Chapter 5
Module Architecture View

The main purpose of the module architecture view is to bring you closer to the system's implementation in software. The reason for keeping the module view distinct from the conceptual view is that they each make explicit different things.

In the conceptual view, the functional relationships must be explicit, but in the module view that is secondary to making explicit how the functionality is mapped to the implementation. In the module view, the relationships among the implementing elements must be made explicit, including how the system uses the underlying software platform (for example, operating system services, other system services).

For example, in the image-processing pipeline of IS2000, the images are passed from one pipeline stage to the next. This is explicit in the conceptual view. In the module view, each stage requests a pointer to its image data from the pipeline manager, then reads and writes the image using this pointer. The pipeline manager ensures that the pointers are given successively to each of the stages, but this control information is now embedded inside the implementation of the pipeline manager, instead of being visible at the architectural level.

It is certainly not impossible to keep all the important control information visible in the module view, but this is one of the design trade-offs you must make. Often (as in this case) performance requirements force you to choose a more efficient implementation—one that sacrifices logical clarity for performance.

As another example, suppose in the conceptual view two components communicate over a connector with call/return semantics. If the two components are assigned to the same CPU, they are implemented as a local procedure call, and if they are on different CPUs, they are implemented as a remote procedure call (RPC). In the conceptual view you

are interested in describing the behavior of these connectors: The functional behavior is the same, but the connectors could have different failure modes or timing differences.

However, because the implementation of the two variations is very different, in the module view they have different modules associated with them. In addition, for the RPC variant, the supporting services need to be included. These are not important in the conceptual view.

The module view is not simply a refinement of the conceptual view. Mapping conceptual elements to modules is a repartitioning. In IS2000 you'll see an example in which a component, three connectors, their roles, and two ports are grouped together, then split into two modules—one containing the processing and the other containing the data. Although the modules must fulfill the behavior specified for the conceptual elements, they must also take into account the implementation constraints.

The conceptual and module views are based on two fundamentally different models. In the conceptual view, components are where the main application functionality resides. They interact using connectors, which can have sophisticated control functionality, and use ports and roles to adapt or to mediate the interaction.

In the module view, all the application functionality, control functionality, adaptation, and mediation must be mapped to modules. Modules require and provide interfaces, but these have no associated implementation; they can't adapt or mediate the interactions between modules. Modules interact by invoking services declared in their required interfaces. They are passive with respect to their provided interfaces, waiting to be invoked by another module.

Unlike the conceptual and execution views, for the module view you do not define a configuration. You define the modules and their inherent relationships to each other, but not how they are combined into a particular product. That information is in the execution view, where you define how modules are mapped to runtime elements, including when and how they are created and destroyed.

Modules are organized into two orthogonal structures: decomposition and layers. The decomposition of a system captures the way the system is logically decomposed into subsystems and modules. A module is also assigned to a layer, which then constrains its dependencies on other modules.

This chapter, like the previous one, has three sections. Section 5.1 describes the design activities, Section 5.2 shows how they are applied to IS2000, and the final section summarizes the elements and relations of the module view, and how you should describe and use them.

5.1 Design Activities for the Module Architecture View

Figure 5.1 shows the design tasks for the module architecture view, and how they interact with other design tasks. Like the conceptual view, the module view has three basic phases:

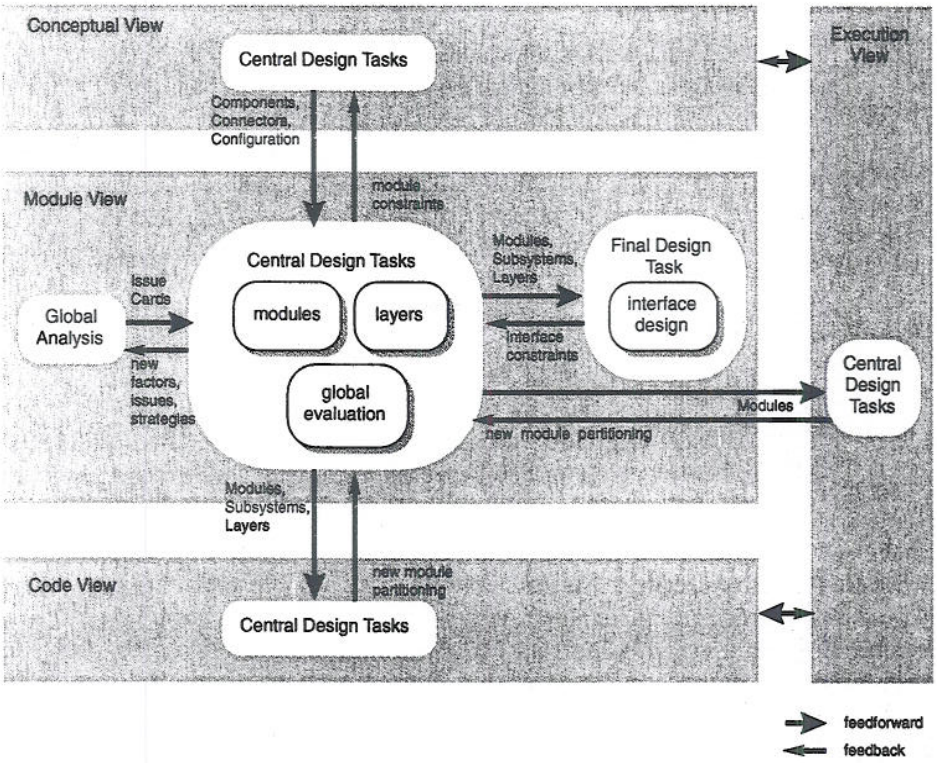


Figure 5.1. Design tasks of the module architecture view

global analysis, central design tasks, and interface design. The middle phase consists of three tightly coupled design tasks: modules, layers, and global evaluation.

This central phase uses the issue cards from global analysis, and the components, connectors, and configuration from the conceptual view as input. The results of this phase—the modules, subsystems, and layers—are used during interface design, and by the central design tasks of the execution and code architecture views.

During the central design phase you may identify new factors, issues, or strategies, causing you to revisit the global analysis task. You may also discover constraints about the modules or layers that cause you to make changes to the conceptual view.

After the central design phase, you may also get feedback from the interface design task, or from the execution or code architecture views, causing you to adjust the modules or layers.

5.1.1 Global Analysis

The first design task for the module view is global analysis. You have done much of the global analysis already, during the design of the conceptual view. When designing the module view, you may identify new factors or may create new issues and strategies, and the analysis of these follows the same approach described in Chapter 3, Global Analysis.

Before starting the central design tasks you should identify the strategies you expect to be relevant for the module view. Although you should check all the factors, certain categories are more likely to be relevant to this view. For the organizational factors, make sure you check the staff skills, process and development environment, and development budget categories. For the technological factors, the general-purpose hardware, software technology, and standards categories probably contain factors that affect this view.

Once you've identified the relevant factors, you then determine which strategies are influenced by them. Look for strategies related to modifiability, portability, and reuse. Use these strategies to guide the decisions for the module view. Also look for strategies related to properties like performance, dependability, and failure detection, reporting, recovery to make sure your module view design supports them.

