Archetypes: Constraint-based Domain Models for Future-proof Information Systems

Thomas Beale
Deep Thought Informatics Pty, Ltd
Mooloolah, Qld, Australia
(thomas@deepthought.com.au)

OOPSLA 2002 workshop on behavioural semantics.

Abstract

Most information systems today are built using "single-level" methodologies, in which both informational and knowledge concepts are built into one level of object and data models. In domains characterised by complexity, large numbers of concepts, and/or a high rate of definitional change, systems based on such models are expensive to maintain and usually have to be replaced after a few years. However, a two-level methodology is possible, in which systems are built from information models only, and driven at runtime by knowledge-level concept definitions, or "archetypes". In this approach, systems can be built more quickly and last longer, whilst archetypes are authored directly by domain specialists, rather than IT personnel. Executed properly, the approach has the potential for creating future-proof systems and information. Work in the medical informatics domain on electronic health records (EHRs) has shown that a two-level methodology is implementable, makes for smaller systems, and empowers domain users.

Introduction

Many of today's information systems are developed in such a way that the domain concepts which the system is to process are hard-coded directly into its software and database models. While this "single-level" approach may allow for relatively quick development, the usual legacy in domains characterised by complexity, size, and high rate of concept change, is systems which are expensive to modify and extend, and consequently have a limited lifespan. In many such systems (particularly those developed in relational and/or SQL- and older programming languages) a particular shortcoming is the non-explicit (i.e. implied) expression of domain concepts, because the programming and database formalisms can express models only in the simplest data attribute terms.

With no explicit formal model available which developers can reason about or extend as needs change, the ability of software and databases to keep up with their requirements is limited. Even in more object-oriented systems, where the model is clear, a basic problem remains: the software can never be "finished", since new and changed domain concepts will always be appearing, forcing continual rebuilding, testing and re-deployment of systems. If changes are not made, the system suffers creeping obsolescence, and as a result, diminishing utility over time. (It has to be said in any case, that in many object-oriented systems, domain processes,

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

workflows, enterprise concerns and component arcitectures are not well separated, as they would be if an RM/ODP methodology were followed).

As a consequence of these shortcomings, not only do many information systems today not serve their local users well in the long term, they also exhibit limited interoperability. Typically, they are only interoperable if they (i.e. their relevant vendors or development organisations) subscribe to the same formal model of information or services, i.e., they are standardised or productised.

In many domains, both the total number of concepts and the rate of change is high; in health for example, there are thousands of constantly-changing concepts. For example, the SNOMED [35.] medical termset codes some 350,000 atomic concepts and over 1 million relationships. Change factors in medicine have been characterised by Rector [19.] as follows:

> Not only is medicine big, it is open-ended:
>
> 1. *In breadth*, because new information is always being discovered or becoming relevant
>
> 2. *In depth*, becase finer-grained detail is always being discovered or becoming relevant
>
> 3. *In complexity*, because new relationships are always being discovered or becoming relevant.

A different approach is needed, predicated on an idea of the world as a changing place, not a static one in which changes to requirements can somehow be regarded as exceptional. The approach proposed here is a rigorous knowledge-modelling one, and is founded on the basic tenet of *the separation of knowledge and information levels in information systems*. The definitions of these terms used here are as follows.

Information: statements about specific entities. For example, the statement “Gina Smith (2y) has an atrial septal defect, 1 cm x 3.5 cm” is a statement about Gina Smith, and does not apply to other people in general.

Knowledge: statements which apply to all entities of a class, e.g. the statement “the atrial septum divides the right and left atrial chambers of the human heart”, which might be found in a medical knowledge-base.

The term *archetype* is used to denote knowledge level models which define valid information structures. Archetypes serve various purposes:

- To enable users in a domain to formally express their concepts.
- To enable information systems to guide and validate user input during the creation and modification of information at runtime, guaranteeing that all information “instances” conform to domain requirements.
- To guarantee interoperability at the knowledge level, not just the data structure level.
- To provide a well-defined basis for efficient querying of complex data.

A re-conception of information system engineering based on two-level modelling is a possibility for achieving widespread, knowledge-level interoperability. This paper describes the basis for such a methodology.

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

Existing Methodologies

Before considering information system development methodologies, it is worth stating the primary purpose of an information system in the terms used here:

> the creation and processing of instances of business entities

Here, “business entities” means the types understood by the system, such as PERSON, ORDER, or PACKET; “instances” means actual occurrences of such types, usually in the form of the structured data representing a particular PERSON, ORDER or PACKET. (The term “domain concept” is also used here, to denote business entities but also more abstract entities which appear in models but do not have direct data instances.)

The Single-Level Approach

Most published information system development methodologies (e.g. the object-oriented methodologies of Booch [6.], Jacobsen [14.], Martin and Odell [17.], Rumbaugh et al [21.], Walden and Nerson [22.] and numerous derivative works, UML-related publications, and relational texts based on e.g. Date [8.]) work on the premise of a single level of “concrete” models, or what is called here a “single level” approach. That is to say, business entities are modelled directly in software and database models, via an iterative process of writing use cases, finding classes, and building models which will eventually become software. Most information systems today are constructed on this basis, which is illustrated in FIGURE 1.

