

Report on Flight Service Station Information Systems

Prepared by: Andrew D. Wylie

Prepared for: COMP3202-A; Co-operative Work Term Report 3

Employer: NAV Canada

Manager: Mark Libant; Manager, FSS Information Systems

Supervisor: Andrew Gibb; Senior Systems Analyst

Group: Flight Information Systems Automation

Abstract

NAV Canada is the country's civil air navigation services company. Their headquarters are located in Ottawa, Ontario, with air navigation service facilities spread throughout the country. The company has different groups working on hardware and software solutions to provide services which relate to radar, weather information, and flight information and billing. The Flight Information Systems Automation (FISA) group is responsible for the production and deployment of software and hardware solutions provided for use in Canada's air navigation services network. During the cooperative work term I provided assistance to a sub-group of FISA to create a product for use by pilots throughout Canada. Working under Mark Libant, manager of Flight Service Station Information Systems, our team focused heavily on developing a new system called the Collaborative Flight Planning System (CFPS). Throughout the work term I was mainly involved with the CFPS project, and as such this report will cover much of the methodology and process that our team followed. The development environment will be explained, and certain parts of the project which were worked on will be outlined in more detail. In the conclusion, I summarize much of what I have learnt, along with explaining how this experience will influence my future in regard to both my studies and life.

Table of Contents

Introduction.....	3
Involvement	5
CFPS Overview	5
Procedures & Methodologies.....	6
CFPS Contribution.....	9
Conclusion	14
Glossary	15
Bibliography	16
References.....	17

Introduction

NAV Canada is the country's civil air navigation services provider. It is a private, not-for-profit, non-share capital corporation which provides systems that handle air traffic control, weather information, advisory services, electronic aids, and flight information, billing, and planning¹. NAV Canada was founded in 1996 to take over responsibility of air navigation services from Transport Canada, which also had the effect of privatizing Canada's air transportation system². As such, NAV Canada operates independently of any government funding, with revenue generated through flight billing and commercial projects. NAV Canada's air navigation service facilities comprise seven area control centres and 41 control towers, as well as 60 flight service stations and eight flight information centres³. Each type of facility serves a different purpose, with the area control centres and flight information centres acting as information distribution hubs for the region. Other NAV Canada facilities include the Technical Systems Centre (TSC) in Ottawa, and the NAV Canada Training and Conference Centre located in Cornwall.

Development, testing, and certification of new hardware and software components is done at the TSC, which staffs around 350 employees. The new components are tested for compatibility with numerous other systems, internal and external interfaces, and different operating systems and hardware. During testing, documentation is also written for technical and end user support for use in the field by Air Traffic Controllers, Flight Service Specialists, and Technologists. Introduction of new software systems is a must, for the Air Navigation Services Industry is constantly moving forward to provide both the end users and those employed in the industry with up to date systems for use with all aspects of the industry. The main two reasons for new software is that either the

application is reaching the end of its production cycle and will become unavailable within the near future, or that there are external factors pushing for an alternate product to be used. Usually the reason to switch to an alternate product is the result of needed improvements to compatibility or functionality. After the new configuration is researched, sourced, and constructed, it is rigorously tested at the TSC with its documentation being written as part of the development process.

The services provided by NAV Canada are split up into different systems and sub-systems such as the Canadian Lightning Detection/Distribution Network (CLDN), Internet Flight Planning System (IFS), Flight Service Station Information Management System (FIMS), Digital Aviation Weather Camera (DAWC) system, Flight Service Station Weather Graphics System (FWGS), etc. Each has a different purpose for monitoring weather and controlling air traffic. For example, CLDN is a software program designed to supply a live feed of the current lightning strikes and their locations from across Canada.

Throughout this report, a new product in development will be discussed; the Collaborative Flight Planning System (CFPS). CFPS is the next generation flight planning system, designed to replace the current Internet Flight Planning System, and to integrate with the FSS Information Management System. The CFPS development environment will first be explained, followed with a description of the deployment environment. Afterwards certain parts of the project in which I was involved will be outlined in more detail. Then, in the conclusion I summarize much of what was learnt during the term, along with explaining how this experience will influence my future in regard to both my studies and life.

Involvement

During the four month work term at NAV Canada I was fully involved with the CFPS project. This included developing, debugging, and testing code, as well as writing unit tests and integration tests. As a member of the development team I also had input relating to the features included in the project, and the order in which the features were added into the project.

CFPS Overview

As briefly mentioned above, the push for this new product is about added functionality, and the replacement of IFS, a decade-old system. Designed as a web application, CFPS will allow pilots to file flight plans for both commercial and leisure flights. The current systems which provide this functionality are not all that flexible in what they allow. For instance IFS allows bookings for Instrument Flight Rules (IFR) and Visual Flight Rules (VFR) flight plans, although the graphical interface is quite clunky. CFPS will provide the same functionality, along with added abilities providing flight templates to be utilised. Another motivation for the new system is that currently, many pilots call in to a Flight Information Centre to submit flight plans. It is hoped that the introduction of CFPS will release some pressure from the flight specialists who take these calls, so that they will have more time for other work.

