[1 Introduction 1](#_Toc510687899)

[2 API and Service interfaces 1](#_Toc510687900)

[2.1 Service interfaces 1](#_Toc510687901)

[2.2 API interfaces 2](#_Toc510687902)

[2.3 General comments 2](#_Toc510687903)

[3 Language considerations 2](#_Toc510687904)

[3.1.1 Administration versus programming 3](#_Toc510687905)

[3.1.2 Data driven versus function driven versus object driven metaphors 3](#_Toc510687906)

[3.1.3 Environment based modelling versus technology based modelling 3](#_Toc510687907)

[4 Location of access considerations 3](#_Toc510687908)

[4.1 Browsers versus application 4](#_Toc510687909)

[4.2 Browser perspective 5](#_Toc510687910)

[4.3 Desktop perspective 5](#_Toc510687911)

[4.4 Mobile perspective 5](#_Toc510687912)

[5 Integrity considerations 6](#_Toc510687913)

[5.1 Consistency between human versus machine readable data 6](#_Toc510687914)

[5.2 Consistency across multiple copies of the data 6](#_Toc510687915)

[5.3 Data integrity 6](#_Toc510687916)

[5.4 Business rules 7](#_Toc510687917)

[6 Data binding considerations 7](#_Toc510687918)

[6.1 Data sets and data objects 7](#_Toc510687919)

[6.2 Transaction management and concurrency control 8](#_Toc510687920)

[7 Data security considerations 8](#_Toc510687921)

[8 Big picture 9](#_Toc510687922)

# Introduction

This is not an exhaustive coverage of all the architectural considerations for the whole cluster of MIMS systems.

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**

operatingbetween the presentation layer and the underlying core system.

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 monolithic systems.

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

I.e. attractive in the sense that you get the impression that you can combine anything in any way anywhere at any time.

Technically, service orientation manifests 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 that can be processed by the 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 by as link libraries.

In the Microsoft 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 integrated with each other.

## General comments

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.

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

Even so, the MIMS cluster of applications are distributed across many servers.

For in-house administration, support personnel use the full power of Dotnet on desktop application, merely by connecting to the common SQL database.

# 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 does 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. In some cases, you need a little more than parameter twitching, in which case a scripting languages and command languages can be very useful.

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. Some, such as RESTFUL services are largely restricted to data-centric approaches. One of them cater for object orientation, which combines data and functional semantics. From that perspective, object oriented API seems to be an advantage.

### 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 implementation considerations. ( SDRS )

# Location of access considerations

Historically MIMS was designed as a classic three tier architecture with a Data, Business and Presentation Tier in an attempt to satisfy the abovementioned requirements. Lately, an attempt has been made to upgrade it to Dotnet 4 and to align the architecture towards the future enhancements. (80% done)

Like anything else, it can be improved upon. What is most urgent now is to also cater for the Web environment and for the Android environment.

## Browsers versus application

MIMS shop should be a web GUI. That is a no brainer.

What might be discussed is:

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

Considerations:

1) Economies of expertise.

If all GUIs could run on a single technology, one could achieve economies of expertise, because you do not need both web and Desktop and Android developers.

The best person to adjudicate this issue, if only from a technical perspective, would be the one who is an expert on all three technologies. Selecting any particular technology purely on the basis that that is the only technology you know might not be the ideal.

2) Essential complexity

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

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

a desktop interface versus

an Android app is something to be investigated.

a) Broadly speaking, it might hinge on the difference between a transaction system on the one hand and the mere rendering of content 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.

For web access, the outside world connects to the website that in turn connects to the common SQL database. In particular, the connection is made by standardising on HTML5 and CSS.

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.

## 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 and the system. The model of the MVC is implemented as a combination of the Data and Business Tier in the classic three tier architecture.

## Desktop perspective

The desktop is implemented with the MVVM design pattern, which is a variant of the MVC pattern. In this case, the ModelView is based on the classic Data tier objects. In some cases it is necessary to preserve the state of the user interaction in the View Model that might wrap a data tier object.

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 app, since they want to take advantage of the functionality on the device that goes beyond what can be provided by the browser.

# 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 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 strings.

At any point in time, there might be 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 aid human memory.

It is also necessary to have mechanism to interop between Enums in the code and keys in the database. This can be done via the EnumTable and via C# cast operations.

## Consistency across multiple copies of the data

At any point in time, the same data item might exist in a GUI control, in core memory and in the database. 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.

Note that the essence of a user interface is to bring logically related data into physical proximity – in order to facilitate decision making. In this situation it might happen that a user changes data in one control. But that change has could have ripple effects to other controls on the same screen – and the screen has to be kept consistent with itself. ↓ In this context, it is important to differentiate between loading the data, including the derived data originally, and updating it subsequently.

Even so, while experimenting between various scenarios, the changes to data should not be persisted unless it is in a consistent state, not only in the current screen, but within the whole 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. These constraints are applied from within the Data tier objects, by using properties rather than just fields. In addition, constraints are applied in the datasets that reside inside the data objects.

This makes units of data consistent in the sense that it contains both the data plus the constraints that apply to the data.

Strictly speaking, 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.

## 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 provides mechanisms to support data binding, i.e. determining when data should be copied, as well as in which directions.

Note that data binding links to:

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

You can bind only to properties in the ModelView. For the original data, you can use SQL joins to obtain derived data.

When you update a primary value in core, then some derived data will also have to be changed, specifically if the GUI is bound to them.

## Data sets and data objects

Whereas SQL tables satisfy the requirements of normalisation, datasets satisfy the requirements of using the data. Thus, for query purposes, there need not be a one to one relationship between SQL tables and Datasets.

In some cases one can get the best of both worlds by creating a data table such that the top fields map directly to an SQL table, usually machine readable data, and the subsequent fields map to corresponding human readable data. Whereas the former fields are involved in insert, update and delete commands, the rest could be seen as read only and as being disposable.

In very simple cases, one could use graphical programming and utilise the ADO.net dataset along with its ‘built in” adaptor and methods.

Datasets interact seamlessly with SQL Reporting Services that is used extensively, also to generate business documents.

Datasets interact seamlessly with SQL database Engine

SQL database Engine interacts seamlessly with SQL Analysis Services that is used in Business Intelligence.

SQL database interacts seamlessly with Excel that is used extensively for end user computing.

In more complicated cases, one would want to wrap the dataset *within* a larger data tier class. Most of what follows is an elaboration on why that would be advantageous to do that. Here you could use the dataset designer or you could create your own Command and Adapter objects.

Even so, one can wrap such a dataset in a data object, in order to get the benefit of using properties, as well as providing a basis for data binding, both to WPF forms and typed MVC views.

Such 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. Clashing updates have to be resubmitted.

# Data security considerations

Access to data is generally controlled at table level, using Windows integrated security. In addition to this, all database records are stamped by two fields, called

* ModifiedBy
* ModifiedOn

This keeps track only of the *last* update. Transaction specific tasks are recorded in the Transaction table.

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

# 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, as well as what 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.

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

So much for a MIMS eco system!