Bachelor thesis
Computer Science

Radboud University

# Digitizing an interactive collaborative teaching methodology using web technologies

*Author:*
Aaron van Geffen
3058026

*First supervisor/assessor:*
prof. dr. Erik Barendsen
e.barendsen@cs.ru.nl

*Second assessor:*
dr. Peter Achten
p.achten@cs.ru.nl

August 21, 2015

## Abstract

In the Netherlands, several schools have adopted a pilot programme in which every student in a class owns a *tablet*. While publishers have been adapting their learning methodologies into e.g. ebooks, there is little focus on adapting existing topic-specific teaching methodologies to digital platforms. We aim to change this by doing a case study on the digitisation process of a methodology that is used to teach history classes: *Active Historical Thinking*.

We will show how we have adapted the teaching methodology as directly as possible, outlining the design choices we have had to make. Moreover, we will discuss and preview how we extended the methodology to include a supportive interactive whiteboard application, which can be used by teachers to illustrate and settle disputes in classroom discussions after each round.

To achieve this, we will show how to make use of modern web technologies, including WebSockets and JSON, to make real-time communication possible.

### Acknowledgments

I would like to thank the following people for their direct or indirect involvement in my thesis:

- Erik Barendsen, who pitched the idea to me and acted as supervisor throughout the process. While the project ended up taking more time than we both had anticipated, he has patiently seen the journey through with me to the end.
- Harry Havekes and Arnoud Aardema of the *Radboud Docenten Academie*, for their involvement in the development process of the application. Their dedication and enthusiasm have really helped my resolve in wanting to bring the project to a successful end.
- My parents, Joop and Loes van Geffen, who I have made more than a little worried about me, when fretting my work would not be good enough.
- My friends Tim Steenvoorden and Judith van Stegeren, whose support has meant the world to me, especially during the summer months I spent finishing this thesis.

# Contents

# Chapter 1

# Introduction

Since the end of the 90s, there has been a continuous increase in the use of digital media in classrooms, sparked by affordable hardware and the rise of the Internet.

Recently, *personal devices* have been on the rise. Most notably, the introduction of the iPad in April 2010 changed how we interact with digital surfaces. Previously, tablets worked with special *styli* to move the pointer to a different position – the software itself was still the classic desktop software. The iPad changed this with its *capacitive* touch screen, allowing the use of fingers, and an operating system fully tailored to tablet needs.

The technology has been lauded by many sectors, one of which is the education sector. In the Netherlands, several schools have adopted a pilot programme in which every student in a class owns one of these *tablets*. As a direct result, publishers have reacted to this by adapting existing text and workbooks into either educational apps or ebooks, making use of the ecosystem that modern tablets provide, and relieving the need for students to carry heavy books in their backpacks.

Application listings show educational apps are mostly limited to such digitisations of text and workbooks, as well as apps that focus on playfully communicating knowledge through games. (*Eduapp*, 2013)

Unfortunately, there is little focus on adapting existing topic-specific teaching methodologies to digital platforms. We aim to change this by doing a case study on the digitisation process of a methodology that is used to teach history classes: *Active Historical Thinking*, which has been used in secondary education in the Netherlands since 2004.

## 1.1 Aim of the study

Through this, we hope to both directly and indirectly contribute to the education process, as the techniques we use may potentially be applied to other teaching methodologies as well.

In this thesis, we aim to answer the research question '*How can we digitise the teaching methodology Active Historical Thinking?*'. To answer this question, we have formulated the following subquestions:

1. What are the original goals of the teaching methodology, and how can we ensure they are met by the digitised system?

2. What are the requirements for the digitised system?

3. How can we facilitate the collaboration process do we synchronise canvases between students, keep canvases in sync?

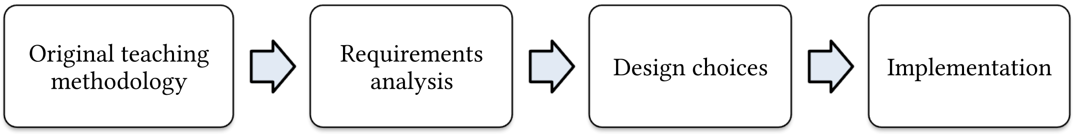
4. What choices need to be made with respect to techniques that come into play?

## 1.2 Reading guide

First, we will look at how the original teaching methodology works in chapter 2. From its principles and stakes, we will do a brief requirements analysis in chapter 3.

To meet these requirements, we will be making several technical design choices. In chapter 4, we will look at a variety of options, before making the choices in chapter 5.

Finally, in chapter 6 we will go into detail on several aspects of the implementation, before briefly discussing our findings in chapter 7.



## Chapter 2

# Active Historical Thinking

In this chapter, we will look at a methodology that can be used to support teaching history classes in high school: Active Historical Thinking.

### 2.1 Motivation

In order to reason about history, it is imperative to have background knowledge about the time period in question. When we speak of learning history, we usually mean the process of looking at a certain period of time, while focussing on a certain place – a country or continent, in order to get a better understanding of the *changes* that occured during this time. Of course, this involves thoroughly looking at events that transpired, and (political) movements and ideologies that bridge these.

While learning the *facts* involved may be relatively straightforward, this process is more involved than one might initially think. Often, there is no such thing as *the* truth when re-constructing a historical context. This becomes apparent when we consider the facts given may be selective, or even untrue. Moreover, there are many angles to look at the same problem – change does not happen overnight and is the result of many events transpiring before it.

The process of historical contextualisation is therefore a hard process; there is not just one 'right' answer. In addition, students have difficulty connecting apparently discrete events into a continous, chronological overview. Havekes et al. acknowledge this (2012, p. 86):

> "Constructing a historical context is so complex that guidance through a well-structured learning sequence is not enough. [...] Guidance by a teacher is needed to stimulate students to challenge their construction and stimulate their thinking. The teacher must ask questions to further assess and challenge the (initial) answers and arguments by the students. To do this, students' thinking needs to be visible to the teacher (brains-on-the-table). Working in groups and classroom discussions are useful tools to make the thinking of the students visible."

### 2.2 Methodology

At the end of a series of classes, teachers can use the *Active Historical Thinking* teaching methodology to give such an overview of historical events and movements. The idea here is to give students insight into how history *develops*: not just as a discrete series of events, but rather as a continuous process. (Havekes et al., 2012) This process is called historical contextualisation.

Students are placed in groups of two or three students, each sharing an A3 paper with a copy of a timeline printed on it, containing several placeholders, as well as persons of interest. The students are tasked with filling out this timeline using *cards*. This is done in several rounds.

At the start of each round, every group gets an envelope containing the next set of cards. During a limited amount of time, students then have to place the cards from this set on the timeline, in addition to the cards already on the canvas. Disputes may arise here, and with all students having had the same lessons prior to filling out the timeline, this can lead to interesting discussions. This stimulates critical thinking about how events relate to one another.

When students are out of time, the round is evaluated. This discussion is led by the teacher, who assumes a guiding and overseeing role in this methodology: why were certain cards placed at certain points in time? How do they relate to upcoming and active movements? How did the people mentioned on the cards play a role in them? Et cetera.

## 2.3 Example session

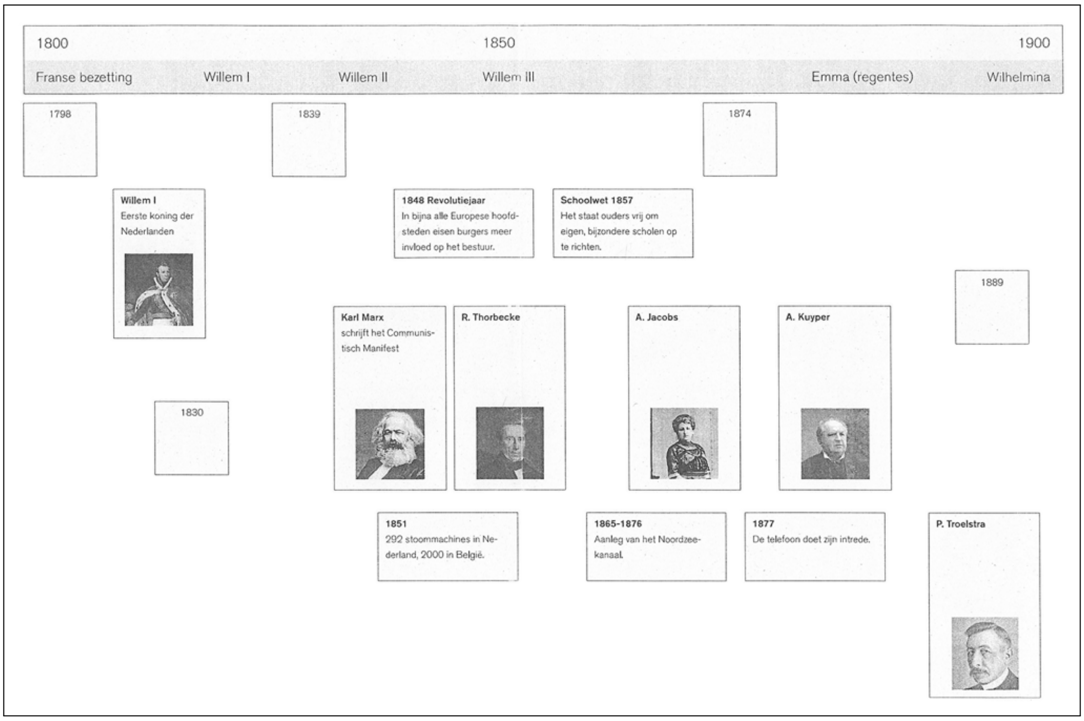

1800
1850
1900
Franse bezetting
Willem I
Willem II
Willem III
Emma (regentes)
Wilhelmina
1798
1839
1874
Willem I
Eerste koning der Nederlanden
1848 Revolutiejaar
In bijna alle Europese hoofdsteden eisen burgers meer invloed op het bestuur.
Schoolwet 1857
Het staat ouders vrij om eigen, bijzondere scholen op te richten.
1889
Karl Marx
schrijft het Communistisch Manifest
R. Thorbecke
A. Jacobs
A. Kuyper
1830
1851
292 stoommachines in Nederland, 2000 in België.
1865-1876
Aanleg van het Noordzeekanaal.
1877
De telefoon doet zijn intrede.
P. Troelstra

*Figure 2.1:* Scan of the canvas for Active Historical Thinking's *'The Netherlands in the 19th century'*.

To illustrate the method, we will take a look at the chronology 'Nederland in de 19de eeuw' (The Netherlands in the 19th century), which consists of four rounds. Figure 2.1 shows the canvas for it.

In the first round, students to place historical events and descriptions of key figures in the timeline. Each of the corresponding cards should be placed in one of the placeholders, mapping directly onto parts covered in lectures. Students are expected to have placed the cards in the right slots within 10 minutes. Afterwards, the round is briefly evaluated: where did the groups place certain cards, and why? Especially in case of dissonance: this creates a cognitive incongruity. (Havekes et al., 2012, p. 84)

The second round, also consisting of 10 minutes, has the students identifying the beginnings of movements and ideologies and place their cards above the years on the timeline. In order to do this, students will have to look closely at the cards on the canvas and link them together. During the evaluation, the teacher can incite discussions about where cards were placed. For instance, the card for socialism is linked to the rise of the radio.

Building on the ideologies placed in the second round, the third round has students connecting keywords to these political or societal movements. This process takes slightly longer than the first two rounds: a total of 15 minutes is allotted. Again, there is an evaluation afterwards, where students debate why certain keywords are associated with certain ideologies.

Finally, the fourth round sees students work with slightly bigger cards, that have a quoted *source* on them. After placing them in the right time, they also need to connect it to at least one keyword from the previous round. Students get approximately 20 minutes to do this, after which discussion follows, where the entire canvas can be taken into account.

See appendix A.1 for the cards and appendix A.2 for the timeline for *'The Netherlands in the 19th century'*.

## 2.4 Design principles

One of the main stakes in the method is *collaboration* between students. Through deliberation and discussion, they create their own contextualisation of the historical events. As we mentioned earlier, this is harder than it may seem: while they may know the facts, students may not always have the required background knowledge readily available.

We may abstract from the historical background altogether and focus solely on the *canvas* the students are working on. In this thesis, we design the system such that teachers can create their own modules, including timeline and cards, thereby delegating the responsibility for the content to the teacher.

In the process of students collaborating on their canvases, the teacher has an overseeing, guiding role. He or she does not necessarily *correct* students, as much as *stimulate their thought process*: why are cards put where they are? How do events relate to one another? Making students question their own knowledge is not an easy process, but needed to get students to progress in their epistemic beliefs. (Havekes et al., 2012, pp. 84–85)

Havekes et al. summarize their work with the following design principles (2012, pp. 87–88):

---

1. Challenge historical knowledge: create a cognitive incongruity
    - Create a historical tension (the past as being different) by providing the students with information that conflicts with their prior knowledge and/or is conflicting in itself.
    - Challenge their knowledge and skills (knowing and doing history).
    - Challenge their epistemic beliefs.
2. Stimulate substantiated considerations
    - Ask the students an (evaluative) question so several reasonable answers are possible.
    - Focus the (evaluative) question on historical contextualisation through explicit attention for time/duration, cause/change and location.
    - Nudge the development of their epistemic beliefs by asking them to formulate alternative answers.
    - Nudge their development of their epistemic stance by assessing their answers on how knowing and doing history are combined.
    - Stimulate the use of prior knowledge through challenging questions.
    - Stimulate thinking about the relationships between facts and concepts by using colligatory concepts.
3. Scaffold students' learning
    - Structure the question through
        - providing students with factual information and an evaluative question at the start of the learning sequence;

    - providing students with relevant colligatory concepts;
- Structure the learning sequence through the learning activities by
    - a clear introduction and giving clear instructions
    - alternating the learning activities (working in pairs or triads, whole classroom discussions and individual tasks);
    - varying the pace of the learning activities;
    - a clear debriefing of the learning sequence, focussing on content, process, historical thinking and goal of the learning sequence.
- Foster the learning of the students through constant guidance:
    - observe and ask questions for justification during their work in groups;
    - react through both responsive questioning and assessing answers during classroom discussion;
- Make the historical thinking of the students visible (brains-on-the table) through debriefing of the whole learning sequence, explicitly addressing knowing and doing history and their epistemic beliefs.
- Give special attention to the second-order concepts of time/duration, space/location and change/cause.

## 2.5 Computer Supported Collaborative Learning

Computer Supported Collaborative Learning (CSCL) is the term used to refer to *knowledge building* in classrooms. (Resta & Laferrière, 2007) In a CSCL process, computers are used to facilitate in the collaboration process. This can be either a synchronous or asynchronous process – that is, both parties can collaborate simultanously (while in the same room, or through e.g. Skype) or 'take turns' if they are remote.

In past research, Lipponen, Rahikainen, Lallimo, and Hakkarainen (2003) have found that the quality of discussion decreased when moving discussion into a CSCL environment, noting that "students do not participate very intensively" and "there is often a lack of sustained and connected discussion" in CSCL environments. Conversely, Fjermestad (2003) found CMC (Computer Mediated Communication) to improve performance in tasks requiring decision-making, attributing it to the lack of social presence.

Even though students work on separate devices, thereby opening up the possibility for moving discussion to a digital platform, we have decided not to, with the above studies in mind. Furthermore, the aims of *Active Historical Thinking* actively promote classroom discussions. Given that students will be sitting next to each other in class, verbal communication is therefore likely to be a more efficient choice, too.

# Chapter 3

# Requirements

Having looked at how the method works as a teaching methodology, we will now attempt to break it down into smaller components, from a technical point of view.

As we discussed, in the original method students collaborate by filling out a timeline on an shared A3-sized sheet of paper, using paper cards to fill in the blank space of the canvas. During this process, they verbally discuss the historical context to determine where to place cards.

In the transition to a digital system, we will be moving the canvas to tablet clients, making use of the rise of interactive personal devices in the classroom. However, the communication between students will still be done verbally.

Three aspects of our design are in need of requirements analysis:

1. The *tablet client* that students will be using on their devices;
2. The *whiteboard client* that teachers will be using on an interactive whiteboard;
3. The *server* that is used to exchange information;

We will go into detail for each of these in the following sections.

## 3.1 Students' tablet client

The students will use their own *tablets* to fill in a digitised version of the timeline canvas. The model of tablets will vary between schools, but will either be an iPad or Android device, so the software should be compatible with both the iOS and Android operating systems.

### 3.1.1 Set-up

When a student opens the application, the first step will be a login process to *authenticate* themselves as a member of a class. This should be done through a straightforward combination of email address and password, linked to a class in the storage back-end. What class they are in influences which sessions they can see – as such, it acts as an access level.

After a successful login, the user is taken to the list of current and past *sessions* they can access. Each of these sessions is a link between a *teaching module* and *class*. The same module may be taught in several classes, but as students need to be physically present for the classroom discussions, there is need for such *instantiation*.

Selecting one of these sessions will take the user to the *group selection* screen. Here, they can pair their device with other groups present. Most modules will specify working in pairs or groups of three students. Any user can join any group, provided the quotum has not yet been met. If none of their group members has created a group yet, one of them should click the 'Add new group' button. This will open a new group, with the creator as the only member. The other student(s) can then join the newly created group.

### 3.1.2 Canvas behaviour

Each of the groups shares a *canvas instance*. This canvas instance is the collection of cards the students need to position — the variable part that is group-dependent. Every such a *card instance* can be either positioned on the canvas, or be on the *stack* of cards. As each of the group members shares a canvas, resultingly, changes by one user should be seen by the other group member(s). We will look into facilitating this on a technical level in the next chapter.

Card instances that have not yet been positioned on the timeline, will be stored on a *stack*. This will be a collapsible overlay on the side of the screen. Students can drag a card off this stack onto the canvas. When this *dragging* stops, the change should be visible on all tablets involved.

This manner of collaboration raises interesting *synchronisation issues*. We will be discussing these in the next chapter.

We should also note that cards can continue to be dragged after their initial placing. The system *does not* automatically check whether cards are in 'correct' positions. Through discussion and collaboration, students should eventually find the 'right' positions themselves.

As mentioned in the previous chapter, each teaching module consists of several *rounds*. Students will only be able to see the cards involved in the consecutive rounds thus far, and depend on the *teacher* distributing new cards onto the group's stack once the teacher is satisfied with the results of the current round. As such, the teacher should be able to *add cards* to the canvas instances of the groups in his or her classes. We will cover this in the next section.

## 3.2 Interactive whiteboard software

Like the students' client, when a teacher opens the application, the first step will be a login process to *authenticate* themselves as a teacher. Again, this should be done through a straightforward combination of email address and password, linking these credentials to teacher privileges in the storage back-end.

### 3.2.1 Aggregation system

Paramount for the teacher's whiteboard software is an *aggregation system* with which the teacher can get an illustrative overview of the canvasses of all the groups. This may be used not just to identify gaps in students' knowledge, but also for demonstration purposes: why have some students decided to place a certain person in a particular time, whereas others place them several years later?

This system does not require many features, but should work intuitively. The idea is that, after opening the module in question, the teacher can select a few cards from the stack to study. These are then assigned a unique colour. For each of the groups, the cards in question will then be represented by a coloured dot in the positions the groups have placed it. If more than one group has placed the same card in the same place, the dot will be bigger and get the amount of groups involved as a caption.

### 3.2.2 Management system

Aside from an instrument to be used in the classroom, the teacher's client software should also function as a management tool. Teachers should be able to create accounts for their students and add them into classes. Furthermore, they should have the ability to create, import and export modules, including adding cards and timeline labels. Finally, they should be able to start and manage the aforementioned sessions, as well as *advance* them to the next round.

## 3.3 Software stack

As the original system was developed by a not-for-profit organisation, it is the wish of the original authors that we design the new system to be freely available as well. For this reason, we will design our approach with *open source* technologies in mind. This means we will be using tools, frameworks and programming languages that are freely available – that is, they do not require a licence fee to use.

While limiting the choice to not include software that is only available commercially, we are hopeful this means the resulting software will be easy to deploy in schools as well.

## 3.4 Schematic overview

Figure 3.1 shows a schematic overview of the distribution of clients in the system. In general, two clients are grouped together, but still function and communicate autonomously.

As shown, the teacher is not grouped directly to any of the students, but gets their information directly from the server. As the groups will share the state of their canvasses with this server, the teacher will be able to see their progress this way.

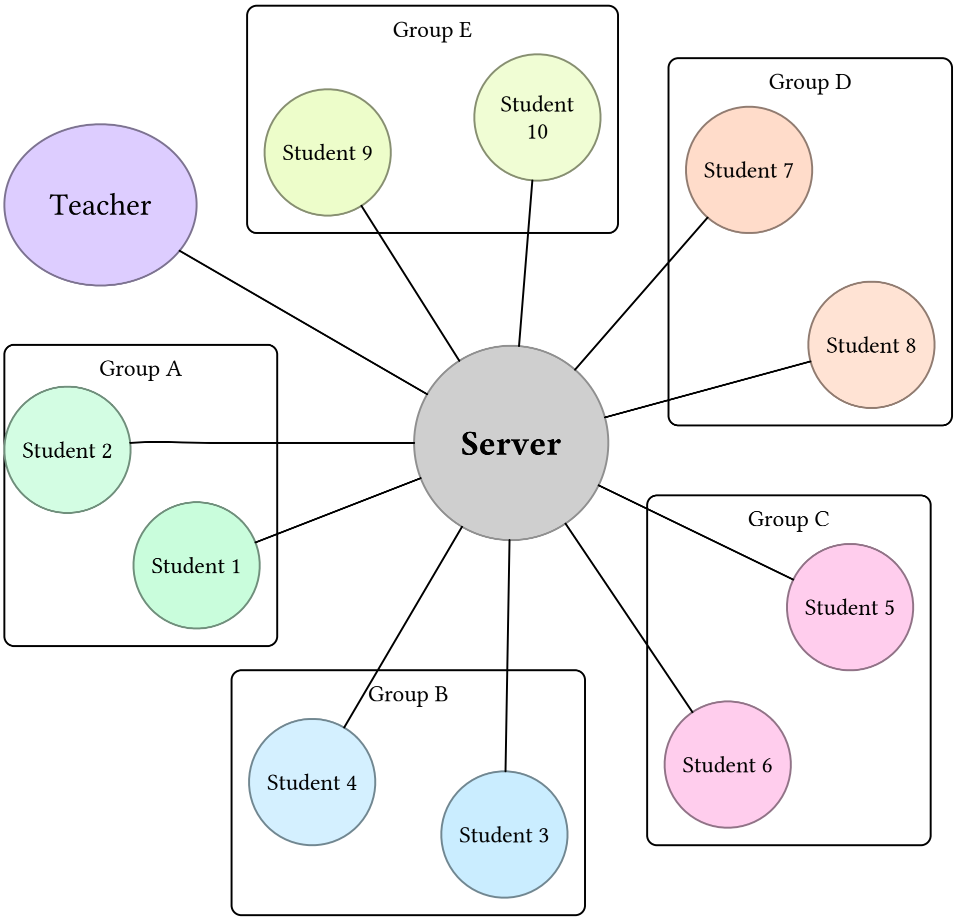


*Figure 3.1:* Schematic overview of client-server communication. Grouped clients are contained in boxes.

# Chapter 4

# Background information

This chapter presents the options we have considered to make the choices that facilitate the prototype implementation we present in chapter 6 on page 29.