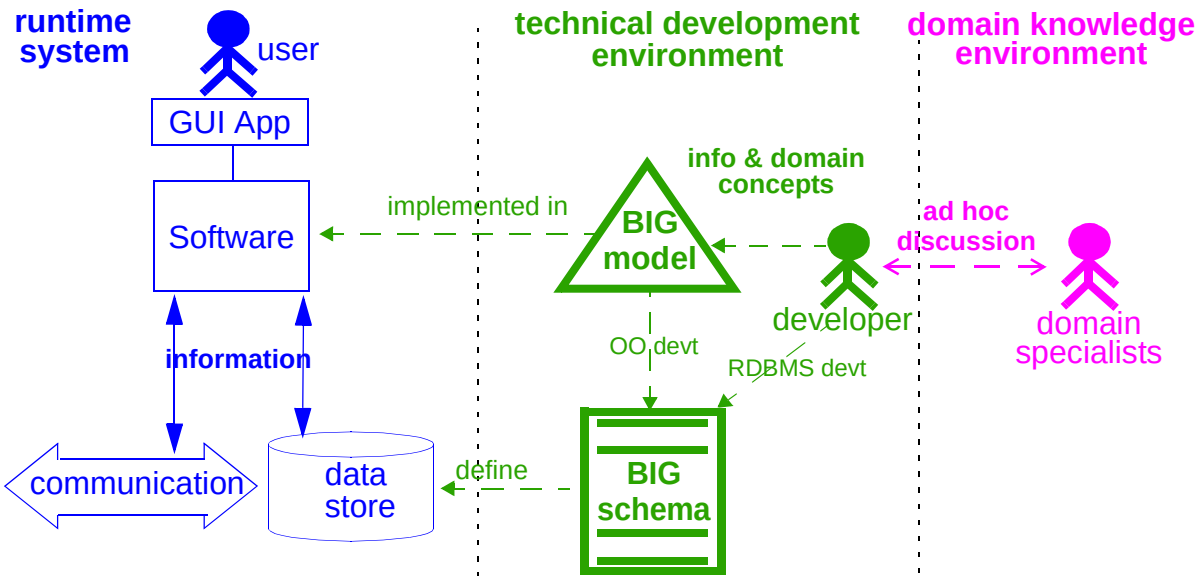

FIGURE 1 Single Level Methodologies

The system shown on the left side of the figure creates information as instances of business entities, stores it, transmits it, and formats it for human use. The database, software and graphical user interface are developed based on an object-oriented (OO) or entity-relationship (ER) model, formally describing the semantics. In typical relational developments, concepts are encoded in the relational schema and into program code or stored procedures. In object-oriented systems, they are expressed as an object model in a formalism such as UML. Many systems use a combination of approaches, with object-oriented models being implemented in software, and also translated into relational schemas, resulting in the well-known “impedence mismatch”. As a result, in a majority of object-oriented and relational systems, the semantic

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

concepts are nearly all hard-coded. For instance, the concept PERSON will be modelled with explicit attributes and relationships for name, address, sex, date of birth and so on, and each sub-concept will be recursively modelled in a similar way.

Exceptions to this approach are systems in which business specifications are well separated and then engineered as explicit (editable) rules governing the runtime behaviour of systems. Authors such as Kilov [16.], Bjørner, Simmonds, Ash and in medical informatics, Blobel [5.] have extensively treated this area, in most cases using the explicit separation of concerns of RM/ODP.

Shortcomings of the Single Level Approach

For situations where the number, complexity and rate of of definitional change of entities is small (i.e. low spatial and temporal variability) the single-level approach may well be the most economic one. However, in large, information-rich domains subject to constant change, such as clinical medicine, high-tech manufacturing and defence, single-level modelling has a negative consequences, including:

- The model encodes *only the requirements found during the current development*, along with best guesses about future ones.
- *Models containing both generic and domain concepts in the same inheritance hierarchy are problematic*: the model can be unclear since very general concepts may be mixed with very specific ones, and later specialisation of generic classes is effectively prevented.
- It is often *difficult to complete models satisfactorily*, since the number of domain concepts may be large, and ongoing requirements-gathering can lead to an explosion of domain knowledge, all of which has to be incorporated into the final model.
- There may be a *problem of semantic fitness*. It is often not possible to clearly model domain concepts directly in the classes, methods and attributes of typical object formalisms; more powerful constraint-oriented languages may be needed.
- Modelling can be *logistically difficult to manage*, due to the fact that two types of people are involved: domain specialists, and software developers. Domain specialists are forced to express their concepts in a software formalism such as UML, or else participate in *ad hoc* discussions with developers. Likewise, software developers have difficulty in dealing with numerous concepts they don't understand. The typical result is a substandard modelling process in which domain concepts are often lost or incorrectly expressed, and software which doesn't really do what users want.
- *Introduction of new concepts requires software and database changes, and typically rebuilding, testing and redeployment*, which are expensive and risky. If conversion of legacy data and/or significant downtime is also involved, the costs can become a serious problem.
- *Interoperability is difficult to achieve*, since each communicating site must continually either make its models and software compatible with the others, or else continually upgrade software converters or bridges. Heterogeneous (multi-vendor) computing environments where the software has been created using single-level methodologies typically do not interoperate well, because of the complexity of models underlying each system.

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

Even when some level of interoperability is initially achieved, it generally degrades over time, due to systems diverging from agreed common models, as they follow differing local requirements. See [20.] for a discussion of interoperability issues.

- *Standardisation is difficult to achieve*. With large domain models, it is logistically and technically (and often politically) difficult for different vendors and users to agree on a common model. Lack of standardisation not only makes interoperability difficult to achieve, it makes automated processing (such as decision support or data mining) nearly impossible, since there are almost no general assumptions such systems can make about the models underlying the queried systems.

In summary, information systems in domains with a large, complex or constantly changing number of concepts, built using single-level methodologies exhibit the following characteristics:

- large class models and database schemas;
- long-term unmaintainability and eventual obsolescence.

The Problem of Knowledge

The primary reason why single-level modelling is not a good long-term solution in domains where information is complex and volatile is because *information* and *knowledge* are conflated. Typical examples of information and related knowledge are:

- "Gina Smith has a resting BP of 110/80" (information) and "Blood Pressure consists of two quantitative data items, called 'systolic' and diastolic' (each with units = mmHg), and an optional 'protocol' indicating position of patient, instrument used and type of cuff" (knowledge);
- "`PERSON_1234`: 'David Chang', 15/Sep/1972, Male, 'Hong Kong', ..." (information) and "`PERSON` consist of name: `STRING` [1], date_of_birth: `DATE` [1], sex: `CODED_TERM` [1], place_of_birth: `STRING` [1], occupation: `STRING` [0..n], employment: `PARTY_RELATION` [0..n]...."

In the examples above, the knowledge statements hold true for all individual entities of their class. Despite this, in many domains, knowledge entities can be *volatile*. Thus, the definitions of "PERSON" and even "Blood Pressure" can change in time (particularly the former - demographic models are a prime example of volatility, where definitions of name, address, contacts, relationships etc are nearly impossible to standardise). Not all knowledge is volatile, of course - a semantic network describing basic human anatomy and physiology is likely to be quite stable. Volatility is not the only problem. The number of knowledge entities in a domain may be vast, as implied by the size of clinical terminologies such as SNOMED [35.] and UMLS [37.], each numbering in the millions of entries.

