[1 Introduction 1](#_Toc511292151)

[2 API and Service interfaces 1](#_Toc511292152)

[2.1 Service interfaces 1](#_Toc511292153)

[2.2 API interfaces 2](#_Toc511292154)

[2.3 Firewall considerations 2](#_Toc511292155)

[2.4 Conclusion 3](#_Toc511292156)

[3 Language considerations 3](#_Toc511292157)

[3.1.1 Administration versus programming 3](#_Toc511292158)

[3.1.2 Data driven versus function driven versus object driven metaphors 3](#_Toc511292159)

[3.1.3 Environment based modelling versus technology based modelling 3](#_Toc511292160)

[4 Location of access considerations 4](#_Toc511292161)

[4.1 General considerations: 4](#_Toc511292162)

[4.1.1 Economies of expertise. 4](#_Toc511292163)

[4.1.2 Essential complexity 4](#_Toc511292164)

[4.2 Browser perspective 5](#_Toc511292165)

[4.3 Desktop perspective 5](#_Toc511292166)

[4.4 Mobile perspective 6](#_Toc511292167)

[5 Integrity considerations 6](#_Toc511292168)

[5.1 Consistency between human versus machine readable data 6](#_Toc511292169)

[5.2 Consistency across multiple copies of the same data 6](#_Toc511292170)

[5.3 Data integrity 7](#_Toc511292171)

[5.4 Business rules 8](#_Toc511292172)

[6 Data binding considerations 8](#_Toc511292173)

[6.1 Data sets and data objects 8](#_Toc511292174)

[6.2 Transaction management and concurrency control 8](#_Toc511292175)

[7 Data security considerations 9](#_Toc511292176)

[8 Economies of opportunity considerations 9](#_Toc511292177)

[9 Big picture 9](#_Toc511292178)

# Introduction

Under **eco-system**, I understand the drive to look at the bigger picture, at the role of the parts in the bigger wholes.

This is not an exhaustive coverage of all the architectural considerations for the whole cluster of MIMS systems, nor of the environment in which it operates.

It is somewhat biased towards the issue of deciding how to integrate the MIMS SHOP to the existing cluster.

This does not exclude the possibility that we might stumble upon considerations that have a wider scope of applicability.

This presentation is aimed at highlighting the difference between an **API interface** and

a **service oriented interface.**

In addition, I will also touch on the selection of various GUI interfaces at the presentation layer.

All of this is an attempt to elaborate on the theme of what it means, in this context, to use the right horses for the right courses.

What this boils down to is to find the optimal trade-off between many considerations, including:

* Economies of expertise
* Economies of scale
* Economies of opportunity
* Minimisation of technological debt
* Functionality
* Reliability
* Adaptability…..

# API and Service interfaces

As a very broad guideline one would expect to find service oriented interfaces between tiers, and

API interfaces within tiers.

In practice, it turns out to be a bit more complicated than that.

## Service interfaces

From a design perspective, **service orientation** can be seen as an extreme form of reductionism

as opposed to creating monolithic systems.

I.e. all systems can be seen as a set of atomic and yet cooperating services.

↑ This approach is attractive in the sense that you get the impression that you can combine anything in any way anywhere at any time.

Technically, service orientation comes to its own as a bridge between disparate and otherwise incompatible technologies.

In these cases, service interfaces are implemented via web services, WPF services etc.

As could have been expected, this entails that each of these pieces of data have to be converted to and from a format,

A format that can be processed by the respective end points.

One way to make this as universal or open ended as possible, is to transport the data in as technology neutral form as possible.

Hence, the use of data contracts, JSON and XML, SOAP packages, amongst others.

→ This approach could be seen as being pregnant with possibility, not committed to anything in particular.

Some people experience power, not so much in the actuality of what they own,

but rather in the potentiality of what they own

leaving the process of converting the possible to the actual on the back-burner – for now.

## API interfaces

Application programming interfaces also support modularity, but generally use larger building blocks,

that can be tied together much more intimately.

In practice this is achieved by creating assemblies of code that can be integrated as link libraries.

In the Microsoft Dotnet world it is referred to as dynamic link libraries.

The code is stored in an intermediate format called MSIL.

As such, it can be generated from many different programming languages, and yet be integrated with each other.

## Firewall considerations

In the case of the MIMS shop, the whole system operation can be implemented on the same platform using the same technology.

Even so, it is necessary to use a service interface to the SQL server and to MobiMims.

There is also a service interface between the CPD website and the MIMS Subscription system. Note that the Website resides outside of the firewall. One would not like to give an external entity direct access to the SQL database. So we give the external Website access to the subscription system service only via the WCF service, whereas we give it SQL access to the CPD database.

