Mediation and Software Maintenance

Gio Wiederhold
Stanford University
September 26, 1995

Part of this report has been submitted for publication in the Proceedings of the 1995 OO-ER conference by the Springer Verlag.

Abstract:
This paper reports on recent work and directions in modern software architectures and their formal models with respect to software maintenance. Related earlier work, now entering practice, provides automatic creation of object structures for customer applications using such models and their algebra, and we will summarize that work. Our focus on maintenance intends to attack the most costly and frustrating aspect in dealing with large-scale software systems: keeping them up-to-date and responsive to user needs in changing environments.

We introduce the concept of domain-specific mediators to partition the maintenance effort. Mediators are autonomous modules which create information objects out of source data. These modules are placed into an intermediate layer, bridging clients and servers. These mediators contain knowledge required to establish and maintain services in a coherent domain. A mediated architecture can reduce the cost growth of maintenance to a near-linear function of system size, whereas current system architectures have quadratic factors.

The domain knowledge in a mediator defines the terms and relationships among the source elements and desired information. It is represented as an ontology which models the domain. These models provide the means for the maintainer to share knowledge with the customer. We sketch a conservative algebra for interoperation among these models. The customers can become involved in the maintenance of their task models without having to be familiar with the details of all the resources to be employed. These resources encompass the many kinds of databases that are becoming available on our networks. The functionality of mediators will only be touched upon and referenced within this paper. Software maintenance is sufficiently important to warrant our attention for a while.

1. Introduction

Maintenance of software amounts to about 60 to 85% of total software costs in industry. These costs are due to fixing bugs, making changes induced by changing needs of customers, by adaptation to externally imposed changes, and by changes in underlying resources [CALO:94]. Most maintenance needs are beyond control of the organization needing the maintenance, as new government regulations or corporate reorganizations, changes due to expanding databases, alterations in remote files, or updates in system services. Excluded from this percentage are actual improvements in functionality, i.e., tasks that require redesign of a program. However, in practice, minor improvements in functionality are regularily needed to keep customers. These cahnges are difficult to distinguish from other forms of maintenance. Maintenance is best characterized by being *unscheduled*, because maintenance tasks require rapid responses to keep the system alive and acceptable to the customer. In operational systems, fixing bugs, that is, errors introduced when the programs were written, is a minor component of maintenance.

Traditional software engineering tools and methods, such as specification languages, verification, and testing, only address the reduction of bugs, and hence have had little impact on long-term maintenance costs [Tracz:95]. Due to the longlevity of software, typically 15 years, most software maintenance is due to change, namely that at the time of delivery, or in subsequent years, the expectations and the environment no longer are what they were when the program was specified. Spending large efforts on specifications, to the extent that software delivery is delayed, actually increases maintenance costs, since each delay further obsoletes the design specifications.

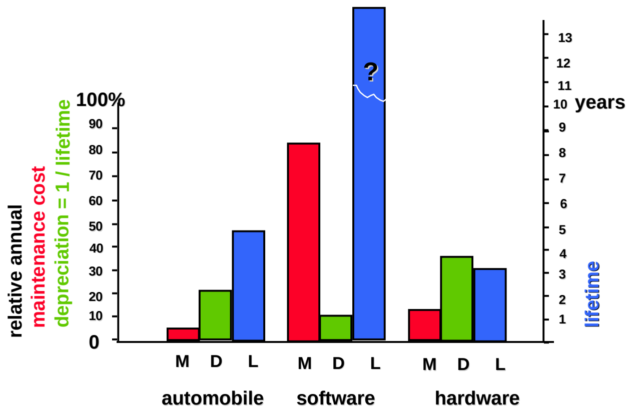

Figure 1. Maintenance is good for you

Why is software so much affected by change, apparently more so than hardware? The primary cause is that we expect software to be flexible and adaptable, while we expect hardware to be static. Maintenance of hardware is mainly performed to restore its original capabilities, and perhaps to add capacity in terms of storage or the number of users being served. Hardware is regularly replaced, preferably with equipment that is compatible with the existing software. For any system component where change is expected, we choose a software solution. If we are unsure about the use or eventual configuration or uitilization level of a system we maximize the software portion as well. The essence of software is its presumed ability to accomodate change, so that maintenance is a positive attribute of software, rather than a negative attribute to be avoided. Figure 1 aims to make that point.

Once maintenance is seen as a positive feature of software it becomes clear that we should invest to make software convenient to maintain, and not make software changes difficult by insisting on rigid adherence to specifications. However, maintenance must be planned for, so that it does not come as a surprise and a distraction. In long-lived systems most bugs are actually introduced during maintenance, making the issue of establishing maintainable architectures and models yet more important.

1.1 Maintenance cost

In large, multi-user systems maintenance costs are especially high because of two factors. First, the number of change requests increases in proportion to the number of types of users. Secondly, the cost of maintenance includes substantial efforts outside of the core application area. A change request to one module requires interaction with the owners of all other modules and finally orchestrating a changeover. These costs are universally much higher than the core cost of the requested change. The product of these two factors leads to an order squared cost in terms of system size for maintenance, as expanded in Section 3.2

A second order effect drives the cost of system maintenance yet higher. Since system changes become traumatic if they occur too frequently, all incremental requests are batched into periodic updates and performed perhaps twice a year. Composing and integrating the batch will take three months. Batching reduces responsiveness to the customer: it will take on the average at least a half year to have anything fixed. In practice responsiveness is much worse. Batching also increases the risk and the cost of failure, since some apparently independent changes introduced simultaneously may have interactions and these interactions will affect unsuspicious users. If the seriousness of the interaction errors requires a rollback to an earlier global version of the system, even users not affected by the failure will have to rollback their procedures.

1.2 Architectural effects

Taking these factors into account, it is not surprising that many data-processing organizations find that they lack the resources to advance the functionality of their systems. They may be bypassed by individual stand-alone systems, built by the customers themselves to gain the needed functionality. These systems will soon demand access to resources as corporate databases. *Open system* architectures respond to these demands and initiated the trend towards client-server architectures, now in full swing. Given stable and comprehensive servers, client applications can be rapidly constructed. However, the client-server architecture also fails to address the maintenance requirements outlined above. If a new client or a revised client application induces a need for a change in a server, then all other clients that use this server must be inspected and perhaps altered, as sketched in Figure 2. The problem has not changed, but might be more difficult now, since organizationally many clients are autonomous, and likely remote. In Example 1 we provide a simple problem case, and also its solution in a mediated architecture.

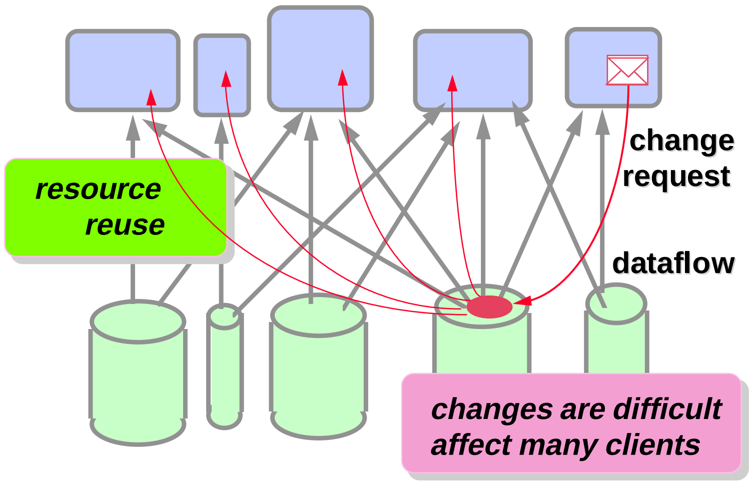

Figure 2. Client server model and the path of a client change request.

The mediated architecture uses an intermediate layer to provide isolation of one user application from other applications, even though resources are shared [ASK:95]. The use of formal, that is manipulable models, based on Entity-Relationship (E-R) concepts to create application objects, permits rapid regeneration of linkages to those resources. Figure 3 sketches modules in the three layers, and their linkages. An application can call on multiple mediators and each mediator, in turn, can invoke multiple resources, including other mediators.

