An Object Oriented Application in EsiObjects

by Steven Popkes

By the spring of 1998, ESI Technology had completed a non-trivial project that put the object paradigm to the test in the M community. This paper reviews our methods and results.

The Goal

The Department of Defense wished to modernize their hospital information system (CHCS). Part of the project involved representing the information stored in a CHCS File Manager database in an object oriented way. The File Manager and CHCS code were not to be used. The database wrapper had to be complete in and of itself. EsiObjects™ was chosen as the appropriate technology. The CHCS database contains thousands of patients in the system. Each patient has several hundred fields of varying types including data fields, enumerated fields and pointer fields. A subset of the patient record, called the"MiniReg" fields, were chosen for the proof of concept. These fields included such data values as name, date of birth, branch of service, etc., and were intended to act as the minimum registration of a patient until the full registration could be done.

The resulting collection of objects would have to create new patients, update old patients and be able to operate in parallel with the existing CHCS system. In addition, no part of the original CHCS system could be used in the project. We were instructed to consider FileMan, CHCS and other existing software as disposable and not to rely on them.

Clearly, with the large number of fields and files that were to be accommodated, some automated means would have to be used to generate the code. The process flow is shown in Figure 1. The construction of the compiler and its support classes is not the focus of this article. Instead, we will discuss the approach we made in designing the classes that were to be generated by the compiler.

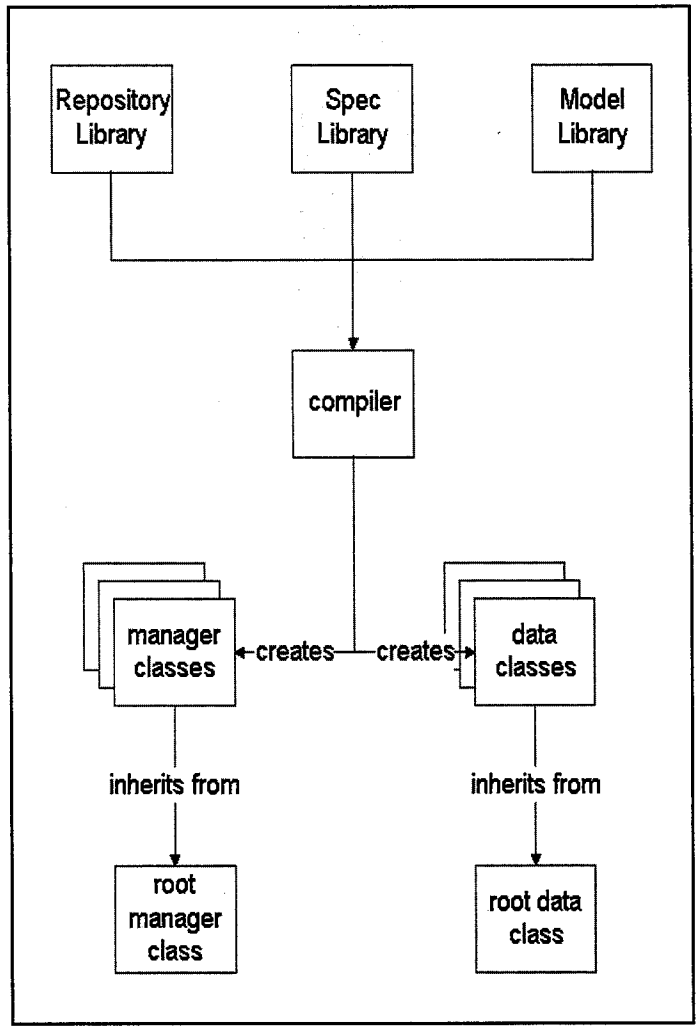

Figure 1. Process flow

Objects

We will not go into the fundamentals of objects in general within this article. Some familiarity with the object paradigm is assumed. Further, information is available in *Object Oriented Technology: A Manager's Guide*, or *Business Engineering with Object Technology*, both by David A. Taylor. There are also numerous other such references.

