Developing Fusion Objects for Instruments

The successful application of object-oriented technology to real-world problems is a nontrivial task. This is particularly true for developers transitioning from nonobject-oriented methods to object-oriented methods. Key factors that improve the probability of success in applying object-oriented methods are selecting an object-oriented method, developing a process definition, and continually improving the process.

by Antonio A. Dicolen and Jerry J. Liu

Object-oriented technology is fast approaching mainstream status in the software community. Many software developers are interested in becoming object-oriented practitioners. Managers, once skeptical of its value, are considering its use in their business enterprises. This technology is old enough not to be a fad and new enough to be recognized by customers as high technology.

Within the embedded community (i.e., microprocessor-based instrumentation) at HP, there is significant interest in adopting object-oriented technology for the development of new products. However, the adoption rate of object-oriented technology at HP has been hampered by earlier negative experiences. Attempts to use object-oriented technology in instruments occurred as early as the mid 1980s. At that time the technology was in its infancy. The methods for employing the technology were immature and the development tools necessary for its effective use were nonexistent. Application of the technology at that time resulted in unmet product requirements.

These experiences hindered further development using object-oriented technology. Object-oriented technology became synonymous with slow speed, high risk, and failure. This perception imprinted itself on the culture of HP divisions using embedded software technology. It was not until the early 1990s that this perception began to change. As engineering productivity became an issue for management, software reuse emerged as a possible solution. With reuse as a business goal, an object-oriented approach was once again considered as a means of achieving that goal.

It is important to recognize that reuse and object-oriented technology are not synonymous since it is possible to achieve reuse without an object-oriented approach. Software math libraries are a prime example of this fact. This type of reuse is called *library* reuse. It is the most common and the oldest form of software reuse. *Generative* reuse, such as that provided by tools like lex and yacc, is another form of software reuse. In general these tools use a common implementation of a state machine and allow the user to modify its behavior when certain states are reached.

Another type of reuse is *framework* reuse. Microsoft® Windows' user interface is an example of framework reuse. In framework reuse, the interaction among the system components is reused in the different implementations of the system. There may be certain common code components that some, but not necessarily all, of the implementations use. However, the framework is what all these systems have in common. Microsoft foundation classes are an example of common code components. Menu bars, icon locations, and pop-up windows are examples of elements in the framework. The framework specifies their behaviors and responsibilities.

One reuse project based on this approach was a firmware platform for instruments developed at our division. The goal was to design an object-oriented firmware framework that could be reused for different instruments. With this project, we hoped to use object-oriented technology to address reuse through framework reuse. We chose to use Fusion,[1,2] an object-oriented analysis and design methodology developed at HP Laboratories, to develop our instrument framework.

In this article, we first describe the firmware framework and our use of the Fusion process. Next we present our additions to the analysis phase of the Fusion process, such as object identification and hierarchical decomposition. A discussion of the modifications to the design phase of Fusion then follows, including such topics as threads and patterns. We conclude with the lessons we learned using Fusion.

Firmware Framework

The new firmware framework is an application framework. An application framework provides the environment in which a collection of objects collaborate. The framework provides the infrastructure by defining the interface of the abstract classes, the interactions among the objects, and some instantiable components. A software component, or simply a component, is an

atomic collection of source code used to achieve a function. In many situations, a component will have a one-to-one correspondence with a C++ object. At other times, a component may be made up of multiple objects implemented in C++ or C source code.

Users of the firmware framework contribute their own customized versions of the derived classes for their specific applications. Note that the framework approach is very different from the traditional library approach. With the library approach, the reusable components are the library routines, and users generate the code that invoke these routines. With the framework approach, the reusable artifacts are the abstractions. It is their relationships to one another, together with the components, that make up the solution to the problem.

The firmware framework contains a number of application objects. These are different kinds of applications that handle different kinds of responsibilities. The responsibilities of these application objects are well-defined and focused. For example, there is a spectrum analyzer application that handles the measurement aspects of an instrument and also generates data, a display application that is responsible for formatting display data, and a file system application that knows how to format data for the file system.

There is always a root application in the system, which is responsible for creating and destroying other applications and directing inputs to them. Other components of the application framework include the instrument network layer and the hardware layer. The applications communicate with each other via the instrument network layer. The hardware layer contains the hardware device driver objects, which the applications use through a hardware resource manager. Fig. 1 shows an overview of the firmware framework.

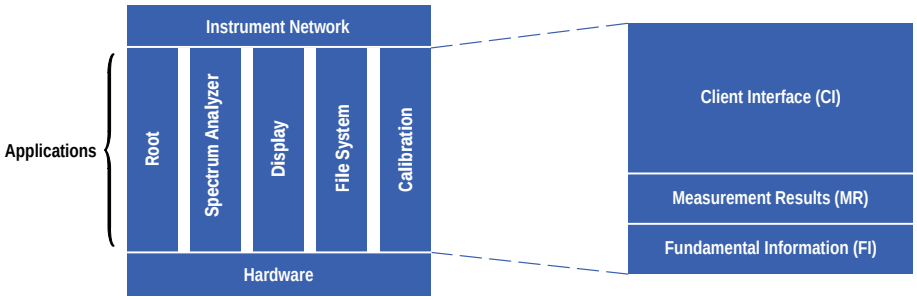

Fig. 1. *An overview of the new firmware framework.*

Application Layers

An application in the firmware framework is a collection of objects organized into three layers: client interface, measurement results, and fundamental information. These layers deal with information at different levels of semantics. The semantics at the client interface layer deal with instrument functionality while the semantics at the fundamental information layer are more related to areas such as hardware control.

Client Interface Layer. This layer represents an abstraction containing user-selectable parameters, the interface for setting these parameters, the results, and the sequence for generating the results. Thus, the client interface layer defines the features and the capabilities of an application. It is responsible for maintaining application state information and creating the requested results. This layer also contains a collection of application parameter objects that store the state of the application, and a dependency manager that manages the parameter limiting and coupling dependencies. The dependency manager also triggers events on state changes. These state changes cause the selection of the correct MeasurementResult to use to satisfy the user's request.

Take, for example, a simplified multimeter instrument. It could be an ohmmeter, a voltmeter, or a current meter. To select the voltmeter mode, the instrument software must deselect the ohmmeter or current meter mode and then select the voltmeter mode. The user interface simply turns on voltmeter mode. The dependency manager knows that these three modes are mutually exclusive and automatically sets the current meter and ohmmeter modes to off. In addition, the user could set the measured voltage to be the average value or the rms (root mean square) value. This corresponds to the selection of a specific MeasurementResult that provides the information the customer is interested in.

Measurement Result Layer. This layer is made up of objects derived from a base class called MeasurementResult. These objects contain the measurement algorithms that specify the methods for combining raw data into meaningful data.

MeasurementResult objects subscribe to and respond to events in the client interface layer and in other MeasurementResult objects. Complex measurement results contain simple MeasurementResult objects. Examples of MeasurementResult objects in an instrument application are SweepMR, MarkerMR, and LimitLineMR. These could be be measured values from a spectrum analyzer. An example of a MeasurementResult object in a display application could be a TraceDisplayItem that knows how to read a MarkerMR and generate marker information for the display.

The measurement result layer has no knowledge of how or where its input data is generated. Its input can come either from other MeasurementResults or from the fundamental information layer. It is thus free of any hardware dependencies. This layer uses the fundamental information layer to coordinate the hardware activity.

Fundamental Information Layer. This layer performs the specific activities that orchestrate the hardware components to achieve a desired result. The objects in the fundamental information layer know about specific hardware capabilities. They keep the hardware objects isolated from each other and also generate self-describing, hardware-independent data. The fundamental information layer applies hardware corrections (e.g., compensations for hardware nonlinearities) to the measured results.

