GEHR System Architectures

Authors: Thomas Beale

Revision: 2.1 draft B

Pages: 31

Amendment Record

Issue	Details	Who	Date
1.1 draft A	Initial Writing; content from architecture document	T Beale	9 Feb 2000
2.1 draft A	Major re-organisation of structure, addition of new content, including general architectural pattern.	T Beale	5 May 2000
2.1 draft B	Addition of DEMOGRAPHIC_MANAGER to main diagram.	T Beale	20 Aug 2000

Table of Contents

1 Introduction

1.1 Purpose

This document describes application and system architectures based on the Good Electronic Health Record (GEHR) kernel.

The intended audience includes:

- GEHR application and system architects.
- Health Information System managers wishing to understand the GEHR architecture.

This document references the The GEHR Object Model Technical Requirements and requirements in that document in particular. The format of these requirement references is <requirement label> p<page number> appearing in the side column. An example is given of the requirement for faithful recording of information which appears on page 19 of that document.

Req: legal:faithful p19

1.2 Browsing This Document

This document has active web links which will launch your web browser. These links are displayed as a `hyperlink` and will usually give the web address. If clicking on this hyperlink does not launch your web browser then you can set this up by choosing the Acrobat menu options File > Preferences > Weblink... and completing the dialog box.

1.3 Status

This document is under development, and will be published for inclusion in standards proposals and as documentation for software implementations.

The latest version can be found on `http://www.gehr.org`.

1.3.1 Peer review

Known omissions or questions are indicated in the text with paragraphs like the following:

To Be Determined: not yet resolved

To Be Continued: more work required

Reviewers are encouraged to comment on and/or advise on these paragraphs as well as the main content.

The content of this document can be supplied in Adobe PDF or HTML format on request, to facilitate electronic markup.

Please send requests for information and review comments to `info@gehr.org`.

2 Overview

2.1 General Architectural Model

Because GEHR proposes an information model from which software artifacts can be derived, numerous architectural possibilities are available, undoubtedly including some unforeseen by the original authors of GEHR. The aim of this document is therefore to describe the most likely architectural possibilities. While the details may be quite different, all GEHR-based systems consist of the following elements:

- **User Applications**, consisting of:
  - An application specific part, typically a GUI interface.
  - The GEHR kernel.
  - A client for a persistence mechanism (database).
- **Archetype Initialiser** application, which converts XML archetypes to internal form for GEHR applications to use.
- **Demographic Manager** application, which converts information from the demographic server to internal form for GEHR applications to use.
- A **database**, containing repositories for:
  - EHRs created/modified/viewed by applications.
  - Locally used GEHR archetypes.
  - Locally created GEHR archetype XML documents, where these exist.
- **Terminology server**(s), which provide access to well-known term sets such as UMLS, ICPC and so on. The simplest version of this might be a small local application serving terms from a file of locally defined terms; a more sophisticated version might be an implementation of the CORBAmed TQS specification.
- **Demographic server**, providing identification of person and organisational entities. At its most minimal form, this might be a simple local database; the most sophisticated version might be an implementation of the CORBAmed PIDS specification.
- **Archetype domain server**, providing access to the GEHR archetype domain system.

FIGURE 1 illustrates a notional GEHR system containing these elements.

2.2 Document Overview

The remainder of this document begins by considering the philosophical basis for using an abstract, formal model to describe health record semantics, and shows why such a model is essential in avoiding the weaknesses of protocol- or schema-based record definitions. In this approach, the GEHR Object Model (GOM) is the central definitional artifact from which protocol definitions, database schemas, software applications and documents are developed.

