Case Study of Object-Oriented Software Development

Dennis de Champeaux, Al Anderson, Dave Lerman
Marco Dalla Gasperina, Ed Feldhousen, Matt Glei
Floyd Fulton, Colleen Groh, David Houston
Charlie Monroe, Rommel Raj, Dave Shultheis*
Software and Systems Laboratory
HPL-91-170
October, 1991

object-oriented
analysis,
design and
implementation
methods

These are the highlights of a successfully completed application of object-oriented software development for a new product. The project was of medium size, the duration was less than 24 months (from the end of the requirements specification to product shipment), and the average team size was 8-10 software engineers. We discuss how the team dealt with major new aspects: a different paradigm, a different programming language, a different user interface environment, and a different development environment. In spite of all these novelties and in spite of the fact that the code size had been underestimated by about 75%, the project schedule slipped only by 20%. We touch upon all phases of the development life cycle: requirement capture, OO analysis, OO design, OO implementation and the verification phase. Some management perspectives are addressed as well.

Internal Accession Date Only
*All authors except de Champeaux are from the McMinnville Division
© Copyright Hewlett-Packard Company 1991

Contents

1 Introduction **1**

2 Summary Project Description **2**

3 The Requirements Capture Phase **2**

4 The Analysis Phase **3**

4.1 The Analysis Method 3

4.1.1 Static and inside object 4

4.1.2 Static and between objects 4

4.1.3 Dynamic and inside object 5

4.1.4 Dynamic and between objects 5

4.1.5 Other Models 6

4.2 Highlights of the Analysis 6

4.3 Analysis Example 8

5 The Design Phase **10**

5.1 The Design Method 10

5.2 Highlights of the Design 11

6 The Implementation Phase **12**

6.1 Rationale for the Implementation Language 12

6.2 Highlights of the Implementation 13

6.3 Implementation Example 15

7 The Verification Phase **17**

7.1 Defect Rate Data 18

8 The Overall Development Process **18**

9 Summary **18**

10 References **19**

1 Introduction

We will describe a software development project that used object-orientedness throughout all the phases. This project is considered a success story because it overcame many challenges. The development team switched from the structured way of development to the object paradigm. Multiple aspects of this transition will be discussed. This team dealt with other novelties as well: They were the first in their organization to base their product on a modern, object-based graphical user interface. They were the first to use C++. In spite of all these novelties and in spite of the fact that the code size had been underestimated by about 75% the project schedule slipped only by 20%. We will discuss why they obtained this favorable result.

The confluence of factors that led to the paradigm switch is unique. Four years ago John Young, the CEO of Hewlett-Packard, announced that software quality and productivity had to rise by a factor of ten in five years. Corporate Engineering, a central resource, responded by giving increased attention to metrics in order to measure the impact of steps to improve common practice. At yearly internal conferences best practice experiences were exchanged. Internal courses were developed in order to bring the most recent insights about software engineering to practitioners. One of those courses was on object-oriented (OO) development and was attended by the project leader. He was immediately convinced that this paradigm applied to this project, which was still in the investigation phase at that point.

Since there are no hard and fast rules for doing software engineering, divisional management had the wisdom not to impose a particular style on the project. Instead, they made clear that improvements were to be made but they left it up to the team how to obtain improvements. Positive results were, and are, to be adopted by new projects.

Given this benevolent context, the project leader – who came from the structured approach – convinced the management, and his team, that an OO approach should be taken. No contingency plan for backing out of the OO approach was in place when the project started.

During the proposal phase of the project, new methods and supporting toolsets were investigated. Several members of the team had prior, unsatisfying experiences with the structured approach to analysis/ design (SA/SD) and were interested in a new approach – OO analysis (OOA). Since none of the team members had experience with OO, the project leader began a search within HP for literature and expertise on OO methods, as well as for case studies of other projects' use of OO technology. No case studies from other projects were found. However, the project leader was able to establish a consulting relationships with an OOA expert at HP-Labs, and a software engineer from HP-SESD that was one of the authors of a prototype toolset that supported an approach to OOA. The project was signed on as an Alpha site for use of this prototype toolset.[1] Based on these two steps, the

[1] The prototype tool, developed by HP-SESD, supports an experimental OOA method. This method is under development

project leader was able to convince the project manager that by using OOA the project team would be able to accomplish the deliverables established for the analysis and design phases, and would produce better analysis and design results than would be possible by using SA/SD.