The fundamental information layer contains three major components: a state machine with sequencing information that controls the objects in the layer, a production object that is responsible for orchestrating the hardware components, and a result object that is responsible for postprocessing data. Examples of fundamental information layer objects include SweepFI, which is responsible for measuring frequency spectra in a spectrum analyzer application, and the display list interpreter in the display application, which is responsible for controlling the instrument display.

Instrument Network

The instrument network contains the objects that facilitate interapplication communication, including an ApplicationArchive object, which is responsible for naming and providing information to applications, and an ApplicationScheduler object that schedules the threads that make up the applications.

Hardware Layer

The hardware layer contains the objects that control the instrument hardware. These objects contain very little context information. There are two types of hardware objects: device objects, which drive a simple piece of hardware, and assembly objects, which are collections of hardware objects. Hardware components are organized in a hierarchy much like the composite pattern found in design patterns.* Hardware objects are accessed through handles like the proxy pattern described in the patterns book.[3] Handles can have either read permission or read-write permission. Read permission means that the client can retrieve data from the object but is not able to change any of the parameters or issue commands. Read-write permission allows both. Permissions are controlled through the hardware resource manager.

Communication Mechanisms

Two main communication mechanisms glue the architecture together: agents and events. Agents translate the language of the user (client) into the language of the server (application). Different kinds of agents apply different kinds of translations. For instance, a client may enter information in the form of a text string, while its target application may expect a C++ method invocation. Thus, the client would use a specialized agent to translate the input information into messages for the target application (the server).

Events are mechanisms used to notify *subscribers* (objects that want to be notified about a particular event) about state changes. We decided to use events because we wanted to have third-party notification, meaning that we did not want the *publishers* (objects that cause an event) to have to know about the subscribers.

There are two types of events: active and passive. Active events poll the subject, whereas passive events wait for the subject to initiate the action. Our event mechanisms and the concepts of subscribers and publishers are described in more detail later in this paper.

Use of Fusion

In selecting an object-oriented method to guide our development, we were looking for a method that would be easy to learn and lightweight, and would not add too much overhead to our existing development process. We were a newly formed team with experience in our problem domain and in embedded software development, but little experience in object-oriented design. We wanted to minimize the time and resources invested in working with and learning the new technology until we were fairly certain that it would work for us. At the same time, we wanted to have a formal process for designing our system, rather than approach the problem in an ad hoc manner.

Fusion (Fig. 2) met these requirements. It is a second-generation object-oriented methodology that is fairly lightweight and easy to use.[4]

For the most part, our use of Fusion was very straightforward. We started with the system requirements, and then generated a draft of the system object model and the system operations of the interface model. We also generated data dictionary entries that defined our objects and their interrelationships. These documents made up the analysis documents. We did not develop the life cycle model because we did not see how it contributed to our understanding of the system. As time went on, we discovered that we really did not need it.

* Design patterns are based on the concept that there are certain repeated problems in software engineering that appear at the component interaction level. Design patterns are described in more detail later in this article.

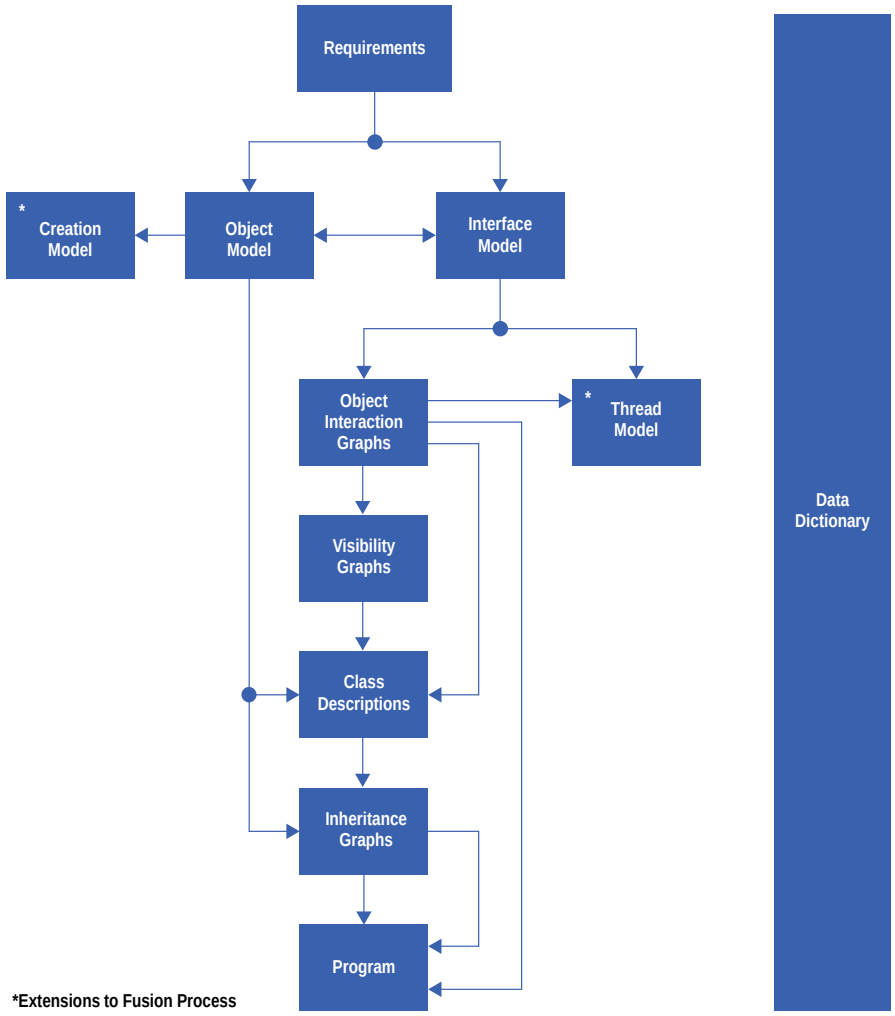

Fig. 2. *The Fusion process for software development.*

From the analysis model, we mapped the analysis onto a design and generated the object interaction graphs to show the interactions between the objects. We then generated the visibility graphs and derived the class descriptions. These were straightforward processes.

By no means did we go through this entire process in one pass. For us, using Fusion was an iterative process. Our system was clearly too large to analyze and design in one pass. If we had tried, we would have been overwhelmed with the details. Instead, we made a first pass to identify the primary objects. We then divided the system into subsystems and recursively applied the Fusion method to each subsystem level to discover the higher-order objects at that level.

For instance, at the topmost level we identified the major components of the firmware framework: the client interface layer, the measurement result layer, and the fundamental information layer (see Fig. 3). We then sketched out the interactions between these components, repeated the process for each of the subsystems, and explored the details within each of the components of the subsystems.

We did not apply the iterative process simply to find details. It was also a way to check the top-level analysis and design and feed back into the process anything that we had overlooked in the higher-level passes. These checks helped to make our system implementable. Through external project reviews with object-oriented experts, we also discovered other ways to look at our abstractions. For instance, with our original analysis, our focus was on the subsystem that performed the measurement functionalities of the instruments. Thus, we ended up with an architecture that was focused on measurement. We had layers in the system that handled the different aspects of obtaining a measurement, but few layers that supported the instrument firmware. It was not until later, with outside help, that we saw how the patterns and rules for decomposing the instrument functionality into layers applied equally well to subsystems that were not measurement related, such as the display or the file system. We were also able to abstract the different functionalities into the concept of an application and use the same rules and patterns to decide how the responsibilities within an application ought to be distributed.

We found Fusion to be an easy-to-use and useful methodology. This method provided a clear separation between the analysis and the design phases, so that we were able to generate system analyses that were not linked to implementation details.

