Technical Report

August 25, 2012

Contents

1   Introduction

1.1   Motivation

What is the internship; motivation and description

As part of my undergraduate studies I have elected to participate in the internship program run by the School of Computer Science and Statistics in Trinity College Dublin. The main motivation for this is the personal belief that education is made meaningful by application; that the context of a professional working environment would help to cement the knowledge gained in university. It should also draw attention to area of weakness that need to be improved upon.

Additionally, a professional working environment has a substantially different focus to that of a university. While both depend on the individual to self-educate to achieve a certain goal, the motivation for this, how they go about it, and what goals are set differ substantially. My hope was that an internship would remove the play-pen walls and make me accountable for what I achieved or failed to achieve. University has a focus on collaboration, but in the majority of cases assessment is carried out on and individual

basis. When working on a team, an individuals effort directly impacts the work done by others and as such there is greater accountability for ones actions. I have experienced all of this during a previous, shorter internship and seen the benefit in my attitude towards work and study as well as my overall personal development.

Description here

1.2   Goals

Upon arriving at MasterCard I was introduced to some of the team and my mentor, Peter Groarke. We immediately set to work and a brief road map was laid out. No formal goals were decided but Peter and the others walked me through a series of steps:

- Set up a working development environment.

- Become familiar with software development life cycle at MasterCard and supporting infrastructure.

- Contribute functioning code.

The main objective was to put me to work as soon as possible. This was my own personal objective and, happily, seemed to be in line with the objective of my mentor. As far as I am aware, there was no overall plan for the entire internship at the beginning. An assessment of my skill level was needed before that could be considered.

The first task assigned to me was to interface SMART with Cronos. SMART, Something MasterCard Authorization Rules Tool is an in-house system testing tool used to send mock requests to the Retail API server. Cronos is another server used for retrieving certain system information. My own personal goal was to learn new and interesting programming concepts and technologies.

1.3   Approach (State of the art)

Slightly reflective, what was my approach to the internship. Set goals with mentor, spent time learning system, learning new skills, development. Repeat steps 2 to 4. In reality is more development and learn as I go.

As stated, my initial goal was to start contributing as quickly as possible. This meant checking working code into subversion. The first step was to setup a working development environment.

The majority of the first week was spent trying to install and configure a working GNU/Linux environment. The vast majority of the development work done in InControl targets Unix platforms. Linux is also my personal preference of operating system. The InControl department maintains a script used to automatically configure a new development environment. The idea was to simply setup a working Linux install and run the script. The first distribution I tried was one I had used for years. However, the updated version was considerably unstable when I attempted to deploy it. It took three days for me to identify the problem as the distribution, not the in-house environment setup script. I switched distribution and had a working environment within hours. The next step was to become somewhat familiar with the InControl Platform. This consisted of reading any documentation I could get my hands on, experimenting with the system itself and looking through the system database, the core of the InControl platform.

InControl have a substantial but somewhat poorly indexed wiki which become my main source of documentation, other than the developers. Aside from the InControl platform I also tried to understand how credit card transactions were routed and processed. This was necessary to give the InControl platform context.

A large portion of my first few weeks was spent learning how _____ Complete

1.4 Overview of contents

2 Internship in Context

Orbiscom Ireland, acquired by MasterCard in 2009, develops a product called InControl. The acquisition allowed MasterCard to incorporate InControl into their Value Added Services, a range of products that tie in closely with their core business, the processing of electronic payments. Until 2010 MasterCard Ireland was still primarily concerned with the development of InControl. A division of MasterCard Labs was setup within MasterCard Ireland in 2011. As MasterCard continues to grow, more and more departments are being given a presence within Ireland. This report documents the activities and projects undertaken while working for InControl. Since their

acquisition Orbiscom have become completely integrated into MasterCard and have ceased operating as Orbiscom entirely. InControl has been split into two different products. InControl Direct exists to support Orbiscoms original customers as is managed and maintained entirely within the InControl department. The second product, InControl is an adapted version of the original product. Where InControl Direct is deployed to and hosted by banks, InControl is hosted on BankNet, MasterCards global payments network. In order to explain the services offered by the InControl platform, it is useful to explain how the credit card payment process works.

InControl and Orbiscom are used somewhat interchangeably in the rest of the document. Orbiscom is a keyword used throughout the code base. All of the packages still begin with *com.orbiscom*

2.1 The Four Party Model

The credit card payment scheme employed by MasterCard involves four separate parties, for this reason it is referred to as the four party model. There are alternate models in use, but MasterCard employs the four party model as it allows the issuing of payment cards to be handled by separate financial institutions. This leaves less overhead for MasterCard and also insures interoperability between the various financial institutions. The credit card holder typically initiates the payment and is represented by a credit card issuing bank, or an issuer. The merchant involved in the payment is represented by an acquiring bank, or an acquirer. Each successful payment goes through three stages; Authorization, Clearing and Settlement.

2.1.1 Authorization

The card-holder submits their payment card details to the merchant. The merchants bank, the acquirer, sends a request to MasterCard to identify the card-holders issuing bank. Once this has been determined and the card has been verified the payment is forwarded, by MasterCard, to the issuing bank. It is worth noting that this is the stage in the process where the InControl service is applied if needed. The card-holders bank approves the purchase and blocks the funds in the card-holders account. No money has been transferred from the card-holders account, the money has merely become unusable by the card-holder. The issuing bank forwards the approval

6

to MasterCard, who forwards it to the acquirer who in turn forwards it to the merchant. The payment has now been approved.

2.1.2 Clearing

Sometime after the payment has been authorized, usually at the end of the week, the merchant submits all of their authorized payments to clearing. This is a batch process that occurs at certain set times. After a payment has been cleared the funds have effectively been transferred.

2.1.3 Settlement

Settlement concerns the actual movement of funds. This is handled entirely by the banks.

To facilitate this, a fee is applied to each transfer. Interchange is typically charged by the issuer to the acquirer. This is also where MasterCard makes money from each payment. The rate of interchange varies from bank to bank and can even be different depending on the nature of the purchase. In order to sustain credit card usage and adoption the services offered by MasterCard must outweigh this additional fee.

Interchange fees and who pays them is subject to much debate. This report will simply assume the following. The interchange fee is to some extent ultimately paid by the merchant. This means that they lose an amount on every purchase made with a payment card as opposed to cash. In order for merchants to continue to accept card payments there must be sufficient consumer pressure for acceptance by the merchants. This is done by incentivising consumers with additional benefits not available through cash payment. Some of these incentives are part of MasterCards core network and come as benefits of using an electronic system such as added security, access to credit and accountability. MasterCards Value Added Services range is a set of products designed to add additional benefits for consumers. [1]

2.2 InControl

The InControl platform provides personalized, real-time card controls. It allows consumers to set limits on their credit card and monitor their spending. It is a service provided by MasterCard to credit card issuing banks, henceforth referred to as issuers, who in turn provide the service to their

customers. The platform offers a number of services. These are sold as five separate products.

- Cardholder Alerts & Controls: Allows consumers to establish personalized spending profiles on their accounts, and allows for real-time notification of spending activity.

- Issuer Portfolio Control: Aims to help prevent cross border fraud in Europe by placing controls on large sets of credit cards.

- Small business controller: Allows small business owners to confidently delegate spending to employees.

- Purchase control: For large corporations. The service offers unique limited-use account numbers and authorization controls to improve reconciliation for card payments and enhance visibility into spending activity.

- Family Solutions: Allows parents to give family members access to credit.

At a more technical level, the InControl platform provides two services; the creation of virtual card payment numbers and the creation of rule sets to be applied to credit card transactions.

The virtual card numbers (hereafter referred to as VCNs) are linked back a real card and account and can be used exactly like a normal payment card. At the core of the system is the algorithm used to create VCNs. Similar systems have also existed, but the unique feature offered by InControl is that no changes to existing infrastructure are required. If a payment requires InControl to continue processing then the payment is routed to the InControl platform. This happens entirely within the payment network. Neither the merchant nor the acquiring bank has any knowledge that InControl has been applied, so no action is needed on their part. Other similar technologies required the merchant to add knowledge of the process to their point of sale, i.e., they had to replace all of their card readers.

InControl allows an extensive range of controls to be set on payment cards. Rules can be placed on the amount spent, where it is spent, what it can be spent on, along with many others, can all be controlled. These controls can be applied by individuals to single payment cards, known as

Account Level Spend Rules, or by banks to ranges of issued cards, known as Issuer Portfolio Control Rules. The ability to control personnels spending has proved a popular feature. The bulk of InControls traffic is made up of corporate accounts. Many businesses have found that VCNs are an effective way of managing employee spending.

The original InControl Software platform was designed as a generic server framework to host various payments services such as InControl VCNs as well as other third party technologies such as Verified by Visa. The VCN service was developed by Orbiscom. If a bank chose to provide this service they could use the InControl platform to host other financial services as well. The entire platform is written in Java and configured using XML. Work began around the same time as the first versions of Java Enterprise were being released. This had a very clear effect on the development of the platform. InControl was designed as a generic server framework because no suitable server framework existed at the time.

MasterCard operates BankNet, a global telecommunications network linking all MasterCard card issuers, acquirers and data processing centres into a single financial network. The operations hub is located in St. Louis, Missouri. BankNet uses the ISO 8583 protocol [2] . The network is peer-to- `fix` peer mesh network with a set of endpoints. At no time during my internship did I have any involvement with any aspect of BankNet, but it does have one very notable effect on development within the company, the release cycles.

BankNet is the core of MasterCards business. One of the most important aspects of an electronic payments service is reliability. If MasterCard is unable to serve its customers in anyway it would mean a huge blow for their brand, a brand they have invested a substantial amount of money in. BankNet must be reliable. This is achieved in part with an extremely conservative attitude towards updating the software that controls the network. The servers can only be restarted twice a year. This takes a considerable amount of effort. Each node must be updated or "flipped" across the entire network. The network cannot be taken down completely during this time and there are many possible issues that are taken into account, hence the twice yearly cycle. There are a further two periods each year where the network can be updated without restarting the core services. All of this means MasterCard uses a quarterly release cycle, with fixed deadlines. Due to the inflexibility of the release cycle, great care is given to selecting

new functionality to be included in each quarters release. These factors, a rigid deadline, fixed requirements and a very strong reliance of reliability in the end product are traits commonly associated with the waterfall design process, and this is the design process employed by MasterCard.

The waterfall design methodology arguably fits well with the requirements surrounding MasterCards core network. Other services, such as InControl are not subject to the same requirements but are none the less subject to the employment of waterfall. During my time at MasterCard a switch an agile design process was beginning to get underway within InControl.

2.3   Software development life-cycle

An InControl release goes through several phases before it is shipped. Throughout this process a large amount of attention is payed to quality control. The release cycle resembles a standard waterfall type model. However, development and in particular the testing phase are divided into several sub-phases.

Before any work is done a customer or product owner must first request some new functionality. This takes the form of a high-level requirements document which is sent to a Business Analyst within MasterCard.

The Business Analyst can then approve or discard the feature request. If it is approved, they translate it into business requirements and functional requirements.

At this stage, the development team are engaged. A design specification is created, based off of the functional requirements. This specification is sent back to the Business Analyst for approval. This process may be repeated several times until the BA is satisfied that the design specification accurately reflects the functional requirements.

After the design is approved, the Quality Assurance team are engaged. The QA team begin to create system tests based off the design specification. An additional team, the Testing Center of Excellence or TCOE, create integration tests. The integration tests are to verify that the system will interact correctly with BankNet.

The release then undergoes development, testing and eventual release. The process so far is not strictly waterfall. There are mechanisms for feedback and in practice a certain amount of iteration to achieve the desired result before moving onto the next phase. Development and testing within

InControl is largely an iterative process. Development and QA work together closely to ensure no bugs are introduced. Typically during this phase several releases will be made to QA. After the functionality has been completed and QA are satisfied with its quality it begins a process of progression through several testing environments, culminating in release.

The first environment is used to conduct the integration tests. The release then moves to user acceptance testing. After this has been completed the release is made available to customers to make sure it is what they wanted. After all of this has been completed the release is ready to be shipped. During any one of these stages a problem may be found with the implementation or even specification of the release. It could be argued that the development process employed by InControl has some agile elements. The process is somewhat iterative, with time set aside to allow for changes to be made mid-cycle.

As a member of the development team I became quite familiar with the practices employed. The InControl development team have a very open approach, which sometimes goes against the attitude adopted within the rest of MasterCard. Each member of the team is given freedom to go about their work in whatever way they deem to be the best approach. Everyone has root access to everyone else's machine. Development roles are defined, but not in any database. Any developer can modify and deploy any component. In practice this creates a very flexible working environment. No one is going to modify a component and release it without consulting with the team. The lack of restrictions exists so working on separate components, if it is required, as fluid as possible.

2.3.1 Terra

Terra is the build system used to manage projects within InControl. It is a web application developed in-house with a number of features. For the sake of brevity only the core project management features will be discussed; building, deploying and administering a project.