EsiObjects is a formally defined representation of the object paradigm. It presumes that the only means by

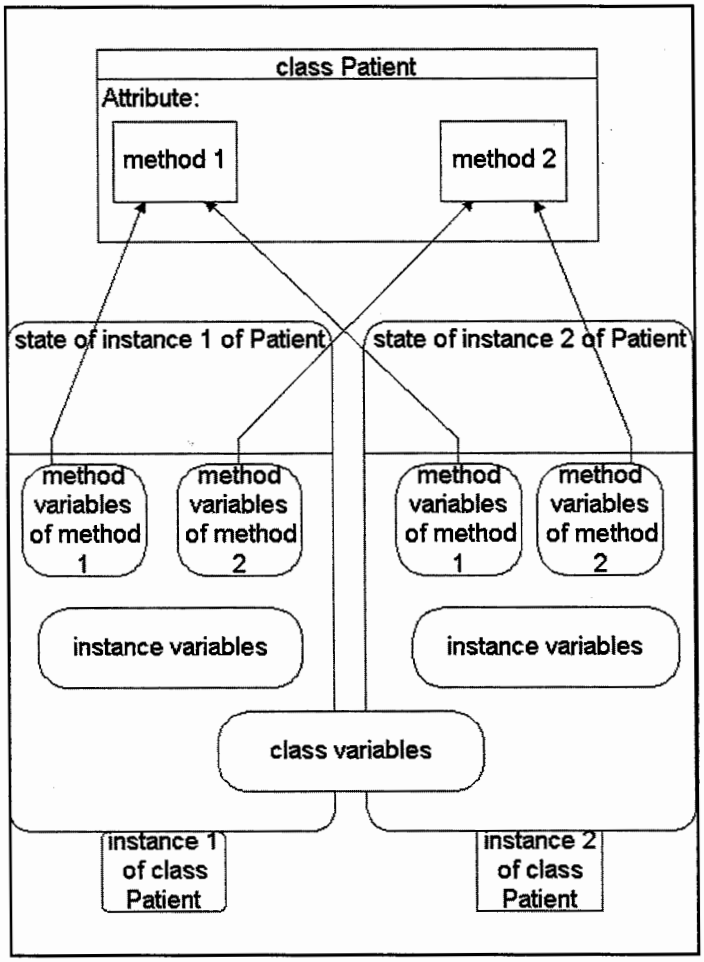

Figure 2. Variable types

which information can be propagated from a method directly to the caller is by returning a value.

As in other object languages, EsiObjects separates the code and state of the object. The executable code resides in classes and is available to all objects of the same class. Code is executed in segments known as methods or properties. Methods or properties can be grouped together in interfaces. An example of the syntax of a method call would be:

```
S A%X=A%Oid.Internal::Filter
```

In the above, A%X and A%Oid are variables. The period (".") separating A%Oid and Internal::Filter indicates that A%Oid contains an object identifier. Internal::Filter identifies a method (Filter) in an interface (Internal) in that object. EsiObjects provides the first interface for a class, the Primary interface.

The state of the object is represented by the variables contained within the instantiation of that object. This is a

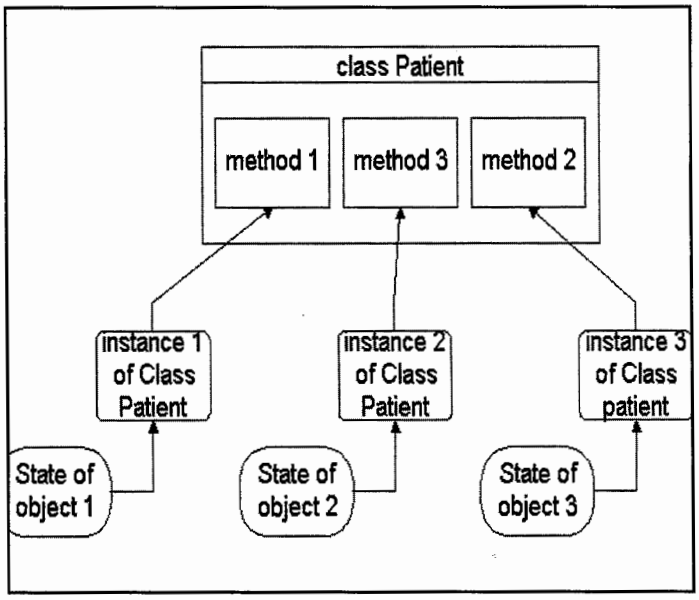

Figure 3. Object state and method relationships

common dichotomy: code is the province of the classes and state is the province of the instantiation of those classes. Often the state of the object is unique to the object. This is an important difference between OO code and non-OO code. For example, a given function in M might have state that is local to the function for the duration of the call. The scope of EsiObjects variables can be limited to the duration of a function call, the lifespan of the object instantiation, or be shared between all instantiations of the same class.

