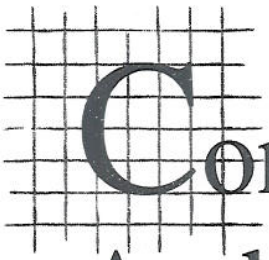

Chapter 4

Conceptual Architecture View

The conceptual architecture view is closest to the application domain because it is the least constrained by the software and the hardware platforms. In the conceptual view, you model your product as a collection of decomposable, interconnected conceptual components and connectors.

In this component-connector model, the components are independently executing peers. The notion of building a system by interconnecting components is appealing because of the potential for reuse and for incorporating off-the-shelf components, but that isn't possible without unbundling much of the communication and control aspects and putting them in connectors. So a critical goal of this model is to keep the control aspects of the components simple, and to isolate control in the connectors.

Because today's software technology doesn't directly implement this computational model, you need the other architecture views to show how the component-connector model is mapped to today's programming languages, operating systems, communication mechanisms, and so forth.

When designing the conceptual view, in addition to mapping your product's functionality to components and connectors, you must also treat global properties such as performance and dependability. Not all properties should be considered in the conceptual view. For example, portability is primarily a concern of the module architecture view, because there you must consider the factors relating to the software and the hardware platform.

For the global properties that you do address in the conceptual view, you will have begun to design an architecture that fulfills them, but you still have to consider them in other views. For example, the design of your conceptual view should address the perfor-

mance requirements. But in the execution view, when you map modules to hardware, you must revisit the performance requirements to make sure they are fulfilled.

If you are using a domain-specific or reference architecture, this could be the starting point for your conceptual view. Whether it can be the starting point depends on whether the architecture uses a computational model consistent with the conceptual view. Sometimes a domain-specific or reference architecture is defined in terms of the module or execution view, or it is a mix of several of the views.

When the conceptual architecture view for your system is complete, you will be able to reason about the ability of the system to fulfill functional requirements and global properties. If you are using use-cases and/or scenarios to capture the system's desired behavior, your conceptual view should be able to handle satisfactorily all the use-cases and scenarios that describe interactions with outside actors (the user, other systems, and so on).

This chapter has three main sections. The first section describes the design activities for the conceptual view. Section 4.2 presents a detailed example of how to design this view using the IS2000 system. The third section, Section 4.3, summarizes the results of designing this view, describing its elements, relations, artifacts, and uses.

4.1 Design Activities for the Conceptual Architecture View

There are three phases to the conceptual view design: global analysis, central design tasks, and the final design task. Figure 4.1 shows these design tasks and their relations to other tasks.

The central design tasks consist of four tightly coupled tasks. During this phase you identify the components and connectors for building your system and use them to construct it. Throughout this process, you use the issue cards developed during global analysis, and evaluate your design decisions with respect to the criteria established in the global analysis task.

The results of the central design tasks—the components, connectors, and configuration—are fed to the final design task, which is resource budgeting. Here you assign resources to the components and connectors in your configuration. These budgets are refined later during the execution view, but the initial budgets help you identify potential resource problems. These additional constraints are fed back to the central design tasks, where you can adjust the design to accommodate them or, if that's not possible, return to the global analysis to develop new strategies. In the most extreme case, you would revisit the system requirements and negotiate changes with the system stakeholders.

The conceptual architecture view provides input to the design tasks of both the module and the execution views, but can be influenced by them as well. It is the first part of the architecture that you will design.