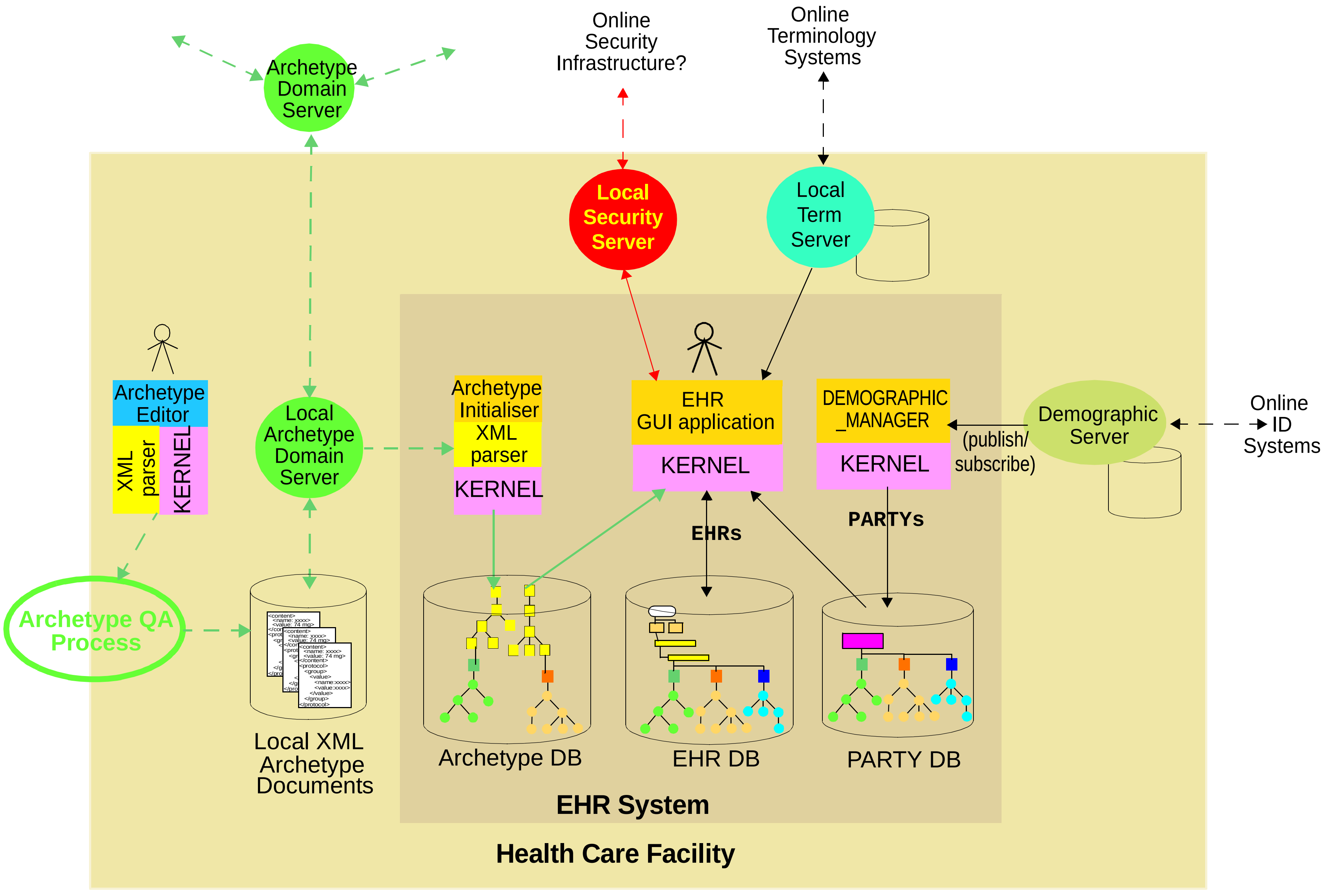

FIGURE 1 General Architecture of GEHR Systems

The development taxonomy of the GOM and its software is described, showing the relationships between the GOM, the GOM kernel, the kernel API, applications, database schemas and archetype definitions.

The subsequent major sections then describe application architectures, database architectures and system architectures in turn. The application architectures section deals with building standalone, client/server and distributed applications, while the database section shows how EHRs can be stored using various database technologies. The systems architecture section describes how these can come together in real systems to be used for clinical settings, from the smallest general practitioner clinic to large hospitals.

3 Philosophical Approach

As a basis for understanding how GEHR systems are built, it is worth addressing some philosophical questions commonly asked of models such as the GOM, namely:

- Why use an abstract model (the GEHR Object Model), why not just describe the record format or transmission protocol?
- How does it translate to a "concrete" health record, i.e. a file or string of bytes which can be stored in a database, or read by software?
- How does it facilitate development of software, databases, etc?

The following subsections deal with these questions in turn.

3.1 Abstract Model versus Format Prescription

To answer the first question, refer to FIGURE 2. This places the GOM at the centre of a number of computer abstractions (see the red triangle marked "Formal Model"). The GOM is:

- A computer version of what we want to say about information structures, hence it is shown as being a formalisation of mental concepts.
- A source from which various other representations can be created, hence the transformations of the model into software, database schema, and documentary forms.

The need for a model is a consequence of the need to make *explicit* the semantics of the record and its operations. Without a model whose primary purpose is to express semantics, implementation expressions such as protocols or database schemas whose primary job is to concretely define tables or data packets end up also trying to *implicitly* express semantic concepts, usually imperfectly.

The purpose of an explicit model is to be a *repository of concepts*, not a prescription for representation. Not only does this avoid it being tied to a particular representation, it addresses the problem that not all semantics we want to record are expressible in formalisms used to describe particular formats, in particular constraints and functional interfaces. In terms of a piece of software, a format description only addresses the question of how to encode a record in a database or on a network, but does not comprehensively treat how applications should allow only valid information structures to be built.

see: section 4.5 on page 48 of The GEHR Object Model Technical Requirements

Using a neutral formalism is a safe thing to do as long as there is a way to get from it to any particular formalism required for software development. The formalisms of interest are those by which electronic healthcare systems will *communicate* records; there is no direct requirement on the construction of software. As discussed in the Requirements, communication implies *representation*, which in turn requires particular format descriptions. For the typical modes of communication - file, database, object distribution or messaging - there are various standard and not-so-standard formats.