Unfortunately, the one thing we most require of an information system if it is to be economic in the long term is *stability*, the opposite of volatility. Yet, due to the use of single-level methodologies, most information systems today are not very stable, for two reasons. Firstly, because knowledge concepts are directly encoded into the models used to build the software and databases - they become the names and attributes of classes, tables and columns. Worse, every data instance created is built according to such fragile models. In other words, not only software correctness but informational validity are *directly dependent on the definitions of knowledge entities from which the system is constructed*. Changes in the definition of knowledge entities at least require that the system be rebuilt, re-tested and re-deployed, and may

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

involve expensive data migration or validation. The second factor leading to instability is the sheer size of models which reduces maintainability.

A Two Level Methodology

Overview

An alternative approach is to separate the semantics of information and knowledge into two levels of model. The former is the one familiar to most developers - the level of software object models and database schemas, denoted here by the term *reference model* (RM) - and is used to build information systems. It must be small in size, in order to be comprehensible, and contain only non-volatile concepts in order to be maintainable. The second level is the knowledge level, requiring its own formalism(s) and structure, and is where the numerous, volatile concepts of most domains are expressed. FIGURE 2 depicts such a scheme graphically. In the centre is an information system, which stores information and communicates it with other systems. The software and database models of the system are small and non-volatile, shown on the right, whilst knowledge concepts are shown in the “concept library” on the left. The formal models or languages of which such concepts are instances are included on the right side, in the technical development environment.

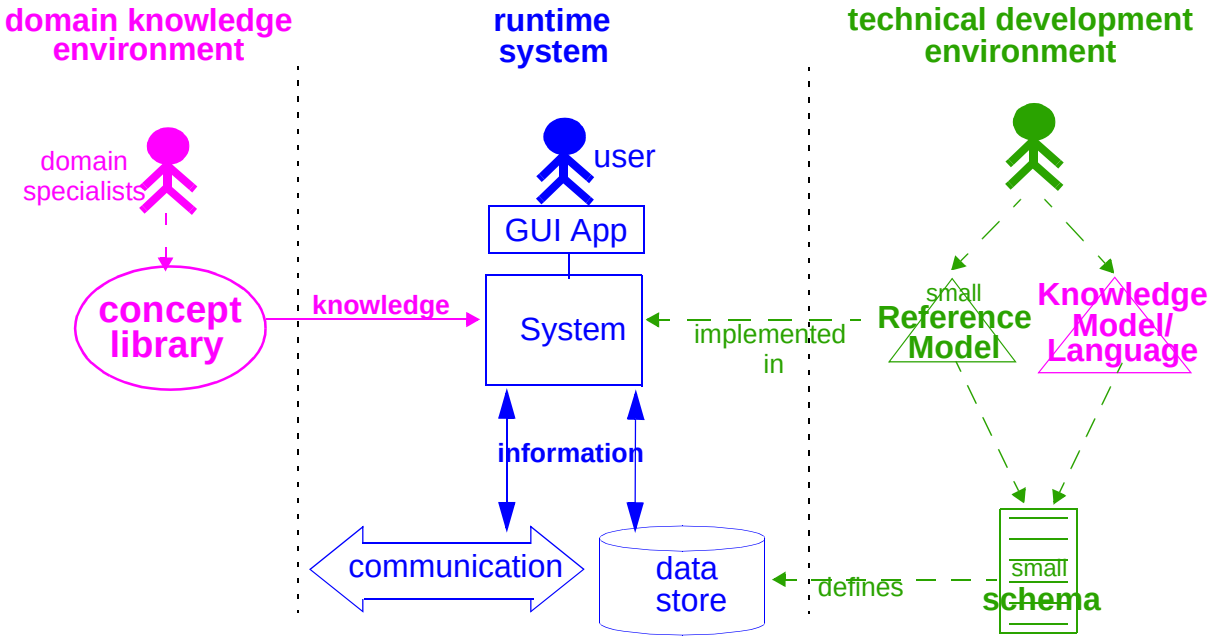

FIGURE 2 A Two-level Methodology

The key consequences of using a two-level methodology are as follows.

Future-proof Systems and Data: software, databases, and data now depend only on small, non-volatile models. They can thus be developed and deployed quickly, without waiting for the knowledge concepts to be defined, and they will rarely need to be changed. This radically changes the economics of information systems.

Domain Empowerment: technical models are developed by software engineers, whilst knowledge concept definitions are developed by the people who know about them - domain specialists. The two development processes are disengaged, and domain

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

specialists are empowered to directly produce artifacts which will control how their information systems function.

Knowledge-level Interoperability: if models at both information and knowledge levels are shared, systems may interoperate not only at the data level, but also at the concept level.

Intelligent Querying: knowledge-level models can be used to draw conclusions about the possible shape and contents of data in advance, allowing for highly efficient querying.

Executed correctly, a two-level modelling methodology stands to radically change the economics and quality of information systems. As far as is known, none of the existing software development methodologies takes a two-level approach, which is somewhat surprising, since the distinction between knowledge and information is well understood in artificial intelligence and ontology circles; the concept of "data-driven" systems is also not new. Of texts reviewed, only Fowler's "Analysis Patterns" [10.] implies two-level thinking, although its treatment of the knowledge level is *ad hoc*. Other authors who may have something to contribute include Wisse ("Metapattern" [23.]) and Forman and Danforth ("Putting Metaclasses to Work" [9.]). Previous work in Health Standardisation which foreshadows a two-level approach includes the CEN health record pre-standard ENV 13606:1999 Part 2 [24.].

The Challenges

The challenges in devising a two-level methodology include:

- knowing how to perform the separation of concepts in a domain of interest (e.g. as specified in a requirements document) into two levels;
- knowing how to structure the models at each level;
- understanding the formal relationship between the two model levels;
- understanding how to engineer systems based on the first level, but which are aware of the second level.

These are quite profound questions. While they do not have trivial answers in any particular case, the governing principles are fairly easy to grasp.