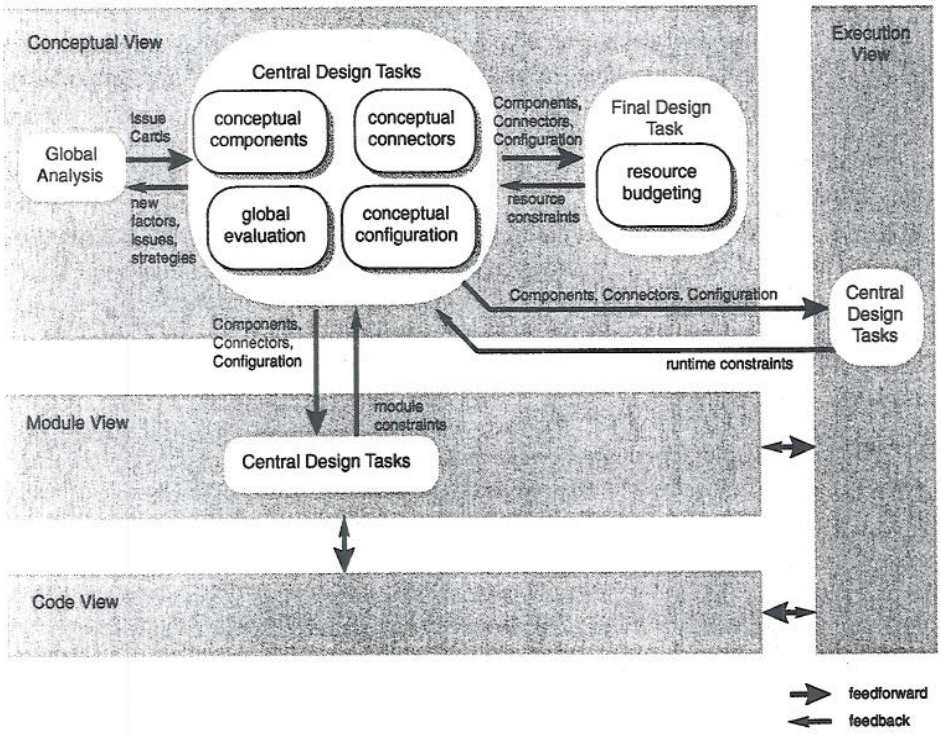

Figure 4.1. Design tasks of the conceptual architecture view

4.1.1 Global Analysis

The first step in conceptual view design is the global analysis task. This task is described fully in the previous chapter, so here we only mention aspects most important for the conceptual view.

Before beginning the global analysis for the conceptual view, you should have reviewed the product requirements, use-cases, and the system requirements and interactions. Make sure you understand the interface to the environment, the users, and other systems that interact with this system. Identify the modes of operation for the system. Look at the functional requirements, and pay particular attention to system qualities and global properties.

Next you analyze the product, technological, and organizational factors, producing factor tables for the three categories. Then you identify issues and develop strategies to solve them. Because the factors drive the issues and strategies, you should now focus on the factors most relevant to the conceptual view.

For this view, you should look at all the product factors. For the technological factors, those in the categories of domain-specific hardware and architecture technology are important. In the standards category, you should consider primarily the domain-specific standards.

Of the organizational factors, those from the categories of management, development schedule, and perhaps development budget are most likely to have an impact on the conceptual view.

It is not realistic to think that you'll be able to identify all of the factors at the beginning of the architecture design, but you should have captured most of them, and the most important ones. New factors may come up during the central design tasks, and during the design of the other views.

4.1.2 Central Design Tasks

To complete the central design tasks, you use the strategies developed during global analysis to guide design decisions and to improve the ability of the system to respond to change. You define the components and the connectors of the system, and define how the system is built using them. From the requirements and the application domain come the majority of components. Others may arise to support global properties. Using the results of the global analysis task as inputs, you may need to adjust component boundaries, or add new components or connectors. You do the following four activities during central design:

1. Define component and connector types.
2. Define how component and connector types interconnect.
3. Map the system functionality to components and connectors, concentrating functional behavior in the components and control aspects in the connectors.
4. Define the instances of the component and connector types that exist in the product, and how they are interconnected.

The order in which you do these things is not fixed. For example, you may start with the third activity in the list, in conjunction with the fourth, then do the first and second activity. You will also likely repeat many of these activities as you refine the design.

If you're starting with a domain-specific or reference architecture, you can expect it to provide the component and connector types, and some constraints on how they interact. It should also give you information regarding how to map the system functionality to components and connectors, at least for portions of your system.

A product-line architecture provides a similar starting point, and it may define some of the instances of the components and connectors. In this case, you would also expect it to provide the implementation of these components and connectors because they are used for all products in the family.