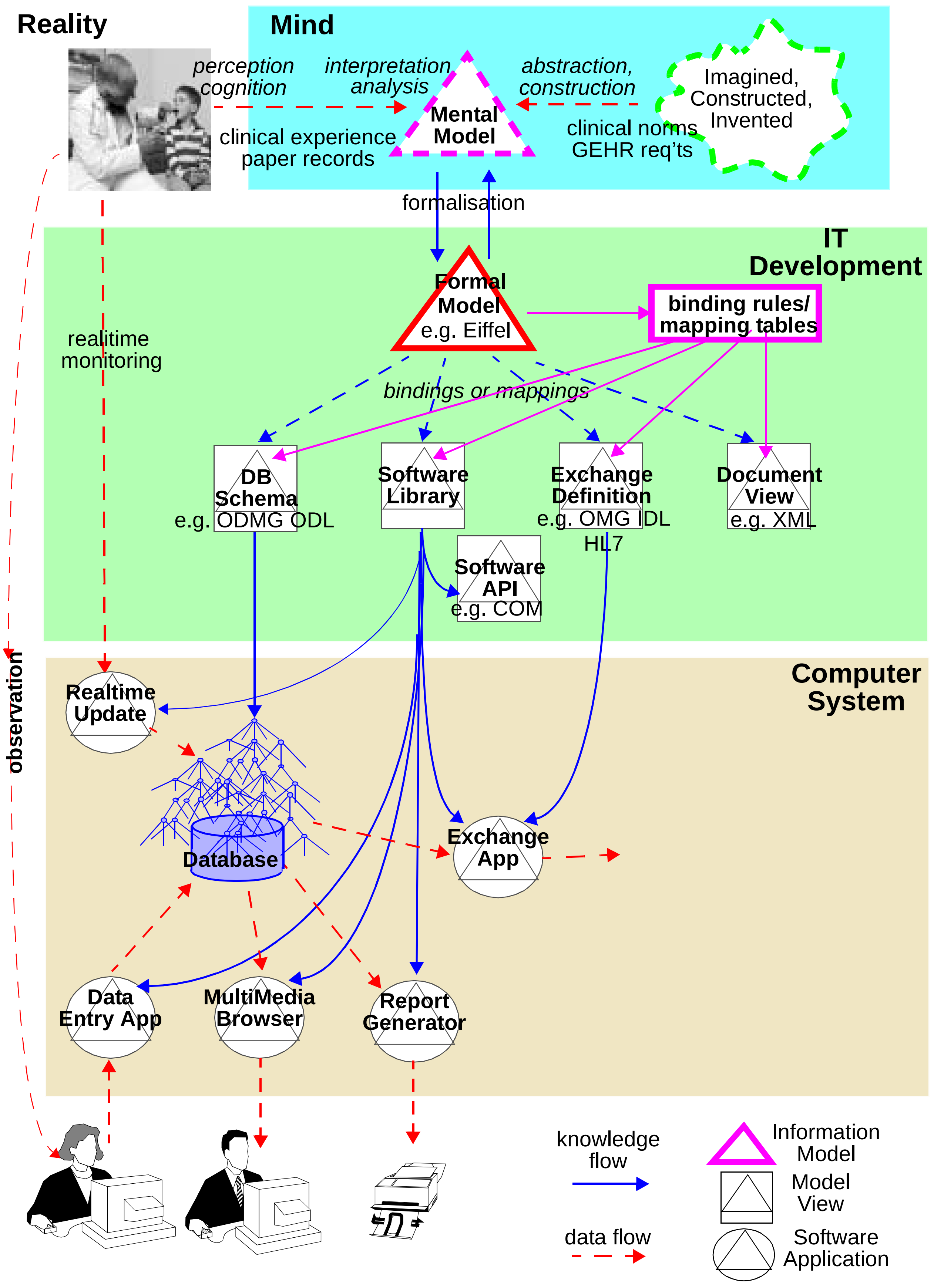

FIGURE 2 Taxonomy of Object Models and Software

3.2 Record Representation

The problem of how to get from an abstract model to an actual health record is addressed by providing expressions of the model in such formats. As mentioned in the Requirements, there are two ways to do this: via a *binding* (set of rules via which a transformation can be made at any time), or via a *mapping* (a complete expression of the model in the target format). There are already standard bindings for popular media, and it is expected that certain mappings (e.g. the GEHR Exchange Format) will be devised as part of this project.

3.3 Seamless Software Development

Moving from an abstract model to a concrete representation in a given protocol or format requires a *seamless* relationship between two software entities.

A number of criteria for modelling formalisms were discussed in the Technical Requirements, including implementability, semantic power, and clarity. We can add a further property, highly desirable from the design and implementation perspective: that of *seamlessness*. Briefly put, seamlessness is a property of software artifacts such that the expression of the model embodied in the software at different levels of abstraction is done using essentially the same formalism. More concretely, if software is constructed seamlessly, the analysis, design and implementation artifacts will all be *views*, at decreasing levels of abstraction, of a single underlying model (some theorists believe even that the requirements can be seamless with the other artifacts, but this relies on assumptions too strict to be generally useful, at least with today's tools).

Contrast this with the usual (and lamentably, often accepted) situation in software development where analysis and design models are typically expressed in some diagramming notation, and used as a basis for system design, review, and as a *source document* for the implementation, itself done in a different language. Once the implementation starts, it takes on a life of its own: discoveries are made which change the original design model. However, it is usally too difficult to track these changes and make the corresponding changes to the design model. Consequently, the implementation drifts away from the design, rendering it an out-of-date, untrustworthy document, of little use to maintainers or new developers learning the system. A similar lack of traceability to the requirements document leaves it in an even worse state, relegating it simply to a historical snapshot.

The use of seamless formalisms on the other hand allows us to pursue what should perhaps be one of the basic goals of any systems engineering endeavour: the creation and maintenance of "living" models and documents, whose validity does not diminish with time. To see how important this goal is, one only needs to consider the costs of not being seamless: vastly increased maintenance costs, the high cost of modifications and enhancements (particularly to requirements or design, let alone implementation), the lost opportunity costs of being *unable* to implement desired changes, and the costs of being unable to effectively *reuse* any of the previously developed artifacts. "Maintenance" is generally estimated to cost a minumum of 50 - 70% of the total cost of a system over its lifetime.

The importance of maintainability and extensibility of systems based on a model for clinical records is clear: the first release of the model (and software based on it) will only be the beginning. Initial user feedback, new clinical practise models (e.g. evidence-based medicine), increasing sophistication of second order requirements (reporting, decision support, population medicine etc) and so on will all contribute to the evolution of the model; but *so will implementation concerns*. If any GEHR implementation is not to drift away from its originating model, making the claim that the implementation “conforms to” or “is based on” the model increasingly doubtful, seamlessness is essential.

In FIGURE 2, the knowledge flow lines (solid and dashed blue) from the formal model (red triangle) indicate seamless relationships. For any such relationship, the entities at either end are (as much as possible) rigorous transforms of each other, expressed in different formalisms. Thus, an expression of the model in Eiffel can be converted to an XML documentary form, which is useful for viewing records in a web browser, transferring textual versions of the record and so on.

While FIGURE 2 represents an ideal, many aspects of it are attainable (all the relevant technology exists), and given that expressions of the GOM are likely to be required in numerous forms, a crucial one to keep in mind. The alternative is a sea of informally “GEHR-compliant” software, databases, and documents.

4 Taxonomy of Deliverables

4.1 Overview

Let us consider the actual expressions of the GOM likely to be required for constructing computer systems, beginning with the simple application illustrated in FIGURE 3.

