[56] P. Wegner and E. D. Reilly. Data Structures. In *Encyclopedia of Computer Science*, pages 507–512. John Wiley and Sons Ltd., Chichester, UK, 2003.

[57] R. J. Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[58] xAPI. xAPI.com Homepage. URL `https://xapi.com/`. Accessed: Feb 2020.

A. Appendix

A.1 JSAV Objects

A.1.1 Array

```
{
    _indices: [...
      { container: {...} ,
        element: {
          // JSAV index DOM element object
          0: { ...className: "jsavnode jsavindex" }
        },
        // JSAV visualization object
        jsav: {...}
        options: {
            autoresize: Boolean,
            center: Boolean,
            indexed: Boolean,
            layout: String,
            template: String,
            visible: Boolean
        },
      },
    ],
    _values: [ Number ],
    element: {
      // JSAV array DOM element object
      0: {...className: "jsavautoresize jsavcenter jsavindexed jsavarray
     jsavbararray"}
    },
    // JSAV visualization object
    jsav: {...},
    options: {
        autoresize: Boolean,
        center: Boolean,
        indexed: Boolean,
        layout: String,
        template: String,
        visible: Boolean,
    }
}
```

Listing A.1. JSAV array object.

A.1.2 Binary Tree

```
{
  _layoutDone: Boolean
  constructors: {...},
  element: {
    // JSAV binary tree DOM element object
    0:  {...className: "jsavtree jsavcommontree jsavautoresize jsavbinarytree" }
  },
   // JSAV visuaization object
  jsav: { },
  options:
  {
    autoresize: Boolean,
    center: Boolean,
    constructors: {...},
    left: Number,
    nodegap: Number,
    visible: Boolean,
  },
  // JSAV node object
  rootnode:
  {
    _edgetoparent: undefined,
    _layoutDone: Boolean,
    childnodes: [ {...}, {...} ],
    constructors: {...},
    container: {...},
    element: { 0:  {...} },
    options:
    {
      visible: Boolean
    },
    parentnode: undefined
  },
  // SVG Object
  svg: {...},
}
```

Listing A.2. JSAV binary tree object.

A.1.3 Common Tree

```
{
  _layoutDone: Boolean
  constructors: {...},
  element: {
    // JSAV common tree DOM element object
    0:  {...className: "jsavtree jsavcommontree jsavautoresize" }
  },
  // JSAV visualization object
  jsav: { },
  options:
  {
    autoresize: Boolean,
    center: Boolean,
    constructors: {...},
    left: Number,
    nodegap: Number,
    visible: Boolean,
  },
  // JSAV node object
  rootnode:
  {
    _edgetoparent: undefined,
    _layoutDone: Boolean,
    childnodes: [ {...}, {...} ],
    constructors: {...},
    container: {...},
    element: { 0:  {...} },
    options:
    {
      visible: Boolean
    },
    parentnode: undefined
  },
  // SVG Object
  svg: {...},
}
```

Listing A.3. JSAV common tree object.

A.1.4 Tree Node

```
{
  // JSAV edge object
  _edgetoparent: {...},
  _layoutDone: Boolean,
  // JSAV node objects
  childnodes: [ {...}, {...} ],
  constructors: {...},
  container: {...},
  element: {
    // JSAV node DOM element object
    0:  { ...className: "jsavnode jsavtreenode jsavbinarynode" }
  },
  jsav: {...}, // JSAV Object
  options: { visible: Boolean },
  parentnode: undefined
}
```

Listing A.4. JSAV tree node object.

A.1.5 Edge

```
{
  container: {...},
  element: {
    // JSAV edge DOM element object
    0:  { ...className { baseVal: "jsavedge", animVal: "jsavedge" } }
  },
  // JSAV node Object
  endnode: {...},
  g: {...},
  // JSAV Object
  jsav: {...},
  options: { display: Boolean },
  // JSAV node object
  startnode: {...},
  _layoutDone: Boolean
}
```

Listing A.5. JSAV edge object.