Terra is used to automatically resolve dependencies and simplify building software projects within InControl. A project can have any number of dependencies. These dependencies can be either external jars uploaded to Terra or components managed by Terra. These dependencies are automatically updated when their corresponding component is updated.

terra build system

Edit Component

Field	Value
Name	r12q4Smart
FileType	tar
Repository	mastercard ▼
Type	PRODUCT ▼
Module	r12q4Smart
Component Directory	.
Build Command	ant disttar
Build Output	dist/r12q4smart.tar
Description	Smart Tool - creating rules on CPN's and putting through adhoc authorizations
Copyright	Orbiscom
Customer	MasterCard ▼
License	Orbiscom ▼
Profile	JDK1.6 ▼
Build Dependancy Directory	lib
Documentation Directory	

Name	Build	Test	Run
abu12q3	☐	☐	☐
abu12q4	☐	☐	☐
abu13q1	☐	☐	☐
activation-1	☐	☐	☐

The Terra administration page for SMART

Terra is also used to build components for release. When the developer is satisfied that requirements have been implemented sufficiently they can "tag and build". This sets the release number and builds the component. The command used to build the project is specified in "Build Command" in the above image. This corresponds to an entry in the projects Ant configuration file. The newly built version is then available for download any will be

updated in any dependent projects.

Finally, Terra can be used to quickly deploy projects. A component version and target machine are selected. Terra then logs onto that machine and sets up and runs the component automatically. This can save a great deal of time when debugging. Due to the number of components used by the InControl platform simply setting up an environment to reproduce an error was quite time consuming. Terra simplified this process.

Releases are managed by Andrew, an new web application also developed in house. Essentially it groups specific versions of components into an archive that can then be released. It interfaces with Terra to simplify selecting the components to include within a release.

Terra is also very useful when someone wishes to begin working on a project. After the project is checked out from the subversion repository it only takes a few simple Ant commands to fetch all of its dependencies and configure the project. All of these dependencies are downloaded from Terra.

3 SMART

The MasterCard Authorization Rules Tool (MART was taken) is an in-house platform testing tool used to send mock data to the Retail API Server.

3.1 Overview

The bulk of platform testing preformed within InControl is done manually. SMART is used to somewhat simplify this process by providing a GUI application to generate mock requests for a wide range of possible use cases. It began as a small application to generate account level services rule sets and submit them to the Retail API Server. Over time it grew into a testing platform in its own right, capable of exploiting the vast majority of functionality offered by the Retail API Server and thus the InControl Platform. SMART provides an interface into a reasonable slice of InControl platform functionality. The InControl platforms main function is to provide virtual card numbers and enforce rules associated with them. SMART allows for the creation of virtual and real card numbers, enabling InControl services on those cards and assigning various controls and services.
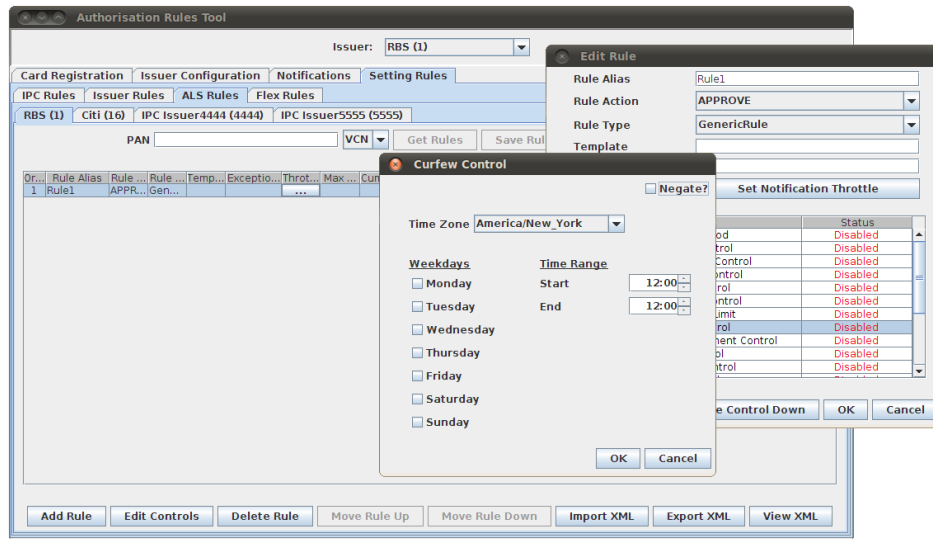
Creating a new curfew control in Smart

3.2 Objectives

SMART is used by InControl developers, the InControl Quality Assurance department and several other offshore departments involved in the development and testing of InControl. The vast majority of work performed during the internship involved the development and maintenance of SMART. The work done can be split into two categories, integration of SMART with Cronos and maintenance.

SMART was my first task, so the initial goals were reasonably broad and changed as the work progressed. The first task, to integrate SMART with Cronos had two objectives. First, to produce some sort of meaningful contribution, and second to allow evaluation so future tasks could be planned with respect to my abilities. After this task was completed I was given official ownership of SMART and tasked with maintaining and updating it.

3.2.1 Cronos

Cronos is a dynamic InControl server. It provides functionality that is not exposed to an end user or issuer. Its purpose is to help automate some of the steps needed when adding a new issuer into the InControl system.

Adding a new issuer is not a trivial process. There are several significant database inserts required. These steps differ depending on the product the issuer intends to use. A large amount of the set up data is static and thus a large amount of the set up process can be automated. Cronos can also be used to retrieve certain information from the system.

I was tasked with extending SMART to allow easy interaction with Cronos. There was no existing way of communicating with Cronos other than sending raw XML requests. The desire was to have SMART provide a GUI interface to create these requests and view the responses. In order to accomplish this I had to learn how to interface with the existing InControl components. The amount of time taken to do this was vastly underestimated as I had no comprehension of the code complexity of a commercially developed software platform. To complete the work successfully I broke the task into several subtasks;

- Download, configure and use SMART.

- Become familiar with the structure of code SMART.

- Get basic communication with Cronos working.

- Create an interface for Cronos in SMART.

- Add support for all of the Cronos requests into SMART.

The main problem encountered while trying to initially use SMART was configuration. As previously described, most developers have slightly different ways of doing things. I was unaware of a running instance of the Retail API server because the developer I asked never worked on it, and their work flow usually involved running any needed components locally. An instance of the Retail API Server was setup on my local machine. Terra (p. 11) automated this process to a large extent but some configuration information still had to be specified manually. Once this was done some time was spent using SMART.

3.2.2 Maintenance

As SMART is a testing tool I began to research software testing and read some of the surrounding literature. I quickly realized that while there is a lot of information and discussion about how to test software, what to look for

15

and the strengths of a particular method over another, there is very little on how to implement the tests themselves. The Standard for Software Component Testing, produced by the British Computer Society does not describe how to achieve the required attributes of any test process [3]. This seemed odd at first, but after further reading I understood such a description is impossible. The essence of a software entity is a construct of interlocking concepts. The difficulty in software development is the specification, design, and testing of this conceptual construct, not the construction or its verification. [4] Every piece of software has a different set of requirements and thus a different design. There is no true standard for developing software so there is no true standard for testing it, just a collection of additional concepts that may be employed.

> In a broader view, we may start to question the utmost purpose of testing. Why do we need more effective testing methods anyway, since finding defects and removing them does not necessarily lead to better quality. An analogy of the problem is like the car manufacturing process. In the craftsmanship epoch, we make cars and hack away the problems and defects. But such methods were washed away by the tide of pipelined manufacturing and good quality engineering process, which makes the car defect-free in the manufacturing phase. This indicates that engineering the design process (such as clean-room software engineering) to make the product have fewer defects may be more effective than engineering the testing process. Testing is used solely for quality monitoring and management, or, "design for testability". This is the leap for software from craftsmanship to engineering.

[5]

The testing done within InControl is done to ensure platform reliability. This is carried out through black box testing using techniques such as Equivalence Partitioning and Boundary Value Analysis. As the specification of the inputs used in these techniques is essentially arbitrary, SMART must allow for arbitrary input. In practice, SMART is able to construct structurally correct InControl XML requests containing arbitrary data elements. SMART does preform some checks before sending the requests. I decided to

leave practical checks, such as not allowing the user to input a string when a number is expected, but removed any controls I saw as blocking possible test cases. The argument can be made that any checking done by a testing tool is blocking a valid test, but there are practical considerations to make. SMART has a small group of users, but these people spend many hours per week using it. Some checks done within SMART exist to make the users life easier. Furthermore, some of the basic checks are simply out of scope for testing, i.e. It is expected that types will be enforced. It is well understood that entering a string where a number is expected may happen, and there are well established mechanisms already implemented to handle such cases. Conversely, there are scenarios that need to be tested. A card might be registered on the system with an expiry before the date of registration. The expected behaviour here is ambiguous and depends on the design of the system, so it makes for a valid test case. This was the balance I tried to achieve while working on SMART.

reflect

3.3 Description

SMART began as a very small Java GUI application to allow controls to be placed on a card for testing. It has since grown into quite a substantial testing tool, consisting of over 30,000 lines of code. Its high level function is very simple, to allow a user to easily create valid XML messages and send them to the Retail API server. It has grown to be quite complex because of the varied nature of the requests it constructs. The UI contains most of the code, about 25,000 lines. It could well be argued there are many opportunists in the design to shrink this number down, as there is a lot of duplication, but it gives a clear indication on where the focus lies. The code follows an MVC type of model. This is reflected in the projects package structure . The model is supplied by external libraries. These libraries contain the request objects and their supporting data objects. All of these libraries are generated by the OIL compiler (p. 35). The OIL specifications for these libraries are contained within their respective components, e.g. Pulse is the library used to communicate with the Retail API Server and the OIL definition file is part of the Pulse project. The Pulse library is marked as a SMART dependency within the build system, Terra (p.11). OIL is an interface definition language. The model, in its entirety, is generated automatically. This automated process, a small simple specification trans-

Diagram of package structure

17

formed into many thousand source lines of code, helps to eliminate errors from the model almost entirely. Model objects are generally quite simplistic in nature but are often large, particularly when the model is considered as a whole. If they were written manually there would be many opportunities for error. The use of an IDL language also ensures consistency of the model design throughout the platform. [6]

The controller, as expected contains the logic to translate the request and responses issued to and from the server into objects that is represented to the user. This is also where it begins to become pretty clear that SMARTs design does not conform to an MVC structure very strictly. The main purpose of an MVC design is to decouple data from its presentation. The data, in the form of the model is represented by the view. When a model changes, the view can be updated to reflect this change. Representing data as abstract objects allows for greater decoupling. The underlying structures of a model or view can change completely so long as the presented behaviour does not change. [7] SMART does not pass model objects from the views to the controllers. Rather, the view passes a number of simple data types to the controller which are assembled into a model object and sent to the server. The result is a massive amount of code duplication. One method is used to send the request, but each separate request has its own separate method used to construct it. Similarly, each request has its own distinct view. This was something I wanted to avoid doing. When I began work on SMART I created sub-packages within the MVC structure to contain my work. This was to make the difference in functionality absolutely clear and also so I could learn more about interacting with the InControl platform by doing so from scratch while using the existing code as a base.

As SMART is used heavily by the QA team to interact with the Retail API server a lot of bugs logged against SMART are in fact problems with the Retail API server. The vast majority of these bugs could be traced to differences in environments or uncompleted code being checked into the build system. I did however find some small bugs in the Retail API server misclassified as SMART issues.

3.4 Cronos Integration

After reading some of SMARTs code, testing its functionality and reading relevant InControl documentation work began on the new features. SMART

had previously only interacted with the Retail API server. It was to be extended to interact with Cronos, another server within the InControl platform.

3.4.1   Initial work

The first goal set out was to get SMART to communicate with Cronos by sending a simple request. Cronos sub-packages in the API and View packages where created. A simple request object was created and sent in a requestor object to the Cronos server and the response printed to the terminal. These first few steps took quite a while. There was some documentation but this initial work was based mainly off the existing code in SMART. The existing code in SMART seemed needlessly complicated and it was quickly assumed there was a better way of achieving the same result. The existing code in SMART for sending a request executes as follows:

- Each individual request has a unique trigger somewhere in the view, usually in the form of a button.

- This trigger is hard coded to a unique call in the API class. A number of simple data types are passed from the view to the controller (API) via this call.

- This initial call within the controller constructs a model object representing the request and passes it to send method.

- The send method decomposes the request(s) into their XML representation. A HTTP request is then constructed and sent to the server.

- A HTTP response object is returned by the server and the XML response object is extracted.

- The response is passed back up and a model object is constructed from the XML.

- The model objects populated by the response data are sent back to the view where the relevant data is displayed. In many instances, a single simple type is passed back up to the view instead.

I was unaware of the hierarchal structure of the requests at first but immediately questioned the need to manually deconstruct the model to XML.

After talking one of the original system architects a far simpler approach was devised.

3.4.2 Generic approach

- Trigger each request from the view. The trigger should populate a model to send.

- This model is passed into the API and sent to the server. Each In-Control request to be sent has a distinct representation, but they all extend a request interface. These request objects are contained within a requester object, which can also be used to send the requests.

- The request responses, which all implement the response interface, are passed back to the view where they are cast the relevant response and displayed appropriately.

This method when implemented saved a large amount of time. The controller was significantly smaller than the existing one used to communicate with the Retail API server, about 200 lines of code compared to 4000. It also required far fewer supporting objects. While the Cronos controller may not have had some of the more advanced features featured in the Retail API controller, the vast majority of the code cut out was strictly boiler plate.

