Eye-Tracking in User Experience Testing

Design, Implementation and user Testing of the Front end Application

Daniel Lindgren
2015

Bachelor of Science in Engineering Technology
Computer Game Programming

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
"Can eye-tracking based user tests be used to enhance and improve the development process of web applications from a user experience perspective?"

Eye-Tracking in User Experience Testing

Design, implementation and user testing of the front end application.

Daniel Lindgren
Luleå University of Technology
2015-06-14

Supervisor: Patrik Holmlund
Abstract

Tieto is one of the leading providers of information and communication technology services in Europe. To be able to deliver high quality products to their customers, the software developers need as much feedback as possible during the development process. During this project, the use of eye-tracking in user tests is tried out, to see if the information gathered is of any help in enhancing the development of future products. This is made using the Tobii EyeX eye-tracking device, in cooperation with recorded mouse movement, mouse and keyboard events and sound. A Chrome extension is designed and used as an interface to a C# server, which communicates directly with the Tobii EyeX framework. The extension is made using the web technologies of HTML5, CSS, JavaScript and JSON. The end result is a fully working recorder and player, which can record all of the things mentioned above, and then simulate the entire recording on any suitable web application. It is also able to calculate and show a lot of different statistics on the web application, making it possible for the software developers to see and hear in detail, exactly what the user was doing. User tests will be made using the program, so that conclusions can be made about the usability for this kind of product.

Sammanfattning

Acknowledgements

I would like to thank everyone that has helped me during this project. First of all, my colleague Daniel Johansson for his dedication and hard work the last two months. This could of course not have been done without you. Many thanks to Andreas Blom and the rest of Team Chronos, and everyone else at Tieto Healthcare & Education for letting us work at your fantastic company. Special thanks to Magnus Vestin at Tieto for setting this all up to begin with.

I would also like to thank Patrik Holmlund, Johannes Hirche and all the other teachers at Luleå University of Technology, Campus Skellefteå, for helping out with everything these last three years.

Abbreviations and Terms

- C# - A multi-paradigm programming language developed by Microsoft.
- CSS - Cascading Style Sheet
- GUI - Graphical User Interface
- HTML - Hyper Text Markup Language
- JavaScript - A dynamic prototype-based scripting language often used for programming websites.
- JSON - JavaScript Object Notation
- TCP - Transfer Control Protocol
- UX - User Experience
1 Introduction

When developing computer software using agile software development [1], user acceptance testing is an important part of the process. These tests are designed to help the developer understand how the user is using the software [2]. With that information, the developer can design the software to better suite the user’s needs.

Naturally, when making user tests of any kind, the gathered data is desired to be as exact as possible. With modern technology, precision in data gathering of any kind becomes easier and easier, making way for new interesting methods of testing software. Imagine a scenario where a developer could send his customer a product that is under development, have her test it using automated user test data gathering, and then analyze it without ever having to leave his desk. Even though this might be unrealistic, since the customer most likely does not have the required hardware, a more realistic way to do this would be by having software developers have their test groups use dedicated test environments. These would be equipped with modern recording equipment that could gather a lot of useful data about the users behavior, which might help increasing the understanding of how users think and behave depending on the applications layout and how it gives feedback to the user. Hopefully, this would help develop applications that are more in line with what the customer needs, even if they themselves don’t know it.

1.1 Tieto

Tieto is the largest IT service company in the Nordic region, and provides services in the entire IT cycle. They also provide global services for product development in the areas of information- and communication technology. The company was founded in 1968 and it has its headquarters located in Helsinki, Finland [3].

The company has fully adopted the thinking of Lean Software Development (LSD) [4]. Lean is not only a recommended working method at development level, but a philosophy that is found on all levels throughout the company. Another methodology used at Tieto in combination with (or maybe rather as an extension to) LSD is Scrum, which is an iterative and incremental agile software development method for managing product development [5]. It was Tieto’s wish that Scrum would be used as the main working method for this project, to fit into the companies work flow.

1.2 Lean Software Development

It is hard to define LSD, as there are no specific LSD method or process. There are however, a few larger schools of thought within LSD, the largest and arguably leading school being the Lean System Society. In 2011, the Lean Systems Society published a set of values [4] that helps us get a basic understanding of the thought behind LSD. These values are:

- Accept the human condition
- Accept that complexity and uncertainty are natural to knowledge work
- Work towards a better economic outcome
- While enabling a better sociological outcome
- Seek, embrace and question ideas from a wide range of disciplines
- A values-based community enhances the speed and depth of positive change
By adapting these values, any company could hope to enhance both their economical and sociological outcome, making both staff and stakeholders happy.

### 1.3 Scrum

The leading agile product development framework is Scrum, which provides a foundation and path to delivering business goals in a collaborative manner between developer and product owner. Scrum was initially made with software development in mind, but other industries have applied this framework as well, including education, marketing, operations and more [5].

The concept upon which Scrum would spring from, was first introduced in 1986 by Hirotaka Takeuchi and Ikujiro Nonaka in “the New New Product Development Game” (Harvard Business Review, January/February 1986). They defined it as a “flexible, holistic product development strategy” and proposed it would be a fast, flexible way of developing products [5].