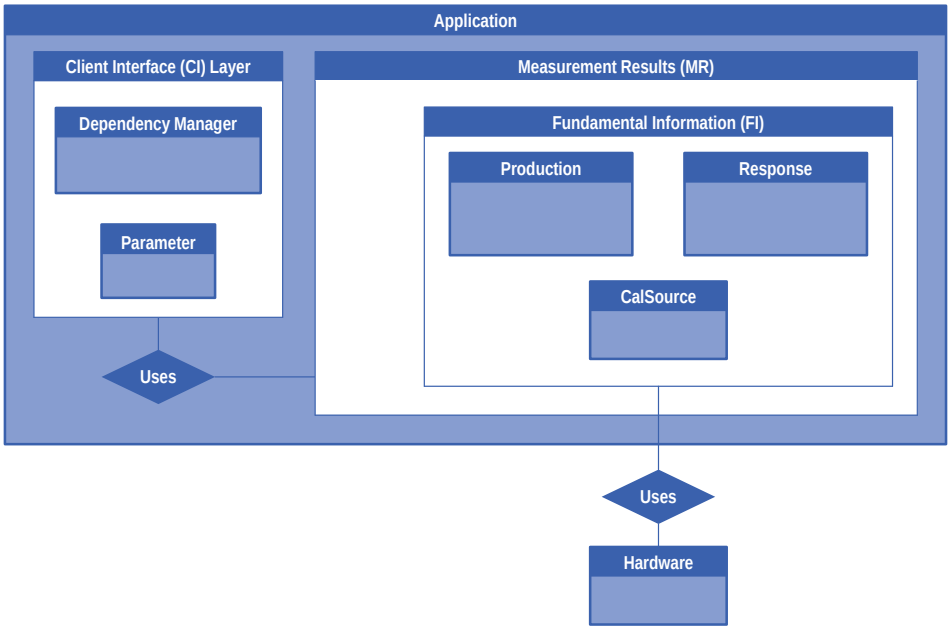

Fig. 3. *The object model for the client interface, measurement results, and fundamental information objects.*

Of course, no methodology is perfect for every situation. We made some minor modifications to the method along the way, as well as some extensions (see Fig. 2), which will be described later. For instance, we omitted the life cycle models. Since we knew that we were going to implement our system in C++, we used C++ syntax to label our messages in the object graphs and C++ class declarations when we generated the C++ classes. We also did not use the state diagram portions of Fusion to generate states for our state machines. We felt that we did not need this state machine facility and thus freed the staff from having to learn yet another notation.

Extensions to Fusion—Analysis Phase

In our desire to perform object analysis more consistently, our team developed extensions to Fusion that helped non-object-oriented practitioners make the paradigm shift to the object-oriented mind-set much more easily.

Many developers and managers naively assume that a one-week class on object-oriented technology is sufficient to launch a team into developing object-oriented software. While this may be a necessary condition, it is not sufficient for the successful acquisition and application of object-oriented technology.

Many texts and courses on object-oriented methods treat the analysis phase as merely the identification of nouns that are objects and their relationships with one another. Having conveyed this, the analysis sections of these books then focus on method notation rather than helping the novice overcome the biggest obstacle in object-oriented analysis, the identification of objects.

Without sufficient help, novices produce analysis diagrams that conform to the object notation, but merely recast ideas from either the structured analysis paradigm or from some home-grown paradigm. The circles of structured analysis and design are turned into boxes and, voila, an object diagram is born.

Our team was not spared this experience. Fortunately, we consulted object-oriented experts who taught us what to do. Thus, we developed an analysis technique that could be consistently applied project-wide to help the developers transition from structured to object-oriented analysis. This was critical to our facilitating software reuse, the primary goal of the project.

Object Identification

Successful object-oriented analysis begins with identifying a model that captures the essence of the system being created. This model is made up of behaviors and attributes that are *abstractions* of what is contained in the system to accomplish its task.

What makes a good abstraction? The answer to this question is critical to the effective use of object-oriented technology. Unfortunately, identifying the wrong abstraction encourages a process known as “garbage in, garbage out.” Furthermore, the right abstraction is critical to the ease with which a developer can implement the object model. It is possible to generate a proper object model that cannot be implemented. The key is in the choice of the abstraction.

What makes an abstraction reusable? The answer to this question is critical to achieving the value-added feature of object-oriented technology that is needed to achieve software reuse. Understanding the context in which reuse can occur is important.

An analysis framework exists that can be used to guide the identification of abstractions. This framework has the added benefit of guaranteeing that the resultant object model derived from its use is realizable. Furthermore, its foundation is based on the premise that software reuse is the ultimate goal.

In developing our analysis, we noted the questions the experts would ask when presented with our work. Fundamentally, their questions focused on understanding the responsibilities of the abstractions that we had identified. Responsibility, it turns out, gives rise to the state and behavior of an object. Previous research on this topic yielded an article[5] that discusses responsibility-based design, and describes an object-oriented design method that takes a responsibility-driven approach. We synthesized this knowledge into what can be described as *responsibility-based analysis*.

This new analysis technique is based on a pattern of three interacting abstractions: the *client*, the *policy*, and the *mechanism*. Fig. 4 illustrates the object model for the client-policy-mechanism framework.

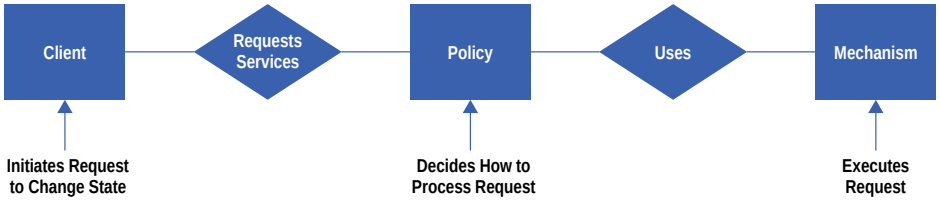

Fig. 4. *The object model for the client-policy-mechanism framework.*

The client abstraction requests services, initiates activities that change the system state, and queries for request-specific status within the system.

The policy abstraction decides when and how a request will be acted upon. It accepts the client request and, based on the responsibility given to it by the analyst, chooses the appropriate way in which the work will be done. In performing this responsibility it sets the context for the relationships between the system components.

The mechanism abstraction actually performs the change to the system state. If the operation is a state query, it returns the desired information. It does not return context information related to the operation being discussed. The mechanism abstraction makes no decision as to whether it is appropriate for it to perform an operation. It just does it.

As an example, consider creating a software application to read the current market value of HP stock. The client-policy-mechanism analysis of the problem, at a very high level, yields at the minimum three abstractions: an abstraction representing the user (the client), an abstraction that represents when and how the HP stock information is to be acquired (the policy), and lastly, an abstraction that knows about the value of HP stock (the mechanism). The mechanism abstraction, when implemented, becomes the software driver for acquiring the stock price. In one instance, the mechanism object reads the value of HP stock from a server on the Internet via telnet. In another instance, the mechanism acquires the stock value via http. (Telnet and http are two internet communication protocols.) The policy abstraction determines how often to access the mechanism. In our case it determined how often, that is, the time interval used, to poll the mechanism. The client object receives the resultant information.

From a software reuse perspective, mechanism abstractions are the most reusable components in a system. Mechanisms tend to be drivers, that is, the components that do the work. Since the responsibility of a mechanism is context-free, the work that it does has a high probability of reuse in other contexts. Being context-free means that it does not know about the conditions required for it to perform its task. It simply acts on the message to perform its task. In the example above, the mechanism for acquiring the stock price can be used in any application requiring knowledge of the HP stock price.