The Information Level

As stated above, the reference model should consist of a relatively small number of non-volatile concepts (i.e. classes). The first challenge is that of finding these classes. The starting point is, as in single-level modelling, an informal description of the concrete entity types which the system is to process, typically compiled during requirements capture. Consider for example, a demographic system for hospitals, which must represent the following kinds of entities:

- Person
- Health Care Facility
- Health Care Professional (a role of a person who performs care delivery)
- Health Care Agent (a role of a person, software, or device which performs clinical actions)
- Staff Member, Administrator
- Patient, Inpatient, Outpatient

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

- Surgeon, Consultant, Specialist, GP, Nurse, Intern, Student, Visitor, Family Member
- Pathology Laboratory
- Contact details, address, etc

It is not hard to see that a large concrete model could be constructed based on these concepts. However, it is also not hard to see that if the aim is to build an information system which is future-proof and interoperable, only some of the above concepts should be concretely modelled, characterised as those which are known to be valid for all instances, and constant in time. There are in fact far fewer concepts satisfying this profile: Person, Organisation, Role, and a few others, because the possible data attributes of most demographic types are very variable, even for the common attributes of name, gender, address, date of birth and so on. In general, only reasonably abstract classes will be defined in the reference model, rather than concrete business entity types.

An appropriate reference model is illustrated in FIGURE 3 (derived from the *open*EHR demographic model [4.]) and is built on only those classes, relationships and attributes which are truly non-volatile in time. Accordingly, it was judged in the analysis that all `PARTYs` would always have a legal identity, possibly other identities, roles, contacts and so on. However, only a small number of facts are concretely modelled in this way - most of the classes in this model include the attribute *details*: `COMPOUND`, which is the generic part of the model. The `COMPOUND/ELEMENT` pattern is a simple node/arc structure which allows any logical structure to be encoded (it could well have been a directed acyclic graph, to be more general). The special class `LOCATABLE` has been added to ensure that every object in an instance network has a name, making it locatable by a URL-style path. The class `DATA_VALUE` has numerous subtypes such as `TEXT`, `QUANTITY`, `DATE`, `TIME`, and so on.

The Knowledge Level

With just a reference model, the ability exists to create information which is partly structured, partly unstructured. The problem now is to define the semantics of those entities *not* included in the reference model, such as Patient and Health Care Agent. Stated in general terms, a first principle of defining models in the knowledge level is:

1. knowledge-level models define whole, distinct business entities.

Before proceeding to show how this can be done, it is worth once again recalling the primary purpose of an information system: to create and process *instances* of *business entities*. The only means available for creating instances is the reference model - all data is constructed from instances of this model and nothing else. The technical problem is therefore to show how data representing entities not concretely modelled in the reference model can be constructed.

The basic premise is fairly clear: any business entity can be represented using instances of the reference model classes, with instances of the generic part (`COMPOUND/ELEMENT` in this case) expressing further structure as required. The generic *type* attribute in most classes enables objects to be named according to what they really are (e.g. “Australian Consumer”), rather than being understood only by the class name (e.g. “ROLE”). A definition of a business entity like “Australian Consumer” (e.g. according to AS5017) could be expressed as a *template*, i.e. a predefined instance structure with some attributes filled in. A fragment of such a template is illustrated using a UML instance diagram in FIGURE 4.