A few noticeable key elements of Scrum that will be used throughout this report are:

- **Product Owner** - The person who holds the vision for the project. In this case, it is the project supervisor at Tieto.
- **Product backlog** - A list of ideas for the product, and how they should be prioritized.
- **Sprint** - An iteration in the production process, in this case a period of two weeks.

### 1.4 Background

One of the departments at Tieto is **Healthcare & Welfare**, and one of their fields is web applications for educational institutions. They have already made tests with applications that uses eye-tracking for several purposes, and would now like to see if the technology can be used in general during user tests, to help gather valuable data that will make the development process smoother in ways that were not possible before.

Earlier, the way the company has been making user tests, is through so called think-aloud tests. This means that the user sits down with personell from the development department and describes how she is using the program as she is using it. This includes what she sees, thinks and why she makes the decisions she makes. As a result, the information gathered tells what the user perceives that she is doing when using the software, rather than what the user is actually doing [6]. Gathering this data with eye-tracking gives an “unfiltered” version of the data, and may show the developers things that the test subject did not notice herself [7].

Recently, the technique of eye-tracking [8] has entered the commercial market, making eye-tracking hardware available at lower prices than before. This makes the idea of eye-tracking based user tests a reality. With this technique, it is possible to collect the exact information about how the user’s eyes move across the screen when testing computer software.

### 1.5 Goal and Purpose

The goal of this project is to develop an application that uses the Tobii EyeX eye-tracking device [9] to record the user’s eye movement while testing web applications. This data should be able to be presented on top of the web application in form of a heat map, as this is a clear way of presenting the areas that have been viewed by the user during the test. Mouse movement should be recorded in a similar way, so that differences between eye and mouse movement can be calculated and used as statistics. To make it easier for the developers to analyze the test results,
sound recording will also be possible, which will help in getting a further understanding of how the user thinks when using a web application. The sound recording is also a good tool for knowing when a user is given a specific task during a user test, if such are given. It is also necessary to record mouse clicks and keyboard input, to make it possible to authentically replay a recorded scenario.

The company is interested in getting as much statistics out of the tests as possible, this being things like:

- Time spent on page.
- Percentage of page seen.
- First fixation point.
- Point most fixated.
- Number of fixations points.

By using this data in combination with heat maps, mouse movement recordings, mouse clicks, and a graphical view over where the user has fixated her eyes, Tieto is quite confident that conclusions about user behavior can be made. With this information available to them, developers may be able to compare user tests, evaluate exact eye movement, and get hold of other useful test data that can improve the development process of future software.

1.6 Related Work

Eye-tracking has been used in user tests before, but mostly for specific tests, not in a such a broad perspective as in this project. Often, the technology is used to gather some sort of proof of a hypothesis. Even at Tieto, eye-tracking has been used to test specific applications earlier.

Tieto made an attendance registration application for kindergarten, and used eye-tracking to test differences in the user’s behaviour depending on the design of the application.

Back in 2000, Jakob Nielsen wrote an article about a study made by the Poynter Institute [12], of how people read news on the web. They found out that text attracts attention before graphics. Of the users’ first three fixation points on a page, only 22% were on graphics and 78% were on text. In general, users were first drawn to headlines, article summaries and captions. They often did not look at the images at all until the second or third visit to a page. Another interesting finding made in the study, was that users often took a brief look at an article, rather than to read the full article. Even when reading a “full” article, the users only read about 75% of the text.

Penzo wrote an article in 2005 [13] regarding eye-tracking tests, and how they could be used to show which parts of the user interface that is visible to the user, and which parts seem to be invisible - not just by observing users, but also by analyzing their gaze plots. Even back then they used heat maps to display the results of these gaze plots. He also stated that a very effective methodology combines standard usability evaluation techniques - such as think-aloud tests - with the latest eye-tracking capabilities.

In December 2011, Breeze [14] published an article about the ability that eye-tracking offers people working with user experience (UX). He states that it makes it possible to leave the participant alone during a test to focus on the task at hand, and therefore enables capturing of real physiological data about their conscious and unconscious experiences. Compared to think
aloud tests, eye-tracking is able to show where the participants actually looked, not only where they think that they looked. He also stated that eye-tracking could be used to analyze how users learned how to use an interface, by looking at the order of the eye fixations.

### 1.7 Method

The implementation process for this project has been iterative, in tune with the concept of agile software development. When the project started, the product owner had already made a basic backlog, containing a few items concerning the main objectives of the project. As the project progressed, the backlog has been extended to include more than just eye-tracking. The final product also includes mouse and keyboard events and sound recording.

Every sprint has been divided into sessions. First a thorough research is done, to find out what is necessary about the tasks ahead. Then the solutions to the tasks are designed and implemented. When the implementation is complete, everything is tested to see if it is working as expected. After that, things that can be improved gets improved, and the product is made ready for demoing. By showing the product to other people, their views can be taken in, making way for improvements in the next sprint. The workflow is described in figure 1.1.

![Figure 1.1 The iteration loop that is the development process, from research to demo.](image)

The project can be divided into three parts, the first two being the server and the client side of the application. The Tobii EyeX will communicate directly with a C# server, as supported by the Tobii EyeX framework. This report will however focus on the client side of the application. For more information about the server, see [15].