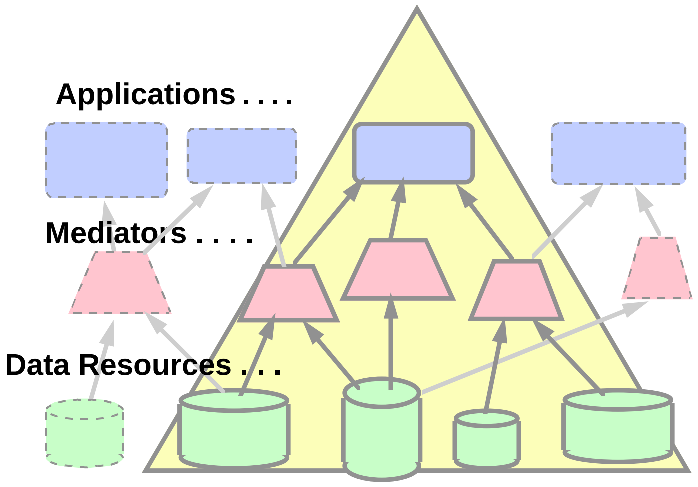

Figure 3. Modules in a mediated architecture and the scope of an application.

Example 1: A change mandated by scientific progress.

In a hospital many subsystems rely on information from the clinical laboratories. In addition to the treating physician's record, there will be charts to track progress, analysis for the quality of care, watching for iatrogenic incidents, information for billing, and evaluation of interaction with currently prescribed drugs. Recently the distinction of low-density and high-density cholesterol has become important. The clinic which surveys patients for cardiovascular risks needs these two values, instead of total cholesterol, and the server in the clinical laboratory is prepared to deliver such data. Without a mediator module every other client using cholesterol data will have to be changed. A synchronous switch has to be scheduled, and all clients are warned to have updated application programs ready at the cut-over date.

In a mediated environment the change in the laboratory server is only synchronized with the diagnosis support mediator and other mediators using laboratory data. That number will always be much smaller than the number of clients, as assessed in Section 2.4. Those mediators will sum the cholesterol values obtained from the laboratory and deliver the sum to the clients. As sketched in Figure 4, a new version of the mediator serving the cardiovascular clinic is created which delivers both values, and the clinic can switch over at any time. If the changes work reliably for their originators, any other clinic can switch over when it deems best. Since there is a cost to maintaining many versions of a mediator, there should be some inducement for all customers to convert, perhaps by increasing the price of obsolecent information over time. Figure 8 illustrates the concept.

For this particular change, several alternatives exist for its resolution, but these increase the complexity of the laboratory system and not deal with the case where the diagnosis support mediator also obtains information from other sources, as the medical record, say, to track the significance of any changes in the patients' cholesterol. This example should have motivated the maintenance-derived critera for mediation.

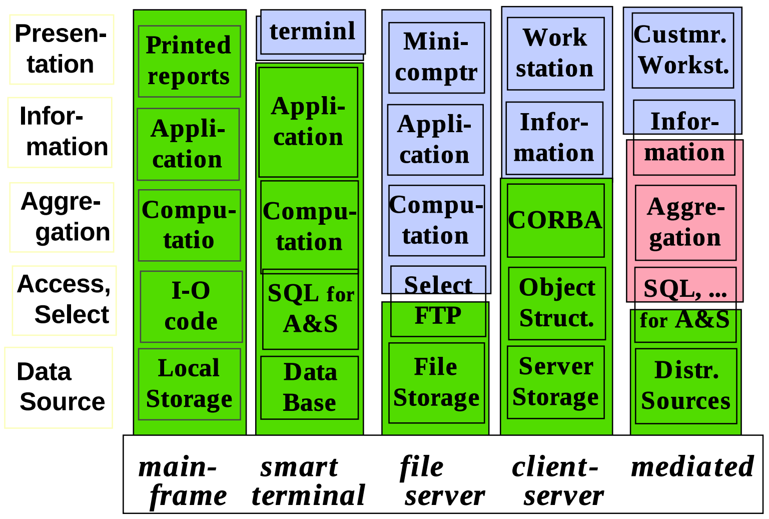

Figure 4. Architectural Evolution of Information Systems

2. The Mediated Architecture

A mediated architecture is a departure from several prior architectures, which have implemented variations of two-layer separations for the last 30 years, as indicated in Figure 4. The boundary has moved up and down, as the capabilities of the hardware components evolved. Mediation inserts a third layer; its function is best described starting from a client-server model, as shown in Figure 5. In a client-server model the servers provide data resources, often derived from databases or perhaps object-bases. The clients autonomously access these resources, select data, and load them into their workstations for further processing.

The architectural relationship in the client-server model is $n : m$ from clients to resources. This squared relationship leads to the steep growth curve for maintenance, as recognized in Section 1. Another problem in the client-server approach is an impedance mismatch between specific customer needs and general servers. That problem, and a solution will be addressed in Section 4. Our PENGUIN solution also helps in managing the internal maintenance of a mediator, but we first focus on the overall mediating architecture of Figure 3.

In a mediated architecture a layer of modules, *mediators*, is inserted between the client and server [W:92C]. Mediators provide more than just a *thick* interface; they provide significant functionality, transforming the data provided by a server to information needed by an application. Such transformations require knowledge, as knowing where the data are, specifications about the data representations, as well as an understanding about their level of detail versus the users' conceptual expectations [W:92I].

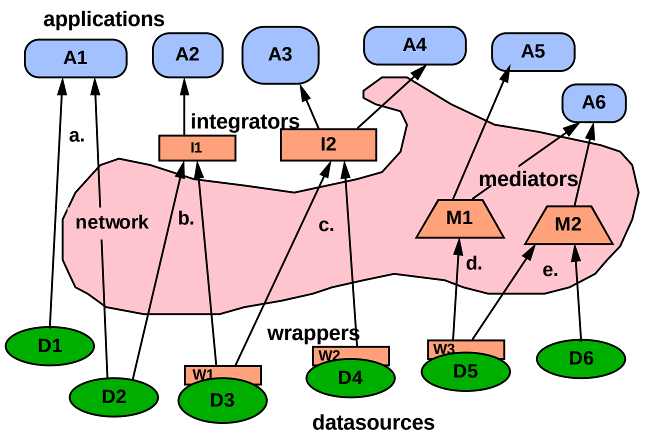

Figure 5. Mediator Evolution

2.1 Functions in mediation

Making computers understand and bridge the gap from server to customer may seem hard, but the job can be decomposed and executed in a logical flow of data from servers to clients. The resulting tasks select and access the relevant servers, integrate related data obtained from those servers, summarize, filter, and structure the result to match the customer's request [WG:95]. In a client-server architecture these tasks are primarily performed by client software. The mediator now acts as an information server with respect to the client module. The client software is still responsible for integrating information from multiple domain mediators and the user interface [ACHK:93]. Several ancillary tasks may be needed during mediation as well, for instance, processing to bring data from distinct servers to the same level of abstraction prior to integration, and resolving semantic mismatches [DeMichiel:89]. Once the job of mediation is decomposed, all of the tasks become manageable. The needed functions cited have all been demonstrated in other contexts. The contribution of the mediator concept is essentially the recognition and extraction of these tasks into an architecture that reduces their complexity. *Only simple systems work.* Figure 6 illustrates the task assignment within a mediator and Example 2 expands the diagnostic mediator.