5.1.2 Central Design Tasks

For the central design tasks you map conceptual elements to subsystems and modules, create layers, and assign modules to layers. You also determine the interfaces of the modules and layers. Guiding these decisions are the strategies from global analysis, experience with prior products, and general software engineering knowledge.

Modules

To design the modules you map the elements from the conceptual view to subsystems and modules. A subsystem usually corresponds to a higher level conceptual component (one that is decomposed into other components and connectors). It can contain other subsystems or modules.

A module can correspond to a single conceptual element (component, port, connector, or role) or to a set of them. Modules can also be decomposed into other modules, but in this case the parent module is really a container; only the leaf modules, in the end, correspond to implemented code. So a subsystem must eventually contain modules, if it is to have any implementation.

Figure 5.2 is a meta-model that describes these relations among subsystems and modules, and the relations between modules and interfaces. A module encapsulates data and operations to provide a service. The services provided by a module are defined by the interfaces it provides. A module may also need the services of another module to perform its function. These required services are defined by the interfaces it requires.

Modules can only interact with each other through these interfaces. Let's define the "use" relation (also called use-dependency) between modules as a relation derived from the "require" and "provide" relations. One module uses another when it requires the interface provided by the other.

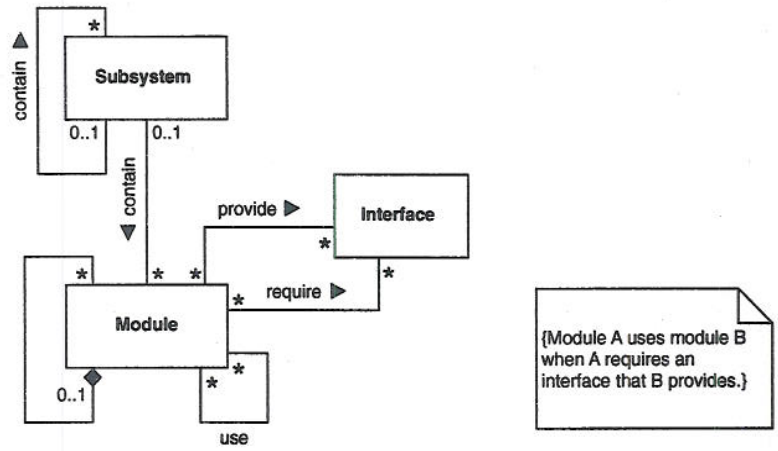


Figure 5.2. Meta-model for subsystems and modules

The conceptual elements get mapped to these modules. In the process of this mapping you assign the modules a responsibility, and determine their decomposition and use relationships. After the initial mapping you may decide to refine and split modules so they can be developed independently. Or, using strategies from the global analysis, you may decide to combine modules for efficiency.

You may also need to add supporting modules that don't have counterparts in the conceptual view. Look for factors that you can't pin down to a particular component, such as failure detection or recovery. Also look for services that are needed by existing modules and aren't provided by the software platform. These can lead to new modules.

Modules should be decomposed to the point when the responsibilities of each module are well understood, along with any implementation or integration risks. Modules grouped into a containing module are more tightly coupled than the modules contained in a subsystem, so they should be assigned as a group to a particular person or team.

Although the leaf modules are eventually implemented as source code, they are abstract modules. A module could, in the end, be implemented as an Ada package, a Modula module, a set of C++ classes, a set of C functions, or other programming language-specific elements.

Layers

Layers organize the modules into a partially ordered hierarchy. When a module is assigned to a layer, it can use any of the other modules in that layer. But whenever a module usage crosses a layer boundary, the interface required and provided by the modules must also be required and provided by the layers. Thus layers are used to constrain the module "use" relations.

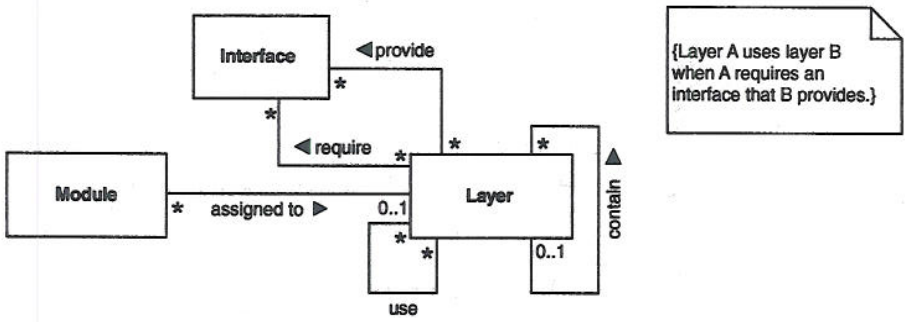


Figure 5.3. Meta-model for layers

The relations for layers are described in the meta-model in Figure 5.3. Layers can also contain sublayers to provide additional structuring within a layer.

Layers are a time-tested way of reducing complexity—for example, by encapsulating external components (such as COTS software or hardware) or by separating systems services from user interface software. They can be used to support reuse by assigning common services to an application services layer. Layers are also useful in providing independence between parts of the system so that, for example, a change in the operating system does not affect the entire system.

Although we present the design of modules and layers as two separate tasks, in practice you start identifying the layers as the modules are identified. You can do this in a bottom-up fashion, in which layers and the dependencies among them grow from the module responsibilities and their dependencies. The layers emerge as the modules are defined.

Another way is to begin with a set of layers based on your experience from similar applications in the domain. As they are identified, modules are assigned to the preexisting layers. In this top-down approach, the layers serve as a guide for defining modules.

The most common approach is to proceed with a combination of the two. Often, architects have in mind a coarse division of layers; for example, applications, user interface, and system services. As modules are defined they usually refine the layer model by adding additional layers for domain-specific functionality or by creating sublayers when a layer gets too complex.

Global Evaluation

To define the modules and layers, you get guidance from multiple sources: your conceptual view design, the strategies from global analysis, your experience with software architectures, and your general knowledge of architecture and software engineering. An important part of global evaluation is deciding which source of information to use at which time.

Global evaluation also means being on the lookout for feedback to the tasks and decisions you made earlier in the design. You should look for additional factors, issues, and strategies that feed back to the global analysis task. You need to evaluate whether any of your decisions about modules and layers warrant a change to the design of the conceptual view.

The third aspect of global evaluation is evaluating your module view decisions with respect to each other. You should expect to adjust the modules and subsystems based on your decisions about the layers, and vice versa. You will have to define new interfaces or revise them to satisfy the "require" and "provide" relations of both modules and layers.

5.1.3 Final Design Task: Interface Design

The final phase of module view design is to describe the interfaces for each of the modules and layers identified during the central design phase. This is done after the other tasks are complete. Here you do the detailed design of interfaces required or provided by modules or layers based on their use-dependencies. For this task you may decide to create new interfaces or to split or combine some. These decisions feed back to the central design tasks, and you may need to revise your modules or layers as a result.

5.2 Design of Module Architecture View for IS2000

Now that we have introduced the tasks for the module view design, let's look at how these are applied to the example system.