Though not as obvious, using the client-policy-mechanism framework gives rise to policy abstractions that are reusable. In the example above, the policy abstraction identified can be reused by other applications that require that a mechanism be polled at specific time intervals. Making this happen, however, is more difficult because the implementer must craft the policy abstractions with reuse in mind.

The analysis technique described above attempts to identify client, policy, mechanism, and the contexts in which they exhibit their assigned behaviors. When policy roles are trivial, they are merged into the client role, producing the familiar client/server model. This reduction is counterintuitive, since most client/server model implementations imbed policy in the server. However, from a software reuse point of view, it is important to keep the server a pure mechanism. On the other hand, it is also important to resist the temptation to reduce the analysis to a client/server relationship. Doing so reduces both the quality of the abstractions and the opportunity for reusing policy abstractions.

These three abstractions together define the context of the problem space. Experience has shown that to produce a clean architecture, it is important for each abstraction to have one responsibility per context. That is, a policy abstraction should be responsible for only one policy, and a mechanism abstraction should be responsible for doing only one thing.

On the other hand, abstractions can play multiple roles. In one context an abstraction may play the role of mechanism and in another context be responsible for policy. An example illustrates this point more clearly. Consider the roles in a family unit. A young child performs requests made by a parent who in turn may have been asked by a grandparent for a specific activity. In a different context, for example, when the child grows up, it plays the role of parent for its children and its parents, who are now grandparents. In this latter setting, the parents are the policy makers, the grandparents are the clients, and the children (of the child now grown up) are the mechanisms (see Fig. 5).

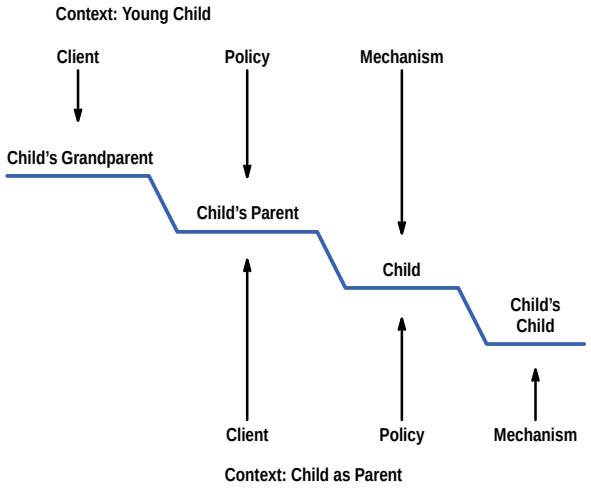

Fig. 5. *The client-policy-mechanism model as applied to a family unit.*

Just as, depending on context, a specific individual plays different roles, so it is true with abstractions. In one context an abstraction may be a mechanism and in another, a policy. The critical rule to keep in mind when using the client-policy-mechanism framework is that there should only be one responsibility per abstraction role.

Hierarchical Decomposition

Another example of systems that illustrate the single-role- per-context rule is found in the hierarchy of the military forces. In the United States, the commander in chief issues a command, the joint chiefs respond to the command and determine the methods and the timing for executing the command, and the military forces complete the task. In a different context, the joint chiefs may act as clients to the admiral of the navy who determines the methods and timing for the subordinates who execute the task (see Fig. 6).

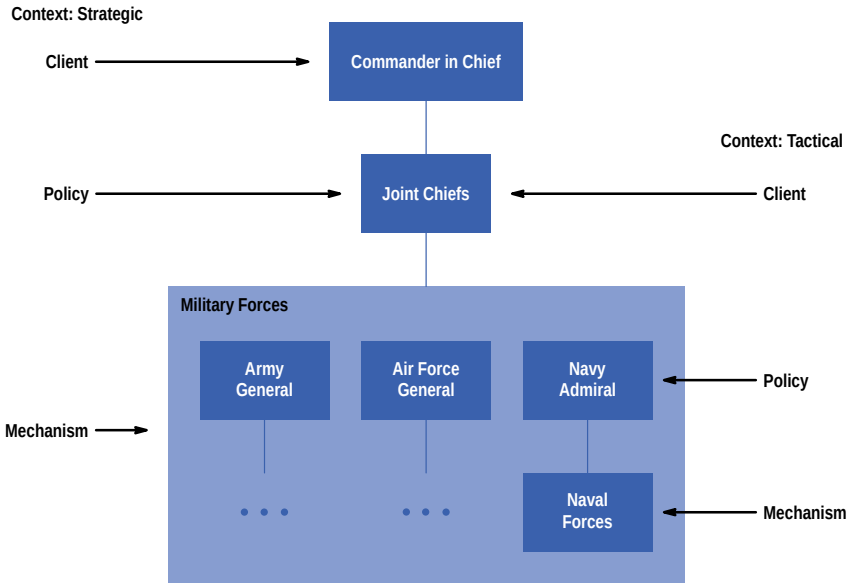

Fig. 6. *The client-policy-mechanism model as applied to a military hierarchy.*

In each of these examples there is a client, a policy, and a mechanism. In one context, a person is responsible for policy decisions. In another, the same person is responsible for mechanism or client activities. It is this concept that gives rise to the use of the client-policy-mechanism framework in helping to perform hierarchical decomposition of a problem domain. The repetitive identification of roles, contexts, and responsibilities at ever finer levels of granularity helps identify the solution space for the problem domain.

The firmware framework team performed hierarchical decomposition by identifying roles, contexts, and responsibilities. These responsibilities defined abstractions that produced objects and groups of objects during the implementation phase. In the early phases of our novice object-oriented project, it was expedient to use the words object and abstraction interchangeably. As the team gained experience and became comfortable with object-oriented technology and its implementation, the distinction between the abstraction and its resulting objects became much better appreciated.

The analysis technique based on the client-policy-mechanism framework resulted in a hierarchical decomposition that yielded layers and objects as shown in Fig. 7. Layers are groups of objects that interact with one another to perform a specific responsibility. Layers have no interfaces. Their main function is to hold together objects by responsibility during analysis to facilitate system generation. For example, many software systems include a user interface abstraction. However, upon problem decomposition, the user interface abstraction typically decomposes into groups of objects that collaborate and use one another to satisfy the responsibilities of the user interface. When the abstraction is implemented, it usually does not produce a single user interface object with one unique interface.

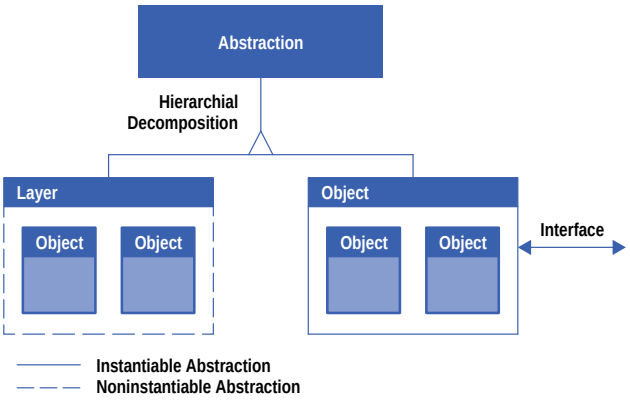

Fig. 7. *An abstraction decomposition.*

Much of this may not be discovered or decided until the design phase. However, knowing about it in the analysis phase maximizes the identification of abstractions and the completion of the analysis.

Creation Model

Many times discussions about abstractions resulted in intangibles that were difficult to grasp. To alleviate this problem, the team supplemented Fusion with a dependency model showing object dependencies and indicating when objects should be created. This provided developers with a concrete picture of which objects needed to be available first.