When performing the central design tasks you have to decide whether it's appropriate to use an available architectural style, and for which parts of the system. An architectural style defines component and connector types, and constrains how they interconnect. You need to execute the third and fourth activities before you know whether the style works for your system.

The strategies from global analysis constrain or guide many of your decisions. You must continually evaluate the results, particularly for the third and fourth activities, to make sure you're following the strategies. You should also review the factor tables to make sure you account for the relevant factors, and to see if any are missing.

Conceptual Components

For the conceptual components task, you define the components that exist in your system by defining both component types and instances. It is not absolutely necessary to define component types first, then instances of those types, but we recommend it. If you don't define types, you'll have to describe all the properties of a component for every instance.

A component contains ports, which are the interaction points for the component. A port is not the same as the common notion of an interface for several reasons. First, an interface defines services or operations that the component provides, but not what it uses. With this kind of asymmetry between "provides" and "uses," components would not be true peers. Ports in our model do not have this bias: They define both incoming and outgoing messages (operations), so the components can truly be peers. Second, interfaces are commonly defined as having no associated implementation; they are just a list of the operations provided. In our case, ports can have an implementation, so they can incorporate functionality to adapt, combine, or otherwise process incoming and outgoing operations. Lastly, each port has an associated protocol that mandates how the incoming and outgoing operations can be ordered.

A component can be decomposed into components and connectors. Figure 4.2 contains a UML diagram that summarizes the aspects of components we've just described. The boxes are the UML class notation, so we have elements for a component, a port, a protocol, and a connector.

The lines in the diagram are relations. The lines with the solid diamond are a standard UML relation that means composition: The class at the diamond end of a relation is decomposed into or contains the class at the other end. The multiplicity of the relation is shown by a number (or an asterisk, for zero or more) near the end of the relation. So a component contains zero or more ports, and a port is contained by at most one component. A port obeys exactly one protocol, but a protocol can be obeyed by zero or more ports.

Figure 4.2 is part of the meta-model of the conceptual view that we show at the end of the chapter. We call it a *meta-model* because it describes how the elements of this view are related to each other. This is in contrast to a normal UML model, which we reserve for describing the component and connector types, instances, and relations for a particular system. A meta-model diagram doesn't necessarily look any different from a model diagram, but the term *meta-model* tells you that you've stepped up a level of abstraction.

We use meta-models to describe the elements of each architecture view and how they are related to each other.[1] A meta-model is a compact, precise description, but it takes

1. Our meta-models are not intended to be part of the UML meta-model, which serves to define UML. We use the term *meta-model* in a more general way—to mean, simply, a model of a model.

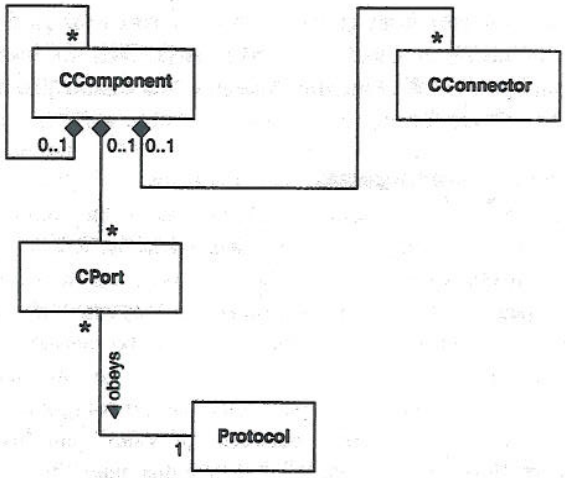

Figure 4.2. Definition of a conceptual component

some practice to understand it easily. Feel free to just skim over the meta-models and focus instead on the examples for IS2000 and the case studies.

In addition to describing the structure of a component, you should also describe its behavior. You may want to describe aspects of its behavior, such as control aspects, formally with a UML Statechart Diagram. This description would likely be associated with a component type rather than the instances, because it shouldn't vary across instances. To describe functional behavior, either of a component type or an instance, you can use a natural language description or a UML Statechart Diagram.