## 4.1 Platform: native or web-based application

One of the core decisions in the translation process, is whether we want to build a *native* application, or a *web-based* application. Whichever we choose influences e.g. what communication technologies will be available to us. For example, a web application does not have direct access to Bluetooth or WiFi technologies, so all communication will have to go through a hypertext protocol.

Charland and Leroux (2011) briefly discuss the main advantages and drawbacks of either approach. They note that web-based applications are significantly cheaper to produce due to all platforms sharing the same code. Their biggest drawback is not sharing the performance of their native counterparts, which is notable when doing, for example, 3D image manipulation.

A notable disadvantage of building a native application is that it is constrained to a single platform. While it is possible to share logic code to some extent, each platform has their own UI toolkits. Therefore, the code for the user interface, which typically takes up a vast portion of the code, will need to be written specifically for the platforms in question. Regrettably, this typically results in a lot of similar, but ultimately different, code.

There has been effort in abstracting the individual UI toolkits into platform independent code (cf. Qt, Xamarin), but at the time of writing this is mostly in an experimental phase.

A notable cause for slower responsiveness of web apps, is JavaScript performance and DOM (Document Object Model) construction. As such, optimising it is subject to much research (Martinsen et al., 2013; Ahn et al., 2014). Fortunately, since 2011, this performance has increased dramatically. Efforts by major browser producers, including Microsoft, Google, Mozilla and Apple, have sped up the interpretation and execution process greatly.

Considering their differences, it is no surprise the choice between native and web-based application has great influence over the software stack we will end up using. In terms of programming languages, native apps are generally built using strongly typed programming languages (e.g. C++, Java, Objective-C), while web-based apps are often built using weakly typed scripting languages (e.g. JavaScript, PHP, Python) on top of a webserver.

In the case of web-based apps, developers may also choose to build their application in a strictly typed language that is then served through a webserver. Examples of these are Java on TomCat, or any compiled language through the use of CGI (Common Gateway Interface) to interface with the webserver.

## 4.2 Transmission

Depending on whether we build a native app, we will need to assess which hardware capabilities of the tablets we will be using to *transmit* and *exchange* data. All tablets involved have two means of short-range wireless communication: a Bluetooth stack and a WiFi stack. We will briefly discuss both stacks in this section.

### 4.2.1 Bluetooth: direct pairing

Bluetooth communication works by *pairing* devices with each other: one master device initiates and manages connections with up to seven 'slave' devices. The pairing process is done with explicit consent on both ends through the exchange of a key: the initiating device will provide it, after which the receiving party needs to enter it to confirm the pairing. After this, the devices can then communicate through a continuous, encrypted connection.

A notable advantage of the Bluetooth protocols is that devices are easy discover, as they advertise themselves by name as needed. For our purposes, these could contain the names of the students involved (e.g. 'John Smith's iPad').

As mentioned, there is a hierarchy of devices: at any one time, only one of the devices can be the master device. Only the master device can send messages; the others are only on the receiving end. Through a protocol, the devices can switch roles – i.e. a different device in the network becomes the master. At any time, only one device in the network can be the master device – all others are assigned a 'slave' role.

While all tablets are outfitted with a Bluetooth stack by default, the same might not the case for the teacher's whiteboard — these are often desktop computers outfitted with a specialised projector. This is not a big hurdle to overcome: there are USB peripherals available ('dongles') that can provide the required functionality.

### 4.2.2 WiFi: intranet or internet

The WiFi stack is an *access point* based network solution. This means that, to establish a network, one of the devices involved must take the role of the *routing* device, or router. While this may seem somewhat cumbersome, it has one major advantage: the network is *full-duplex*: it allows for constant communication in both directions.

Typically, using a wireless connection we will have access to not just one other device, but an entire network consisting: either a local network (LAN), or the entire Internet. This is advantageous in that it allows us to easily set up a connection with a central server from all clients.

Using WiFi in the classroom has another advantage: we can use the existing infrastructure. Classrooms have been outfitted with WiFi routers to provide students with an internet connection. Similarly, it means the teacher's computer does not necessarily need to be on the same wireless network: their computer can be on a wired connection on the same subnet.

## 4.3 Network layer

If we choose to build a native application, we will have to decide on what network protocol to use to communicate with the server. There are two common protocols used in conjunction with the Internet Protocol (IP): the Transmission Control Protocol (TCP) and the UDP (User Datagram Protocol). Both can be used to transmit the same sets of data, but come with different properties and guarantees. We will briefly explain both protocols in this section.

### 4.3.1 Connection-based: TCP

TCP is a connection-based protocol, which means that there is a *handshake* before every payload. A connection is opened by sending a SYN (synchronise) packet, which the receiver can then accept through a SYNACK (synchronise-acknowledge) packet. After this, the connection is established: the sender sends their messages in order, and delivery of each message is acknowledged by the receiver using an ACK packet before the next message is sent. Figure 4.1 illustrates this with a sequence diagram.

One of the important takeaways for our purposes, is that the TCP protocol guarantees the payload (data) transferred remains completely intact, as well as in the right order. This may seem trivial, but if this is handled in the network layer using TCP, we may abstract from it altogether in the application layer.

Because of these advantages, TCP has some overhead: it requires setting up a connection before any data can be sent. While this comes with several guarantees that are nice to have, some purposes might not require guaranteed delivery and would instead benefit from the speed gains of a stateless protocol instead.

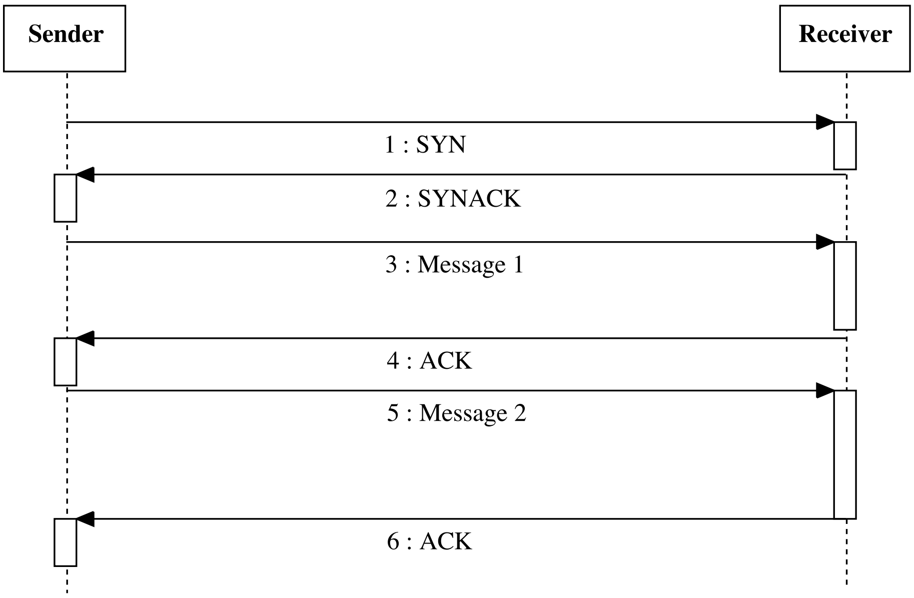


*Figure 4.1:* Sequence diagram showing message passing using TCP. Note the first two messages establish a connection through a handshake.

### 4.3.2 Stateless: UDP

Unlike TCP, UDP is a stateless protocol, which means packets are sent directly to their destination without setting up a connection beforehand. Like TCP, every packet contains a checksum for packet integrity validation. However, there is no acknowledgement of receipt from the receiving end. While this makes for a lightweight protocol, it also limits options after receiving a corrupted message. Figure 4.2 on the following page illustrates this with a sequence diagram.

The stateless nature of UDP makes it ideal and fast, however. It is therefore used for protocols that deem speed to be of the essence, like the Domain Name System (DNS) and DHCP (Dynamic Host Configuration Protocol) used for network lookup and regulation.

Another common use of UDP is found in multiplayer games. Here, a game state is sent several times per second. Especially in games where low-latency is paramount (e.g. games requiring fast responses), the advantages of sending a state in a timely manner outweigh the disadvantages – errors can be handled by the game through succeeding packets.

Here, we come to the essence: UDP is *fast*, but defers the responsibility of handling errors to the application layer. For our intents, we will need to decide whether this is most beneficial.

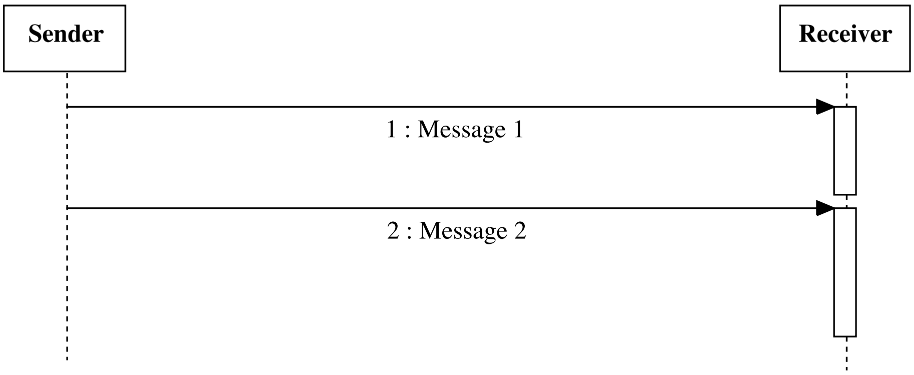


*Figure 4.2:* Sequence diagram showing message passing using UDP. Note there is no connection between messages, nor confirmation of receipt.

## 4.4 Communication

### 4.4.1 Polling

An naïve way to approach the communication, is by having the clients involved *poll* for changes periodically. To give the impression of a real-time system, we would need to poll frequently – at least once every second. This is relatively easy to implement, but quite inefficient: most of the time, there are no changes to send to the polling client ('push'), hence most of the traffic is overhead.

The matter is complicated further by *traffic congestion*: polling too often will result in many packets needing to be exchanged through the routing device and, as we are using wireless technologies, the air. Each of these packets will not be going directly to the intended receiver either: all devices in the direct vicinity will pick up the packets. Of course, all devices except the routing device and recipient will disregard packets not intended for them. However, by its nature this solution is *difficult to scale.*

For our purposes within a classroom, polling is still an option. However, depending on the implementation, it may cause problems in practice if several neighbouring classrooms use similar software at the same time.

### 4.4.2 WebSockets

To achieve real-time communication, WebSockets may be used. This is a relatively new technique: the IETF standardised it per RFC6455 (Fette & Melnikov, 2011), and in September 2012 it received a candidate recommendation from the W3C. It has since been implemented by all major browsers.

Results in previous academic work with WebSockets has been encouraging (Swamy & Mahadevan, 2011; Wessels et al., 2011; Qveflander, 2010), with browser implementations maturing nicely.

#### Setting up a connection

A WebSocket connection finds its inception in an HTTP (HyperText Transfer Protocol) handshake. The request contains headers that a supporting server understands as an *upgrade request.* After this initial handshake, the connection resumes through the WS (WebSocket) protocol. What follows is an example of such an HTTP request from RFC6455:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
```

```
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

As with a normal HTTP connection, it is possible (and indeed recommended) to secure the connection through the use of *TLS* (Transport Layer Security). For HTTP, the resulting protocol is called HTTPS. Correspondingly, for WS the protocol is called WSS.

It is interesting to note that, for security reasons, it is possible to use WSS from an HTTP connection, thereby upgrading security. Conversely, from an HTTPS connection, we cannot use an unsecured WS connection.

### Invocation in JavaScript

A simple WebSocket can be opened in JavaScript by invoking the WebSocket constructor over a URL address, for example:

```
var socket = new WebSocket("https://websocket.url");
```

Like most constructs in JavaScript, we can assign functions to a websocket to be invoked when an event triggers — again, a means of asynchronous programming. According to available documentation in the Mozilla Developer Network (MDN) [1], the WebSocket object has the following event-related attributes:

**onopen** An event listener to be called when the WebSocket connection's readyState changes to OPEN; this indicates that the connection is ready to send and receive data.

**onmessage** An event listener to be called when a message is received from the server.

**onclose** An event listener to be called when the WebSocket connection's readyState changes to CLOSED.

**onerror** An event listener to be called when an error occurs.

The following example illustrates how we can hook up to these events, logging each of them to the console, along with their messages:

```
var socket = new WebSocket("https://websocket.url");
socket.onopen = function() {
  console.log("WebSocket has been opened.");
};
socket.onmessage = function(message) {
  console.log("New message received: " + message);
};
socket.onerror = function(message) {
  console.log("An error occurred: " + message);
};
socket.onclose = function() {
  console.log("WebSocket was closed.");
};
```

[1] https://developer.mozilla.org/en-US/docs/Web/API/WebSocket

## 4.5 Protocol design

As our environment will includes many strings, we will have to make use of serialisation to format messages. This section aims to briefly explain two common *human readable* serialisation techniques: JSON and XML. Both handle international text well: they fully support Unicode encodings.

We will not be taking *binary* serialisation techniques into consideration, as their use is limited to native applications.

### 4.5.1 JSON

JSON (JavaScript Object Notation) is a common serialisation technique, originating in JavaScript. Because of its origins in a scripting language, a major advantage of JSON is its human-readable syntax, while still keeping overhead to a minimum. Its standardisation by RFC7159 (Bray, 2014) has led to many implementations in both loosely and strictly typed languages.

Even though JavaScript is a *loosely-typed* language, JSON supports strict typing. Types include integers, floats, strings (encapsulated between double quotes), booleans (literals `true` and `false`), arrays and objects. A notable advantage of this is that JSON documents do not require a complex data structure for their representation in a programming language. This in contrast to *DOM* structures that SGML derivatives utilise (e.g. XML, HTML).

While originally a strict subset of JavaScript, JSON has been extended upon since to allow full encoding in Unicode, allowing JSON documents to be encoded in UTF-8, UTF-16 or UTF-32.

In JSON, arrays are a sequence of elements delimited by commas and encapsulated between brackets `[` and `]`, with an implicit zero-based index:

```
["apples", "oranges", "bananas"]
```

Objects, on the other hand, are keyword-based *mappings*. Colons `:` separate keywords from their values, and commas delimit the key-value pairs. Objects are encapsulated between braces `{` and `}`. Note that, while JavaScript does not require it, keywords are mandatorily encoded as strings in JSON:

```
{"name": "Douglas Crockford", "achievement": "First specified JSON"}
```

Note that objects and arrays can be nested, in both ways. To illustrate this, please see the example below for an encoded object, strings, integers and array.

```
{
  "name": "Radboud University",
  "location": "Nijmegen",
  "age": 91,
  "num_students": 19200,
  "studies": [
    "Biology",
    "Chemistry",
    "Computer science",
    "Information science",
    "Mathematics",
    "Physics"
  ]
}
```

### 4.5.2 XML

Another common serialisation technique is the *Extensible Markup Language* (XML), most notable for its use in the exchange of data over the internet. Like HTML, the main markup language for the web, it has its roots in the *Standard Generalized Markup Language* (SGML). XML was designed to be a subset of SGML, using similar syntax, but less features to ease implementation of a compliant parser.

XML represents data as a *tree* of *nodes*, each containing either a textual body or another tree. Each node is encapsulated between a start tag and end tag, both encapsulated between <and >. Furthermore, start tags may contain *attributes*. For example:

```xml
<province name="Gelderland">
  <city>Arnhem</city>
  <city>Nijmegen</city>
</province>
```

Here, Arnhem and Nijmegen are understood to be instances of type `city` belonging to a parent `province` named Gelderland.

Like JSON, XML is fully Unicode compliant. Much like SGML, its parser is unforgiving for violations of its syntax rules: the document will not continue to be parsed if an error occurs. This is in contrast to HTML5, which has been designed to be lenient in this regard and has well-defined specifications for such edge cases.

Serialising a node tree, similar to the object used in the JSON example, results in the markup code below. Note that an XML document does not explicit contain information on *typing*; this is done in a separate XML Scheme Document, if needed. Such an XSD file can define a document scheme from 19 primitive data types, including integers, strings, dates and URLs, but discussing these is outside of the scope of this thesis.

Unlike JSON, there is no primitive for arrays. Rather, array-like structures can be represented by making all of them children of the same parent element. In the example below this is done by adding `study` elements to the `studies` node.

Finally, elements without a body may be used to represent optional boolean expressions. Rather than using a closing tag, the start tag is closed with an extra forward slash: `/>`. The `is_public` node in the example below is an instance of such an *empty tag*.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<university>
  <name>Radboud University</name>
  <location>location</location>
  <age>91</age>
  <num_students>19200</num_students>
  <studies faculty="science">
    <study>Biology</study>
    <study>Chemistry</study>
    <study>Computer science</study>
    <study>Information science</study>
    <study>Mathematics</study>
    <study>Physics</study>
  </studies>
  <is_public />
</university>
```

When working with XML documents, it is common practice to work with a *Document Object Model* (DOM) to represent the node tree and its elements, ensuring validity of the resulting document. Of course, an XML document may be created by writing strings directly to file, too. However, as the language is not regular, for processing a document the use of a DOM parser in some form is unavoidable.

## 4.6 Synchronisation problems

As we are dealing with an interactive canvas that involves several users, we anticipate concurrency issues may arise when communicating with the server. Fortunately, canvas user interaction will be limited to dragging cards from one position to another, which reduces the synchronisation problem to that of finding out which is the *actual reality*.

There are several ways to deal with the issue. We will briefly discuss three of them.

### 4.6.1 Preventive measures

One way to deal with synchronisation issues, is to try to prevent them from ever occurring. For example, for our purposes, we might implement an *exclusivity lock* that locks a card as soon as the first user interaction takes place, preventing another user to interact with it. This lock would then be lifted once the other user stops interacting with it, i.e. when they lift their finger from the screen. Figure 4.3 illustrates this.

While this will solve the symptom of a user seeing their changes overridden by a concurrent action, it may cause another problem: a *race condition* may be triggered. If two students interact with the same card at once, who will get the exclusivity lock? Technically, one of them will always be first by a few milliseconds, but ideally we will not have to wait for a server to give a verdict before interaction can begin.

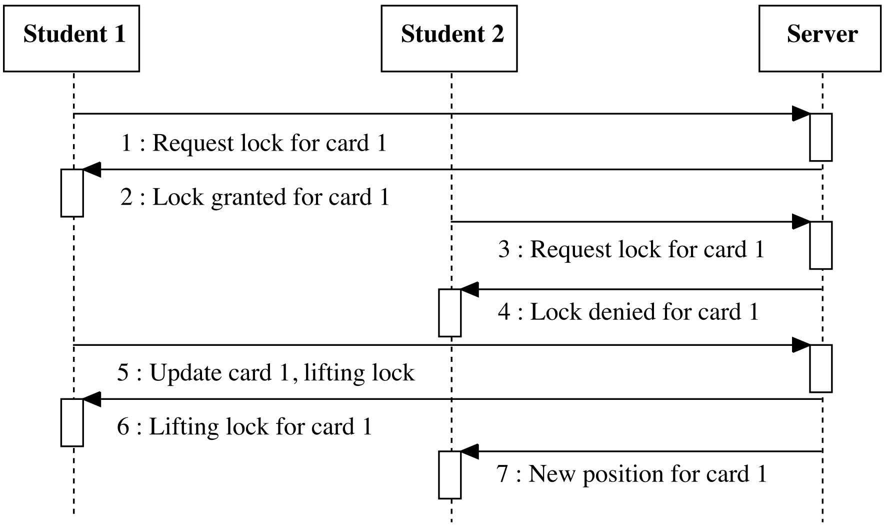

Student 1
Student 2
Server
1 : Request lock for card 1
2 : Lock granted for card 1
3 : Request lock for card 1
4 : Lock denied for card 1
5 : Update card 1, lifting lock
6 : Lifting lock for card 1
7 : New position for card 1

*Figure 4.3:* Sequence diagram showing a possible protocol for two clients trying to obtain an exclusivity lock.

### 4.6.2 Linear message parsing

Another way to deal with concurrent messages, is to look at them linearly: whatever message comes first, is processed first. This is fairly easy to implement: messages are queued in a first-in, first-out (FIFO) queue. After processing the message at the top of the queue, the results are sent to other clients. If another client had interacted with the same element on the canvas, their result will be overridden. Of course, this means that succeeding messages in the queue will override this result *again*, however.

Key here is how the user expects the mechanism to work. If we process messages linearly, depending on how many concurrent changes there are, elements may be re-positioned several times on a user's canvas.

Ideally, students first discuss where cards should be located before they move a card. However, it is likely students will move cards around for illustrative purposes: consider a student explaining the context of a particular historical event, for example. As such, it is likely two clients will both be dragging the same card from its original position to a different position on the canvas.

As soon as the first student lets go of the card they are dragging, the change will be pushed to the server and thereby to the other student – overriding the position of their local instance of the card. However, as they are still interacting with it, the change event may be queued, depending on the implementation. Then, after the second student lets go, the change done by the first student will, perhaps unintentionally, be overridden. Figure 4.4 illustrates this.

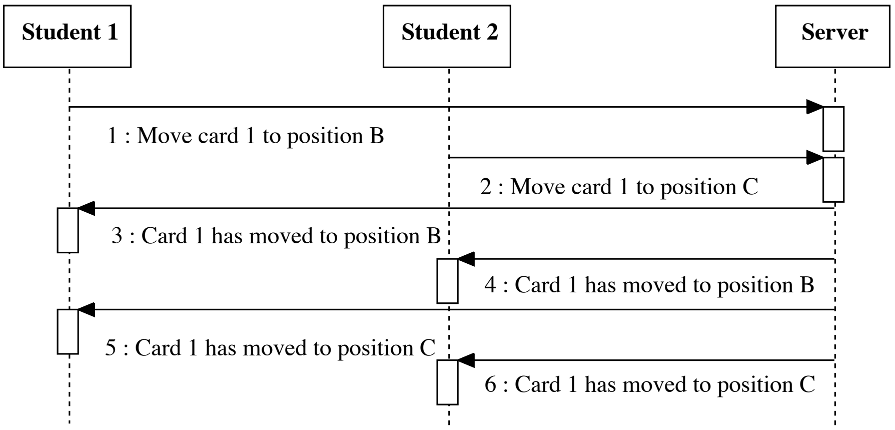

Student 1
Student 2
Server
1 : Move card 1 to position B
2 : Move card 1 to position C
3 : Card 1 has moved to position B
4 : Card 1 has moved to position B
5 : Card 1 has moved to position C
6 : Card 1 has moved to position C

*Figure 4.4:* Sequence diagram showing a possible protocol for two clients moving the same card.

### 4.6.3 Message pre-conditions

The third method we are considering, is the assignment of a *pre-condition* to every card update message. In our case, this pre-condition will be the position the card was in when the user started interacting with it. Effectively, this means the first change to be submitted will be considered the actual reality — a concurrent change will have an outdated pre-condition and will therefore be discarded.

Consider the situation we looked at in section 4.6.2. If this occurs when using pre-conditions, the user who first submitted a change will not see the card jump to another position shortly after. Rather, the second user to change the card will see their change ignored, as the pre-condition to their change is no longer valid. Figure 4.5 illustrates this.

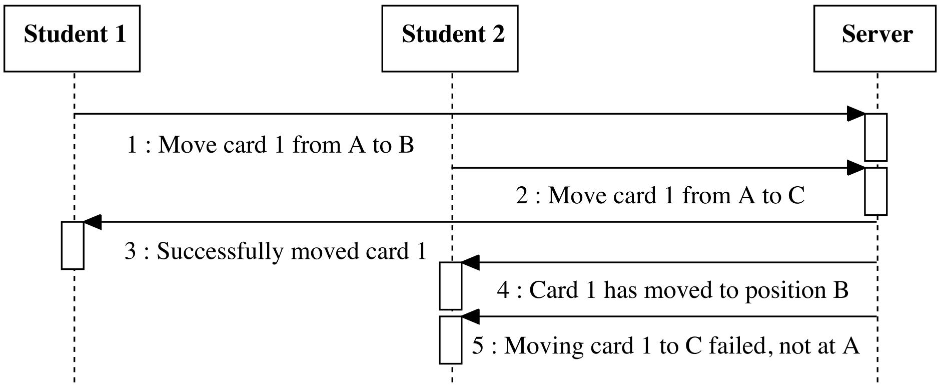

Student 1
Student 2
Server
1 : Move card 1 from A to B
2 : Move card 1 from A to C
3 : Successfully moved card 1
4 : Card 1 has moved to position B
5 : Moving card 1 to C failed, not at A

*Figure 4.5:* Sequence diagram showing a possible protocol for two clients trying to move the same card.

## 4.7 Asynchronous programming

Traditionally, statements in an imperative programming language are typically executed one-by-one: only after a statement has finished executing, is the next one executed. The net result is a program that is always 'in sync' with what the programmer dictated. Hence, it is called *synchronous programming*.

Consider the following synchronous program, retrieving a record set from a database:

```
function getDataSynchronously(dbconn) {
  console.log("Querying itemset");
  var set = dbconn.query("SELECT * FROM items");
  while (var row = set.fetchRow()) {
    appendResult(row.title, row.value);
    console.log("Result #" + row.id + " appended");
  }
  console.log("End of function");
}
```

Its statements are executed sequentially, hence the console output after invoking the `getDataSynchronously` function will be:

```
Querying itemset
Result #1 appended
Result #2 appended
...
End of function
```

Synchronous programs are relatively easy to understand, because their execution flow is fairly straightforward: the program does not continue until the previous statement has been fully executed. A downside to this is that the program effectively *blocks* if a statement takes longer to execute: there is no concurrency. Traditionally, such bottlenecks have been disk I/O, and communication with other systems for specific services, e.g. a database server. As a direct result, such operations could render the user interface (UI) for the program unresponsive for noticeable time.

When faced with these problems, programmers often make use of *threading*, a concurrency technique used to split the program's execution flow. Typically, a computationally intensive program will run its user interface in a separate thread from the main program.

Threading is not always an option. For this thesis, we are using a web-based environment, which typically run single-threaded, supervised by a browser. As a result, any synchronous client-server communication will delay or block any interaction the user tries to invoke on the canvas. In real-time systems, such communication may take place several times a second, noticeably hindering the user.

Fortunately, we can mitigate such blocking by making use of a concurrency technique applicable to web browsers: *callback* functions. We pass an extra argument to blocking functions: a pointer to *another* function, to be executed after finishing the blocking function.

