Development of a Web Based Departure Control System

Axel Nilsson
Filip Skogstjärna

Supervisor: Robert Hansson
Examiner: Soheil Samii
Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Axel Nilsson, Filip Skogstjärna
Abstract

A departure control system (DCS) is an administrative tool used by airports to handle boarding and baggage management. However the DCSs used by smaller Swedish airports are outdated and tied to desktop applications which are a hassle to update. The solution to this problem is to develop a new web based DCS. A DCS needs hardware to fully function (scanners, printers), this poses the problem of communicating between a web browser and hardware. To solve this problem a local server was developed to handle the hardware and communication with the browser which was then integrated into a web application. When developing the DCS, readability and further development was kept in mind making code quality and code commenting key factors. To evaluate whether these factor positively affected readability, an interview was conducted. The foundation of a web based DCS was developed and according to the interview results the conclusions is that code commenting, proper naming and modularity increases source code comprehension.
1. Introduction
   1.1 Delimitations
   1.2 Research questions
   1.3 Background

2. Theory
   2.1 Hardware
      2.1.1 Scanner
      2.1.2 Printers
   2.2 Serial solutions
      2.2.1 chrome.serial
      2.2.2 Node serialport
      2.2.3 Chrome Native Messaging
   2.3 External libraries
      2.3.1 Angular.js
      2.3.2 CakePhP
   2.4 Code comprehension
      2.4.1 Code commenting
      2.4.2 Coupling and Cohesion
      2.4.3 Code quality

3. Method
   3.1 Implementation of prototypes
      3.1.1 Selection of serial solution
      3.1.2 Server
      3.1.3 Front end interface
   3.2 Integration of system
      3.2.1 Analysis of system
      3.2.2 Adapting prototype
      3.2.3 Adding a GUI
   3.3 Evaluation
      3.3.1 Code Comprehension

4. Result
   4.1 Prototype
      4.1.1 Back end
      4.1.2 Front end
   4.2 Basic client
      4.2.1 AngularJS
      4.2.2 CakePhP
      4.2.3 View design (GUI)
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3 Evaluation</td>
<td>16</td>
</tr>
<tr>
<td>4.3.1 Interview</td>
<td>17</td>
</tr>
<tr>
<td>5. Discussion</td>
<td>18</td>
</tr>
<tr>
<td>5.1 Results</td>
<td>18</td>
</tr>
<tr>
<td>5.2 Method</td>
<td>19</td>
</tr>
<tr>
<td>6. Conclusion</td>
<td>20</td>
</tr>
<tr>
<td>References</td>
<td>21</td>
</tr>
<tr>
<td>Appendix I – Interview questions</td>
<td>22</td>
</tr>
<tr>
<td>Code comments</td>
<td>22</td>
</tr>
<tr>
<td>Code quality</td>
<td>22</td>
</tr>
<tr>
<td>Open questions</td>
<td>22</td>
</tr>
<tr>
<td>Appendix II – AngularJS example</td>
<td>23</td>
</tr>
<tr>
<td>Appendix III – CakePhP Controller example</td>
<td>24</td>
</tr>
</tbody>
</table>
1. Introduction

In today’s modern airports there is a need for administration and advanced computer systems to make sure boarding and baggage management function well. Departure Control Systems (DCS) were developed to fulfill this purpose (Hansson, 2017). There are several international and well developed DCSs\(^1\) which are scalable and adapted for large airports, however these can be quite expensive and therefore not very affordable for smaller airports. Because of the price, many small Swedish airports are left with old and outdated DCSs.

The outdated DCSs used in the smaller Swedish airports are tied to desktop applications since, during their time of development, web applications lacked the functional utility required by a DCS. Desktop applications have a large drawback: each software update needs to be sent and individually installed at every location. A common solution to this issue is to transfer the desktop application to a web based application, which has centralized updating to one location.

A significant issue for web based applications is their communication with hardware since operating systems (OS) consider it a security risk to openly communicate with internet webpages. In a DCS several different scanners and printers are needed to procure baggage tags and boarding passes. This makes hardware communication a key factor.

Most software developers who work on a project were not present when it was started from scratch; instead they joined at a later stage. This means that they would have to read and understand the already existing source code. Software engineers generally know exactly what their written code does. However they do not always convey their understanding in the way they code. This can lead to a significant learning curve when another developer tries to understand the code. Code commenting and modular coding are a couple common techniques, relevant to most situations, used to counteract this.