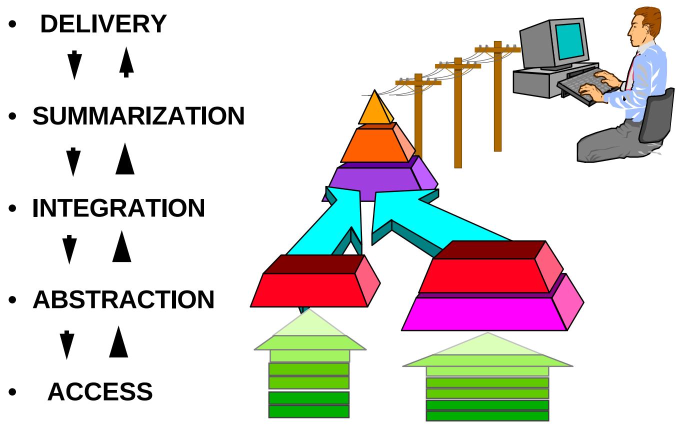

Figure 6. Task and Flow in a Mediator

2.2 Multi-level mediation

In large systems it is likely that there will be mediators using other mediators as their information sources. Such configurations are feasible, and are already in operation in some instances. However, every additional interface carries a performance penalty. Unless the added value provided by mediation is adequate to overcome that cost, it is wise to be very careful in adding levels within the mediator layer.

The architectural maintenance benefits of mediation are largely achieved within a single layer. Only when these mediators themselves become too large for effective maintenance by a coherent group of individuals is a further partitioning desirable. Committees are great to achieve compromise when there is a lack of coherence, but they are not effective for software maintenance. Coherence is defined by having a single ontology, as described in Section 3. In Section 3.3 we also cite a two-level example to cope with a larger domains: a `lipid-research` mediator serving the `diagnostic` mediator.

2.3 Building mediators

Mediators can be hand-crafted, that is, built using available programming languages, as C++, or can use artificial intelligence techniques, where the operations are presented as rules. Rules can be used to compose the primitive operations in a mediator, and we make further gains in maintenance. Today most mediators are hand-crafted, and the gains in maintenance derive solely from the architecture.

The programmer writing a mediator will interact closely with domain experts, to assure that the models are accurate, that the functionality of the mediator is appropriate, and that the results are obvious to the client. Tools to build mediators are now being developed [Lehrer:94], so that the task in time can devolve upon individuals who are primarily domain experts, rather than programming experts. Crucial in mediation are interfaces which allow their composition. Interfaces from the mediator to the resources can be derived from client-server standards, as SQL, CORBA, DCE, OPENDOC, and OLE. Interfaces supplied by a mediator to the client need better facilities for multi-tasking, asynchrony, and task-sensitive representation, as well as provision for meta-information.

Example 2. Functions in the diagnostic mediator:

Information output: Current cholesterol level, trend lines, warning flag, ...

produced by:

Exception flagging: Provide meta-information to trigger a warning when cholesterol level has increased more than 5% per month.

Summarization: Compute trend line parameters from past medical record and recent data.

Integration: Combine past medical record data with current observations and standards for patient's in similar age/gender groups.

Representation change: Convert laboratory findings to UGS values.

Abstraction: Convert irregular observations to periodic (monthly) values.

Selection: Extract medical record data from past sites for current patient.

Search: Determine past treatment sites by navigation from current record.

Within the ARPA Knowledge-Sharing Initiative [FCGB:91] a structure has evolved consisting of a Knowledge Query and Manipulation Language (KQML), which defines a transport layer [FFMM:94], and a Knowledge Interchange Formalism (KIF) [GK:94], which is a vehicle for transmitting first-order-logic rules. KQML specifies the operation or *performative*, the destination type, the vocabulary or the *ontology*, and the representation. KQML operations provide a reference handle, so that multiple transactions can overlap, and full asynchrony is implied, unless explicit constraints are specified [W:89]. While SQL has `select` as its only operation, KQML performatives extend, beyond the equivalent (`ask`, to `tell`, `infer`, `subscribe`, `advertise`, etc., in an open-ended syntax. Destination can be indirect, allowing multiple mediators to bid on supplying the information service. In addition to the KIF represenation for rules, representations transmitted in KQML have included object structures, dynamic equations, text, and tuples [CEFGGMTW:93].

Having a common, representation-independent interface protocol simplifies system maintenance, since otherwise a distinct interface is required for every data representation. Most of these interfaces differ in their approach to communication management in obvious and in subtle ways. In terms of the ISO networking model, KQML lives at the highest level and has used transport mechanisms ranging from TCP/IP to email [EIT:94].

The mediator, acting as a server to its clients, does not provide a human-friendly interface. It provides the requested information and some meta-information, which can be used by the client to understand and display the information. Mediators, not having internal persistent databases nor extensive graphical user interfaces, can be kept small, further enhancing their maintainability.

3. Partitioning by Domains

In Section 2 we focused on the layering and the data-to-information processing flow that connects these horizontal layers. We now address the vertical partitioning, based on domains. Vertical or domain-specific or partitioning is crucial to maintenance, otherwise the middle layer will become the bottleneck.

3.1 Domain-specificity

As indicated earlier, mediators are domain-specific. This means that many mediators will inhabit the middle layer, each focused on one domain. A domain is often determined by existing organizational assignments. A financial mediator will be owned by the chief financial officer (CFO) of an organization, the diagnostic mediator by the chief pathologist, the drug mediator by the pharmacist, etc. Guidance to domain expertise may be obtained from their professional organizations, just as now professional organizations may proscribe minimal schemas for the collection of data in their field [McShane:79]. It is not a major step to move from data formats to the rules for interpretation. Technically such a move is supported by object-oriented approaches, where data structures are augmented by programmed, encapsulated methods to assure their consistent interpretation [Wegner:90].

More formally, a domain is defined by having a coherent *ontology*. An ontology is comprised of the terms or vocabulary understood in the domain and the relationships among the terms. The relationships define terms as defining sub-or-superclasses, synonyms or antonyms, and attributes that are implied, complementary, or exclusive [Gruber:91]. An ontology can be viewed as a knowledge-based equivalent of the E-R structure of a supporting database. Since terms deal with concepts that are unique, rather than being entity designators, they are not immediately mapped to sets of instances. At the same time, an ontology will have a much larger set of terms. We review ontologies and a proposed algebra over ontologies in Section 3.4, but deal first with the maintenance issue.

3.2 Partitioned Maintenance

Figure 3 illustrated the mediated architecture. In Figure 7 we indicate how changes are handled in that architecture. We take the case of having a new or revised application, which demands new services from the data resources. The existing mediator in that path serves multiple applications. To accomodate the new application that mediator is revised to obtain and present the new data. If the data resources will no longer supply the old data, then the old mediator has to be adapted to supply surrogate information. In our cholesterol case this is simple; high-density and low-density cholesterol data are summed and reported as cholesterol to the old applications. The models for the two versions differ little.

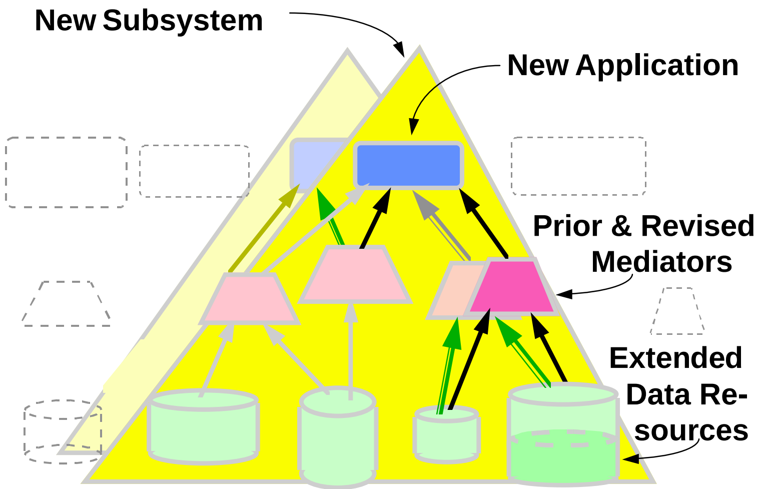

Figure 7. Changes induced by a client request in a mediated architecture.

All the required changes can be made by a domain specialist, in charge of the lipids mediator (seen in Figure 16). The cost of the change is linear, and the changeover can be scheduled at the convenience of the the new application alone. Any problems arising in the change are easily undone, only affecting the mediators and the new application. As other applications require specificity in cholesterol data, they can start using the new, proven mediator. The old version can be retired when all customers become up-to-date. The cost of the change is $\mathcal{O}(3)$, linear in the number of modules involved in the modernization.