```
function getDataAsynchronously(dbconn) {
  console.log("Querying itemset");
  dbconn.query("SELECT * FROM items", callback);
  console.log("End of function");
}

function callback(set) {
  while (var row = set.fetchRow()) {
    appendResult(row.title, row.value);
    console.log("Result #" + row.id + " appended");
  }
}
```

In our example, the database query function now calls the processing function only when it has fully retrieved the requested recordset from the database. Effectively, this schedules the processing of the resulting payload for a time when it is available. In the mean time, the program will resume its normal flow — here, this will likely be awaiting user input. This is called *asynchronous programming*.

Because communication is now no longer blocking the program's execution flow, our console output will change:

```
Querying itemset
End of function
```

```
Result #1 appended
Result #2 appended
...
```

Some programming languages allow the programmer to pass an anonymous (lambda) function as the callback argument, typically leading to fewer code. A drawback of this is that it may *obfuscate* the execution flow of a program: at first glance, the code may appear to be executed sequentially, whereas it is in fact still executed as a callback. Example 4.7 illustrates this: while it syntactically mostly resembles example 4.7, it is semantically equivalent to example 4.7.

```
function getDataAsynchronously(dbconn) {
  console.log("Querying itemset");
  dbconn.query("SELECT * FROM items", function (set) {
    while (var row = set.fetchRow()) {
      appendResult(row.title, row.value);
      console.log("Result #" + row.id + " appended");
    }
  });
  console.log("End of function");
}
```

# Chapter 5

# Design choices

This chapter details on the choices we have made to come to an implementation, given the options we considered in the previous chapter.

Please note that the choices in this chapter are not outlined in any particular order. As one might expect, any particular choice my have consequences for another – as such, they are intertwined.

## 5.1 Platform: web-based

Decisive in our decision to go for a web-based approach, is the intention to share it across three platforms: tablets running either iOS or Android, as well as a teacher system to run on an interactive whiteboard ('digiboard'). While, in principle, the latter is a system running Windows, it is subject to several restrictions. In practice, this means most of its applications run in a web browser. For us, this means we would be building a web framework regardless. As such, it makes sense to re-use this code base to build the tablet client for iOS and Android.

In the previous chapter, we discussed some disadvantages of web-based applications, notably the performance of JavaScript execution. As we mentioned, this area has been subject to much research, and as such performance has increased considerably since 2011. In 2013, this led us to conclude performance in a 2D canvas environment is sufficient for our needs. The performance issues mentioned by Charland and Leroux were found to be no longer an issue per Android 4.1 and iOS 6.

## 5.2 Network: TCP

As mentioned in the previous chapter, TCP comes with notable advantages. Firstly, because it is a connection-based protocol, every connection has a *state*. This means we will only need to authenticate the channel once, at the start of the connection. Assuming we secure the connection properly using TLS, this means we not only have a safe connection, but also prevent certain overhead that would come with providing an authenticated connection on a stateless protocol (e.g. security tokens on UDP).

Furthermore, the order and integrity guarantees TCP comes with will be paramount to our method of providing real-time communication to the software. As we will see later in this chapter, techniques like WebSockets rely on working with a sole connection per client, which furthers our need for a state-based protocol.

## 5.3 Transmission: WiFi

Given the choice between Bluetooth and WiFi, we initially leaned towards the former, as it generally consumes fewer energy. However, as the Bluetooth stack only provides communication between one master and one slave device, is not really viable as a means of real-time communication between more than one device. Even if we only account for two devices (i.e. students), they will still need to alternate the role of master (sender) and slave (listener). This, of course, leads to much overhead, as the stack was not really intended for this purpose.

An alternative proposal would be to set up the teacher's whiteboard computer as a local server, connecting every tablet to this computer, making the tablets master devices (senders) and the server the slave (receiver). Unfortunately, this is not a feasible solution for several reasons. Any network may only consist of up to only *eight* devices, one of which is a master.

This is complicated further by the need for a teacher's computer to send new cards to the students' tablets after each round, necessitating it to be in *master* mode as well. Furthermore, the Bluetooth stack does not allow a device to be paired with more than one master at a time. We could probably work around this by implementing some form of time slicing, but all in all, a WiFi-based approach looks to be the better, much more scalable solution.

## 5.4 Communication: WebSockets

To facilitate the collaboration process, it is crucial that collaborating clients see each other's changes as they happen: we want the canvases to be updated in *real time*. Thus, it is imperative that clients continuously exchange information. While most groups will consist of two individuals, there will be at least three clients involved in every group's session: two students and one teacher, as we showed in figure 3.1 on page 11.

As we have decided to build a web-based application, application data is essentially retrieved over the HTTP protocol. However, HTTP in and of itself does not provide for real-time communication: after the payload has been delivered, the TCP connection is closed.

So how can we do better? Ideally, we set up a communication channel to the server for every client, such that it is both *persistent* and *full-duplex*. This means the connection will remain open after the initial payload has been transferred, and that communication on it works both ways.

Effectively, this delegates the polling process to the hardware: when a new message arrives on this open channel, it will trigger an interrupt in the operating system. In turn, the operating system will then delegate the message to the application. This means, however, that the application does not have to actively poll for messages: it can handle the calls asynchronously.

Considering the bulk of the clients will be tablet computers, we have to take energy consumption into consideration. Obviously, if the device has to handle more traffic, it will consume more energy. This is another vote for the passive, persistent connection.

This is where WebSockets come into play. As we mentioned in the previous chapter, this is a relatively new, but fully supported technique that can upgrade a regular HTTP connection to a WS connection. Not only does this achieve our needs, it also does this in such a way that it plays nicely with firewalls, as communication will continue over HTTP's port, 80, or the HTTPS port, 443.

As such, we will design our application such that it communicates its basic pages over HTTPS, and once we enter the interactive component, we will initiate a WebSocket connection to exchange data real-time.

## 5.5 Protocol design: JSON

Both JSON and XML are excellent techniques for data serialization and are well supported in all modern browsers. When working with JavaScript, however, JSON has a major advantage in that it does not require a Document Object Model (DOM). This means it has fewer overhead: objects can be accessed directly, instead of through function calls on the DOM. As JavaScript is a (generally) an interpreted language, this has some nice speed gains over XML.

Seeing as we will be developing a web-based application, the user front-end will be working with JavaScript. As such, we have chosen to implement our protocol using JSON serialisation as well.

We intend for the protocol to follow a *message passing* pattern. This means a client may request the server to, for example, look up the list of active sessions, leaving the server to do the actual querying. Similarly, a server may send a set of new cards to the client, leaving the lay out process to the client. Most notably, however, is the simple passing of *deltas* between server and clients: the moving of cards from position A to B. Again, only the position, i.e. container and coordinates, are transferred. The actual moving on the client side will be computed on the client side.

## 5.6 Synchronisation: preconditions

As more than one client can simultaneously edit the same item on the canvas, concurrency issues may occur when communicating with the server. We could argue that, considering the limited amount of concurrent clients working on the same canvas, the worst thing to happen is a *race condition* – a slight inconvenience for one of the users involved. However, without a protocol these may lead to the kind of unexpected behaviour that causes confusion, such that it distracts or even removes the user from the mindset that they are currently in, losing immersion.

We expect that, while discussing, both students may be dragging the same card around, if only for illustrative purposes, but should not cancel each other out accidentally.

In the previous chapter, we have seen various methods of dealing with potential concurrency issues. When we look at how students collaborate, we believe assigning pre-conditions to messages has the most desirable results. Applying this method will reject succeeding messages if their starting conditions are no longer met – for instance if another student has already moved the same card. Recall, we illustrated a similar process in figure 4.5 on page 21.

# Chapter 6

# Implementation

This chapter goes into detail on several aspects of the implementation process, notably the storage design and user interface principles. We will not show source code in this chapter. For reference, we have included listings of source code headers in Appendix D.

## 6.1 Entity identification

Before starting work on the implementation, we must first assess what entities we will be dealing with: we must make a *model* of the entities and their relations.

We identify the following entities in *Active Historical Thinking*:

**Users** are the teachers and students using the system. They each have an e-mail address to uniquely identify them with, as well as a name with which they can be identified by fellow classmates or collegues.

**Modules** are the overarching entities for the individual lessons: aside from a title and description, they have an entity relation to sessions, labels, types, cards, and card instances.

**Module sessions** are the instantiations of modules, linking a module to a class of students.

**Groups** are the entities with which students are linked in a module session. For identification, each of the groups have an automatically generated name, using the first names of the students, e.g. "John and Mary".

**Group membership** is the entity relationship between a user and a group.

**Cards** are the canvas items: rectangular objects with a caption or image, varying in size. Cards belong to a particular round; those belonging to round 0 are positioned by the teacher, while the others are mutable by the students instead.

**Instances** are the instantiations of cards, linking a card to a particular group. Hence, they also specify the position a card has on the canvas or stack for a certain group.

**Labels** are captions positioned above the canvas by the teacher, to indicate a year or time period. Examples are '1789', 'Early Middle Ages', 'Queen Regent Emma'.

**Types** specify the kinds of cards. They are defined by module and define e.g. the size and colour of cards.

**Type behaviour** specifies what can be done with a type of card. Depending on the round, a card type can have different behaviour, e.g. a card may be draggable in round 2, but round 3 may extend the reach of the card to the label area.

## 6.2 Database design

To store all the data involved, we have chosen to make use of a relational database. Each of the entities described in section section 6.1 on the preceding page will be mapped to their own table, as shown in figure 6.1.

Aside from the entities mentioned, we include two artificial tables: one used for error logging (`log_errors`) and one for storing client-server sessions, storing e.g. authentication (`user_sessions`). Both are included in figure 6.1.

We have chosen to implement the database in MariaDB, an open source database system based on MySQL. To make re-using existing database servers possible, we will use a database abstraction layer that also facilitates the use of other popular open source relational database servers, e.g. PostgreSQL and SQLite.

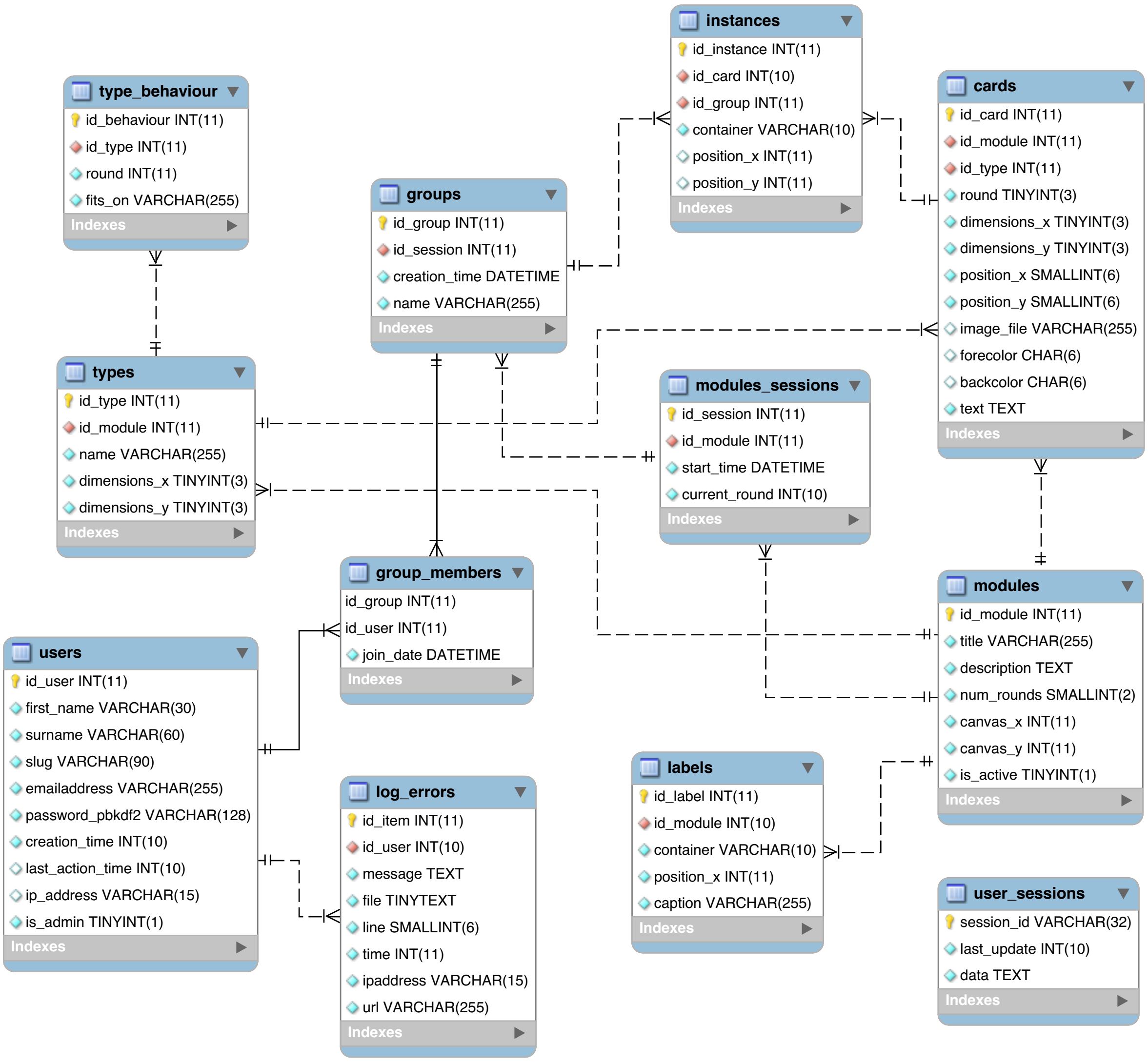


*Figure 6.1:* Schematic displaying the database design. Primary keys are identified using a yellow key; foreign keys use a red diamond.

## 6.3 Server design

To facilitate communication between clients, we will use a centralised system, as opposed to a distributed (decentralised) system. While both have their merits, a centralised approach makes most sense for our purposes. As all communication will go through a central server, this makes pairing devices to groups easier and more persistent. Furthermore, a teacher will be able to easily aggregate the data without interrupting the individual devices.

Using the database as its storage back-end, the server keeps track of the states of canvases of all groups involved. Hence, if a group is formed, the server can quickly send the entire state (including cards, labels, stack, etc.) to each of the users involved. Similarly, if a device accidentally disconnects, they can resume where they left off after a handshake with the server.

These states may be manipulated by users: students can modify the card instances for their own groups, and teachers can make more cards available. As we mentioned in previous chapters, this will be done through *messages*: simple JSON-encoded objects detailing the modifications made. All of these messages will pass through the server. After the server has applied the messages to the state as needed, they will send the results to all parties involved.

For instance, if user A changes position of a card from position P to Q, their client will send a message detailing this change to the server. Assuming the card has not yet been moved from its position P, the server then applies this change, and subsequently distributes the change to the device for both user A and B. This has the added benefit of notifying A their change was successful.

## 6.4 Protocol design

In order to make use of the message passing pattern in an effective manner, we need to define a protocol for all client-server communication. As the messages passed are limited in scope, we can suffice with making an exhaustive list of all possible messages, leaving us with specifying how they are formatted.

As per the format, every message will be an object serialised in JSON encoding, exchanged over the TCP/IP stack. Request messages pass a 'call' parameter to identify how the other attributes of the message object should be handled. Response messages do not contain such a parameter; the server handles messages synchronously, hence responses are guaranteed to be in the same order as the requests are sent.

Below is a list of all request messages exchanged through the protocol. For each of the calls, an example request (=>) and response (<=) message is included. To improve legibility, the messages are indented here. In the actual application, this is not the case.

**authenticate** *{ cookie }*

Authenticates the socket connection based on the session-id in a user's cookie.

```
# Client to server
=> {"call":"authenticate", "cookie":"PHPSESSID=1234...ABCD"}
# Server acknowledges to client (empty = success)
<= {}
```

**announcement** *{ msg }*

Server broadcasts an announcement to be displayed to the user by the clients.

```
# Client sends announcement message to server
=> {"call":"announcement", "msg":"Hello world!"}
# Server acknowledges to client if they are a teacher.
<= {}
# Server broadcasts announcement to all other clients.
<= {"call":"announcement", "msg":"Hello world!"}
```

**error** *{ msg }*

Server reports error in response to an action issued by a client. This error is only sent to the client in question.

```
# Client to server
=> {"call":"authenticate", "cookie":"PHPSESSID=1234...ABCD"}
# If the client's session is not authenticated, an error message is
    sent back:
<= {"call":"error", "msg":"Session is not authenticated."}
```

**list_sessions** *{ }*

Server returns all sessions open on the server.

```
# Client sends enquiry to the server.
=> {"call":"list_sessions"}
# Server returns list of sessions.
<= {"sessions": {"1": {
    "title": "Session title goes here",
    "description": "The description for the module goes here."
   } } }
```

**reload_sessions** *{ }*

Forces a server to reload its session from database, disconnecting all clients. Can only be initiated by a teacher.

```
# Client sends reload command to server
=> {"call":"reload_sessions"}
# Server acknowledges to client if they are a teacher.
<= {}
```

**register_session** *{ id }*

Registers a client's connection to a particular session.

```
# Client expresses intent to join session 1
=> {"call":"register_session","id":"1"}
# Server acknowledges
<= {}
```

**unregister_session** *{ }*

De-registers a client's connection from a session.

```
# Client expresses intent to leave session 1
=> {"call":"unregister_session","id":"1"}
# Server acknowledges
<= {}
```

**list_groups** *{ }*

Returns all available groups for the current session as an array.

```
# User Alice calls 'list_groups'
=> {"call":"list_groups"}
# Server responds with the list of groups, noting she's a member of
    group with id=1.
<= {"groups": {
    "0": {"id": 1, "name": "Alice and Bob", "is_member": true},
    "1": {"id": 2, "name": "Charlie and David", "is_member": false}
   } }
```

**create_group** *{ }*

Creates a new group for the current session. Current connection automatically joins the group and lends its user's name to the group. This call is typically followed by a get_canvas request.

```
# User Alice calls "create_group"
=> {"call":"create_group"}
# Server creates new group for Alice and adds her to it.
<= {}
```

**join_group** *{ id }*

Current connection joins the requested group. If the user's name is not present in the group's name, it is appended to it. This call is typically followed by a get_canvas request.

```
# User Bob joins the group session he shares with Alice.
=> {"call":"join_group", "id":"1"}
# Server adds Bob to the current session.
<= {}
```

**get_canvas** *{ }*

Retrieves a serialised version of canvas data for the currently joined group session. This includes not only the cards laid out on the canvas, but also timeline labels and cards on the stack.

```
# Client requests a full listing of the current session.
=> {"call":"get_canvas"}
# Server responds with a full serialisation of the session.
<= {
    "moduletitle": "Example module's title",
    "dimensions": {"width": "106", "height": "55"},
    "labels": {
        "year": {
            "0": {"id_label": "1", "id_module": "1",
                  "container": "year", "position_x": "15",
                  "caption": "1800"},
            # [...]
        },
        "context": {
            "0": {"id_label": "4", "id_module": "1",
                  "container": "context", "position_x": "15",
                  "caption": "French occupation"},
            # [...]
        }
    },
    "cards": {
        "0": {
            "id_card": "10", "id_module": "1", "id_type": "2",
            "round": "0", "dimensions_x": "12", "dimensions_y": "2",
            "position_x": "4", "position_y": "2", "image_file": "",
            "forecolor": "000000", "backcolor": "ffffff",
            "text": "Willem I<br>First king of The Netherlands"
        },
        # [...]
    },
    "instances": {
        "0": {
            "id_instance": "1", "id_card": "5", "id_module": "1",
            "id_type": "1", "round": "1",
            "container": "stack", "dimensions_x": "12",
            "dimensions_y": "2", "image_file": "",
            "forecolor": "000000", "backcolor": "ffffff",
            "text": "First constitution"
        },
        # [...]
    }
}
```

**update_instance** *{ id, old_position, new_position }*

Updates the state for a particular instance of a card in the current group session. Both the old position and new position are sent: in case conflicts arise, the old position acts as a precondition.