1.1 Delimitations

Development in this project will have a main focus on establishing fully functional communication to a scanner and two different kinds of printers. There are a couple of additional minor goals, these consist of communication with the airport database and a simple graphical user interface (GUI) developed in AngularJS. However, these minor goals are secondary in priority.

The direct communication between a web browser and hardware is restricted by most OSs since it presents a security risk. By solving this communication there is a new vulnerability for web based attacks. This vulnerability is not considered in the project since the potential damage of an attack is very small and since the server runs on a closed network.

---


1.2 Research questions

- How can a basic web based DCS be implemented concerning readability and simplicity in further development?

1.3 Background

InspireIT is the company that has requested a web solution to their DCS. It is a small company based in Norrköping (Sweden) that specializes in computer systems for airports. Their expertise revolves around high level system development and they felt that their time was ill spent trying to figure out hardware communication.

They also doubles as a consulting firm, providing extensive support and helping out with complex solutions.
2. Theory

There are several topics that need to be covered in order to fully answer the research question. They have been divided into four categories: Hardware, Serial solutions, External libraries, and Code comprehension. The hardware section will describe the hardware that will be used. The Serial solutions section will describe different solutions to the hardware communications problem. The External libraries section will describe some of the Frameworks that is going to be used later on. Finally the Code comprehension section will describe the theory around evaluating the quality and readability of the code.

2.1 Hardware

To properly run airport operations several devices are necessary. Printers provide passengers with boarding passes and baggage tags so that they can identify themselves and their luggage. Barcode scanners allow the airport staff to scan these passes and tags to confirm and help passengers. These are only a couple of the necessary hardware tools needed to run an airport terminal, however this project will only be covering these two.

2.1.1 Scanner

The scanner used in this project is a Datalogic Gryphon D230\(^3\). The scanner has a physical button which triggers the scan, and when it successfully reads a barcode of some sorts it sends the data serially back to the computer through a COM port. If the computer lacks COM ports, like most modern computers, a USB port can be emulated so that it works like a COM port. The scanner sends its data serially and notifies the end of the message with the character CR (carriage return).

2.1.2 Printers

There were two different printers used in this project: Intermec EasyCoder PF2i and Intermec EasyCoder PF4i\(^4\). The first one for printing baggage tags and the second for printing boarding passes. Both of these require specific drivers to run serial communication over USB ports. Fortunately modern operating systems (Windows 10, Mac OS etc.) contain these drivers natively.

The printers are configured through sending a so called *pectab*, which defines a grid on the tag. Each field in the grid has an identifier and through sending a string that first states the identifier of the field and then the value that is to be printed, the different fields are set. As shown in figure 1, printing is done through sending the name of the *pectab* followed by the fields and their corresponding values. If the print was successful the printer will respond with the string “PROK” and once it is received the final step of the sequence can be done, which is to send the ascii


character 6 (<ACK> or ‘Acknowledged’). The initial string is only recognized by the printer if it starts with the second ascii character (<STX> or ‘start of text’) and ends with the third ascii character (<ETX> or ‘end of text’). If the printer does not return the “PROK” message, something went wrong with the print. Either the initial string was incorrect or something else happened (for example, if the printer is out of paper).

![Flowchart of the printing sequence](image)

Figure 1. Flowchart of the printing sequence
2.2 Serial solutions

A large drawback of web based applications is that verification of content is not done automatically. Therefore the content of a web page cannot always be trusted since it could contain potentially harmful code. The solution OS developers came up with was to limit the communication between the hardware installed on the computer and the browser. This becomes a slight inconvenience when, as a web developer, developing a web based application that needs to communicate with hardware.

A few different solutions were considered: building a Chrome app that utilizes chrome.serial API, running a local Javascript server through NodeJS and running a desktop application that utilizes the Chrome Native Messaging API to communicate with the web application.

2.2.1 chrome.serial

Chrome has had support for so called Chrome applications, which work as desktop programs but they’re downloaded and launched from Chrome. Building a Chrome applications is much like building a web application (with small variations). Chrome have also added support for serial communication through their API chrome.serial\(^5\).