The lifespan of method variables is limited to the call to a given operation. This is similar to the way that the NEW command works, though the mechanism is implemented differently. In addition, variables have a lifespan that is associated with the lifespan of the object. These instance variables are available to all operations of the object. In EsiObjects, class variables are available to instances of a given class. The different variable types are illustrated in Figure 2. Method variables are local to the state of a particular object and associated with the operation of a method in the class. Instance variables are available to all methods of the particular object. Class variables are available to all members of the class.

Syntactically, variables are scoped by a prefix. A%* variables are method variables, I%* variables are instance variables and C%* variables are class variables. This avoids the necessity of some sort of declaration command within the EsiObjects code. There are also means by which variables can be created automatically when the object is instantiated.

Variables are protected within their scope. Method variables are not exposed beyond the confines of the method. Instance variables are not exposed beyond the confines of the object and class variables are not exposed beyond the confines of the class. This means that a given method operating on an instance variable, for example, can be guaranteed that the contents of the variable are reserved for the local instance and no other.

The relationship between object state and object code is shown in Figure 3. Notice that though the state is unique to the instance of Class Patient, the code is unique only to the class. It is worth repeating that state refers to the data associated with the instantiation of the class and is separated from the code, which is associated with the class. This allows the code to be shared between objects while preserving the integrity of the variables. The protection of variables within their scope is called encapsulation. This project would have been much harder to complete if EsiObjects had not supported encapsulation.

Another quality of objects derived from classes is inheritance. Inheritance means that the structure of an object as defined in the class of that object can be inherited from other classes. For example, patients share many similar qualities: date of birth, sex, name, etc. However, a patient that is also enrolled in military service will have qualities unique to the fact he is in the Army or Navy. He will have rank, a military service number, perhaps a duty station etc.

If we build a description of patient class inheritance (as shown in Figure 4) we would expect the common components of patient to be part of the patient class. The Army patient class would have components peculiar to the Army, but also inherit the more general properties of the patient class.

Methods, properties and instance variables of a given class can be inherited from the class's parent. However, methods and properties can be overridden. In Figure 4, the method AddressValidation is common between both the general patient class and the Army patient class. Note, however, the method is located in both places. In this case, the AddressValidation method of Army overrides the AddressValidation method of the general patient class.

This behavior was important to our design because it allowed us to write general methods that could then operate against specific local methods and properties.

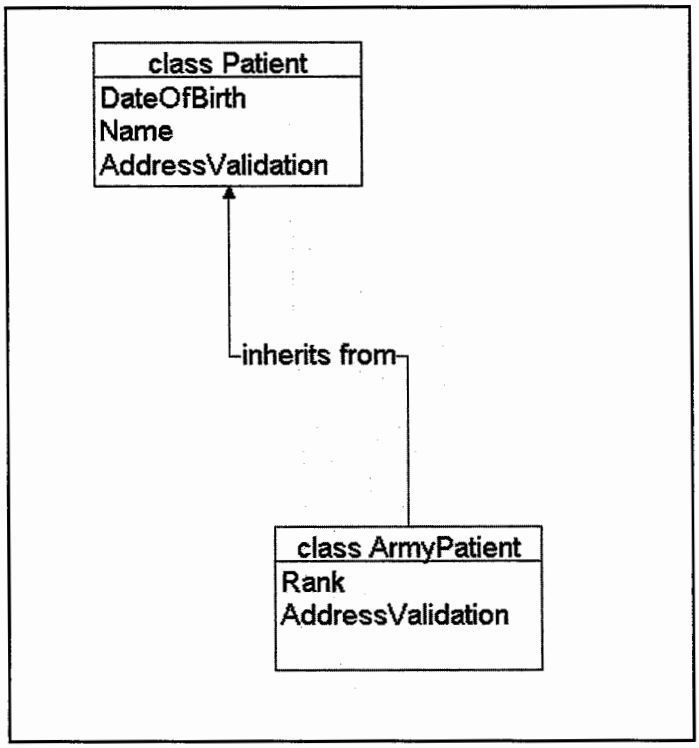

Figure 4. Patient Class inheritance

This description is somewhat abstract in order to place EsiObjects in the same venue as other object oriented languages. EsiObjects derives from M but implements all of the fundamental aspects of the object paradigm: inheritance, encapsulation, message, etc.