Consider again the HP stock price application. Let the mechanism object be represented by object A and let the policy object be represented by object B. Fig. 8 represents a creation model for the objects under discussion. It shows that object A has to be created before object B. This means that the mechanism for acquiring the HP stock price is created first. The object that determines how often to acquire HP stock price can only be created after object A. This example creation model is one of several that were discussed during the analysis phase to clarify the roles of the abstractions.

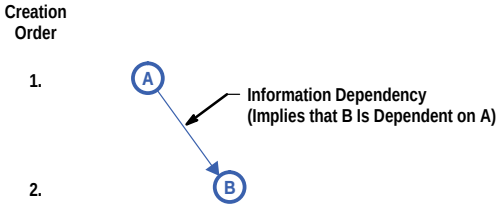

Fig. 8. *A creation model.*

Extensions to Fusion—Design Phase

We made extensions to the Fusion process with threads, design patterns, and reuse.

Threads

Our most extensive modifications to Fusion in the design phase were in the area of threads. Our real-time instrument firmware systems, which are very complex, must deal with asynchronous events that interrupt the system as well as send control commands to the measurement hardware. For example, measurement data from the analog-to-digital converter must be read within a certain time period before it disappears, and there may also be measurement hardware that needs to be adjusted based on these dynamic data readings.

There are also many activities going on in the system that may or may not be related. For example, we may want to have a continuous measurement running at the same time that we run some small routine periodically to keep the measurement hardware calibrated. Traditionally, a monolithic code construct performs all of these tasks. However, since some of these activities may only be peripherally related, it makes more sense to place these tasks in different threads of execution. Each thread of execution can be thought of as a path through the code. These threads of execution may be either regular processes or lightweight processes, and they may or may not share resources. In this paper, the term thread is used to mean a thread of execution, not necessarily to denote the preference of a lightweight process over a regular one. For instance, it would make sense to keep the task that performs the measurements separate from the task that checks the front panel for user input.

Fusion provides us with information on how to divide the behavior of the system into objects, but Fusion does not address the needs of our real-time multitasking system. It does not address how the system of objects can be mapped into different threads of execution, nor does it address the issues of interprocess communication with messages or semaphores. Lastly, no notation in Fusion can be used to denote the threading issues in the design documents.

Thread Factoring

We extended Fusion thread support in two ways. First, in the area of design we tried to determine how to break the system into different threads of execution or tasks. Second, in the area of notations we wanted to be able to be able to denote these thread design decisions in the design documents.

Our main emphasis was on keeping these extensions lightweight and easy to learn and keeping our modifications to the minimum needed to do the job. We wanted a simple system that would be easy to learn, rather than a powerful one that only a few people could understand.

We adopted portions of Buhr and Casselman's work on time-thread maps to deal with thread design issues such as the identification and discovery of threads of control within the system.[6,7,8] In our design, a time-thread map is essentially a collection of paths that are superimposed on a system object model (see Fig. 9). These paths represent a sequence of responsibilities to be performed throughout the system. These responsibility sequences are above the level of actual data or control flows, allowing us to focus on the responsibility flow without getting involved in the details of how the exact control flow takes place. We then applied the process of thread factoring, as described by Buhr and Casselman, where we brought our domain knowledge to bear on decomposing a single responsibility path into multiple paths. These paths were then mapped into threads of execution throughout our system.

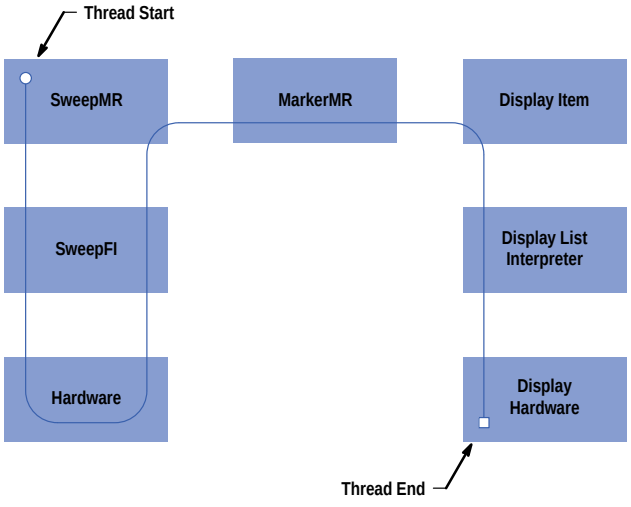

Fig. 9. *An example of a time-thread map.*

With the Fusion method, we had already identified the areas of responsibility. We then used this thread heuristic at the beginning of our design phase in those places where we had already identified the objects in the system, but where we had not yet designed the interaction among the objects. We dealt with the concurrency issues at the same time that we dealt with the object interaction graphs shown in Fig. 10. We also performed thread factoring and divided the system into multiple threads.

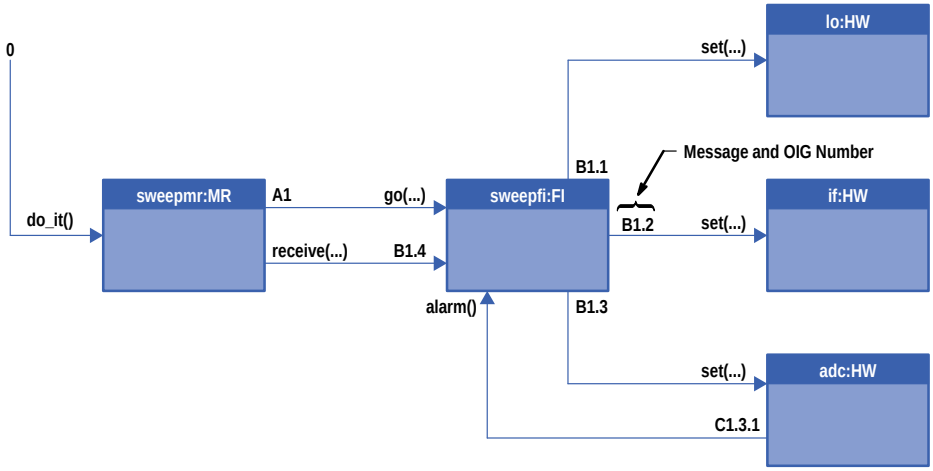

Fig. 10. *An object interaction graph (OIG). This representation is an extension of a Fusion object interaction graph. The letters in front of the OIG numbers associate a thread of execution with a particular message.*

The thread map in Fig. 11 depicts an example of thread factoring an application in our system. Using Fusion, we identified a path of responsibility through the objects CI, MR, and FI (client interface, measurement results, and fundamental information). Inputs enter the system through CI, and the responsibility for handling the input goes through the various layers of abstraction of MR and FI. Since information from the measurement hardware enters the system through FI, FI may have to wait for information. The information then flows goes back up fundamental information to MR and then possibly to other applications.