There are several advantages with using a Chrome application to communicate serially to hardware, the biggest being that it is completely web based. You only have to download the application into your Chrome browser for it to work. However there are also several disadvantages, the biggest being that Google is discontinuing their support for Chrome applications in every operating system except their own Chrome OS.

2.2.2 Node serialport

NodeJS\(^6\) is a framework for running Javascript files and specializes in server functionality. Having a local server presents the opportunity to request information about the hardware and to communicate with it. NodeJS has a module system which is convenient and contains its own package manager (npm). A module is installed through npm and then requested in the Javascript file. Using a module called ws (websocket) the server can connect to a web application (using the native support for websockets in HTML5).

Serialport is a NodeJS module that can be accessed through npm. It adds functionality to open a serial connection to hardware. When opening a connection, serialport requires a path to the port in which the hardware is connected to. Serialport has events handlers which are called when specific events happen. The onOpen event is triggered when a serial port is opened and respectively the onClose event is triggered when it is closed. When data is received through the serial port, the onData event is triggered and if the data happens to contain an error the onError event is triggered instead.


Each of these events needs to be defined specifically for each serial port.

A property that serialport has which the other solutions lack is the predefined parser that parses the data received on the port, for example this can be used to convert the raw data received to a string which it then forwards to the onData event when a certain delimiter is reached (in our case: carriage return).

2.2.3 Chrome Native Messaging

Chrome Native Messaging\(^7\) (CNM) is an extension to Google Chrome which supplies an API for cross-application communication. CNM identifies different web applications by a Javascript file called “manifest.js” in which properties for each application is defined. Defining these manifest files for both client and server would enable the API to handle serial communication over standard output and input streams.

2.3 External libraries

To make the prototype easy to add to the company’s current product, it has to follow the same standards and methodologies as the rest of their product. The most important external libraries that will be used to achieve this is Angular.js and CakePhp.

2.3.1 Angular.js

AngularJS\(^8\) is a framework that expands HTML functionality and allows for easier binding between models and views, this allows for more seamless synchronising between variables in Javascript and elements in the HTML (Branas, 2014).

Angular JS introduces some new concepts: view, model, controllers, directives and services (amongst others).

**View** is what the user sees and interacts with, typically a HTML document.

**Model** is what runs in the background, interacting with the view (and by extension the user).

In short, the user and the model interact with each other through the view.

**Controllers** are the way Angular controls the data in the application. They are defined in the root element from which the application is running from.

**Directives** are the way Angular assigns its attributes to the HTML elements. Examples of directives are ng-model (binds data between variable defined in the controller and input-field in HTML), ng-click (an event that triggers when the mouse has clicked an element)

**Services** are a way to increase modularity in the code by moving functionality from the controller to a separate service-file that is injected into the controller.

---

\(^7\) https://developer.chrome.com/extensions/nativeMessaging, retrieved 2017-03-23

\(^8\) https://docs.angularjs.org/api, retrieved 2017-04-12
In Appendix II there is a small AngularJS application that has two input fields which are bound to respective variables, the variables are then evaluated in an expression. The result of this is that each time a change is made to the input field the evaluated variable will also be updated.

2.3.2 CakePhP

CakePhP\(^9\) is a php based framework that provides an easy to use database layer and is based on a scaffolding system. The scaffolding base makes CakePhP compatible with AngularJS. The usage of CakePhP allows the web application to communicate with the database through HTTP requests (get, put, post etc). It also provides an easy to use syntax for querying the database.

Since the framework is based on a scaffolding system it is inherently modular, making it easy to add new modules and functionalities to the system. In this project all routing and the infrastructural work in CakePhP was already established by the company. Only new controllers for specific queries had to be defined. A small example of how such a controller could look can be seen in Appendix III. To reach this controller in this particular example a POST request with a parameter named firstname would have to be sent to the correct URL (as specified in the routing file).

2.4 Code comprehension

There are several ways to make modules more readable, and in extension easier to develop. The ones explored in this thesis are code commenting, modularity, and hierarchical structuring. There are several more but according to Pressman (2005), Steidl, Hummel, Juergens (2013), and Sedano (2016) these are the most essential attributes.

---

\(^9\) https://api.cakephp.org/3.0/, retrieved 2017-04-12
Figure 2 showcases a modular function with a commenting style that explains what each section of the source code is supposed to do. The source code in this specific example is not very complex nor difficult to understand. In a scenario where the source code is complex, or difficult to read, code commenting like this can save a lot of time. The name of the function and the variables it uses are chosen to reflect their purpose, making comprehension of the code as easy as possible.