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

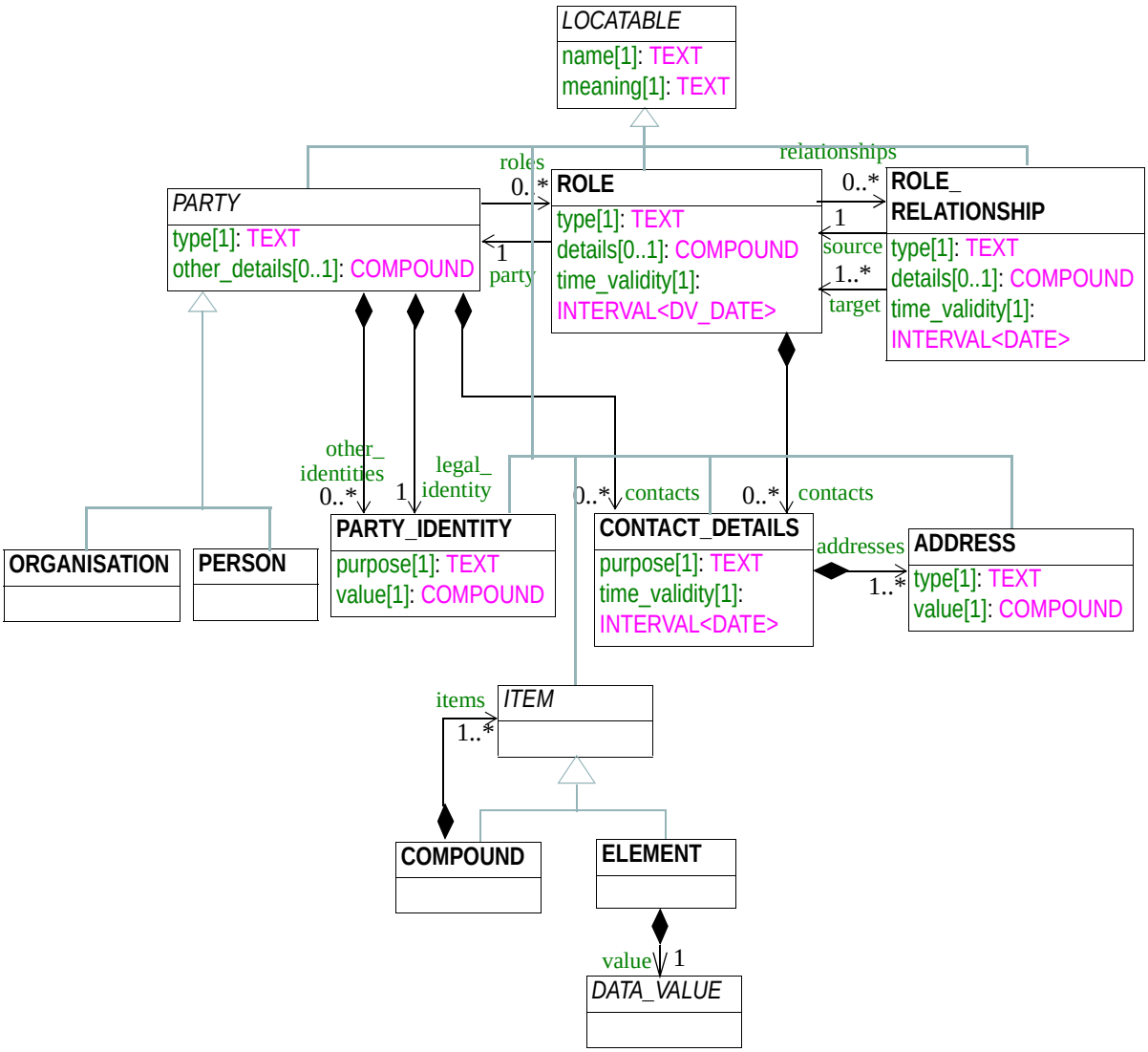

FIGURE 3 A Demographic Reference Model

The Problem of Variability

Initially it seems as if the template approach would work - it provides a default structure, names, and leaves values blank, as they should be. However there are shortcomings as soon as one considers what needs to be expressed. For a start, the model for Australian Consumers expressed in AS5017 includes cardinalities of relationships and attributes. It is also clear that the types of the *value* attributes of the `ELEMENT` objects should be stated. In some cases, it may also be necessary to make statements about names or actual values, for example that a date value is greater than a certain value, that a quantity must be in a certain range and so on. In general, the kinds of semantics needed in knowledge models are those of *constraint* - statements which say how instances of a reference model should be constrained to form a valid business entity of some kind. A basic list of required constraint semantics includes:

- constraints on names;
- constraints on types e.g. of the *value* attribute;
- constraints on structure; e.g. attribute and relationship cardinalities;
- constraints on relationships between attributes;

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

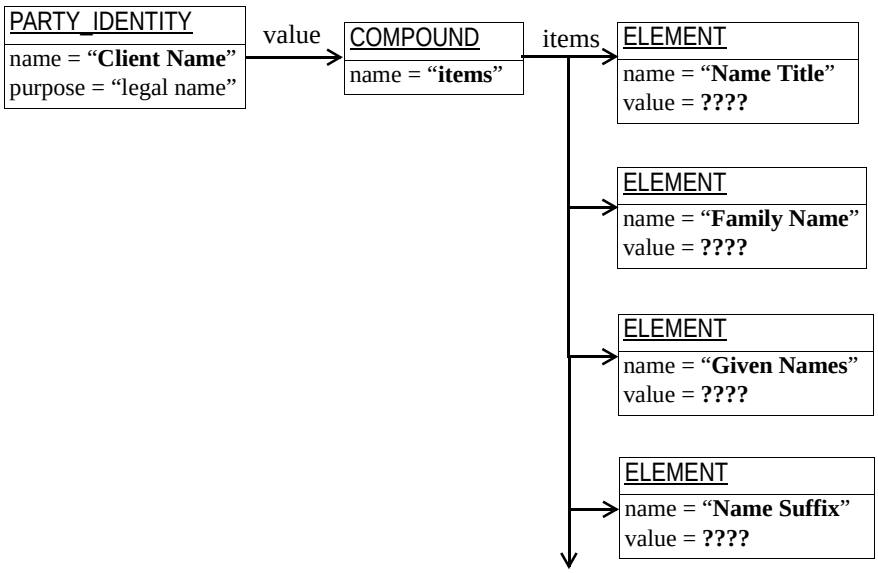

FIGURE 4 Fragment of template for Business Entity "Client Name"

- validity constraints, including invariants.

A second principle of models in the knowledge level can thus be stated as:

2. knowledge-level models express constraints on instances of an underlying reference model.

Archetypes

The name *Archetype* is introduced to denote *a model defining some domain concept, expressed using constraints on instance structures of an underlying reference model*. (The Concise Oxford Dictionary defines an archetype as "an original model, prototype or typical specimen"). The word "archetype" indicates a model with significantly more expressive power than a fixed template.

Consideration of the notion of *constraints on instances of a reference model* leads to an approach for designing a formal model of archetypes, or equivalently, a "language of archetypes". Any given archetype, such as for "Australian Consumer" is an instance of this model.

It has already been said that an archetype model/language allows statements constraining instances of a reference model to be made. One way to formalise this would be, for each class `XXX` in a reference model, to create a class `C_XXX` in the archetype model whose meaning is "constraint on objects of type `XXX`". FIGURE 5 illustrates an example; in a reference model, suppose there is a class `QUANTITY`. In the corresponding archetype model, we will create a class `C_QUANTITY`, whose instances would act as constraint statements on instances of `QUANTITY`.

An instance of `QUANTITY` is a datum such as `110 mm[Hg] (units.property = "pressure")`, as one would expect to find in a normal information system. An instance of `C_QUANTITY` has the *semantics of constraint* on instances of `QUANTITY`; such as:

- `0 <= value <= 500, units.name = "mm[Hg]"`
  {allow any pressure between 0 mm[Hg] and 500 mm[Hg]}
- `value >= 0, units.property = "length"` {allow any positive-valued length}

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

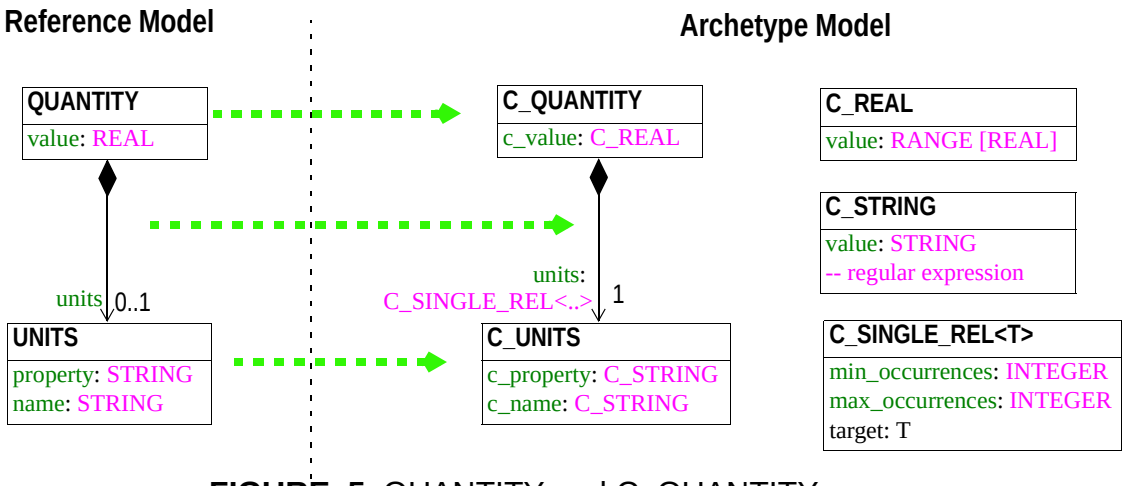

FIGURE 5 QUANTITY and C_QUANTITY

The archetype model on the right-hand side of FIGURE 5 will allow these constraints to be expressed. `C_REAL` is modelled as a range of `REAL`, allowing the range 0 - 500 to be expressed; `C_STRING` is modelled using a regular expression, enabling constraints such as `"km/h | m\.s-1 | mph"` to be expressed. (Both of these could be modelled in other ways.) The class `C_SINGLE_REL<T>` allows constraints on relationship cardinality to be expressed, in this case, whether `QUANTITY.`*units* is mandatory or optional.