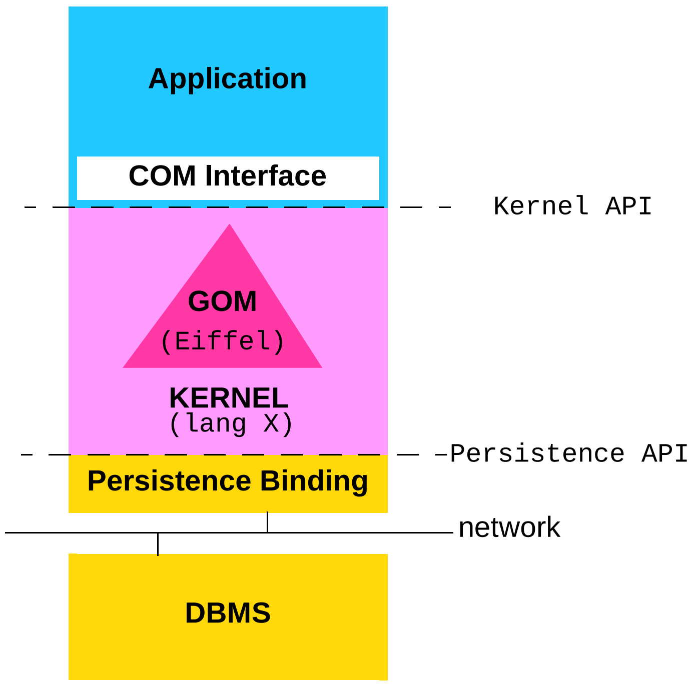

FIGURE 3 Typical Application Architecture

This typical EHR application which uses a client/server database, is constructed using a GOM kernel component. In this architecture, the kernel component creates and manipulates EHR information according to the GOM semantics, in response to calls through the kernel API interface (shown in this example as a COM interface). The view of the record seen by the application software is provided by the kernel API.

Important questions for software developers include:

- Why use a kernel component?
- How is the kernel developed from the GOM?
- How is the kernel API specified?
- How are applications developed?
- What are archetype definitions, and how are they developed?

The first of these is possibly the most important. The GEHR kernel is a component which may be used by other applications and systems for the purpose of reliably creating, storing and retrieving EHRs. Since it is reusable component, its development can be limited to a small number of organisations, rather than being reinvented at each GEHR site. There are a number of advantages to such an approach:

- More *reliable kernels* can be built, since each one will benefit from wider usage (and therefore defect detection) than numerous separate implementations of the GOM.
- A kernel approach vastly *reduces the number of implementations requiring compliance testing*, since only the kernel itself needs to be tested; it enables a more realistic (i.e. small) number of developers to take part in the standards process, thereby ensuring that software requirements are fully satisfied.
- *GEHR compliance* of an application or system comes automatically by using a GEHR-compliant kernel.
- Application *developers can ignore the intricacies of the GOM,* since they only have to comply with a kernel API.
- *Vendors do not need to be directly involved in the standards and development process* in order to ensure their software will be attractive for users: all that is required is to conform to the published software interface of the GOM kernel
- Software users, i.e. clinicians, have the freedom to use software from any GEHR-compliant vendor, rather than being locked into a single vendor. *No one vendor will own the "doctor's desktop".*

The overall effect is to raise the quality of EHR systems, and vastly increase their interoperatbility.

FIGURE 4 shows the principal lines of development of the various parts of GOM software architectures. The arrows indicate how each deliverable is derived from the previous one.

4.2 The Kernel

The kernel is developed based directly on the GOM. Its purpose is to implement the semantics of the GOM, that is, to be capable of constructing GOM EHR structures, and process them according to the GOM semantics. Further, it has to make appropriate calls to a database layer in order to store and retrieve EHRs.

see: Gamma E., Helm R., Johnson R., and Vlissides, J. - Design patterns of Reusable Object-oriented Software

The process of developing a kernel implementation of the GOM therefore involves building a completely implemented object model (class model), either as a direct extension of the GOM as expressed in Eiffel, or as an equivalent in another language. Various development approaches are possible, including direct inheritance, or the use of the so-called "bridge" pattern.

4.3 The Kernel API

The kernel API is the interface with which software developers are primarily concerned, and may be quite different from the GOM. In fact, any number of APIs are possible, and whilst each must be technically a formal mapping of the GOM, this is not the best way to understand it; rather, the API should be understood as a *functional interface* designed to make application development easy and efficient. As a consequence, the kernel API should be designed to reflect the *temporal* function call patterns, and to some extent, "convenience" functions and ready-made queries expected of applications. Since there may be different classes of applications (e.g.