The client is a Chrome Extension written in HTML5 and JavaScript, and its main function is to work as a graphical user interface (GUI) to interact with the server. Through the GUI, the user and the developer will be able to send input to the server, load earlier recorded tests, record tests, watch recorded tests and see statistics about them. The use of HTML5 and JavaScript was natural, since those are the techniques commonly used at Tieto.

The communication between the client and the server happens through a WebSocket connection. WebSocket is a protocol providing full-duplex communications channels over a single transmission control protocol (TCP) connection [10]. This way, the client can send requests to the server, and receive data through the same connection.

The third part of the project is to make user tests, gathering as much data as possible, so that conclusions can be made about whether eye-tracking can be used in user tests, enhancing the development process of web applications. These tests are done internally at Tieto, so that their applications are prioritized when it comes to compatibility, although the design is meant to support all possible web applications.
1.8 Tobii EyeX

The Tobii EyeX eye tracking device [9] is a hardware for measuring where a person is looking on a computer screen. It has been developed by the Swedish company Tobii, which was founded in 2001. The Tobii EyeX is limited to screens no larger than 27”, and it needs to be connected through USB 3.0.

The eye tracking device incorporates near-infrared micro projectors, optical sensors and image processing. The micro projectors creates reflection patterns on the eyes. Image sensors registers the image of the user, and the projection patterns, in real time. Image processing is used to find features of the user, the eyes and projection patterns. Mathematical models are used to exactly calculate the eyes’ position and gaze point.

1.9 Social, Ethical and Environmental Considerations

When tests are made with the eye-tracking recorder, the only mandatory information that the user has to apply is the name of the application, and the name of the one testing it. It is however possible to write anything into the latter field, so the users will not have to use their real names. In this way, tests can be done anonymously. In addition, user data is only saved locally on the computer where tests are performed, meaning that sharing of test data can be done in any desired way, using secure and encrypted file sharing services. The data sent between the client and the server is also encrypted, even though they only send data back to the localhost. The eye-tracking recorder may not be suitable for testing applications containing classified information, as it records keyboard events such as password entries.

From an environmental perspective, the use of this application could cut down the requirement of travelling a lot. Instead of having personnel from the development team travel to a customer on the other side of the planet, the customer can do the testing herself, sending the result back to the development team through the internet.
2 Design and Implementation

The client side of the eye-tracking recorder is made as a Chrome extension, making it usable for all web applications that can be used with Google Chrome. Its task is to act as a GUI and a controller, sending the user’s requests to the server, starting new tests, loading saved data, displaying that data and present statistics. The reason for making the client as a Chrome extension is basically that it was Tieto’s wish from the beginning. The extension is also responsible for recording mouse and keyboard input. Mouse and keyboard events are saved into lists with timestamps every time they occur, and are sent to the server for saving when the test is finished. This needs to be done in the extension, since it is here we are actively doing stuff when recording. It is interesting to know how the browser registers our actions, since it is there we will simulate the test after the recording is done.

2.1 Building the Chrome Extension

Extensions are small software programs than can modify and enhance the functionality of the Chrome browser. They are written by using web technologies such as HTML, JavaScript and CSS. Extensions bundle all their files into a single file that the user downloads and installs. This bundling means that, unlike ordinary web applications, extensions don’t need to depend on content from the web.

A Chrome extension consists of the following files:

- A manifest file (written in JSON format).
- One or more HTML files.
- Optional: One or more JavaScript files.
- Optional: Any other files your extension needs - for example, image files.

2.1.1 Extension Manifest

The manifest file, called manifest.json, gives information about the extension, such as important files and capabilities that the extension might use. In the case of this project, the manifest.json file contains the following information:

- Extension name - the name of the extension.
- Version - the current version number.
- Description - a short description of the extension.
- Browser action - default properties of the extension, in this case the location of the default popup (see The Popup below) and the location of the extensions icon image.
- Icons - the location of all icon files used in the extension. The icon which opens the extension in the Chrome browser should be 16*16 pixels, while the icon inside the extension browsing tab in the Chrome settings should be 48*48 pixels.
- Background - defines the html-file that will run in the background. This “page” is not displayed in any way, but as the name implies, runs in the background all the time. In this project, the background.html-file has only one job to do, and that is to load a set of JavaScript files.

Web Accessible Resources - Specifies the path of packaged resources that are expected to be usable in the context of a web page. For example, in this project, the extension injects a content
script with the intention of displaying information formatted by a CSS-file residing in the current web page. To access this file, its path needs to be specified here.

**Commands** - Defines keyboard commands which the extension answers to, making it easy to start recordings without opening the GUI every time.

**Permissions** - Specifies the URLs that the extension has permission to access and affect by injection of content scripts.

The Chrome extension’s functionality is handled by three sets of JavaScript files, which are working at three different locations. These three parts are the background scripts, the popup scripts and the content scripts (See figure 2.1). Background scripts run in the background along with the browser itself, and popup scripts are run when the extensions popup window is being displayed. When the popup window is closed, these scripts and all the information that they hold is destroyed. Content scripts are scripts that are injected into a browser tab, and run along with the web page it holds.