Conceptual Connectors

For the conceptual connectors task, you, in a similar way, define connector types and instances. Again, it is not essential that you define types, but we recommend it.

Whereas components had ports, connectors have roles as the point of interaction with other architecture elements. The roles, like ports, obey an associated protocol. Connectors can also be decomposed into components and connectors. These aspects of a connector are summarized in Figure 4.3. The formal names of these elements start with C for conceptual (for example, CComponent) to avoid confusion with existing UML terms.

You should also describe the connector behavior. Because the functional behavior of the system is concentrated in the components, and control is concentrated in the connectors, to describe connector behavior you must focus on the control aspects.

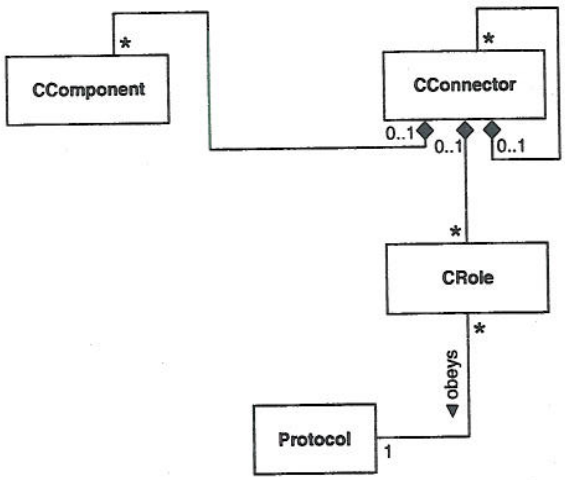

Figure 4.3. Definition of a conceptual connector

Conceptual Configuration

During the conceptual configuration task you define the relations among your components and connectors. A conceptual configuration that contains component and connector types constrains how instances of these types can be interconnected. A conceptual configuration that contains instances defines which instances exist in the product and how they interconnect.

Components and connectors are interconnected through their ports and roles. Figure 4.4 shows the rules for these relations: A port can be connected to zero or more roles, and vice versa. We used a UML Note to describe additional constraints on the "cconnection"

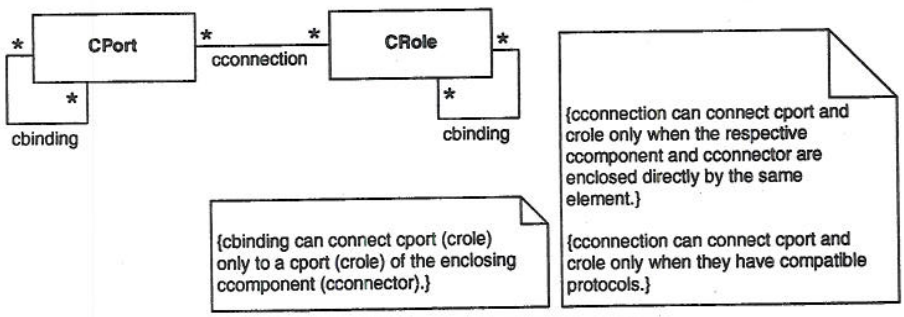

Figure 4.4. Definition of the relations between ports and roles

relation—constraints that would be hard to capture with the graphical notation alone. Connections are possible only when the associated protocols are compatible and only when the elements are nested within the same component or connector.

The binding relation is used when a component or connector is decomposed to bind an inner port to a port of the enclosing component, or to bind an inner role with a role of the enclosing connector.

Global Evaluation

We described earlier how you must continually evaluate your design decisions to see that you are following the results of the global analysis. During the central design tasks you must also periodically evaluate the interactions among your decisions.

4.1.3 Final Design Task: Resource Budgeting

The remaining task is to budget resources for the component and connectors instances identified during the central design tasks. This is done after the other tasks have been accomplished. Typically, resources get allocated rather late in the design process. Establishing budgets at the time the conceptual view is designed gives you the opportunity to evaluate properties of the architecture earlier in the design process.

