Composite as Metamodel: a Design Pattern for Implementing SIGOBT-style Object Wrappers for HL7 Messages

Frederick KOH, Centre for Medical Informatics, Nanyang Polytechnic, Republic of Singapore
Tel: (65) 550 1675 Fax:(65) 452 0700 Email: Frederick_KOH@nyp.gov.sg

We describe a Design Pattern applicable to the design and implementation of SIGOBT-style object wrappers for HL7 messages. The Pattern described is a special case of the Composite Pattern and is found in ProtoGen, the MS-HUG SIGOBT implementation and in our own implementation of the SIGOBT recommendations at the Centre for Medical Informatics.

1. Introduction

As part of our efforts to develop applications for our collaborators in the Singapore healthcare industry, we have developed a component library for HL7 interfaces [HL7 1997]. This library follows the recommendations of [SIGOBT1998] in the definition of its software interfaces, and the Encoding Rules and Lower Layer Protocol recommendations in the implementation of its parsing, generation and transport of HL7 messages. The topic of this paper is the design of the library and what its design shares with that of two other libraries, ProtoGen [Schadow1996a] [Schadow1996b] and the MS-HUG Implementation of SIGOBT [MSHUG1997a] [MSHUG1997b]; in other words, the design pattern they all follow.

2. Context

2.1 Messaging Standards and General Architecture of Message Handling Applications

Standard message formats facilitate the exchange of data between independently developed computer applications. The HL7 Standard [HL7 1997] defines a set of abstract messages which healthcare applications can exchange in order to share data or to notify each other of important clinical events. It also defines the encoding rules which are used to map instances of abstract messages into a text-based representation that can actually be transmitted using relatively simple communication technologies and protocols.

An application that exchanges such messages is best implemented in two software layers:

- a lower layer that does two things :
    1) parse a message that has been received into program data structures [Seliger1995]
    2) translate program data structures into valid messages and transport them to their destinations.
- a higher layer client that fills in such program data structures to pass on to the lower layer and also processes the program data structures it receives (from the lower layer).

2.2 The HL7 Standard

As in most other message formats, HL7 messages have a hierarchical containment structure. Messages are made up of segment groups, which are made up of other segment groups and also segments and so on. At the bottom of this hierarchy are the primitive data types. Through its various releases, the HL7 Standard has introduced new message types and also new message component types and also modified existing ones. However, the general structure described above, and the primitive data types have remained virtually unchanged. Also left unchanged are the encoding rules mentioned in 2.1.

2.3 Structure of HL7 Messages

A HL7 message is composed of segment groups and segments. Segment groups are recursive structures, i.e., they may contain segment groups (and also segments). Segments are made up of fields, which may have components. Segment groups and segments may repeat.

In the example of an `ADR` message in 2.5, each line of text is a segment. The first 3 characters of a segment is the name of the type of the segment. A `PID` segment followed by a a `PV1` segment forms a segment group. In the `ADR` message, this group is a "repeating group" or collection. In this case there are 4 repetitions. Within each segment, fields are delimited by "|". A field may repeat and repetitions are separated by "~". A field or each of its repetitions (depending on its type) may contain components separated by "^". Components(depending on its type) may be further subdivided into components delimited by "&". There are no sub-subcomponents of fields.

Fields of a primitive type (as opposed to composite types) may not contain components. Examples of primitive types are NM (numbers), ST, FT and TX (various types of strings).

This does not appear in the message instance itself, but fields in a segment are given names to indicate the kind of information they contain. For example the fifth field of the PID segment is named "Patient Name".

2.4 The SIGOBT Recommendations

Not to be confused with the HL7 Standard itself are the SIGOBT recommendations [Seliger1995] [SIGOBT1998]. Whereas the HL7 standard defines the message format, the SIGOBT recommendations define something like the "program data structures" mentioned in 2.1. The Recommendations serve as a kind of API specification for random access to any component of a message. The API is in object oriented form in the sense that message components are modeled as objects. This provides a convenient syntax to "locate" a message component anywhere in the hierarchy of message components. See next section.