5.2.1 Global Analysis

Before starting the central design tasks for the module architecture view, let's revisit the global analysis. We need to look for applicable strategies, paying particular attention to those that have influencing factors related to staff skills, the process and development environment, general-purpose hardware, and software technology. We expect the following strategies to be applicable to the module view:

Issue: Aggressive Schedule
Strategy: *Reuse existing components.*

Issue: Skill Deficiencies
Strategy: *Encapsulate multiprocess support facilities.*

Issue: Changes in General-Purpose and Domain-Specific Hardware
Strategy: *Encapsulate domain-specific hardware.*
Strategy: *Encapsulate general-purpose hardware.*

Issue: Changes in Software Technology
Strategy: *Use standards.*
Strategy: *Develop product-specific interfaces to external components.*

Issue: Real-Time Acquisition Performance
Strategy: *Separate time-critical from nontime-critical components.*

Issue: Implementation of Diagnostics
Strategy: *Reduce the effort for error handling.*
Strategy: *Encapsulate diagnostic components.*

5.2.2 Central Design Tasks: Modularization and Layering

Next let's turn to the central design tasks of the module view: defining modules and organizing them into layers. As with the central design tasks of the conceptual view, these tasks are tightly coupled, so we need to switch back and forth between the tasks.

In this section let's start by using the conceptual components to come up with some initial layers, then map all the conceptual elements to subsystems and modules. Let's define the use-dependencies between the modules, then revisit the layers to refine them and add new supporting layers.

Initial Definition of Layers

One way to get started with the module view is to associate the main conceptual components with layers. When you start with the layer structure in this way, the layers you create come from your experience. We show such an initial layer definition in Figure 5.4. This is only a working diagram, because in the end we want the layer diagram to show modules assigned to layers, not to components. In all of our diagrams we use the UML package notation for a layer: It is drawn as a box with a tab attached to the upper left edge of the box.

Of the conceptual components in Figure 4.6, there are three components for the user interface: Acquire, Monitor, and Export. The graphical user interface (GUI) aspects of these components should be mapped to modules and put in a separate GUI layer. The Acquire component is mainly a user interface functionality, so it goes in the GUI layer.

The Monitor component contains GUI plus other functionality. The GUI functionality goes in the GUI layer, but the other functionality should be placed in a layer with similar components—in this case, a lower layer that we'll call Applications. The Export component contains GUI, image-processing, zoom, and image export functionality. The GUI part goes in the GUI layer, and the other modules also go in the Applications layer.

The GUI layer implements the graphical user interface for applications. Architecturally, a separate GUI layer is desirable for four reasons:

1. Design of GUI components is distinctly different from design of components with a programmatic application programming interface (API).
2. A separate GUI layer helps to promote a single, unified user interface design and reusable widgets for implementation.
3. Procedural and GUI components are enabled to be reused in other contexts.
4. Separation and decoupling of procedural and data components from GUI components reduces reworking resulting from changes to GUIs.

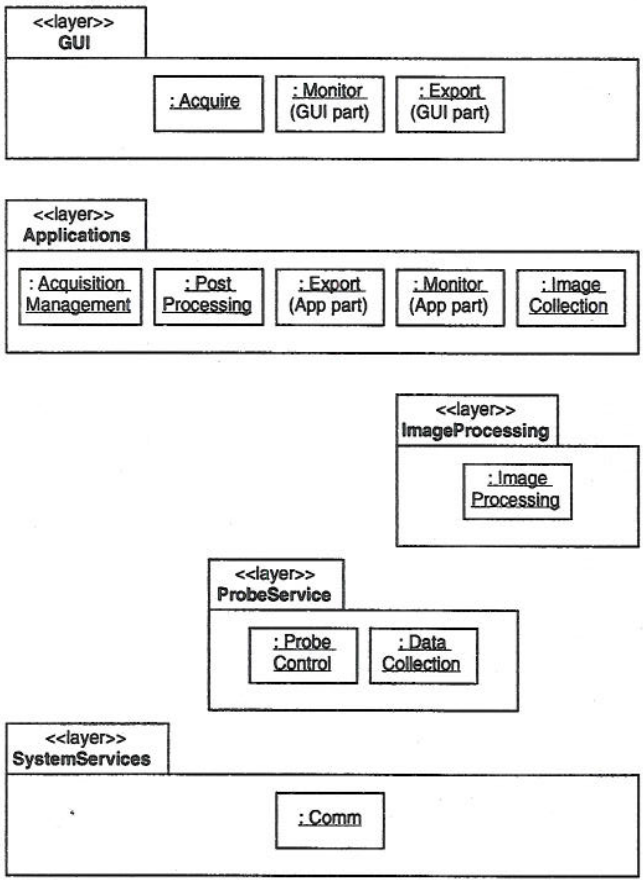


Figure 5.4. Initial creation of layers based on conceptual components. GUI = graphical user interface.

We make the distinction that the GUI is responsible for defining and managing the window display and handling user events, whereas the policy that defines what is done when an event occurs is handled by the applications. For example, the acquisition applications interact with the acquisition GUIs to set up the acquisition parameters, display the acquired data, display warning and error messages, and report the status of the acquisition. The GUI handles all the user requests by presenting the user with interface screens that can accept input from the keyboard, the mouse, and the menus on the screens. The GUI isolates the applications from the low-level interface toolset (for example, X/Motif).

The AcquisitionManagement, PostProcessing, and ImageCollection components get put in the Applications layer. You may recall that we used the strategy *Separate time-critical from nontime-critical components* to split the processing of images into Image-

Processing and PostProcessing. For assignment to layers, we should follow the same strategy, and create a new ImageProcessing layer for time-critical processing.

Next let's look at the ProbeControl and DataCollection components, which encapsulate the probe hardware. We originally used the strategy *Encapsulate domain-specific hardware* to split these two components from the rest of the acquisition and imaging components. As before, the layering structure should support this strategy, so we create a new lower layer—ProbeService—for these two components.

The last of the components is Comm, which provides the network communication and domain-specific communication protocols. This goes in the SystemServices layer. This layer lets us hide the details of the communication-specific code from the modules implementing the high-level connector protocols. When we apply the strategy *Encapsulate multiprocess support facilities*, the modules that provide these services are also assigned to this layer.

Defining Modules for Image Processing

Now let's look at defining the modules for the image-processing components from the conceptual view. The image-processing subsystem produces framed images from the raw camera data. As defined in the conceptual view, image processing consists of a packetizer for collecting the raw data and one or more image pipelines that process the packets into images. An image pipeline is composed of a sequence of stages: The "packetized" data is input to the first stage, and each stage's output is input to the next stage. The output of the final stage is framed images. There are several types of possible image pipelines, each of which performs functions for a specific acquisition procedure. The conceptual configuration for ImageProcessing is shown in Figure 5.5.

For defining modules, let's start by assigning each component to a single module. In general we want to separate as much of the communication infrastructure as feasible into connector- and port-specific modules. Then the modules implementing a conceptual component are insulated from changes to the software platform.

First we need to map the ImageProcessing and ImagePipeline components to subsystems, and the Packetizer component to a module. Next we need to consider whether the PacketPipe connector should be mapped to a separate module. Here, experience suggests that because of the volume of data, we should use a central data buffer for the packetized data rather than give each pipeline its own copy. Therefore the Packetizer and the PacketPipe are implemented as the single central buffer manager MPacketizer.