3.4.3 Dynamic approach

There was still the issue of having to manually create the request object in the view and populate it with data supplied by the user. This issue is a more complex one. The manner in which the user interacts with the data is quite important. However, as the initial task was to simply display lists of information a table could be used to represent display the data. It was decided, although not know at the time, that there must be a way of doing this generically. The idea that every requests object must be created and populated manually was not one worth perusing because of the sheer amount of time it would take. There was also the problem that very little could be learned by writing large amounts of boiler plate code. The option to simply generate the code was also considered but such an approached would not solve the problem as the end result would still be a large amount of boiler plate code. Creating the requests dynamically proved to be difficult. I was

not aware of any method that would allow be to create objects dynamically, i.e. to create an instance of an object and populate it without any knowledge of the object at compile time. The functionality desired was outlined and a solution was simply presumed to be possible.

After the work was done several developers pointed out that this was a somewhat unusual way to approach a problem. In order to properly design a piece of software some idea of how to implement it is required. The approach I took was one I see being encouraged during my time at university. The mentality of experimentation is actively encouraged. The emphasis in a working environment is lies strongly with code robustness and readability, not obtuse code overly dense in functionality. The desired functionality was as follows:

- Populate a list of requests at runtime and allow the user to select one. An interface should be created to allow the user to supply any needed attributes.

- Construct and send the requests using the method described above (p. 20)

- Display the response(s) in a list, giving each member a meaningful name.

This body of work was completed over several weeks. A few different approaches were tried before the final approach was decided upon and implemented.

Requests are specified in and XML configuration file, along with a name to display to the user. SMARTs toplevel configuration file has a link to this configuration file. Using this link, the request specification is read at runtime and stored as an InControl configuration object. A configuration file object simplifies the process of reading an XML configuration file and is used throughout the InControl Platform. It reads in the configuration file and allows its contents to be accessed using a simple XPATH notation. The requests are also grouped by functionality, such as requests to retrieve, insert and update information.

Once the requests are read in a list of the request objects is created. This took a long time to figure out how to implement correctly and many, many approaches were tried. Java supports runtime creation of objects in

pageref this

a number of ways, ranging from the more complex to the arrived solution which is relatively simplistic. [8]

```
Request request = (Request) Class.forName(requestClass).newInstar
```

These requests are then stored inside one of several hash-maps, one for each type of requests, i.e. get, set, update, indexed by their human readable name. The requests now in memory were useless if their members could not be accessed. The first method used to achieve this was reflection. Reflection allows a Java program to introspect upon itself at runtime. A list of method names can be extracted from an object and called. All of this is done at runtime, with no compile time knowledge of the objects structure needed.[9]

This approach worked but the code produced was obtuse. Java is a statically typed language and was not designed to support this kind of behaviour. After querying some developers and some reading it was decided that reflection should be used sparingly. Different languages have different strengths and encourage development in certain ways. Departure from these conventions, while at times necessary, can produce code that is very difficult to read. Maintenance is widely considered to be the largest cost in software development . If a piece of code is quick to develop but very hard to understand then it is often regarded as a bad piece of code.

For this reason it was decided a different solution to the problem was needed. Thankfully the InControl framework, specifically OIL, was designed with this exact scenario in mind. As stated (p. 17) OIL generates all of the requests objects and their associated data types. The resulting objects are more complicated than a collection of simple data types and their accessors. An OilBean, an abstract class extended by every OIL generated object, contains a hash-map of member objects. These members can be accessed via unique accessors or directly via the hash-map. This completely eliminated the need for reflection. The OIL generated classes could be introspected at runtime simply by accessing this hash-map. The map also contained metadata for each member. The metadata object contains information such as member type, any constraints, the container (i.e. If the member is a list of attributes) as well as several other bits of information. This allowed the OIL objects to accessed in a static manner, and resulted in code that was substantially easier to understand and write.

The hash-map of members was also used to generate the user interface

Reflect on why this took so dammed long

CITATION NEEDED

There is not citation, coding is based off heuristics etc.

at runtime. The requests were partitioned by functionality so separate interfaces could be used to display them. Get requests return large amounts of information. There were two main considerations taken into account when displaying this information. The user should be able to view all of the information quickly, and have the ability to define filters to find the information they want. A table was used to allow the user to view the information quickly. The table is generated dynamically from the response data. Certain request responses contain values that can be used to index further requests. Such requests can have a number of related requests specified in its configuration.

```
<Request Name="Bin Ranges" Class="GetBinRangeList" >
    <Related Name="Vcn Bin Range"/>
</Request>
```

This manifests to the user in the form of a right click menu. If a request has one or more related requests, the user can right click on any of the table items and choose from the related requests. The members of the selected row of response data are matched against the parameters to the selected requests. The matching is done by member name, i.e. If a response contains the member "CardId" it can be used to index a GetCardById request. When a match is found, the request is sent using the values from the selected row. This allows a user to quickly drill down through the data. Prior knowledge of what requests are related is needed when writing the configuration file. No checking is done at run time. If a request is specified as related and the parents response does not contain any related members and error is thrown.

Tabs: Card Registration | Issuer Configuration | Notifications | Setting Rules

Sub-tabs: Locale | Lookup | Add | Set | Stats | Categories | Add issuer

Issuer: MC-TestIssuer (HSBC) (1)

Request Type: Bin Ranges ◄ Send View Filters ☐ Apply Filters

Context menu: Vcn Bin Range | Bin Range Properties | Card Products

#	CardConfigId	Prefix	StartRange	EndRange	CardConfigType	CardType	CpnType	Status	LanguageId
157	500011	500011	5000110000...	5000119999...	R	70	DF	A	1
158	511000	511000	5110000000...	5110099999...	<	70	EV	A	1
159	510101	510101	5101010000...	5101019999...	<	1	EV	A	1
1	541333	541333	5413330000...	5413339999...	R	1	DF	A	1
2	611634	611634	6116340000...	6116349999...	R	2	RC	A	1
3	556951	556951	5569510000...	5569519999...	R	3	RC	A	1
4	553257	553257	5532570000...	5532579999...	<	1	SP	A	1
5	690478	690478	6904780000...	6904789999...	<	2	SP	A	1
6	520478181353	520478181353	5204781353.30...	5204781353.39...	<	3	SP	A	1
10	552233	552233	5522330000...	5522339999...	<	19	SP	A	1
29	512312	512312	5123120000...	5123129999...	R	19	DF	A	1
30	529292	529292	5292920000...	5292929999...	<	21	SP	A	1
28	518007	518007	5186070000...	5186079999...	<	1	DF	A	1
32	588007	586007	5886070000...	5886079999...	<	19	SP	A	1
33	589007	589007	5896070000...	5896079999...	<	19	EV	A	1
50	553258	553258	5325800000...	5325899999...	<	1	EV	A	1
40	566566	566566	5665660000...	5665669999...	R	21	DF	A	1
41	556556	556556	5655600000...	5655699999...	<	1	EV	A	1
34	616161	616161	6161610000...	6161619999...	<	1	EV	A	1
35	666111	666111	6661110000...	6661119999...	<	2	EV	A	1
38	565656	565656	5656560000...	5656569999...	R	1	DF	A	3
39	555666	555666	5556660000...	5556669999...	<	1	EV	A	1
186	581234	581234	5812340000...	5812349999...	R	1	DF	A	1
286	593211	593211	5932110000...	5932119999...	R	93	DF	A	1
287	512033	512033	5120330000...	5120339999...	<	93	SP	A	1
285	541010	541010	5410100000...	5410109999...	<	92	SP	A	1
284	569090	569090	5690900000...	5690909999...	R	92	DF	A	1

Table view of request response data and related requests.

Response data can also be filtered. Custom filters can be defined on any column in the table. The user specifies a string to pattern match on. A basic match, analogous to an sql like filter, is then performed on all of the entries in that column. Any number of filters can be applied to any number of columns. These filters are then joined using an "and" operation, allowing the user to precisely locate a piece of information. Again, all of this is done at runtime based on the response data.
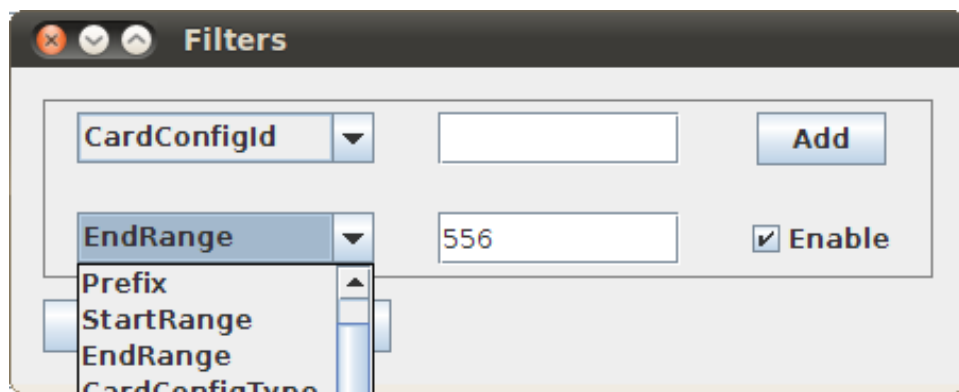
Filter creation.

Pad by 25mm?

Add and set requests (sql insert and update respectively) require a different user interface. The primary concern here is allowing the user to enter a number of parameters quickly. The parameters for a given request are used to populate a panel. This is done in a relatively simple manner. A grid is created and the attributes are inserted into the grid. An attribute consists of a label for the name and a component used to input data. The input components change based on the value of the attribute. The possibilities are a text-field restricted to a string or a number, or a combo-box for binary attributes. OIL allows default data to be specified when defining a request. This default data is hard-coded into the generated objects. If default data is present, the relevant UI elements are created with this default data already inserted. This saves the user from having to re-enter the data, but it can still be changed if desired.

A slight concession was made when handling binary attributes. OIL

does not have a binary type, only string and number. Binary attributes are simply strings that can only be "Y" or "N". If an attribute had a "Y" and "N" as default data, it was presumed to be a binary attribute and displayed as a combo-box.

Strict correctness is at times compromised to create a slightly better user experience. Automatically generating the user interface resulted in substantially less code, about 2,000 lines compared to 20,000 lines. However, unlike the API classes where most of the functionality was retained, the existing user interface contains a lot of unique functionality. The auto generated UI sacrifices much of this functionality in favour of a more generic approach. While this did save a lot of time writing the API, such an approach is harder to justify for a UI. Coding each UI individually, while liable to creating a large amount of redundant code as in SMART, leads to greater level of customization overall.

Generated UI for an AddIssuerScheme request

3.5 Maintenance

After the work integrating Cronos was completed focus shifted to maintaining and updating SMART. SMART is used heavily by members of the Quality Assurance team within InControl, as well as several off-site teams. Due to this relatively intense usage bug reports are fairly frequent, some two hundred and twenty plus since work on SMART began in late 2010, nearly all of which are now closed.

There are a few important factors that contribute to the amount of bugs in SMART. The primary reason is shifting requirements. SMART began as a very small application to allow the creation of card controls. Over time it has evolved to leverage the majority of the functionality provided by the Retail API server. With each new release of the Retail API server comes new features, and hence new bugs are introduced. These updates must be reflected in SMART. Sometimes these new features are implemented in SMART after they are implemented in the Retail API, when QA is ready to begin testing. This was not always the case, but there were many times when new Retail API features were disclosed only after they broke some of SMARTS functionality. Another contributing factor is the nature of SMARTs usage. It is QAs main "window" into the Retail API server, so many bugs are miss-classified as SMART issues when they are in fact problems within the Retail API server.

The final factor is the nature of QAs test cases. A certain chunk of Retail API functionality may or may not be tested to the same degree in each release. Typically, bugs in SMART are found when a certain piece of functionality is exercised extensively in-line with a particular test-case. A brief summary of the work done is included in the appendices

3.5.1 New functionality

After it was established I had the capacity to maintain SMART I was tasked with updating it to allow testing of new features in the Retail API to be released in quarter three. This was done on a very tight schedule, the first release was to be made to QA three days after the task had been assigned. This release was important to InControl. It had been decided to

[Reflect on using oil the way it was intended without instruction]

[3rd person for everything]

[SVN LOG]

use InControl to combat fraud, within the German market at first. This was one of the biggest releases ever for InControl in terms of new issuers being on-boarded. The aim was to utilize the existing InControl rules, along with its well developed support services to cut down on cross boarder fraud. The release window was tight in order to compete with similar a service being offered by another large credit card company. Although SMART is by no means core to an InControl release, it does help speed up testing and ultimately the release cycle. There were a number of items that had to be completed in SMART;

Pagerec IPC and ALS

- The addition of a new product, "Fraud Control", to be enabled on a card.

- The addition of a velocity control within the "Issuer Portfolio Controls".

- Retrieval of rules associated with a card based of an ID.

- The addition of a new Region control within both the IPC and ALS controls.

- The addition of a new Multi De control with the IPC controls

These tasks varied substantially in complexity and together serve as a good illustration in the variance of work carried out on SMART. This was also the first time the existing SMART code had to be modified since my work began on it. Up until this point the work done was in relative isolation to the rest of SMART. Not only did these changes have to be implemented, the structure of SMART and how it operated had to be understood in some detail. Additionally, these changes were not yet completed within the Retail API, so once a feature was implemented there was not always a way to verify correct operation.

The first task was a simple addition to the toplevel configuration file, an extra element had to be included.

A velocity control allows a spend limit to be applied to a card, either cumulative over a time period or a maximum allowed spend. This control was already implemented in the ALS controls section, so it was merely a matter of finding it and copying it across to the IPC section and changing the request sent out. In practice this functionality was not as trivial to copy

as both the UI and the API needed several changes. This change was not tested as the corresponding call had yet been implemented in the Retail API server. This was my first exposure to seat of your pants coding.