```
    # A client moves a card from the stack to the grid...
    => {"call":"update_instance",
        "old_position": {
            "container":"stack", "position_x":"0", "position_y":"0"
        },
        "new_position": {
            "container":"grid", "position_x":"58", "position_y":"28"
        } }
    # Server acknowledges the change ...
    <= {}
    # ... and passes it on to other clients in the same group.
    <= {"call":"update_instance",
        "old_position": {
            "container":"stack", "position_x":"0", "position_y":"0"
        },
        "new_position": {
            "container":"grid", "position_x":"58", "position_y":"28"
        } }
```

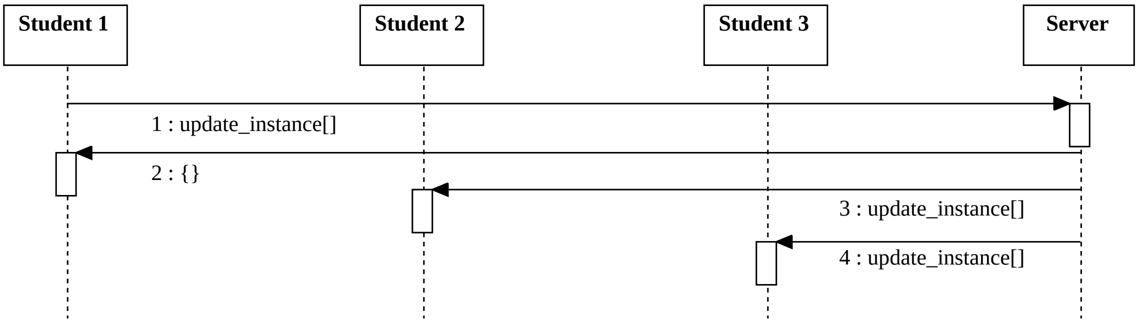


*Figure 6.2:* Schematic representation of message passing for the update_instance call.

**advance_round** *{ new_round }*

Advances a session to the next round. The new_round parameter is passed to prevent accidentally advancing by several rounds due to user interaction.

```
    # Teacher tells server to advance to the second round.
    => {"call":"advance_round","new_round":"2"}
    # Server sends every client a list of card instances to be added to the
    # stack. Note that each group has their own set of card instances.
    <= {
        "instances": {
            "0": {
                "id_instance": "25", "id_card": "14", "id_module": "1",
                "id_type": "1", "round": "2",
                "container": "stack", "dimensions_x": "12",
                "dimensions_y": "2", "image_file": "",
                "forecolor": "000000", "backcolor": "ffffff",
                "text": "Socialism"
            },
            # [...]
        }
    }
```

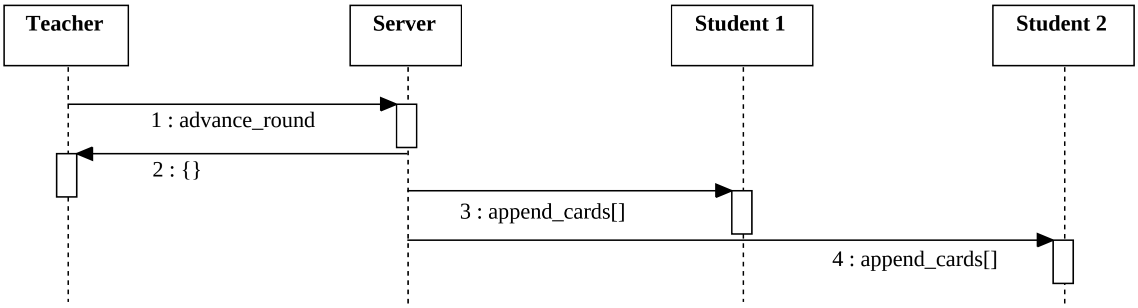


*Figure 6.3:* Schematic representation of message passing for the advance_round call.

## 6.5 Programming languages

Our prototype has been implemented using PHP on the server side and JavaScript on the client side. In accordance with the requirements, we chose an open source language for the server software. PHP is freely distributable and available cross-platform, so the server software may be deployed to schools without issue. As PHP is an interpreted language, the software does not require any platform-specific re-compilation.

The choice for JavaScript on the client side is inherent to developing a web-based application. While our PHP back-end generates some HTML, the majority of the interface is generated through JavaScript and DOM manipulation.

## 6.6 WebSocket abstraction layer

As we have seen in section 4.4.2 on page 16, the WebSocket protocol by itself is rather crude. For our purposes, we would therefore like to abstract from the basic WebSocket events by adding an abstraction layer on top. We will do this by implementing our message passing calls on top of the `onmessage` event, using the JSON encoded protocol we described earlier.

Per the protocol, every incoming message will be a serialised object containing a *call* attribute. These messages will be parsed by our abstraction layer, delegating the message object to the relevant function based on the *call* attribute. By centralising this delegation, we simplify the calls to our individual call functions: instead of having all functions 'listen' to all messages, only relevant messages will be delegated from a central dispatcher.

### 6.6.1 Client-side: JavaScript

WebSockets saw their inception in JavaScript, so we will not be requiring an extra framework to work with them. To illustrate the client-side implementation of our abstraction layer, its function headers are printed below:

```
function Socket(address, open, debug)
  Socket.prototype.send = function (event, params, callback)
  Socket.prototype.on = function (event, listener)
Socket.prototype.unbind = function (listener)
Socket.prototype.unbindAll = function ()
Socket.prototype.close = function ()
```

After a Socket object is constructed, other functions may add an *event listener* to it by invoking the on method. Typically, such a call consists of two arguments: the expected call and a function the message needs to be passed to. If no listener is added for a particular message event, the message will be dropped when it arrives.

To illustrate this, the following snippet will listen to two event messages: the first line invokes appending cards after receiving them, and the second one invokes updating cards when a *delta* message is sent:

```
socket.on('append_cards', this.drawer.push_cards);
socket.on('update_card', this.canvas.update_card);
```

Similarly, sending a message can be done through the `send` method, which uses three arguments: the call, its parameters, and an optional callback function to be executed after receiving a reply. For example, this may be used to show a list of active sessions to the user after requesting the list.

Internally, sending a message will serialise the first two arguments (both the call and its parameters) into a message object serialised using JSON. We will also add a callback to the 'waiters' queue — if the programmer has not provided one, an empty one will be passed.

By convention, every message the client sends will be replied to by the server. This may simply be an acknowledgement, or a list of objects requested by the previous message — for example, the list of available cards on the canvas.

### 6.6.2 Server-side: PHP and Ratchet

To achieve WebSocket communication in our PHP server back-end, we have opted to use the Ratchet framework (`http://socketo.me/`). Its modular approach means we can readily build our own server on top of the foundations laid by Ratchet. This means our server classes only have to deal with the actual incoming and outgoing JSON payloads, while the actual connections with clients are handled and abstracted by Ratchet.

Discussing the class structure of the server is outside the scope of this thesis, but the relevant headers are attached for reference in Appendix D.2.

As the server keeps a state of all canvases through a database, we have outfitted it with model classes that map directly to tables in the database. Their headers are attached for reference in Appendix D.1.

## 6.7 User Interface design

The user interface has been designed to be analogue to the original teaching methodology. The canvas lay-out uses a square-based grid to position cards, but the initial set of cards (round 0) are positioned manually by the teacher upon module creation to ensure they are placed in a relevant position.

We designed the user interface in collaboration with the authors of the original methodology. The design sketches that preceed the implementation have been enclosed in Appendix B.

We will briefly go into detail on two central aspects of the design in the subsections below. Additional screenshots are enclosed for reference in Appendix C.

### 6.7.1 Student collaboration canvas

Central in our user interface design is the canvas on which the students are exchanging their ideas: the historical timeline.

We have attempted to closely recreate the original canvas elements, while making use of the advantages the digitisation process brings:

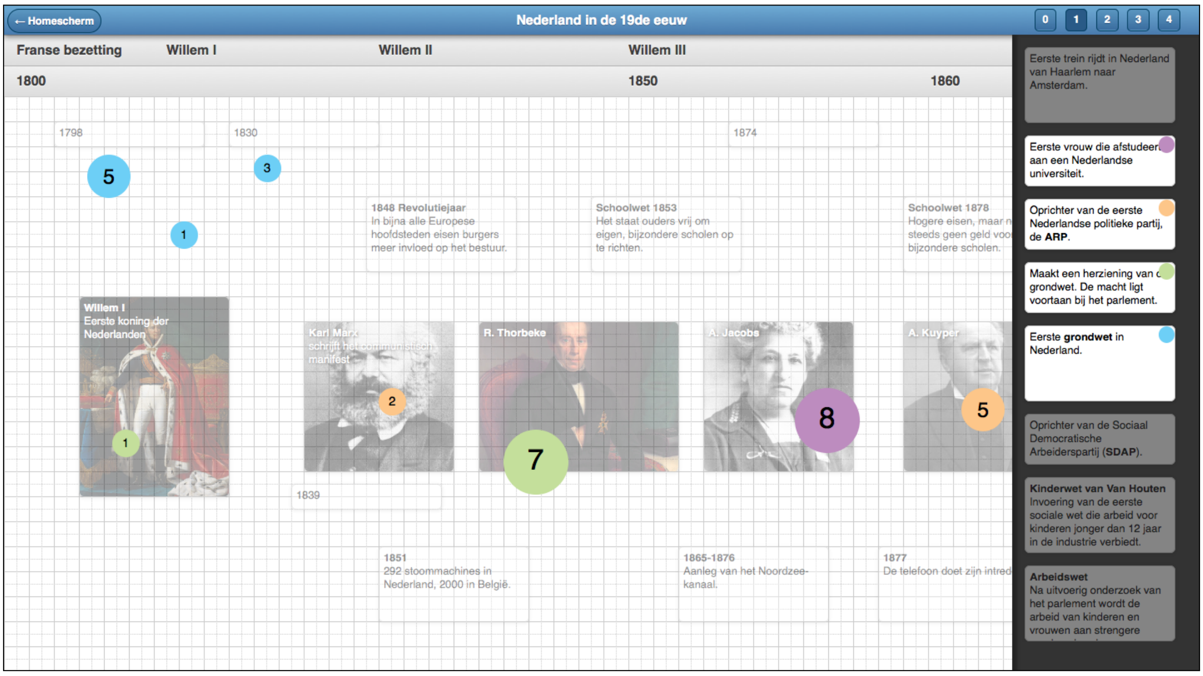
- Cards are now fully displayed in colour;
- Cards automatically align to a grid when positioning them;
- The canvas can be zoomed in and out;
- Changes are briefly highlighted as they happen through a brief 'pulse' effect;

Compare figure 6.4 on the following page to figure 6.5 on the next page. We have followed iOS user interface conventions, as the pilot classes all use iPads. For Android devices, a different *style sheet* may be adopted to give the application a *native* looking lay-out.

### 6.7.2 Aggregation system

As we have discussed, the aggregation system allows a teacher to get an overview of the knowledge in the classroom. We achieve this by allowing the teacher to select relevant cards from the drawer on the right-hand side of the screen. Upon selection, each of the cards is then assigned a unique and visibly distinct colour. For each of the cards in question, the server then clusters the positions students have placed the cards.

In these positions, each of the cards is displayed as a circular blob in the assigned colour. Cards of the same face in close vicinity are grouped together, showing their cardinality as a number on top of the blob. Figure 6.6 shows this process for four cards.



1848 Revolutiejaar
In bijna alle Europese hoofdsteden eisen burgers meer invloed op het bestuur.

Schoolwet 1853
Het staat ouders vrij om eigen, bijzondere scholen op te richten.

Schoolwet 1878
Hogere eisen, maar n steeds geen geld voo bijzondere scholen.

Willem I
Eerste koning der Nederlanden

Karl Marx
schrijft het communistisch manifest

R. Thorbeke

A. Jacobs

A. Kuyper

1851
292 stoommachines in Nederland, 2000 in België.

1865-1876
Aanleg van het Noordzee-kanaal.

1877
De telefoon doet zijn intred

Eerste trein rijdt in Nederland van Haarlem naar Amsterdam.

Eerste vrouw die afstudeer aan een Nederlandse universiteit.

Oprichter van de eerste Nederlandse politieke partij, de **ARP**.

Maakt een herziening van d grondwet. De macht ligt voortaan bij het parlement.

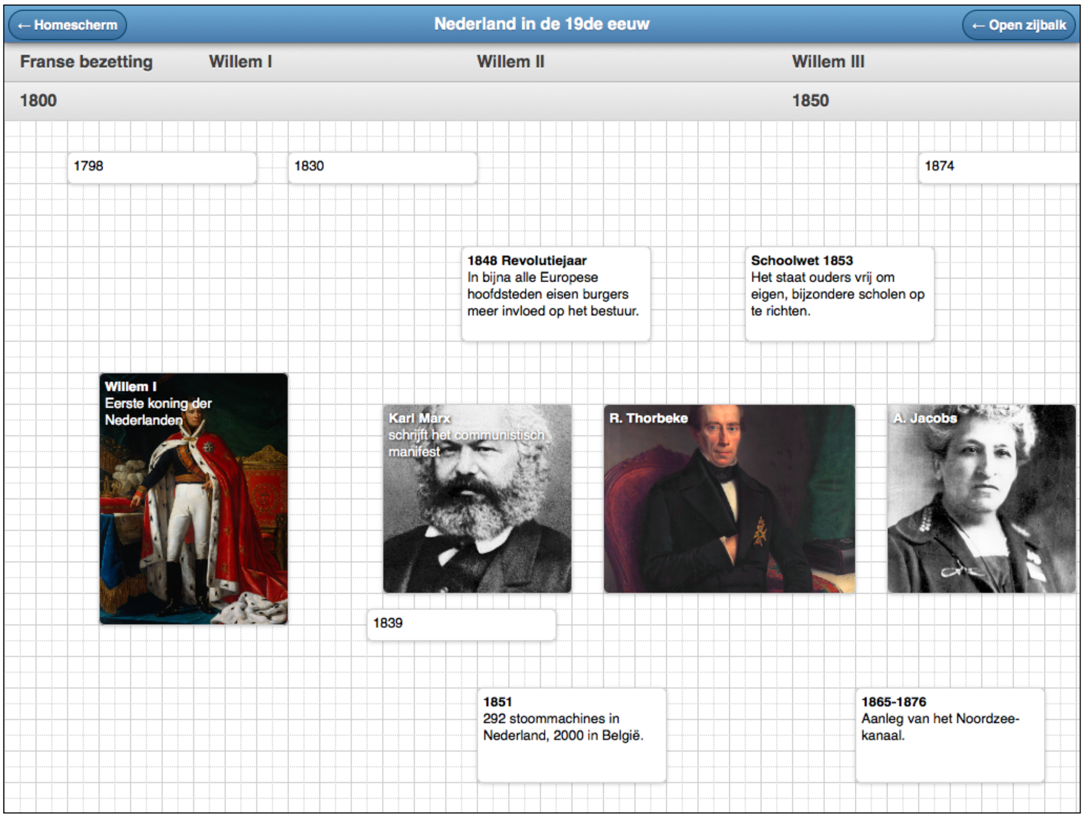
Eerste **grondwet** in Nederland.

Oprichter van de Sociaal Democratische Arbeiderspartij (**SDAP**).

**Kinderwet van Van Houten**
Invoering van de eerste sociale wet die arbeid voor kinderen jonger dan 12 jaar in de industrie verbiedt.

**Arbeidswet**
Na uitvoerig onderzoek van het parlement wordt de arbeid van kinderen en vrouwen aan strengere

*Figure 6.6:* Teacher aggregation system. Cards selected in the drawer will see their positions plotted on the canvas on the left.



**1848 Revolutiejaar**
In bijna alle Europese hoofdsteden eisen burgers meer invloed op het bestuur.

**Schoolwet 1853**
Het staat ouders vrij om eigen, bijzondere scholen op te richten.

**Willem I**
Eerste koning der Nederlanden

**Karl Marx**
schrijft het communistisch manifest

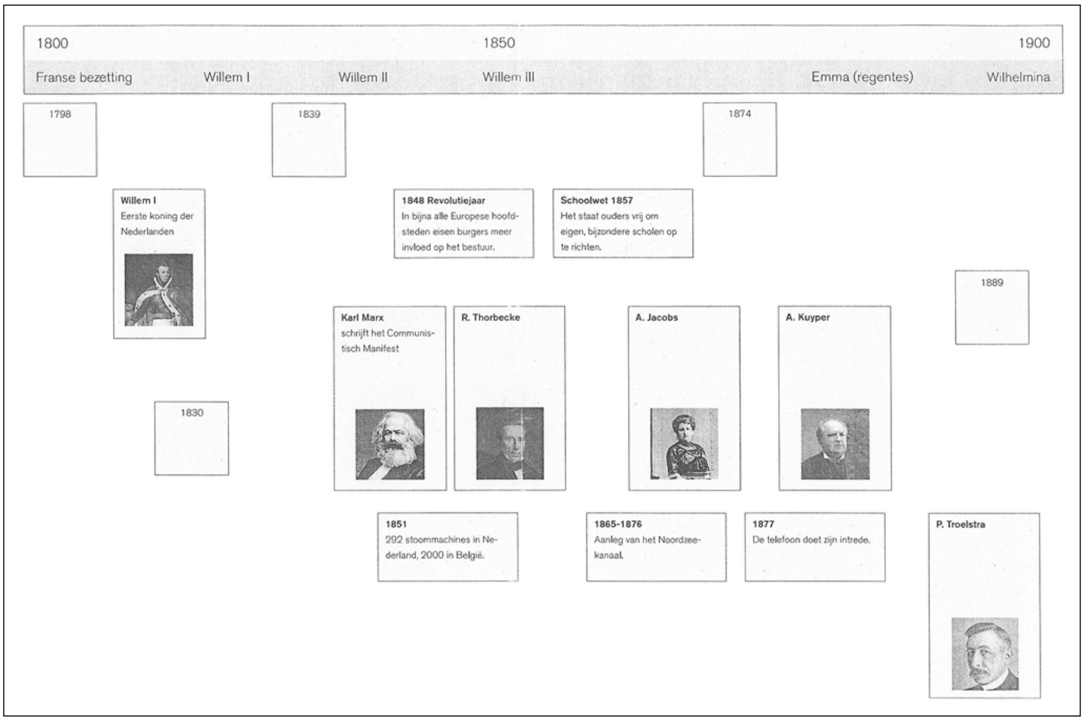
**R. Thorbeke**

**A. Jacobs**

**1851**
292 stoommachines in Nederland, 2000 in België.

**1865-1876**
Aanleg van het Noordzee-kanaal.

*Figure 6.4:* Student collaboration canvas for Active Historical Thinking's *'The Netherlands in the 19th century'.*



**Willem I**
Eerste koning der Nederlanden

**1848 Revolutiejaar**
In bijna alle Europese hoofdsteden eisen burgers meer invloed op het bestuur.

**Schoolwet 1857**
Het staat ouders vrij om eigen, bijzondere scholen op te richten.

**Karl Marx**
schrijft het Communistisch Manifest

**R. Thorbecke**

**A. Jacobs**

**A. Kuyper**

**1851**
292 stoommachines in Nederland, 2000 in België.

**1865-1876**
Aanleg van het Noordzee-kanaal.

**1877**
De telefoon doet zijn intrede.

**P. Troelstra**

*Figure 6.5:* Original paper canvas for Active Historical Thinking's *'The Netherlands in the 19th century'.*

# Chapter 7

# Conclusion and discussion

As we have seen, adapting an existing, established teaching methodology involves many design decisions to either directly map existing principles to a digital system, or adjust them slightly so they can be adapted.

We decided to adapt the teaching methodology as directly as possible, involving the tablets only as instruments with which to position cards on a canvas. From a computer science perspective, we might have elected to move the discussion to a chat function, but as studies suggest this may negatively impact discussion quality, we have chosen not to do this.

To implement the application, we have chosen not to use a native toolkit. Instead, we chose to use web technologies to make the software *cross-platform*, allowing the application to be used on tablets of both major operating systems, iOS and Android. Using *style sheets*, we mimic the appearance of native applications, making the application more intuitive.

We have made use of WebSockets and JSON technologies to make real-time communication possible in an energy efficient manner. This has also allowed to make an extension on the methodology in a supportive interactive whiteboard application, which can be used by teachers to illustrate and settle disputes in classroom discussions after each round.

## 7.1 Future work

Due to time constraints, we have not been able to do a pilot in classrooms to see how well the system performs. In future research, we would therefore like to see an analysis of how well the system works in practice.

The choices we have made in developing this application are relevant to future work as well. We have shown that web technologies allow the creation of real-time collaborative software, and how to we can deal with concurrency issues that may arise. It will be interesting to see what can be done using these technologies in other educational fields.

After finishing our implementation, another possible choice for implementation was brought to our attention. The iTask system (Achten, Koopman, & Plasmeijer, 2015), utilising the Task Oriented Programming paradigm, *generates* web-based applications using an embedded domain specific language. A similar canvas-based application may be developed as iTask software, expressing the requirements in iTask's EDSL, removing the need to develop separate server software.

As noted in the requirements, we will be releasing the software we produced as *open source* software. We hope this will not only mean the software will be adopted by schools, but also that it will see others extend upon it to make use of it for other collaborative methodologies.

# Bibliography

Achten, P., Koopman, P., & Plasmeijer, R. (2015). An introduction to task oriented programming. In V. Zsók, Z. Horváth, & L. Csató (Eds.), *Central european functional programming school* (Vol. 8606, p. 187-245). Springer International Publishing. Retrieved from `http://dx.doi.org/10.1007/978-3-319-15940-9_5` doi: 10.1007/978-3-319-15940-9_5

Ahn, W., Choi, J., Shull, T., Garzarán, M. J., & Torrellas, J. (2014). Improving javascript performance by deconstructing the type system. In *Proceedings of the 35th acm sigplan conference on programming language design and implementation* (pp. 496–507). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2594291.2594332` doi: 10.1145/2594291.2594332

Bray, T. (2014, March). *The JavaScript Object Notation (JSON) Data Interchange Format* (RFC No. 7159). RFC Editor. Internet Requests for Comments. Retrieved from `http://www.rfc-editor.org/rfc/rfc7159.txt`

Charland, A., & Leroux, B. (2011). Mobile application development: Web vs. native. *Communications of the ACM*, *54*(5), 49 - 53.

*Eduapp.* (2013). Retrieved 2014-01-20, from `http://eduapp.nl/zoeken/apps`

Fette, I., & Melnikov, A. (2011, December). *The websocket protocol* (RFC No. 6455). RFC Editor. Internet Requests for Comments. Retrieved from `http://www.rfc-editor.org/rfc/rfc6455.txt`

Fjermestad, J. (2003). An analysis of communication mode in group support systems research. *Decision Support Systems*, *37*, 239–263.

Havekes, H., Coppen, P. A., & Luttenberg, J. (2012). Knowing and doing history: A conceptual framework and pedagogy for teaching historical contextualisation. *International Journal of Historical Learning, Teaching and Research*, *11.1*, 72-93.

Lipponen, L., Rahikainen, M., Lallimo, J., & Hakkarainen, K. (2003). Patterns of participation and discourse in elementary students' computer- supported collaborative learning. *Learning and Instruction*, *13*, 487–509.

Martinsen, J. K., Grahn, H., & Isberg, A. (2013, March). Using speculation to enhance javascript performance in web applications. *Internet Computing, IEEE*, *17*(2), 10-19. doi: 10.1109/MIC.2012.146

Qveflander, N. (2010). *Pushing real time data using HTML5 Web Sockets* (Unpublished master's thesis). Umeå University.

Resta, P., & Laferrière, T. (2007). Technology in support of collaborative learning. *Educational Psychology Review*, *19*(1), 65-83. Retrieved from `http://dx.doi.org/10.1007/s10648-007-9042-7` doi: 10.1007/s10648-007-9042-7

Swamy, R., & Mahadevan, G. (2011). Event Driven Architecture using HTML5 Web Sockets for Wireless Sensor Networks. *White Papers, Planetary Scientific Research Center.*

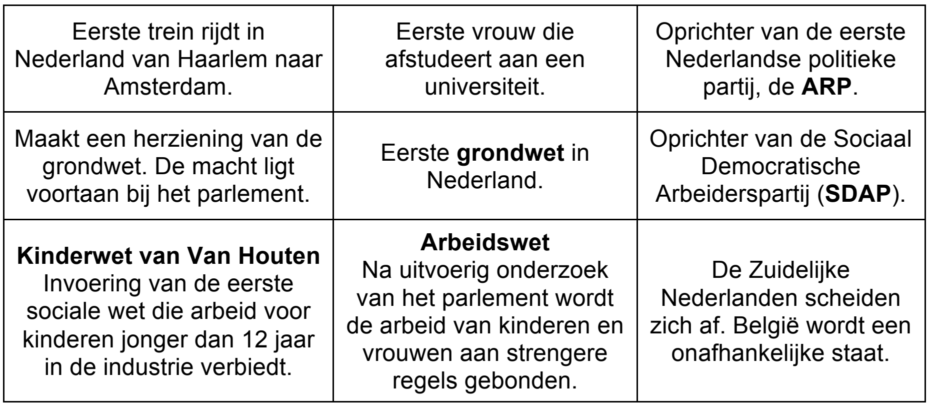
Wessels, A., Purvis, M., Jackson, J., & Rahman, S. (2011). Remote data visualization through websockets. In *Information technology: New generations (itng), 2011 eighth international conference on* (pp. 1050–1051).

# Appendix A

# Original teaching methodology

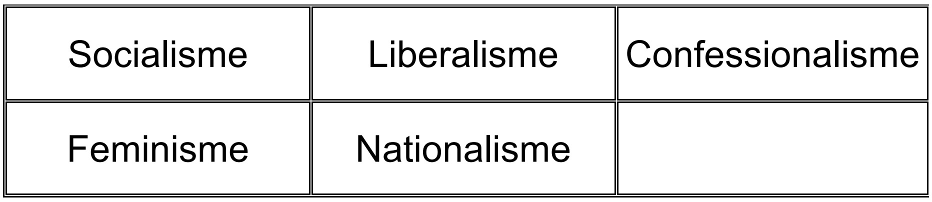
## A.1 Chronology cards for 'Nederland in de 19de eeuw'

### Events (to be printed on green paper)



| | | |
|---|---|---|
| Eerste trein rijdt in Nederland van Haarlem naar Amsterdam. | Eerste vrouw die afstudeert aan een universiteit. | Oprichter van de eerste Nederlandse politieke partij, de **ARP**. |
| Maakt een herziening van de grondwet. De macht ligt voortaan bij het parlement. | Eerste **grondwet** in Nederland. | Oprichter van de Sociaal Democratische Arbeiderspartij (**SDAP**). |
| **Kinderwet van Van Houten**<br>Invoering van de eerste sociale wet die arbeid voor kinderen jonger dan 12 jaar in de industrie verbiedt. | **Arbeidswet**<br>Na uitvoerig onderzoek van het parlement wordt de arbeid van kinderen en vrouwen aan strengere regels gebonden. | De Zuidelijke Nederlanden scheiden zich af. België wordt een onafhankelijke staat. |

### Movements (to be printed on blue paper)



| | | |
|---|---|---|
| Socialisme | Liberalisme | Confessionalisme |
| Feminisme | Nationalisme | |

Keywords (to be printed on yellow paper)

| | | |
|---|---|---|
| eenheidsstaat | nachtwakersstaat | Vrijheid van het individu |
| gelijkheid | Sociale wetten | Algemeen kiesrecht |
| Harmonie tussen werkgevers en werknemers | Gezin is hoeksteen van de samenleving | Overheidsfinanciering bijzondere scholen |