Compare this cost with the cost of making changes in the traditional federated or client-server architecture, illustrated in Figure 2. We recognize 7 phases.

1 The application designer has to remind the resource provider to determine which other applications will be affected by the proposed change. In an open systems architecture no records of resource utilization need to be kept, so it is wise to communicate to all customer applications and await their responses. Some customers may not be aware that their applications depend on cholesterol values. For instance, there is an interaction between prednisone administration and cholesterol, but the typical clinician in an immunology practice will not be concerned with that interaction, since the patients in that clinic have more pressing concerns than cholesterol levels.

2 Agreements have to be worked out about the change with all identified applications. Some applications may be induced to take the updated now; others will want to compute a surrogate value. A mutually satisfactory change-over time has to be negotiated.

3 To validate that the changes will work correctly and not bring down the affcted existing applications testing must be performed. For ongoing operations the changes must be testing outside of the operational setting, using scaffolding code and data. Replicating the environment adequately is costly and distracting, but it is risky and unacceptable to have failures that affect many customers beyond the requestor.

4 Since the system change will affect ongoing operations, it will be scheduled a few months in advance. In that time-frame other changes will be requested within the overall system. All of these changes will be batched to occur at the same time.

5 The aggregated revision becomes a major task, requiring meetings of all parties to minimize problems due to possible interactions. An example is the massive orchestrated change from Microsoft Windows 3.1 to Windows 95, wich, when analyzed, is essentially an aggregation of many minor changes with their underlying effects.

6 During the changeover, all involved programmers must be available to participate in the effort. They must deal both with changes in their application and domain, and with the changes induced by others. They must also communicate with their customers about the effects of the changes.

7 Any failure in any of the batched changes will either require a rollback of all changes and a return to phase 3, or disable some customer applications for some time.

8 Changes which failed, or did not make it into the batch are now candidates for the next revision. It is unlikley that a new revsion can be scheduled in less than three months. All benefits that the changes are due to bring are delayed by the cyclic nature of batched system revisions.

The costs of traditional maintenance are dominated by the cost of inter-module interactions. These are then order $\mathcal{O}(m^2)$, where m is the number of modules. The mediated architecture will have more modules, at least one for every domain, but the cost of maintenance is $\mathcal{O}(\text{path length})$, or $\mathcal{O}(3)$ for systems with a single layer of mediation, and linearly more for more complex mediated systems. For all but small systems mediated maintenance will be less costly.

3.3 Ontologies and Maintenance

The vertical partitioning, which enables economical maintenance, is based on avoiding the costs incurred when dealing with problems and people that have different views and objectives. To have software that performs to expectations, it is obviously important that the people involved with its design and use understand each other. In a formal sense, this means that they share a domain ontology. Since in large systems there must be a large degree of specialization, there will be multiple ontologies. Within each specialized domain some autonomy is needed to deal with evolving concepts.

For instance, referring to Example 1, the detailed understanding of the interaction of cholesterol levels with diet is still evolving, and any new research finding will add terms and refine the meaning of prior terms. At the same time, research in cardiology is refining knowledge about the effect of cholesterol on heart disease. By keeping the domains distinct it is not necessary that the diagnostic mediator and the lipid research mediator have to share all of their ontologies. Joint maintenance would require a committee of cardiologists and lipid researchers and become a tedious process.

The function of a committee is to develop compromises, but reliable software services are not constructed on top of compromises, but based on dealing correctly with a gamut of details. The argument of our example becomes even more obvious when we consider broader domains, as finance, inventory, and the like. Here committees are even less likely to be constructive. When a committee to define some software has to include people that differ in authority or their outlook, the result can be disastrous; just envisage the process of designing software by a committee composed of physicians and nurses, or a committee made up of physicians and lawyers.

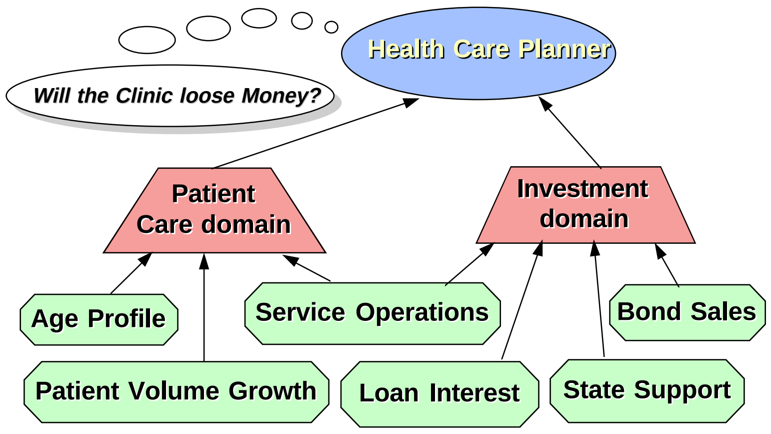

Figure 8. An application accessing multiple domains

3.4 Domain interoperation

In our systems mediators will have to cooperate. This, in turn, implies an interaction among ontologies. To keep the interaction minimal and give each domain expert as much autonomy as possible, we define an intersection operation among domains [W:94N]. This operation is rule-based, and creates a new, shared sub-ontology with only the terms that are needed to interoperate.

We use a conservative assumption, namely that terms from distinct domains *never* refer to the same object-class unless the rules state explicitly that there is an identity or a matching procedure. Such *matching rules* form a knowledge-base to be managed by collaborators or their designate from the interacting domains [W:94F]. Now no restrictions are imposed on the evolution of local terms within a domain. Figure 8 provides an example of a customer accessing multiple distinct domains.

Terms that are covered by matching rules form a new, second layer abstract ontology. The proposed knowledge-based ontology algebra also includes Union, Difference, and a Match operator. For instance, several subontologies defined by shared intersections can be merged to form a higher abstract layer. No ontological layer should be too large, i.e., contain too many terms, so that coherence is hard to achieve.

3.5 Operations of the algebra

Given multiple domain-specific ontologies and rules that define their interaction we can summarize the function of the domain algebra as shown in Figure 9. The *DKB* label on the operations identifies the knowledge base that contains the matching rules.

Operation	symbol	semantics
DKB-Intersection	$\bigcap(DKB)$	create a new subset ontology, comprised of sharable entries
DKB-Union	$\bigcup(DKB)$	create a new joint ontology, merge entries, append a source ontology identifier
DKB-Difference	$-(DKB)$	create a distinct ontology remove shared entries

Simple negation is avoided, so that no infinite ontologies are created.

Figure 9. An Algebra for Managing Domain Ontologies.

The relative autonomy of the local terms provides the scalability essential to large system maintenance. We have seen how a layered, hierarchical naming structure provided scalability and autonomous growth in the Internet [Kahn:87]. The Internet has managed growth well. During the last three months of 1994 one new computer was added to the Internet every second, without affecting more than local participants. However, the mediated approach is not strictly hierarchical since underlying data resources may be shared among mediators. In Figure 8 the patient services documented in the medical record provides the basis for some of the financial information as well as data for the medical aspect of the clinic operation. The intersections of some ontologies is sketched in Figure 10 for another domain set.

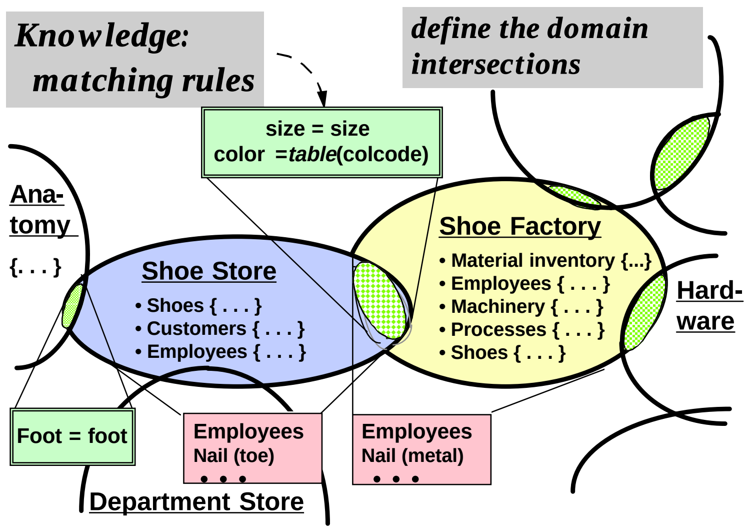

Figure 10. Domain Intersections

3.6 Using the shared sub-ontologies

Once an intersection has been computed it can be used for the integration of information. Lower level data have to be processed within their domain, say the cost of individual treatments. Once the data are aggregated to the information level in the mediator, they can be shared by the client.