2.5 Example of SIGOBT style of accessing message components

Below is an example of how the SIGOBT specifications allow random access to HL7 message components. Given the message instance

```
MSH|^~\&||||||ADR^A19|server48492727644|P|2.3
MSA|AA|v-fred223@4541
QRD|19980519023725|R|I|red845467|||100^RD|*|APN|*
PID|||SF   7244487||^TOM|||||||||||19985090018I
PV1||E|||||||||||||||||||||||||||||||||||||19980109135400
PID|||SA   1234546||^DICK|||||||||||19985090026Z
PV1||E|||||||||||||||||||||||||||||||||||||19980109140500
PID|||SI   0007255||^CHEOK KIAT HUAT|||||||||||19972915369F
PV1||E|||||||||||||||||||||||||||||||||||||19971018143800
PID|||SA   7689081||^GEOGRE FARNADES|||||||||||19978060125J
PV1||E|||||||||||||||||||||||||||||||||||||19971102084900
```

the SIGOBT specifications allow the following expressions (in Visual Basic 5 through OLE Automation):

`Message.RepGrpPatientRecord` returns the object encapsulating the message from the fourth segment onwards. The returned object is of type `HL7RepGrpADRA19Record`. (The naming convention requires the prefixes `HL7RepGrp`, `HL7Grp` etc according to the "metatype" of the class)

`Message.RepGrpPatientRecord.Item(1)` returns the object encapsulating the group containing the sixth and seventh segments. The returned object (no matter what the index in `Item`) is of type `HL7GrpADRA19Record`.

`Message.RepGrpPatientRecord.Item(1).PatientIdentification` returns the object encapsulating the sixth segment. The returned object is of type `HL7SegPID`.

`Message.RepGrpPatientRecord.Item(1).PatientIdentification.PatientName` returns the object encapsulating the fifth field of the sixth segment. (Containing the value "^DICK"). The returned object is of type `HL7RepFldXPN`.

`Message.RepGrpPatientRecord.Item(1).PatientIdentification.PatientName.Item(0).GivenName` returns the object encapsulating the second component of the fifth field of the sixth segment. (Containing the value "DICK"). The `Item(0)` before `GivenName` is due to the fact that the fifth field is a repeating field. The object returned is a primitive string type.

2.6 Two different models for two different levels/layer of programming

The motivation for the SIGOBT way of modeling a message is that it is a convenient and natural way of referring to the various components of a message. ***It is not a recommendation on how to design the lower layer mentioned in 2.1.*** Its emphasis, rather, is on providing a domain (healthcare) level view of the messages parsed, translated and transported by the lower layer.

Whereas the tasks of the lower layer are the parsing, translation, and transport of messages, the tasks of the higher layer vary greatly from application to application. These include GUI-based applications where information from the messages are displayed and user entry is written into messages, "server" applications which update a (hospital or clinical) database with information from messages and which retrieve database information to write to messages.

The thought processes involved in writing these higher layer applications are in the "domain" level terms mentioned in the first paragraph of this section (2.6): in our case, in terms of "Lab Results" information, "Next of Kin" information, "Insurance" information etc. ***The SIGOBT specification provides such a way of referring to the information contained in these messages.*** Not only that, the message component containment hierarchy reflects the relationships in the domain. For example the segment group that contains a PID segment (demographic data) for a particular patient also contains the PV1 (admission information) segment for that patient.

What this paper wishes to accomplish is to show a pattern which allows for efficient implementation of the lower layer and yet provide the sort of interface specified by SIGOBT to programmers programming at the higher layer. The interest of this pattern is the sheer number of classes required for this at the higher layer (due to the number of types of messages and their components defined by SIGOBT). Also of interest is the fact (mentioned in 2.2) that while the new message types may be introduced and existing ones modified, the general structure remains unchanged.