![Chrome Browser Diagram](image)

*Figure 2.1 The relation between the three parts of the extension*

2.1.2 The Background

The background scripts are persistent scripts that run in the background along with the browser, meaning that they run as long at the browser is running. When the browser starts running the extension at startup, an html-file is being loaded, which includes all the JavaScript-files that are supposed to run in the background. All of these scripts have a task of their own, and they are described below.

**Command.js** - This script basically catches the keyboard commands specified in the manifest.json-file and executes their functionality.

**Display.js** - Whenever a recorded test is to be displayed in the browser, this script is responsible for communicating with the content script, telling it what to display and how to do it. This script continuously checks if the extension has connection to the content scripts inside the current browser tab. If not, it injects new versions of those scripts. Display.js will then, when it receives
such requests, send new test data to the content scripts, tell it to start, pause, resume and stop animation, and tell it to clear the canvas onto which the content script paints the heat map.

**Keyrecorder.js** - This script is responsible for saving every keyboard event that occurs inside the browser during recordings. There is a content script (injectedtabinfo.js) that is responsible for sending the key code and timestamp of these events when they occur, and this information is then handled by the key recorder and put into an array. When the recording is complete, this data is sent to the server to be stored.

**Messagehandler.js** - As a lot of messages are being sent - not only between the server and the client, but within the client as well - a message handler was written to, in an object oriented way, manage both incoming and outgoing messages. There are two types of incoming messages, these being messages from the server and messages from the popup scripts. Those that come from the server are separated with message types, and executes some kind of functionality inside the extension. The extensions internal messages are written in a slightly different way. They include two things, being the name of the target script, and the name of the message. These two parts are separated with a double colon, and should look something like this example: “mouserecorder::startRecording”.

Both the client and the server have a limited stream buffer where they receive messages, meaning that if too many messages are sent at the same time or if the messages are too big, the buffer gets corrupted. Therefore, the message handler must do two things:

- Check the message size. If it is too big, the message is divided into parts which are sent separately (or added to the message queue, see below).
- Send messages via a queue. When a message is to be sent, it is added to the message queue. This queue is checked every 25 milliseconds, and if it contains at least one message, the first message in the queue is sent through the WebSocket. As the message is sent, it is removed from the queue, placing message at the second position in the queue first.

**Mouserecorder.js** - The mouse recorder works in a similar way as the key recorder. There is a content script that catches every mouse event and sends its info from the browser tab into the extension. The mouse recorder will try to save mouse events at 60 frames per second (fps), which means that the mouse’s coordinates will be pushed into an array at an interval of 16.67 milliseconds. If a click event has occurred since the last update, this will be saved as well.

**Persistentpopupvariables.js** - When the popup window is closed, everything that it contains is being destroyed, making it a little hard to use it as a GUI. For example, if you start a recording, the record button is being changed into a pause button. However if you close the window, which you might want to do to access the entire web page displayed below it, the information telling what kind of buttons the popup should contain is destroyed, therefore the pause button will become the default record button again, even if we are still recording. To solve this problem, a script in the background part of the extension is saving all this information. Even if the popup window is closed, the background scripts are still up and running, so when we open the popup window again, all this information is sent back so that it can be restored as it was. It is therefore important that this script receives updated information each time something is changed inside
the popup window, for example user information, recording/playing status, connection to the server and a lot more.

**Tabinfo.js** - When recording user input from a web application, a lot of information is required about it and the tab that is running it. All this information is gathered by a content script, which sends the information to this script to be handled. An example of this is the current mouse position, which is being stored inside mouserecorder.js. Every time the mouse position changes, this script gets a message from a content script about the new position, and then updates the mouse position variables in mouserecorder.js.

**Websocket.js** - The last of the background scripts is the script responsible for the actual communication between the client and the server. As long as there is no connection established, the script will look for a server to connect to. When a server is found, all incoming messages is forwarded to the message handler, and all outgoing messages received from the message handler is sent through the WebSocket.

### 2.1.3 Content Scripts

To be able to access the content of a web page, the extension needs to inject JavaScript-files directly into the browser tab, running them alongside the currently loaded web page. When injected, these scripts run as normal, and can communicate with the extension by setting up a channel between them and send messages through a virtual port (see figure 2.2).

**Injecteddisplay.js** - This script is responsible for everything that has to do with displaying recorded content. It gets a set of data from the extension via display.js. When the script receives an animation request, it creates a canvas on top of the current web application, starting to draw a heat map from the recorded data, frame by frame. To get a feeling of real time, timestamps are used to set the time between two frames.

This script is also responsible for simulating mouse clicks and keyboard input. When the animation has come to the point where the next mouse click’s or keyboard input’s timestamp has passed, an event is created and dispatched to the element currently in focus. An element gets focus if a mouse click event occurs upon it. This way, input fields can be written to, and submit events triggered. This will result in a correct simulation of what was being done when the test was recorded.
Injectedtabinfo.js - When recording what is being done on a web page, there are a lot of different parameters to take into consideration. For example, the eye-tracking recorder only knows the size of the screen you are looking at. So, if we are recording our eye movement on a web page that is scrollable, it would only know the eyes’ location on the screen, not on the scrolled web page. Therefore, the extension needs to feed the server with information about the current scroll position continuously during the recording process. This script listens for such events to occur, and sends this information to tabinfo.js in the background part of the extension, which forward this information to the server through the message handler. It also listens for mouse movement, mouse clicks and keyboard events, and feeds the mouse and key recorders with this information. Besides that, it also listens for an event that happens when a web page is starting to unload. This means that we are starting a switch to a new page in the web application, and triggers a pause in the recording/animation. Since the content scripts will be destroyed along with the current page, the information about their statuses is stored in the background scripts and passed along with the new injected scripts on the next page.