With these budgets, you are able to use the conceptual structure as an application-level model for performing early analysis. The model may show you that the system meets certain properties, but it could make assumptions that are overly optimistic if it does not factor in overhead associated with the execution environment. If the application-level model does not meet the requirements, then you know that a major redesign of the system is needed.

During this resource budgeting phase you identify critical application-level resources that are limited or are to be shared, and assign budgets. For example, you might assign memory budgets to the components involved in data-intensive activities or communication bandwidth. You might also assign computation time budgets to the components responsible for time-critical activities. The architecture-level decisions were made during the central design tasks, so the decisions at this stage are more localized.

4.2 Design of Conceptual Architecture View for IS2000

Let's now go through these design activities in detail using IS2000 as an example. We won't do the complete design of the conceptual view, but for one section of the system—the ImagePipeline—let's go down to a final level of detail.

4.2.1 Global Analysis

In the previous chapter we began the global analysis for IS2000. Let's now continue it, looking specifically for strategies that are useful in designing the conceptual view. Of the strategies developed so far, the following should be useful here:

Issue: Aggressive Schedule
Strategy: *Make it easy to add or remove features.*

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

Issue: Easy Addition and Removal of Features
Strategy: *Separate components and modules along dimensions of concern.*
Strategy: *Encapsulate features into separate components.*
Strategy: *Decouple the user interaction model.*

We explained earlier that the third issue is an expansion of the strategy *Make it easy to add or remove features*. Now we specialize this issue further by creating a new issue, Easy Addition and Removal of Acquisition Procedures and Processing Algorithms, and provide strategies for solving it.

Also in the previous chapter we identified product factors related to the system's performance, but provided no specific strategies for addressing it. So we also add the issue Real-Time Acquisition Performance. The strategy listed for this issue applies to the conceptual view. Later, as we design the other views, we expect to add more strategies.

Easy Addition and Removal of Acquisition Procedures and Processing Algorithms

There are many acquisition procedures and processing algorithms. Implementation of each of these features is quite complex and time-consuming. There is a need to reduce complexity and effort in implementing such features.

Influencing Factors

O4.1: Time-to-market is short.

O4.2: Delivery of features is negotiable.

P1.1: New acquisition procedures can be added every three years.

P1.2: New image-processing algorithms can be added on a regular basis.

Continued

Easy Addition and Removal of Acquisition Procedures and Processing Algorithms (continued)

P1.3: New types of image and signal data may be required on a regular basis as the probe hardware changes.

Solution

Define domain-specific abstractions to facilitate the task of implementing acquisition and processing applications.

Strategy: *Use a flexible pipeline model for image processing.*

Develop a flexible pipeline model for implementing image processing. Use processing components as stages in the pipeline. This allows the ability to introduce new acquisition procedures quickly by constructing pipelines using both old and new components.

Strategy: *Introduce components for acquisition and image processing.*

Develop components for domain-specific acquisition and image processing to minimize the effects of any changes to the application domain. Developing a component model for domain-specific processing makes it easier to add or to upgrade components that are more efficient, and to offer new features.

Strategy: *Encapsulate domain-specific image data.*

To add or to upgrade to improved image-processing techniques, introduce components for domain-specific processing. Thus we can take advantage of industry standards and vendor solutions for transporting data over the network.

Related Strategies

See also *Encapsulate domain-specific hardware* (issue, Changes in General-Purpose and Domain-Specific Hardware).

4.2.2 Central Design Tasks: Components, Connectors, and Configuration

Next we turn to the central design tasks of the conceptual view: defining components and connectors, how they are configured, and global evaluation. These tasks cannot be done in a strictly sequential order. As you will see with IS2000, the design activities are interleaved.

Real-Time Application Performance

Meeting real-time performance requirements is critical to the success of the product. There is no separate source code for meeting the real-time performance requirements directly. The source code that implements functional processing must also meet the performance constraints.

Influencing Factors

T1, T3: A single standard processor running a real-time operating system is expected to provide sufficient computing resources.

P3.1, T2.1: The maximum signal data rate changes with changes in the probe hardware.

P3.2: Acquisition performance requirements are slightly flexible.