The decisions about mapping components and connectors to modules generally come first, before you decide how to map the ports and roles to modules. Although in the conceptual view a port is contained in a component, we often want to introduce separate modules for ports. Ports play an important role in insulating a component from connector details, so the advantage of mapping ports to separate modules is to encapsulate connector-specific knowledge and code.

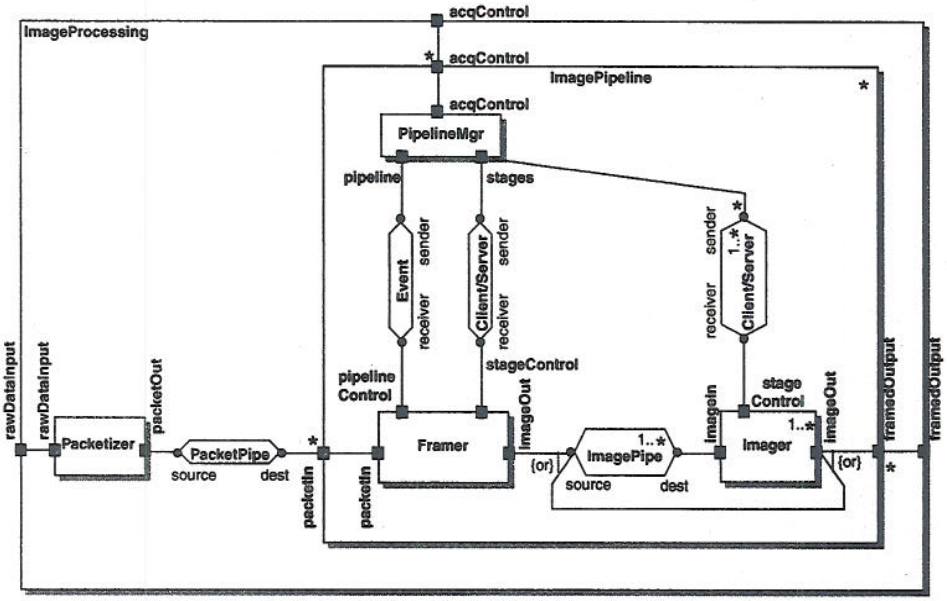


Figure 5.5. Conceptual configuration for ImageProcessing (from the conceptual view)

You get a similar benefit by mapping roles to separate modules. In the case of the PacketPipe, we've already made the decision to combine it with the Packetizer, so we have nothing to gain by putting their ports and roles in separate modules.

However, we do map the packetIn port on the ImagePipeline to a separate module—MPacketMgr. The pipeline uses module MPacketMgr to access the incoming data; it cannot access any other modules of the packetizer.

Similarly, the acqControl port on the ImageProcessing component is the interface used by the acquisition manager for controlling the ImagePipelines. We want to make sure the acquisition manager only has access to the MAcqControl module, not to the modules making up the pipeline directly, so this port also gets mapped to its own module. The decisions we've made so far are summarized in Table 5.1.

The PipelineMgr, its connections to the stages, and the connectors between the image pipeline stages themselves are mapped to the single central image pipeline manager MPipelineMgr in the module view. This manager controls and coordinates the stages of the pipeline. Because of the volume of data passed between the stages of the pipeline, again it will be more efficient to keep the data in a central buffer with each stage updating it in place, rather than passing the data between stages.

Conceptual Element		Module Element	
Name	Kind	Name	Kind
ImageProcessing	Component	SImaging	Subsystem
ImagePipeline	Component	SPipeline	Subsystem
Packetizer packetOut PacketPipe, source, dest	Component Port Connector and roles	MPacketizer	Module
packetIn	Port	MPacketMgr	Module
acqControl	Port	MAcqControl	Module

Table 5.1. Mapping Conceptual Elements to Module Elements: ImageProcessing

Next let's look at the relationship between the pipeline manager and its internal clients—the pipeline stages. We need to consider the ports on the pipeline manager, the connectors, and the ports on the clients.

Ideally we should encapsulate the high-level protocol specified by the client/server and event connectors as a single module. However, in practice, implementations for connectors are not always available and we are constrained by the available mechanisms in the software platform. When this is the case, the modules for the adjacent ports often have to implement what is missing.

Thus we combine the ports and connectors mediating the interactions between the stages (the clients) and the pipeline manager service with the MPipelineMgr module. The decision to use a single module comes down to a trade-off of providing an efficient implementation of our particular system versus building in flexibility for future uses. If we plan to reuse these components and ports in other systems that make up a product line for example, then we would be justified in devoting the necessary resources to implement a more general solution and add to our collection of implementations for domain-specific ports and connectors.

Now the pipelineControl, stageControl, imageIn, and imageOut ports on the stages are at the boundary of the new MPipelineMgr module. We group these ports into the MImageMgr module. The stages access this module only and do not have access to MPipelineMgr directly.

Conceptual Element		Module Element	
Name	Kind	Name	Kind
PipelineMgr	Component	MPipelineMgr	Module
pipeline, stages	Ports		
ImagePipe, source, dest	Connector and roles		
Client/Server, sender, receiver	Connector and roles		
Event, sender, receiver	Connector and roles		
pipelineControl, stageControl, imageIn, imageOut	Ports	MImageMgr	Module
Framer	Component	MFramer	Module
Imager	Component	MImager	Module

Table 5.2. Mapping Conceptual Elements to Module Elements: ImagePipeline

The stages in the pipeline such as Framer and Imager get mapped to their own modules. The mapping between conceptual elements and modules for ImagePipeline is summarized in Table 5.2.

Next let's begin to identify the decomposition dependencies among the modules and subsystems using the following rule:

> *Identifying decomposition dependencies.* If a conceptual component is decomposed into lower level components, there is a dependency from the corresponding parent module or subsystem to the child module or subsystem.

In the example, the dependencies among the conceptual elements give rise to the containment relationships shown in Figure 5.6. The notation we use for a module is a UML stereotyped class. Although people sometimes use the UML "component" notation for a module, it doesn't fit with our notion of an abstract module. A UML component is associated with a physical module of source code, so this notation is more appropriate for our code architecture view.

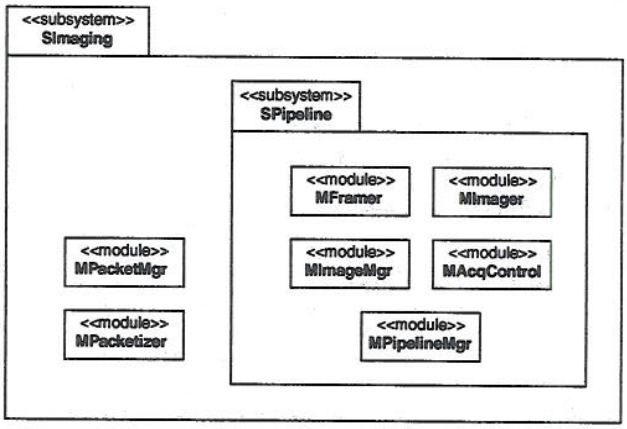


Figure 5.6. Initial containment relationships in imaging subsystem

The notation for a subsystem is a stereotyped UML package. Decomposition dependencies are shown by nesting the modules inside the containing module or subsystem. A subsystem can be decomposed into subsystems and modules (as in Figure 5.6), and a module can be decomposed into modules.