2.1.4 The Popup
For the user, this part of the extension is the most essential one. It is the GUI itself and is, with the exception of keyboard shortcuts, the only way for the user to communicate with the server. The popup can be accessed by clicking the eye icon in the top right corner of the Chrome browser (see figure 2.3). It consists of four tabs, all responsible for their own part of the extensions functionality.

Every tab runs with at least one JavaScript, which handles the functionality of that particular tab. There is also a script called popup.js, which handles the initialization of all the tabs. It is responsible to get all the saved variables from persistentpopupvariables.js from the extension background, distributing information to the other scripts inside the popup at startup. As explained earlier, this is done every time the popup window is opened, since everything is destroyed each time the popup window is closed.

Figure 2.3 Popup Icon
The record tab (see figure 2.4) has two major responsibilities. First, it contains a form for the user to fill out. There are only two required fields, these being the application name and the user’s name. Optional information is age, gender, occupation/role, location, computer usage and other information. Each time the value inside a field is changed, an updated form is submitted to the background script which saves this information, should the popup window close for some reason. If the popup window is opened again, the information will be loaded back into the form. When the form is filled out, and if the client is connected to the server, a recording session can begin by pressing the start button or using the keyboard command alt+r. When the recording starts, the popup window will automatically close, so that no unnecessary mouse data is being recorded in the beginning of a test, and the same goes for eye fixation. The extensions icon will change to show that a recording session is taking place, and if the icon is being clicked during that time, it will automatically be paused. To resume the recording, the user can either click on the icon again, or press the resume button, resuming the recording session and closing the popup window again. The user can also choose to stop the recording at this time by clicking the stop button, or at any time by using the keyboard command alt+r again.

Whenever a recording session is started, the information from the form is sent to the server, so that it can set up folders and files with the correct names, where it will store the recorded data.

The load test tab is used to load specific tests into the extension, showing statistics and rendering animated representations of the test. When the connection to the server is established, the extension receives all earlier tests saved on the computer, and displays a list of the applications names. If an application name is clicked, the tab will show a list of all the dates that tests have been recorded on that specific application. When a date is clicked, the tab will show a list of all the tests that have been made for that application, on that specific date. Each test is represented by the name of the user and the exact time that the test started. When one of these tests are clicked, a request is sent to the server for all gathered data from that test. It is then loaded into the extension, and the user is forwarded to the player tab. These three levels of test browsing are showed in figure 2.5.
The player tab (see figure 2.6) shows information about the currently loaded test (if any) and lets the user choose what kind of data to display by the use of check boxes. By pressing the animate button, or pressing the keyboard shortcut alt+a, the extension sends a request to the server that it wants to start an animation, and when a response is received, the content scripts starts to animate the eye data as a heat map, and the mouse data as a black mouse pointer, depending on that data the user has chosen to view. The show button will initiate calculation of the heat maps for eye and mouse data and display these in their entirety at once, instead of animating them in real time. The hide button is used to stop the animation/show of the data, and clear the canvas that they are being painted on.

The data from the currently loaded test that is being displayed in this tab contains information such as:

- The name of the application being tested
- The name of the user testing the application and
- The date and time that the test occurred.
It is possible for the user to click on the application name at this point, sending the user back to the load test tab, to the level of test browsing that shows all the dates that a recording occurred on this specific application.

The statistics tab (see figure 2.7) shows statistics for the currently loaded test, such as time on page, percentage of page seen, the coordinates and duration of the first fixation point and the most fixated point, and the number of fixation points. It also contains a button that lets you view all the fixation points directly in the web application. These are represented as round circles, containing an index number, so that the user knows in which order the points were fixated. By hovering the mouse over these points, a popover appears, showing more information about that specific fixation point (see figure 2.8). If two fixation points are so close to each other that the circles they form would overlap, they are merged into one single point.

Although, if such a point is hovered over, the information about all the points that the point was merged from can be found inside the popover.
2.1.5 Client-Server Communication

As pointed out earlier, the server and the client communicates through a WebSocket connection. This is set up as soon at the client finds a server to connect to. Both the client and the server will do continuous tests to see if the connection is still up and running. If the connection is lost, the client will start looking for servers again, until one is found, and then it connects to that server.

To make it easy to create new messages and to make them easy to understand, a message protocol was created (see appendix 1). Every message is a JSON string, containing two things. A message type represented by a number, and the message content represented by a string. In case that objects or arrays are sent as content, these are made into separate JSON strings.

In the case of messages being too large, a third part of the message is being sent. This is a number that tells if this is the first, middle or last part of a message. The number 0 means that this is
the first part of the message, making the receiver wait for more data. The number 1 indicates that it is the second part of the message, but that it is not the last part. The receiver will then add this data to the previous data received. When the number 2 is received, the receiver will add the third and last part as well, and then stop waiting for more data. It will then carry on and parse the concatenated data (see figure 2.9).
3 User Testing

To be able to make some conclusions about whether eye-tracking based user tests actually give the software developers useful information or not, tests with the eye-tracking recorder had to be done with real test subjects. Since Tieto is always developing new web applications, there were always opportunities for user testing directly on their software. This also helped assuring the compatibility between the Chrome extension and real web applications, not only custom made test environments.