CFPS will be hosted as a part of the Aviation Weather Web Site (AWWS), allowing users of the site to link their current AWWS account to a CFPS account so that more functionality can be gained. The users of this web platform will be allowed to log in and view the status of their currently filed flight plans. Details of each flight can be viewed as well as new flight plans submitted. Templates for flight plans are also available

to be used, and can be edited, removed, and even filed as flight plans. This feature allows the flight plan form of forty-odd fields to be saved for future use, thereby enhancing the user experience when the decision is made to file the flight plan.

Procedures & Methodologies

Regarding the development process there were several tools used, ranging from source control, bug tracking, and defect tracking to more process oriented aspects such as the agile development techniques which were used. Also related on the technical side of things is the design of the CFPS deployment environment. To begin with I will discuss the development environment, followed by an overview of the deployment environment, and conclude with an explanation of the agile techniques used.

When developing software there are often certain models, or patterns, which are followed from time to time. One of these is the Model-View-Controller (MVC) architecture. Its purpose is to isolate certain components of a software program from each other, allowing independent development, maintenance, and testing. As the name describes, the three components are the model, view, and controller, respectively. One of the reasons behind using such a model is to allow any of the components to be individually changed. For instance, statistical data could be interpreted differently visually by using charts, graphs, or any other relevant visual aid. This pattern is made somewhat more difficult to implement with web technologies, which add in the complexity of a client-server architecture. To deal with this, and to also speed up development, the choice was made for CFPS to be constructed using a web framework called Django, with additional code to be written in Python.

The Django web framework employs the use of a MVC architecture created through the use of its object models, views, and templates. The object models can represent any information needed by the developer, and are automatically mapped and converted to a database table representing the information. The models can be just as easily mapped to their visual counter-parts, thus helping with the view component of the pattern. HyperText Markup Language (HTML) usually written to display the needed information can be created dynamically using built-in template tags to incorporate any data that is chosen to be passed to the HTML page for viewing.

The development took place on machines running the Fedora Core 12 operating system, a Linux distribution. Since this was a different distribution than what was used on the target deployment environment, some tools had to be used to allow targeted development. To do this the Mock application was used to allow a chroot to be built with certain packages populated, mainly the CFPS package, along with the PostgreSQL and Python packages. The chroot operation allows a specified directory to be seen as the root directory for certain processes, thereby forcing those processes to be restricted to a certain directory tree.

In regard to the deployment configuration, the project is hosted internally at the TSC on a cluster which provides web services with fail-over capabilities. The cluster itself has four nodes, each running Red Hat Enterprise Linux 5 (RHEL) as the operating system with their own copy of Apache Web Server, PostgreSQL, and any other needed components. The project versioning itself is managed through the built-in package manager using official Red Hat repositories and Extra Packages for Enterprise Linux (EPEL).

All of these tools used would be useless without a process behind them. Of the methodologies available, the scrum practices were used which supported the agile software development principles adhered to. This entailed much group communication through peer programming and code reviews, as well as full unit testing and integration testing. As a team, daily meetings would occur in the mornings where each member would explain what they worked on the previous day, along with what they planned to get done that day, and anything standing in their way. The project was split down into smaller tasks prioritized by a member of the group representing the user, which were worked on by an individual or two. The tasks themselves were estimated by effort required to complete, and several were chosen to be completed such that at the end of each one week iteration the product would be left in a deliverable state. At any point in time the project was to be kept in working condition, through continuous integration accompanied by automated testing. At the end of each iteration a demonstration was first held by the team documenting the product, along with additional integration testing of new features added from the last iteration. After the demonstration a meeting was held for team members called the retrospective, the purpose of which was to discuss aspects of the last iteration which went well, didn't go well, or actions which needed to be taken. Then, after the retrospective one last meeting for the sprint was held, the planning meeting for the next iteration. In this meeting any finished tasks were marked as completed from the list, and new tasks were picked for the next iteration. Overall this process encouraged much communication amongst the team members, creating an environment in which the product could be a result of the true customer needs, which were verifiable at any given moment.

CFPS Contribution

Over the course of the four month co-op position at NAV Canada I have been involved with the development of CFPS. The following paragraphs give a breakdown of the specific tasks which I worked on. With each task I will also convey any challenges encountered, along with how they were approached and inevitably solved. To begin with, I will first relate my academic studies to the work which I have completed.