The Design

The intention of the class design was not to solely represent the patient database. Instead, we planned to be able to represent all of the relevant data associated with any of the File Manager files used in CHCS. The design, therefore, had to be specific enough to manage the data efficiently and flexible enough to be used across very different file designs.

The initial class design is shown in Figure 5. Note that there are three fundamental classes. These are the Manager class, the Data class and the System class. An instantiation of the System class was intended to act as a means by which objects were created and tracked. In this way, the objects could be maintained along in parallel with the existing CHCS system. As objects reflecting individual records were looked up, they were compared against a master list and against the actual global reference. Objects that were no longer valid were automatically deleted.

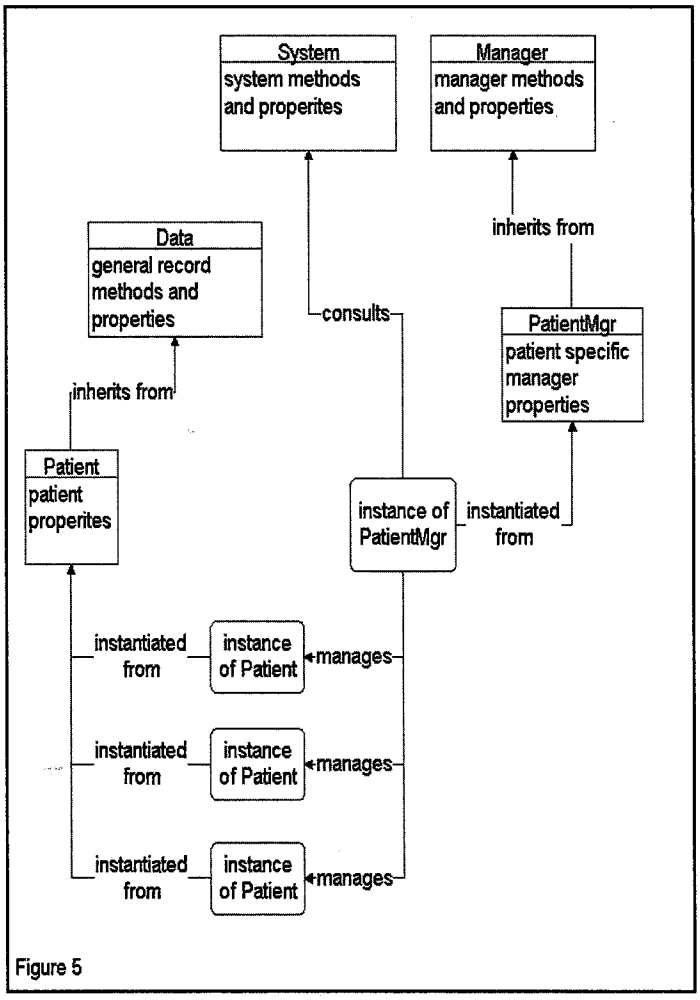

Figure 5. Initial Class relationship design

The remaining two root classes, Manager and Data, had to do with the management of the actual data. All databases have to support two essential sets of operations: 1) operations associated with the collection of components that describe a "record" and 2) operations associated with deriving a collection of "records."

The Manager class served up objects associated with particular criteria. The Data class represented all of the data associated with a "record." The intention was to make the Manager and Data classes as general as possible and to delegate the specialization of the data to the specific class. In this design, every File Manager file exposed in this system would have two classes representing it, a Data class and a Manager class.

Objects are often classed as "light" or "heavy" according to the amount of data contained in their state. We intended that each object would be as light as possible.

In the case of the Manager classes, the state could be made very light indeed. Each instantiation of a Manager class required only four pieces of information. This made it possible to take advantage of the EsiObjects virtual class construction.