No expertise was available for bridging the analysis and the implementation phases with OO design. No decision had been made at the start of the project regarding the implementation language. As mentioned above, the team embarked into other uncharted territory: the user interface for the target system. OO development tools that would run on the development platform were limited to an alpha version of HP-SESD's OOA tool, which ran as single user, and an industry leading C++ compiler.

The balance of this paper explains how this team managed to obtain success in spite of these adversarial circumstances. We highlight aspects of the different development phases, how the team coped with problems and how they invented short-cut solutions.

2 Summary Project Description

We will confine our discussions to the system software development aspects.

The project was to create a clinical workstation for use in reviewing, editing, and printing Electrocardiographic (ECG) data collected from patients by an acquisition device. From the data gathered, the workstation produces a report detailing significant cardiac events, as well as providing access to the patient's ECG (electrical activity of the heart muscle) for review. The workstation is used by both technicians and physicians and the user interface is graphical based.

3 The Requirements Capture Phase

The requirements capture phase produced marketing data, project scheduling data, platform description, acquisition device/ data/ post-processor relationship definition, and a description of the functionality of the post-processor. This textual document was complemented by a prototype mockup of the user interface. An "internal design" document expanded the requirements. An early attempt was made in this document to describe the functionality in terms of objects. A diagram depicted data- and/or control-flow interconnections of these high-level objects. This diagram suggests that this document described a design commitment. On the other hand, we see the descriptions of the objects, the essential part of this document, as an elaboration of the requirements.

by faculty at Brigham Young University, Provo Utah [2].

4 The Analysis Phase

We describe here in summary the OOA method used and the highlights of its application.

4.1 The Analysis Method

The purpose of analysis is to capture in a semiformal way the functional requirements of a target system. Resource constraints, performance constraints, contextual constraints, etc. are beyond the (graphical) notations and are to be captured in a separate document. The analysis activity is described in [1] as:

- *Input:* A fuzzy, minimal, possibly inconsistent target specification, user policy, and project charter;
- *Output:* Understanding, a complete, consistent description of essential characteristics and behavior;
- *Techniques:* Study, brainstorming, interviewing, documenting;
- *Key notion for the descriptions:* Object.

It is the last item in this list that distinguishes OO analysis from structured analysis (and Jackson's method).

The output of the analysis serves two different purposes. First, it provides a check on the requirements. By using another semiformal notation, we obtain a deeper insight into the development task. A client, if different from the development team, will be able to agree (or not) that the analysis output captures the best insights about *what* the target system is supposed to be. Secondly, the output of the analysis is channeled into the design phase, which will give an abstract description of the *how*, the "blueprint" solution of the implementation task.

The analysis achieves its aim by producing multiple models that give different perspectives of the functionality of the system. Some of these models can be characterized along two different dimensions: first, whether a model describes a static aspect or a dynamic aspect of the system, and second, whether aspects regarding a single object are described or whether aspects regarding multiple objects are described. On the basis of these dimensions, we obtain the following matrix [1]:

	inside object	between objects
static	attribute constraint	relationship inheritance acquaintanceship
dynamic	state net and/or interface	interaction and/or causal connection

The information captured by these four components can be seen as a foundation. Other models can provide summary information, which does not require that they be made *after* the base models are constructed. In fact, summary models can drive the development of the base models. The process of constructing the analysis models is underdefined. As a default the team worked from left to right in the first row and similarly in the second row. However, it is noteworthy that the requirements document already contained an object interaction diagram, which contained summary information. Iterations of the modeling activity are possible and after a number of these, we see that "random" traversals among the different models are usually made.

In the remaining part of this section, we describe the main modeling notions that the team used.

4.1.1 Static and inside object

Although it is possible to describe unique objects in a target system (for instance the president of a company), usually we describe collections of objects – classes. A class is characterized through a prototypical element. The description of the "inside" of the prototypical element is done through attributes. For example, the class *Hand* has the attribute *finger*. An attribute is superficially just a binary relationship (for more on relationships see the next subsection). However, an attribute is special in that it *defines* the object, in contrast to a relationship which merely captures an incidental connection between objects. An attribute can be annotated with a multiplicity descriptor to express, for instance, that a hand has *five* fingers.

Sometimes, we know that certain combinations of attribute values in an object must not occur. We can express such knowledge with a constraint, an expression that refers to the relevant attribute names. For example, a square with the attribute *length_of_side* and *surface_area* should have the constraint that the latter is the square of the former.

4.1.2 Static and between objects

Entity-relationship modeling has been emulated by most OO analysis methods. While entities correspond with objects (and their grouping into classes), relationships can be used for capturing transient connections among the instances of the classes. For instance,

a $Husband - Wife$ relationship could be used in a domain relating objects in the classes Men and $Women$.