The intersections provide a basis for interrogating multiple databases which are semantically disjoint, but where a shared knowledge-base has been established. This process mirrors the approach used in CARNOT, where a knowledge base is used to create *articulation axioms* for joining of data [CHS:91]. However, CARNOT's single knowledge base uses the default assumption that everything matches. When CARNOT uses a large and broad CYC knowledge base, many irrelevant retrievals can occur, so that in practice, CARNOT applications limit the depth of search.

With the conservative assumptions embedded in the *DKB-model*, the risk is that too little information will be retrieved. By letting domain experts create and maintain the matching rules, we expect that high quality operations over data from distinct, but overlapping, domains can be maintained at a reasonable cost. To evolve these systems effectively, feedback loops must exist that permit users to suggest new candidate matching rules, or to modify existing ones. Having small, distributed groups to maintain the partitioned *DKB-model*s will help ensure responsive maintenance of the domain knowledge.

4. The Structural Model and Object Generation

Mediators should deliver information in a format that matches the customer's task model. Such a service is then semantically friendly, a deeper aspect of user-friendliness than having graphics, windows, point-and-click or drag-and drop interaction. Object-oriented systems provide information in a format which is organized to satisfy a specific programmed task; object-oriented approaches are in fact an outgrowth of programming concepts as abstract data types (ADT) [Liskov:75].

Servers have a problem satisfying the customers' desire for objects, since they must try to serve a variety of customers, and different customers need different object configurations. For instance, the pharmacist needs to be able to survey all patients receiving a certain drug, but a physician will want to know about all drugs one patient is receiving. We denote the relationship from root to dependent data as `R` $\longrightarrow\!\!\ast$ `D` and show the conflicting models in Figure 11.

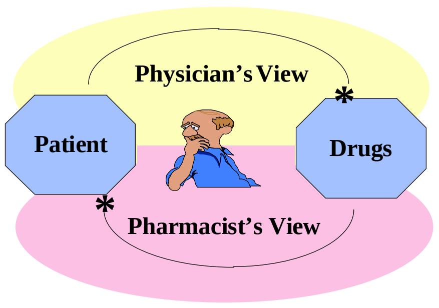

Figure 11. Conflicting Customer's Models

4.1 Relationships, relations, and objects

Relational data representations are general, but they demand from the user an understanding of possible relationships among the entities represented by the relations. An Entity-Relationship (E-R) model applied to such a database will describe all likely relationships. Most databases will be represented by a network more complex than a simple hierarchy. As stated above, we believe that a user working on a specific task will want an object-oriented representation of the data in the domains of current interest. There is an impedance mismatch of satisfying specific users' needs versus generality.

Again, appointing a committee to determine which object-configuration is right is only confusing. Many of the ongoing efforts by object-model enthusiasts to force data into a single, *correct* object structure are either naive or misguided [W:86]. The demands of differing constituencies generate a maintenance nightmare.

The approach we take here is to formalize the E-R model to match a relational infrastructure. We then can extend the relational algebra to operate on relationship representations as well as on relations that represent entities. In the structural model the relationships are represented by *connections* [WE:80]. The formalization of relationships leads to connections of five different types. Connections are characterized by semantics from which E-R relationship cardinalities can be derived, as indicated in Figure 12. The structural model was described in an early Entity-Relationship conference. Four of these connections are significant within and among object classes. Two support inheritance and hence simplify the customer's world.

Connections are concepts at the modeling level, and drive potentially complex computations, maintaining information system consistency. A significant early application of the structural model was to achieve provably correct integration of database semantic schemas [EW:79]. This work, however, was again limited to the databse design phase, as most early E-R research, and could not have the impact of a model which is interpretable and maintained. Its use in object management enhances its importance and provides a basis for ongoing validation during maintenance. Having a connection between object classes implies having linkages between their instances. Such linkages can be represented by the structure within an object, or by references among objects, as shown in Figure. 13.

Connection Type	Symbol	Direction
Ownership	$\longrightarrow\!\!\ast$	from single owner to multiple owned tuples
Part-of	$\longrightarrow\!\!\ast$	from single assembly to multiple owned parts
Reference	$\succ\!\!\longrightarrow$	from multiple primary to single foreign tuples
Subset	$\longrightarrow\!\!\ni$	from single general to single subset tuples
Identity	$\longrightarrow\!\!=$	from single source to single derived tuples

Figure 12. Basic connection semantics.

Relevant details of these semantics are given with the description of their use in object generation below. The Identity connection is not used in object generation, and our example also ignores the Part-of connection.

Objects represent views. In Figure 8 we saw that the medical record is used for financial and patient care data. In the view of the financial model the details of the service objects will be aggregated by facility to determine their utilization and revenues. For the patient care view the aggregation of the patient-objects is by diagnosis, so that projections can be made for the patient-base. At the decision making level, linkages will have to be made between services rendered in facilities and services appropriate for the patients' illnesses.

Servers today face the tension among relational and object databases. Should a server present objects or a relational presentation of data? If it presents objects suitable for one category of users, it will lose customers which have an alternate view. If the server presents a relational model to the customer, it has very general purpose capabilities, but the users have to build the object hierarchies themselves, perhaps with the guidance of an E-R diagram. Without tools to convert the knowledge inherent in such a diagram into code, a focus on relational databases is unfriendly.

4.2 View-objects generation

A concept of view-objects can resolve this issue, by transforming base data into objects according to user needs [W:86]. Mediation provides an architectural bases to position this transformation in larger systems. The PENGUIN project [BSKW:91] demonstrated how object-oriented data structures can be automatically generated. Now only the model has to be maintained. PENGUIN applies the semantic knowledge embodied in the *structural model*, to data stored in a relational database.

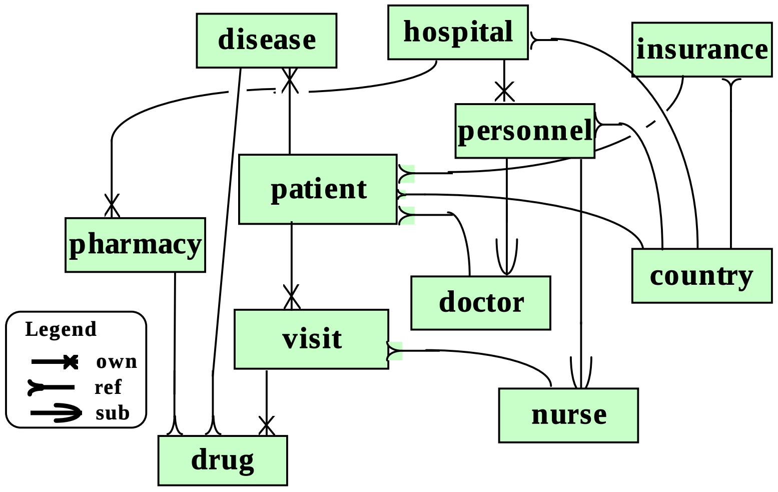

Figure 13. A structural model of a simple health-care setting

The structural semantics permit not only the creation of objects, but the retention and transformation of relationships between objects. Although general methods are not supported, methods for retrieval and storage of objects are created. More general methods applicable to the generated object have to be provided by the customers. However, it is actually rare for persistent objects to have internal methods for functions other than for fetch, store, and update of data. Since the fetch, store, and update semantics are inherent in the relational model, we can guarantee the correctness of their mappings to the objects, while avoiding the complexity needed to deal with arbitrary methods.