Many of the tasks completed throughout the term were of little relation to academic studies at a fine-grained level. However, at a higher level the knowledge which I have learnt has helped me tremendously. Rather than having to learn language by language, both the specific language courses that I have taken, and those which taught logic have helped me to be able to see a problem and then apply the process needed to solve it. As a result, the languages required to solve a required problem become less of an importance. Obviously there is much difference between the C programming language and Java, and even a far wider gap between Assembly and an interpreted language such as Python. The point here is that rather than seeing the solution in regards to a specific implementation, it is able to be seen more abstractly in a form which can be applied to any situation. In this way the web application programming course that I have taken also has helped during the work term. The current state and logical flow of the web-based CFPS system was able to be visualized, allowing for ease of debugging and explanation of the systems architecture. Also, the courses that I have had which taught algorithms and recursion have helped, allowing situations which required certain processes to be recognized and code to be implemented which quickly and properly handled the situation. Lastly, a software engineering course taken has helped with some of the skills needed to work in a team

environment, by use of a term project that was done in groups. This course also taught the basics of source control tool usage which was useful in the team environment. However, to my dismay none of the courses taken yet have taught any actual techniques or methodologies regarding the software development process itself.

The first task which I became involved with was creating a message formatter. The purpose of this python module was to take an input flight plan object, and format it into a text string which would be passed over the Aeronautical Fixed Telecommunication Network (AFTN) to other systems ready to receive messages. There were no major challenges here, with perhaps the most advanced aspects of this task being to learn how to use Python and to properly convert the message so that it complied with the protocol. Being the first task I was assigned, and only a few days into the term, I was also given some reading materials on the Scrum practices, Django, Python, and of course, the AFTN protocol. The task was completed in at most a day or two, finishing with a module which accepted a flight plan object and returned a string to be sent over AFTN. Over the course of the term the message formatter - along with several other modules created - were incrementally built upon as the project gained functionality. At one of these changes it was assigned that I was to implement a feature to comply with the AFTN protocol wherein each line of text in the output string needed to be at most sixty-nine characters in length. After several attempts which ended by producing oddly punctuated and unstable output I was reminded of the recursive partition algorithm taught in an algorithm class, and decided to implement the solution in a similar way. The solution ended up working, and achieved the goal by running a recursive procedure on each line of the string that was to be output. The procedure worked by beginning sixty-nine characters into the input

string, and it would search from that point backwards until a specified separator was found. A newline character was then added at that point, and the string was returned with the procedure called on the remainder of the string.

Another task which I chose to work on was the server side flight plan submission form validation. These ended up being Python methods written in the model which corresponded to the flight plan form. As mentioned previously, by using Django all of the data models could easily be converted to forms suited for easy display or storage. The model used for specification of a flight plan was no different. A class was created which derived from the built-in form class which used the metadata of the flight plan object. This in itself allowed the flight plan to be displayed as a form in the HTML page by simply passing in a flight plan form object using Django's template system. From this point there were some problems which were run into. The first of which was a simple versioning problem, for the code that I was writing complied to a newer version of Django than what we were using. It took some time before this problem was found and corrected, causing it to be little more than a small delay along the way. The next problem that was faced had to do with the way data types were being converted into their graphical counterparts. There were several fields which were basically a list of choices, with them being displayed as such; a drop down box of choices. In a few of these cases the choices were meant to have an 'and' relationship rather than an 'or' relationship. In graphical terms, these fields should have been displayed as a list of checkboxes rather than a drop down list. To solve this problem a new field type and accompanying widget were created for the fields to use. The field type class defined the behaviour for its data upon serialization, meaning that it applied a constant method for storage and retrieval to

and from the database. In contrast, the widget defined how the field data was to be displayed to the user; as checkboxes. Overall, this task took a few days and mainly taught the use of Django and Python, as well as how to use regular expressions to check validity of data.

To continue along with the subject of validation, another task done nearer to the end of the term was that of client side validation using Asynchronous Java And eXtensible Markup Language (AJAX). Depending upon the form field, JavaScript code was attached which either tested the input data entirely on the client side, or tested the data by making a request to the server for validation. It is at this point where AJAX comes in. It allows parts of an HTML page to be refreshed without reloading the entire page. This was needed on a few of the fields as their possible values are dynamic, to some extent at least. One example of this is aircraft type, another being the list of available aerodromes for departure and arrival. The code for these types of fields used a JavaScript library called jQuery, which simplified and reduced the amount of needed code. The solution created to handle this problem was created to be generic, and as such it could be applied to any field and work. The JavaScript code worked by using introspection to find the field which it was being attached to, and based on this it would send information to the server containing the correct validation method to call along with the data to be verified. On the server side reflection was then used to get the function to call from the passed in information, and call it on the data. A JavaScript Object Notation (JSON) document was then passed back from the server to the client side, containing the validated data or a particular error message. This whole process worked because of the features offered by both Python and JavaScript, along with the stable naming conventions

Django used for the created form fields. While working on this solution, one small challenge was developing software using multiple languages at the same time. Although it cost at most an extra ten minutes of debugging, I found myself writing JavaScript code using Python syntax, in effect causing the block-structured JavaScript code to not run. This was the most challenging part of the task, as I had done some thinking beforehand of how it was to be solved. All in all, this task reinforced some of the more important concepts behind problem solving relating to computer science; that finding a solution to a problem before its implementation does indeed greatly increase the speed of development.