3. *COMPOSITE* and the SIGOBT metamodel

We refer the reader to [GoF 1995] for a description of *COMPOSITE*.
In the SIGOBT metamodel (see section 2.3 of [SIGOBT1998]), messages, segment groups, segments, composite fields and their repetitions are Composites. The primitive types are Leaves.
If the metamodel describes messages, segments etc. in general, then the model describes particular message types, segment types etc. such as QRD, MSH etc.
[Martin+1995] defines metamodels as models of models: types in a metamodel have instances which are also types. HL7MsgADRA19 (the message used in the example in 2.5), an instance of Message type, is itself a type. In other words, a class in the metamodel describes classes in the model, or more precisely, the characteristics they (the classes described) share.

3.1 Example: ProtoGen

In ProtoGen the `Structure` class is a Component. The `Segment` class is a Leaf and the `SegStruc` class is a Composite. All segment types (`MSH`, `PID`, `PV1` in the example in 2.5) are subclasses of `Segment`. Segment groups are subclasses of `Group`, itself a subclass of `SegStruc`. Message types (like `ADR`) are subclasses of `Message`, itself a subclass of `SegStruc`. A different hierarchy is used to implement the aggregation aspects of `Segments` because its containment nesting level is limited.

4. The Pattern: Implement Metamodel and Maintain Large Model

4.1 The Problem

All the SIGOBT types or classes (except those modeling repeating segment groups, repeating segments, repeating fields and primitive field types) in the model (as distinguished from those in the metamodel) are wholes whose parts are exposed as class methods . This is why there is one class modeling the MSH segment and another modeling the QRD segment even though both are wholes containing parts that are fields and are therefore structurally the same. ***The need to expose individual parts as differently named methods requires them to be distinct classes. To model and implement this (exposing parts as methods) is not only trivial, it is also, tedious, repetitive, error prone (transcribing from one notation (e.g., the HL7 Standard), to another, (e.g., UML) then to the implementation language) and formidable (the sheer number of different message, segment group, segment, and field types).***

4.2 Some observations

We can make the following remark: the problems of hierarchical message component containment (part/whole relationships) and serialization/deserialization into HL7 encoded ASCII can be modeled and solved in the **metamodel** whereas the problem of exposing parts of wholes as class methods is expressed in the **model**.

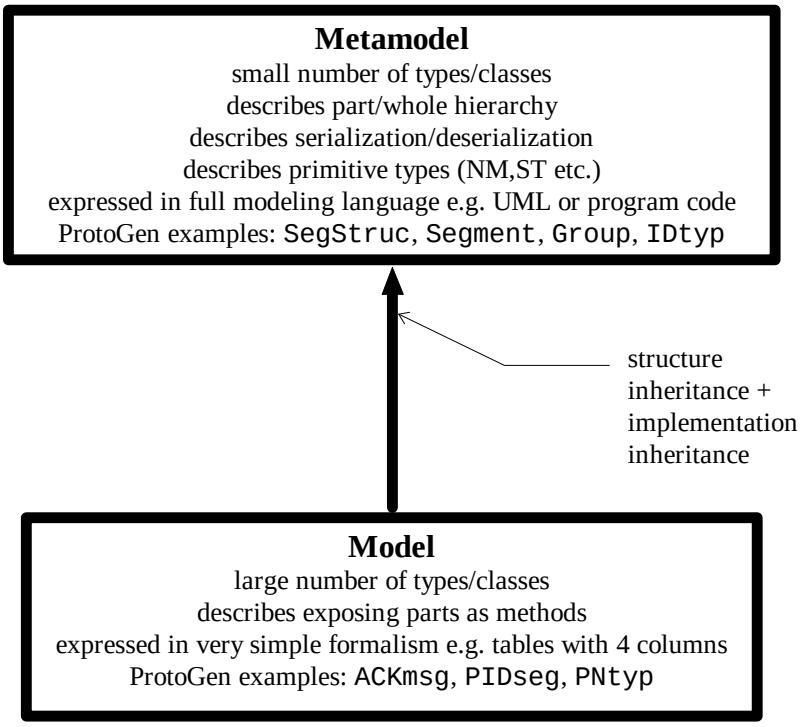
We clarify and expand on this remark by the following observations about the metamodel and the model.