2.4.1 Code commenting

Commenting code has been the standard way of giving developers an overview of a piece of code. Determining the quality of a code comment has always been a hassle since each individual software developer uses different language and logic when writing their comments. A recent study defined a few different metrics to use when measuring the quality of code comments: coherence, usefulness, completeness and consistency (Steidl, 2013).

**Coherence** property describes how well the comment and the code relate to each other. In other words the actual functionality of a method or member should be described in the comment, instead of just the method name or a short irrelevant description. **Usefulness** defines whether a comment contributes to the general understanding of a system. If a reader considers a comment to be helpful when trying to understand the code, the comment is defined as useful. **Completeness** covers how well the entire system is commented on a global scale. This means that every single method and member should have a comment describing it in detail. Furthermore making sure each class document has a header for each individual method will help users to find them and overall make the system more complete. Finally **consistency** described all features that should be consistent throughout all comments in the entire system such as language and tense.

2.4.2 Coupling and Cohesion

Coupling and Cohesion are two metrics used to determine code quality. The two of them measure how independent each module is (coupling) and how relevant the content of a module is to its purpose (cohesion). By using these two measurement it is quite easy to determine whether a module is well designed. Keeping these in mind when writing code will hopefully result in a low coupled and highly cohesive module, meaning that there are few code dependencies and highly relevant content in the module. (Pressman, 2005)

2.4.3 Code quality

Writing code that is easy to read is an acquired ability which can vary a lot from one software developer to another. A recent study has concluded that improving a few different properties in a code snippet can have large impact on how readable it is (Sedano, 2016). Writing explicit variable names will make it easier to understand the variables purpose and the same applies when defining new methods or functions (Sedano, 2016). Making conditions and loops as simple as possible also has a positive effect on the readability of the code (Sedano, 2016).
3. Method

This chapter covers the basic steps of how the project was approached and executed. In the early stages of the project, a few properties had to be decided. Since the project was done by more than one person, a system needed to be chosen in order to divide work and encourage cooperation within the team.

Once the prerequisites for the project had been decided, a solution to the serial communications problem had to be chosen, implemented, and at last integrated into the company product. Finally an evaluation of functionality and code comprehension was done.

When developing the software solution, the software prototype model was used (Prototyping). The benefits of using this method include early feedback regarding functionality and deadline estimation. In this specific project these two aspects were important to establish a solution method early on.

3.1 Implementation of prototypes

The prototype was divided into a back end server and a front end interface (the reason for this division is discussed briefly in chapter 3.1.1 “Selection of serial solution”). The purpose of this front end interface was to make the prototype easy to use. The back end handles serial communication with the printers and the scanner. The frontend interface allows a client to send requests to the server and to receive status reports from the server.

3.1.1 Selection of serial solution

The three different options that were considered are briefly described in chapter 2.2 Serial solutions. Most of the solutions works in the same way: using a local server and some sort of technique to communicate locally to that server. NodeJS was chosen for this project since the company product was already utilizing NodeJS and since the communication between a local NodeJS server and a web client is simple and functionally sound. Performance was not an issue when selecting a serial solution since the solution does not have to be scalable and all the solutions are only limited by the speed of the hardware. The criteria when selecting a solution was therefore only simplicity in implementation and that it actually worked as intended.

3.1.2 Server

The NodeJS server listens for connecting printer connections and scanner connections from the client. It will then instantiate one scanner or two printer depending on which connection was opened (representing the ones described in the theory section). However since there are no scenarios where two clients are connected, the server only has support for one connection. Client print requests was processed and forwarded to the correct printer. Scanned data and printer responses were also forwarded to the client over a websocket.
**Scanner**
To communicate with the barcode scanner a NodeJS module was developed to receive and parse serial data. This module needed to be able to find the serial port in which the scanner is plugged in. When reading data, the module listened on the scanners COM port and when data is sent it will parse the data serially until a carriage return character is received and then proceed to forward the information to the server.

**Printer**
To communicate with the printers a NodeJS module was developed to correctly format the string that is sent to the printers, as well as handle the communication sequence. The module needed to be able to read a configuration text file to find the correct port to connect to. It then had to establish a connection to said port and wait for print-request from the server. A print request starts the printing sequence which includes formatting the data-object and the request to the correct format, sending it to the printer, waiting for the printer to respond with “PROK” and then replying with <ACK> (ascii code 6).