| vaderlandsliefde | Scheiding kerk en staat | Gemeenschappelijk verleden |

## Sources (to be printed on white paper)

1. Uit het programma van een Nederlandse politieke partij:
(...) steunt elke politieke of economische beweging van de arbeiders die er op gericht is hun levensvoorwaarden te verbeteren. Zolang de arbeiders de staatsmacht nog niet in handen hebben, zal de (...) proberen, alle politieke rechten te veroveren en die te gebruiken (…).

2. Alleen dat deel van de vaderlandse geschiedenis, dat de leerling een overzicht schenkt van de wording van de Nederlandse staat en hem de grote daden leert kennen van het voorgeslacht, verdient brede ontwikkeling. Door dit voorgeslacht werd onder leiding van Oranje ons onafhankelijk bestaan gegrondvest.

3. ‘Wil dit zeggen dat de staat voor alles moet zorgen, alle kwalen en gebreken van de maatschappij moet genezen? Integendeel. Een eerste wet is onthouding; onthouding van datgene wat buiten de rechterlijke macht van de staat ligt. (...) regeren vereist dat de staat datgene wat niet tot het recht behoort, overlaat aan anderen.’

4. Doel is het verkrijgen van overheidsfinanciering van het christelijk onderwijs, omdat de liberale overheid zich slechts verantwoordelijk voelde voor het openbaar onderwijs, dat godsdienstig strikt neutraal was.

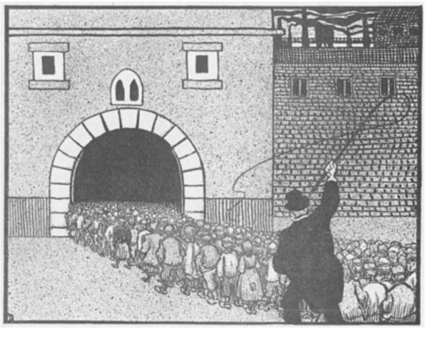
5. Wien Neêrlandsch bloed in de aders vloeit,
Van vreemde smetten vrij,
Wiens hart voor land en koning gloeit,
Verheff' den zang als wij:
Hij stell' met ons, vereend van zin,
Met onbeklemde borst,
Het godgevallig feestlied in
Voor vaderland en vorst.

6. Kan van den fabrieksarbeid van vrouwen en meisjes in het algemeen niet veel goeds gezegd worden, in het - bijzonder is die arbeid, waar hij door gehuwde vrouwen verricht wordt, dikwijls noodlottig voor de gezinnen.

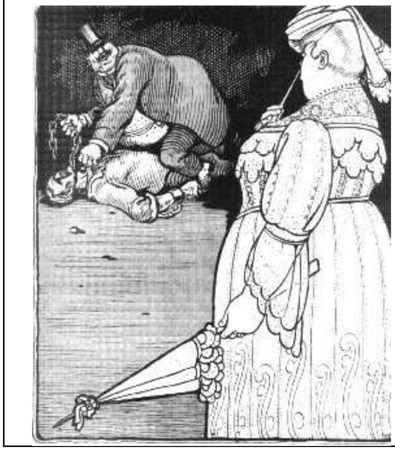
7. In eerste plaats zijn onze waarden bijzonder geschikt om patroons en arbeiders met elkaar te verzoenen en tot samenwerking te brengen, namelijk door beide klassen te herinneren aan hun wederzijdse plichten.

8. Art. 28. Er zal een wetboek gemaakt worden zowel van burgerlijke als van lijfstraffelijke wetten (…) algemeen voor de hele republiek.

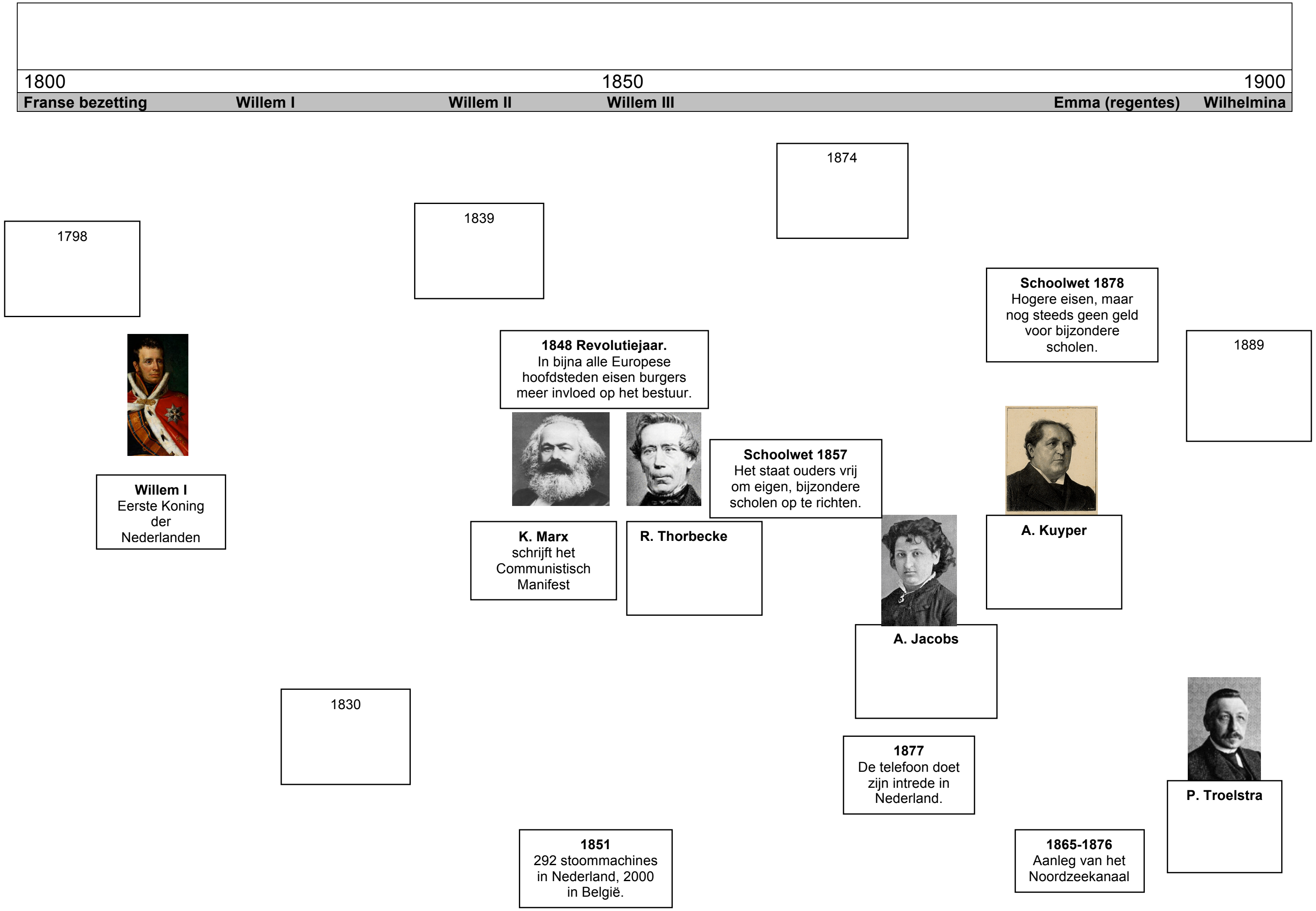
9.



10. ‘Ik eisch dezelfde rechten als gij’



# A.2 Timeline for ‘Nederland in de 19de eeuw’



1800 1850 1900

Franse bezetting | Willem I | Willem II | Willem III | Emma (regentes) | Wilhelmina

1874

1798

1839

**Schoolwet 1878**
Hogere eisen, maar nog steeds geen geld voor bijzondere scholen.

**1848 Revolutiejaar.**
In bijna alle Europese hoofdsteden eisen burgers meer invloed op het bestuur.

1889

**Schoolwet 1857**
Het staat ouders vrij om eigen, bijzondere scholen op te richten.

**Willem I**
Eerste Koning der Nederlanden

**K. Marx**
schrijft het Communistisch Manifest

**R. Thorbecke**

**A. Kuyper**

**A. Jacobs**

1830

**1877**
De telefoon doet zijn intrede in Nederland.

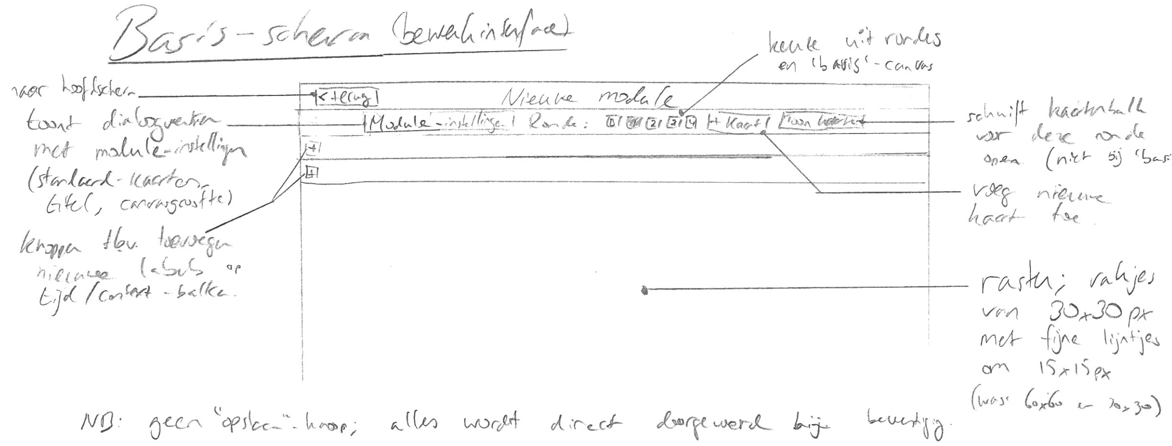
**P. Troelstra**

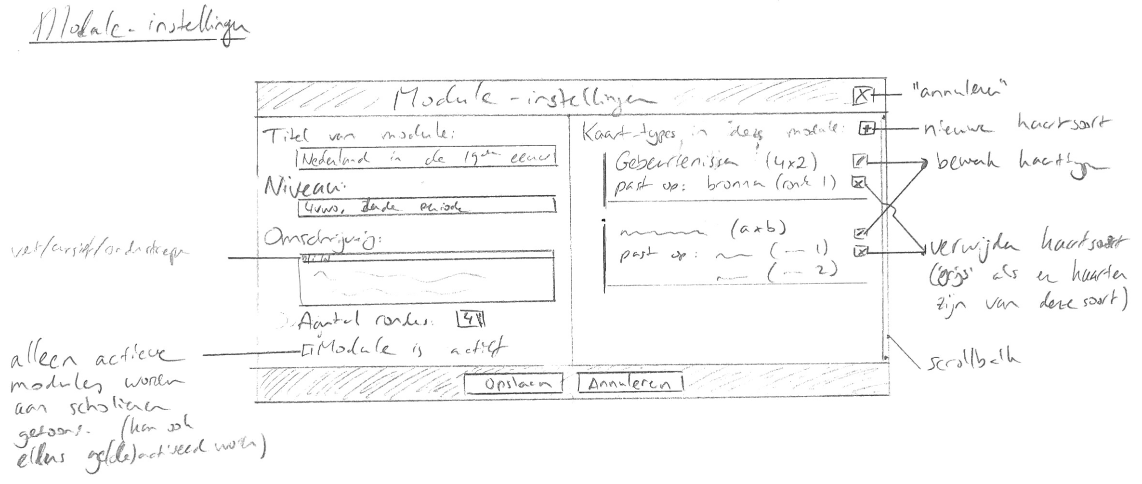
**1851**
292 stoommachines in Nederland, 2000 in België.

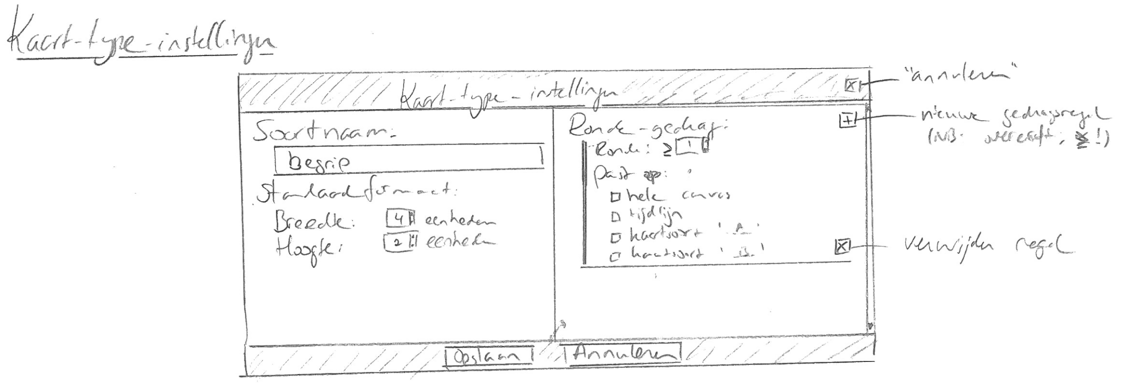
**1865-1876**
Aanleg van het Noordzeekanaal

# Appendix B

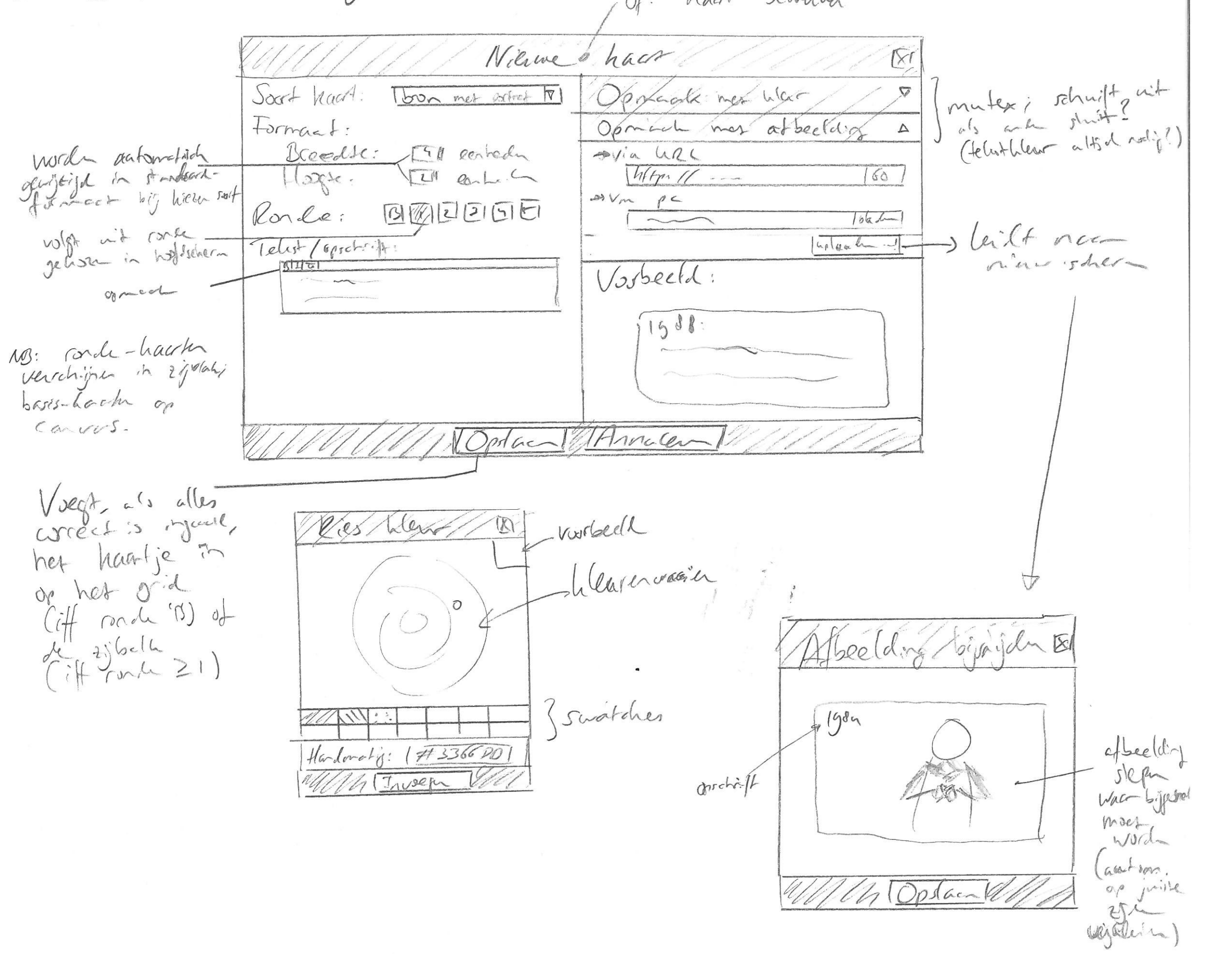
# Design sketches


Basis-scherm (bewerkinterface)
naar hoofdscherm
< terug
Nieuwe module
toont dialoogvenster met module-instellingen (standaard-kaarten, titel, canvasgrootte)
Module-instellingen
Ronde:
+ Kaart
keuze uit rondes en 'basis'-canvas
schuift kaartenbalk voor deze ronde open (niet bij 'basis'!)
voeg nieuwe kaart toe
knoppen t.b.v. toevoegen nieuwe labels op tijd/context-balken
raster; vakjes van 30×30 px met fijne lijntjes om 15×15 px (was 60×60 en 30×30)
NB: geen "opslaan"-knop; alles wordt direct doorgevoerd bij bevestiging.





Nieuwe kaart toevoegen



of: "Kaart bewerken"

worden automatisch gevuld in standaard-formaat bij kiezen soort

volgt uit ronde gekozen in hoofdscherm

opmaak

NB: ronde-kaarten verschijnen in zijbalk; basis-kaarten op canvas.

mutex; schuift uit als ander sluit? (tekstkleur altijd nodig?)

leidt naar nieuw scherm

Voegt, als alles correct is ingevuld, het kaartje in op het grid (iff ronde 'B') of de zijbalk (iff ronde ≥ 1)

voorbeeld

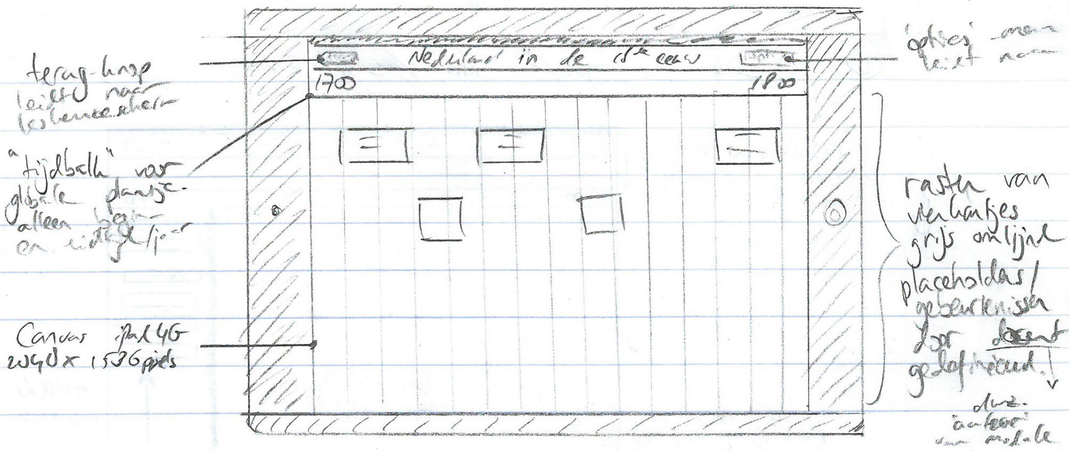
kleurenwaaier

swatches

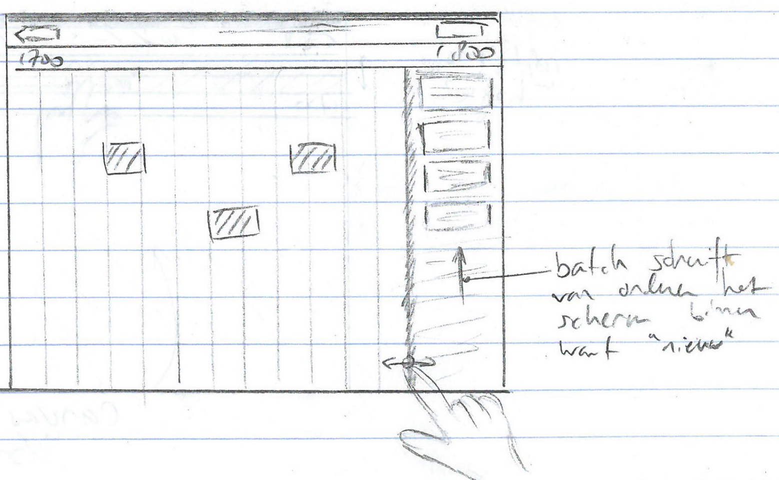
opschrift

afbeelding slepen waar bijgesneden moet worden (aantoon. op juiste zijde uitgelijnd)

A

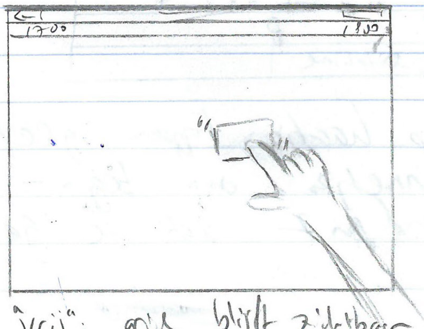


B Docent geeft eerste batch kaarten vrij. Rechts klapt een menu open met een 'zoom' beweging om aan te geven dat het met vingers weg te werken en op te roepen is.

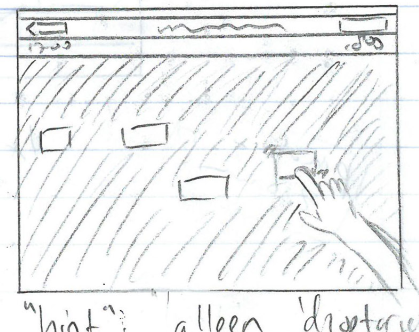


C Kaartjes kunnen uit de zijbalk gesleept worden om zo op het canvas geplaatst te worden. Kaartjes die niet vrij geplaatst kunnen worden krijgen 'hint'

NB: zijbalk klapt in zodra "flink" gesleept wordt

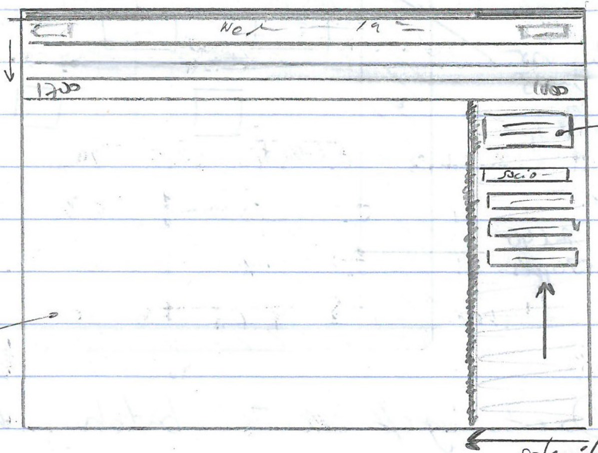


"vrij": grid blijft zichtbaar



"hint": alleen 'droptargets' goed zichtbaar, rest grijs

## (D) Docent geeft kaartjes ronde 2 vrij



Mini-raster boven tijdbalk schuift open

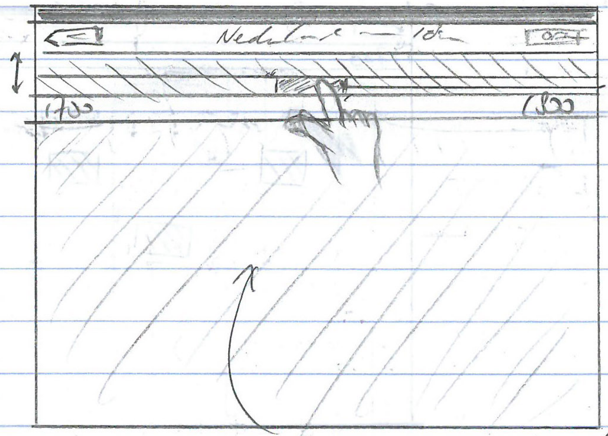
eventuele kaarten van vorige ronde blijven liggen

tweede batch kaarten schuift naar binnen van onder

raster verandert niet

schuift open als ie gesloten was

## (E) Kaartjes verslepen in ronde 2: met "hints"



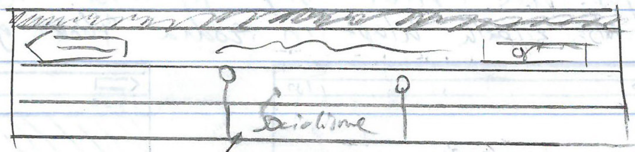
rekt iets uit in de hoogte?

kaartje "snapt" naar tijdbalk

elders loslaten => terug naar zijkant

Canvas wordt bedekt "donkergrijs"

## (F) Kaartjes strekken in ronde 2



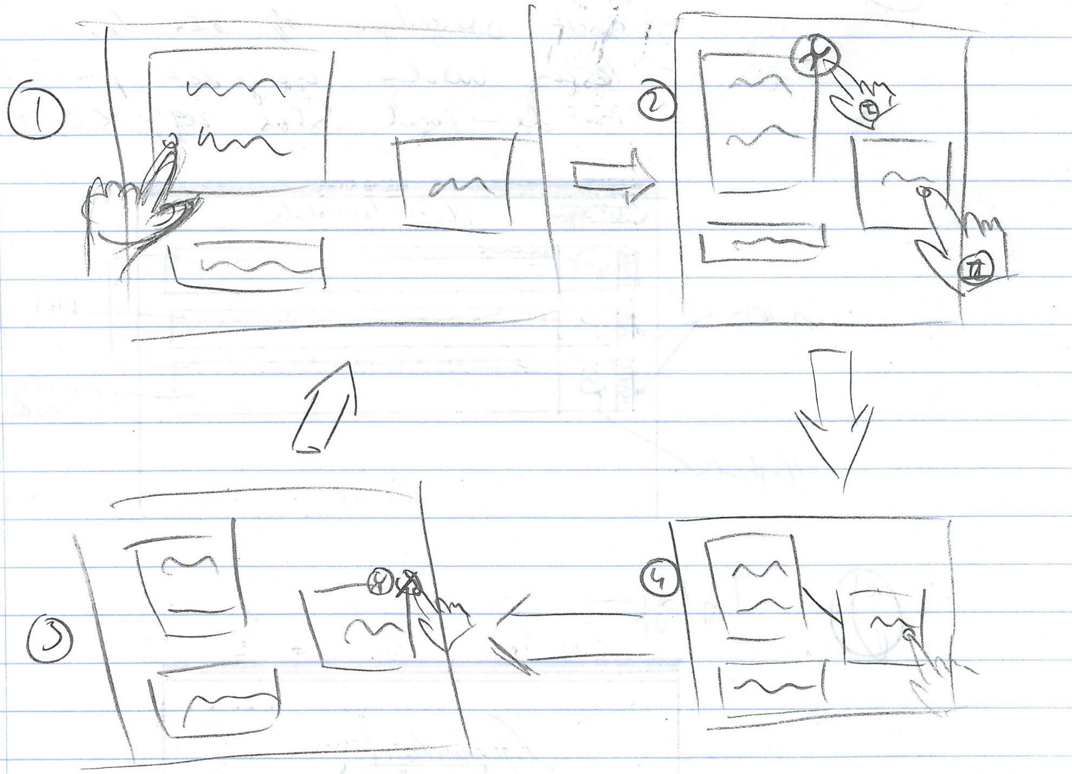
1x op kaartje tappen geeft pinnetjes om begin- en eindpunt aan te geven

pinch + zoom?

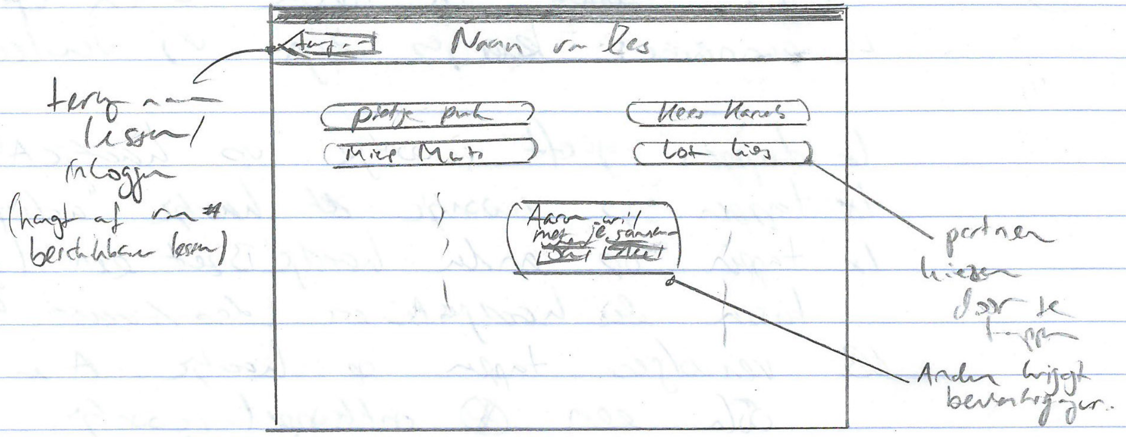
Ⓖ Kaartjes koppelen in ronde 3 en 4

↳ aanname: kaartjes zijn vrij versleepbaar.

1x tappen geeft icoontje op kaartje A ⊗
1x tappen op icoontje zet kaartje "actief"
1x tappen op ander kaartje B zet een lijntje
tussen de kaartjes A&B en deactiveert kaartje A.
NB: vervolgens tappen op kaartje A en B geeft
ook een ⊗ ontkoppel-icoontje. Dit is
nodig om meerdere kaartjes aan
elkaar te linken.

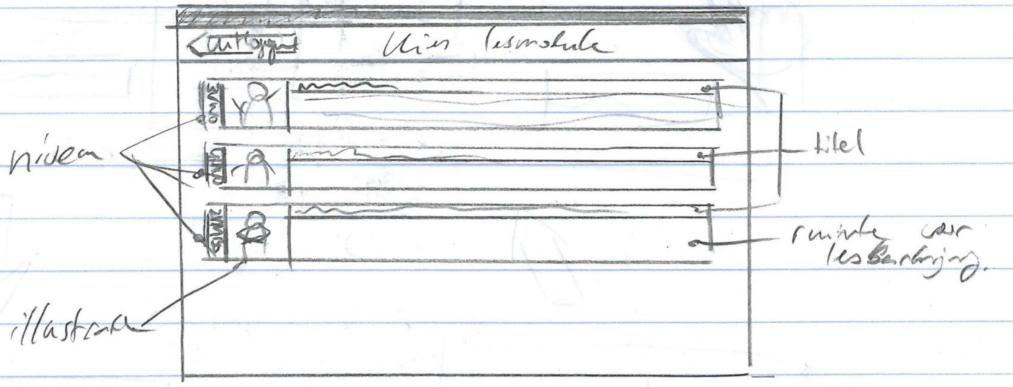


## H Koppelen leerlingen.

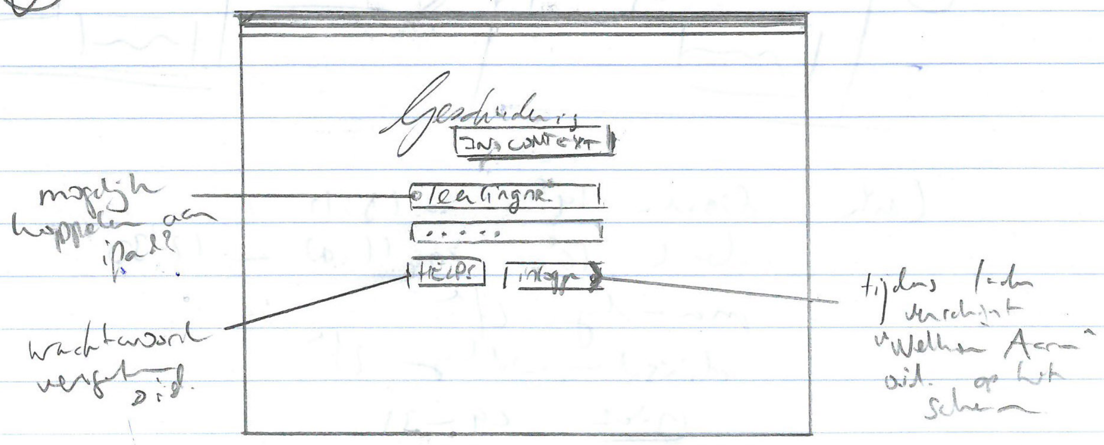

terug
Naam van les
pietje puk
Kees Karels
Mies Muts
Lot Lies
Aaron wil met je samen
Ok
Nee
terug naar lessen/ inloggen (hangt af van # beschikbare lessen)
partner kiezen door te tappen
Ander krijgt bevestiging.

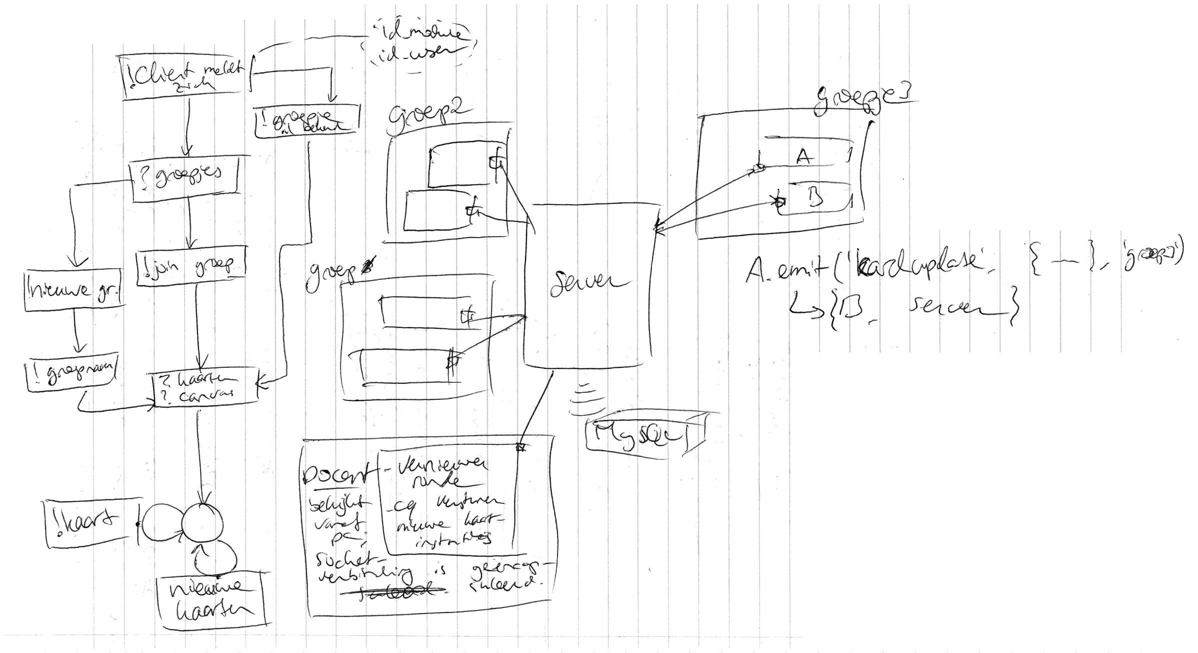
## I Kiezen les

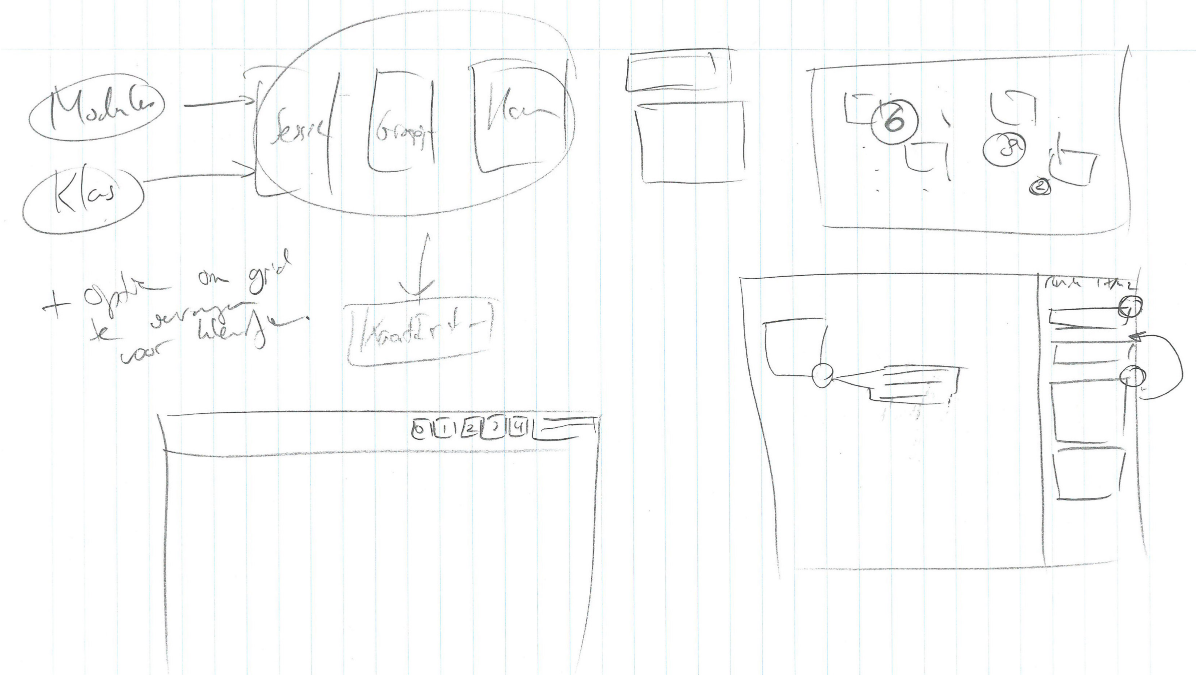
- wordt overgeslagen, bij één les.
- lessen worden aangezet per school
- in de regel dus, zeg, <0, 5> lessen



## J Inloggen


Geschiedenis
IN CONTEXT
leerlingnr.
• • • • •
HELP!
inloggen
mogelijk koppelen aan iPad?
wachtwoord vergeten o.i.d.
tijdens laden verschijnt "Welkom Aaron" o.i.d. op het scherm

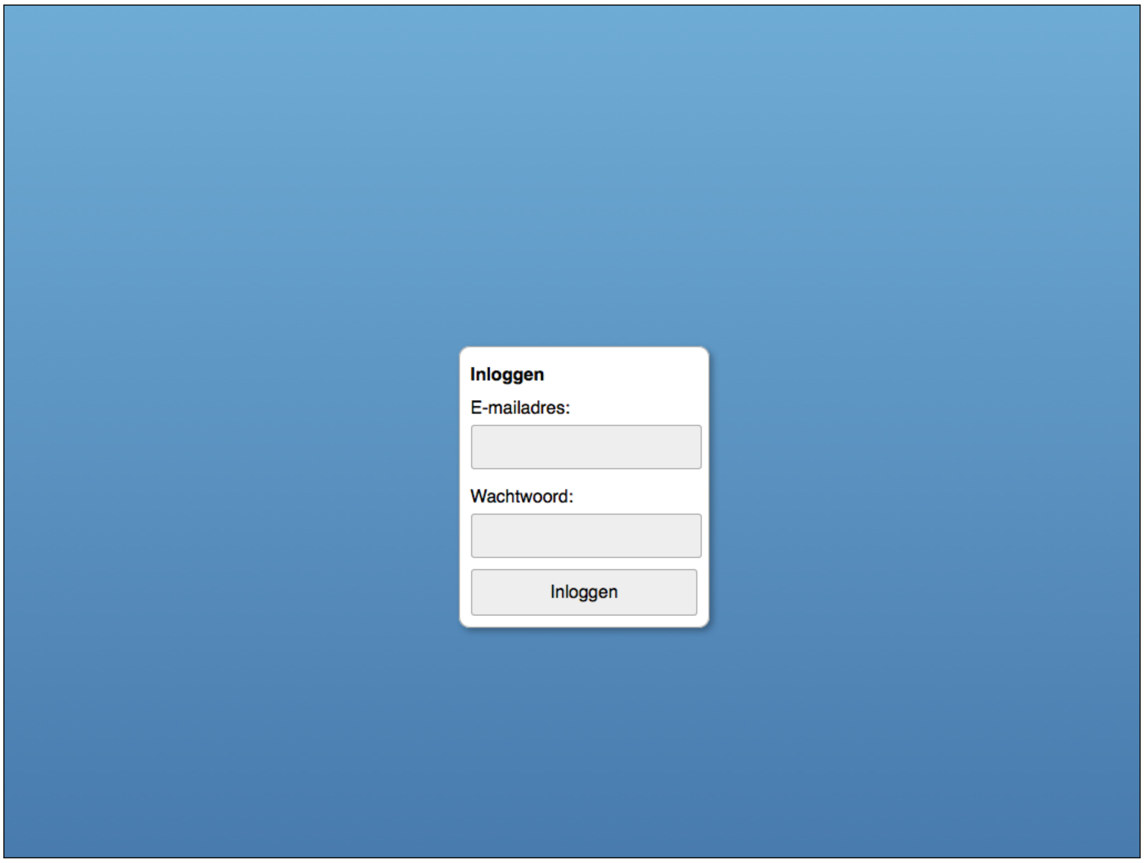

'id_module'
'id_user'
!Client meldt zich
!groepje al bekend
?groepjes
!nieuwe gr.
!join groep
!groepnaam
? kaarten
? canvas
!kaart
nieuwe kaarten
groep2
groep3
A
B
1
1
groep
Server
MySQL
Docent
bekijkt verloop pc
- Vernieuwen ronde
- cq Versturen nieuwe kaart-instructies
Socket-verbinding is ~~[illegible]~~ geëncapsuleerd.
A.emit('kaartupdate', {—}, 'groep3')
↳{B, server}


Modules
Klas
Sessie
Groepje
Klas
+ Optie om grid te vervangen voor blokjes.
Kaartinst-
0
1
2
3
4
6
2
ronde 1 / 2

# Appendix C

# Additional screenshots

## C.1 Logging in



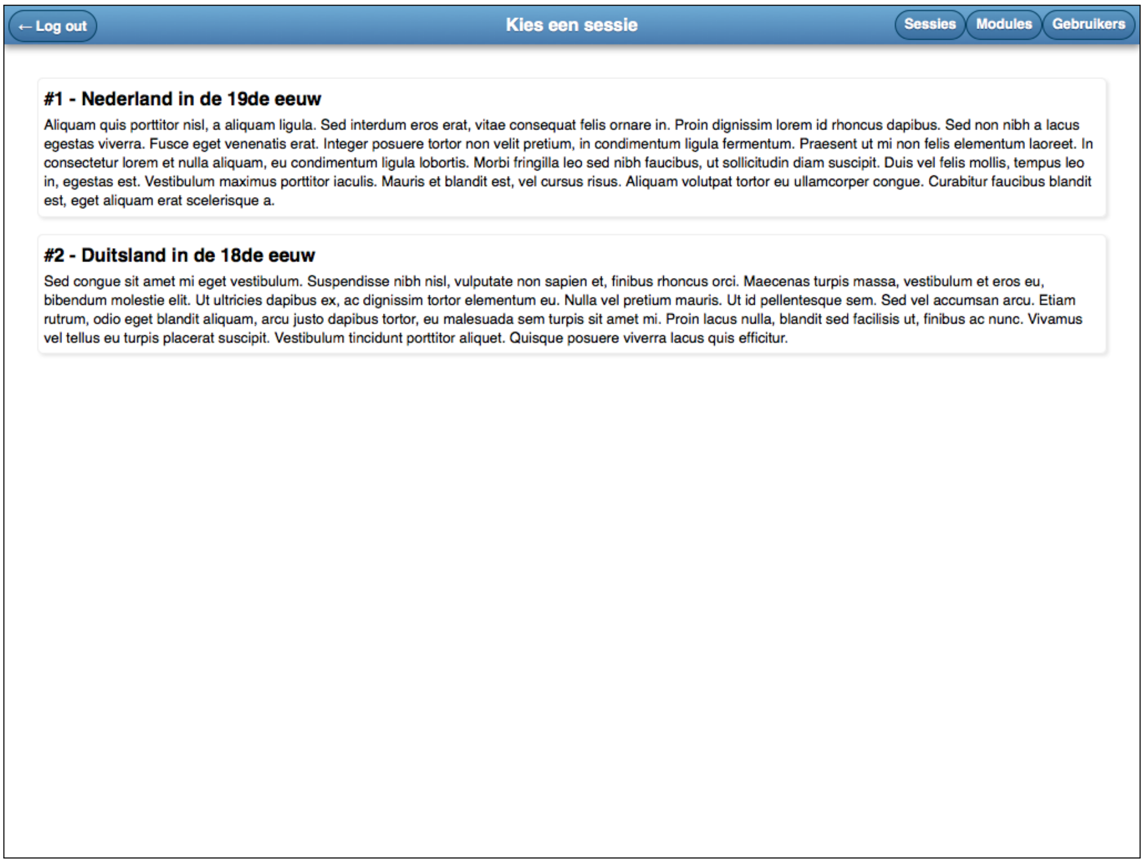
Inloggen

E-mailadres:

Wachtwoord:

*Figure C.1:* The log in screen. Student and teachers can log in with their school email address and password.

## C.2 Choosing a session and group



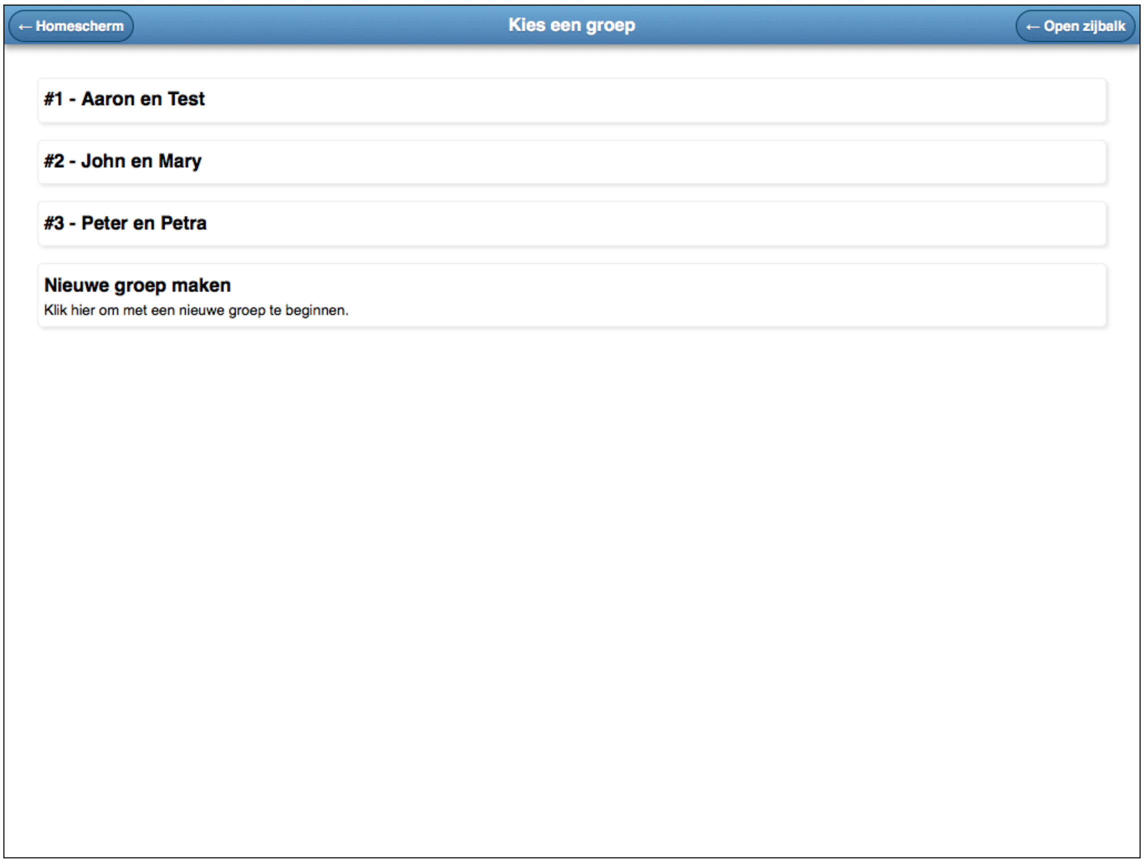
**#1 - Nederland in de 19de eeuw**

Aliquam quis porttitor nisl, a aliquam ligula. Sed interdum eros erat, vitae consequat felis ornare in. Proin dignissim lorem id rhoncus dapibus. Sed non nibh a lacus egestas viverra. Fusce eget venenatis erat. Integer posuere tortor non velit pretium, in condimentum ligula fermentum. Praesent ut mi non felis elementum laoreet. In consectetur lorem et nulla aliquam, eu condimentum ligula lobortis. Morbi fringilla leo sed nibh faucibus, ut sollicitudin diam suscipit. Duis vel felis mollis, tempus leo in, egestas est. Vestibulum maximus porttitor iaculis. Mauris et blandit est, vel cursus risus. Aliquam volutpat tortor eu ullamcorper congue. Curabitur faucibus blandit est, eget aliquam erat scelerisque a.

**#2 - Duitsland in de 18de eeuw**

Sed congue sit amet mi eget vestibulum. Suspendisse nibh nisl, vulputate non sapien et, finibus rhoncus orci. Maecenas turpis massa, vestibulum et eros eu, bibendum molestie elit. Ut ultricies dapibus ex, ac dignissim tortor elementum eu. Nulla vel pretium mauris. Ut id pellentesque sem. Sed vel accumsan arcu. Etiam rutrum, odio eget blandit aliquam, arcu justo dapibus tortor, eu malesuada sem turpis sit amet mi. Proin lacus nulla, blandit sed facilisis ut, finibus ac nunc. Vivamus vel tellus eu turpis placerat suscipit. Vestibulum tincidunt porttitor aliquet. Quisque posuere viverra lacus quis efficitur.

*Figure C.2:* The session screen. In this example, there are two active sessions a student can choose to join. Teachers can use this screen to navigate to their management interface using the buttons in the top-right corner of the screen. These are not visible for students.



Kies een groep

**#1 - Aaron en Test**

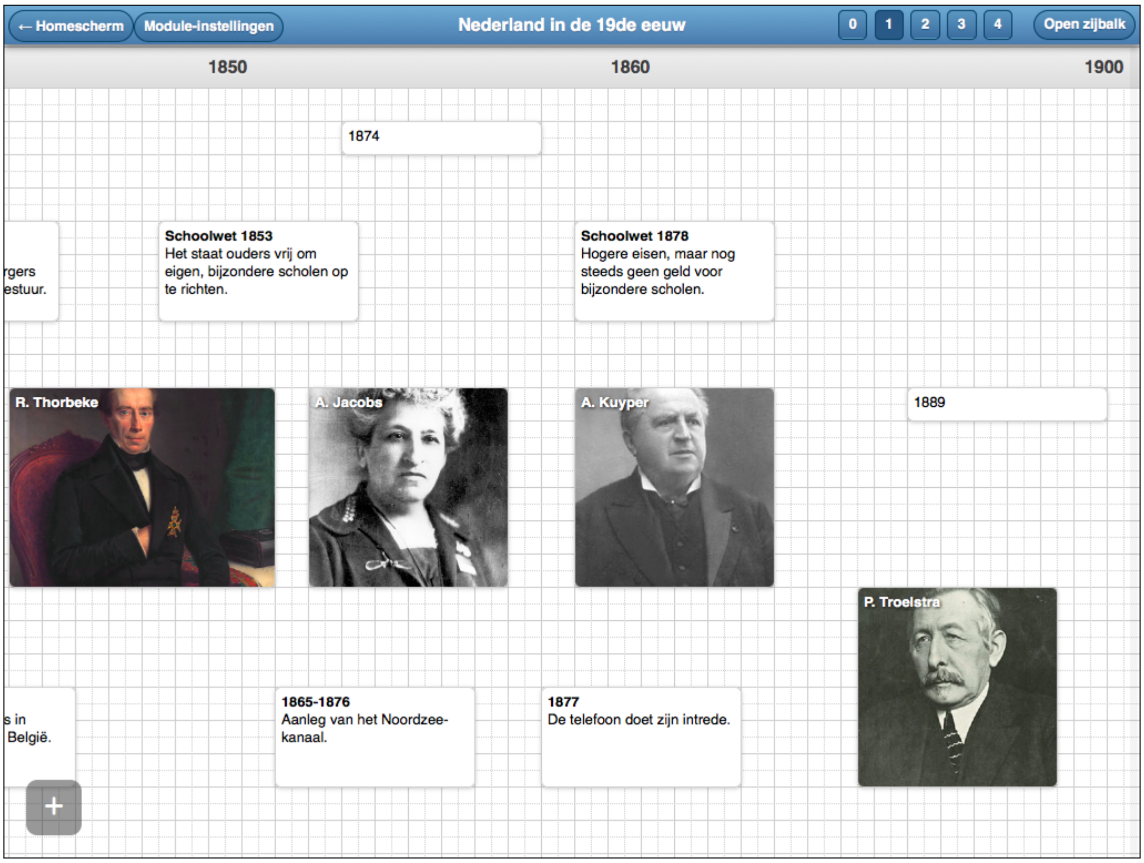
**#2 - John en Mary**

**#3 - Peter en Petra**

**Nieuwe groep maken**
Klik hier om met een nieuwe groep te beginnen.

*Figure C.3:* After choosing a session, a student can join a group. Their name will be appended to the group name. If a group has already been joined, it will be placed at the top and highlighted in yellow.

## C.3 Module management



1874

rgers
estuur.

**Schoolwet 1853**
Het staat ouders vrij om eigen, bijzondere scholen op te richten.

**Schoolwet 1878**
Hogere eisen, maar nog steeds geen geld voor bijzondere scholen.

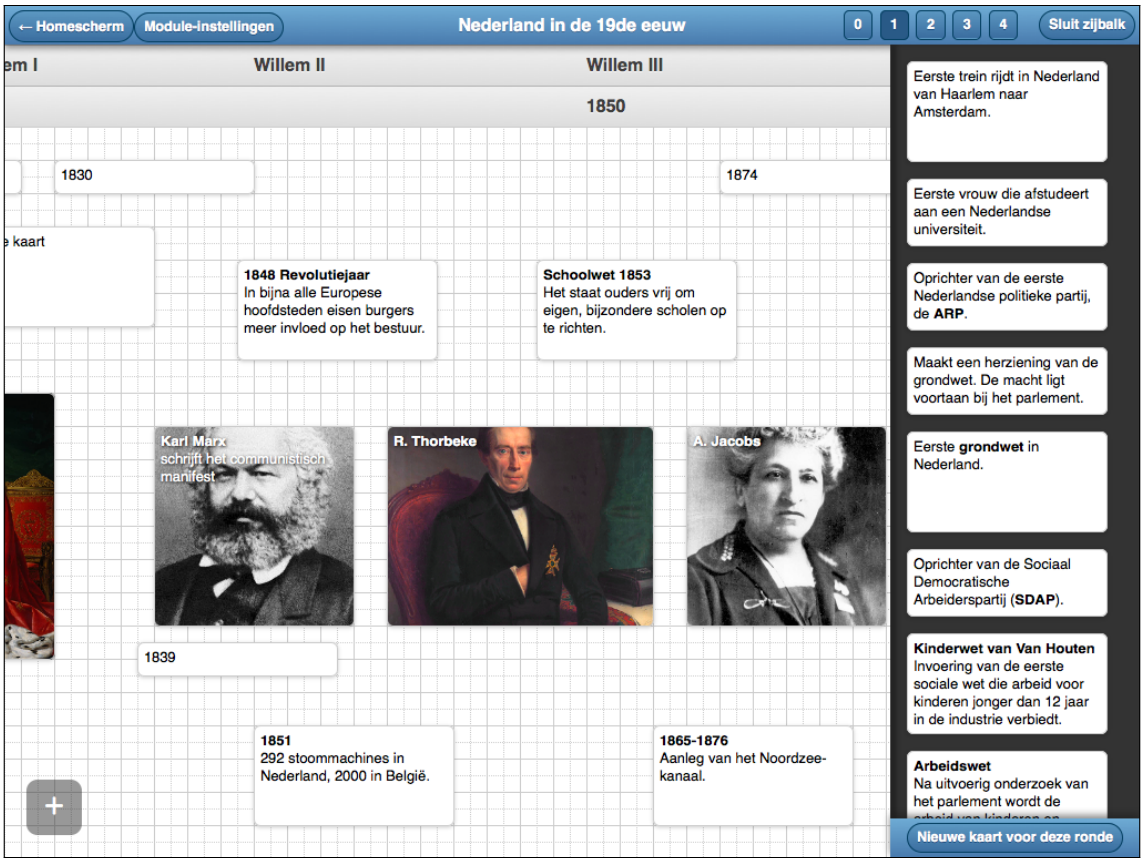
1889

s in
België.

**1865-1876**
Aanleg van het Noordzee-kanaal.

**1877**
De telefoon doet zijn intrede.

*Figure C.4:* Administrative interface for managing modules. New cards can be added by clicking the [+] button in the lower-left corner of the canvas. 'Fixed' cards on the canvas may be dragged into the desired position while in this editor mode.



**1848 Revolutiejaar**
In bijna alle Europese hoofdsteden eisen burgers meer invloed op het bestuur.

**Schoolwet 1853**
Het staat ouders vrij om eigen, bijzondere scholen op te richten.

Karl Marx schrijft het communistisch manifest

R. Thorbeke

A. Jacobs

**1851**
292 stoommachines in Nederland, 2000 in België.

**1865-1876**
Aanleg van het Noordzee-kanaal.

Eerste trein rijdt in Nederland van Haarlem naar Amsterdam.

Eerste vrouw die afstudeert aan een Nederlandse universiteit.

Oprichter van de eerste Nederlandse politieke partij, de **ARP**.

Maakt een herziening van de grondwet. De macht ligt voortaan bij het parlement.

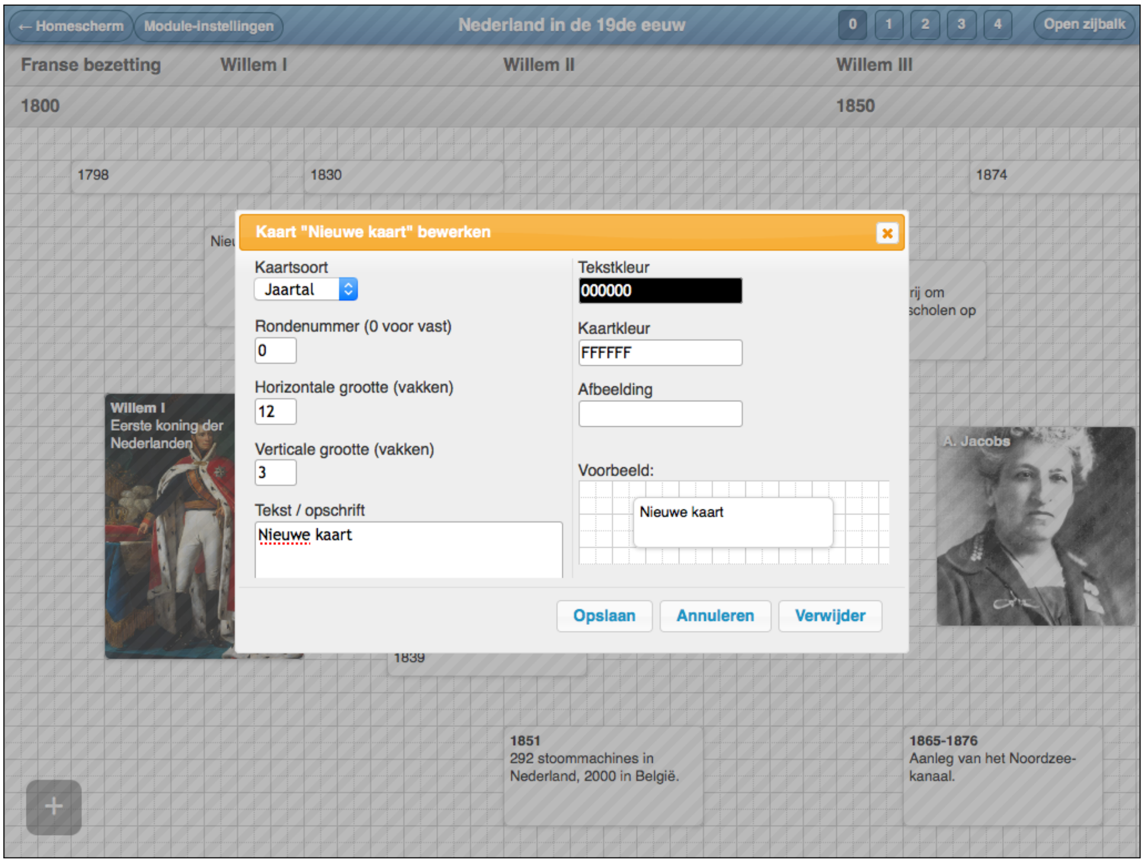
Eerste **grondwet** in Nederland.

Oprichter van de Sociaal Democratische Arbeiderspartij (**SDAP**).

**Kinderwet van Van Houten**
Invoering van de eerste sociale wet die arbeid voor kinderen jonger dan 12 jaar in de industrie verbiedt.

**Arbeidswet**
Na uitvoerig onderzoek van het parlement wordt de arbeid van kinderen en

*Figure C.5:* The cards that are to be positioned by students are shown in a collapsible drawer on the right side of the screen. Cards for a particular round may be edited after clicking the relevant round number in the top toolbar.



Kaart "Nieuwe kaart" bewerken

Kaartsoort: Jaartal

Rondenummer (0 voor vast): 0

Horizontale grootte (vakken): 12

Verticale grootte (vakken): 3

Tekst / opschrift: Nieuwe kaart

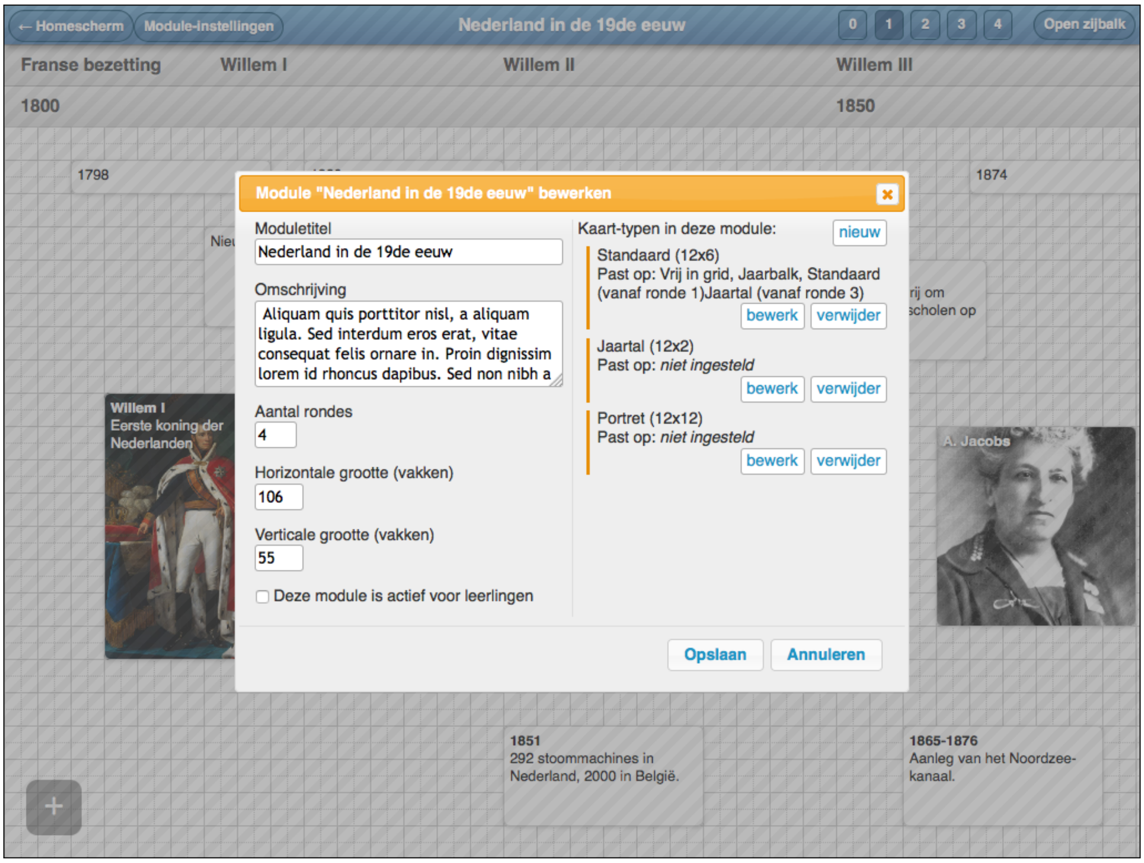
Tekstkleur: 000000

Kaartkleur: FFFFFF

Afbeelding:

Voorbeeld: Nieuwe kaart

*Figure C.6:* The card editor screen. This is shown after double-clicking an existing card, or when creating a new card. Note that caption, size, colours and type may be set using this interface.



Module "Nederland in de 19de eeuw" bewerken ×

Moduletitel
Nederland in de 19de eeuw

Omschrijving
Aliquam quis porttitor nisl, a aliquam ligula. Sed interdum eros erat, vitae consequat felis ornare in. Proin dignissim lorem id rhoncus dapibus. Sed non nibh a

Aantal rondes
4

Horizontale grootte (vakken)
106

Verticale grootte (vakken)
55

□ Deze module is actief voor leerlingen

Kaart-typen in deze module: nieuw

Standaard (12x6)
Past op: Vrij in grid, Jaarbalk, Standaard (vanaf ronde 1)Jaartal (vanaf ronde 3)
bewerk verwijder

Jaartal (12x2)
Past op: *niet ingesteld*
bewerk verwijder

Portret (12x12)
Past op: *niet ingesteld*
bewerk verwijder

Opslaan Annuleren

*Figure C.7:* The module settings screen. This allows a teacher to set the title and description of the module, as well as the width and height of the canvas. Furthermore, card type behaviour can be defined from this window.

# Appendix D

# Source code header listings

## D.1 Implementation: PHP model classes

### Authentication.php

Authentication class, containing various static functions used for account verification and session management.

PBKDF2 implementation by havoc@defuse.ca (https://defuse.ca/php-pbkdf2.htm)