1. It is possible to describe in the metamodel not only what is structurally similar (part-whole relationships) among the types in the model, but also how the collaboration between the whole and its parts work during serialization/streaming and the reverse process (parsing and generation).
2. The metamodel is relatively ***rich*** in its structural and behavioral descriptions. It also contains a relatively ***small*** number of types.
3. The model is ***large*** in terms of the number of types it describes; but it is also ***simple*** since all it needs to record are which parts are contained in which wholes, the metamodel having already described the containment structure and streaming behavior completely. This means we can use a much ***simpler formalism*** (than that used for the metamodel) to completely describe it (not quite yet, see 4.). An example of this formalism is the one given below to describe the ADR message type (definitions for segments and finer grained components not shown). It needs only a table of four columns: the first, the type of the part, and the second, the name of the method that exposes it.

```
MESSAGE ADR^A19
MessageHeader MSH OBJECT REQUIRED
MessageAcknowledgment MSA OBJECT REQUIRED
Error ERR OBJECT OPTIONAL
QueryDefinition QRD OBJECT REQUIRED
QueryFilter QRF OBJECT OPTIONAL
RepGrpPatientRecord ADRA19Record COLLECTION REQUIRED
ContinuationPointer DSC OBJECT OPTIONAL

GROUP ADRA19Record
EventType EVN OBJECT OPTIONAL
PatientIdentification PID OBJECT REQUIRED
AdditionalDemographics PD1 OBJECT OPTIONAL
RepSegNextOfKinAssociatedParties NK1 COLLECTION OPTIONAL
```

```
PatientVisit PV1 OBJECT REQUIRED
PatientVisitAdditionalInfo PV2 OBJECT OPTIONAL
RepSegDisabilityInformation DB1 COLLECTION OPTIONAL
RepSegObservationResult OBX COLLECTION OPTIONAL
RepSegAllergyInformation AL1 COLLECTION OPTIONAL
RepSegDiagnosisInformation DG1 COLLECTION OPTIONAL
DiagnosisRelatedGroup DRG OBJECT OPTIONAL
RepGrpProcedures PR1_ROL COLLECTION OPTIONAL
RepSegGuarantor GT1 COLLECTION OPTIONAL
RepGrpInsurance IN1_IN2_IN3 COLLECTION OPTIONAL
AccidentInformation ACC OBJECT OPTIONAL
UniversalBillInformation UB1 OBJECT OPTIONAL
UniversalBill92Information UB2 OBJECT OPTIONAL GROUP

GROUP PR1_ROL
Procedure PR1 OBJECT REQUIRED
RepSegRole ROL COLLECTION OPTIONAL

GROUP IN1_IN2_IN3
Insurance IN1 OBJECT REQUIRED
AdditionalInfo IN2 OBJECT OPTIONAL
AdditionalInfoCert IN3 OBJECT OPTIONAL
```

4. So that it is possible to use the simpler formalism described in 3) to fully express the model, it is necessary to move the descriptions of the primitive data types (NM, ST, TX etc.) from the model to the metamodel. The description of these types requires the specification of their storage and encoding scheme for streaming, hence the impossibility of using the simpler notation to model them.
5. The metamodel is much more stable than the model: the HL7 committees are likelier to change the specification of particular message or segment types than their general structure. This leverages the simpler formalism that we use for the model: being simpler, it is also easier to maintain when changes occur.
6. Classes in the model inherit from the classes in the metamodel.