**3.1.3 Front end interface**
To make the communication to the server easy to use, a handler was created to handle the communication. The handler consists of two parts, a scanner connection module and a printer connection module. The scanner connection module opens a scanner connection to the server and listens for data objects. The printer connection module opens a printer connection and it gives the web developer functions for printing either boarding passes or baggage tags, as well as receiving statuses from the printer.

The server is asynchronous since it is purely event driven. Similarly the connection modules in the front end interface are also completely asynchronous as they are also event driven. To send data from the websocket, a callback function was used and therefore sending several requests simultaneously was not an issue.
3.2 Integration of system

When the prototype was completed it had to be integrated in the already existing company system. There were a couple of challenges when doing this since the prototype had to be adapted to the frameworks and inner workings of the system. To tackle these issues an analysis of the system was done to determine how the infrastructure of the system works and which frameworks it uses. Then the prototype was adapted to these frameworks and distributed in relevant parts of the system. To make sure the newly added functionality works in conjunction with the web application a simple graphical user interface (GUI) was added. An overview of how the system architecture looks can be seen in figure 2.

![Figure 3. Overview of system architecture](image)

3.2.1 Analysis of system

An understanding of the current system and how it works was established by scouring through the source code and asking relevant questions to the author of the source code. After outlining the frameworks used by the company, a broader understanding was achieved by viewing and testing code snippets from similar modules.

3.2.2 Adapting prototype

The prototype was implemented as an independent server which runs in the background and handles all hardware communication. This means that only the client side interface has to be adapted to the company product. The interface built to communicate to the server was then converted to an AngularJS service which can be invoked or instantiated from views and models within the angular application.
In communication between controllers and services (in AngularJS) callback functions are not supported. To counteract this issue all callback functions handling this communication was converted to AngularJS adapted promises\(^{10}\).

### 3.2.3 Adding a GUI

When both the back end server and the front end interface was fully integrated and working as intended, a GUI was created to connect the scanner functionality with the printer functionality through the web application. The implementation consists of a form where the user is able to see information that is going to be printed, but not be able to manually edit it. This form will then be filled out by the information retrieved from the database (and identified by the scanner). The prototype created in this project is only the foundation of a complete DCS, which means that a lot of functionality expected in the final product was not implemented in the project.

### 3.3 Evaluation

To filter out all unnecessary source code, make sure everything works as intended, and to make sure that the source code is readable there was a code evaluation step at the end of the project. At the start of the project InspireIT made a list of functionalities that the prototype should have once the project is done. Each of these functionalities were thoroughly tested and reviewed by the company.

#### 3.3.1 Code Comprehension

According to McIntosh et al. (2014) the easiest and most relevant approach to evaluate code comprehension is to ask a fellow software engineer to read, and try to understand, the source code. Then to perform an interview asking questions about how the code was perceived, and asking questions about functionality (to determine the readability). Noteworthy is that reading and understanding code is something quite individual and therefore more than one software engineer was interviewed to get a more general view of the code quality.

To make sure that the source code was compliant with the company standards it was of utmost importance to get other company developers input regarding source code quality and code comment quality. The last evaluating step before handing the code over to InspireIT was therefore to conduct interviews with several employees of the company. In these interviews the questions covered properties regarding code quality, code comment quality and general code perception. The questions used can be found in Appendix I.

In terms of code commenting there are four different metrics described in 2.4.1 Code Commenting (coherence, usefulness, completeness and consistency). All four of these attributes can accurately describe the quality of code comments but since this project is limited to only one module in a much larger system the attribute completeness was left out.

\(^{10}\) [https://docs.angularjs.org/api/ng/service/$q](https://docs.angularjs.org/api/ng/service/$q), retrieved 2017-04-19
4. Result

The project was divided into three major parts: building a prototype, building a basic client, and then evaluating both. For each part of the project the initial plan will be described along with all major changes that were made during the process and finally the end results of each part will be summarized.

4.1 Prototype

The solution that was chosen for implementing the hardware communication prototype was a back end server (ran by node JS) and a front end interface. Together they solved the problem of communicating with hardware from a web application.