```
class Authentication
  // Checks whether a user exists in the database.
  public static function checkExists($id_user)

  // Finds the user id belonging to a certain emailaddress.
  public static function getUserId($emailaddress)

  // Verifies whether the user is currently logged in.
  public static function isLoggedIn()

  // Checks a password for a given username against the database.
  public static function checkPassword($emailaddress, $password)

  // Resets a password for a certain user.
  public static function updatePassword($id_user, $hash)

  public static function create_hash($password)

  public static function slow_equals($a, $b)

  /*
   * PBKDF2 key derivation function as defined by RSA's PKCS #5: https://www.ietf.org/rfc/rfc2898.txt
   * $algorithm - The hash algorithm to use. Recommended: SHA256
   * $password - The password.
   * $salt - A salt that is unique to the password.
   * $count - Iteration count. Higher is better, but slower. Recommended: At least 1000.
   * $key_length - The length of the derived key in bytes.
   * $raw_output - If true, the key is returned in raw binary format. Hex encoded otherwise.
   * Returns: A $key_length-byte key derived from the password and salt.
   *
   * Test vectors can be found here: https://www.ietf.org/rfc/rfc6070.txt
   *
```

```php
    * This implementation of PBKDF2 was originally created by https://defuse.ca
    * With improvements by http://www.variations-of-shadow.com
    */
  public static function pbkdf2($algorithm, $password, $salt, $count, $key_length, $raw_output = false)
```

### Card.php

```php
class Card
  public $id_card;
  public $id_module;
  public $id_type;
  public $round;
  public $dimensions_x;
  public $dimensions_y;
  public $position_x;
  public $position_y;
  public $image_file;
  public $forecolor;
  public $backcolor;
  public $text;
  private function __construct(array $data)
  public static function fromId($id_card, $return_format = 'object')
  public static function getByModule($id_module, $round = 'all', $return_format = 'object')
  public static function createNew(array $data, $return_format = 'object')
  public function save()
  public function delete()
```

### CardBehaviour.php

```php
class CardBehaviour
  public $id_behaviour;
  public $id_type;
  public $round;
  public $fits_on;
  private function __construct(array $data)
  public static function fromId($id_behaviour, $return_format = 'array')
  public static function getByType($id_type, $return_format = 'array')
  public static function getByTypes(array $id_types, $return_format = 'array')
  public static function createNew(array $data, $return_format = 'object')
  public function save()
  public function delete()
```

### CardType.php

```php
class CardType
  public $id_type;
  public $id_module;
  public $name;
  public $dimensions_x;
  public $dimensions_y;
  private function __construct(array $data)
  public static function fromId($id_type, $return_format = 'array')
  public static function getByModule($id_module, $return_format = 'array')
  public static function createNew(array $data, $return_format = 'object')
  public function save()
  public function delete()
```

### Database.php

The database model used to communicate with the MySQL server. Based on SMF 2.0's database abstraction layer, (C) 2007 Simple Machines NPO. Used under BSD licence.