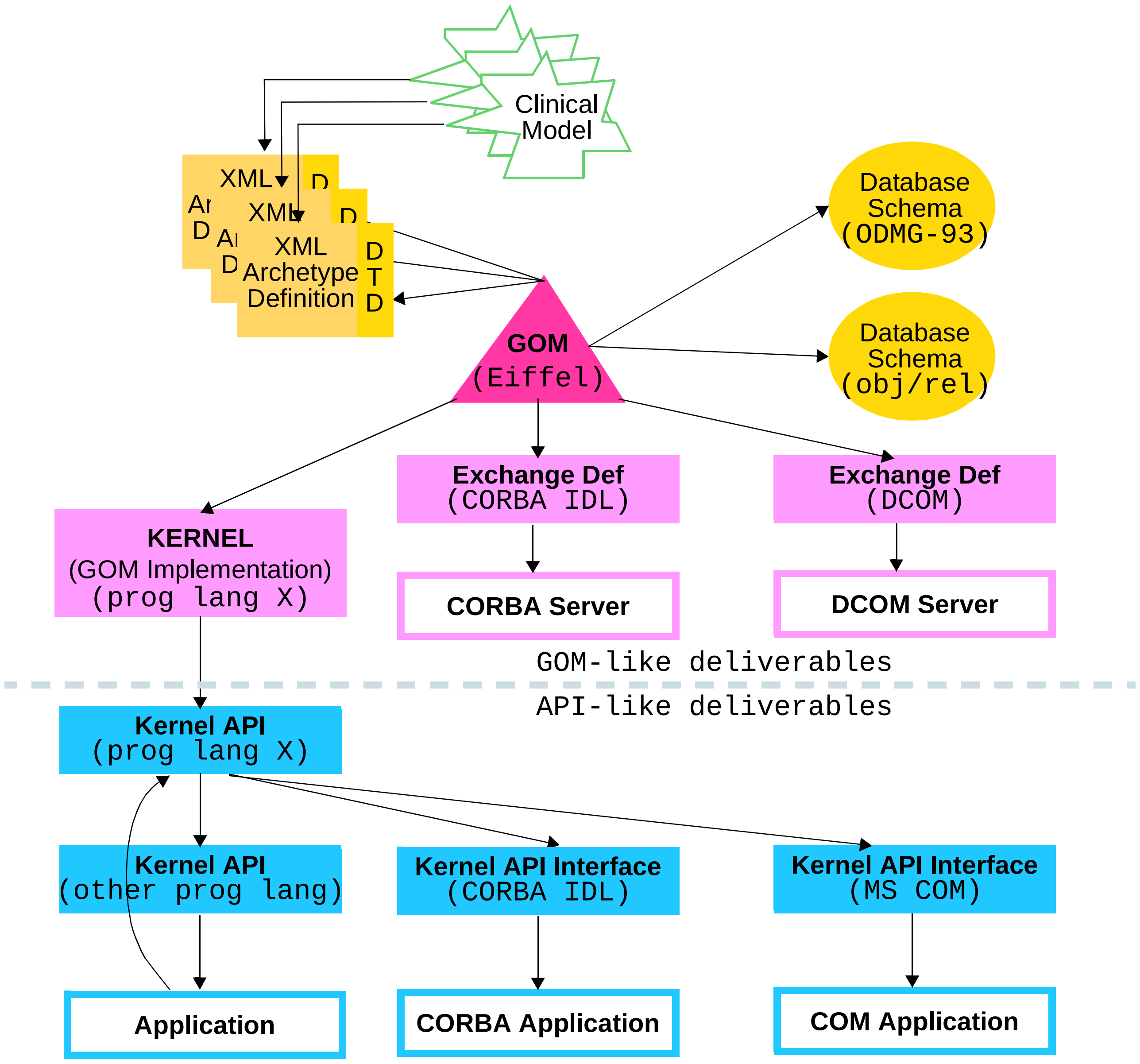

FIGURE 4 Taxonomy of GOM-derived Deliverables

an intensive care unit software application may be concerned only with immediate life signals and observed data of the patient, whereas GP software has broader concerns), more than one API may be appropriate for interfacing with a kernel.

The best general approach for a kernel API is to remain simple, and to use a “proxy” concept, that is to say: the kernel creates and modifies EHR structures *on behalf of applications*. The following characteristics should be aimed for:

- A fairly flat set of objects and member functions.
- Routines whose arguments and return types are either:
  - basic types, such as string, integer, date, or multimedia blobs, or
  - archetype definitions, expressed using XML (which is in fact just a long string).

- The kernel uses internal cursors to keep track of what it is building, and returns “paths” (unique locators) enabling the application to remember parts of the record it has created or retrieved.
- No genericity or inheritance as in the GOM, thus allowing applications to be written in non-object or partly-object languages such as Visual Basic and Delphi.

Notwithstanding the above, the API will still reflect the semantics of the GOM, by requiring functions to be called in a certain order, preconditions to be met and so on. Thus it acts as *an interpretation of the GOM* suitable for use in software development.

see: www.omg.org/home/corbamed

Published APIs on which GEHR APIs may be based include the CORBAmed COAS, PIDS and LQS interfaces, and the APIs of existing applications. The former are appropriate for use in a middleware environment as well as for local applications, however existing applications may have further requirements for using the kernel as a linked component.

4.4 Derived API Expressions

Once the kernel API has been defined, many derivations of it are possible. There are at least two reasons to do this:

- To provide the same API in a different language. For example, a native Eiffel API will usually need to be made available in C, C++ and Java.
- To make the kernel available as a binary component, for local or distributed application construction. In this case, COM or CORBA mappings of the API are used.

In both cases, the development of the API expressed in a new formalism should be very simple, and could probably be automated in some circumstances.

4.5 Archetype Definitions

see: www.w3.org for XML, XMI

Archetypes define particular clinical models in terms of the informational primitives available in the GOM. Each definition is thus created by considering a concrete clinical model, such as “blood pressure”, “kidney disease”, etc and determining how to express it using the `PROPOSITION_XXX`, `PHENOMENON_XXX` etc structures of the GOM. Archetypes need to be available as structured documents, so as to be authored outside the EHR system environment, and are thus expressed in an XML schema language; consequently, standard schema expressions for the GOM concepts will be developed once and used in all archetypes. Thus, the GOM determines the schema for archetype documents, while the content is defined by clinical models.

4.6 Applications

Software developers building new applications, or integrating existing ones, are concerned only with the published API in their chosen language, e.g. Java, or COM. Remembering that only simple data or XML crosses the API, there are no real constraints on how applications can function, other than they need to obey the requirements of the API in creating or modifying EHRs.

Distributed applications can be developed using a COM or CORBA mapping of the API.

4.7 Database Schemas

Unlike the application software, schemas for database will be driven directly from the model rather than the API, since the aim of a persistence medium is to store information structures as faithfully as possible. Therefore the same object concepts as appear in the GOM - transactions, extracts, propositions, phenomena etc - will appear in database schemas.

For an object-oriented database, the schema will be expressed in ODMG-93's Object Definition Language (ODL - a language similar to CORBA IDL) or a variant, and will be a relatively straighforward transformation of the model.

see: Cattel R.G.G. (ed.) - The Object Database Standard: ODMG-93, Release 2.0

For a relational database system, a schema needs to be devised which enables object structures to map cleanly to table structures. The best approach to use is to develop an "object/relational" schema, in which concepts like inheritance and aggregation have standard mappings, rather than the usual 3rd normal form type of schema.

4.8 Exchange Definitions

In order to reliably transmit EHRs between HCFs (Health Care Facilities), exchange software must be used. Today, this mainly follows the models of CORBA and DCOM, in which a logical structural definition is used, (similar to an object database schema), and tools are used to produce client and server "stubs", which may be integrated into the respective end of a distributed application. The definitions are written in yet another object formalism, OMG IDL for CORBA, and Microsoft IDL for DCOM.

In terms of the taxonomy of deliverables above, they may derived from the API (light blue API mappings in FIGURE 4) if they are intended to facilitate component-based application building (i.e network-aware applications able to communicate with an EHR database via a GEHR kernel server), or from the GOM itself (mauve exchange definitions in FIGURE 4), if the intention is kernel-to-kernel transmission of records in their GOM form. In the former case, the communication is of record content fragments to and from applications, while in the latter, it is of complete record extracts between servers. Both methods could be used to achieve the logical movement of a record from one place to another, but clearly, kernel-to-kernel communication is likely to be more efficient, and easier to ensure faithful reconstruction at the receiver's end. For this reason, the semanitics of the movement of EHR extracts is defined primarily in the GOM, not the API (although API functions to do this are by no means out of the question).