Inheritance is a "system" relationship between classes. Inheritance allows enhancement of the conciseness of the class descriptions by factoring out common features. When class B should have at least the attributes of class A, then we can make class B inherit these attributes from the class A. This causes B to inherit all the properties that we may know about A. If C and D have attributes in common, we may increase the conciseness of their descriptions by defining a class E with these common attributes, removing the common attributes from C and D, and making C and D subclasses of E. For example, common attributes of *Arms* and *Legs* can be made attributes of *Limbs*, and then both *Arms* and *Legs* can inherit these common attributes from *Limbs*.

Multiple inheritance can increase the conciseness even further. This occurs when a class inherits from multiple parents. We should be careful, however, not to create an ambiguity by inheriting an identically named, but different attribute from multiple parent classes (or from classes that they inherit attributes from).

Acquaintance relationships play a role in expressing object interaction connections, as explained below.

4.1.3 Dynamic and inside object

The dynamics of a prototypical class instance are captured by a state-transition network. The states and transitions should correspond with states and changes of the corresponding entity in the world that is being modeled; we do not represent the *design* of the object's intended behavior.

A precise description of a state can be done in terms of attribute value configurations. For example, when an *account* has the attribute *balance*, we can say that it is in the *overdrawn* state when its *balance* is negative.

A transition from a source state to a target state has a condition that needs to be satisfied in order for a transition event to occur. In addition to the satisfaction of this condition, some transitions need an external trigger (see below). An action is usually associated with a transition that is executed during the transition event before the target state is entered. As a side effect of entering a target state, a transition may generate triggers aimed at transitions in recipient objects.

4.1.4 Dynamic and between objects

As mentioned above a transition may generate triggers and may require a trigger in order to "fire". Triggering constitutes the behavioral "glue" that captures the causal connections between objects. Trigger semantics come in different flavors with respect to the question

of what to do when a recipient object is not in the proper state to honor an incoming trigger. The following cases are possible: the trigger is lost, the trigger is queued, or an exception is raised. Examples can be given that justify each approach. Thus we leave it to the analyst to indicate which version is employed.

Triggers can be strengthened by passing data along (which yields an asynchronous send). Waiting for acknowledgement of reception can be added, as well as time outs, etc.

In order to express trigger sender/recipient connections, we use acquaintance relationships. We can represent such a relationship by an attribute in the sending object.

4.1.5 Other Models

As mentioned earlier, summary models can facilitate in gaining a better overview of the intended system. For instance, a graph with classes as nodes in which attributes are omitted, but with inheritance connections and other relationships as arcs, gives us a broad view of the static structure. Similarly, a triggering diagram can show which (instances of) classes are causally connected, giving us a global view of the dynamic structure.

For more details on OOA, refer to [1], which has numerous references to the OOA literature.

4.2 Highlights of the Analysis

The requirements of the project (the input to the analysis phase) were well thought-out. Translating the requirements in the (semi-) formal notation of OOA helped the team to rethink the development task and produced a more detailed description. The commonly shared OOA notation allowed the team to match up the expectations and obligations of the separate components.

For every prototypical instance of a class, the team produced:

- a textual summary of the attributes;
- a graphical Object Relationship Model that contained attribute descriptions, local inheritance relationships, and local other relationships;
- a graphical State Model with summary descriptions of the conditions and actions on the transitions; and
- a graphical Dependency Model that provided client/server connections.

Relationships were problematic in the early phase of the analysis. When objects in two classes were related in the dynamic realm by a trigger connection or by a client/server

connection, it was thought originally that a relationship existed between the classes. No regular relationships remained after these connections were transformed into acquaintance relationships.

The team transgressed unknowingly into design. At a certain point the question arose whether displaying on the screen was to be done by a user-interface (UI) object digging the information out of the "bowels" of the system, or, alternatively, whether a deep object would take the initiative to push the information toward the UI object. To separate these two realms and to facilitate replacing the UI software or replacing a deep core object, an intermediate layer of presentation objects was added, which appears to be a design decision.

The analysis phase slipped by 15%. Unfamiliarity with this novel method is most likely the reason for the apparent slippage, and, since some design activity took place, it is possible that no slippage occurred at this point.

Some members of the team were committed to the analysis activity out of the expectation that by using a rigorous analysis method, they would produce a set of conceptual models of the problem space. These models, it was believed, would be used primarily as a stimulus for asking design questions and to ensure that all facets of the problem would be considered.