```
class Database
  private $connection;
  private $query_count = 0;

  /**
   * Initialises a new database connection.
   * @param server: server to connect to.
   * @param user: username to use for authentication.
   * @param password: password to use for authentication.
   * @param name: database to select.
   */
  public function __construct($server, $user, $password, $name)

  // Fetches a row from a given recordset, using field names as keys.
  public function fetch_assoc($resource)

  // Fetches a row from a given recordset, using numeric keys.
  public function fetch_row($resource)

  // Destroys a given recordset.
  public function free_result($resource)

  // Moves internal pointer in given recordset.
  public function data_seek($result, $row_num)

  // Returns the amount of rows in a given recordset.
  public function num_rows($resource)

  // Returns the amount of fields in a given recordset.
  public function num_fields($resource)

  // Escapes a string.
  public function escape_string($string)

  // Unescapes a string.
  public function unescape_string($string)

  // Returns the last MySQL error.
  public function error()

  // Returns server info.
  public function server_info()

  // Selects a database on a given connection.
  public function select_db($database)

  // Returns the amount of rows affected by the previous query.
  public function affected_rows()

  // Returns the id of the row created by a previous query.
  public function insert_id()

  // Do a MySQL transaction.
```

```php
  public function transaction($operation = 'commit')

  // Function used as a callback for the preg_match function that parses variables into database queries.
  private function replacement_callback($matches)

  // Escapes and quotes a string using values passed, and executes the query.
  public function query($db_string, $db_values = array())

  /**
   * Escapes and quotes a string just like db_query, but does not execute the query.
   * Useful for debugging purposes.
   */
  public function quote($db_string, $db_values = array())

  // Executes a query, returning an array of all the rows it returns.
  public function queryRow($db_string, $db_values = array())

  // Executes a query, returning an array of all the rows it returns.
  public function queryRows($db_string, $db_values = array())

  // Executes a query, returning an array of all the rows it returns.
  public function queryPair($db_string, $db_values = array())

  // Executes a query, returning an array of all the rows it returns.
  public function queryPairs($db_string, $db_values = array())

  // Executes a query, returning an associative array of all the rows it returns.
  public function queryAssoc($db_string, $db_values = array())

  // Executes a query, returning an associative array of all the rows it returns.
  public function queryAssocs($db_string, $db_values = array(), $connection = null)

  // Executes a query, returning the first value of the first row.
  public function queryValue($db_string, $db_values = array())

  // Executes a query, returning an array of the first value of each row.
  public function queryValues($db_string, $db_values = array())

  // This function can be used to insert data into the database in a secure way.
  public function insert($method = 'replace', $table, $columns, $data)

  // This function tries to work out additional error information from a back trace.
  private function error_backtrace($error_message, $log_message = '', $error_type = false, $file = null, $line = null)
```

## Dispatcher.php

```php
class Dispatcher
  public static function dispatch()
```

## Error.php

```php
class Error
```

```
  public $id_entry;
  public $message;
  public $file;
  public $line;
  public $time;
  public $ipaddress;
  public $url;
  public $id_user;
  private function __construct($data)
  public static function fromId($id_entry)
  public static function log(array $data, $return_format = 'bool')
  public static function flushLog()
  public static function getCount()
```

### ErrorHandler.php

ErrorHandling class, contains the static triggerFatalError function used to trigger fatal errors, halting all other processes.

```
class ErrorHandler
  private static $error_count = 0;

  // Triggers a fatal error, presenting the message in $text to the user, unless it contains debug info.
  public static function triggerFatalError($text, $contains_debug_info = false, $needs_logging = true, $format = 'html')
  public static function trigger400()
  public static function trigger403()
  public static function trigger404()

  // Kicks a guest to a login form, redirecting them back to this page upon login.
  public static function kickGuest($message, $url = '')
  private static function logError($error_message = '', $file = '', $line = 0)
  public static function handleError($error_level, $error_string, $file, $line)
```

### Form.php

```
class Form
  public $request_method;
  public $request_url;
  public $content_above;
  public $content_below;
  private $fields;
  private $data;
  private $missing;
  // NOTE: this class does not verify the completeness of form options.
  public function __construct($options)
  public function verify($post)
  public function setData($data)
  public function getFields()
  public function getData()
  public function getMissing()
```

### GenericTable.php

```
class GenericTable extends PageIndex
  protected $header = [];
  protected $body = [];
  protected $page_index = [];
  protected $title;
  protected $title_class;
  protected $tableIsSortable = false;
  protected $recordCount;
  protected $needsPageIndex = false;
  protected $current_page;
  protected $num_pages;
  public $form_above;
  public $form_below;
  public function __construct($options)
  private function generateColumnHeaders($options)
  private function parseAllRows($rows, $options)
  protected function getLink($start = null, $order = null, $dir = null)
  public function getOffset()
  public function getArray()
  public function getHeader()
  public function getBody()
  public function getTitle()
  public function getTitleClass()
```

### Group.php

```
class Group
  public $id_group;
  public $id_session;
  public $creation_time;
  public $name;
  private function __construct(array $data)
  public static function fromId($id_group, $return_format = 'object')
  public static function getAll($return_format = 'object')
  public static function createNew(array $data, $return_format = 'object')
```

### GroupMembership.php

```
class GroupMembership
  public $id_group;
  public $id_user;
  public $join_date;
  private function __construct(array $data)
  public static function getMembershipBySession()
  public static function joinGroup($id_user, $id_group, $new_name = '')
```

### Guest.php

Guest model; sets typical guest settings to the common attributes.

```
class Guest extends User
  public function __construct()
  public function updateAccessTime()
```

### Instance.php

```
class Instance
  public $id_instance;
  public $id_card;
  public $id_group;
  public $container;
  public $position_x;
  public $position_y;
  private function __construct(array $data)
  public static function fromId($id_instance, $return_format = 'object')
 public static function getByGroup($id_group, $container = 'all', $return_format
      = 'array')
 public static function getByGroups($container = 'all', $return_format = 'array'
     )
 public static function createNew($id_group, $return_format = 'array',
     $from_round = 1, $to_round = 1)
 public function save()
 public static function update($data)
 public function delete()
```

### Label.php

```
class Label
  public $id_label;
  public $id_module;
  public $container;
  public $position_x;
  public $caption;
  private function __construct(array $data)
  public static function fromId($id_label, $return_format = 'array')
  public static function getByModule($id_module, $container = 'all',
      $return_format = 'array')
  public static function createNew(array $data, $return_format = 'object')
  public function save()
  public function delete()
```

### Member.php

```
class Member extends User
  private function __construct($data)
  public static function fromId($id_user)
  public static function fromSlug($slug)

  // Creates a new member from the data provided.
  public static function createNew(array $data)

  // Updates the member using the data provided.
  public function update(array $new_data)

  // Deletes the member.
  public function delete()

  /**
   * Checks whether an email address is already linked to an account.
   * @param emailaddress to check
   * @return false if account does not exist
   * @return user id if user does exist
   */
  public static function exists($emailaddress)
  public function updateAccessTime()
  public static function getCount()
  public function getProps()
```

### Module.php

```
class Module
  public $id_module;
  public $title;
  public $description;
  public $num_rounds;
  public $canvas_x;
  public $canvas_y;
  public $is_active;
  private function __construct(array $data)
  public static function fromId($id_module, $return_format = 'object')
  public static function getAll($return_format = 'object')
  public static function getPairs()
  public static function createNew(array $data, $return_format = 'object')
  public function save()
  public function delete()
  public function getCards($round = 'all', $return_format = 'array')
  public function getTypes($return_format = 'array')
  public function getLabels($container = 'all', $return_format = 'array')
```

### ModuleSession.php

```
class ModuleSession
  public $id_session;
  public $id_module;
  public $current_round;
  private function __construct(array $data)
  public static function fromId($id_session, $return_format = 'object')
  public static function getAll($return_format = 'object')
  public static function createNew(array $data, $return_format = 'object')
  public function delete()
```

### NotAllowedException.php

```
class NotAllowedException extends Exception
```

### NotFoundException.php

```
class NotFoundException extends Exception
```

### PageIndex.php

```
class PageIndex
  protected $page_index = [];
  protected $current_page = 0;
  protected $items_per_page = 0;
  protected $needsPageIndex = false;
  protected $num_pages = 0;
  protected $recordCount = 0;
  protected $start = 0;
  protected $sort_order = null;
  protected $sort_direction = null;
  protected $base_url;
  protected $index_class = 'pagination';
  protected $page_slug = '%AMP%page=%PAGE%';
  public function __construct($options)
  protected function generatePageIndex()
  protected function getLink($start = null, $order = null, $dir = null)
  public function getArray()
  public function getPageIndex()
  public function getPageIndexClass()
  public function getCurrentPage()
  public function getNumberOfPages()
  public function getRecordCount()
```

### Registry.php

```
class Registry
  public static $storage = [];
  public static function set($key, $value)
  public static function has($key)
  public static function get($key)
  public static function remove($key)
```

### Session.php

```
class Session implements SessionHandlerInterface
  public static function start()
  public static function resetSessionToken()
  public static function validateSession($method = 'post')
  public static function getSessionToken()
  public static function getSessionTokenKey()
  public function read($session_id)
  public function write($session_id, $session_data)
  public function destroy($session_id)
  public function gc($maxlifetime)
  public function open($save_path, $name)
  public function close()
```

### User.php

User model; contains attributes and methods shared by both members and guests.

```
abstract class User
  protected $id_user;
  protected $first_name;
  protected $surname;
  protected $emailaddress;
  protected $creation_time;
  protected $last_action_time;
  protected $ip_address;
  protected $is_admin;
  protected $is_logged;
  protected $is_guest;

  // Returns user id.
  public function getUserId()

  // Returns first name.
  public function getFirstName()

  // Returns surname.
  public function getSurname()

  // Returns full name.
  public function getFullName()

  // Returns email address.
  public function getEmailAddress()

  // Have a guess!
  public function getIPAddress()

  // Returns whether user is logged in.
  public function isLoggedIn()

  // Returns whether user is a guest.
  public function isGuest()

  // Returns whether user is an administrator.
  public function isAdmin()
```

## D.2 Implementation: PHP controller classes

### EditCard.php

```
class EditCard extends HTMLController
  public function __construct()
```

### EditCardType.php

```
class EditCardType extends HTMLController
  public function __construct()
```

### EditLabel.php

```
class EditLabel extends HTMLController
  public function __construct()
```

### EditModule.php

```
class EditModule extends HTMLController
  public function __construct()
```

### EditUser.php

```
class EditUser extends HTMLController
  public function __construct()
```

### GetCards.php

Contains the module grid controller.

```
class GetCards extends JSONController
  public function __construct()
```

### HTMLController.php

The abstract class that allows easy creation of html pages.

```
abstract class HTMLController
  protected $page;
  public function __construct($title)
  public function showContent()
```

### InstanceServer.php

Implements our instance server on top of the WebsocketServer class.

```
class InstanceServer extends WebsocketServer
  private $sessions;
  private $members;
  protected static $allowedMethods;
  public function __construct()
  public function reloadSessions($forced = false)
  public function authenticate($params)
  public function sendAnnouncement($params)
  public function listSessions($params)
  public function registerForSession($params)
  public function unregisterForSession($params)
  public function showGroups($params)
  public function createGroup($params)
  public function joinGroup($params)
  public function getCanvas($params)
  public function updateInstance($params)
    public function advanceRound($params)
```

### JSONController.php

The abstract class that allows easy creation of JSON replies.

```
class JSONController
  protected $payload;
  public function showContent()
```

### ListModules.php

Lists all available modules.

```
class ListModules extends HTMLController
  public function __construct()
```

### ListSessions.php

Lists all available module sessions.

```
class ListSessions extends HTMLController
  public function __construct()
```

### Login.php

The controller class that allows users to log in. Passes user input to the Authentication model. Redirects user after logging in.

```
class Login extends HTMLController
  public function __construct()
```

### Logout.php

```
class Logout extends HTMLController
  public function __construct()
```

### ManageUsers.php

```php
class ManageUsers extends HTMLController
  public function __construct()
```

### NewCard.php

```php
class NewCard extends JSONController
  public function __construct()
```

### NewLabel.php

```php
class NewLabel extends HTMLController
  public function __construct()
```

### NewSession.php

```php
class NewSession extends HTMLController
  public function __construct()
```

### SignUp.php

```php
class SignUp extends HTMLController
  public function __construct()
```

### ViewLandingPage.php

Contains the controller that creates the user landing page.

```php
class ViewLandingPage extends HTMLController
  public function __construct()
```

### ViewModule.php

Loads and displays the grid page.

```php
class ViewModule extends HTMLController
  public function __construct()
```

### WebsocketServer.php

Creates a websocket server based on Ratchet's MessageComponentInterface.

```php
class WebsocketServer implements MessageComponentInterface
  protected $clients;
  protected $client;
  protected static $allowedMethods = [];
  public function __construct()
  public function onOpen(ConnectionInterface $conn)
  public function onMessage(ConnectionInterface $from, $msg)
  public function onClose(ConnectionInterface $conn)
  public function onError(ConnectionInterface $conn, \Exception $e)
  public function send(ConnectionInterface $client, $event, array $data)
  private function sendData(ConnectionInterface $conn, array $data)
  private function sendError(ConnectionInterface $conn, $msg)
  protected function println($msg, $conn = -1)
```

## D.3 Implementation: PHP template classes

### Dialog.php

Contains the dialog template, used for jQuery dialogs.

```
class Dialog extends SubTemplate
  private $_title;
  private $_data;
  protected $_forms;
  public function __construct($title, $data)
  protected function html_content()
  public function addForm($id, $form)
```

### DialogForm.php

Contains the dialog form template, used in jQuery dialogs.

```
class DialogForm extends SubTemplate
  public function __construct($form)
  protected function html_content($exclude = [], $include = [])
  protected function renderField($field_id, $field)
```

### DummyBox.php

```
class DummyBox extends SubTemplate
  public function __construct($title = '', $content = '', $class = '')
  protected function html_content()
```

### EditCardDialog.php

Contains the edit card dialog template, used for jQuery dialogs.

```
class EditCardDialog extends Dialog
  public function __construct($title, $card)
  protected function html_content()
```

### EditCardTypeDialog.php

Contains the edit card type dialog template, used for jQuery dialogs.

```
class EditCardTypeDialog extends Dialog
  public function __construct($title, $type)
  protected function html_content()
```

### EditModuleDialog.php

Contains the edit model dialog template, used for jQuery dialogs.

```
class EditModuleDialog extends Dialog
  public function __construct($title, $module)
  protected function html_content()
```

### FormView.php

Contains the form template.

```
class FormView extends SubTemplate
  public function __construct(Form $form, $title = '')
  protected function html_content($exclude = [], $include = [])
  protected function renderField($field_id, $field)
```

### GridView.php

Contains the timeline grid template.

```
class GridView extends SubTemplate
  public function __construct($title, $module)
  protected function html_content()
```

### LandingPage.php

```
class LandingPage extends SubTemplate
  protected function html_content()
```

### LogInForm.php

```
class LogInForm extends SubTemplate
  private $error_message = '';
  private $redirect_url = '';
  private $emailaddress = '';
  public function setErrorMessage($message)
  public function setRedirectUrl($url)
  public function setEmail($addr)
  protected function html_content()
```

### MainTemplate.php

```
class MainTemplate extends Template
  private $title = '';
  private $page_id;
  private $classes = [];
  public function __construct($title = '')
  public function html_main()
  public function setArea($id)
  public function setPageId($id)
  public function addClass($class)
  public function removeClass($class)
```

### ModulesList.php

Template that lists all available modules.

```
class ModulesList extends SubTemplate
  public function __construct(array $modules)
  protected function html_content()
```

### NewSessionForm.php

```
class NewSessionForm extends SubTemplate
  public function __construct(array $modules)
  protected function html_content()
```

### Pagination.php

```
class Pagination extends SubTemplate
  private $index;
  public function __construct(PageIndex $index)
  protected function html_content()
```

### SignUpForm.php

```
class SignUpForm extends SubTemplate
  private $error_msg;
  private $full_name;
  private $affiliation;
  private $emailaddress;
  private $emailaddress2;
  protected function html_content()
  public function setFormData($data = array())
  public function setErrorMessage($err_msg)
```

### SubTemplate.php

```
abstract class SubTemplate extends Template
  public function __construct($title = '')
  public function html_main()
  abstract protected function html_content();
```

### TabularData.php

Contains the template that displays tabular data.

```
class TabularData extends Pagination
  public function __construct(GenericTable $table)
  protected function html_content()
  protected function showForm($form)
```

### Template.php

```
abstract class Template
  protected $_subtemplates = array();
  abstract public function html_main();
  public function adopt(Template $template, $position = 'end')
  public function clear()
  public function pass($id, $data)
```

## D.4 Implementation: JavaScript function prototypes

### AppView

```
function AppView(options)
  AppView.prototype.setTitle = function (title)
  AppView.prototype.setBody = function (html)
  AppView.prototype.clearToolbar = function (bar)
  AppView.prototype.addButton = function (bar, caption, click)
```

### GridView

```
function GridView(options)
  GridView.prototype.setDimensions = function(dim)
  GridView.prototype.appendCards = function(cards)
  GridView.prototype.reloadCard = function(cardId, data)
  GridView.prototype.addCardEvents = function(card)
  GridView.prototype.openSettings = function()
  GridView.prototype.createNewCard = function()
  GridView.prototype.deleteCard = function(id)
```

### LabelBar

```
function LabelBar(options)
  LabelBar.prototype.appendLabel = function(data)
  LabelBar.prototype.addBarEvents = function()
  LabelBar.prototype.addLabelEvents = function(label)
```

### Card

```
function Card(card, doPositions)
```

### EditSettingsDialog

```
function EditSettingsDialog(id, grid)
```

### EditCardDialog

```
function EditCardDialog(id, grid, module)
```

### Sidebar

```
function Sidebar(options)
  Sidebar.prototype.show = function(callback)
  Sidebar.prototype.hide = function(callback)
  Sidebar.prototype.toggle = function()
  Sidebar.prototype.initializeRound = function(round)
  Sidebar.prototype.switchRound = function(round)
  Sidebar.prototype.fetchCards = function()
  Sidebar.prototype.appendCards = function(cards)
  Sidebar.prototype.flush = function(callback)
  Sidebar.prototype.addCardEvents = function(card)
  Sidebar.prototype.reloadCard = function(cardId, data)
  Sidebar.prototype.createNewCard = function()
  Sidebar.prototype.deleteCard = function(id)
```

### RoundPicker

```
function RoundPicker(options)
  RoundPicker.prototype.getDOM = function()
  RoundPicker.prototype.markActive = function(round)
```

### LabelBar

```
function LabelBar(options)
```

### Card

```
function Card(card, doPositions)
```

### Sidebar

```
function Sidebar(options)
  Sidebar.prototype.setCabinetSize = function()
  Sidebar.prototype.show = function(callback)
```

```javascript
  Sidebar.prototype.hide = function(callback)
  Sidebar.prototype.toggle = function()
  Sidebar.prototype.destroy = function()
  Sidebar.prototype.appendInstances = function(cards)
  Sidebar.prototype.addCardEvents = function(card)
  Sidebar.prototype.toGrid = function(card, gridPos)
```

### GroupCanvas

```javascript
function GroupCanvas(app, options, socket)
  GroupCanvas.prototype.reset = function()
  GroupCanvas.prototype.setDimensions = function(dim)
  GroupCanvas.prototype.appendCards = function(cards)
  GroupCanvas.prototype.appendInstance = function(card, atPos)
  GroupCanvas.prototype.sendUpdate = function(card)
  GroupCanvas.prototype.snapToGrid = function(pos)
  GroupCanvas.prototype.appendInstances = function(cards)
  GroupCanvas.prototype.reloadCard = function(cardId, data)
  GroupCanvas.prototype.addCardEvents = function(card)
```

### GroupPicker

```javascript
function GroupPicker(app, groups)
  GroupPicker.prototype.reset = function()
```

### SessionPicker

```javascript
function SessionPicker(app, sessions)
```

### Socket

```javascript
function Socket(address, open, debug)
  Socket.prototype.send = function (event, params, callback)
  Socket.prototype.on = function (event, listener)
  Socket.prototype.unbind = function (listener)
  Socket.prototype.unbindAll = function ()
  Socket.prototype.close = function ()
```