We now define the five connection types of the structural model and how the PENGUIN implementation interprets their semantics in order to generate objects [BW:90]:

1 An *ownership connection* describes the relationship between a superior class and its members. For instance, an object class definition describes a potentially large set of object instances. The classes are not necessarily abstract object-oriented programming concepts, but are often real, with actual modifiable data that can be inherited by their members. For example, under the description of a specific drug class are all the instances of that drug administered to the patients. There will be an instance record for data associated with the class as well, in which the name of the supplier and the price are recorded. PENGUIN proposes that owned elements be incorporated within an object.

Different applications may incorporate the same elements in different models. For the pharmacy the administered instances fall within the drug object, while for a patient record the same instances fall within the patient or a visit object. The conflicts that can arise due to simultaneous access have to be addressed at the execution level of the system, but can be formally recognized. Current object database systems, without this algebraic competence, can allow elements to appear in only one object configuration, limiting the practical size of objects.

2 Similar in structure are *part-of-hierarchies*. A physical object is often composed of parts that are of quite different types, although some attributes essential to the composition may be inherited. For example, a hospital room contains some beds, chairs, diagnostic equipment, and the like. But beds and chairs are specific instances of the class of furniture and inherit most of their properties from that class. The attribute that defines the composition, namely location, derives from the hospital room designation. These hierarchies can also be expressed by the ownership connection, and are treated identically in PENGUIN, since the structural model supports multiple ownerships and hence multiple inheritance. Further work to clarify the semantic distinctions could be beneficial if it could lead to generalizable conceptual differences. No part-of connections appear in Figure 13.

Conflicts between *class-of* and *part-of* object configurations are common. While PENGUIN cannot resolve the semantic differences automatically, it can handle objects created in either fashion and provide update protection if both configurations are in use.

3 A general type of connection is a *reference connection*; it is used to expand attributes by referencing foreign objects. For example, the location of a patient visit, perhaps the name of a clinic, references a building, which in turn is an object of interest to health care planners, and as such carries much detailed data. But the visit is not *owned* by the patient, not a part of it, and not a subset of the clinic. Reference connections are typically employed between independent objects. In Figure 13 several classes reference the country object class. PENGUIN proposes that references among heavy-weight objects remain external.

4 Divide-and-conquer is an essential approach in science, and in information systems as well. The *subset connection* defines such specialized groupings. For example, a specific type of antibiotic drug, as Gentamycin, inherits by default all properties of antibiotics in general. Structurally, we connect the more general class to the specific class by a *subset connection*. While subsets can easily be incorporated into objects, constraints due to differences in their referencing structure may make it unwise. For example, both 'patients' and 'nurses' are subsets of 'people', but their roles in a hospital are quite distinct, so that it would be unwise to create 'people' objects and encapsulate all the differences internally. However, many methods can be shared as `nationality` in Figure 13.

5 The identity connection defines relationships among replicated information, as are commonly found in distributed systems [WQ:87].

Three of the five connections types define hierarchical $(1 : n)$ relationships of differing semantics. Structures declared within objects in today's programming languages are restricted to hierarchies. More complex structures can be implemented by using programmed linkages within objects, but these will make algebraic manipulation difficult. PENGUIN creates hierarchies within objects, and uses external connections to link other objects needed in an application.

The semantics of all four connections lead to construction rules which are summarized in Figure 12. The connections are also associated with operational *insert and deletion* rules [W:83]. The objects constructed by PENGUIN have structures that satisfy those rules, so that correct operational behavior is encouraged. For example, elements owned by an object are deleted when the object is deleted, satisfying the semantics of ownership. Data elements will be retained, however, if they are also part of another structure. In today's relational databases consistency rules can be defined that inhibit errors, but the programmer still has to make the implied updates explicitly.

Note that the structural model has no connection with an $m : n$ cardinality. Such a relationship is described by two connections and one relation, since a relational implementation always requires a relation to define the subset of the $m \times n$ possible links. The composition extends to the semantics, giving $4^2 = 16$ different choices. Only a CODASYL-like database system is able to represent the links within its internal structure.

4.3 Relationship to the basic E-R model

Note that the semantics define relative cardinalities. Those cardinalities are presented in E-R Models. The additional semantics of the structural model convert the static E-R model to a dynamic structure. The symbols listed in Figure 12 create our conventions for representing E-R models with semantic constraints on database operations.

A dynamic capability is essential if we wish to achieve associative access, since the transformations required to achieve optimization must maintain the correct semantics. The PENGUIN system constructs objects as needed out of relational databases, given the structural model of connections. A PENGUIN query identifies the root of the object hierarchy, the *pivot*, and object templates are generated automatically.

The same relational algebra operations, used to optimize queries, or to convert relations to achieve database integration, or to define user-based views, are also applicable to connections. The algebra used for relations maps directly to the structural connections. Specifications for the join operation can be simplified, since only one connection needs to be named, rather than the two endpoints. FigureOBs 14 and 15 illustrate the extraction of an object model. The process starts by matching the root node of the object desired by the customer to the entities described by the structural model. This becomes the pivot, as applied to the database structure modeled in Figure 13. The algebraic transformations are applied to the model, so that the actual conversion is compiled, similar to the compilation of optimized database queries. Methods are created to automatically instantiate, select, and update object instances. We present the outline of the implemenation here because the work was published mainly in journals catering to the application areas; details are found in [B:90].

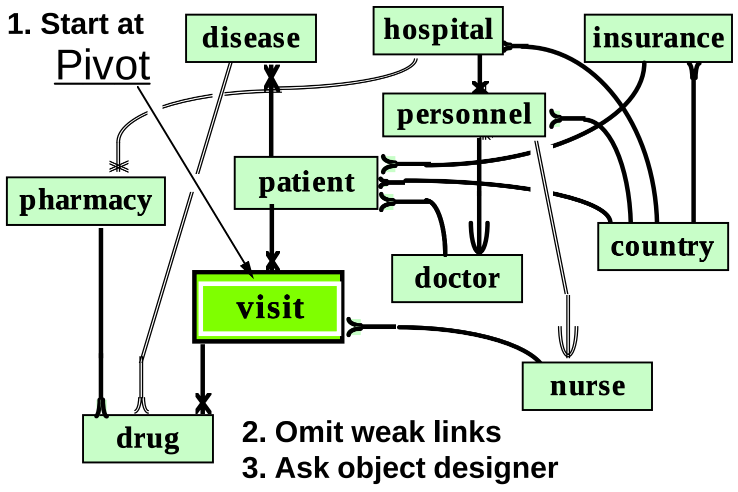

Figure 14. Applying the pivot and locating the object attributes

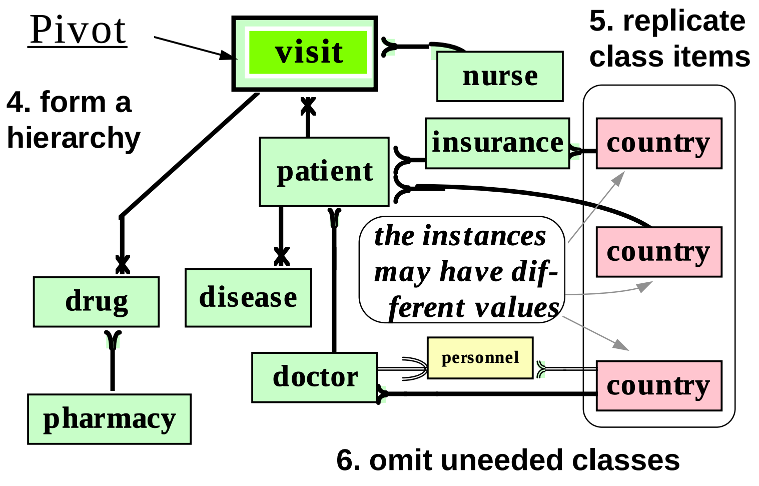

Figure 15. Restructuring and replicating shared leaves.

When connection types are mixed, the semantics become complex, but remain formally manageable [WE:80]. PENGUINincludes heuristics ti assess the strength of catenated paths from pivot to leaf nodes, and prunes the object tree accordingly. The mediator can now utilize the generated templates. When information about, say, a patient, is requested, all subsidiary data, say, the drug profile, identified in the template is included, as if a programmer had designed an appropriate object and written the code for it.