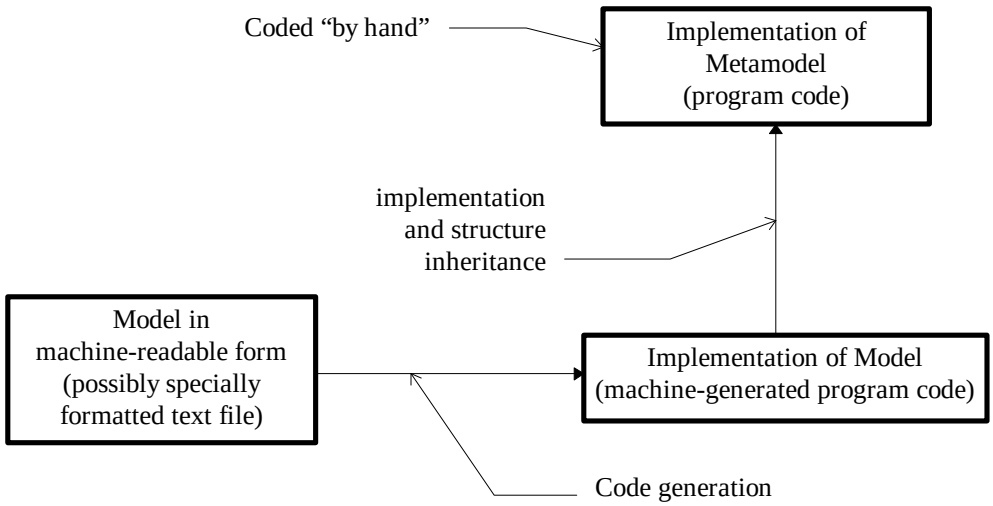


4.3 The solution

Part of the solution is actually already hinted at in the SIGOBT recommendations. By constructing a metamodel, a model is implied: the problem of size and tediousness is isolated to the model. The non trivial aspects of implementation reside in the metamodel.
There are two variations to the solution. The first is for situations where the client language is a statically typed language like C++, Java and Eiffel, where methods of classes are fixed at compile time. The second is for client languages like Visual Basic 5 and Perl where the methods of an instantiated object can be determined at runtime (we describe the mechanism further below).
In both cases the metamodel is implemented first, and independently, of the model.

4.3.1 When the client language is statically typed: the code generation solution

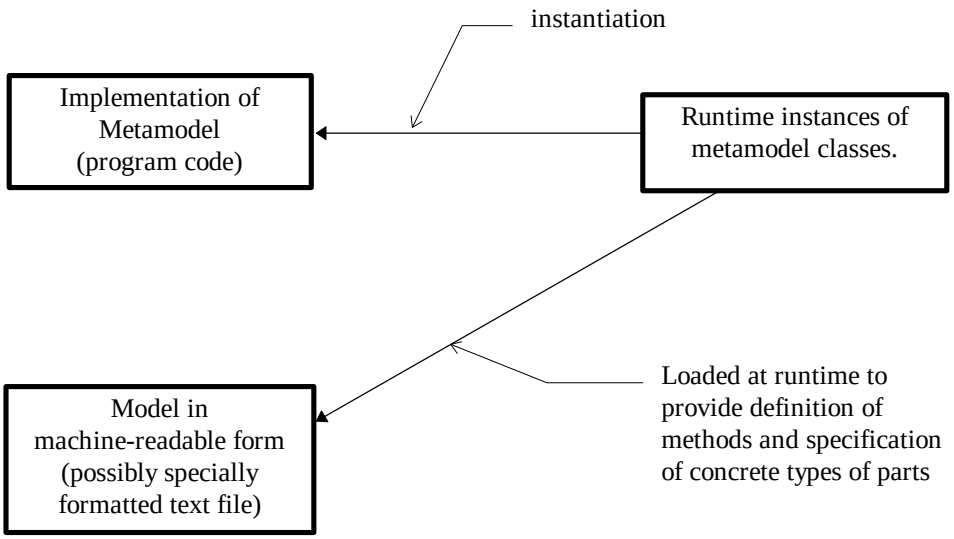
As was remarked above, the model is expressed in a simple formalism. The implementation of the model is nothing more than the wrapping of Composite descended classes to expose parts as individual methods. This means that it is not difficult to write a program that reads the model and generates the corresponding wrapper classes. The generated code is the implementation of the model.
For the convenience of the rest of this exposition, we call this the code generation solution.
The diagram below summarizes this.