![Figure 4. Prototype system design](image)

The prototype was designed with regards to simplicity and modularity where each part of the program had a very defined purpose and low dependence on other modules. A large part of the functionality was placed on the back end of the prototype to encapsulate the more complex functionalities. An overall view how the prototype was designed and modulated can be seen in figure 3.

4.1.1 Back end

The back end server was split into three modules: the main ‘server.js’ module, the printer communications ‘printer.js’ module, and the scanner communications ‘scanner.js’ module. The server module opens and listens to websockets and connections, when receiving a connection it will (depending on which socket) open a connection via a COM port to the specified hardware.

The printer.js and scanner.js modules contain a Javascript function that is instantiated in the server when a new connection is to be opened. When either a printer or a scanner method is instantiated a connection is automatically opened. A callback function is passed to the constructor which the method then evokes when wanting to send data back to the server.
4.1.2 Front end

The front end interface worked much like the back end with separate modules for the printer and scanner. To open a connection (of either sort) to the server one of these modules simply have to be instantiated, for example if a scanner connection is to be opened a new scanner connection is instantiated with a callback to be notified when data is received.

When adapting the front end interface to Angular and InspireITs own system, the two modules were converted to resources. They both had an initialization function to start a connection and as well as a close function to end it. The big issue with the conversion to an Angular resource was that callbacks are not permitted as parameters to the resource functions. A solution to this was to use the customized promise functionality built in to Angular. A promise was created and returned in the end of the asynchronous init function and resolved when the process of closing the connection was done, and then by using the notify functionality information was able to be sent back to the controller from the resource.

4.2 Basic client

When integrating the prototype into InspireITs product the front end of the application had to be adapted to their system. This means that the same source code had to be converted to fit with their frameworks and models. The front end side of the web application is extended with AngularJS, the database communication is handled by CakePhP and the View presented to the user is similarly extended with AngularJS directives.

The main goal of the project was to solve the hardware communications issue with support for one scanner and two printers. The back end of our solution has this support, however the front end only supports the scanner and one printer (baggage tag). Since only one of the printers actually needed the information from the scanner and the other printer would need a manual search in the database which was not a part of this project.

4.2.1 AngularJS

The AngularJS framework lets a developer divide an application into a few different elements. There are many types of AngularJS elements but the main three that were encountered in this project was services, controllers and resources.

Services contain an isolated module which can be used from different parts of the code base. Two services were implemented when integrating the prototype into the system: the scanner service and the printer service. These two handled the websocket communication to the local NodeJS server.

The AngularJS controllers handle the relation between the a HTML object (for example a div) and the AngularJS model. The main purpose of the controllers in this project was to connect the
prototype services to buttons and to present the retrieved database data in separate data fields of the view.

There was only one resource that had to be defined when integrating the prototype. This resource mapped a specific function call on the front end to a specific CakePHP function on the back end. It also defined what type of HTTP request should be used and what fields should be present.

In figure 4 the conversion from the Javascript front end module to a AngularJS service is depicted (since the source code is proprietary to InspireIT the actual source code in each function is hidden). The “ScannerConnection” module was already defined in the earlier stage of the project and could simply be pasted into an AngularJS service.

```javascript
'use strict';

angular.module('services.scannerConnection', []);

angular.module('services.scannerConnection').factory('ScannerConnectionService', function($q){
    // Variable which holds the scanner connection
    var myScanner;

    // Scanner handler module, establishes a scanner connection when instantiated
    function ScannerConnection(OnDataCB){
    }

    // Instantiates a scanner connection and returns a promise which is notified when data is scanned
    function init(){
    }

    // Closes the scanner connection
    function close_scanner_socket(){
        myScanner.close();
    }

    // Public functions
    return {
        init: init,
        close: close_scanner_socket
    };
});
```

*Figure 5. AngularJS service wrapped around ScannerConnection module*

4.2.2 CakePHP

To be able to print a correct baggage tag, information about the passenger, the flight, the airline, and the company needed to be present. The solution to this was to define a new cake controller with a new specified class function that retrieves the data needed from all the relevant tables and joins them correctly before sending them back.