3.1 Performing User Tests

The first two test scenarios were designed by one of the UX designers at Tieto, and were designed to put the eye-tracking recorder to test in a real life situation. To be able to gather valuable data, some sort of problem was defined in every scenario. The tests also grew in complexity as the project progressed, making it possible to gather more data and also switch between pages in the later tests.

In these tests, the participants were logged in on a web application, taking on the role as system administrator. They were supposed to complete four objectives in order, the next given as the previous was completed. Before every test, the test environment was set up and hidden, so that the participants had not seen the application before the recording started. This was important, since one of the most important pieces of data that the eye-tracking recorder records is the first point of fixation. If the participants were to take a look at the start page before the test started, this data would be inaccurate and not of any use in the later analyze of the data.

The third and last test was designed by the author of this report, and was designed to answer the question about whether eye-tracking based user tests could tell more about user’s behavior than the participants perceived themselves.

3.1.1 Test 1

The first test was made four weeks into the project, with only eye-tracking and mouse movement data. At this time, tests could only be made on a single page, making it impossible to do tests in a more complex environment. For this test, a single overview page was being monitored, having the participant complete the following objectives:

- Where do I switch role?
- Where do I search for the student Xxxx Xxxxxxxxx, by using the student’s social security number XXXXXXX-XXX.
- Where do I change the student’s phone number?
- Where do I log out?

The major objective in this test was to see how much of the page the participants were actually watching, and how long it took for them to complete the four objectives.
3.1.2 Test 2
The second test was made two weeks after the first test and could, in addition to the data that could be gathered in the first test, handle mouse clicks, keyboard events and sound. This made it possible to access multiple pages, making way for more complex test scenarios than before. Also, it made it possible for the participant to actually perform actions inside the application, like searching for students, logging out etc. Therefore, the objectives were changed from “Where do I do this?” to “Do this!”. Instead of asking the participant to find out where to do something, the objectives could tell the participant to actually do it. This is what the four objectives where changed into:

- Open the help for this page.
- Search for the student Xxxx Xxxxxxxx, by using the student’s social security number XXXXXX-XXXX.
- Where do I change the student’s phone number?
- Where do I log out?

Some objectives were still of the “Where do I…?” kind, because of permission restrictions. The main objective in this test was to see where the participants looked when searching for the help, because the overview page contained two buttons with a question mark as symbol, though one of them also contained the text “Help”.

3.1.3 Test 3
The third test focused more on the differences between the users’ perceived experiences and the results of the eye-tracking tests. A short scenario, where the participants took on the role of a system administrator viewing a student page, was created with the following three objectives:

- Where can you find the students address information?
- Where is the help for this page?
- Where do you log out?

In addition to these objectives, the participants were asked to answers the following questions:

- Where on the screen do you think that your eyes focused first?
- Where do you think that your eyes focused the most?
- How long time do you think the test took?
- Have you participated in user testing of web applications before?
- What kind of testing methods were used?
- What did you think of this method, in comparison to your earlier testing experiences?

Comparing the answers of the first three questions with the data gathered by the eye-tracking recorder would give a sense of how much the perceived user behavior and the recorded data differed. The reason for asking for the test time has nothing to do with eye-tracking itself, but is interesting when comes to how the human mind perceives things when their focus is somewhere else.
4 Result

The question at issue for this project was - “Can eye-tracking based user tests be used to enhance the development of web applications from a user experience perspective”. To answer this question, the design and implementation of a eye-tracking recorder was necessary. To further increase the usability of the recorder, a few other features were added, making it able to do the following things:

- Record eye movement and calculate statistics such as:
  - Percentage of page seen
  - Time spent on page
  - Coordinates and time of first fixation point
  - Coordinates and time of most fixated point
  - Total number of fixation points
- Record mouse movement and simulate mouse click events
- Record keyboard input and simulate their events
- Record sound and play it during simulation of the things mentioned above

The eye-tracking recorder is written as a Chrome extension, which communicates with a C# server that works as an interface to the eye-trackers framework. The server is also responsible for storing data and recording sound. The Chrome extension is responsible for recording mouse and keyboard data, as well as displaying the graphical representation of all this gathered data. It is able to simulate the entire recording of a test made by a user on most of Tieto’s web applications, displaying the entire scenario to the one analyzing the test, including switching pages during the recording.

Eye data can be displayed as a static heat map, or as a heat map rendered in real time, while mouse data can be displayed either as a static heat map, or as a black mouse pointer animated in real time. Statistics can be shown for every page that has been recorded, where you can see fixation points directly on the web application and hover over these to get additional information.

4.1 User Tests

Below is the result of the different user tests that were held through the project.

4.1.1 Test 1

The first test was designed mainly to test the eye-tracking recorder’s functionality, and to test it out on Tieto’s own web applications. The data gathered is therefore not suited to be used as proof regarding the question at issue in this thesis. The tests themselves were however very successful, and the feedback we got from the participants was good.