4.3.2 When the client language is dynamically typed: the reflection solution

In Visual Basic 5 (VB5), the underlying implementation object model is based on OLE Automation [Brockschmidt1995]. All VB5 classes are implementations of the `IDispatch` interface. It is also possible to implement VB5 classes in C++ by deriving classes from `IDispatch` and implementing them; these classes would be, from the client point of view, indistinguishable from VB5 “native” classes. The interest of using C++ to implement classes for VB5 clients is the full access the language has to `IDispatch` methods. The `IDispatch` interface defines methods that support a limited form of ***reflection*** [Buschmann1996], [Buschmann+1996]. This allows the runtime definitions of an objects methods: methods are not fixed a compile time. Thus it is possible for a C++ class to load method definitions from an external source (a file, say) at runtime and allow VB5 clients to invoke them.
We elaborate a little more on the mechanism here. If a VB5 object reference `obj` is bound, the method invocation expression `obj.Foo` would first call the `IDispatch::GetIDsOfNames` member function (the method name `Foo` is passed as argument)of the implementation of the object to see if the method is indeed implemented and also to get the identifier of the method. This identifier is then passed as argument

in a call to `IDispatch::Invoke` for the actual invocation of the method. From this, it is clear that the definition of VB5 object methods at runtime is within the control of a C++ implementation.
In contrast to the code generation solution, no code generation is required. Whereas in the case of the code generation solution the instances are of classes in the model, here, the instances are of the metamodel classes. In the case of VB5, the metamodel classes are implementations of `IDispatch`. The signature and names of methods to exposes are loaded from the model by the metamodel class implementations; in our case, the model was in the form of a specially formatted text file (in the spirit of the four column tables mentioned above).
For the convenience of the rest of this exposition, we call this the reflection solution.
The diagram below summarizes this.



4.4 Consequences

- Once the metamodel is elaborated, implementation can start even before the model is completed (it can even start without a model, testing can be done on a "dummy" model). This was indeed the case with us.
- Changes, removals, and additions of new message types can be made without any change to the implementation code. Only the model needs to be changed.
- Maintenance, in the sense of changes, removals, and additions of new message types, can be done by non-software professionals and is totally clerical rather than technical in nature.
- There is no need to maintain consistency between model and implementation (both equally huge), or rather, the maintenance of consistency is computerized.

5. Known Uses

5.1 The ProtoGen Implementation

Client language for ProtoGen is C++, hence it uses the code generation solution. What is interesting about ProtoGen is that it includes tools to extract the model (in the form of Prolog predicates) from the HL7 Standard in Microsoft Word format. The generated code supports streaming through sockets and iostreams.

5.2 The MS-HUG Implementation

In the MS-HUG implementation (ATL based and coded in C++), SIGOBT classes are exposed as DCOM ***dual interfaces*** [Brockschmidt1995]. These are still implementations of `IDispatch` but the OLE Automation methods are determined at compile time. This allows efficient access for both C++ and VB5 clients because a given method has two means of access: OLE Automation through `IDispatch::Invoke` and also as C++ virtual member functions.

The need to implement individual methods as C++ member functions means the code generation solution is used. A caveat is appropriate here. Our inclusion of MS-HUG in this paper is based on a ***rational reconstruction*** after careful inspection of the code. We are able to isolate the modules that can be reasonably interpreted as being implementations of the metamodel and also to discern a repetitive pattern in the modules (which allows the conclusion that they can be program generated) that we recognize as implementing the classes in the model. But, we do ***not*** know if the modules were ***in fact*** program generated.

5.3 The HL7 Implementation in NYP-CMI

Among the three implementations studied, ours is the only one using the reflection solution. The metamodel is implemented in C++. As indicated above, the model is in the form of a simply formatted text file and is loaded at runtime. Whereas the MS-HUG implementation uses dual interfaces, we use ***dispinterfaces*** [Brockschmidt1995], i.e., the SIGOBT class methods are not required to be C++ virtual member functions.

To ease the development work on the VB5 client side, we have also used the code generation solution to create the ODL file to generate a DCOM ***type library*** [Brockschmidt1995] from the same text file (the model).

See Appendix A for our interest in this approach.