A.1.6 Graph

```
{
  // JSAV edges objects
  _alledges: [ ... {...}, {...} ],
  _edges: [ ...[{...}], [{...}] ],
  // JSAV node objects
  _nodes: [ ... {...}, {...} ],
  element: {
    // JSAV graph DOM element object
    0: {...className: "jsavgraph jsavautoresize jsavcenter"}
  },
  // JSAV visualization object
  jsav: {...},
  options:
  {
    autoresize: Boolean,
    center: Boolean,
    directed: false,
    height: Number,
    layout: automatic,
    nodegap: Number,
    visible: Boolean,
    width: Number
  },
  // SVG object
  svg: {...}
}
```

Listing A.6. JSAV graph object.

A.1.7 Graph Node

```
{
  container: {...},
  element: {
    // JSAV node DOM element object
    0:  { ...className: "jsavnode.jsavgraphnode" }
  },
  // JSAV Object
  jsav: {...},
  options: {
    center: Boolean,
    left: 0,
    top: 0,
    visible: Boolean,
   },
}
```

Listing A.7. JSAV graph node object.

A.1.8 Linked List

```
{
  // JSAV list node object
  _first: {...},
  element: {
    // JSAV list DOM Element Object
    0:  { ...className: "jsavlist jsavautoresize jsavhorizontallist" }
  },
  options:
  {
    autoresize: Boolean,
    nodegap: Number,
    visible: Boolean,
  },
  // SVG Object
  svg: {...},
}
```

Listing A.8. JSAV linked list object.

A.1.9 Linked List Node

```
{
  // JSAV edge object
  _edgetonext: {...},
  // JSAV list node object
  _next: { },
  _value: Number | String,
  container: {...},
  element: {
    // JSAV list node DOM element object
    0: {...className "jsavnode jsavlistnode" }
  },
  // JSAV visualization object
  jsav: {...},
  options:
  {
    autoresize: Boolean,
    first: Boolean,
    nodegap: Number,
    visible: Boolean,
  },
}
```

Listing A.9. JSAV linked list node object.

A.1.10 Matrix

```
{
  // JSAV array objects
  _arrays: [ ... {...}, {...}],
  element: {
    // JSAV matrix DOM elment object
    0: { ...className: "jsavmatrix jsavmatrixtable jsavcenter"}
  },
  // JSAV object
  jsav: {...},
  options:
  {
    autoresize: Boolean,
    center: Boolean,
    style: table,
    visible: Boolean,
  }
}
```

Listing A.10. JSAV matrix object.

A.1.11 Exercise

```
{
  _undoneSteps: [...],
  // if 1 ds object -> {}; else []
  initialStructures: {} | [],
  // JSAV visualization object
  jsav: {...},
  options:
    {
      controls: {...},
      feedback: String,
      feedbackSelectable: Boolean,
      fixmode: String,
      fixmodeSelectable: Boolean,
      gradeButtonTitle: String,
      grader: String,
      model: function modelSolution(),
      modelButtonTitle: String,
      reset: function initialize(),
      resetButtonTitle: String,
      undoButtonTitle: String
    },
  score:
    {
      correct: Number,
      fix: Number,
      student: Number,
      total: Number,
      undo: Number
    }
}
```

Listing A.11. JSAV exercise object.

A.1.12 Events

```
{
  av: String, // JSAV canvas className or id
  currentStep: Number,
  initialHTML: String, // Contains exercise instructions
  tstamp: TimeStamp,
  type: "jsav-init"
}
{
  av: String, // JSAV canvas className or id
  currentStep: Number,
  tstamp: TimeStamp,
  type: "jsav-recorded"
}
```