A virtual class has no real state in the normal sense of the term; virtual classes have no instance variables. Essentially, all methods in a virtual class must be able to operate without any information that's not immediately available to them. Such information must either be passed in or be present in a special variable ($ZVIRDATA) that is potentially unique to any virtual object. Information can be associated with a virtual object at the time of creation. This is done by passing a string to the CREATE command. Later, the code of the virtual object can retrieve that string by referencing $ZVIRDATA. In effect, a virtual object can be viewed as an object that has a single instance variable: $ZVIRDATA. This approach was sufficient to pass in all the information to an instantiation of a Manager class.

We determined that each Manager class would have a "weight" of at most a hundred or so bytes. Since each file and subfile would have only a single Manager object, the cumulative weight would be at most a few hundred K bytes. This was not considered excessive in a database of several gigabytes.

In the case of the Manager classes, we were helped by the regularity of the File Manager lookup. The File Manager lookup process is very complex but it is also common to all of the files. Consequently, the specialization of screens, filters, etc., could be preserved in methods for each of these operations. For example, the Internal::Filter method is used to mimic the action of a File Manager screen. The method returns either a 0 (failure) or a 1 (success) to determine whether or not to include a particular object in a lookup collection.

In figure 6, the method "Internal::Filter" is implemented in the root Manager class as merely a "Q 1", returning success on any object. This is the default. Its descendant, the PatientMgr class, has code associated with it that further specializes the process. Since the Internal::Filter method of the PatientMgr class overrides the Internal::Filter in the root Manager class, when an instantiation PatientMgr class executes a lookup, the PatientMgr Internal::Filter method is executed. For files that have no screen, the Internal::Filter method for that class would never be implemented and the Internal::Filter of the root Manager class would be executed.

This is a general pattern of implementation development not only in EsiObjects but also in other object oriented languages. The process is to 1) determine the root or default form of the method. This can be as simple as a "Q 1". 2) Specialize the method for the individual classes. 3) Design general code to use the method by name and parameter, neglecting the specifics of how it is implemented. In this way, the same code can reference the appropriate method. The method itself can be anywhere in the inheritance hierarchy. In the example of Figure 6, for any given lookup class, the method can either be local, such as in the PatientMgr, or in the parent root Manager class.

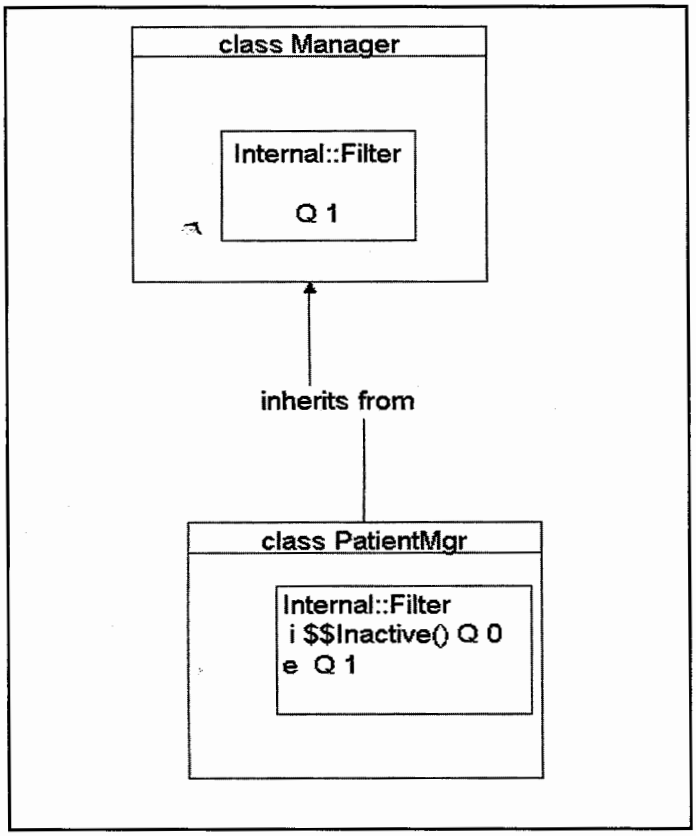

Figure 6. Implementation of "Internal::Filter" Method

The Data classes were more difficult to implement than the Manager classes. They had to be heavier since they had more specific information to maintain. For example, the Data root class had an internal variable and matching property to contain the "root" of the object: the actual global reference. Methods across the class used subscript indirection to retrieve global information for processing. For example, in the following structure,

```
^DPT(n,1)=data^data^date of birth^data
```

Date of Birth is contained in the third piece of the structure, the number 1 is the node specifier and n is the File Manager Internal Entry Number. Consequently, for an IEN of 50, the instance variable I%Root would be:

```
I%Root="^DPT(50)"
```

In the property, DateOfBirth, in the value accessor, we might expect to see the code:

```
S A%Value=@I%Root@(1)
S A%Value=$P(A%Value,"^",3)
```

These sorts of retrievals were created in the classes by the compiler.