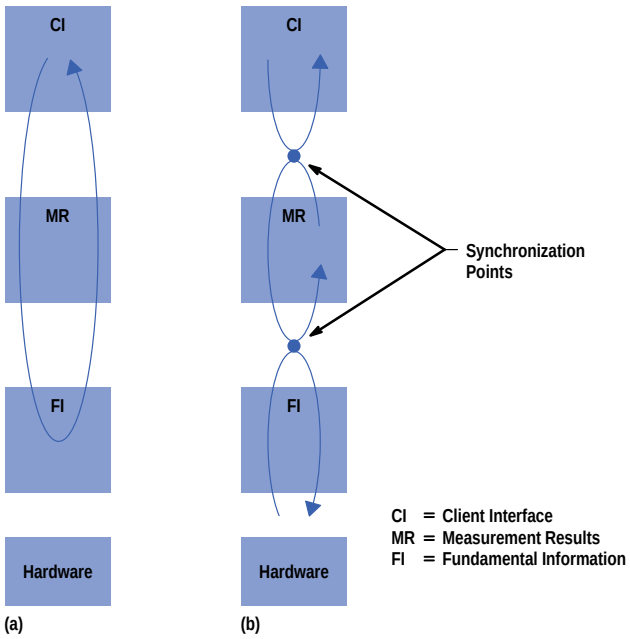

Fig. 11. *A thread map showing an example of thread factoring. (a) Before factoring. (b) After factoring.*

Clearly, the system worked fine as it was. However, we wanted to find where we could break the thread of execution and perform thread factoring. Many issues, such as questions about performance, were raised at this point. For example, if the thread is executing in part A of the system, it may not be available to perform services in part B of the system. Thus, in our system, we could have a thread pick up a user request to change the measurement hardware settings and then traverse the

code in the hardware setup routines to perform the hardware adjustments. However, while it was doing so, the thread would not be available to respond to user requests. This might impact the rate at which the system was able to service these requests. Therefore, we broke the user thread at the CI object boundary and gave that layer its own thread.

Next, we tried to find a place where we could break the thread that goes through MR and FI. Clearly, the place to break was between MR and FI. Making the break at this point gave us several flexibilities. First, we would be able to wait at the FI thread for data and not have to be concerned with starving MR. Second, developing components that were all active objects allowed us to mix and match components much more easily.

Mapping a system onto threads is a design-time activity. Thinking about the thread mapping at this stage allowed us to consider concurrency and the behavioral issues at the same time.

Thread Priorities

After we had identified the threads of execution, we needed to assign priorities to the threads. Note that this is mostly a uniprocessor issue, since priorities only provide hints to the operating system as to how to allocate CPU resources among threads.

In the firmware framework project, we took a problem decomposition approach. We reduced the architecture of our system to a pipeline of simple consumer/producer patterns (see Fig. 12). At the data source we modeled the analog-to-digital converter (ADC) interrupts as a thread producer generating data that entered the system with FI as consumer. FI, in turn, served as the producer to MR, and so forth. Inputs may also enter the system at the user input level via either the front panel or a remote device.

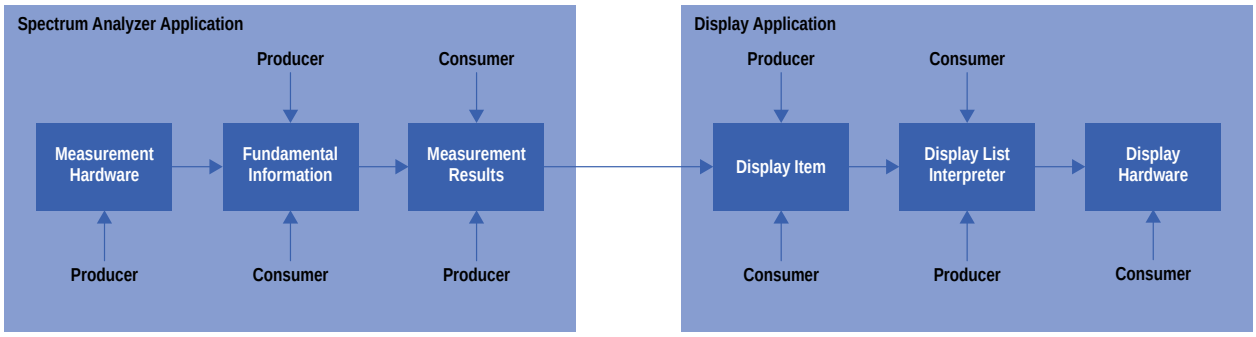

Fig. 12. *An example showing some of the producer/consumer chains used in the firmware framework project.*

We decided to give the highest priority to those threads that introduced data into the system from the physical environment so that they could respond to events in the environment quickly. Those threads included the user input threads and the ADC interrupt thread.

For thread priorities in the rest of the system, we considered three possibilities: that the producer priority was higher than that of the consumer, that the two priorities were equal, or that the consumer priority was higher than the producer priority. We ruled out setting the priorities to be equal because that would be equivalent to having no policy and would just let the systems run without any direction.

Making the producer priority higher than that of the consumer made sure that data was generated as quickly as possible. Unfortunately, since we continuously acquired data in our system, our data generation could go on forever unless we explicitly stopped the process and handed control to the higher level.

Alternatively, if we gave the consumer thread the higher priority, it would have priority over the producers with regard to CPU time. However, without the data generated from the producers, the consumers would block and be unable to run. Thus, if the data consuming chain had a higher priority than the data producers, the threads would run when data was available for them to process. This eliminated the necessity for the consumers to give up the CPU explicitly.

Threads and Synchronization

Another thread issue we considered was how to present the thread communication and synchronization operating system primitives to our system. We saw two alternatives. We could either expose the system level operating system calls to the system or encapsulate the operating system primitives inside objects so that the rest of the objects in the system could talk to these objects. For other system objects, it would be like communicating with nonoperating system objects.

We chose the latter approach. We created operating system objects such as tasks and semaphores to encapsulate operating system functionalities. This approach allowed us to model the operating system primitives as objects so that they would fit in well with the Fusion process and give us a clean model and good code reuse. This approach also had the side affect of

isolating our system from the operating system API. There were drawbacks with this approach, but they were not major. **Reference 7** contains more details about both of these approaches.

Thread Notation

We used thread notations within our Fusion diagrams in two ways. First, we used the thread map notations to show sketches of thread flows (Fig. 11). These simple notations gave us a general idea of the thread activities in the system. We adopted only a few of the notations that Buhr and Casselman use, employing the paths and waiting places that show the behavior of the system. We did not use their notation to handle the different types of synchronizations because we did not feel that this was the level of detail appropriate for what we needed. This method gave us an overview of what the system looked like without bogging us down in the details of how communication and synchronization were implemented.

For our second method of using thread notations, we extended the Fusion object interaction graph (OIG) notations to describe threads more formally (Fig. 10). We added letters in front of the OIG numbers to associate a thread of execution with a particular message. We also experimented with coloring the threads.

Design Patterns

Design patterns have become popular in the object-oriented world only recently. Design patterns evolved from the realization that certain software engineering patterns are repeated. These patterns are not at the implementation level, but at the level of component interactions. The idea here is to look at software design problems at a higher level so that recurring patterns can be identified and a common solution applied.

For instance, there is often a need in software systems for one or more objects to be notified if the state changes in another object. For example, if the value in a database changes, the spreadsheet and the word processor currently displaying that value need to change their displays. The *observer pattern*, described in the design patterns book,[3] shows how to set up the relationship among these objects. It describes when a pattern may be appropriate for solving the notification problem and some implementation hints and potential pitfalls.

Design patterns came to our attention a year or so into the project. By then, we had already completed most of the design. Therefore, we did not use them as templates to build the system from scratch. Instead, we used the design pattern catalog to validate designs. In looking through our system, we found potential applications for over half the patterns in the design patterns book. We then compared our design with those patterns.

We found patterns to be useful for design validation. In some places, they helped us improve the design. For instance, the hardware components are accessed through hardware handles, which are very similar to the protection proxies described in the patterns book. The hardware architecture itself is an example of a composite pattern. A composite pattern is an organization of objects in which the objects are arranged in a tree-like hierarchy in which a client can use the same mechanism to access either one object or a collection of objects. The descriptions of composite patterns in the design patterns book also helped us to identify and clarify some of the issues related to building composites.

In other areas in the system, we found our analysis to be more detailed because of our extensions to identify objects using the client-policy-mechanism framework. We have an event mechanism in the system to inform some component when an action has occurred. This mechanism is very similar to that of the observer pattern mentioned earlier. The observer pattern describes two components: a publisher and a subscriber, which define a methodology for handling events.