IPC Velocity control user interface.

IPC rules are applied to ranges of credit cards, specifically the bin range . An IPC rule set is assigned a name. This name is used to retrieve rules from the database for a range of cards. The new release mandated that the card number could be used to retrieve the associated IPC rule set. SMART was modified so they key used to retrieve the rules, the profile name or card number, could be specified by the user. The appropriate request was then called. No modification in how the rules were displayed was needed, save for not allowing rules to be updated when retrieved with a card number. These changes were relatively minor but still required a moderate amount of modification to the code.

InControl has a control to limit card usage based on country. It was decided to group countries into regions and add it as a control. This was the core feature of the release. German credit cards are stolen and then used in countries with old infrastructure and authentication, i.e. Countries without chip and pin. The existing Country control code was used. All

that needed to be changed was the data assigned into the control, a list of regions as opposed to a list of countries, and the underlying request sent to the server. This control was to be added to both IPC and ALS rules. Due to SMARTs code structure, this meant a large amount of code had to be directly copied. This effectively duplicated the amount of time needed to test the change.

The addition of the Multi De control was done in a similar way to the Region control. There was a pre-existing De control . Happily it had been implemented so the user could create multiple De controls within the one interface. These were then assigned to the rule set as distinct controls. All that had to be done was take these distinct controls and assign their values into one Multi De control.

These five tasks are representative of the changes I made to SMART with respect to the Retail API server functionality.

3.5.2 Bug fixes and testing

After the fraud control work was completed I was assigned as product owner of SMART and charged with general maintenance. The InControl development team use a software package called "JIRA" to track bugs within the Dublin office. A typical JIRA report contains the issue or "ticket" number, effected components and release, the steps to reproduce the bug, the bug status, i.e. Fixed, In progress etc., the person who reported the bug and the person assigned to fix it.

What is a DE control

Mention that this somewhat exceeded the expectations of what I could do

A JIRA ticket.

When the maintenance work began SMART had a few outstanding issues. Everyone in QA was more than happy to answer questions and supply feedback as it had been some time since someone had done regular maintenance on SMART. The typical work flow for a given bug was:

Replace ";" with ":" in most places

- Follow the steps to reproduce and determine the nature of the error.

- If the issue was determined as a genuine SMART issue, i.e. Not a local configuration issue, mark it as requiring a fix in JIRA. If it was determined to be an issue with the Retail API server no action was taken in JIRA until the problem was found.

- Locate where the error was happening in SMART and fix it. If the error was the server it was usually due to one of three reasons.

  - The server was released with an incorrect configuration file.
  - Incomplete server code was checked in. This was my fault for not building SMART against full server releases, usually.
  - A genuine bug in the server. This was the hardest and most interesting situation to deal with. The Retail API server consists of around 20,000 lines of Java code and a further 10,000 lines of code in the stored procedures. The generated code contains 120,000 lines, although it can be considered almost entirely error free. Tracking an issue through the server was quite time consuming. First, the call from SMART had to be found and verified to be functioning correctly. The request handler in the server and its related classes then had to be checked, as well as the server configuration. Many requests go through one or more pre-processors before reaching the handler so these also had to be checked. Finally the stored procedure was checked. In the few Retail API issues this is where the majority of problems came from. SQL code can be tricky to debug and errors can result from very subtle changes. Additionally, a typical stored procedure will make calls to other stored procedures and perform operations on a number of tables.
  If the bug was confirmed to be an issue with the Retail API server the standard process was to verify it with one of the full time developers and have them log the fix. Only on a few occasions did I commit a fix to the server, and these were very trivial bugs. I was not strictly involved in the Retail API server. It is one of the core InControl products, so the rigor surrounding it is of a far, far greater standard than SMART.

- Once the bug was fixed it was re-tested. This usually amounted to going through the steps to reproduce the issue and verifying the resulting behaviour was the expected behaviour, not the behaviour reported in the issue.

During the months spent maintaining SMART I worked with nearly every member of the QA team to some extent. SMART is a tool they use daily

so they were happy to help me in fixing issues as quickly as possible.

3.5.3 TOM

Along with the maintenance work I received several feature requests from a developer charged with InControl implementations, i.e. Managing the platform in the various environments (p.). He wanted to use SMART as window into InControl to view overall system health. This resulted in implementing several new requests within the Cronos server, which is ultimately released with a production system. These requests were largely different forms of statistics for different events. Some of these requests were later charted graphically in SMART to give a very quick overview of system health. SMART is not authorized for use within the higher environments but these charts are still valuable for testing.
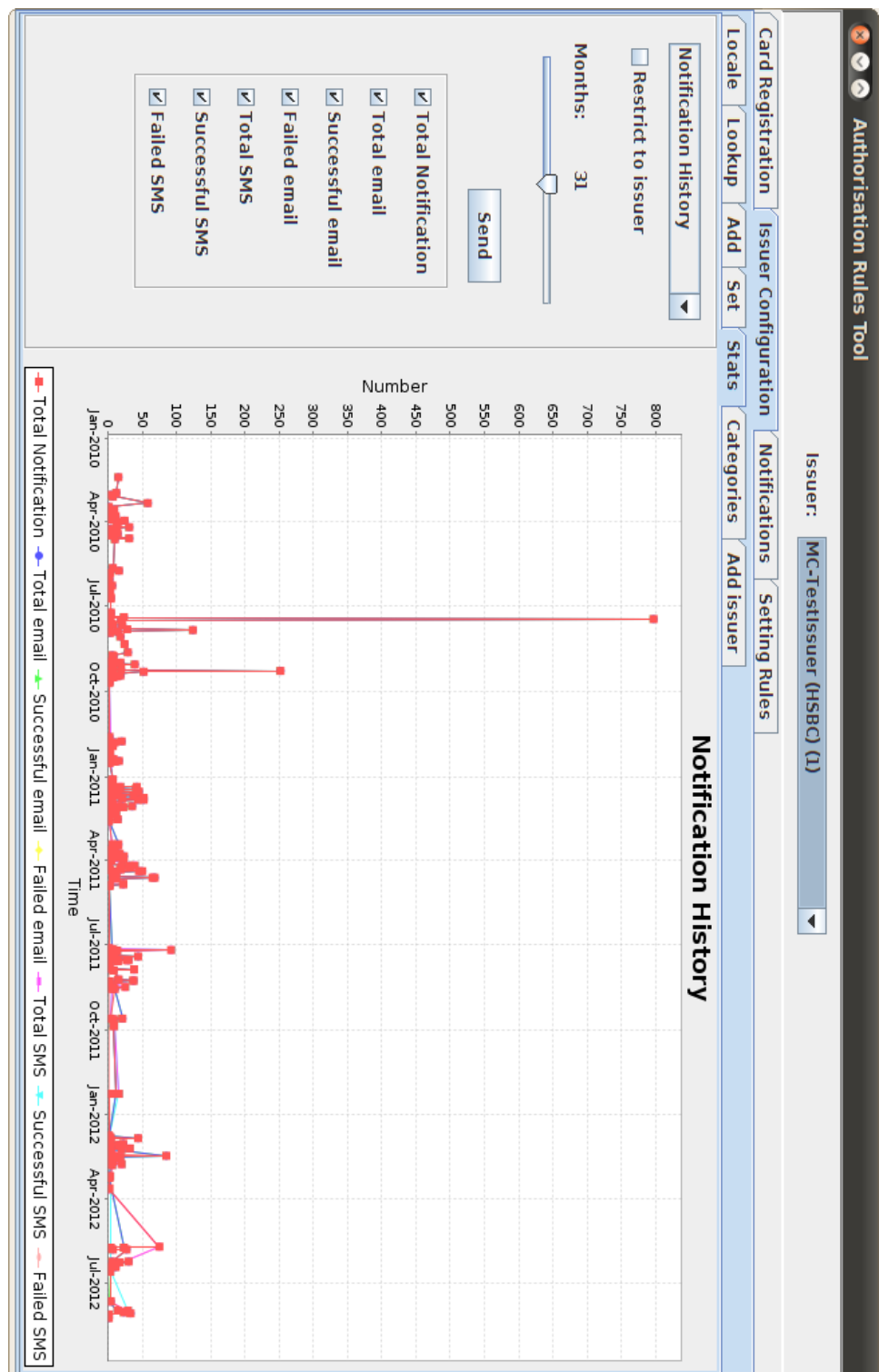
Rename this section

ENVIRONMENTS

Make sure these are explained

Spent a good bit of time on this stuff, needs more discussion

Customer XML also

A chart showing a history of notifications issued by the InControl system.

[?]

3.6 OIL

Orbiscom Interface Language is an in house Interface Definition Language. It is used to specify how various components communicate with each other. It can be used to specify basic data objects, request definitions and their behaviour. I used OIL extensively throughout the internship. There was some existing documentation that was helpful, but the bulk to the knowledge I acquired was from asking questions and reading the OIL grammar file. The OIL grammar file is a standard BackusNaur Form grammar and was quite easy to read because BNF grammars were covered extensively in the Discrete Mathematics and Compiler Design courses taken in second and third year respectively. There were some difficulties surrounding the implementation of the OIL compiler. Functionality described in the grammar was not implemented in the compiler. The most prominent being the inability to set default values for request parameters.

OIL is a fully fledged language with its own compiler. The compiler generates program code that can then be compiled. At the moment, Java and Flex can be generated using OIL. OIL is used throughout the InControl Platform. Virtually every component uses it in some respect. Even the components that do not use OIL directly, e.g. do not call the OIL generated code, use OIL indirectly as all of the XML messages are generated by OIL classes.

If a new request is needed it must be created using OIL. There are several steps involved, beginning with creating the database code to service the request. The stored database procedure is invoked by and OIL request handler. This handler must be written manually. I wrote a script to automate this process somewhat. The handler is invoked by an OIL generated handler (p. 61), which is configured to run when a specific request comes in. This request is an OIL generated type. The steps to add a new request to a component called Cronos follows

3.6.1 Create the stored procedure

A stored procedure is set of pre-compiled SQL statements stored inside a database. This example is just a simple select statement but any SQL is

36

valid. Stored procedures are saved to *release*/database/procedures/src/o_*modu* <mark>Does that look a</mark> e.g. ***r12q4/database/procedures/src/o_cronos.pls***. <u>Each stored proce-</u> <mark>bit much</mark> dure package consists of two sections, procedure prototype definitions and procedure bodies. This is the same as any programming language, the prototypes are function heads and the implementing code is contained within the body. A prototype definition:

```
PROCEDURE GetIssuerConfig (
        p_issuer_id IN VARCHAR2,
        r_issuer_config OUT Globals.ref_cursor);
```

Here the procedure takes a string and a reference cursor. A cursor acts like a pointer to the result set returned by the query. A ref cursor is similar, the key difference being that it is not bound to a single result set and can also be passed back up to the client, which is somewhat important. If no result set is returned, i.e., an insert, no ref cursor is needed.

The implementing code. This is just standard SQL code;

```
PROCEDURE GetIssuerConfig (
        p_issuer_id IN VARCHAR2,
        r_issuer_config OUT Globals.ref_cursor)
    IS
    BEGIN
        OPEN r_issuer_config FOR
            SELECT issuer_name, authentication_type, authorisation_type,
            session_timeout, authenticate_pan_password, allow_change_email_addr,
            allow_set_user_name, create_date, update_date
            FROM issuer_config
            WHERE issuer_id = p_issuer_id;
    END;
```

The stored procedure must then be loaded into the database. If there are any errors in the stored procedure file loading will fail and the entire file and all of the contained procedure will be rejected. It is generally a good idea to test the procedure works as intended before writing code to access it. It can be executed by specifying the package and procedure name. If the procedure requires a reference cursor it must also be declared

```
var blah refcursor
execute 'cronos12q4db.GetIssuerConfig('1', :blah);
```

3.6.2 OIL Interface

There are two files used to define the classes OIL generates, the OIL definition file and the OIL procedure definition file. The procedure definition file is nearly identical to the stored procedure definition file, so it is easier to write it first and then the definition file.

The OIL definition file, e.g. ***cronos.oil*** is used to define the request and any data types needed by that request. The file begins with an interface definition, this is the package where the generated code is stored. This is optionally followed by any imports needed. This allows OIL files to be split up very easily. Typically, that is by convention only, data types are defined first. The two built in data types are **String** and **Number**. Any user defined data types are collections of simple types and other user defined defined data types. Default values are supported. This is where the default values mentioned in the Retail API Server section come from . They are pageref specified in the OIL files and are thus hard coded into the generated Java code.

```
type IssuerConfig
   {
       String IssuerName;
       Number AuthenticationType;
       String AuthorisationType;
       Number SessionTimeout;
       String AuthenticatePanPassword;
       String AllowChangeEmailAddr;
       String AllowSetUserName = "N";
   }
```

The request definitions follow the type definitions

```
request GetIssuerConfig(
       required String IssuerId)
   {
   response
   {
       IssuerConfig IssuerConfig;
   }
   }
```

This is a very basic example of what OIL is capable of and will result in a basic data model Object and request definition. OIL supports annotations to support various features, such as comments to be included in the generated code and enforcing mandatory members in request parameters etc. Arrays are also supported. They can be restricted to containing elements of a specified type or multiple elements of different specified types. OIL definition files also have access to a Context Object. Values can be stored and retrieved from this Object when the generated code is running. This is particularly useful for large and complex operations. Values can be saved in the context to avoid having to pass them between requests.