There are also use-dependencies among the modules based on the relationships among their corresponding conceptual elements.

> *Identifying use-dependencies.* If a conceptual component provides a service to another component, there is a dependency from the user to the provider of the service.

These dependencies are a little more difficult to determine because we must first identify the provider of the service and the client. It is helpful to look for patterns in control flow (for example, master/slave, service/client) or data flow direction. Note that there may be dependencies in both directions.

For each interaction between modules (derived from a relationship between their corresponding conceptual elements), we must define an interface through which the interaction occurs. One of the modules provides the interface, and the other requires it. The module that requires the interface "uses" the module that provides it.

For example, MAcqControl interacts with MPipelineMgr, so we need to define an interface—IPipelineControl—for this interaction. As we said earlier, the acqControl port is used by the acquisition manager to control the ImagePipelines. So in this case, the pipeline manager is the passive module, and it must provide the IPipelineControl interface.

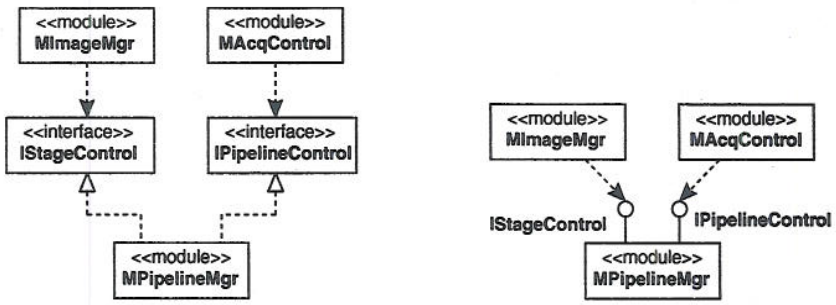


Figure 5.7. Alternative notations for requiring and providing an interface

The MImageMgr module also interacts with MPipelineMgr, but for a different purpose. The pipeline stages use MImageMgr to obtain their image data and to check for special processing requests from acquisition management. So MPipelineMgr provides a second interface—IStageControl—that is required by MImageMgr.

Figure 5.7 shows two different ways to describe these relationships in UML. On the left, the interfaces are shown as stereotyped classes, and the "provide" relation is shown as a dotted line with a triangle. The right side uses the "lollipop" notation for the interfaces, with the "provide" relation indicated by a solid line. In both notations the "require" relation is shown as a dotted line with an arrow pointing to the required interface.

Figure 5.8 shows the use-dependencies among the modules of the imaging subsystem. Let's add two new modules to this figure, enclosing some of the earlier modules. Because MAcqControl and MImageMgr act as proxies to request services of MPipelineMgr, these three modules are tightly coupled and should be implemented together. Thus let's group them in a containing module—MPipeline.

Let's do a similar thing with MPacketizer and MPacketMgr, and group them within MPacket. After adding these two higher level modules, the SPipeline subsystem isn't needed to organize the modules, so we can remove it.

In general, after you know which modules are in a subsystem, you should consider whether any of them should be grouped into a new containing module. Group modules that have many logical dependencies on each other, making them hard to develop independently. Then get rid of any subsystems or containing modules that are no longer needed.

We have often found it useful to produce separate diagrams for the decomposition and use-dependencies, although we combined them in one diagram here. Diagrams can then be organized according to different subsystems, providing more detailed views for each.

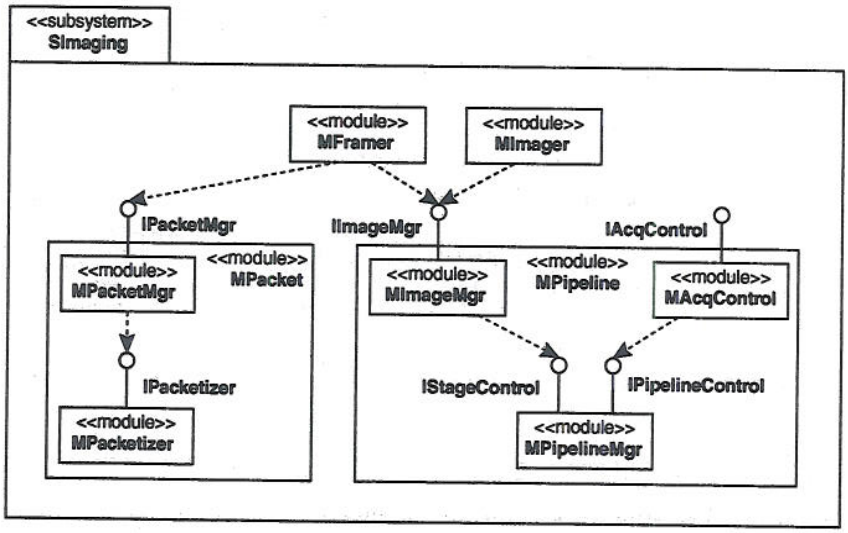


Figure 5.8. Imaging subsystem use-dependencies

Reviewing the ImageProcessing Layer

As planned during our initial layering task, we need to assign the image-processing modules to the ImageProcessing layer. Next let's take the modules corresponding to acquisition management and put them in the Applications layer. The acquisition manager client (MClient) accesses the MPipeline module through the IAcqControl interface. Therefore we can use it as an interface to the ImageProcessing layer as well.

For the probe services, let's apply the strategy *Reuse existing components*, and incorporate the existing modules MProbeControl and MDataCollect into our design. In both cases we must provide a separate module for adapting the old module. The new modules—MDataAcq and MDataMgr—provide the probe service's interface to the rest of the system. These four modules abstract and encapsulate the data aspects of the probe hardware, so let's assign them to the ProbeService layer.

The ProbeService layer provides a complete abstraction of the probe such that the layers above it need not use the hardware services directly. This makes it possible to switch the hardware without affecting higher layers.

Our assignment of modules to layers is summarized in Figure 5.9. We show the use-dependencies between modules, which are derived from the interfaces they require and provide.