Our event pattern is slightly more sophisticated. We placed the event policies into a third object, so we have three components in our event pattern: a subscriber, an actor (publisher), and the event itself. Actors perform actions, and subscribers want to know when one or more actors have performed some action. The subscriber may want to be notified only when all of the actors have completed their actions. Thus, we encapsulated policies for client notification into the event objects. An actor is only responsible for telling events that it has performed some action. Events maintain the policy that determine when to notify a subscriber.

This arrangement gives more flexibility to the system because the design-patterns approach allows the policy for notification to be embedded in the actor. In our case, we also have the freedom to customize the policy for different instances of the same actor under different situations.

We feel that the main advantage of not using the patterns until the system design is done is that the developer will not fall into the trap of forcing a pattern that resembles the problem domain into the solution. Comparing our problem domain with those described in the patterns book helped us to understand more about our context and gave us a better understanding of our system. Also, as many other object-oriented practitioners have reported, we also found patterns to be a good way to talk about component interaction design. We were able to exchange design ideas within the team in a few words rather than having to explain the same details over and over again.

Scenarios

Part of our system requirements included developing scenarios describing the behavior of the system. Scenarios describe the system output behavior given a certain input. These scenarios are similar to the use cases described in reference 1 and are part of the Fusion analysis phase. However, for people not conversant in object-oriented methods, these scenarios often do

not have much meaning because the descriptions are far above the implementation level. Whenever we presented our analysis and design models, our colleagues consistently asked for details on what was happening in the system at the design level. Although Fusion documents provided good overviews of the system as well as excellent dynamic models for what happened in each subsystem, people still wanted to see the dynamics of the entire system.

To explain how our system works, we developed scenarios during the design phase. These scenarios were a collection of object interaction graphs placed together to show how the system would work, not at an architectural level but at a design and implementation level. We used the feedback we received from presenting the scenarios to iterate the design.

The Fusion model is event-driven, in that an event enters the system and causes a number of interactions to occur. However we had a real-time system in which events happen asynchronously. We needed scenarios that were richer than what the object interaction graph syntax could provide.

For example, our instrument user interface allows the user to modify a selected parameter simply by turning a knob, called the RPG (rotary pulse generator). One attribute by which our customers judge the speed of our instruments is how quickly the system responds to RPG input. The user expects to get real-time visual feedback from the graphics display. The empirical data suggests that real-time means at least 24 updates per second. As the layers were integrated, we looked at the scenario in which the user wanted to tune the instrument by changing a parameter (e.g., center frequency). This scenario led to questions such as: How would the system's layers behave? What objects were involved? What were the key interfaces being exercised? Were the interfaces sufficient? Could the interfaces sustain the rate of change being requested? What performance would each of the layers need to deliver to achieve a real-time response from the user's point of view? The answer to these questions led to a refinement of both the design and the implementation.

These design-level scenarios provided a better idea of what would happen in the system and presented a more dynamic picture. Since the scenarios encompassed the entire system, they gave the readers a better view of system behavior. We found them to be good teaching tools for people seeking to understand the system.

We also found that instance diagrams of the system objects helped us to visualize the system behavior. A diagram of the instantiated objects in the system provided a picture of the state information that exists in the system at run time.

Reuse

To build reuse into a system, the development method has to support and make explicit the opportunities for reuse. The analysis extensions described earlier serve to facilitate the discussion of reuse potential in the system. The design is driven by the biases encoded into the analysis.

At the end of the first analysis and design pass, an entity relationship diagram will exist and a rudimentary class hierarchy will be known. The more mature the team in both object-oriented technology and the domain, the earlier the class hierarchy will be identified in the development method. Additional information can be gathered about the level of reuse in the class hierarchy during the analysis and design phase. These levels of reuse are:

- Interface reuse
- Partial implementation reuse
- Complete implementation reuse.

The ability to note the level of reuse in the work products of the development method is valuable to the users of the object model. A technique developed in this project was to color code the object model. Fig. 13 shows two of these classes.

Except for defect fixes, complete implementation classes cannot be modified once they are implemented. This type of color coding aids developers to know which components of the system can be directly reused just by looking at the object model.

Process Definition

The pursuit of object-oriented technology by a team necessitates the adoption of formal processes to establish a minimum standard for development work. This is especially true if the team is new to object-oriented technology. Various members of the team will develop their understanding of the technology at different rates. The adoption of standards enables continuous improvements to the process while shortening the learn time for the whole team.

In the firmware framework project, we adopted processes to address issues like communication, quality, and schedule. We customized processes like inspections and evolutionary delivery to meet our needs. It is important to keep in mind that processes described in the literature as good practices need to be evaluated and customized to fit the goals of a particular team. The return on investment has to be obvious and the startup time short for the process to have any positive impact on the project.

Coding standards, for example, can help the team learn a new language quickly. They can also be used to filter out programming practices that put the source code at risk during the maintenance phase of the project. They also facilitate the establishment of what to expect when reading source code.