The OIL procedure file maps the values returned by the stored procedure to the members of the OIL Object returned in the query response. It is an Object relational mapping similar to technologies like Hibernate.

```
procedure GetIssuerConfig[Cronos12q4DB.GetIssuerConfig] (
      String IssuerId => p_issuer_id)
  {
      output
      {
          IssuerConfig IssuerConfig <= r_issuer_config
          {
              String IssuerName <= issuer_name;
              Number AuthenticationType <= authentication_type;
              String AuthorisationType <= authorisation_type;
              Number SessionTimeout <= session_timeout;
              String AuthenticatePanPassword <= authenticate_pan_password;
              String AllowChangeEmailAddr <= allow_change_email_addr;
              String AllowSetUserName <= allow_set_user_name;
          }
      }
  }
```

One important thing to note here is that the type definitions declared in the OIL definition file must be imported into the OIL procedures file. Another more subtle issue is the presence of duplicate column names in a result set. The values will always be taken from the first column in the result set. The easiest way I found to fix this was to modify the stored procedure to

rename the conflicting columns in the result set. Oracle makes this very easy, the column name need only be appended with the desired name, i.e. *table_name.column_name "desired_name"*

The OIL compiler is then ran against the files which outputs several classes per request. The final step is to write a handler. The handler is executed by the server on receiving a particular request, mapped in the config files as outlined (p. 61)

3.7 OIL Handler

An appropriate handler class must be created in order to service the request. The handler must implement OilProcessor, specifically the OIL processor generated above . The purpose of the second, generated processor is to enforce parameter types. . For simple queries very little is needed. The request parameters are extracted from the incoming request object. The stored procedure is then called with these parameters. The output is put into the response object and sent back to the calling client.

Hierarchy tree here

Get input

```
package com.orbiscom.cronos.oil;

import java.sql.Connection;

import com.orbiscom.apollo.core.MessageProcessingException;
import com.orbiscom.atlas.core.ErrorMessage;
import com.orbiscom.atlas.core.MessageContext;
import com.orbiscom.atlas.xml.ApplicationHandler;
import com.orbiscom.cronos.oil.proc.GetIssuerListOutput;
import com.orbiscom.cronos.oil.proc.GetIssuerListProc;
import com.orbiscom.oil.db.ProcFactory;
import com.orbiscom.oil.db.StoredProcedureException;


public class GetIssuerListHandler
    implements GetIssuerListProcessor
{
    private final GetIssuerListProc iProc;
```

```
    public GetIssuerListHandler()
        throws StoredProcedureException
    {
        iProc = ProcFactory.getProc(GetIssuerListProc.class);
    }


    public void process(MessageContext ctx, GetIssuerListRequest request,
        GetIssuerListResponse response)
        throws MessageProcessingException
    {
        try
        {
            Connection conn =
                ctx.getIssuerDatabaseConnection(ApplicationHandler.APPLICATION_DB);

            GetIssuerListOutput out = iProc.call(ctx, conn);

            response.setIssuers(out.getIssuers());
        }
        catch (MessageProcessingException ex)
        {
            throw ex;
        }
        catch (Exception ex)
        {
            throw new ErrorMessage.GeneralError(ex);
        }
    }
}
```

Writing these handlers is particularly tedious. However, as they are usually very similar the process was automated with a simple script. The handler must then be added to the servers handler configuration file as shown on page 61

[10]

4 Web Admin Console

The second project worked on during the internship was an Issuer Administration web console. The requirement was to build a web application to allow a member of development, customer implementation support etc., to configure a new issuer on the InControl platform.

4.1 Overview

Adding a new issuer to the InControl platform is a complicated process. Currently, the database administrator manually inserts the new issuer each time. The majority of the data inserted is static but each new issuer still requires a large amount of custom data. Traditionally the setup is done by modifying an sql script. The issuer specific information is included along with the static data and ran against the relevant databases. This is a time consuming process for the database administrator. There are a number of common steps needed to setup a basic issuer and several optional steps, the use of which is dictated by the issuer and the products they wish to use. As InControl continues to grow more and more new issuers are being brought onto the InControl platform. Manually inserting each new issuer is no longer practical so a faster solution is needed.

4.2 Objectives

The objective of the project was to create a web application to allow issuer setup and maintenance. The application was to be used internally within MasterCard, starting in the lower environments and gradually progressing to use in a production setting. The main objective of the project was to allow issuers to be setup quickly and easy, giving the database administrators more time to work on areas other than tedious updates.

Initially it was decided to integrate the new application with the existing Client Support Services web application. CSS is an application released to customers to enable them to support end users, i.e. Card holders. This is a large application, about 180,000 lines of code. Some time was spent investigating the best way to integrate the new application with the existing CSS. Ultimately however, the requirements were changed. A new, standalone application was to be created. This was decided for several reasons. CSS is a

product released to issuers to support their users. The issuer setup application was to be used internally at MasterCard, there is no overlap in the user base at all so it makes little sense to integrate the two. The time needed to create a new set of separate functionality within the CSS was also a factor.

It was then decided to create a new application from scratch. There were two phases to this, a prototype to act as a proof of concept and help get the project approved, and a full project to be used by the various teams. I was to work with another intern on the prototype and eventually the full application. The application was to be built using the Spring Framework using JSP to render the web interface.

The prototype needed to demonstrate core functionality. A series of steps were to be supplied to the user to allow them to input the information need to create a new issuer. They could then review their work in some way. The full project was to expand on this with the inclusion of user rolls, environments and scope to expand the functionality.

The static data could be hidden from the user with only the core, issuer specific, information presented for the user to supply. This application would use the Cronos server to insert the data. Cronos contains calls to setup, configure and maintain an issuer. This is done using 60 separate requests, some of which are very substantial. The steps to setup an issuer were already done which simplified the project immensely.

4.3   Client Support Service

Initially the CSS web application was to be extended to include issuer setup functionality. It is split into a client and server, with the server making subsequent calls to the various other servers that make up the InControl platform. The client and server make up about 180,000 lines without counting the servers used to processes the requests and their underlying stored procedures. As such, CSS is quite a substantial project, and where to integrate a completely separate set of functionality was not obvious.

CSS is implemented in Java and user Java Server Faces. A few weeks were spent, in addition to working on SMART, researching and experimenting with JSF and CSS. Ultimately little functionality was achieved.

More? Find notes on JSF

4.4 Prototype

The prototype web application was written using the Spring Framework [11] and the existing Cronos requests. The prototype was to implement one specific use case, the creation of a new issuer for the IPC Fraud Control product. 29 Spring was chosen because several of the developers within InControl were familiar with the technology, it is approved for use within MasterCard and is one the most popular inversion of control containers for Java.

This project was very different from working on SMART. It gave me exposure to the full software development life-cycle in a scaled down manner. The functional specification above was finalized and a collection of documents needed to achieve this were gathered. At the core of this was the existing issuer database setup script and Cronos API requests, as well as some customer documentation used to explain the meaning for specific values, allowing them to be grouped logically in the application. Using the script and the Cronos source code, a series of 10 steps were drawn up. The first 6 were concerned with creating the basic issuer, with the remaining used to add a bin range and add the fraud control product.

4.4.1 Spring

Spring is an enterprise Java framework built around a single core principle, dependency injection. Dependency injection is a solution to the object coupling problem. If a number of objects make direct calls to each other they are said to be tightly coupled. Removing or changing any of them will cause the whole system to break. When implemented, this type of design quickly leads to large amounts of code that cannot be touched and extensive code duplication. The classic example of this is some sort of client-server application. A client fetches data through a file reader, the server. The client implementation can only fetch data in one way. If the server is changed to another source, a database or a socket, the client code must be re-factored.

Object oriented design mitigates this but does not completely solve the issue. Code written against interfaces allows functionality to be changed as long the interface contract remains the same. The problem is a reference to the implementing class must still be hardwired into the code. In the example above, the server would implement an interface. The interface would define

standard behaviour needed by the client. The implementation details are left up to the concrete server implementation. This concrete implementation still needs be instantiated somewhere in code. While this is a good solution to the problem, code still needs to be re-factored if the data-source changes.

Dependency injection, or Inversion of Control principle, aims to solve this problem by changing the way a program executes. Normally when a program starts some main function is called, the dependencies are created and execution proceeds. Inversion of Control does the reverse. All of the dependences and relationships are create by the IoC container and injected into the main program as properties. In the example above, the server implementation to be injected can be configured without having to refactor any code.

Spring is a very large framework with a wide range of functionality, but Inversion of Control is the principle on which it is built.

Until Spring version three, all configuration was done using XML. This allows for greater flexibility without having to recompile code but essentially splits development in two. In addition to writing code, various properties have to be created in separate XML files, slowing development. Spring version three supports annotations. The properties traditionally stored in XML files can now be written alongside Java code. This helps to keep everything together and makes code easier to read. However, if a property is changed, code refactoring is needed. The solution lies somewhere in the middle. Once the properties have been finalized, they can be moved put to XML files.

[11]

4.4.2 Design and Implementation

The web application was to use an MVC structure. The Spring framework has its own MVC web application framework, Spring MVC. The framework defines strategy interfaces for all of the responsibilities which must be handled by a requests-based framework. For each layer within the MVC structure, an interface is defined. The various components of the MVC implementation can then be swapped out at runtime using dependency injection. The basic structure is show below;
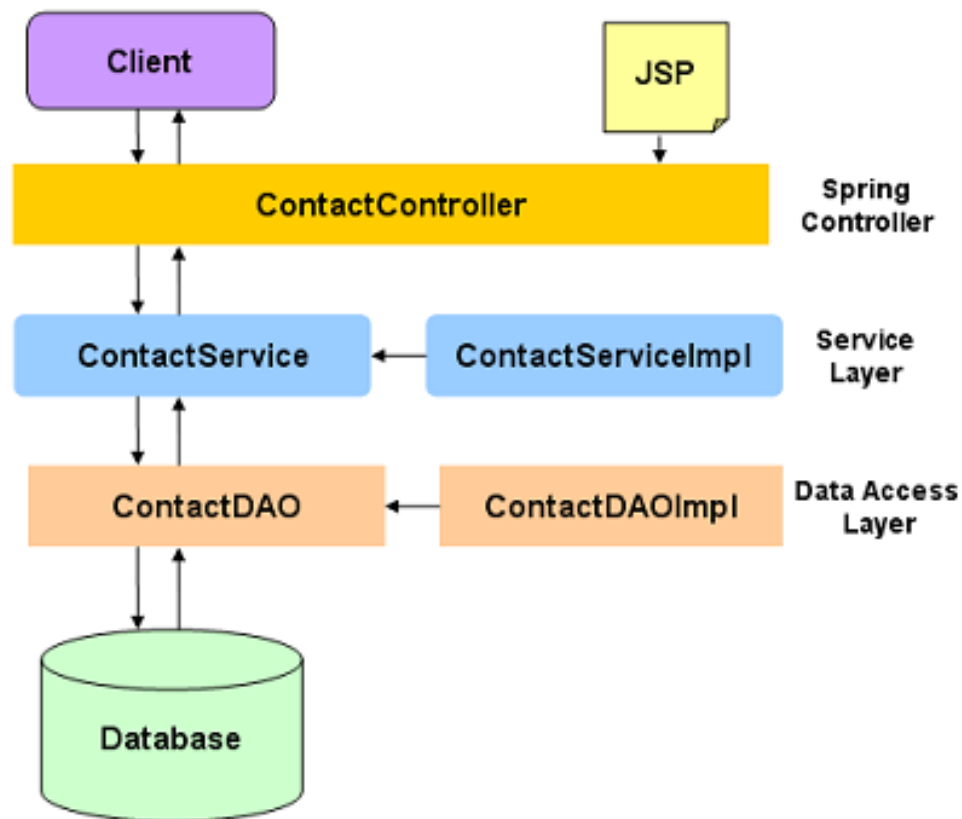
Overview of Spring MVC design [12]

- Model: Plain old Java objects. Spring supports object relational mapping using technologies like hibernate. If this is used, the model objects are declared as "@Entity". The class name must map to the underlying table in the database and its members must map to the columns. All member access is done through appropriate accessor methods. OIL was to be used for database access in this project, as the bulk of the logic was contained within the stored procedures. The OIL generated classes supplied the model objects. OIL generated objects are not traditional Java objects (p. 23) but were full inter-operable with Spring. Although the members of an OIL class are contained within a map, they can still be accessed using specific accessor methods.

  There were also some custom model object specific to the application. Many of the parameters in a typical Cronos request had sensible de-

46

fault values so it did not make sense to present these to the user. Thus, the model objects the user populated in the views were logical groupings of parameters used in several requests, i.e. The issuer name and email address are set in separate requests but it makes sense to have the user supply the information on the same screen. A model object was created to encapsulate this screen and its various members were then used to populate several requests to Cronos.

- DAO: The data access layer interacts with the application data. In the majority of cases this is an sql database but it can be any sort of data source. The DAO in this application sent requests to the Cronos server, which in turn interacted with the database. This became the most substantial component of the application as the bulk of the business logic had already been implemented in stored procedures. It was decided to continue to use Cronos to access these stored procedures for several reason. Cronos utilizes the Apollo framework which is very well understood within InControl. It was decided that given the collective amount of experience with Apollo, it would be far quicker to build the application using it. Whether it did or not is debatable, but if the product was to be developed fully it would be very hard to justify not using Apollo.