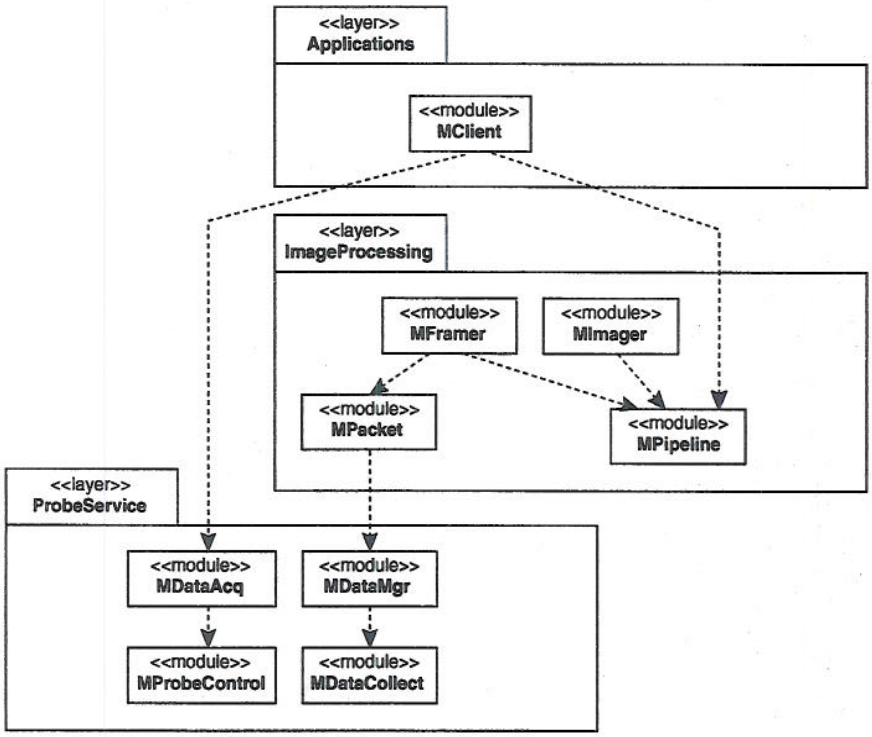


Figure 5.9. Assigning modules to layers

As the details of the module view are being worked out, it is useful to keep a current version of the layer diagram that fits on one page. Thus you can see at a glance the major partitioning of the system and get additional details by following the decomposition diagrams or looking up module assignments to layers in tables.

Now let's derive the dependencies between these three layers based on the dependencies among the modules belonging to them. For example, the data collector is used by the packetizer; thus the ImageProcessing layer depends on the ProbeService layer. The layers shown in Figure 5.10 follow the convention that arrows generally flow downward, meaning that layers positioned above are dependent on layers below them.

Within a layer, the layering does not impose any restrictions on module dependencies: Theoretically, any module can use any other within a layer. But the interfaces of a layer restrict what modules in other layers can use. Layers, like modules, can both provide and require interfaces. So two modules in separate layers can have a use-dependency only if one provides an interface that is also provided by its layer, and the other layer and module both require this interface.

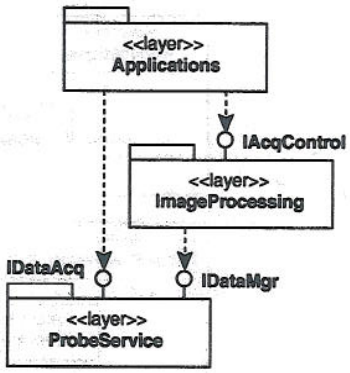


Figure 5.10. Use-dependencies between layers

For example, if we add another module to the Applications layer, it can use the services of modules in this layer as well as those provided by the ProbeService layer. (Although, if the Applications layer didn't previously require those services, we have to add the newly required interface.)

The layering scheme also facilitates buildability. Layers reduce and isolate external (for example, hardware and operating system) and internal dependencies, and facilitate bottom-up building and testing of various subsystems. Lower layer functionality can be implemented and tested before higher layers.

Adding Supporting Layers

In addition to mapping conceptual components, connectors, and ports to modules and organizing them into layers, an architect often needs to provide supporting layers. What is needed depends on the services the software platform provides.

For IS2000, because of the strategies *Encapsulate general-purpose hardware* and *Develop product-specific interfaces to external components*, we need to encapsulate the operating system, and use an operating system that is POSIX compliant. Thus let's introduce an OperatingSystem layer that supports a standard interface to operating systems.

Next let's look at the domain-specific support services for storage and communication. The ImageCollection component stores images for as long as 24 hours, but we haven't yet decided *how* these images will be stored. Let's use a commercial database to store these images, largely because a database has worked well for this purpose in past products. In addition, we can use the database to support recovery: It will implement the PersistentDataPipe connector between the ImageProcessing and PostProcessing components.

This decision means we need to revisit the technological factors, adding new factors, analyzing them, and checking whether we have any new strategies. The particular data-

base we use could change over the product's lifetime; for example, to maintain compatibility with other products. Other reasons for a database change are if a vendor goes out of business, if the product becomes obsolete, if a better platform becomes available, or if customers request features not available with the current database.

Recalling our strategy *Use standards*, and with a change in database likely, we decide to use an existing interface standard for the database, Open Database Connectivity (ODBC). This makes a change in database systems transparent, provided the new or upgraded system conforms to the interface standard. Changes to the standard itself would have a significant impact on many of our modules, but the standard is fairly stable.

The two new factors are listed in Table 5.3. After reviewing our existing issue cards, we update the issue Changes in Software Technology to add these influencing factors and to note that both existing strategies also apply to the database.

Technological Factor	Flexibility and Changeability	Impact
T3: Software technology		
T3.3: Database management system (DBMS)		
Use a commercial DBMS.	The DBMS may be upgraded every five years as technology improves.	The impact is transparent, provided it conforms to our DBMS interface standard. Change from a relational to an object-oriented DBMS may have a large impact on all components.
T5: Standards		
T5.5: Standard for DBMS interface		
Open Database Connectivity has been selected as the database access standard.	The standard is stable.	There is a large impact on components interfacing with the DBMS.

Table 5.3. Factors Added During Module View Design

Now, following the strategy *Develop product-specific interfaces to external components*, let's encapsulate the database, introducing a DatabaseService layer. The modules for image collection, which we had earlier placed in the Applications layer, are now moved to the DatabaseService layer.

A database service is a vendor-independent interface to a database management system (DBMS), and it provides support for data administration and management. We partition the DatabaseService layer into three sublayers. The DBMS is the lowest layer, and it supplies the mechanism for storing data. It is supplied by a vendor. The database interface at the next level is the vendor-independent interface to the DBMS. The top level is database administration, which is responsible for administration activities such as installation, configuration, maintenance, and database utilities.

To implement the requirements for error handling and logging, let's introduce modules for a logger and its interface. The interface is used by any other module that needs to communicate with the logger (strategy, *Encapsulate diagnostic components*). The logger is responsible for receiving and storing event messages in corresponding log files. Standard services such as message catalogs and the file system are used to review log files (strategy, *Reduce effort for error handling*). Let's collect these modules and put them into an ErrorHandling layer.

We may combine this ErrorHandling layer with the SystemServices layer later on. These kinds of services are often applicable to more than one product. If so, it makes sense to maintain them separately as part of a product-line architecture for a family of related systems. The final version of our layers is shown in Figure 5.11.

5.2.3 Final Design Task: Interface Design

For the final task in designing the module view we must describe the details of the interfaces, respecting the dependencies defined by the decomposition and layer structures. The protocol definitions from the conceptual view help us with this task. There isn't necessarily a one-to-one correspondence between a protocol definition and an interface definition because the ports or connectors that adhere to a protocol may be mapped to modules that are split or combined. However, using the protocol definitions is a good way to get started with supplying information about the details of the interface. Details of the interface include the names and characteristics of the operations of the modules. For operations, we must define the signature, including arguments and type information.

We also need to document the module with enough semantic information to describe what someone would need to know to use it (for example, preconditions, postconditions, exceptions). This is usually written in text. One way to integrate this documentation with the code is to establish coding conventions in which the detailed design is embedded in the comments in the header files. Instead of text, this semantic information could be written in a more formal assertion language.