However since the company of the airport is always the same, an initial request is sent to collect the full name, and IATA, of the company when initializing the DCS module.
4.2.3 View design (GUI)

The overall design of the view was based on the design of the rest of the system, using the same structures and same type of lists with the same directives. Since the project was mainly focused around the functionality revolving around hardware communication very little effort was put into the design of the user interface. In figure 5 the complete design is presented. In order to print a tag using the print button, every field must contain relevant information. If a boarding pass is scanned, and a corresponding passenger is found in the database, all fields will be filled out automatically with values from the database.

Figure 6. View of GUI
4.3 Evaluation

The third and final step of the project was to evaluate code quality, code commenting and overall code comprehension of the source code written. An interview asking questions about these subjects was done with software developers at InspireIT.

4.3.1 Interview

The interview was split into three parts. First part was about code commenting, the second part was about general code quality and understandability, and the last was a bit more open questions. What follows is a summarization of the general contents of the interview.

**Code commenting** was split into three further parts: Coherence, Usefulness, and Consistency. The interviewees all got the impression that the code was coherent, useful, and consistent. Regarding the usefulness the interviewees stated that even though code usually is understandable without any comments it gives a better overview and greatly reduces the effort it takes to understand it, and in turn reduces the time it takes to understand the code.

Regarding **code quality** the interviewees didn’t have much to say, the names were relevant and the code was properly split into modules and functions which made understanding easier. The general impression was that relevant and clear names helped a lot along with consistent comments, which concurs with what Steidl, Hummel, Juergens (2013) concluded. Another thing that the interviewees stated was that using a code commenting standard (with documentation) in the whole system could further increase the readability.
5. Discussion

There were several elements to the project that could have been done differently. Most of them were related to how the project was organized and which methods were used and how these decisions affected the results.

5.1 Results

During development of the serial communication to the two different printers the prototype was hardcoded to the language these specific kind of printers use. So if a user were to attempt to use a different printer it would not work. To make it work the encoding of the message sent to the printer would have to be updated. Furthermore, printers need to be pre-configured with so called “pectabs” in order to work. This configuration is not something that our prototype supports.

The decision to use a local server as solution to the hardware communications problem might seem a bit counter intuitive, since the main argument to moving a web based platform is to not need anything installed locally. However in comparison to the original full desktop client the server is miniscule, the whole solution is done in just below 400 lines of code. It’s incredibly easy to set up: the node library has to be installed, two node packets need to be installed and the top of the Javascript file (containing all the code) need to be configured with the pathnames of the hardware. The point being the solution is small, very easy to set up, and it works as intended.

When developing the first client used to test the server, the decision to split the code into a client and a connection handler was made to make it easier to read the main Javascript file. However one great benefit (unbeknownst to us at the time) was that the modules very neatly translated to Angular services. The original source code did not have to be changed; the modules were simply wrapped in the headers defining a service. Only two things had to be added to make the services fully functional. Firstly the instantiating part was moved from the client to the service which was accomplished through an init function, and secondly callbacks were not supported in the communication between service and controller and were instead replaced with promises.

The project is bound by strict deadlines which means that there is not enough time to develop a fully functional DCS. This, in turn, means that the system that is produced during the project has to be developed with further development in mind in order to make ease the initial learning curve for future developers. Furthermore there is an academic merit in analysing code quality and comment quality since it a widely discussed subject with a large base of previous works to base the evaluation upon.

In the software industry there are a lot of different standards for how each company writes their code. Either using an already existing one or creating a new one could have further improved
how readable the code is. However, this is an aspect that was discovered during the evaluating interview making it a bit too late to apply to the large amount of comments throughout the source code.

The interviews that were done in order to evaluate and give InspireITs software developers a chance to give input about source code format and commenting was quite limited since it is a small company and there was only a few software developers available to answer the interview questions. This however does not taint the results of the evaluation since the developers interviewed gave in-depth answers to the questions.

5.2 Method

The front end of the prototype was written in Javascript since it inherently works with HTML. The back end however is not bound to any specific programming language. In this project the back end was also written in Javascript due to NodeJS being a Javascript based solution. Since the framework CakePhP was used in the front end to handle database communication, PhP was also used.

Since one of the main goals of the project was to create a prototype capable of communication between a web browser and hardware, Prototyping seemed like the obvious choice when choosing a software development method. Using Prototyping, a developer can get an easy overall grasp of the project and its different aspects during the early stages of a project. This allows the developer to distinguish what requirements the system needs to function, and in the case of this project determine which hardware communication route is most viable.