An important consequence of the inbuilt variability permitted by archetypes is that more than one information instance can conform to the one archetype. In other words, the number of archetypes required to control a large number of variations in information need not be great. As an example, in the GEHR GPCG project [28.], one archetype was defined for all biochemistry results, and one for all microbiology results, even though there are thousands of different such tests in pathology.

Granularity and Composition

While this approach could be slavishly followed in order to create an entire archetype model from a reference model (such as the simple demographic one above), some account needs to be taken of the structural nature of most reference models. Reference models are constructed not just of classes, but of *groups of classes defining domain concepts*. In FIGURE 3, for example, the classes `PARTY` and its descendants are likely to be considered as defining domain concepts; `PARTY_IDENTITY`, `ROLE` etc might also be considered this way, but the classes `ITEM` and descendants, and basic types `STRING` etc do not define business entities in their own right. Instead, they are internal objects making up business objects. Another example which makes this even clearer is the kind of organisational reference model which defines concepts such as Enterprise, Division, Management Unit, Establishment - each of these will of course have numerous internal parts defining names and other details. Logically we are led to a third principle of knowledge-level models, namely:

3. The granularity and composition of a knowledge-level model corresponds to that of domain concepts in the reference model.

The consequences of this principle are that distinct archetypes defined for distinct business entities can be *composed* to form constraint definitions for aggregated business objects.

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

Ontological Analysis

In more complex domains, domain concepts fall into identifiable levels of abstraction, which can be expressed as *ontologies*. In clinical medicine for example, various ontological levels are recognised, characterised here as follows:

Principles (level 0): an ontology of the *language* and *principles*, including principles relating to subjects like anatomy, parasitology, pharmacology, biochemistry, psychology, sociology, measurement and so on. This level contains the knowledge of processes and entities which constitute the generally accepted facts of the domain - things which are true about all instances of entities (such as the human heart) or processes (such as fetal development). As such, level 0 knowledge is independent of particular users of information or processes such as health care or education; we might say it has *no point of view.* Medicine is one of the few domains to also have some domain knowledge in a computable form: it exists in 'controlled vocabularies', or 'terminologies', some of which are rich semantic nets such as SNOMED-CT [35.].

Descriptive (level 1): concepts which are expressed as structural compositions of level 0 elements. Each expresses a tightly cohesive description of an observation, analysis or prescription of something in the real world (including patient mental state). Examples of level 1 concepts whose definitions are almost universally accepted include blood pressure, body mass index, and body part measurement. Pathology also provides a myriad of level 1 concepts, such as Biochemistry results, Microbiology results, ECG results, Computerised Axial Tomography scan radiology results.

Organising (level 2): concepts created by health care professionals in an attempt to make sense of what might otherwise be a sea of unrelated descriptive items. Their form is based on the logic of enquiry and reporting used in the domain, and they act as a navigational aid to both authors and readers of information. Organising concepts are typically defined according to high-level methodological or process ideas; for example the "problem-oriented health record" gives rise to a very common hiererarchical heading structure known as the "problem/SOAP" headings.

Thematic (level 3): coarse-grained collections of information from lower ontological levels. Includes concepts such as "family history", "current medications", "therapeutic precautions", "problem list", "patient contact", "care plan" and "pregnancy".

The above breakdown particularly suits how information is recorded in electronic health records (EHRs), and has been used as the basis of developing the reference and archetype models in GEHR [26.], *open*EHR [34.], and CEN ENV:13606 [24.].

Ontological analysis is the basis of a further principle of two-level modelling:

4. The ontological levels of a domain can be used to structure both reference and knowledge level models, and are populated by instances of knowledge-level models. Composition occurs between concepts at each of the levels.

The formal consequence of this principle is that both reference models and their archetype models explicitly contain 'root classes' (i.e. classes representing distinct domain concepts) corresponding to the ontological levels of the domain. The *open*EHR EHR reference model [2.] for example includes five such levels. Accordingly, its archetype model has the same five levels, and the resulting archetypes define concepts at each level.

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

Specialisation

A final characteristic of business entities which is usually assumed is that they may be *specialised*. For example, Consultant and General Practitioner are specialisations of Health Care Professional. When such business entities are concretely modelled (in a single-level methodology), specialisation is expressed using the object-oriented inheritance relationship. At the knowledge level, this specialisation relationship is still required, but its semantics will differ from inheritance, because of the constraint nature of archetypes. One meaningful way to formally define specialisation of archetypes is as follows:

- an archetype B is a specialisation of another archetype A if the data instances which conform to archetype B also conform to archetype A.

Specialisation leads to the following principle:

5. The specialisation relationship may exist between distinct knowledge-level models.

Approaches to Defining Archetype Languages

A key technical problem to be solved is how to devise a particular (or even a general) archetype model/language for a given reference model, since with such a model available, actual archetypes can be created, and software built. One basic choice available is between syntactical and structural approaches. Structural and syntax expressions of constraints are semantically equivalent: the structural form of any set of statements which can be expressed syntactically can be thought of as the parse-tree output of a program which parses the syntax form of the statements. The difference between the approaches is that the structural version appears explicitly in the type system (hence the software), whereas a syntax expression consists of string instances only.

The Model Approach

The semantics of archetypes for a reference model can be expressed in a model form, as implied in the example in FIGURE 5. This approach has been followed in the electronic health record projects GEHR [26.] and *open*EHR [34.], with the latter introducing some syntax as well. Models are easier to develop, since they don't require the definition of a special language, but are inherently tied to the type system, and changes mean recompilation.