5 Application Architectures

5.1 Standalone (1-tier) Applications

The standalone application is suitable for prototyping, testing, and small sites. It consists of an application linked or integrated directly with the GEHR kernel, via one of the available component integration mechanisms. Typical possibilities are illustrated in FIGURE 5.

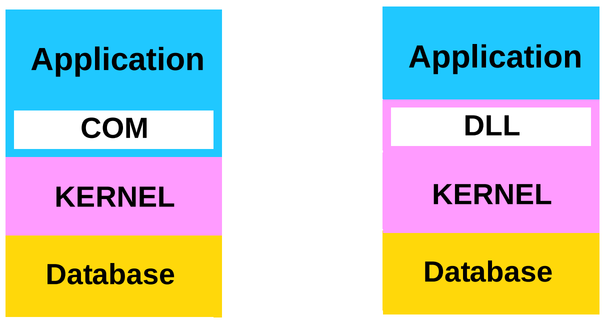

FIGURE 5 Standalone Applications

EHRs are stored in a local database, using technologies such as Btrieve, Forpro, MS Access and so on. Transfer of EHRs is achieved by saving the EHR in GEHR Exchange Format, and sending the resulting file by any available means, such as email.

Advantages

- Cheap to implement.
- Useful for prototype development
- Applications developed using this model can be migrated directly to larger environments.

Drawbacks

- Supports only a small number of users comfortably, as determined by the database used.
- Performance is likely to be an issue for large records, especially for multi-media items (assuming they can even be stored in the database) and for complex queries.
- Sharing of EHRs with other sites is more or less “manual”, and inefficient.
- Database security is likely to be a problem.
- Likely to be platform specific.

5.2 Client/Server (2-tier) Applications

A more attractive architecture, even for smaller sites is client/server, as illustrated in FIGURE 6. In this type of system, applications exist in a networked environment, and access data from a common database elsewhere on the network.

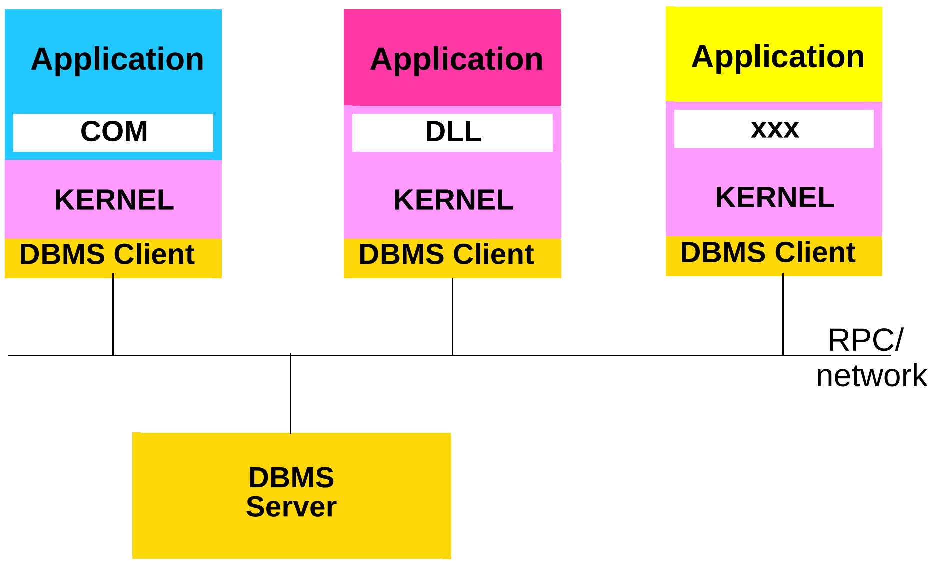

FIGURE 6 Client/Server Database Architecture

Persistence is provided by a database management system, in which persistent data is accessed via a dedicated server process(es), which take care of sessions, (some) security, journalling, transactions, prioritisation of requests, locking and many other issues. Typical examples of client/server databases include the relational, such as Oracle, Informix, Sybase, etc, and the object, such as Versant and ObjectStore.

Applications communicate with the database via a client library which is typically linked into the client application; this client talks to the DBMS server, and often provides a cache, and a separate level of transactioning. Depending on the type of database, client/server communication will be in remote procedure calls (RPC), OQL, SQL, or even straight sockets communication. HTTP may also be used as a transport layer for high-level querying protocols.

Applications in such systems are built by linking with the GEHR kernel, again using available component integration mechanisms.

Advantages

- Relatively easy to implement, given the number of available client/server DBMS products available, and vast amount of experience with them.
- Many existing HCFs will have a DBMS already.
- Most RDBMS systems have built-in or complementary means of creating visual applications quickly.
- Client/server systems are very reliable.

Drawbacks

- Difficult to scale up to wide area networks, and outside a single enterprise, due to absence of naming and resource location services.
- Security may not be strong enough, since it exists only for session login and possibly to prevent direct access to database disk volumes. Security model may not correspond to the object model seen by application users in relational systems.
- Performance may be limited by the amount of client/server traffic required for clients to obtain the objects they require. Intelligent design of the client is required to enable only sections of EHRs to be retrieved. In the GEHR case, this is available due to the OSTORE object persistence library.
- There are many other potential drawbacks in comparison with 3-tier systems, due to the tight coupling between client and server, including limited ability to mix databases, difficulties in legacy application integration, and ability to support internet clients.

Despite the drawbacks above (which are mainly relevant to enterprises looking to build large and flexible systems), the client/server approach is very appropriate for many health care facilities, including hospitals.

One of the drawbacks with RDBMS client/server systems of the past is that the monolithic construction of applications, often in a mixture of SQL, VB, and similar languages has made for low maintainability. However, the GEHR kernel prevents this, since client application developers are programming to a) the kernel interface, and b) the OSTORE interface, which is a logical persistence interface; in both cases, details of client/server transport, database locking, the network, underlying programming languages are hidden.