- Service Layer: The service layer is where Spring begins to depart slightly from a traditional MVC structure. A traditional MVC structure use the Model to represent the data being manipulated by the system, the View to present this data to the user and the Controller to do everything else. The service layer in Spring is used to house the business logic of an application, leaving the controller to parse user input, delegate the appropriate action and return the response, or in other words to control the application. The business logic in a large application, although not in this case, can be substantial, so it makes sense to segment it. In this application the business logic was contained within Cronos, so the service layers function was simply to give the controller and interface into the DAO.

- Controller: As above, the controller is responsible for parsing input and delegating the appropriate action. It passes this sanitized data to the service layer which then makes calls into the DAO. The flow of the

47

various steps needed to setup an issuer was defined in the controllers, as these classes are responsible for displaying the various views in addition pulling data from them.

- View: As per traditional MVC, the view is responsible for allowing the user to interact with the model. Spring supports a number of different technologies to create the view. Java Server Pages were used, they are the go to technology used to create dynamic pages with Java and are very well supported by Spring. JSP, along with Java Server Faces, is also employed by the majority of the web applications developed within InControl.

The Spring web application is then hosted inside a Java servlet container, Apache Tomcat was used in this instance.

The controllers and DAO implementation took the majority of implementation time. Each page in the application has its own controller. Some of these controllers are quite small but it made sense to segment them by page.

```
@Controller
@SessionAttributes
@RequestMapping("bin_setup")
public class BinRangeController
{

    private static final IssuerConsoleLogger LOGGER =
        IssuerConsoleLogger.getInstance(BinRangeController.class);

    @Autowired
    private IssuerService issuerService;

    @RequestMapping(method = RequestMethod.GET)
    public String showPage(Map model, HttpSession session)
    {
        LOGGER.debug("<<< /bin_setup.htm GET >>> called");

        model.put("binrange", new InitialBinRange());
```

```
        return "bin_setup";
    }


    @RequestMapping(method = RequestMethod.POST)
    public String processForm(@Valid InitialBinRange binrange,
        BindingResult result, HttpSession session)
    {
        LOGGER.debug("<<< /bin_setup.htm POST >>> called");


        Issuer issuer = (Issuer) session.getAttribute("issuer");
        issuerService.setupIssuer(issuer, binrange);
        return "redirect:issuer_list.htm";
    }
}
```

The above code snippet gives an example of a variety of Spring function-
ality.

@Controller: Spring does not enforce the MVC model structurally, i.e. y
package name. Classes are configured to be part of the various layers. This
can be done in an XML file, or in this example with an annotation.

@SessionAttributes: The lifetime of an object can also be configured. In
this example the object expires after the user disconnects. Object lifetime
can be set anywhere from a single page display to the entire application
lifetime.

@RequestMapping: Requests can be mapped to classes or individual
methods. This class is configured to process $issuer_{d}etailsrequests.Therequesttypeisspecifiedintherec$

The logger is an example of Aspect Oriented Programming. Aspect Ori-
ented Programming complements Object Oriented programming by allowing
the separation of cross cutting concerns. Logging is the classic example of
a cross cutting concern because the logging strategy effects every part of
the system. By encapsulating logging changes can be made to its behaviour
without having to impact the rest of the system. [13]

@Autowired: This annotation tells Spring to inject an object at run
time. In this application, IssuerService is an interface. The implementing
class is loaded automatically and injected by the IoC container. As stated

above, this is a very powerful technique but can make the code a little confusing. It is not immediately obvious what implementing class is being used. Here there is only one implementing class so the choice is easy. If there are multiple implementations the one to be used must be specified. This indirection can make projects harder to read and debug.

Different methods are invoked depending of the type of incoming request. Before the page can be shown to the user, the model must be populated for the view to interact with. A new InitialBinRange object is inserted into the model and "bin-setup" is returned. This particular Spring application is configured to display the page with the same name as the string returned by the controller. The model and user session need only be declared as parameters and Spring automatically injects them.

Post requests are handed in a similar method. The view has populated the model object supplied by the Get request. This object is supplied with the post request. As this is the last step it must be committed to the database. The model object populated in a previous view are extracted from the session and passed down to the service layer.

@Valid: The @Valid annotation in the processForm method declares that the object passed in should first be validated. This can be done by writing a custom validation object or by simply annotating the various members of the model object with constraints.

The model objects are passed from the service layer into the DAO layer in the same manner as displayed above. The DAO implementation is resolved at runtime. The members of the model objects are then used to populate Cronos requests. The DAO implementation is quite long and tedious. It simple populates various Cronos request objects and sends them to the Cronos server. The order of these requests turned out to be quite important. Certain steps in the issuer setup must be committed in a certain order. This sequence was inferred from the DBAs scripts. Cronos calls with appropriate functionality were then found and sent in the proper sequence.

There was also a small bug in Apollo. Apollo allows multiple requests to be sent at once, wrapped inside a "requestor". If any one of these requests fails, none of the requests in the requestor are committed. In theory each requestor should have its own scope, so each request is exposed to the changes made in previous requests. This was not the case, so requestors had to be broken up to allow requests to be committed properly.

After the setup steps had been completed the user was forwarded to a screen to allow them to review their work. This was a simple listing of issuers in the system. The bin ranges associated with an issuer could then be viewed. This was a small feature simply to show that the application could also be used to view system information.

4.4.3 Completion

After the prototype had been completed it was shown as part of a pitch to secure funding for a fully developed version. An easy way to set up new issuer on the InControl Platform was something the team had wanted for a number of years. An independent report recently conducted by Computer Science Corporation had highlighted the need for automation of the process. The idea behind the prototype was to show the business leaders in charge of InControl what such an application might look like and that it would not require a significant development effort. The project was approved and given funding for the first quarter of 2013.

4.5 Further development

After the project had been approved work began on a more robust version. Unfortunately, a formal specification was not drawn up during the remainder of the internship. I was to target a single use case but build the application to allow for extension. Additional set up use-cases, information retrieval and user roles and levels of access were to be added in later.

These requirements were very broad and difficult to account for. After researching user roles and access privileges in Spring I was satisfied that they could be integrated as an aspect (p.) later in development. This left me to concentrate on building the application to allow for later extension. In order for the application to be used and supported in the future it had to be as maintenance free as possible. Changes to the issuer setup process should not require code refactoring. After working with SMART it was understood that such an approach was feasible. The principle of dependency injection employees a similar idea to the dynamic loading of requests in SMART, although it is **substantially** more complicated.

A few different approaches were considered before a design was finalized. They are discussed in Appendix B: Issuer Control design on page .

4.5.1 Parser

The desire was to build a dynamic model that could be configured with XML. The sequence of steps would be specified in an XML file, populated with data where appropriate. A similar project had recently been completed within InControl, the Cronos Batch Utility. This was an extension of the Cronos server that allowed mass registration of issuers. The sequence of requests were specified in an XML file and the data to populate them was contained in flat files. The batch process "digests" the data in these flat files and sends them to the server. However, the Cronos Batch Utility differed in two key areas to the intended functionality of the Issuer Setup application.

- The Cronos Batch Utility populated the XML with data from the flat files and sent the resulting XML directly. Request objects were not created.

- The XML messages were not sent over a network, rather they were passed directly to the handlers. This meant the Cronos server had to be running on the machine, not remotely. While the Apollo framework does support sending XML directly, there is substantially more effort involved as opposed to just sending request objects via a requestor.

The Issuer Setup web application had to be able to send requests remotely, so a different approach was required. The templates from the Cronos Batch Utility were used as base, the idea being that these would be maintained and thus the setup process used in the web application would be maintained. Some slight changes were made. Each setup process was contained within a "wizard", and each "wizard" contained a series of "steps". The requests were contained within these "steps". The requests were separated logically into steps, the idea being that the interface could be generated off the XML at a later date. A summarized version of the resulting XML is shown below:

```
<Wizards>

    <ComplexTypes>
        <Type Name="Property" Path="com.orbiscom.atlas.oil"/>
    </ComplexTypes>

    <Wizard Name="Add Issuer">
```

```
        <Step Name="Add Issuer">
        <AddIssuerRequest IssuerId="{IssuerId}"
            IssuerName="{IssuerName}" />


        <SetIssuerPropertyRequest IssuerId="{IssuerId}">
            <Property Name="SecurityManager" Value="{Cipher}" />
        </SetIssuerPropertyRequest>
    </Step>


    <Step Name="Add groups">
        <AddPhoenixIssuerGroupRequest GroupTag="{GroupName}" />
        <AddPhoenixIssuerGroupIssuerIdRequest
            IssuerId="{IssuerId}" GroupId="0" />
    </Step>


    </Wizard>


</Wizards>
```

A parser was than constructed to generate OIL request objects. The resulting parser was relatively complicated, taking several factors into account. It is more complex than the requests creator implemented in SMART, the biggest differences are outlined below.

OIL requests can contain complex and simple types. The parser must be made aware of any complex types so the corresponding classes can be instantiated. After this is done the requests are parsed. First the request name is parsed and the corresponding request object is created using the same approach as SMART . The request objects attributes are then compared against the attributes specified in the XML. If they match, the XML attribute is parsed and stored in the request object. There are a number of possible events relating to this. Pageref this

The attributes are not matched by member name, rather the class name set in the attributes meta-data map is compared to the attribute name in the XML. This is because the member name is defined by however writes the OIL definition. Generally the member is named after its type but this is not always the case. Whether or not a member is in fact a list of members

is determined again from the meta-data map by examining the container attribute of the meta-data. A code snippet generated by the OIL compiler is shown below, demonstrating the difference between a single attribute and a list of attributes.

```
cMetaData.put("IssuerId", tmpMetaData = new MetaData("IssuerId",
    String.class, null, null, null, null));
cMetaData.put("Properties", tmpMetaData = new MetaData("Properties",
    Property.class, List.class, null, null, null));
```

After the nature of the attribute is determined, it is populated. There are two possibilities; if a value is specified, as with "GroupId" above, the value is parsed and stored inside the OIL object. If the value specified is of the form {

$ident$}$theidentifierisstoredasaconstraintwithintheOILobject.Thisissovaluescanbeusedtopopulatemeta-$

$dataforeveryOILobjectcontainsafieldforsuchanidentifier.$

The result is a relatively elaborate data structure containing a number of request objects:

```
Add Issuer
    Add Issuer
        {_BeanType=AddIssuerRequest, IssuerId=null, IssuerName=null}
        {_BeanType=SetIssuerPropertyRequest, IssuerId=null,
            Property={_BeanType=Property, Name=SecurityManager, Value=null}}
    Add groups
        {_BeanType=AddPhoenixIssuerGroupRequest, GroupTag=null}
        {_BeanType=AddPhoenixIssuerGroupIssuerIdRequest, IssuerId=null, GroupId=0}
```

The data structure is:

```
LinkedHashMap <String,
    LinkedHashMap <String, ArrayList
        <Request>
    >
>
```

Linked hash-maps are used so the ordering of steps is preserved. The first hash-map contains all the different setup "wizards". Each wizard is in turn a hash-map containing a series of "steps". Each step contains an array list of requests.

4.5.2 Error Handling

Writing the XML file is quite an error prone process. There is no tolerance for errors naming objects as these names are used to create objects. Exception handling within the parser had to be robust and return specific errors to allow mistakes to be easily diagnosed. After some reading up on best practices I am confident that the parser is relatively robust. [**?**]

5 Conclusion

Technical Conclusion. Goals. BCS goals.

6 Appendix A: Existing System

The InControl platform is very complex. It was not developed as a single service but rather as a generic server framework to host various payment services. The platform is component based and hosts a number of generic services and product specific services. The role of InControl has changed over time, with the emphasis now placed on the VCN and Rule services as MasterCard have no interest in hosting their competitors products.

6.1 Components Overview

The following is a brief overview of the components within the InControl system. At present, the system consists of 21 separate components containing 768,732[14] lines of code between them. This is just the core InControl product, there are many other auxiliary projects in the subversion repository. A detailed look at each component is carried out in subsequent sections.

The InControl platform is comprised of seven major types of component.

6.1.1 Client applications

Client applications include any client interacting with an InControl server. These include the InControl Virtual Card Client and other third part clients, such as Verified by Visa clients. There are two versions of the Virtual Card Application (originally called the O-Card application). The thin and slim client applications both communicate all requests to the VCNs server via the InControl web server environment.

6.1.2 Web Server Environment

The web server environment acts as a gateway to the InControl Dynamic servers for the external client applications. The InControl Payment platform typically requires that a web-server is available. The web server environment may include web servers or application servers, or a mix of both. The main functions of the web server environment are as follows:

- Provide the content files for InControl client applications, e.g. configuration files, web assets. The client content can be hosted on any plain old web-server.

- Host the InControl servlets. The main functions of the servlets are to preform message format conversion to enable InControl servers to process messages from the client applications and the reverse. This is to allow deployment of the InControl Platform into different environments with minimal code impact, a clear benefit of a modular design. Only one component has to be switched out when the platform is moved to a different network. An application server providing the appropriate servlet environment is required. The servlets support a number of servers including WebSphere and Tomcat, allowing it to be easily integrated with existing infrastructure. If there is no existing infrastructure it can be run inside the InControl Apollo framework.

- Host InControl sessions and authentication manager. The session and authentication manager allows for integration with an existing customer authentication API. The InControl Platform can also provide servlets for session management and authentication.