We don't expect the architect to complete the entire interface design. Once the services provided and used are documented, these can be handed off to individual developers or teams to document the details. Details of the IAcqControl interface definition are shown in Figure 5.12.

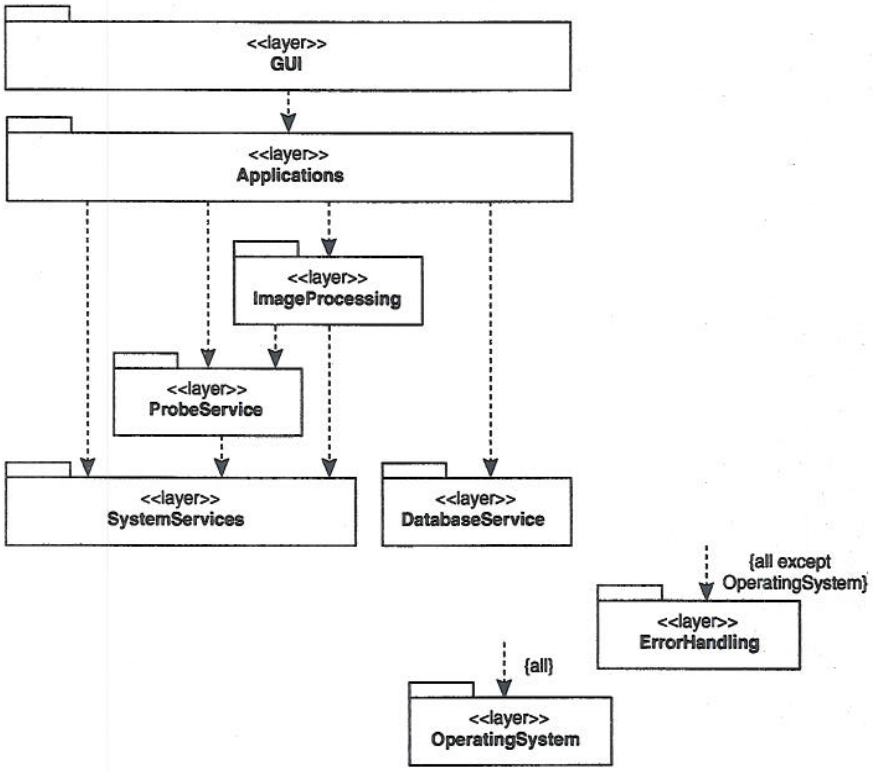


Figure 5.11. Final version of layers and their use-dependencies

```
Initialize()
   Client initializes interface with ImageProcessing.

Create(IP_id, Pipeline_config)
   Create an image pipeline with the given configuration.

Delete(IP_id)
   Stop execution of the pipeline and tear down the stages.

Terminate()
   Client terminates interface with ImageProcessing.

AdjustImage(IP_id, Stage_id, Message)
   Client adjusts the processing of the image
   (e.g., growth rate, persistence).
```

Figure 5.12. Interface definition for IAcqControl

Module Name	Description	Subsystem Name	Layer	Interfaces Provided
MPipelineMgr	Image pipeline manager	SImaging	ImageProcessing	IPipelineControl IStageControl
...				

Table 5.4. Sample Module Summary Table

Besides diagrams, tables are another useful description. Table 5.4 shows an entry from a summary of the modules of a system, with the layers and decomposition relationships. As the size of the system increases, tables for modules and interfaces become more important for managing and understanding the architecture.

5.2.4 Design Summary for IS2000 Module View

To design the module architecture view we identified modules that implement the conceptual elements, and we captured decomposition and use-dependencies among the modules. We also organized modules into layers. Table 5.5 summarizes the design decisions we discussed in this chapter.

Design Decision	Rationale
Initial definition of layers	
Create GUI layer.	Decouple GUI implementation from the rest of the application.
Create ImageProcessing layer.	Strategy: *Separate time-critical from nontime-critical components.*
Create ProbeService layer.	Strategy: *Encapsulate domain-specific hardware.*
Create SystemServices layer.	Strategy: *Encapsulate multiprocess support facilities.*

Table 5.5. Sequence of Design Decisions for IS2000 Module View

Design Decision	Rationale
Defining modules for ImageProcessing	
Implement Packetizer and PacketPipe as a centrally managed buffer (MPacketizer).	Recognize the high volume of data and real-time performance requirements.
Create separate modules for packetIn and acqControl ports.	Insulate the components from connector details.
Implement PipelineMgr, ImagePipe, Client/Server, and Event as a centrally managed buffer (MPipelineMgr).	Recognize the high volume of data and real-time performance requirements.
Create one module for pipelineControl, stageControl, imageIn, and imageOut.	Insulate Framer and Imager from details of MPipelineMgr.
Group MAcqControl, MImageMgr, and MPipelineMgr into a containing module.	Tightly coupled modules should be implemented together.
Group MPacketizer and MPacketMgr into a containing module.	Tightly coupled modules should be implemented together.
Remove subsystem SPipeline.	SPipeline not needed after adding containing modules.
Reviewing the ImageProcessing layer	
IAcqControl is provided by the ImageProcessing layer.	Provide module interfaces that are used by other layers.
Use existing modules MProbeControl and MDataCollect.	Strategy: *Reuse existing components.*
Adding supporting layers	
Create OperatingSystem layer.	Strategy: *Encapsulate general-purpose hardware.* Strategy: *Develop product-specific interfaces to external components.*

Table 5.5. Sequence of Design Decisions for IS2000 Module View *(continued)*

Design Decision	Rationale
Use a database for storing images (ImageCollection component) and for recovery (PersistentDataPipe connector).	Successful approach in past products.
Add new factors to the technological factor table: T3.3: Database management system, T5.5: Standard for DBMS interface.	The product uses a commercial database. Strategy: *Use standards.*
Create DatabaseService layer.	Strategy: *Develop product-specific interfaces to external components.*
Create ErrorHandling layer.	Strategy: *Encapsulate diagnostic components.*
Use standard services: message catalogs and file system.	Strategy: *Reduce effort for error handling.*
GUI = graphical user interface; DBMS = database management system.	

Table 5.5. Sequence of Design Decisions for IS2000 Module View *(continued)*

For IS2000, we started with an approximation of the layers, relying on experience and on strategies identified during global analysis. We then began to map conceptual elements to modules. We started by mapping a component to a module or subsystem. Next we looked at each connector, deciding whether to put it into an existing module or to create a new module for it. We did the same for ports and roles, sometimes creating a new module and sometimes putting them in an existing module.

Next we determined the decomposition and use-dependencies among modules and subsystems, and added interfaces for the modules. During this process we sometimes created new containing modules, and deleted containing subsystems.

We refined the layers, and defined interfaces for the layers. We also added supporting layers, relying on our global analysis strategies and the use-dependencies of the modules to determine the layers' relations. At the end, after the subsystems, modules, interfaces, and layers were stable, we performed detailed interface design.

5.3 Summary of Module Architecture View

Table 5.6 summarizes the elements, relations, and artifacts of the module architecture view. As in the conceptual view, the elements and relations are the building blocks for the architecture view, and the artifacts are the descriptions or documentation of the architecture. Note that the decomposition dependency is a form of the composition relation from the point of view of how a module is decomposed.