In many cases, a GEHR-based client/server development would be a sensible step toward a larger distributed system.

5.3 Distributed Object Applications (n-tier)

Distributed object systems (also called 3-tier or n-tier systems, due to the middleware between user applications and information sources) provide a way of getting past the limitations of traditional client/server systems, leading to systems with multiple servers, and a layer of support services for naming, resource discovery, and security. Most importantly, the information model presented to applications by the middleware is shared by the enterprise (or further) and may not relate strongly to the schemas or even type of databases behind the scenes.

An abstract model of distributed applications is illustrated in FIGURE 7, but it should be understood that this understates the likely diversity and complexity of any real-world n-tier environment.

Where does GEHR fit into this environment? The GEHR kernel provides both an API and implementation of the API; in a distributed system, the API is exported to the network, and client applications are programmed to use it in the same way as for simple and 2-tier systems. However, in this case, the distibuted object environ-

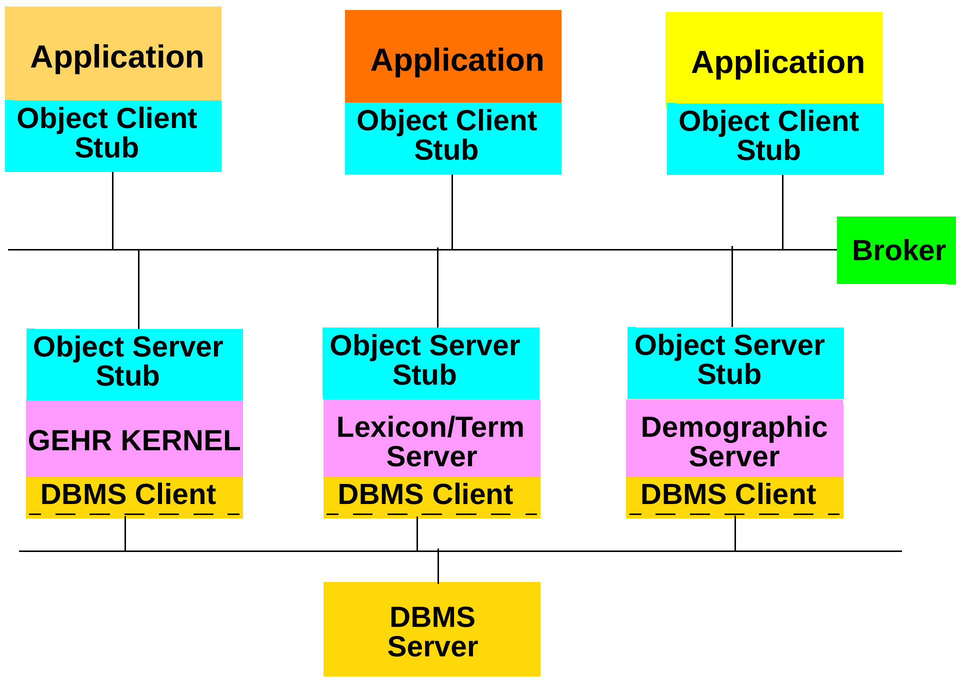

FIGURE 7 Distributed Object Architecture

ment provides services which route client requests to appropriate servers, without the client knowing the details.

In this situation, the GEHR kernel becomes the implementation of an EHR server, and its API, as before, is the programming specification for client application developers.

To Be Determined: it is expected that in the future the GEHR API will align with specifications such as CORBAmed COAS.

Advantages

- Applications are not directly tied to information servers, i.e. the GEHR kernel. Instead, both the application and the kernel respect an agreed-upon interface. This enables piece-wise development of systems, in which servers and applications can be replaced at will, without impinging on the rest of the system.
- Any combination of implementation languages and database technologies can potentially be used.
- Common interface specifications such as the CORBAmed proposals and logical protocols such as HL7 v3 can be developed publicly and agreed by a large number of stakeholders, resulting in a more representative interface design.

Drawbacks

- N-tier distributed systems involve more specification and design work, since the primary point of agreement is the distributed interface; this may delay implementation efforts.

5.4 Web-based Systems

Transport

To Be Continued:

Security

To Be Continued:

XML

To Be Continued:

6 Databases

6.1 Simple Persistence

GEHR Exchange Format (GEF)

GEF is seen as an important base-level technology solution for persistence. The idea is that rather than use a database, health records are stored as self-contained files on the normal file system, allowing them to be handled like normal files. This approach provides a default connectivity mechanism: such files may be attached to email, archived on diskettes or FTP-ed in order to transmit them to another party. All GEHR-compliant software properly implementing the GEF definition will be able to read and manipulate health records transmitted in this way.

6.2 Simple Databases

`To Be Continued:`

6.3 Object Persistence

Object persistence mechanisms (OPMs) constitute the preferred way of storing GEHR EHRs, because the model gives rise to rich content, characterised by complex structures (such as hierarichies), numerous internal cross-references, multimedia elements. Remembering that our aim is seamless systems, it is preferable that the pure object model of the GOM is translated directly to an object model of persistence, as expressed in the ODMG-93 ODL or SQL3 schema languages.

The advantages of using an object persistence mechanism include:

- The OPMs understand objects as first-class entities - there is a direct mapping between OOPL and OPM concepts of class, object, attribute. This enables OPMs to store networks of instances in a natural way, without the semantic gap of mixed object/relational systems.
- An OPM schema is generally a simple transform of software class texts (e.g. it can often be extracted nearly automatically from languages such as C++ and Eiffel). The schema thus presents a minimal maintenance task, as compared to a classical relational design, where the schema is a complex artifact in its own right, requiring its own experts and long-term maintenance.
- OPMs respond to queries, like other databases, as well as “navigational” requests, corresponding to the programming language form `obj.feature.feature...feature`.
- OPMs generally store multimedia objects, whereas this facility may be the exception with RDBMSs.

Object persistence can be implemented by object databases, by an object-relational mapping engine used over an RDBMS, or by other categories of database, such as the MUMPS system.