6.1.3 InControl Servers

The InControl servers provide the processing core for the InControl payment platform. A generic server framework is provided for hosting the generic servers, e.g. maintenance and the payment products services, e.g. Virtual Card Number services. The generic InControl server architecture is based on a dynamic server framework. The InControl servers reside on BankNet, while the InControl Direct servers typically reside on a customers, e.g. A Bank, internal network. The services provided by the platform can include Online Services, Batch Services or Scheduled Services.

- Online Services: The platform can be used to provide a number of online services including registration, session and authentication, authorization services and client support services.

- Batch Services: Batch services read a file as input and apply changes to the server data based on the contents of each record. They can also be run as command line utilities. The platform provides card settlement, registration and maintenance as batch services.

- Scheduled Services: These are programs that can be configured to run periodically in order to preform housekeeping tasks. They can be configured to run inside one of the installed InControl services or to run as standalone processes.

6.1.4   InControl Database

At the core of the system is a relational database shared by all components. The system uses an Oracle database because of its widespread usage within industry, support infrastructure and status within MasterCard. Technologies used by products must be approved and Oracle is the recommended data base used by MasterCard technologies. The database provides both data storage and configuration information to every component within the system.

6.1.5   Customer Service System

The customer service system is web application made available to the clients customer service representatives for dealing with customer service issues in relation to the InControl Platform. The Customer Service System consists of two components. A web server used by the client and a backend server used to query the InControl systems database.

6.1.6   APIs

APIs are available to either access functionality of the system, ex. the registration API, or to provide functionality to the system, ex. the user authentication and session management for the platform.

6.1.7 Administration Programs

The InControl Platform comes with a number of administration programs that are used to preform or initiate housekeeping tasks. These programs can be run from the command line. ex. the archiving process.

6.2 Web Server Environment

The web server environment provides the link between the external client applications and the InControl Servers, the core of the InControl Platform. Typically web servers and, optionally, application servers are used to provide the appropriate web server environment.

The web server environment typically hosts the Orbiscom servlets. The Session and Authentication Manager can also be hosted within this environment but is usually looked after by the customer in the case of InControl Direct, or by MasterCard in the case of InControl.

6.2.1 Content Hosting

A web server is required to provide the content files for Orbiscom client applications. These files are served dynamically each time the client connects to the URL. Any plain old webserver can be used.

6.2.2 InControl Servlets

The Web Server / Application Server typically host the InControl servlets. Athena is the servlet system used to support the InControl Platform.

6.3 InControl Server

This section describes the InControl server architecture, incorporating the generic server framework. The InControl server architecture is described in the following areas:

- Generic Server Framework

- InControl Services Overview

- Configuration and Installation

- Communications Security

- Operational Requirements

6.3.1 Generic Server Framework

The Generic Server Framework is based on a dynamic server framework. Apollo and Atlas provide the generic server framework.

- Apollo provides the generic, dynamic, extensible framework for clients and servers.

- Atlas provides the InControl HTTP and XML dispatchers.

These two components are at the core of the InControl platform. Everything is built on top of them. Together they create a full generic server framework with capacity to manage authentication, thread synchronization, component configuration, database connection and connector pooling, security and session management.

The InControl Servers are UNIX based and implemented entirely in Java. They are designed to be platform independent and will run on any platform that supports an appropriate release of the JVM, at present InControl targets JVM 1.6.

InControl servers can be configured to listen on one or more communications interfaces for incoming connections. Each listener can be configured to process one communications protocol. The platform supports HTTPS, HTTP, raw TCP/IP, MQ Series and X.25.

Each listener can be configured to pass on requests to different types of content handlers. The most common form of content handler is the InControl XML content handler, which process messages that are in a format specified in the InControl Message Document.

Other message handlers can process various authorization card scheme formats, for example Visa and EuroPay handlers.

InControl Servers can be configured to dynamically update various configuration items upon receipt of a signal from an administration utility.

The following is a list describing the behaviour of an InControl server.

- Connection: At boot-up a number of dynamic servers are created to process jobs. Each Server creates a number of connectors. These connector classes allow the server to either listen for requests or connect

to external services. The classes, which implement the actual connector protocol, e.g. TCP/IP, are specified in the Dynamic Server configuration. This allows the connectors to be swapped out easily, The connector specification contains the name of the implementing class and a port number. There may also be a number of optional parameters defined depending on the protocol. As the Connectors are implementations of a Connector interface, there are no mandatory parameters. As such, all of the parameters are in fact optional, but omissions will result in exceptions being thrown. These are caught and a default value is returned if specified. This would appear to be a bad way of returning a default value as exception handling is being used for control flow.

Expand a bit

```
<Connector
    Class="com.orbiscom.apollo.net.TCPServerConnectorFactory"
    Port="12400"
    TcpNoDelay="true"
    ReadTimeout="3"/>
```

- Transport: A protocol handler specifies the Network Transport Protocol that is implemented on the connectors, e.g. HTTP. The protocol handler can specify many content handlers. The handler specification contains the name of the implementing class and a number of handlers. The handlers are indexed by a request type. In the case of HTTP the request type is the path. The handler specification body can be as simple as the name of the implementing class or much more complex. An example is provided below. The first handler specified is an XML Dispatcher. This content handler converts the incoming HTTP content to an InControl XML request for processing. The other elements of note are the implementing class and the HandlerSet. The handler set contains links to more configuration files. These files contain the mapping from requests to the handlers to service them. Different versions of the InControl protocol may require different handler sets. Great care has been taken to ensure that each new release is backward compatible with the previous version, but if needed separate handlers can be used.

```
<ProtocolHandler
```

```
Class="com.orbiscom.apollo.net.HTTPProtocolHandlerFactory"
MonitorHandlerTimes="true">

<Handler
RequestType="/pulse">

<Task
Class="com.orbiscom.atlas.xml.Dispatcher"
RequireAuthentication="true"
LogAuthenticationErrors="true"
UseEncryption="true" >

<DocumentBuilderFactory>
<Property Name="NamespaceAware" Value="true" />
</DocumentBuilderFactory>

<OilContext
MessageTimeZone="UTC">
</OilContext>

<MessageContext>
<Variable Name="DataExtract.Source" Value="RetailXMLAPI" />
</MessageContext>

<HandlerSet>
                <VersionedHandlerSet Version="12.4" xlink:href = "metahandlers"
                <VersionedHandlerSet Version="12.4" xlink:href = "pulsehandlers
                <VersionedHandlerSet Version="12.3" xlink:href = "pulse12q3hand
                <VersionedHandlerSet Version="12.4" xlink:href = "flexhandlers"
        <VersionedHandlerSet Version="12.4" xlink:href = "ipchandlers" />
                <VersionedHandlerSet Version="12.4" xlink:href = "evcnhandlers"
                <VersionedHandlerSet Version="12.4" xlink:href = "sbchandlers"
            </HandlerSet>

        </Task>
    </Handler>
```

```
<Handler
    RequestType="/schema">
    <Task
        Class="com.orbiscom.atlas.util.SchemaServlet">
    </Task>
</Handler>

</ProtocolHandler>
```

An entry in a handler set follows. The example provided is a particularly complicated specification to illustrate the widest possible range of functionality. A simple hander consists of a request type name and handler class. In the example entry a constraint and various preprocessors are specified. The constraint classes are built into the server framework and check certain conditions are met before further processing is done. In this case, the supplied PAN (Personal account number or the number embossed on a credit card) is checked to make sure it is the correct length. The pre-processor takes the supplied PAN and maps it to a unique ID, which is then used to index the account in subsistent operations. Each pre-processor falls through to the next so the order is important.

```
<Handler
    RequestType="GetVCNListRequest">
    <Task
        Class="com.orbiscom.pulse.oil.GetVCNListXMLHandler">

        <Constraints>
            <Attribute Name="Pan" Constraint="PanConstraint" />
        </Constraints>

        <OilProcessors>
            <OilProcessor
                Class="com.orbiscom.pulse.oil.mapper.PanMapperProcessor" />
            <OilProcessor
```

```
                    Class="com.orbiscom.pulse.oil.obo.CheckThirdPartyAccessProc
            <OilProcessor
                Class="com.orbiscom.pulse.oil.GetVCNListHandler" />
        </OilProcessors>
    </Task>

    <Task
        Class="com.orbiscom.atlas.mastercard.audit.AuditLogger"
            EventType="com.mastercard.common.jal.events.EventType.
            ACCESS_CARDHOLDER_DATA"
        AuditMessage="Get VCN List" >
    </Task>

    <ErrorTask
        Class="com.orbiscom.atlas.mastercard.audit.AuditLogger"
            EventType="com.mastercard.common.jal.events.EventType.
            ACCESS_CARDHOLDER_DATA"
        AuditMessage="A request to Get VCN List failed" >
    </ErrorTask>
</Handler>
```

- Application: The XML Dispatcher determines the issuer the request
  is intended for and starts a transaction in the database, configured for
  that issuer. The XML Dispatcher iterates through all the elements
  in the request message. For each node, the dispatcher determines the
  relevant application handler and runs it. The handlers are run in the
  order specified by the request. The response XML document is created
  and each application handler appends its response to this document.
  The platform is state-full. The dispatcher maintains the database
  transactions per request by creating a massage context. This context
  object is passed to each application handler in turn. The dispatcher
  only commits the changes when every handler working on the request
  returns successfully.

  If an exception occurs while iterating through the nodes the dispatcher
  roles back the database transactions. Further processing stops and

failure message is returned. The response will specify a return code and a message to indicate the nature of the failure. At present, the core system, e.g. Atlas and Apollo, have over 100 individual error codes defined. Each component may also define its own failure codes, for example, the Retail API Server has 75. These codes are of the format NNXXX, where NN is the component name and XXX the error code. The code is always followed by a textual description of the failure. ex, AT001 General Error. Here AT is the component name, Atlas and 001 is the error code. Often it is fairly trivial to quickly trace the cause of an error because of this. However, error codes are generally defined for well understood and anticipated error. While this is useful for diagnosing error in a production system it is not very helpful for identifying the cause of new errors introduced during a development cycle.

An additional task is also run in this instance. The event is written to an audit file for logging purposes. The audit file is a separate database supplied by MasterCard called the data-warehouse. The idea is to offload logging from the main system.

After the request has been processed the XML dispatcher completes the audit message by writing a trailer recorded. This record indicates whether or not the transaction was successful. I was curious about the reasoning behind state-full request processing. Once again this can be attributed to the difference between taught best practices and real world scale. The InControl platform, as evidenced, is not trivial. Many of its operations and use cases are complex and it makes sense to preserve state in many instances. This keeps the size of any one operation manageable and allows reuse. While changes in a specific request may ,and have, had a negative effect on other requests it is still a much better approach than trying to make every request stateless, and thus huge.

- Client Mode: These are servers than can be configured to initiate connection to another server rather than listen on a port. The poll this connection for requests. They can use same protocol and same content handlers as any other InControl Server. An example would be the InControl Registration server when communicating over MQ

Series.

6.3.2 Configuration

All communication from applications external to the server is configurable. InControl platform components are highly configurable and the configuration files have grown substantially, accounting for just over 7% of the code base. Every component has its own set of configuration files. There is also a set of common configuration files. The system consists of five general types of configuration file. Each file is used to configure a different aspect of the system. Any file may link to several other files, but the following can be viewed as the five root configuration files.

- Top Level configuration: This is the first configuration file read on start-up. It is passed as a command line argument and is responsible for specifying the various configuration items the server has access to. The following configuration files are contained as nodes.

- Logging: The log4j configuration file contains the logging configuration for the system. Log4j is a third party component used by the InControl platform. It employees configurable logging levels for different aspect of the system. The current debug level can be specified in the configuration file. During development full logging will be enabled, but in a production environment INFO or ERROR level are used. This ensures only relevant information is recorded. It also keeps writes to log files down. IO operations are generally very costly, so being able to easily change the logging level is very beneficial. Additionally, a typical log4j configuration file will contain a number of loggers. These are specific to classes or packages in the classpath of the running Java application. Loggers contain a logging level at a minimum.

```
<logger name = "com.orbiscom.atlas.range">
    <level value = "info"/>
</logger>
```

Appenders are also specified. Crucially, these elements define where a log is written and the implementing class, as well as information such as the character encoding used and time-stamp patterns.

```
<appender name = "Debug" class =
      "org.apache.log4j.DailyRollingFileAppender">
   <param name = "File" value = "log/debug.log"/>
   <param name = "Append" value = "false"/>
   <param name = "Encoding" value = "UTF-8" />
   <layout class = "org.apache.log4j.PatternLayout">
      <param name = "ConversionPattern"
          value = "%d{dd,MM HH:mm:ss:SSS} [%t] %-5p %c{2} - %m\n"/>
   </layout>
</appender>
```

- Database configuration: The database configuration specifies all the databases that an InControl Server accesses. It contains the database URL, the username, password and the database driver class. This information resides in a ConnectionPool node.

- Dynamic Server configuration: The Dynamic Server configuration section defines the operation of the server, the actions it will perform, supported message types and the supported protocols. The Protocal-Handler element shown above is from a dynamic server configuration file. It also contains links to the handlers used by the servers, again illustrated above.

- Classloader configuration: This is used to make the Java classloader aware of certain classes.

6.3.3 Communications security

Inter communications between the components and also between a component and an external server may be secured. SSL can be used, in such cases certificates must also be configured. Signed data can also be used, in which all communication is signed with a pre-shared key. The Java Cryptography Extension framework is used to preform data signing. JCE was used as it is standard cryptography framework for the Java SDK. The system signs the data with hash-based message authentication codes. The keys are contained within jar files as this is the easiest and safest way to store them.