6. Related work

6.1 ASN.1

Not having found a use of this pattern outside of wrapping HL7 messages, we have perhaps unnecessarily restricted its exposition as such. This was the case until recently, when we found another use of the code generation solution, this time for ASN.1 [Lavender+1994]. By then it was too late to study and incorporate it into our paper. ***We believe this pattern is applicable to any message format with recursive structures.***

6.2 Implementation of Associations by Inheritance

We would like to bring to the reader's attention another pattern suggesting the use of code generation. In pages 129 to 147 of [Soukup1994] and in [Soukup1995], the Association, Aggregation, and Graph patterns were implemented with code generation (with the input being also another piece of code). This suggests to us (what may be a triviality to some, but deserves being stated anyway) that in any software development effort where the same piece of information is maintained in two (or more) deliverables (code, documentation, etc.) and is in the form of an unambiguous formalism (such as code, modeling language or specially and rigorously formatted text), then generating one deliverable from the other avoids extra work in maintaining consistency.

Acknowledgments

I wish to thank Dennis DeBruler for reviewing this paper and helping me make clearer the ideas presented in it.

References

[Brockschmidt1995] Kraig Brockschmidt. *Inside OLE 2nd Edition*, Microsoft Press,1995

[Buschmann1996] Frank Buschmann. "Reflection." In John Vlissides, James Coplien, Norman Kerth (eds). *Pattern Languages of Program Design 2*, Addison-Wesley, 1996, pp. 271-294.

[Buschmann+1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. *Pattern -Oriented Software Architecture: a system of patterns*, John Wiley and Sons, 1996

[GoF1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: elements of reusable object-oriented software*, Addison-Wesley, 1995

[HL7 1997] Health Level 7 Inc. *Final Standard version 2.3*, 1997

[Lavender+1994] R.G. Lavender, D.G. Kafura, R.W. Mullins. Programming with ASN.1 using Polymorphic Types and Type Specialization *Proceedings of the IFIP TC6/WG6.5 International Conference on Upper Layer Protocols, Architectures and Applications*, Barcelona, Spain, 1994

[Martin+1995] James Martin, James J.Odell. *Object-Oriented Methods: a foundation*, Prentice Hall, 1995

[MSHUG1997a] Microsoft Healthcare Users Group, ActiveX for Healthcare Committee. *ActiveX For Healthcare Messaging Components Programmer's Guide Version 1.0*, 1997

[MSHUG1997b] Microsoft Healthcare Users Group, ActiveX for Healthcare Committee. *Increasing Interoperability and Lowering Costs with ActiveX For Healthcare: a white paper*, 1997

[Schadow1996a] Gunther Schadow. *ProtoGen/HL7-An Implementation of HL7, Version 0.9*,1996

[Schadow1996b] Gunther Schadow. *ProtoGen/HL7User Manual, Version 1.0*, 1996

[SIGOBT1998] Health Level Seven, Inc. *Recommendations for HL7 Messaging over Component Technologies Version 1.0 Revision 9*, 1998

[Seliger1995] Robert Seliger. *Implementing HL7 v2.2 Using the Object Management Group's Common Object Request Broker Architecture*, Hewlett Packard Medical Products Group, 1995

[Soukup1994] Jiri Soukup. *Taming C++ :pattern classes and persistence for large projects*, Addison-Wesley, 1994

[Soukup1995] Jiri Soukup. "Implementing Patterns" In James O. Coplien, Douglas C Schmidt (eds). *Pattern Languages of Program Design 2*, Addison-Wesley, 1996, pp. 271-294.

Appendix A - the VB5 IDE and how a SIGOBT-based Library improves software productivity

A VB5 programmer while going about his work will need to know where to get data from and where to store data. Usually this requires understanding of the hospital's database schema and the formulation of SQL queries to get or store information required by his program. All this understanding is invalidated when he moves on to another hospital.

If instead of relying on a particular hospital database schema to program to, the VB5 programmer can use the model provided by the SIGOBT specifications. The entities in this model are practically VB5 objects that the programmer can directly program with. No struggling with database schema or SQL statements, just ordinary everyday VB5 objects.

Not only is this model stable or unchanging as the programmer moves from hospital to hospital, it is also an online model available to him in his development environment even as he programs. When he finishes typing the name of an object declared to be of a type described in the model, a list of possible properties and method appears before him (this feature is known as Intellisense), freeing him from the task of looking it up in a reference manual. This is truly software productivity!