Distinct mediators can support distinct object configurations, as required by their domains. For example, the pharmacist can use a mediator for drugs and retrieve all patients receiving the drug as an object appropriate for the pharmacy, while the physician can obtain objects that use the patient as the pivot in another mediator. The conflict illustrated in Figure 11 has now been dealt with. Such flexibility rivals the capability of relational retrieval, but retains the constraints imposed by the semantic model and avoids possibly tricky programming. Having formal rules also permits checking of semantic consistency before any code generation occurs [KW:81].

The objects retrieved will obey the model, whereas a relational query can create nonsense; for example, one might join the daily dosage of a drug with the length of a patient's hospital stay. The lack of a documented connection (in a reasonable structural model) between dosage and length-of-stay makes it impossible to create such a template automatically. A determined programmer can, of course, retrieve drug and patient objects separately, and in the privacy of the mediator, compute anything that pleases the user.

4.4 Resolving the view-update Problem

Databases must also be updated. Here the templates must obey the constraints imposed by the structural model. No drug should be prescribed to a non-existing patient, and no drug class should be removed from the pharmacy that is still being given to an existing patient. In a relational database, the responsibility for maintaining correct program operation rests wholly on the programmer, augmented with some combination of foreign-key constraints, documentation, and common sense. Updates over objects have one serious complication: since the requests are stated at a high level of abstraction, their execution can be ambiguous.

For example, reassigning a patient to another physician could mean either (a) instruct the patient to visit another clinic, or (b) instruct the physician to take on the patient's current clinic. But as shown in [BSKW:91], the candidate ambiguities can be enumerated when the template is established. The mediator designer can choose which alternative makes sense. As these systems get more complex, the ambiguities will increase, but heuristics can rank and prune the number of alternatives to be presented to the manager. In the example above, choice (b), where the physician changes clinics, causes many more changes than the first alternative and is hence ranked much lower. By resolving ambiguities in the mediator, the template is created, the customer is relieved from having to deal with update ambiguities.

4.5 Object design and execution times

Automatic interpretation of the model to generate the templates for object instance generation and update the mediator greatly enhances maintenance. Since this work occurs at design time, the resulting templates can be compiled for fast execution. Rapid regeneration of mediator functions becomes feasible when the database models change.

4.6 Status

The PENGUIN technology has found its way into practice. For instance, it has been adopted by Persistence Software and is now part of the SUNSOFT Distributed Objects Everywhere (DOE) environment, operating with ORACLE, SYBASE, INGRES, and INFORMIX database systems [KJA:93]. It has also been used for some academic and commercial databases [RDCLPS:94].

The approach used also allows update of the underlying databases, typically disabled for relational views, as described in [BSKW:91]. The compilation takes place under management of an object administrator, typically the owner of the mediator in which the transformation code will reside. The customer is spared any need to resolve update problems.

5. Conclusion

We analyzed the factors leading to high software system maintenance cost: need for flexibility, unwarranted interaction among users from distinct domains, and the batching of system updates into periodic, traumatic events. None of these factors are effectively addressed by current exhortations in software engineering.

Implicitly we have clarified the difference between models and specifications. While specifications are developed initially, a program only ensues after a long sequence of transformational steps. No formal linkage from code to specification is provided within that process, although a current software engineering rule states that any section of code should have a documented audit trail to its origin. Comments giving such backward references to specifications are rarely found in programs, and their insertion delays the production of code.

The conflict between the generality of relational and E-R approaches versus object specificity can be resolved by transformations based on models. A model is intended to remain linked to the code objects, enable identification of the affected code, and aid in its regeneration. For example, PENGUIN keeps the models available to resolve hard issues, such as object-derived updates. To achieve such synergy requires a formal representation of the model and an algebraic capability over its components. Then the partitioning and layering into modules can be formally supported.

In general, mediation provides new level of scalability for large, distributed information systems. The economic benefits of system construction and maintenance under mediation are currently being analyzed for some of the early systems. It is unlikely that definite proofs will emerge until a few years have passed. Comparison will be difficult, because some of the early adopters of mediation technology are at sites where early analysis indicated that neither central solutions, designed according to accepted software engineering principles, nor federated client-server approaches were feasible. In smaller systems, the greater initial cost deters hard-pressed system builders to adopt mediation, since they are still encouraged to deliver a product, rather than a maintainable framework.

In time customers will understand the tradeoff, and may place the induced costs of system maintenance due to alternate architectures as a parameter into the contracts they issue for new systems and applications. Of course, familiarity with mediation concepts and tools is a prerequisite to their implementation. Application Program Interfaces (APIs) for KQML are now available for a variety of platforms and programming languages.

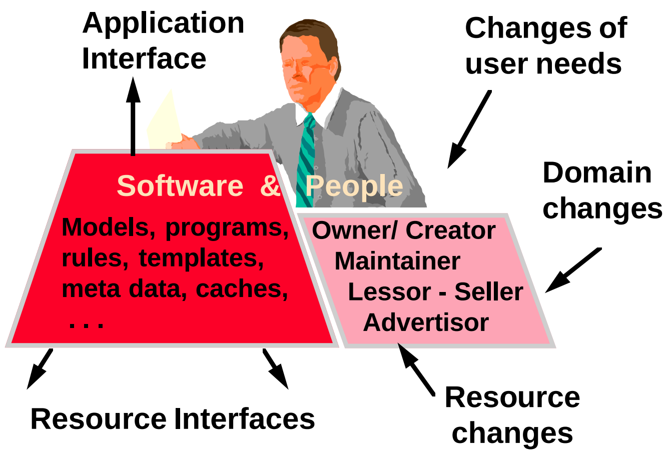

Figure 16. A Mediator Module and its Maintainer

Mediation does not eliminate maintenance, but it partitions the task it and allows its assignment to domain specialists. Every software designer agrees that modularization is essential for large scale software, but the rules have remained ad hoc [WKNSBBS:86]. The entity-relationship model provides a basis for such modeling, but until a better paradigm for software construction is adopted, the tools we have will be inserted in the manual approaches now in use, and the E-R model will be used as a specification rather than as a dynamic tool.

Within a module we benefit from the relational algebra. However, its operations assume the consistency which is only achievable in a coherent domain. The definitions that make objects coherent are also particular to a specific conceptual domain and its ontology. Scalability of software design and maintenance requires that we can build systems which encompass multiple disciplines and subdisciplines. The domain algebra sketched in this paper allows the defintion of articulation points among domains, so that we can maximize autonomy and minimize the cost of maintenance, while still providing interoperation.

Acknowledgement
This research direction would not have been feasible without the support of many students and colleagues. Most current work is performed in the context of the ARPA Intelligent Integration of Information (I3) program. Work on ontologies is supported through the Commercenet project at EIT, Menlo Park CA. The effects of the paradigm change on business models is being investigated within the contect of the Stanford Computer Industry Project. Work in progress on these topics can be located via the World-Wide Web nodes http://www-db.stanford/pub, progress with KQML is documented in http://www.cs.umbc.edu/kqml/mail/, and http://www-ksl.stanford.edu/knowledge-sharing. http://www-scip.stanford.edu/scip.

References

[ACHK:93] Y. Arens, C.Y. Chee, C.N. Hsu, and C.A. Knoblock: "Retrieving and Integrating Data from Multiple Information Sources"; *Int.Journal of Intelligent and Cooperative Information Systems*, Vol.2 no.2, 1993, pages 127-158.

[ASK:95] Ygal Arens, Michael Siegel, and Larry Kerschberg (eds.): I3 Architecture Reference; http://isse.gmu.edu/I3_Arch/index.html.

[B:90] Thierry Barsalou: *View Objects for Relational Databases*; Ph.D. dissertation, Stanford University, March 1990, Technical Report STAN-CS-90-1310.