A belief held by other members of the team, as well as by some of the management, was that a rigorous analysis was not necessary. They felt that the requirements specification, combined with the knowledge obtained by prior efforts in the problem space, produced a sufficient enough understanding to begin design.

The results of the analysis phase only partially dispelled the belief that a rigorous analysis phase was unnecessary. For some parts of the problem being analyzed, like the database subsystem, there was a clear evolution to the understanding of the problem. Some of the entities inferred by an initial inspection of the requirements specification separated later into two or more entities, disappeared completely, or changed appearance significantly. If this refinement to the problem understanding had occurred later in the project, in design or implementation, it would have had a potentially disastrous effect to the efficacy of the system architecture produced, defect density, and/or project schedule.

The database designers also found that the OOA models raised a host of design questions; this not only allowed for a smooth transition to design, but implies a strong shaping of design efforts by the OOA process. In contrast to these clear analysis phase benefits, some members of the project team, particularly those working exclusively on UI, felt that the process of OOA helped them structure the problem they were solving, but they did not find the models produced to be a very useful foundation for design. This led to some questioning about the degree to which the models should be refined and rigorously pursued as a deliverable of the analysis phase.

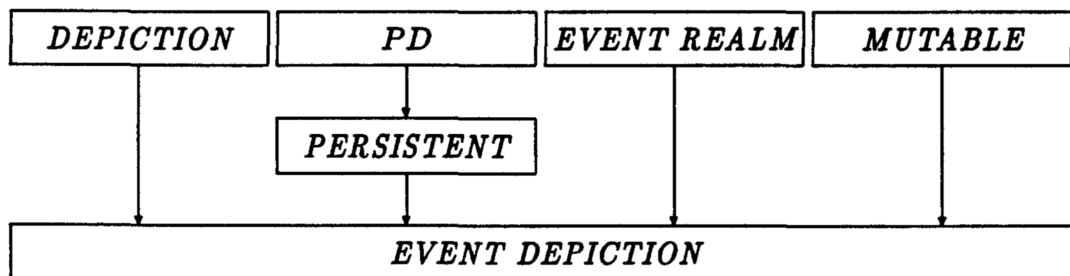
Even the team proponents of using a rigorous analysis method were unsure what additional benefits could be achieved by using the OO approach. In retrospect, it is now clear that a primary benefit of using the OO approach was the way that some of the analysis models became a powerful framework upon which the design was built.

The implementation subsequently exploited this framework again and thus all three steps used the same underlying conceptual semantics. This is in contrast to the structured approach, where there is a discontinuity between analysis models and design specifications, partly due to the difference in semantics (data flow and process models becoming control flow and interface descriptions).

The Object Relationship Model contributed by inferring a class hierarchy; objects and their derivations followed and became C++ classes and subclasses. The State Model contributed by establishing objectives for the design of internal behavior of objects. Object states were reflected by the instance data of C++ objects. Triggers that were catalysts to state changes became messages sent and received by these objects. The Dependency Model provided objectives for the design of dynamic object interaction; an application organization supported synchronous and asynchronous communication that accomplished part of the dynamic interaction implied by this model. A finer-grained realization of object interaction described by the Dependency Model took place within these applications as well.

As the analysis work progressed, the number of distinct base classes of objects began to diminish. The system eventually was represented by relatively few (less than 25) entities, from which almost all others (a total of about 60 significant classes) were derived. The relationship between the base entities and their derivations (represented by the "is a" construct in the relationship model) provided the seeds for inheritance opportunities exploited later during implementation. These relationships emerged directly from examination of attributes of entities – an understanding accomplished by following the disciplined inquiry of the problem dictated by the OOA method.

4.3 Analysis Example



[*PD* in this diagram stands for *PATIENT_DATA*.]

The figure above shows a portion of the highest levels of the patient data hierarchy. The classes PATIENT_DATA, DEPICTION, PERSISTENT, EVENT REALM, and MUTABLE are base classes from which large numbers of objects are derived. The EVENT DEPICTION object is a good example of such a derived object.