Listing A.12. JSAV log events from JSAV visualization object.

```
{
  av: String, // JSAV canvas className or id
  currentStep: Number,
  exercise: {...}, // Exercise object
  tstamp: TimeStamp,
  type: "jsav-exercise-init"
}
{
  av: String, // JSAV canvas className or id
  currentStep: Number,
  tstamp: TimeStamp,
  type: "jsav-exercise-undo"
}
{
  av: String, // JSAV canvas className or id
  currentStep: Number,
  tstamp: TimeStamp,
  type: "jsav-exercise-reset"
}
{
  av: String, // JSAV canvas className or id
  currentStep: Number,
  tstamp: TimeStamp,
  type: "jsav-exercise-model-open"
}
{
  av: String, // JSAV canvas className or id
  currentStep: Number,
  tstamp: TimeStamp,
  type: "jsav-exercise-model-close"
}
{
  av: String, // JSAV canvas className or id
  currentStep: Number,
  tstamp: TimeStamp,
  type: "jsav-exercise-gradeable-step"
}
{
  av: String, // JSAV canvas className or id
  currentStep: Number,
  score: { ... }, // JSAV score object
  tstamp: TimeStamp,
  type: "jsav-exercise-grade-button"
}
{
  av: String, // JSAV canvas className or id
  currentStep: Number,
  score: { ... }, // JSAV score object
  tstamp: TimeStamp,
  type: "jsav-exercise-grade"
}
```

Listing A.13. JSAV log events from JSAV exercise object.

```
{
  arrayid: String,
  av: String, // JSAV canvas className or id
  currentStep: Number,
  index: Number,
  tstamp: TimeStamp,
  type: "jsav-array-click"
}
{
  av: "jsavcontainer",
  currentStep: Number,
  objid: String, // JSAV node DOM element id
  objvalue: Number,
  tstamp: TimeStamp,
  type: "jsav-node-click"
}
{
  av: "jsavcontainer",
  column: Number,
  currentStep: Number,
  matrixid: String, // JSAV matrix DOM element id
  row: Number,
  tstamp: TimeStamp,
  type: "jsav-matrix-click"
}
```

Listing A.14. JSAV log events from JSAV data structure objects.

A.2 JAAL file

```
{
  "metadata": [],
  "definitions": {},
  "initialState":
  {
    dataStructures:
    [
      {
        "type": array,
        "id": "0d15981decd54d55aa6f679eb45e5205",
        "values": [0, 1, 2],
        "options": { "indexed": true, "orientation": "horizontal" }
      }
    ],
    animationDOM: "..."
  },
  "animation": [],
}
```

Listing A.15. Representation of an array in JAAL.

```
{
  "metadata": [],
  "definitions": {},
  "initialState":
  {
    dataStructures:
    [
      {
        "type": binaryTree,
        "id": "e0103a38b6ce417f8eddc232475d3344",
        "values": [0, 1, 2],
        "options": { },
        "nodes":
        [
          {
            "id": "617987d4d3b344be8ed2cc59bdacb92c"
            "value": 0,
            "childNodes":
              {
                "leftChild": "cdaf6a3b92f14ea78cbbab64ceee6089",
                "rightChild": "b1907770876c4a7fbfcb9b46cb0188bf"
              }
          },
          {
            "id": "cdaf6a3b92f14ea78cbbab64ceee6089",
            "value": 1,
            "parentNode": "617987d4d3b344be8ed2cc59bdacb92c"
          },
          {
            "id": "b1907770876c4a7fbfcb9b46cb0188bf",
            "value": 2,
            "parentNode": "617987d4d3b344be8ed2cc59bdacb92c"
          }
        ]
      }
    ]
    animationDOM: "..."
  },
  "animation": [],
}
```

Listing A.16. Representation of a tree in JAAL.

```
{
  "metadata": [],
  "definitions": { ... },
  "initialState": { ... },
  "animation": [
    {
      "type": "click",
      "tstamp": "2020-04-26T12:25:28.527Z",
      "currentStep": 0,
      "dataStructure": {
        "id": "exerArray",
        "values": [
          "102",
          "77",
          "35",
        ]
      },
      "index": 0,
      "animationDOM": "..."
    },
    {
      "type": "click",
      "tstamp": "2020-04-26T12:25:29.275Z",
      "currentStep": 1,
      "dataStructure": {
        "id": "exerArray",
        "values": [
          "102",
          "77",
          "35",
        ]
      },
      "index": 1,
      "animationDOM": "..."
    },
    {
      "type": "gradeable-step",
      "tstamp": "2020-04-26T12:25:29.748Z",
      "currentStep": 2,
      "dataStructuresState": [
        {
          "id": "exerArray",
          "values": [
            "77",
            "102",
            "35",
          ]
        }
      ],
      "animationDOM": "..."
    },
    {
      "type": "grade",
      "tstamp": "2020-04-26T12:25:31.673Z",
      "currentStep": 2,
      "score": {
        "total": 23,
        "correct": 1,
        "undo": 0,
        "fix": 0,
        "student": 1
      }
    }
  ],
}
```

Listing A.17. Animation steps in JAAL.