This discrepancy should be revisited.

## Conclusion

With Service Orientation, a lot is gained in inter-operability but a of lot is also lost in transmission and translation.

When you are forced to work across different technologies, this is the price you have to pay for the benefit to be gained.

# Language considerations

When choosing between scripting languages and full blown languages, and between Services and APIs, the following considerations might be relevant.

### Administration versus programming

When you have a system that provides most of the required functionality, one could get away by having an administrative interface by which you can fine-tune the system to your requirements.

When you need procedural changes you need a little more than parameter twitching, in which case a scripting languages and command languages becomes necessary.

As soon as you have to develop large chunks of new functionality that cannot be assembled out of the existing modules, a full grown development language might be more appropriate.

### Data driven versus function driven versus object driven metaphors

The semantics of services can cater for function or operation expressions.

RESTFUL services are largely restricted to data-centric approaches.

Service orientation, as far as I understand it, forfeit some of the advantages of object orientation.

### Environment based modelling versus technology based modelling

Whatever kind of notation you use, maintenance and adaptability can be improved by

expressing the system in **business environment terminology**,

*rather than* **computer technical terminology**.

If code is written 5 times, it is read 50 times, and if the reader has to translate the technical terms into business terms every time he reads it, 50% of the productivity is already lost.

Some language and technologies lends themselves more to environment based modelling than others.

What seems key, however, is to strive towards the minimum distortion of the essential model due to accidental language considerations.

# Location of access considerations

The CPD test system and MobiMims and the MIMS shop are accessed by the public, so there is no doubt those should provide internet access.

What could be done is to upgrade it to Dotnet 4 MVC technology.

What might be discussed is:

* what portion of the user data should be captured via the MIMS Shop, as opposed to the MIMS Administration system.
* whether the MIMS Administration system should also be a web GUI.
* Whether the CPD test capture system should happen via a browser
* whether the MIMS medical GUI should be on a browser, or a desktop or a tablet or all.

## General considerations:

### Economies of expertise.

If all GUIs could run on browsers, one could achieve economies of expertise, because you do not need:

Web developers and Desktop and Android developers.

To adjudicate on this matter, one will need a good understanding of all the applications and of all the technologies.

Selecting any particular technology purely on the basis that that is the only technology you know might not suffice.

### Essential complexity

Keep in mind that the current MIMS cluster does not only capture subscriptions. It also handles the delivery of the products, stock control, marketing and the financial accounting.

Whether one can and should implement the same complexity and power of functionality on a web interface versus

a desktop interface versus

an Android app is what has to be investigated.

a) Broadly speaking, it might hinge on the difference between a transaction system on the one hand and

a content rendering system on the other hand.

b) It might also be somewhat related to whether you want a wizard driven system as opposed to an

event driven system, and

whether you want the client side to stateless or

stateful.

In what follows, an attempt will be made to at least list the functionality required by and used by the current system.

## Browser perspective

From the perspective of the web shop, one proposal is to implement the MVC pattern.

What that boils down to is that the presentation layer is split up into the **View** that handles the presentation on the browser, and

the **Controller** that handles the communication between the browsers The **Model** of the MVC is implemented as a combination of the Data and Business Tier in the classic three tier architecture.

As for the View, it seems that one should standardise on HTML5 and CSS3.

## Desktop perspective

The desktop is implemented with the MVVM design pattern, which is a variant of the MVC pattern.

The current system is already aligned to fit in with both the MVC and the MVVM design patterns.

There are currently doctors who prefer E-MIMS above Mobi-Mims, since they do not have network connectivity, and they want the information stored on their computer.

## Mobile perspective

I would imagine that some would want to look at the medical information on a **browser** on a mobile device.

Yet others would want to use an Android app, since they want to take advantage of the functionality on the device

that goes beyond what can be provided by the browser.

For Android access, we might indeed make use of a web service or WCF service, connected to the common Database,

since in this case we are trying to integrate disparate technologies.

It seems as though Android supports Restful Services.

I do not know whether Android apps can connect directly to MS SQL databases?

# Integrity considerations

As soon as an application is to update the database, integrity checking and

user guidance becomes important.

One should enforce integrity via: Controls, properties, as well as via the DBMS constraints.

Consistency has to be enforced across many dimensions:

## Consistency between human versus machine readable data

In order to enforce uniqueness, data often have to be encoded into keys, rather than using plain text.

Conversely, in order to for humans to understand data, it often has to be in the form of plain text.

At any point in time, the run time environment might host a machine and a human readable version of the same data.

These two have to be kept in synch.