The method from then on would have the data contained in the method variable A%Value to operate against. At the time the object was created, the value of I%Root would be set. From then on, it would be available for use by the methods. Because I%Root was encapsulated according to the standard object paradigm, an individual instance of an object could guarantee that I%Root was always unique to itself. Therefore, we could use I%Root with impunity without worrying that it would conflict with the state of some other object.

We intended to keep the state (the instance variables) as light as possible. Ultimately, instance variables are stored in globals and take up space. Also, the number of objects was to be multiplied considerably since a File Manager record would have a corresponding object to represent it.

Since we were using data already existing in globals, the value of the instance variables serves only to map a path to the actual data. The majority of the intelligence in the class was located in the method and property code. We only kept information such as the root, the internal entry number, the object id, etc., local to the specific object. For this reason, though the Data classes were not as lightweight as the Manager classes, we did not feel they were prohibitively heavy.

File Manager records were now represented by a corresponding class and were accessible by a call to a single instance of a class that was designed for that purpose. However, as we were working on this project, it soon became clear that a façade class would be useful.

A façade is a class whose purpose is to present a particular API to the caller, while isolating the caller from the complexity of the underlying activity. Façades were developed originally to present a common interface over dif-

fering subsystems; for example, a single API to communicate with both M and a relational database. However, we extended the concept to present differing views of the database, thereby hiding the complexities such a view represents. A full explanation of façades, as well as other design patterns, can be found in Gamma's fine book, *Design Patterns: Elements of Reusable Object-Oriented Software.*

Recall that in this proof of concept project, we were first building a limited representation of the patient database, MiniReg. We realized that a MiniReg operation had its own issues and limitations such as completeness, a pseudo-transaction structure and combined representation. For example, if in the validation, a single property failed, it should fail the entire registration so that the database was not erroneously incomplete.

For this reason, we developed the concept of a client class. A client class is not compiled as are the Manager classes or the Data classes. Instead, client classes are developed for specific needs. A client class could be considered analogous to the relational database concept of a view. The client class MiniReg did not inherit from either the Manager or Data classes; it had no direct parent classes at all.

Figure 7 shows how these different representations operate in the live system. An instance MiniReg would have associated with it an instance of the Patient class. Each of the properties being set in MiniReg would be processed against the corresponding property. When the MiniReg object is first instantiated it loads itself from the Patient instantiation. The MiniReg is not persistent and therefore can process requests faster. MiniReg persisted the data by first validating it and then sending the data to Patient to be stored.

Conclusion

Several techniques, patterns and principles suggested themselves in the building of this system.

Implement generally. We found ourselves following one scenario in design over and over again. First, we would decide we would need a particular method in a particular class. After we had decided what was needed, a general way to represent the same information presented itself. This was then incorporated in the more general parent class.