In the GEHR GPCG projects, the EHR reference model [1.] has an archetype model counterpart (unpublished) from which an archetype editor was built [25.]. Archetypes produced by this editor [27.], [28.] were used in runtime systems to perform data creation and validation. The SynEx project at UCL [36.] also included a reference model, a simple archetype model and an archetype editor [15.].

The Syntax Approach

Archetype semantics can be defined as a language rather than a model, which introduces more design-time complexity but may improve runtime flexibility. Nevertheless, the principles described above should be used for designing the language. Recent work on *open*EHR archetypes and HL7 "templates" is exploring the archetype language approach.

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

The Software Engineering Picture

Relationships Between Models, Information and Knowledge

It is appropriate at this point to reflect on the big picture, now that some idea of the role of archetypes and the characteristics of an archetype model/language has been provided. FIGURE 6 illustrates the two-level development universe.

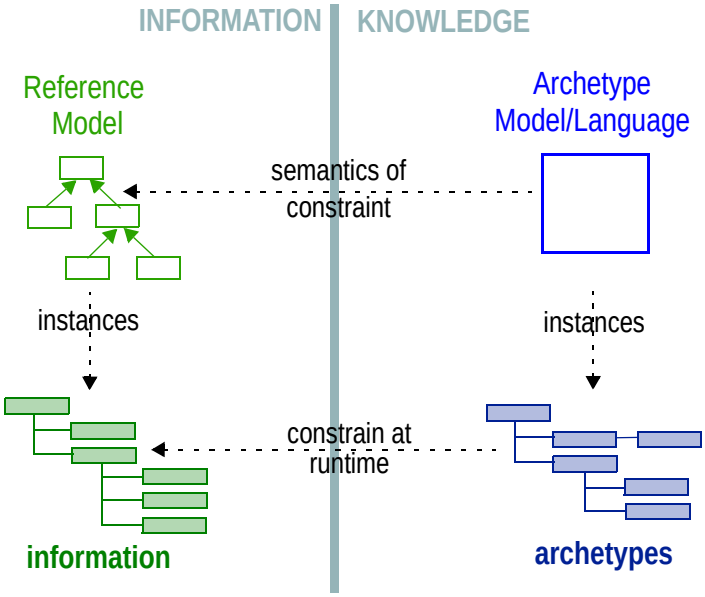

FIGURE 6 Archetype Model Meta-architecture

On the “information” side is actual information, or what many people think of as “data” - the *raison d’etre* of an information system. The structure and semantics of information is defined by a reference model; information items are “instances” of this model, as per the usual object-oriented class/object relationship. On the “knowledge” side are archetypes, i.e. knowledge-level definitions, whose job is to constrain at runtime the structure and semantics of information. However, in contrast with the class/instance relationship which constrains the “hard” aspects of object type, attribute names and types, and function signatures, archetypes constrain the “soft” aspects of information, namely multiplicities, values of attributes and naming.

Archetypes have their own model (or language), which appears in the top right of the diagram, and of which archetypes are instances. As alluded to above, this model has a formal relationship with its underlying reference model.

An Analogy

One way to understand archetypes is to imagine that the reference model defines the engineering specification of LEGO® bricks from which, as every child, and not a few adults know, anything can be built. The semantics of the RM are analogous to the “semantics” of LEGO bricks, i.e. the engineering specification of the particular coupling and joining mechanisms built into the bricks. The set of all possible combinations of a particular set of bricks comprises a vast construction space. However, most combinations are meaningless - only a tiny proportion of the space consists of the interesting constructions of houses, dogs, and tractors; all other combinations are “legal” if the bricks are connected correctly, but have no meaning to us, the users. Likewise, a reference model defines a vast informational construction space, only a small proportion of which contains combinations valid in the domain.

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

Consider further that the valid LEGO brick constructions cannot be divined from the bricks themselves: they come from fertile imaginations, or else printed plans included in LEGO packages. It is often the case that small variations and optional add-ons are suggested for the one model; this means that the set of all possible variants on the model form a constellation of brick combinations corresponding to the one plan, or model definition. Such plans are the LEGO version of archetypes.

The Relationship Between Models and Software

FIGURE 7 extends FIGURE 6 by including the generic software component types that occur in a two-level development methodology.

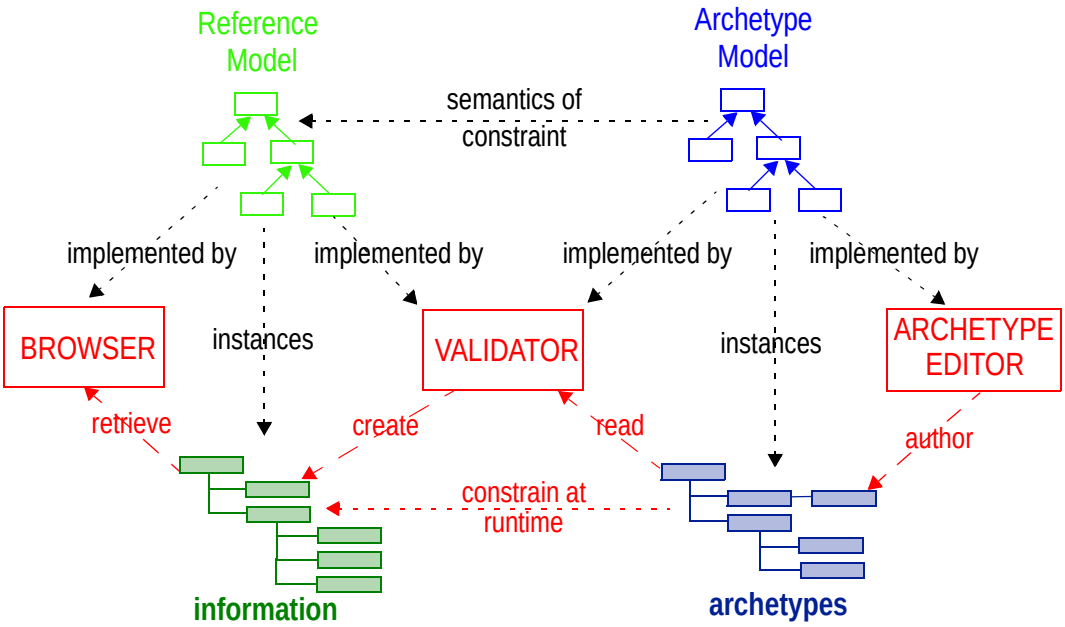

FIGURE 7 Archetype Software Meta-architecture

In this diagram, several software components are shown, being based on the reference and/or archetype models. These are:

Archetype Editor: a GUI application for creating new archetypes. This is based on the archetype class model.