The first prototype that was built in this project used chrome.serial in an attempt to communicate scanned data to Google Chrome. However there was a quite large bump on the road since it was discovered that Google plans to remove support for Chrome applications directly in the web browser. This would make the prototype obsolete. This discovery forced an early turn in the project and the choice of a new hardware communications solution. Had the choice of software development method not been Prototyping, this issue could have been avoided at an earlier stage since some more initial research would have uncovered this.

The evaluation of the source code and code comments that was written throughout the implementation of the prototype was based on two quite recent and completely different studies. One was a purely empirical study which determined what attributes should be considered when writing readable source code (Sedano, 2016). The other study tries to define metrics which measure the quality of comments (Steidl, Hummel, Juergens, 2013). These two give a good introduction to what it means to write readable code and high quality code comments. However using more articles could have given the evaluation a wider base. As mentioned the two articles used were quite recent ones. They are based on earlier research which are well cited and well used. As an additional tool for evaluating these articles could also have been used.
6. Conclusion

Airports have a constant administrative need due to strict timelines, tight schedules, boarding control and making sure flights are as secure as possible. The DCS was developed in an attempt to make all of this possible within a single application. However the DCSs that Swedish airports use today are often outdated and tied to a desktop application which makes the software a hassle to update due to having to update it at every location. Therefore there is an increasing need to redevelop the software into a web based application. This would centralize updating and make the application accessible from various devices. In this project this problem’s solution has been summarized in the following research question:

*How can a basic web based DCS be implemented concerning readability and simplicity in further development?*

A prototype was developed as the foundation of a web based DCS. The prototype was then integrated into a larger airport system, consisting of much more functionality than just a DCS. Throughout the software development the code was written with modularity and code comprehension in mind (Pressman, 2005; Sedano, 2016) and commented according to Steidl, Hummel, Juergens (2013). To evaluate whether the source code fulfilled the research question’s requirements an interview was conducted.

The interview was held with a small team of experienced software engineers where they read the source code and answered questions regarding modularity, code quality, code commenting and overall code comprehension. The questions varied from very specific to very general in order to get broader spectrum of responses.

The answers to the interview questions revealed that the interviewees shared similar thoughts and opinions regarding code comments and code comprehension as Sedano (2016) and Steidl, Hummel, Juergens (2013). This evaluation confirms that the written code is easy to understand and that it is well commented, making way for easier further development.
References


D. Steidl, B. Hummel, E. Juergens, Quality analysis of source code comments, In proceedings of 2013 IEEE 21st International Conference on Program Comprehension, ICPC, 2013, May 20-21, IEEE.


R. Hansson, Lead Developer at InspireIT, interview 2017-03-23.

Appendix I – Interview questions

- Here are two examples (figure 1.2) of commented and properly structured source code. What are your thoughts about them? Are they well formatted or lacking. If they are lacking, what would be a better alternative?

  Figure 1: Commented module*11
  Figure 2: Commented function*

Code comments

- Regarding Coherence. Does the comments correctly describe the content?

- Regarding Usefulness. Do you find the comments useful? Alternatively would a non-conversant developer find the comments useful?

- Regarding Consistency. Are the comments consistent? (level of detail, language)

Code quality

- Are variable and function names relevant and consistent with their inherent functionality?

- Do you think that functions seem true to their purpose?

Open questions

- Which attributes do you consider important when trying to understand source code?

- Further thoughts regarding the subject?

---

11 * contained source code proprietary to the company and cannot be published
Appendix II – AngularJS example

<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js"></script>
<body>

<div ng-app="myApp" ng-controller="myCtrl">

First Name: <input type="text" ng-model="firstName"><br>
Last Name: <input type="text" ng-model="lastName"><br>
<br>
Full Name: {{firstName + " " + lastName}}

</div>

<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
    $scope.firstName = "John";
    $scope.lastName = "Doe";
});
</script>

</body>
</html>
Appendix III – CakePhP Controller example

```php
namespace App\Controller;

class PassengersController extends AppController {

    public function getPassengerByFirstName()
    {
        $firstname = $this->request->data('firstname');

        $passenger = $this->Passengers->find('all', ['fields' => [Passenger.firstname]])
        -> where('Passenger.firstname' => $firstname);

        $this->set(['results' => $passenger,
                     '_serialize' => ['results']]);
    }
}
?>
```