Solution

Partition the system into separate components for algorithms, communication, and control to provide the flexibility to implement several different strategies.

Strategy: *Separate time-critical components from nontime-critical components.*

To isolate the effects of change in the performance requirements, partition the system into components (and modules) that participate in time-critical processing and those that do not. This requires careful consideration at the interface between the real-time and nonreal-time sides of the system.

Related Strategies

See also *Separate components and modules along dimensions of concern* (issue, Easy Addition and Removal of Features).

The Conceptual Configuration

Let's start by representing the IS2000 system as a component with ports for its interactions with the outside world: the user, the probe hardware, and the network. The three kinds of interactions with the user are for controlling an acquisition procedure, displaying images on the monitor during the acquisition, and exporting the acquired images. Let's model these as three separate ports. The interactions with the probe hardware are to send

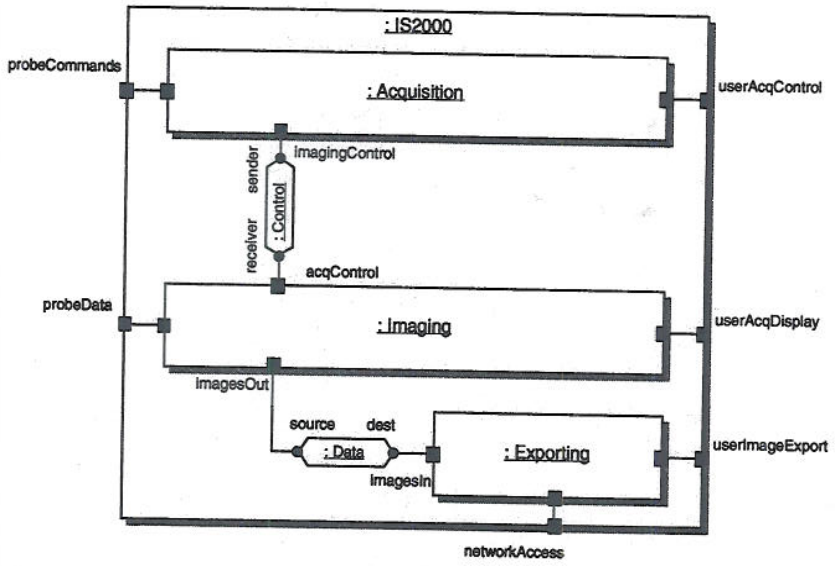

Figure 4.5. Initial high-level configuration of IS2000

commands and to receive data, so let's also make these separate ports. There is also a port for network access.

Next let's start decomposing the system. Here the new strategy *Introduce components for acquisition and image processing* is applicable, so IS2000 is decomposed into an Acquisition component, an Imaging component, and a third component for the rest of the system's functionality, which is to export the images. This initial high-level configuration is shown in Figure 4.5. We haven't yet precisely defined the connectors, but we know that the connector between the Acquisition and Imaging components functions as a control exchange, and the connector between the Imaging and Exporting components primarily passes data.

Although Figure 4.5 doesn't look like typical UML, it is actually a UML Class Diagram. Components, ports, connectors, and roles are modeled as stereotyped classes. For each of these we use a special graphical symbol to make the diagrams more clear. The symbol for a component is a shadowed rectangle, and its ports are shown as small black squares at the edge of the component, with the port name near the port symbol. The symbol for a connector is an elongated hexagon, and its roles are small black circles at the edge of the connector, with the role name nearby.

There are three kinds of relations in Figure 4.5: composition, binding, and connection. To show the decomposition of a component (IS2000) into a set of interconnected components and connectors, we use the UML convention of nesting the decomposition

inside the containing class. A binding between ports is shown with a UML association (a line), as is a connection between a port and a role.

To continue refining the conceptual configuration, let's apply the strategy *Encapsulate domain-specific hardware*. Because both the Acquisition and the Imaging components interact with the probe hardware, within each of these we must create a component that is responsible for communicating with the probe. This helps insulate the rest of the system when the hardware changes. It also gives us more reuse opportunities, for example, COTS components for motion control and in-house components for camera and sensor control, which we identified during global analysis.