A.3 Exercise Recorder

```
function passEvent(eventData) {
  console.log('EXERCISE', exercise);
  console.log('EVENT DATA', eventData);
  switch(eventData.type){
    case 'jsav-init':
      def_func.setExerciseOptions(eventData);
      metad_func.setExerciseMetadata(eventData);
      break;
    case 'jsav-exercise-init':
      exercise = eventData.exercise;
      jsav = exercise.jsav;
      def_func.setDefinitions(exercise);
      init_state_func.setInitialDataStructures(exercise);
      init_state_func.setAnimationDOM(exercise);
      break;
      // Here we handle all array related events
    case String(eventData.type.match(/^jsav-array-.*/)):
      exerciseDOM = helpers.getExerciseDOM(exercise)
      anim_func.handleArrayEvents(exercise, eventData, exerciseDOM);
      break;
    // This is fired by the initialState.js because JSAV sets array ID only on
     first click
    case 'recorder-set-id':
      init_state_func.setNewId(eventData);
      break;
    case 'jsav-exercise-undo':
      setTimeout(() => anim_func.handleGradableStep(exercise, eventData), 100);
      break;
    case 'jsav-exercise-gradeable-step':
      exerciseDOM = helpers.getExerciseDOM(exercise)
      anim_func.handleGradableStep(exercise, eventData, exerciseDOM);
      break;
    case 'jsav-exercise-grade-button':
      break;
    case 'jsav-exercise-grade':
      // JSAV emits the model answer event when grade is clicked
      // We remove the last animation step caused by the model answer event
      submission.checkAndFixLastAnimationStep();
      anim_func.handleGradeButtonClick(eventData);
      def_func.setFinalGrade(eventData) && services.sendSubmission(submission.
     state(), post_url);
      submission.reset();
      $(document).off("jsav-log-event");
      break;
    case String(eventData.type.match(/^jsav-exercise-model-.*/)):
      anim_func.handleModelAnswer(exercise, eventData);
      break;
    case 'jsav-recorded':
      break;
    default:
      console.warn('UNKNOWN EVENT', eventData);
  }
}
```

Listing A.18. The passEvent() function in the Exercise Recorder, which handles JSAV log events.

A.4 Exercise Player

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta content="text/html;charset=utf-8" http-equiv="Content-Type">
  <meta content="utf-8" http-equiv="encoding">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Visual Algorithm Simulatin Player</title>
  <link rel="stylesheet"  href="../css/JSAV.css" type="text/css" />
  <link rel="stylesheet"  href="../css/player.css" type="text/css" />
  <link rel="stylesheet" href="../css/odsaStyle-min.css" type="text/css" />
  <script src="../lib/jquery.min.js"></script>
  <script src="../lib/jquery-ui.min.js"></script>
  <script src="../lib/jquery.transit.js"></script>
</head>
<body>
  <div class="player-content">
    <div class="auto-animation">
      <button class="animation-button" id="play-pause-button"></button>
      <button class="animation-button" id="stop-button"></button>
    </div>
    <div class="slide-show">
      <button id="to-beginning">&lt;&lt;</button>
      <button id="step-backward">&lt;</button>
      <button id="step-forward">&gt;</button>
      <button id="to-end">&gt;&gt;</button>
    </div>
    <div class="import-export">
      <button id="jaal">JAAL</button>
      <button id="export">Export</button>
    </div>
    <div class="animation-instructions">
    </div>
  </div>
  <div id="animation-container">
  </div>
  <div class="model-answer">
    <div id="model-answer"></div>
  </div>
  <div id="myModal" class="modal">
  <div class="modal-content">
    <div>
      <span class="close">&times;</span>
    </div>
    <div>
      <code><pre id="modal-content"></pre></code>
    </div>
  </div>
</div>
</body>
  <script src="./jsav-exercise-player-bundle.js"></script>
</html>
```

Listing A.19. Content of the player.html file.