6.4 Session & Authentication Service

The Session & Authentication Service is responsible for verifying the cardholders credentials, thereby ensuring that they are valid users of the InControl system. A flexible authentication service is provided, where Issuers have the option of implementing any of the following authentication sessions.

6.4.1 InControl Internal Authentication

InControl is capable of managing complete authentication. The user credentials are stored in the inside the InControl database. Verification rules for User credentials can be configured in the database. These rules are validated when creating the user credentials, e.g. Minimum number of numeric characters required, max invalid logins etc. The number of possible rules has grown considerably, from three or four to over thirty five. These values for these requirements have stayed fairly static for a number of years. A default set are usually set for each new issuer, and changes made thereafter if required.

6.4.2 External Authentication

The InControl Session & Authentication Manager (SAM) may be hosted in the web server environment or on the InControl Server platform. The SAM acts as a wrapper for an issuers' authentication and sessions APIs. InControl servlets authenticate VCN requests through the SMA API rather than linking directly through the issuers API. This is again part of the modular design employed throughout the project. The various servlets do not require customization for each customer, only the customer specific SAM. This modularity is only made use of in InControl Direct. After the InControl platform had been updated to operate with BankNet, session and authentication were no longer an issue as BankNet takes care of them. The SAM classes are specified in a servers properties file.

6.4.3 Session Management

To avoid the requirement for the client to send their complete user credentials in every message the InControl Platform supports session management. When a clients credentials are successfully validated a unique session ID is generated using a cryptographically secure hash algorithm and sent to

the client. This token and the unique user ID are stored in the InControl database in a temporary table. On future requests the client need only send their token and the platform will accept the message. The timeout for a session can be configured in the database.

6.5 Online Registration & Maintenance Service

The Registration and Maintenance of cardholder and card information in the InControl database is required to allow cardholders to avail of the payment services provided by the InControl Platform. This service is provided by the Retail API Server and the implementing component is Pulse. The example config files above are taken from the Retail API Server.

6.5.1 Functionality

The Retail API Server receives and processes XML formatted requests. The server listens for incoming valid XML requests from an external system on a configurable port. A defined attribute in the XML request will determine the action to be taken. The service can also supply default values for any non-specified data. These values are specified in the implementation and cannot be configured. Once the request has been processed successfully, a response will be sent back to the source informing them of the success or failure of the request.

The Retail API server encompasses the majority of the InControl Platform functionality and is used by several other components, in addition to being one of the customer hooks into the InControl Platform.

6.5.2 Message Formats

All messages sent to and processed by the Retail API Service are XML documents transported by HTTP requests. The HTTP requests are fairly ordinary, containing information such as host, port, service etc.

The HTTP message body contains the information to be processed by the Retail API Server. The root node is known as the requestor. A requestor can contain N individual requests. If any one request fails, the overall requestor fails. The response is structured similarly, the response root node contains the response value, e.g. Success or Failure, and N response elements containing the response data specific to each request.

An example XML request is shown, populated with mock data.

```
[<?xml version="1.0" encoding="UTF-8"?>
<OrbiscomRequest IssuerId="1" Version="12.4">
  <AddRcnBinRangeRequest IssuerId="99993">
    <Range CardType="83" CpnType="DF" EndRange="1234569999999999"
        LanguageId="1" Prefix="123456" StartRange="1234560000000000" Status="A"/>
  </AddRcnBinRangeRequest>
  <SetBinRangePropertyRequest IssuerId="99993">
    <Property Name="ICA" Value="009661"/>
  </SetBinRangePropertyRequest>
  <SetBinRangePropertyRequest IssuerId="99993">
    <Property Name="AICABRC" Value="5"/>
  </SetBinRangePropertyRequest>
  <SetBinRangePropertyRequest IssuerId="99993">
    <Property Name="IPCAMSRequired" Value="0"/>
  </SetBinRangePropertyRequest>
  <SetBinRangePropertyRequest IssuerId="99993">
    <Property Name="DefaultIPCProfile" Value="DEFAULT"/>
  </SetBinRangePropertyRequest>
  <SetBinRangePropertyRequest IssuerId="99993">
    <Property Name="IPCPorductType" Value="FC"/>
  </SetBinRangePropertyRequest>
</OrbiscomRequest>
```

6.6   Batch Registration and Maintenance Service

The Batch Registration Service is responsible for creating multiple cardholders simultaneously. The data is provided in a fixed width file format. The service can be run while the rest of the platform is running with minimal impact on performance.

6.7   Virtual Card Generation

The Virtual Card Generation Service is a scheduled service responsible for maintaining the queues of generated card numbers. VCNs are generated according to the card scheme rules to which they belong. The VCN typically contain the following:

- BIN; The 5 digit BIN code is used to identify the bank who issued the card.

- Randomly generated sequence of digits. This is the number that identifies a unique card and associated account.

- An optional Luhn check digit and any additional card reference digits

6.8 Virtual Card Settlement and Authorization

The purpose of the settlement service is to map VCN details to real card details so the transaction can be identified. The authorization preforms a similar function. It pre-process incoming authorization requests before they are processed by the issuers main authorization system. It also processes the response sent from the issuer.

[15]

7 Appendix B: Issuer Control design

The idea is was build the system in a generic way such that the interface can be described in XML and built at runtime. This means that if (when) the Cronos API / issuer setup changes the code does not need to be modified. If the system had to be modified every-time something in Cronos changed or the steps involved in issuer setup changed it would quickly become unmaintained and unused. A possible solution is outlines below but the basic functional requirements are as follows.

- Request specification; The request must be specified using some kind of markup that will allow the following information for each parameter

  - Type

  - Required, optional or hidden

  - Default values

  - UI component (ex, ComboBox, TextField, List etc) Might be best to imply this from type and default values but still be able to specify it if needed.

- Request flow specification; The work flow for the wizard should consist of a number of steps, each containing requests.

- Submit requests; Again, need to be done generically. Easiest way is probably dynamic object invocation at runtime.

8 Possible implementation

8.1 Request specification using XSD

Specify requests in XML. The requests specification could be done by using the existing XSD for Cronos and annotating it with default values. Information such as type (already in Oil to some degree) and required / optional use is already contained within the schema and can be easily changed. Important to note that the requests are not built from the XSD, they are built with oil objects as normal, the XSDs are use to generate the UI and map the various fields to the correct parameters in the corresponding Oil object.

```
<xsd:complexType mixed="true" name="GetVCNServiceStatusResponseType">
    <xsd:attribute name="Enabled" type="xsd:boolean" use="optional"/>
    <xsd:attribute name="RequestId" type="xsd:string" use="optional"/>
    <xsd:attribute name="Pan" type="PanConstraintType" use="optional"/>
    <xsd:attribute name="CpnId" type="xsd:decimal" use="optional"/>
</xsd:complexType>
```

An XSD allows constraints to be specified which could be used to populate combo-boxs / lists, or used as default values in text fields (in this case the constraint would be ignored and just used to specify a default value). This does add an additional layer to the parser.

```
<xsd:simpleType name="VelocityPeriodConstraintType">
    <xsd:restriction base="xsd:string">
    <xsd:enumeration value="D"/>
    <xsd:enumeration value="W"/>
    <xsd:enumeration value="Q"/>
    <xsd:enumeration value="C"/>
    <xsd:enumeration value="M"/>
    <xsd:enumeration value="Y"/>
    </xsd:restriction>
```

```
</xsd:simpleType>
```

8.1.1 Positives

- Schemas are already created and can be automatically regenerated every time a request changes using the schema server.

- Allows system to be easily modified to reflect changes to Cronos / issuer setup. No code changes to completely change the UI and underlying requests

8.1.2 Negatives

- Existing XSD parsers (JAXB) seem to be mainly concerned with generating classes from an XSD (the operate like a code generator). Oil already does this, we just want to populate the Oil objects with values. Needs further investigation, but will probably have to parse the XSDs like plain old XML

- Schemas may require a lot of annotation. If this has to be done each time a new schema is generated it would defeat the purpose of using the schemas.

- When changes to Cronos are made the XSD must be generated and copied into the project. This could be automated fairly easily.

8.2 Request specification using Oil

Request specification is done using the Oil objects. An OilBean is a hashmap; the key set can be used to build a request UI. Each member in the OilBean members map contains some limited metadata. The only really useful thing here is the type, Number or String. An additional file containing the metadata described above would be needed.

```
<Request Name="SetIssuerSMSProperties">
    <Param Name="id" Type="string" Use="required"/>
    <Param Name="shortcode" type="number" Use="optional" uiElem="JComboBox">
        <Defaults>
            <Value>12345</Value>
            <Value>56789</Value>
```

```
        <Defaults>
    </Parm>
</Request>
```

8.2.1 Positives

- Using the OilBeans to specify the requests cuts down on a lot of the markup needed in the first approach. Additionally, to update the wizard when changes to Cronos are made it just needs to be tagged and built.

- Separation of request specification and metadata allows requests to change without any modification of XML unless needed.

8.2.2 Negatives

- Separation of request specification and metadata can make it hard to pin down exactly when something in the XML needs to change. If a new Cronos jar contains a modification to a request the XML metadata specification may need to be updated accordingly. This will also require fairly robust error handling to allow for parameters changing to different types than what is contained in the metadata, parameters being removed or entire requests being removed.

8.3 Work flow specification

The steps need to describe issuer setup in an XML file containing a number of steps, each containing a number of requests. Each step would be a separate screen in the flow. Initially it is presumed that the flow within each step is the same, i.e.

- Fill in values

- System sends request

- Request returns success or failure

- Move onto next screen or end.

If this idea needed to change it could perhaps be specified in XML.

```
<WorkFlow>
    <Step Name="Issuer ID" Number="1">
        <Request Name = "AddIssuer"/>
        <Request Name = "AddCPNType"
    </Step>
    <Step Name="Issuer Config" Number="2">
        <Request Name = "AddEncryptionKey"/>
    </Step>
</Workflow>
```

8.4 Interface Builder

Once some representation of the requests and their order has been constructed a UI needs to be generated. The implementation will allow each step to contain any number of requests with any number of parameters. It should be generated so each requests is isolated in some way, and so the parameters in each request are arranged logically. This allows the user to work through a series of steps within each screen, instead of a random jumble of UI components.
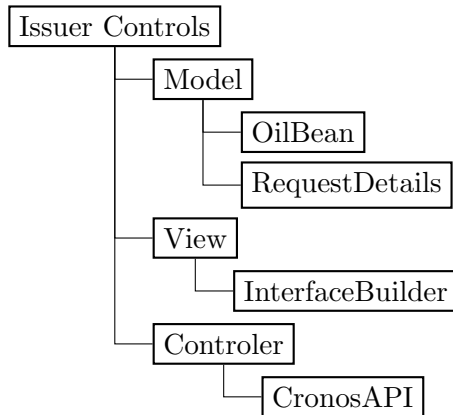
8.5 Request Handler

Once a request has been submitted from the UI it must be sent to Cronos. This can be done in a generic way using Oil. Given the request name, an instance if it can be created.

```
Request request = (Request) Class.forName(requestClass).newInstance();
```

Every Oil request extends the Request interface so they can be processed using polymorphism, instead of having handler methods for each request. Responses contain a simple success or failure that can be communicated to the user. The error codes returned by Cronos on failure are (usually) generic so there's not much point worrying them.

9 Class hierarchy

```
Issuer Controls
    ├── Model
    │       ├── OilBean
    │       └── RequestDetails
    ├── View
    │       └── InterfaceBuilder
    └── Controler
            └── CronosAPI
```

References

[1] M. Worldwide, Benefits of Open Payment Systems and the Role of Interchange. Purchase, NY: Mastercard Worldwide, 2008. `http://www.mastercard.com/us/company/en/docs/BENEFITS% 20OF%20ELECTRONIC%20PAYMENTS%20-%20US%20EDITION.pdf`.

[2]

[3] S. I. G. in Software Testing, "Standard for software component testing," tech. rep., British Computer Society, 2001. `www.testingstandards. co.uk`.

[4] J. Frederick P. Brooks, "No silver bullet," Computer, vol. 20, no. 4, pp. 10 – 19, 1987.

[5] J. Pan, "Software testing," in 18-849b Dependable Embedded Systems, Carnegie Mellon University, 1999. `http://www.ece.cmu. edu/~koopman/des_s99/sw_testing/`.

[6] A. D. McKinnon, Interface Definition Language. Washington State University, 2007.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, p. 15. Addison-Wesley, 1994.

[8] T. Neward, Understanding Class.forName. `http://www.theserverside.com/news/1365412/ Understanding-ClassforName-Java`.

[9] G. McCluskey, Using Java Reflection, 1998. `http://java.sun.com/ developer/technicalArticles/ALT/Reflection/`.

[10] OIL.

[11] M. Konda, Just Spring. O'Reilly, 2011.

[12] V. Patel, Create Spring 3 MVC. http://viralpatel.net/blogs/spring3-mvc-hibernate-maven-tutorial-eclipse-example/.

[13] Aspect-Oriented Programming, Springer-Verlag, 1997. https://www.cs.washington.edu/education/courses/503/08wi/aop-ecoop-1997.pdf.

[14] D. A. Wheeler, SLOCCount.

[15] Orbiscom Payments Platform Technical Architecture, 2002.