The stability of the eye tracking recorder was great, and in the end, the result of this test was encouraging.

4.1.2 Test 2

Moving on to the second test, the primary goal was to test the eye-tracking recorder in a more complex environment, trying to switch between pages and use search forms. It was also designed to figure out where the participants would look, when asked to find the help for the page. This
was interesting, since the page consisted of two possible locations to find this information, these being the question mark above the calendar and the help button - also marked with a question mark - in the left menu.

While conducting these test, a lot of problems occurred. First with the eye-tracking equipment, which were calibrated wrong for the first participant. This issue was fixed by making new calibrations for each participant after that, however the recordings of the first user’s test had a offset on the y-axis by approximately 200 pixels downwards, rendering the test hard to analyze.

The second problem was that when the third user was doing his test, the web application that the test was run on started to behave in an undesirable manner. This caused the test to be interrupted, but at least the recorded eye data from that test was correct.

The test was pretty unsuccessful, but the resulting data showed clearly that the question mark on top of the calendar was never even looked at, so the test answered at least one of the questions.

4.1.3 Test 3

The third test focused on the difference in the users' perceived behavior and the recorded data. Each participant was asked to perform a set of objectives, and then answer a few questions about how they perceived the test. Four users of different gender, ethnicity and nationality participated in the test, three wearing glasses and one not wearing glasses. All were highly educated individuals, used to handling computers on a daily basis. The test took place on the page seen in figure 4.1.

![Figure 4.1 Test environment](image)

**Figure 4.1 Test environment**

**User 1** - The first user stated that his first fixation point was in the middle of the screen. In comparison to the recorded data, this was correct, with a difference of only a few centimeters. The participant also stated that the area that his eyes fixated most of the time was the menu bars in the upper left corner. The eye-tracking recorder clearly shows two main fixation areas, these being the areas a few centimeters below two upper corners. Regarding the time spent on
the page, the participant felt like the test took about 45-60 seconds, while the actual test time was 35 seconds.

This user claimed to have done user tests before, using the think-aloud method and paper tests. He said that he thought that using eye-tracker was a lot easier, and that it was a more fun way of testing.

See figure 4.2 for a complete heat map of the test.

![Figure 4.2 Heat map, user 1](image)

**User 2** - In the case of the second user, she also stated that her first point of fixation was in the middle of the screen, and that the area she fixated on the most was the upper left corner. The test data shows that the participant initially moved her eyes from the left upper corner to the middle of the screen, where she stopped for a few seconds. It is very clear that she spent most of the time looking at the menus in the left upper corner. She anticipated that the test took about 10 seconds to complete, while it actually took 29 seconds.

The second user had also participated in user tests before, and those had been think-aloud tests. She claimed that eye-tracking was a good way of testing, as she did not have to focus on telling the supervisor what she was doing, and could focus on the objectives instead.

See figure 4.3 for a complete heat map of the test.
User 3 - The third user thought that his first point of fixation was in the middle of the screen, just like the two before him, but he stated that the first thing he did was to look across all the text boxes below the top menu. He also anticipated that the area that he spent most time fixating was the upper left corner. The data shows that his eyes were fixating on the upper left corner most of the test. It also shows that he was correct regarding how he had moved his eyes in the beginning of the test, although his eyes initially fixated in the left top corner and not in the middle. As to the time spent on the page, the user perceived that the test took 35 seconds, and the recorded time was 48 seconds.

The participant had done UX testing before, and his thoughts were that eye-tracking was a good complement to other testing methods, as it gives precise data about what the user is actually looking at.

See figure 4.4 for a complete heat map of the test.
User 4 - The last participant had a different experience than the others. He though that both his first fixated and most fixated point was at the box with the heading “Placement”, which is located slightly below the students dropdown menu. The recorded data from the eye-tracker shows that both his guesses were right. The time he anticipated that he spent on the test was 20 seconds, but the actual time was about 60 seconds.

This participant had never done user testing before, so no comparisons between other methods could be made.

See figure 4.5 for a complete heat map of the test.
4.2 Result Summary
The Chrome extension is working pretty well at this state, but the way it is made might not be the optimal way. By trying to make the extensions player simulate the events that occurred using the test, it is required that nothing inside the application has changed position or been added/removed since the recording was made. Being designed the way it is at the moment, analyzing the tests right after they are recorded is essential. Still, problems can occur with mouse clicks not acting exactly as they are intended to. The product works well on non-complex tests on single web pages, but can also manage multiple page recordings. It meets the standards of the original specification from Tieto.

The results of the tests made shows that the eye-tracking recorder correctly displays the eye movement from a test, and that this can be used to gather information about how the users have been using the web application. In combination with mouse movement, mouse clicks and keyboard events, a fairly complete picture of what has occurred during a test can be displayed.
5 Discussion

The debate about whether eye-tracking tests, like those that has been made in this project, are relevant or just an expensive luxury is ever ongoing. The participants of the user tests made during this project are quick to see the pros for eye-tracking tests, but that may be a result of the fascination of experiencing new technique for the first time.