Another task which I took on was creating the template views, and methods for creation, deletion, viewing, and updating, as well as the submission of a flight plan from a template. Nothing related to this task was overly complicated, with perhaps the most involved element being the re-factoring of the underlying Django models. In the end, both the flight plan model and flight template model derived from a common base class, with each having different additional fields. The template views were very similar to the already created HTML for the flight plan view, with only a few different fields as a result of the difference in the models. Behind the scenes, the methods written which were called when updating, viewing, and deleting templates checked the HTML GET variables which would contain the flight template identifier. Database queries were then called to carry out the respective action. The task ended up being quite straight forward, with one of the challenges being the problem of editing a template. If the name of a template was edited this would cause a new template to be created instead of altering the current template. The reason this ended up happening was that the template name was in fact the primary

key of the template object when in its database form. To solve this the old version of the template would be deleted if a name change was detected while editing a template. From this experience, I again learnt much about the Django web framework, with regard to its template system, and view methods which tied the whole project together.

Conclusion

In conclusion, I set out this term with the hope that at the very least I would be working to develop software. Looking back now, I have done both that, and learnt many more skills as well as developing existing ones. Working on the development of the Collaborative Flight Planning System has been an invaluable experience, giving me knowledge of additional programming languages and frameworks, along with practice using software development procedures in a team environment. Relating to its influence upon my future studies and career, the work done over the term has continued to solidify my interest and skills in computer science, and computing in general. Overall, I am pleased by the experience provided by the co-operative education term, as it has helped to both gain skills applicable to a future career, and to provide increased motivation for me to continue my studies.

Glossary

Aeronautical Fixed Telecommunication Network (AFTN)

AFTN is a network protocol supported by many NAV Canada applications, and worldwide as an outlet allowing the reliable transfer of many different types of messages between aeronautical stations.

Canadian Lightning Detection/Distribution Network (CLDN)

The Canadian Lightning Detection Network provides internal data to NAV Canada regarding information about lightning strikes. The location and time of lightning strikes are recorded, and are displayed via a web interface.

Collaborative Flight Planning System (CFPS)

This system is the next generation internet flight planning system. It allows pilots to file flight plans via a web interface, replacing the decade old IFS system, as a result assisting flight specialists by decreasing telephone calls for flight plan bookings.

Digital Aviation Weather Camera System (DAWC)

The Digital Aviation Weather Camera System is a Canada-wide network of weather cameras which enable pilots to visibly determine any weather effects for their given area. Additional data from the sites corresponding to each weather camera is also displayed. This includes data such as temperature, wind speed and direction, humidity, etc.

Internet Flight Planning System (IFS)

The legacy IFS system provides pilots with the ability to file flight plans. It is currently integrated with the Aviation Weather Web Site.

Flight Service Station Information Management System (FIMS)

FIMS provides flight planning and alerting, weather briefings, and additional aeronautical data to Flight Specialists at any Flight Service Stations. The system also stores the flight plans which are sent to it, allowing the FSSs to provide VFR alerting services.

Flight Service Station Weather Graphics System (FWGS)

The FWGS provides weather display and briefing services. Weather charts, satellite imagery, and weather radar images are mainly supported for display. The system is used by Flight Service Specialists to provide weather briefings.

Bibliography

Aviation Weather Web Site

<http://www.flightplanning.navcanada.ca>

Django

<http://djangoproject.com/>

Git

<http://git-scm.com/>

jQuery

<http://jquery.com/>

NAV Canada

<http://www.navcanada.ca/>

NAV Canada Terminav

<http://www.navcanada.ca/logiterm/addon/terminav/termino.php>

Python

<http://www.python.org/>

References

¹ NAV Canada Website.

<http://www.navcanada.ca/NavCanada.asp?Language=EN&Content=ContentDefinitionFiles%5CAboutUs%5CWhoWeAre%5Cdefault.xml> (22 July 2010)

² NAV Canada Website.

<http://www.navcanada.ca/NavCanada.asp?Language=en&Content=ContentDefinitionFiles\Newsroom\NewsReleases\2006\nr1101a.xml> (22 July 2010)

³ NAV Canada Website.

<http://www.navcanada.ca/NavCanada.asp?Language=EN&Content=ContentDefinitionFiles%5CAboutUs%5Cwhatwedo%5Cdefault.xml> (22 July 2010)