A similar argument applies within the programming environment where **enums** are used to cater for human and machine readable data.

To add insult to injury, there needs to be consistency between the keys used in the database and the enums used in the code.

## Consistency across multiple copies of the same data

At any point in time, the same data item might exist in a **GUI control**,

in a **data object** in volatile memory and

in the **database** in persisted memory.

These three have to be kept in sync.

Copying data between these three layers can become unmanageable complex.

It is for this reason that **data binding** and **database adapters** and **data sets** become crucial.

The ability to represent data in different ways lies at the heart of reasoning or problem solving.

It relies on the ability to bring physically dispersed information into cognitive proximity.

So the way data is organised in a database is different from the ways it has to be organised on the GUI.

In addition to displaying the relevant data, the system should also provide the tools to action the decisions made on the basis of the data.

Such actions could have ripple effects on the data currently displayed on the screen.

The challenge is to keep all of this in sync.

After all the what-if scenarios have been played out, the end result has to be persisted back to the database.

This introduces the need for **transaction management** and **concurrency control**.

## Data integrity

Data integrity pertains to what values the data are allowed to assume, under any circumstances.

The sooner you can intercept invalid data, the better. If you can intercept it in the GUI, the user can rectify it immediately.

The next level is in the Data tier objects as discussed previously.

The final check is applied by the DBMS, at the cost of having to have a round-trip to the database. The database can of course span its nets to a wider context when doing its checks.

In the current system the data tier classes have been improved so as to act as **domain models** in both the MVC and the MVVM design patterns.

This is achieved by making use of properties, code attributes, as well as the built-in properties in the dataset and data table classes.

The data objects thus contains both the data as well as the constraints applicable to the data as well as the mechanisms to inform the views of violations.

## Business rules

In addition to the data checks that constraints the values of the data,

the system should also cater for constraints in the transitions between the values in the data.

This is supplied by the Business tier.

# Data binding considerations

Both WPF and Razor provide mechanisms to support data binding, i.e. determining when data should be copied, as well as

in which directions.

Note that data binding applies to:

* Machine readable data
* Human readable data
* GUI state data.

## Data sets and data objects

Whereas **SQL tables** satisfy the requirements of normalisation,

**datasets** satisfy the requirements of conceptualisation.

Thus, for query purposes, there need not be a one to one relationship between SQL tables and Datasets.

Much of the trick lies in translating between the different representations of the data.

Data objects also function as the parameters that is passed to and from the static business tier types.

Data objects can be grouped into collection objects, which immediately gives you all the power of Dotnet to manipulate data in collections, including LINQ.

## Transaction management and concurrency control

Transaction management is implemented in the Business tier. Every successful transaction is also logged to the Transaction log.

Concurrency control is enforced by the technology inherent in the Data table construct. At the moment we implement optimistic concurrency control. Clashing updates have to be resubmitted.

# Data security considerations

Access to data is generally controlled at table level and the stored procedure level, using Windows integrated security.

In addition to this, all database records are stamped by two fields, called

* ModifiedBy
* ModifiedOn

The Windows user*d*  is also recorded in the Exception table. This helps one to follow up on bugs and on inappropriate training.

It stands to reason that one will want to keep stricter control over actions that have wide and far ranging effects as opposed to changes that have limited impact.

For stricter control, I use Windows based authentication and authorisation, whereas the outside world requires a separate security system, based on different considerations.

# Economies of opportunity considerations

The ideal is for the different parts of the MIMS cluster to derive some benefit out of it being integrated.

For a start, that would suggest that the architecture is such that the parts are integrate-able or inter-operable.

# Big picture

What counts for profitability is:

* not information technology
* not the accumulation of information
* but the use of information

An Achilles heel of a publisher is to gauge the value of the information it is publishing.

That would determine what they would be willing to pay for it.

Since the subscription system is the centre of the hub of the MIMS cluster of systems, it should contribute towards determining

* what to publish,
* to whom
* when
* how - on paper and/or on the web.

This is another way of saying that the data in the subscription system should be subjected to Business Intelligence processing.

As soon as we start considering Business Intelligence, the question of how much data about our customers we want to capture becomes relevant, and that in turn has an impact on the complexity of the MIMS shop GUI.

Whether you are going to apply business intelligence on the captured data will also determine the kind of database technology to use.

Failing to apply business intelligence could result in Tiso, and in IT in particular, dumping masses of effort into a bottomless pit,

without yielding any profit.

If you are not really going to exploit the data, we might as well drop the integrity and security controls mentioned above.

BI processing is not really currently done on the MIMS systems.

So much for a MIMS eco system!