The roles of these objects are described below:

- PATIENT_DATA: Most major objects in the system are derived from this class. It embodies important status information such as existence/ activated and error states.
- PERSISTENT: The persistent class consists of all patient data objects that are stored on disk and persist from editing session to editing session. It is a powerful class because it handles the majority of disk transfer operations for patient data.
- MUTABLE: The mutable class consists of objects in the system that are altered by user edits. It provides mechanisms for receiving and propagating user edits throughout the database.
- DEPICTION: The depiction class is the base class for all database objects that are externally visible. PRESENTER objects in the User Interface view database objects through objects derived from the depiction class.
- EVENT REALM: This class (and all other REALM classes) plays a role of grouping objects that are closely related though structurally unique. In addition to the event depiction class, objects derived from the event realm class are: event depiction stream, event, event stream, and diary. The realm groupings form the basis of the file organizations developed during the design phase (i.e., all objects in a realm are stored in the same file).
- EVENT DEPICTION: The event depiction class is a derived class that provides the external view of data that describes a significant cardiac event. This external view, or depiction, is a printed and viewed entity which shows ECG waveform together with ECG measurements and annotation which describes the cardiac event. The event depiction class is derived from the mutable class because it is alterable by the effects of user edits. It is derived from the depiction class because it provides data to clients that allows for visual representation of the event depiction. It is derived from the persistent class because its state changes (user edits) persist from editing session to editing session. And it is a member of the event realm class with all other event-related objects.

5 The Design Phase

5.1 The Design Method

The input of the design method consists of:

- the elaborated functionality as formulated by the analysis phase;
- resource and performance constraints as formulated in the requirements document; and
- other constraints as formulated in the requirements document.

The objects in the analysis phase have a great amount of autonomy. This gives the functional description the flavor of a system with an unbounded amount of parallelism. Consequently, serialization is a major obligation of the design phase to satisfy the resource constraint of having available only a single CPU platform (and an operating system that does not support concurrency). In addition, the design phase is responsible for creating the proper algorithms and data structures inside the objects to satisfy performance constraints.

It turned out that the OO Analysis notation was useful during high-level design. The analysis models provided a semantical framework that could easily be designed against (hierarchy and specialization leading to designing class trees with inheritance and overloading, as an example). The models were also useful because the objects, with associated attributes and behavior, were readily recognizable as refinements of the requirements specification, thus providing an invaluable anchoring of system design to the problem being solved. This second point is regarded by the team as a true breakthrough by OOA, and it offers a clear view of what the structured approach to analysis fails to achieve.

Lower-level design demanded a notation for precise interface characterization of (sequentialized) objects. The team decided to model the required notation after the Unix "man-page" approach to documenting services and tools. Given the lack of a well-defined OO design method, the team members created their own process by first concentrating on what they decided a design process should deliver.

Interface protocols, data structures, required system services, memory requirements of objects, and application organization to groups of objects are examples of deliverables identified. The team used man-pages to document the results of design, where a man-page could specify interface protocols to system services, or describe formats of data or configuration files, or, in the case of system objects, define ontology, instance data, internal states, and the triggers to state changes.

The approach taken by the team to get from the analysis models to design deliverables was ad hoc. Once a real-world context for the problem solution was applied to the analysis

models, a host of design questions arose. The team followed an iterative process to answer these high-level questions and their ensuing lower-level counterparts until enough of the design had been specified that only minor details seemed to remain. These final design details, it was decided, would require some venturing into implementation to obtain a full answer.

5.2 Highlights of the Design

Sequentialization was mainly achieved by investigating dependencies among interacting objects. One of the difficulties encountered during the design of the database system was how to accomplish updating of all objects constituting the database subsequent to a user edit. Initial design efforts showed that when triggers occurred during the processing of a user edit, there were an assortment of objects that needed to be notified of the trigger, seemingly in parallel and independent of one another. Each of the objects notified in turn underwent state changes and produced triggers of its own, some of which would have an effect on objects that also received the original, starting trigger. Synchronization of processing these triggers became a major design topic.

One solution to synchronization was to have the object that originated the trigger notify the recipient objects in a proper sequence. This approach was quickly abandoned, partly on the principal that the object that is the origin of the trigger would require far too intimate a knowledge of the target objects to decide the proper ordering of notification. The second reason this was abandoned was its obviously fragile nature with regard to changes in the system. If the behavior of a target object changed or if new objects were introduced to the system, the trigger originator might need to change. A major concern was deciding where in the system design this knowledge would reside, and how difficult it would be to maintain and verify.

An alternative approach was pursued for notification of trigger events that was based on a target dependent object informing (registering with) the trigger source of its need for notification. This was a more robust method from the perspective of change to the system. As an example, if an object determined that it would no longer need a notification, it would simply unregister itself from the trigger source. Adding new dependent objects to the system would not require changes to the trigger source; this source's list would simply grow as new objects registered their need for that particular form of notification.