It is important to understand that relational databases can be used in two different ways:

1. As an object persistence mechanism, in which case there is a software layer providing an object view, and a non-classical schema design which can be extracted more or less directly from a class model such as the GOM. This might be done via the Object/Relational modelling process, for example.
2. The second way is to simply use the RDBMS in its classic form, with a 3rd-normal form schema, designed from an E/R view of the object model.

Products based on the first method can provide a very good object persistence mechanism, whereas the more naive, second approach will always be difficult, and a maintenance liability in the long term.

6.4 Object Databases

The most direct way to store objects is in an Object Database (ODB). These are typically client/server products, and most conform to a standard known as ODMG-93, created by the Object Database Management Group. This standard defines a schema definition language (ODL), a query language (OQL) and bindings for various languages.

Experience has shown ODBs to be fast, reliable, and a good fit for complex, multimedia information such as EHRs. They are also far less maintenance than relational databases.

see: www.matisse.com

The first version of the Ocean GEHR system uses a library called OSTORE, as a logical binding to the Matisse database. Bindings of OSTORE to other ODBs are being developed.

6.5 Relational Database Management Systems

Relational storage of data is still the most common persistence solution to date. While the use of relational storage with object-oriented software introduces problems, it is often preferred due to an existing investment and expertise in it.

New Database

As indicated above, the preferred approach is to use a fully object-capable relational solution, which will either mean a complete product, or an add-in layer and set of tools. In this system, the relational schema will be produced by the tools and will not resemble the standard 3rd-normal form schema used in E/R database design.

To Be Continued:

Legacy Databases

In many hospital environments, legacy database systems exist. It is often tempting to approach the task of migrating to a new EHR system by asking the question “how do we integrate to the existing database?” This may be the wrong question in many cases. Integration - i.e. forcing the new applications to use the existing database (not just the RDBMS) to store their data - should only be attempted if the following is true:

- The existing system is in fact an “EHR system” or equivalent, i.e., the data it stores largely corresponds to that of a notional EHR, in which case it is likely to correspond to the core of a model like GEHR.
- The majority of the EHR information is in one logical database, i.e. described by one schema.

If both conditions are true, a legacy approach such as the following can be used:

- Introduce new object applications and object persistence mechanism (may use same RDBMS as legacy database).
- Engineer cross-load from legacy database to OPM, e.g. overnight, or in real time as required. This is a relatively common operation (e.g. the OSTORE persistence library provides this capability).
- At this stage the new applications can only be readers of the data coming from the legacy system, via the OPM, but can of course create their own new data as well. However, some update capacity may be possible, if for example certain operations in the new system can fairly easily be turned into SQL queries for the old system (e.g. by hardwiring in query strings which are used as templates to be filled by different parameters each time).
- Engineer read and update capability in the OPM for use by legacy applications. This typically requires supporting SQL queries which are based on the legacy schema. In reality, most legacy systems will execute the same few queries all the time, and a converter can be written relatively painlessly.
- When all applications using a particular legacy system have effectively been migrated to the OPM (if not completely rewritten), the legacy database can be shutdown, and the newer applications can now have unconstrained write access to the OPM data.

Federated Systems

Many hospital databases are *not* EHR systems, but a collection of disparate specialist clinical systems (e.g. radiology, histopathology etc), a pharmacy system, an administrative system for admission and discharge, and a PMI, or patient master index. Each system has a different schema, and there may be duplicates of data from some databases used by others; in some cases, it may not even be clear who is the primary manager of such data.

Such cases represent a *federation* problem rather than a legacy database migration one: the correct approach is to introduce a distributed middleware environment (possibly via a component-based client/server approach) and devise ways of accessing the legacy data with purpose-built interfaces between each legacy system, and the middleware layer (e.g. by implementing mappings between the legacy data and a well-defined CORBA IDL interface). In federated systems, access can typically be engineered economically, but update (i.e. writing) is far harder to implement. This is because the reverse mapping from objects to unrelated relational structures is required, and is different for each target system; things are complicated by different transaction models (or no transaction support at all), variants of SQL and other details. In fact federated systems in which full writing capability has been achieved to the legacy databases have rarely been achieved, and there have been many expensive failures.

A more appropriate route for federated legacy systems is to introduce a new object EHR system which uses the legacy databases as information resources. For example, a GUI screen in the new admission application might appear partly filled out due a query to a legacy database, before the user enters any data. The final commit will cause object structures containing both legacy and new information to be written to the new object persistence mechanism.

6.6 Non-Relational DBMSs

6.6.1 MUMPS

To Be Continued:

7 System Architectures

7.1 Overview

7.1.1 Archetype Domain System

To Be Continued:

7.1.2 Terminology Server

To Be Continued:

7.1.3 Demographic Servers and Identification Systems

To Be Continued:

7.2 Small Clinic Architecture

To Be Continued:

7.3 Hospital Architecture

To Be Continued:

7.4 Distributed Architecture

To Be Continued:

A References

A.1 Health IT

1. GEHR Project - *Deliverable 19,20,24: GEHR Architecture*
   GEHR Project 30/6/1995
2. CORBAmed COAS specification V1.0 - OMG document corbamed/xx-xx-xx
   OMG Jan 2000
3. CORBAmed PIDS specificication - OMG document corbamed/98-02-29
   OMG 1999
4. The Unified Service Action Model - documentation for the clinical area of the HL7 Reference Information Model. Rev 2.4+
   Regenstrief Institute for Health Care 2000

A.2 Software Engineering

5. Meyer, Bertrand - *Object-oriented Software Construction*, 2nd Ed.
   Prentice Hall 1997
6. Walden, Kim and Nerson, Jean-Marc - *Seamless Object-oriented Software Architecture*.
   Prentice Hall 1994
7. Cattel R.G.G. (ed.) - *The Object Database Standard: ODMG-93, Release 2.0*
   Morgan Kaufmann, 1997
8. Gamma E., Helm R., Johnson R., and Vlissides, J. - *Design patterns of Re-usable Object-oriented Software*
   Addison-Wesley 1995

A.3 Technology

9. Orfali, Harkey, Edwards - The Client Server Survival Guide (3rd Ed.)
   Wiley 1999.

END OF DOCUMENT