Use inheritance intelligently. This is the corollary to

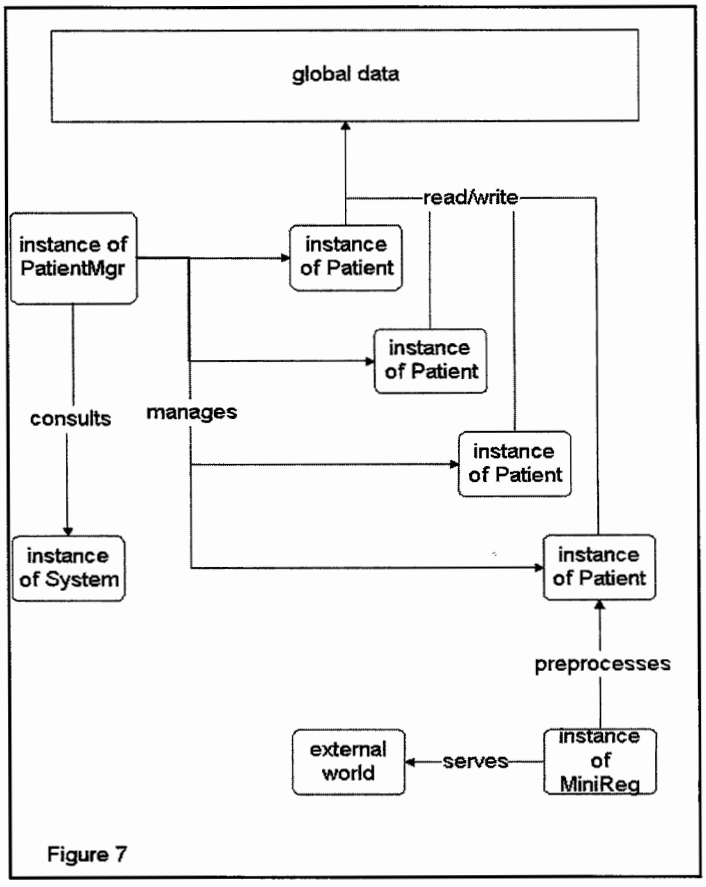

Figure 7. Operational Design

implementing generally. Each class should handle its own duty and only its own duty. Several times in the building of this product, we inadvertently built a method or property that did not directly apply to the purpose of the class. This always became problematic and needed to be rectified.

Preserve the integrity of the interface. By this, we mean the designed intention of the API. In our design, we used the EsiObjects Primary interface to represent the properties that directly represented File Manager fields and then created other interfaces to support the operation of methods and properties of the Primary interface. Several times as we were figuring out the best path, we compromised the primary interface with the best of intentions. It always came back to haunt us. The cost of violating design principles is much higher in OO than in other paradigms.

Use state intelligently. In our project, state was very important. Had it gotten out of control we could have had a wrapper of a database that was as large as the database itself. We were helped immeasurably by the encapsulation feature of EsiObjects. Encapsulation enabled us

to guarantee that the state of a given object was unique to that object. This simplified the required code for each property to the point that it could be easily compiled.

Façades trump complexity. The MiniReg class served as an intelligent façade for the Patient class. Initially, we built it as a means by which we could test the product. Quickly, the MiniReg class became the preferred mechanism for mini-registration. Because the difficult database work was maintained in the data class, the MiniReg class was simple enough to include all of the checks we needed for MiniReg as a whole. It made the testing and the operation of the product much simpler. The MiniReg façade hid the complexity and exposed only what the caller needed. M

Steven Popkes has been a software engineer for twenty years, most recently as a consultant for ESI Technology Corporation in Natick, Massachusetts. He has worked extensively in M, C++ and EsiObjects. He is now an employee at Concept 5, in Burlington, helping develop a CORBA-compliant security system. He can be reached at spopkes@concept5.com.

Strategies & Solutions Conference
September 27-29, 1999 San Diego

Strategies & Solutions Info: Fees, Program, Solutions Center, Laptop Demos

Registration Fees: We're delighted to announce that this year's conference fees are greatly reduced, and there will be no separate fees for tutorials! Online registration will be available by late June.
Registration by August 27: $495 members, $560 nonmembers.
Registration after August 27: $595 members, $660 nonmembers.

Special Discounts:
Distinguished Member Employees: Register 5 employees, pay for the first 3 and send 2 free!
Organizational Member Employees: Register 3 employees, pay for the first 2 and send 1 free!

Solutions Center Fees: Tabletop Displays, two days, $500 for MTA Distinguished Members; $600 for MTA Organizational Members; $800 for all others.

Program: The S&S program focuses on the new skills and related technologies needed to use M technologies to best advantage. It includes M and the Web, Linux, case studies on new approaches to legacy systems, and much more.

Solutions Center: As the central S&S hub, the Solutions Center will offer vendor table-top displays, refreshments during breaks, a place to network with colleagues and new acquaintances.

Laptop Demos: What solutions have your found lately? Contact MTA if you'd like to offer a brief laptop demo of the solutions that are working for you. Check out the MTA Website for the full conference program in late June. We're looking forward to seeing you in San Diego!