[BSKW:91]T. Barsalou, N. Siambela, A. Keller, and G. Wiederhold: "Updating Relational Databases through Object-Based Views"; *ACM SIGMOD Conf. on the Management of Data 91*, Boulder CO, May 1991.

[BW:90] T. Barsalou and G. Wiederhold: "Complex Objects For Relational Databases"; *Computer Aided Design*, Vol.22 No.8, Buttersworth, Great Britain, October 1990.

[CALO:94] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman: "Using Metrics to Evaluate Software Systems Maintainability"; *IEEE Computer*, Vol.27 No.8, Aug.1994, pp.44-49.

[CEFGGMTW:93] M. Cutkosky, R.S. Engelmore, R. Fikes, M.R. Genesereth, T.R. Gruber, W.S. Mark, J.M. Tenenbaum, and J.C. Weber: "PACT: An Experiment in Integrating Concurrent Information Systems"; *IEEE Computer*, Vol.26 No.1, Jan.1993, pages 29-37.

[CHS:91] C. Collet, M. Huhns, and W-M. Shen: "Resource Integration Using a Large Knowledge Base in CARNOT"; *IEEE Computer*, Vol.24 No.12, Dec.1991.

[CW:91] Stefano Ceri and Jenifer Widom: "Deriving Production Rules for Incremental View Maintenance"; *17th Int.Conf.on Very Large Data Bases*, Barcelona, Spain, September 1991, pages 577-589.

[DeMichiel:89] Linda DeMichiel: "Performing Operations over Mismatched Domains"; *IEEE Data Engineering Conference* 5, Feb.1989; *IEEE Transactions on Knowledge and Data Engineering*, Vol.1 No.4, Dec. 1989.

[EIT:94] Jay Weber: KQML Transport Assumptions; http://www.eit.com/creations/research/shade/ kqml-spec/transport.html

[EW:79] Ramez El-Masri and Gio Wiederhold: "Data Model Integration Using the Structural Model"; Proceedings 1979 ACM SIGMOD Conference, pages 191-202.

[FCGB:91] R. Fikes, M. Cutkosky, T.R. Gruber, and J.V. Baalen: *Knowledge Sharing Technology Project Overview*; Knowledge Systems Laboratory, KSL-91-71, November 1991.

[FFMM:94] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire: "KQML as an Agent Communication Language"; to appear in *The Proceedings of the Third International Conference on Information and Knowledge Management* (CIKM'94), ACM Press, November 1994.

[GK:94] Michael Genesereth and Steven Ketchpel: "Software Agents"; *Comm. ACM*, Vol.37 No.7, July 1994, pp.48-53,147.

[Gruber:91] T.R. Gruber: "The Role of Common Ontology in Achieving Sharable, Reusable Knowledge Bases"; in Allen, Fikes, Sandewall (eds): *Principles of Knowledge Representation and Reasoning*: Morgan Kaufmann, 1991.

[Kahn:87] Robert E.Kahn: "Networks for Advanced Computing"; *Scientific American*, Vol 257 No.5; Oct.1987, pp.136-143.

[KB:95] A.M. Keller and Julie Basu: "A Predicate-based Caching Scheme for Client-Server Database Architectures"; *21st Int.Conf.on Very Large Data Bases*, Zurich, Switzerland, September 1995.

[KJA:93] Arthur M. Keller, Richard Jensen, and Shailesh Agarwal: "Persistence Software: Bridging Object-Oriented Programming and Relational Databases"; *Proceedings ACM SIGMOD*, 1993, pages 523-528.

[KW:81] Arthur M. Keller and Gio Wiederhold: "Validation of Updates Against the Structural Database Model"; *Symposium on Reliability in Distributed Software and Database Systems*, IEEE, July 1981, Pittsburgh PA, pages 195-199.

[Lehrer:94] Nancy Lehrer (ed.): Summary of I3 Projects; http://isx.com/pub/I3

[Liskov:75] Barabara H. Liskov and Steve N. Zilles: “Specification Techniques for Data Abstractions”; *IEEE Transactions on Software Engineering*, Vol.SE-1, 1975, pp.7–19.

[McShane:79] D.J. McShane, A. Harlow, R.G. Kraines, and J.F. Fries: “TOD: A Software System for the ARAMIS Data Bank”; IEEE Computer, Vol.12 No.11, Nov.1979,pages 34–40.

[NFFGPSS:93] R. Neches, R. Fikes, T. Finin, T.R. Gruber, R. Patil, T. Senator, and W.R. Swartout: “Enabling Technology for Knowledge Sharing”; *AI Magazine*, Vol.12 No.3, pp.37–56, 1993.

[RDCLPS:94] B. Reinwald, S. Dessloch, M. Carey, T. Lehman, H. Pirahesh and V. Srinivasan: “Making Real Data Persistent: Initial Experiences with SMRC”; *Proc. Int'l Workshop on Persistent Object Systems*, Tarascon, France, Sept.1994, pp.194–208.

[Tracz:95] Will Tracz: *Confessions of a Used Program Salesman*; Addison-Wesley, 1995.

[W:83] Gio Wiederhold: *Database Design*; McGraw-Hill; Second edition, 1983; Third Edition http://www-db.stanford.edu/pub/gio/dbd/intro.html.

[W:86] Gio Wiederhold: “Views, Objects, and Databases”; *IEEE Computer*; Vol.19 No.12, December 1986, pp.37-44.

[W:89] Gio Wiederhold: KQML: Objectives for a Knowledge Query and Manipulation Language; Stanford Internal report, Nov.1989.

[W:92C] Gio Wiederhold:“Mediators in the Architecture of Future Information Systems”; IEEE Computer, March 1992, pp.38-49.

[W:92I] Gio Wiederhold: “‘The Roles of Artficial Intelligence In Information Systems”; *Journal of Intelligent Information Systems*, Vol.1 No.1, 1992, pp.35–56.

[W:94F] Gio Wiederhold: “Interoperation, Mediation, and Ontologies”; Proceedings International Symposium on Fifth Generation Computer Systems (FGCS94), Workshop on Heterogeneous Cooperative Knowledge-Bases,Vol.W3, pages 33-48, ICOT, Tokyo, Japan, Dec.1994.

[W:94N] Gio Wiederhold: “An Ontology Algebra”; *Proceedings of the Monterey Workshop on Formal Methods*, Luqi (ed.), U.S. Naval Post Graduate School, Sept.1994.

[W:95A] Gio Wiederhold: Digital Libraries, and Productivity”; *Comm. of the ACM*, Vol.38 No.4, April 1995, pages 85-96.

[W:95D] Gio Wiederhold: “Value-added Mediation in Large-Scale Information Systems”; Proc.of the IFIP DS-6 Conference, Atlanta, May 1995; to appear in Meersman(ed): *Database Application Semantics*, Chapman and Hall, 1995.

[WCC:94] Gio Wiederhold, Stephen Cross, Charles Channell: *Information Integration*; IEEE Educational Videotape, 2 hours, October 1994, Robert Kahrman, sponsor, IEEE, Picataway NJ.

[WE:80] Gio Wiederhold and Ramez El-Masri:“The Structural Model for Database Design”; in Chen (ed.): Entity-Relationships Approach to Systems Analysis and Design, North-Holland, 1980, pages 237-257.

[Wegner:90] Peter Wegner: “Concepts and Paradigms of Object-Oriented Programming”; *OOPS Messenger*, August 1990.

[WG:95] Gio Wiederhold and Michael Genesereth: ”Basis for Mediation”; *Proc. COOPIS'95 Conference*, Vienna Austria, available from US West, Boulder CO, May 1995.

[WKNSBBS:86] G. Wiederhold, A.M. Keller, S. Navathe, D. Spooner, M. Berkowitz, B. Brykczynski, and J.Salasin: ”Modularizationof an ADA Database System”; *Proc. Sixth Advance Database Symposium*, Information Processing Society of Japan, Aug. 1986, pages 135-142.

[WQ:87] Gio Wiederhold and XiaoLei Qian: “Modeling Asynchrony in Distributed Databases”; *Proceedings Third IEEE Computer Society Data Engineering Conference*, IEEE-CS, Los Angeles, Feb. 1987.