For the module view, one of the most important artifacts is the conceptual-module correspondence. Although this could be represented in a UML diagram, it's generally much more compact to put the mapping in a table. The Safety Vision case study (Chapter 8) contains an example of putting conceptual-module correspondence information in a UML diagram (see Figure 8.6).

The three kinds of UML Class Diagrams describing decomposition, module dependencies, and layer dependencies are essential artifacts for all systems. As we said earlier, particularly for large systems it is better not to put too many kinds of relations in a diagram. For example, the subsystem and module decomposition diagrams shouldn't have use-dependencies, and the use-dependency diagram shouldn't have decomposition dependencies or containment relations.

As you've seen in this chapter, we sometimes use diagrams that do combine these relations in a single diagram. Our module use-dependency diagram also showed containment and decomposition. A project should have conventions for the kinds of diagrams produced, so that the architecture description is consistent. You also have to decide which diagrams show interfaces explicitly, and which variant of the interface notation they use.

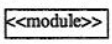
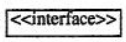
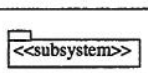
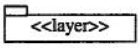
Element	UML Element	New Stereotype	Notation
Module	Class	<<module>>	
Interface	Interface	—	O or 
Subsystem	Subsystem	—	
Layer	Package	<<layer>>	

Table 5.6. Summary of Module Architecture View

Relation	UML Element	Notation	Description
contain	Association	Nesting	A subsystem can contain a subsystem or a module. A layer can contain a layer.
composition	Composition	Nesting (or ◆—)	A module can be decomposed into one or more modules.
use (also called use-dependency)	Usage	- - - - - ->	Module (layer) A uses module (layer) B when A requires an interface that B provides.
require	Usage	- - - - - ->	A module or layer can require an interface.
provide	Realization	—— (with ○)	A module or layer can provide an interface.
		- - ▷ (with ▭)	
implement	—	Table row	A module can implement a conceptual element.
	Trace	- - <<trace>> ->	
assigned to	Association	Nesting	A module can be assigned to a layer.
Artifact			Representation
Conceptual-module correspondence			Table
Subsystem and module decomposition			UML Class Diagram
Module use-dependencies			UML Class Diagram
Layer use-dependencies, modules assigned to layers			UML Class Diagram
Summary of module relations			Table

Table 5.6. Summary of Module Architecture View *(continued)*

Figure 5.13 presents the meta-model for the module architecture view. It contains the elements and relations from Table 5.6, and shows how they can be combined.

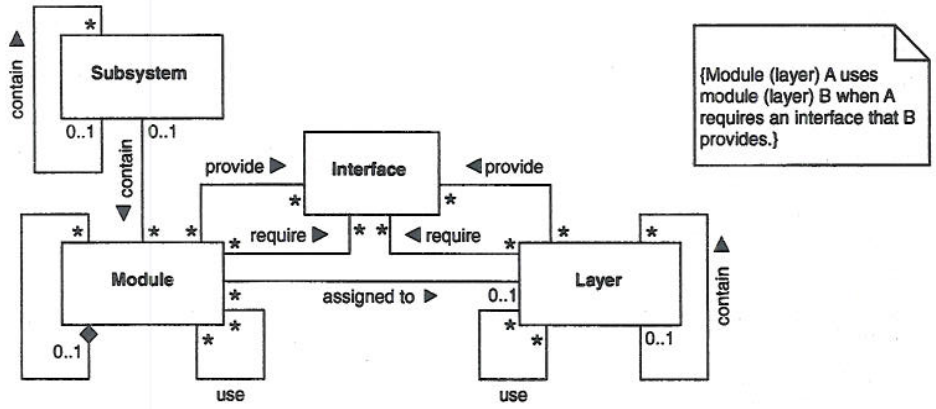


Figure 5.13. Meta-model of the module architecture view

We use the same notation for the "contain" and "composition" relations, but they are not quite the same. "Contain" relates subsystems and layers, which are based on UML packages, and "composition" relates modules, which are based on UML classes. Our convention is that subsystems and layers are simply containers; they have no implementation themselves. When a module is decomposed into other modules, the parent module is also just a container. Only the leaf modules are implemented eventually.

Interfaces are separate elements in the module view, and they can be provided or required by modules and layers. The "use" relation (also called *use-dependency*) is derived from the "require" and "provide" relations. One module uses another when it requires the interface provided by the other. This also holds for layers.

5.3.1 Traceability

Describing the relationships of the module view to requirements, external factors, and other views provides traceability. It is critical to relate the module view to the conceptual view, not just as an aid in designing the module view, but to have a complete architecture description. The following three items should be traceable in the module view:

1. *Critical requirements.* As we saw in the previous chapter, the factor tables and issue cards capture the product features and requirements that affect the architecture. Some of these can be traced to conceptual elements and others can be traced directly to modules or layers. This information can be used to determine the impact of changes in requirements to the module view. We can also use it to trace design decisions back to requirements to evaluate how well they are met.
2. *Organizational and technological factors.* Similarly, the organizational and technological factors are described in factor tables, and their impact on the architecture is

captured in issue cards. Tables like Table 5.5 show the strategies and factors used to make design decisions for the module view.

3. *Elements in the conceptual view.* The mapping of elements in the conceptual view to modules and subsystems should be recorded, for example, in a table such as Table 5.1.

Looking ahead to the next chapters, modules are assigned to runtime elements in the execution architecture view. Then, in the code architecture view, modules are mapped to source components.

5.3.2 Uses for the Module Architecture View

Once the module view is explicit, it becomes a starting point for reasoning about important system properties. The module view descriptions can be used for

- Management of module interfaces
- Change impact analysis
- Consistency checking of interface constraints
- Configuration management
- Effort estimation

The module view can be used for testing. Separation of function from communication simplifies function testing and enables unit testing of protocol implementations. The layer model can be designed to support independent building and testing of each of the layers.

Additional Reading

The work of DeRemer and Kron (1976) describes programming-in-the-large and the need for module interconnection languages. Prieto-Diaz and Neighbors (1986) provide a survey of the several variants of module interconnection languages that have been defined to support programming-in-the-large. Module interconnection languages of note include Intercol (Tichy, 1979), PIC (Wolf, 1985), and NuMIL (Narayanaswamy and Scacchi, 1987). This is similar to our module architecture view.

Parnas (1972) discusses the criteria to be used in decomposition. The criteria include changeability, among others, and are expanded in his article on software aging (Parnas, 1994).

A widely practiced convention for describing layers uses stacked rectangular boxes. Adjacencies between these boxes represent allowable interfaces between modules in different layers. Modules can only depend on modules in lower layers. Exceptions for crossing layers (for example, all applications access system services) are noted in text accompanying the diagram. Our description using arrows to represent the dependencies between layers is influenced by Selic, Gullekson, and Ward (1994). Buschmann et al. (1996) describe an architectural pattern for layers.

Kazman et al. (1996) describe the Software Architecture Analysis Method (SAAM), a scenario-based technique for evaluating the modifiability of an architecture based on structures in the module and code architecture views.