Validator: any component which creates or manipulates valid data using archetypes. This is based on the reference and archetype model classes.

Browser: a generic data browser or editor can be built, based solely on a reference model, although smart browsers and editors are built using the archetype model as well.

This approach is homologous to approaches in which a formal language (e.g. object-Z) is used to write concept specifications; here the archetype model is semantically equivalent to such a language. However the strength of this approach is that archetypes are *instances* in an object-oriented system implementation: they can be created and manipulated by GUI tools, altered as desired without ever changing database schemas, or the reference or archetype models.

The constraint relationship between the reference and archetype models is a new kind of formal relationship, and has not been treated in the object-oriented literature to date. However, it

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

is not technically difficult to devise such a relationship, and it has been implemented in the GEHR [26.] and SynEx [36.] projects .

References

Publications

1. Beale T, Heard S. *The GEHR Object Model Architecture*. 1999, the GEHR project (available at http://www.gehr.org/technical/model_architecture/gehr_architecture.html).

2. Beale T, Heard S, Kalra D, Lloyd D. *The openEHR EHR Reference Model*. At http://www.openehr.org/productRM.htm.

3. Beale T, Heard S, Kalra D, Lloyd D. *The openEHR Data Types Reference Model*. At http://www.openehr.org/productDT.htm.

4. Beale T, Heard S, Kalra D, Lloyd D. *The openEHR Demographic Reference Model*. At http://www.openehr.org/productDM.htm).

5. Blobel B. Application of the component paradigm for analysis and design of advanced health system architectures. In International Journal of Medical Informatics 60 (2000) 281–301.

6. Booch G. *Object-oriented Analysis and Design*. 1994, Benjamin Cummings.

7. Chandrasekaran B, Josephson J R, Benjamins V R. *The Ontology of Tasks and Methods*. Available at http://ksi.spuds.cpsc.ucalgary.ca/KAW/KAW98/chandra/.

8. Date C J. *An Introduction to Database Systems*. Volume I. 5th Ed. 1990 Addison Wesley.

9. Forman I R, Danforth S H. *Putting Metaclases to Work*. 1999 Addison Wesley Longman.

10. Fowler M. *Analysis Patterns: Reusable Object Models*. 1997, Addison Wesley Longman.

11. Fowler M. *UML Distilled*. 2nd Ed. 2000 Addison Wesley Longman.

12. Genesereth, M. R., & Fikes, R. E. (1992). *Knowledge Interchange Format, Version 3.0 Reference Manual*. Technical Report Logic-92-1, Computer Science Department, Stanford University.

13. Gruber, Thomas R. *Toward Principles for the Design of Ontologies Used for Knowledge Sharing*. 1993, Stanford Knowledge Systems Laboratory.

14. Jacobsen I, Christerson M, Jonsson P, Overgaard G. *Object-Oriented Software Engineering*: A Use Case Driven Approach. 1992, Addison Wesley, Reading MA.

15. Kalra D, Austin A, O'Connor A, Patterson D, Lloyd D, Ingram D. *Information Architecture for a Federated Health Record Server*. In: Mennerat F. (ed) Electronic Health Records and Communication for Better Health care; 47-71. IOS Press, Amsterdam, 2002. ISBN 1 58603 253 4 .

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

16. Kilov H. *Business Specifications: The Key to Successful Software Engineering*. Prentice Hall.

17. Martin J, Odell J J. *Object-oriented Analysis and Design*. 1992, Prentice Hall, Englewood cliffs, NJ.

18. Meyer B. *Object-Oriented Software Construction*. 2nd Ed. 1997, Prentice Hall.

19. Rector, A. L. *Clinical Terminology: Why Is It So Hard?* Yearbook of Medical Informatics 2001.

20. Renner S A, Rosenthal A S, Scarano J G. *Data Interoperability: Standardization or Mediation*. 1996, IEEE. Available at http://computer.muni.cz/conferen/meta96/renner/data-interop.html.

21. Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W. *Object-oriented Modelling and Design*. 1991, Prentice Hall.

22. Walden K, Nerson J. *Seamless Object-oriented Software Architecture*. 1995, Prentice Hall.

23. Wisse P. *Metapattern: Context and Time in Information Models*. 2001 Addison-Wesley.

Resources

24. CEN TC 251 EHCRA standards. See http://www.centc251.org/.

25. DSTC GEHR Archetype Editor. See http://www.openehr.org/DSTC1.htm.

26. GEHR (Good Electronic Health Record). See http://www.gehr.org.

27. GEHR Complete list of archetypes. http://www.gehr.org/gpcg/Archetypes/List_archetypes.htm.

28. GEHR XML archetypes. http://www.gehr.org/technical/archetypes/gehr_archetypes_xml.html.

29. GEHR/OACIS data transformation project. See http://www.gehr.org/gpcg/DataTransformation.htm

30. GEHR/OACIS archetypes. See http://www.gehr.org/gpcg/OACIS/OacisArchetypes.htm.

31. GEHR (Good European Health Record). See http://www.chime.ucl.ac.uk/HealthI/GEHR/Deliverables.htm.

32. HL7 version 3. See http://www.hl7.org

33. ICD (International Classification of Diseases). See http://www.who.int/whosis/icd10/.

34. *open*EHR. See http://www.openEHR.org/, http://www.openehr.org/doculist.htm.

35. SNOMED (Systematized Nomenclature for Medicine). See http://www.snomed.org/.

36. SynEx project, UCL. See http://www.chime.ucl.ac.uk/HealthI/SynEx/.

37. UMLS (Unified Medical Language System). See http://www.nlm.nih.gov/research/umls/.

email: thomas@deepthought.com.au **web**: www.deepthought.com.au

email: thomas@deepthought.com.au **web**: www.deepthought.com.au