The synchronization question was explored by refining the notion of dependency between objects. It was found that the more rigorous and precise the description of a relationship between objects became, in almost all cases a clear dependency relationship arose between them. This implied a natural ordering to the handling of triggers and resolved nearly all circular notification paths and renotification of events. The notification paths began to take on a tree-like organization, with no significance to which branch was chosen to follow first, and with no attachments between branches. This approach solved all but one

significant synchronization problem. This final synchronization problem was handled by using the first, less robust approach.

By focusing on dependency relationships of objects, it became apparent that when a clear dependency relationship could not be established, either attributes of the objects involved were not clearly defined (possibly two objects posing as one), or an object (an important role player) was missing from both the analysis and the design. Dealing with sequentialization thus provided a thorough check on analysis output and on other aspects of the design.

The design phase was delayed by approximately 30%.

6 The Implementation Phase

6.1 Rationale for the Implementation Language

The decision to use the OO paradigm was made early in this project. However, the decision to use an OO programming language, or not, came late. It was felt that the OO approach would be beneficial either way. In fact, the project used C++ for some part of the system and C for the balance.

The reasons for this hybrid approach were the following: An early design commitment was to use the programming environment provided by the graphical based operating system to support all subsystems comprising the architecture: user interface, datacom, archive, printing, and database. The database handles management of persistent (file-based) data, as well as providing a set of services to the user interface for gaining access to patient data for the purposes of displaying or printing, and for submitting user edits. When user edits change the basis of the cardiac events reported, the database also performs a redetection of these events subsequent to the edit.

The results of the analysis/design phases showed that the highest leverage and lowest risk in using C++ would be in the database subsystem. Much of the rest of the system, particularly those parts composed largely of user interface elements, are tied intimately to a style of program organization defined by conventions of the UI programming environment. This prototypical organization is based on C and would gain little benefit from C++ programming in its current form. Some OOP tool providers are performing ground breaking work in defining new program organizations for the target environment using C++. We decided however that the risk of using this preliminary and unproven approach, combined with the risk of requiring 2/3 of the team members to learn C++, outweighed the uncertain benefits.

The functionality represented by the database subsystem, by contrast, was constrained little by pre-existing code organizations. This, coupled with obvious, powerful mappings

of database objects identified and characterized during analysis and design to C++ class definitions, led us to the choice of using C++ for the database, which represented about 30% of the total coding effort of the project. This decision required two team members to master the use of C++.

6.2 Highlights of the Implementation

Although almost all of the system was analyzed and designed following the OO method to varying degrees, the transition to implementation for the portions slated for C was different from those to be in C++.

For the C-based implementation, the transition began with an application framework inferred by the object relationship and object dependency models of analysis. Modules were created (not true objects) to constitute this framework; and then interface protocols defined by the man-page design approach produced the surface area of modules. The internal behavior of modules was defined by a structure that was dictated largely by conventions established for programming in C for the target UI environment.

For the C++ portion of the system, the starting point was similar, in that a framework was constructed from work done largely in the analysis phase. The point of departure began with how this framework was fabricated. The structure of the application framework was implemented primarily through two hierarchies of C++ classes. Objects described by analysis and whose behavior was given real-world context by design, were realized in code as instances of C++ classes. As in the case of the C implementation, interface protocols defined by design produced the surface area of functionality, but in this case for actual objects (instance data + operations on the data) and with polymorphism apparent.

Internal behavior of objects generally did not follow UI environment programming conventions, but rather had an organization of generalized code in parent classes (status, file I/O, standard services for clients, undo management, etc.), which was then inherited or overloaded by more specialized behavior and code in subclasses (unique memory needs due to size, special client services, unique dependencies to other objects, etc.).

As was mentioned above, about 30% of the code was written in C++. Another 2-3% was written in assembler, with the remainder written in C. The interface between C++ and the other programming languages was managed in several ways. First, much of the functionality of the system was naturally divided into applications: a data acquisition application, a printing application, a review and edit application, etc. The database is an application itself. It is made up solely of C++ and assembler code; the protocol defined between these two languages was unidirectional, with assembler routines appearing as static or global C functions callable by C++ member functions. No interface was developed for calling either static or instantiated class functions by assembler routines.

Systemwide utility services required by the database application were accessible through both statically and dynamically linked libraries of C-functions. Application-level communication between the database application and client applications was accomplished through the target environment's messaging system, which provides packet-based synchronous and asynchronous communication.