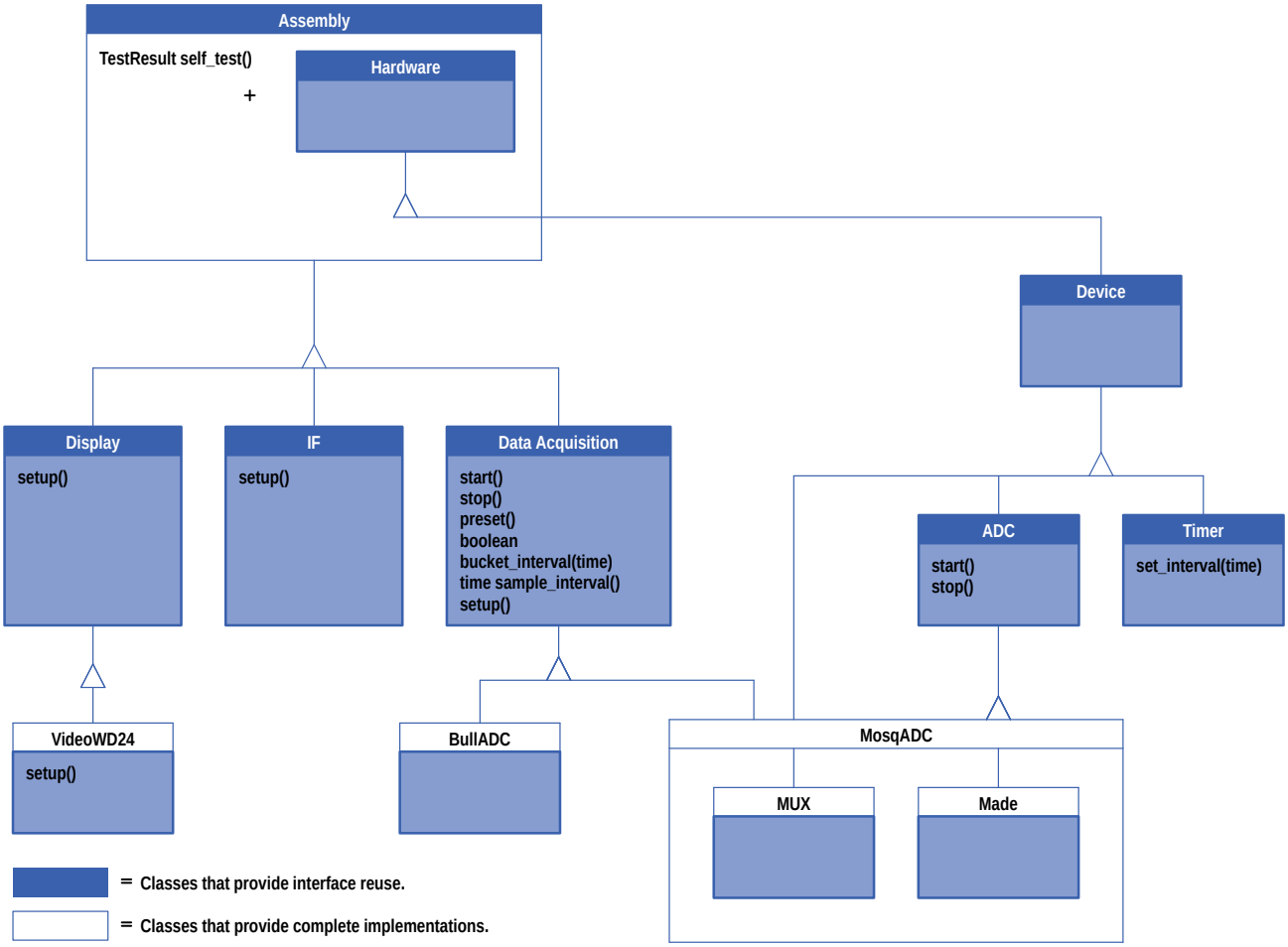

Fig. 13. *An example of an object model of the hardware layer that is coded to show the reuse status of the various objects.*

Evolutionary Delivery

We partnered with HP's Software Initiative program to develop what is now known as EVO Fusion.[9,10] EVO is a management process that constructs a product in small increments. After each increment is completed, the development process is examined for improvements that might contribute towards the successful completion of the next increment.

Each increment can have an analysis, design, code, and test phase. The product evolves over time into the desired product through repeated execution of small development cycles that add greater and greater functionality. This process helps to focus the team and increases the probability that schedules will be met.

Inspections

Much has been written about the value of inspections to software development. Though much of the literature focuses on product quality, the inspection process also identifies (that is, makes explicit) other issues. Once identified, these issues can be quantified into high, medium, and low risk factors. Their impact on the success of the project can be ascertained and the appropriate action can be taken to manage their resolution in a timely manner. Institution of an inspection process thus provides the project manager and the project team with an additional means by which to gather information pertinent to project completion.

In a project, the use of a development method like EVO Fusion, coupled with an inspection process, facilitates the discussion of issues that relate to requirements, software architecture, software integration, and code development. The benefits to the team are significant because these processes enable members to understand the product and its functionality long before the debug phase begins.

Legacy Systems

In many cases, it is not possible to generate a system completely from scratch without using legacy code. The firmware framework project was no exception.

We found that the most straightforward approach is to encapsulate the legacy code inside objects. This works for systems that provide services to client objects. It also works for legacy subsystems that act as clients, such as language parsers. These parser components are not good framework citizens because they already have their own definition of the server interface they expect, which may not coincide with the object-oriented design.

We feel that the proper approach is to perform the object-oriented analysis and design without regard for the legacy system first, and then encapsulate the legacy code inside the proper objects. There is a strong temptation to design the object-oriented system around the existing legacy code, but in our experience the legacy system may not have been designed with the appropriate object-oriented principles. Thus, allowing it to affect the analysis may lead to a faulty design.

Summary

Fusion is the result of the evolution of a long line of software development processes. Like its predecessors, Fusion has its benefits, problems, and areas for improvement.

Benefits. The benefits we derived using Fusion include:

- Lightweight and easy to use. We found Fusion to be easy to learn. There is lot of guidance in the process that leads the user from step to step. It is not mechanical, but the user will not be wondering how to get from one step to the next.
- Enforces a common vocabulary. Often in architecting systems, the different domain experts on the team will have their own definitions of what certain terms mean. Generating data dictionary entries at the analysis phase forces everyone to state their definitions and ensures that misunderstandings are cleared up before design and implementation.
- Good documentation tool. We found that the documents generated from the Fusion process served as excellent documentation tools. It is all too easy, without the rigor of a process, to jump right in and start coding and do the documentation later. What often happens is that schedule pressure does not allow the engineer to go back and document the design after the coding is done.
- Hides complexity. Fusion allows a project to denote areas of responsibility clearly. This feature enables the team to talk about the bigger picture without being bogged down in the details.
- Good separation between analysis and design. Fusion enforces a separation between analysis and design and helps in differentiating between architectural and implementation decisions.
- Visibility graphs very useful. The visibility graphs are very useful in thinking about the lifetime of the server objects. Simply examining the code all too often gives one a static picture and one does not think about the dynamic nature of the objects.

Problems. The problems we encountered with the Fusion method included:

- Thread support. Although the Fusion method models the system with a series of concurrent subsystems, this approach does not always work. The threads section of this article describes our problems with thread support.
- Complex details not handled well. This is a corollary to Fusion's ability to hide details. Do not expect Fusion to be able to handle every last detail in the system. In instrument control, there are a lot of complex data generation algorithms and interactions. Although in theory it is possible to decompose the system into smaller subsystems to capture the design, in practice there is a point of diminishing returns. It is not often feasible to capture all the details of the design.

Areas for Improvement. The following are some of the areas in which the Fusion method could be improved:

- Concurrency support. We would like to see a process integrated with the current Fusion method to handle asynchronous interactions, multitasking systems, and distributed systems.
- CASE support. We went through the Fusion process and generated our documentation on a variety of word processing and drawing tools. It would have been very helpful to work with a mature CASE tool that understands Fusion. Some of the functionalities needed in such a tool include: guidance for new Fusion users, automatic generation of design documents, and automatic checking for inconsistencies in different parts of the system. Throughout the course of our project we evaluated several Fusion CASE tools, but none were mature enough to meet our needs.

Acknowledgments

The authors wish to thank the other members of the firmware framework team: David Del Castillo, Manuel Marroquin, Steve Punte, Tony Reis, Tosya Shore, Bob Buck, Ron Yamada, Andrea Hall, Brian Amkraut, Vasantha Badari, and Caroline Lucas. They lived the experiences and contributed to the knowledge described in this paper. We'd like to also recognize Todd Cotton from the HP Software Initiative (SWI) team who, as a part-time team member, helped us develop our EVO process. Our gratitude also goes to the rest of the HP SWI team for the support they gave us during the project. Thanks to Derek Coleman for helping us use Fusion. Finally, we would like to express our appreciation to Ruth Malan; without her encouragement this paper would not have been possible.

References

1. I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.
2. D. Coleman et al., *Object-Oriented Development: the Fusion Method*, Prentice Hall, 1994.
3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley, 1995
4. R. Malan, R. Letsinger, and D. Coleman, *Object-Oriented Development at Work: Fusion in the Real World*, Prentice Hall, 1996.
5. R. Wirfs-Brock, "Object-Oriented Design: A Responsibility-Driven Approach," *OOPSLA '89 Conference Proceedings*, pp. 71-75.
6. R.J.A. Buhr and R.S. Casselman, "Timethread-Role Maps for Object-Oriented Design of Real-Time and Distributed Systems," *OOPSLA '94 Conference Proceedings*, pp. 301-316.
7. R.J.A. Buhr and R.S. Casselman, *Use of CASE Maps for Object-Oriented Systems*, Prentice Hall, 1996.
8. R.S. Casselman et al., *Notation for Threads*, SCE-92-07, Department of Systems and Computer Engineering, Carleton University, September 1992.
9. T. Cotton, "Evolutionary Fusion: A Customer-Oriented Incremental Life Cycle for Fusion," *Hewlett-Packard Journal*, Vol. 47, no. 4, August 1996, pp. 25-38.
10. T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley, 1988.

Microsoft and Windows are U.S. registered trademarks of Microsoft Corporation.

▶ Go to Next Article

▶ Go to Journal Home Page