Some believe that the use of eye-tracking is unnecessary, because the result of tests with or without it often shows the same things. Even though the tests in this project show that this is true, the test scenarios themselves must be taken into consideration here. All our tests were conducted on individuals with a high education and that uses computers every day at work. They were also somewhat familiar with the web applications they tested, and they are in the ages of 25 and higher. The differences between the perceived experience and the gathered eye-tracking data would probably differ a lot more if the tests was done on small children or people with disabilities. Children are not able to explain themselves, especially not in a technical way, as well as adults are. Taking such things into consideration, the use of an eye-tracking recorder – especially one with the ability to record sound and mouse/keyboard events – would be a much more general tool for making UX testing.

A great thing with the use of eye-tracking is that you take away pressure from the test user, as they don’t have to focus on trying to describe what they are doing. To get as exact results as possible, it is a good thing that the person testing your application can focus on the task of exploring the new environment, instead of having to think about ways to communicate her thoughts to the test supervisor. It is also in the human nature to try to not make mistakes, and since we are herd animals, we will try to make people ignorant of those mistakes that we do. Some people will feel uncomfortable when they don’t understand what they are supposed to do when testing an application, making it possible that they leave out critical information. If you are able to conduct tests on your own, you will be less afraid to make mistakes, making the eye-tracking recorded a less threatening test supervisor. Many of the participants in the user tests during this project had the habit of trying to explain that they a novices at using the application that is to be tested, which indicates that this kind of insecurity exists.

When looking at the tests conducted during this project, it is pretty clear that the scenarios themselves will affect the outcome of the test data. This is also one of the things that objectors to eye-tracking tests have pointed out. It is also easy for the supervisor of a test to affect the test subject during the test by formulating the objectives in different ways. But this should be the case in any kind of testing, and not only applicable on eye-tracking tests. If you are being told to find all the green squares on a page filled with green squares and blue circles, then of course you will focus mostly on them. Does this mean that the test result is irrelevant? No, an eye-tracking test on such a scenario would be of great use, as you can look at the watched areas that do not contain green squares and try to figure out why the user was looking there.

Finally, when making user tests and analyzing them for use in making computer software better, the thing that is of most relevance is how accurate the data is. A human describing what she perceives to another human, who then needs to interpret this information depending on her own earlier experiences, can never be as accurate as a machine recording the same kind of data.
6 Conclusion

Taking the test results into consideration, it is clear that there are slight differences in what the users think they have done, and what the eye-tracker says that they have done. However, these differences are quite small, and this is most likely a result of these tests being rather short. In longer and more complex tests, the participants would have a lot more to focus on, making it harder for them to remain focused on both their objectives and giving an accurate description of what they are doing. The reason for this is most likely because the applications lack of ability to do this until the very last week of the project. By this time, we had no more time for user testing.

The general response from the user tests during this project, has been that the eye-tracking recorder makes it easier to focus on the objective itself, and that the eye-tracking gives more accurate data about the usage of the web applications. The most interesting subject for future work would be how to improve the presentation of recorded data, to make it more accessible for analyzing.

In answering the question whether the gathered data from the eye-tracker can be used to enhance and improve the development process of future software, this project has been both successful and unsuccessful. When making think-aloud tests, there is always the possibility that the user testing the product is lying/not aware of what she is doing. Even if the tests in this project show that the users are pretty much aware of where their eye are fixated, it would be impossible to tell if what they perceived was true, unless eye-tracking had been used to confirm this. As long as it is correctly calibrated, and the technique is working, the eye-tracking recorder tells the exact truth.

It is hard to say, with the results of this project’s tests, if this can enhance or improve future development processes. However, in combination with think-aloud tests (which can be recorded with the eye-tracking recorded) and all the other information that the program is able to gather, the test results will be a lot more accurate and can therefore be of better use to the developers.
7 References

1. Agile Software Development: A gentle introduction

2. Introduction to Agile Usability: User Experience (UX) Activities on Agile Development Projects

3. Tieto webpage


5. Core Scrum, Scrum Alliance (2014)


7. Eye tracking and usability research: an introduction to the special issue

8. Tobii, What is eye-tracking

9. Tobii, Tobii EyeX Engine


11. Heatmap.js, Patrick Wied


8 Appendix

8.1 Appendix 1 - WebSocket Communication Protocol

MessageType {int}

1. StartRecordingRequest
2. PauseRecordingRequest
3. ResumeRecordingRequest
4. StopRecordingRequest
5. RecordedDataRequest
6. RecordedDataResponse
7. StartRecordingResponse
8. PauseRecordingResponse
9. ResumeRecordingResponse
10. StopRecordingResponse
11. DisconnectRequest
12. DisconnectResponse
13. UpdateScrollHeight
14. UpdateUserInfo
15. ApplicationRequest
16. ApplicationResponse
17. GetAllApplicationsRequest
18. GetAllApplicationsResponse
19. GetSpecificDataRequest
20. GetSpecificDataResponse
21. SubTestRequest
22. SubTextResponse
23. RecordedMouseDataRequest
24. RecordedMouseDataResponse
25. UpdateDocumentSize
26. EyeTrackerActive
27. RecordedKeyDataRequest
28. RecordedKeyDataResponse
29. ...
30. MicrophoneStatus
31. StartRendering
32. PauseRendering
33. ResumeRendering
34. StopRendering
99. ErrorMessage

MessageContent {String}

- String containing the data to be sent