The database designers benefited from using C++ in both predicted and unpredicted ways. They understood at the onset and then were shown by their implementation efforts that encapsulation of objects helped prevent system complexity from becoming unmanageable. Encapsulation creates entities whose public behavior is simple and constrained, and whose internal behavior is insulated from the processing in the rest of the system. The object organization of functionality attached to instance data also insulated the programmers from the management of manipulating operations on dozens or even hundreds of unique data sets through arrays or other structures that would otherwise have been required in C.

The users of C++ also were shown and benefitted from polymorphism. This concept helps limit system complexity by preventing rampant production of protocol semantics that would have to be defined, implemented, and then remembered in a typical C-based system. They also benefitted in predicted ways from the inheritance possible with the C++ class organization, from the flexibility of statically and dynamically binding overloaded member functions, and from more minor C++ extensions to C, such as in-line functions, default function parameters, and the rigor of the strict typing of C++. All these benefits of C++ helped the programmers to conceptualize more in terms of problem models and designs and less in terms of program organization and programming language constructs.

Some of the unpredicted benefits of using C++ relate to design changes made during the implementation phase. The team found that the promised extensibility of OO systems and the limits of "damage" to the system from change were benefits that were reaped during coding. In particular, when, due to incomplete design, new specializations of objects were discovered during coding. The team was able to add these objects quickly to the system, largely due to the clear definition and inherited functionality of the parent classes. This was a welcome, if unexpected, boon to the programmers. Another benefit of polymorphism that was not predicted was that by having a smaller set of protocol semantics between objects and the services they provide, the programmers could more easily identify reuse possibilities.

The database designers were concerned during the design phase about the performance that could be achieved by using C++. An example of performance requirements is that the use model of the clients accessing database services predicate an instantaneous update of data elements following each user edit. For some user edits, as many as 100k data elements may have to be scanned. These scans entail the application of pattern-matching algorithms , followed by an ensuing update to potentially hundreds of other dependent

objects. These edits must be completed very quickly, with most of them having a delay undetectable by the user. The team found C++ performance excellent for all but a few operations that were subsequently implemented with low-level primitive facilities written in assembler and optimized for the capabilities of the delivery machine's processor. Even the additional overhead of dynamically binding member functions was found to be an insignificant penalty to performance.

The areas where we had difficulty with C++ were few, but significant, and should be mentioned. The team found that they had to perform more memory management duties than they had had hoped as well as more low-level management of the locations (i.e., pointers) and hence avenues to access of instances of objects. As a result of the latter limitation, static class functions may have been overused as a more direct route to class services.

The implementation phase slipped by about 15%. The slippage encountered was due to underestimating the time needed to become proficient users of C++ facilities, to problems with a buggy C++ compiler, to primitive and limited debugging tools, and to the need to undertake more memory management duties than had been predicted.

6.3 Implementation Example

Presented here are two examples of member functions of objects shown in the analysis example presented earlier. First is an example of inherited behavior, and second is an example of specialized behavior of base class function in a derived class.

The following is a PERSISTENT class function that retrieves data for a persistent object from disk. This is a good example of inherited functionality because this is the single piece of code that retrieves data for all persistent objects in the system:

```
short persistent::RetrievePersistBlock(void)
{
  lpvoid lpTmp;

   if (!MyDir())
      return NULL;

   if (AllocPersistBlock(MyDir()->GetObjectSize(PersistId))
            != OK ||
       MyDir()->GetObject(PersistId,lpBlob) != OK)
       return BRACK;
   else
       return OK;
}
```

The following is an EVENT DEPICTION class function that handles external client requests to the event depiction. This is a good example of specialization of a base function in a derived class. The function depiction::ClientMsg() is virtual. Every instance of a depiction object services clients through this function. ClientMsg() is also defined in each class derived from the depiction class for the specialized response of that class to client requests. Note that the actual code is only given for the TOGGLEINC message type. For the other message types, the code has been removed for brevity.

```
long evntdepictn::ClientMsg(
  WORD wMsg,
  WORD wParam,
  LONG lParam)
{
  intrnchgdesc ChgSpec;
  lphltrusreditstruct lpEdits;

  ChgSpec.ChgEntry.ChgOrigin = this;
    switch (wMsg) {
      case NEWLABEL:
           ....
           break;

      case NEWEVENT:
           ....
           break;

      case NEWHEARTRATE:
           ....
           break;

      case TOGGLEINC:
           // Setup for possible edit undo
           if (!OpenUndo(EVNTDINC))
              break;
           lpEvntDepnStruct->bSelectiveInc = TRUE;

           // Setup struct to notify other objects of change
           ChgSpec.ChgEntry.EditType = ToggleIncAttrib();
           ChgSpec.ChgEntry.EditValType = SHORTVAL;
           ChgSpec.ChgEntry.nValue=lpEvntDepnStruct->
                                       Labels.Labels[0].nToken;
           ChgSpec.ChgTag = NULL;
```

```
            // Notify other objects of change
            IntrnChgNotify(ChgSpec);
            SetMutableChg();
            SetDepictnChg(EVNTDINC);
            break;


        default:
            DEBUG(EVNTDEPN_ERR2)
            wsprintf(sErrString,"Evntdepictn: %d,
                    bogus client msg: %x",
                    MyNum(),wMsg);
            LOGERROR(sBaseName, MAJORERRNUM,
                                BRACK, sErrString, UNKNOWN);
            break;
            }
    return 0L;
}
```

7 The Verification Phase

The project used several different approaches to verify the integrity of the code developed. Design reviews were performed for most subsystems to pinpoint problem areas due to incomplete design or incomplete interface protocols, actual design flaws, and designs with undue complexity and hence potential trouble spots for code verification. During implementation, a programming convention checking tool was run on the C code. The convention checker helped identify casting problems, uninitialized variables, and coding practices that were unconventional and possibly difficult to maintain or easy to break. No such tool was available to run against the C++ code.

Code reviews were also performed on critical sections of the code, again to identify non-standard or questionable coding practices, as well as to point out potential logic flaws for boundary conditions. C++ code presented new challenges in this arena, and consequently little of the C++ code was reviewed in a rigorous fashion. For C++, we found that the code of class member functions was often simple, but in order to verify its correctness in a review setting, the entire class tree, as well as the inherited functionality, had to be understood. Given this additional overhead to the review process, as well as the fact that only two to three members of the team understood C++ nomenclature and organization, we decided to forgo review of the C++ code. The code also underwent unit testing to

varying degrees. For the C code, a branch-flow analysis tool was used to track coverage accomplished by test cases. This tool also was not available for the C++ code.

Although some of the standard tools and metrics described above were not available for verification of C++ code, C++ did provide some advantages in the verification process. One excellent example is our LOGFILE class, a class that was introduced to provide diagnostic information easily. The LOGFILE class has a simple interface that can be instantiated by any object where diagnostics are required. The parent object simply creates the LOGFILE object, providing only a file name, and calls the LogString() function to output messages to that file. All file I/O is handled in the LOGFILE member functions, including some important buffering capabilities. The LOGFILE class was invaluable in the verification phase, in terms of both the ease of adding diagnostics and the prevention of a significant amount of replicated code.

7.1 Defect Rate Data

The C++ code made up about 30% of the overall code written. Although the overall defect rate of the code was equal to industry standards, the defect rate of the C++ part of the code was 50% less than the C code.

8 The Overall Development Process

Most people these days suggest that the overall development process is described better with a spiral or fountain metaphor than with the waterfall model. They even may *prescribe* these alternatives to the waterfall model. The development process of the project, however, followed *grosso modo* the waterfall model. How did that happen?

We conjecture that the well-thought-out requirements document was the key element. Our familiarity with the product domain must also have contributed to the clarity of the requirements. This solid foundation allowed us to come up with a carefully thought-out analysis, which in its turn provided the support for a complete design. Marketing contributed by supplying only a minimal number of functionality changes during the development process. The well-thought-out analysis and design accommodated these modifications readily.

Estimates for the reuse fraction of the output of the analysis, design, and implementation phases range between 80% and 100%.

Whether the maintenance cost for this system will be reduced is as yet unknown.

9 Summary

We have reported a success story of OO software development in a division of HP.

A team of 8 - 10 software engineers developed a product in less than 24 months after the requirements analysis was completed. Code size had been under-estimated by 75%, while the project schedule slipped by only 20%.

This team had no experience with the OO paradigm and no experience with C++. Neither had they experience with the new graphical user interface or its encumbent programming environment.

The team was able to do the full OO life cycle with minimal tool support. Backtracking between the development phases was minimal.

The team expects concept, design, and code reuse to range between 80% and 100%.

Acknowledgements

We thank Derek Coleman and his team, Penelope Faure, Mohamed Fayad, Donna Ho and Susan McBane for feedback on earlier versions of this document.

10 References

[1] de Champeaux, D. & P. Faure, A Comparative Study of Object-Oriented Analysis Methods, Report HPL-91-41, 1991 April.

[2] Kurtz, B., S.N. Woodfield & D.W. Embley, Object-Oriented Systems Analysis and Specification: A Model-Driven Approach, Hewlett-Packard & CS Dept. Brigham Young University, 1989 August.