In the Acquisition component, ProbeControl sets up the probe and controls it during the acquisition. In the Imaging component, DataCollection accepts the image data from the probe, and receives formatting information from ProbeControl that is interjected into the image data (Figure 4.6). Later, in the module architecture view, we will likely put the

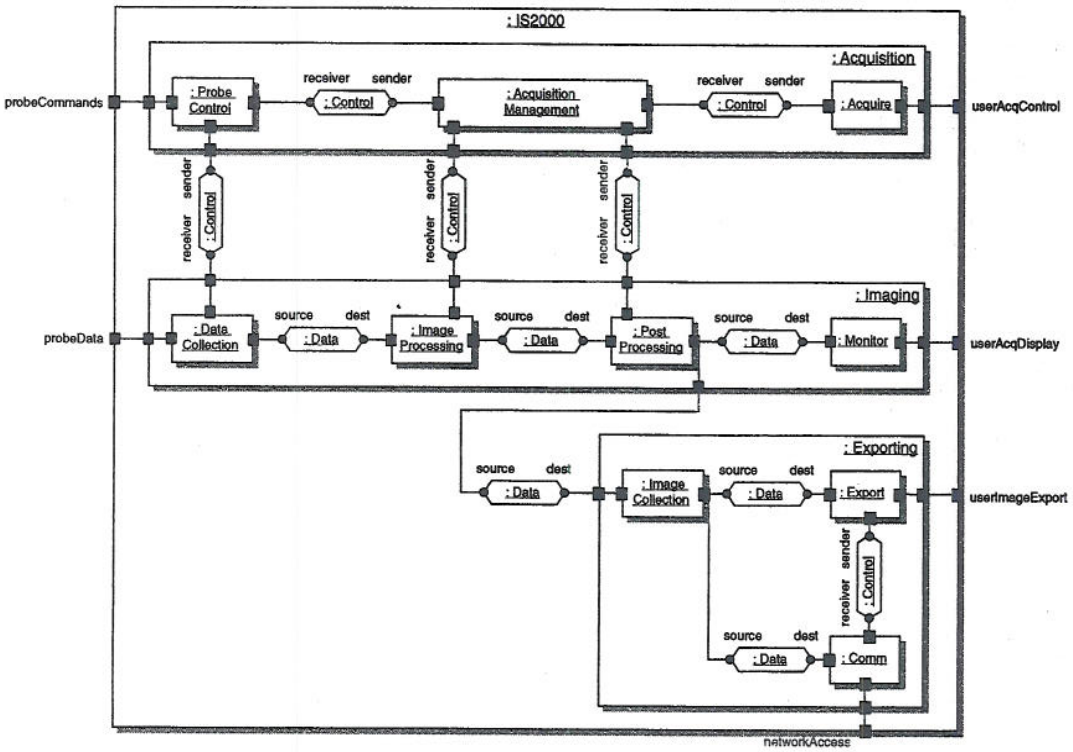

Figure 4.6. Refined high-level configuration of IS2000

modules for ProbeControl and DataCollection into the same layer, but in this view they are in separate components.

Similarly, applying the strategy *Decouple the user interaction model*, we introduce separate components within Acquisition, Imaging, and Exporting to handle the interaction with the user. In Figure 4.6 these are Acquire, Monitor, and Export.

To finish decomposing the Acquisition component, let's create the component AcquisitionManagement, which sets the policies for processing and organizing images. These are different for each acquisition. This is the result of applying the general strategy *Separate components and modules along dimensions of concern.* The Imaging component must abide by these policies, so there is a control connector from AcquisitionManagement to Imaging.

In Figure 4.6 there are actually two control connectors from AcquisitionManagement to Imaging, because we split the processing of images into two components: ImageProcessing and PostProcessing. Functionally these components are responsible for framing the raw data and processing it into images. But the strategy *Separate time-critical from nontime-critical components* indicates that one of the components (ImageProcessing) should contain only the time-critical initial processing of the raw data, and the rest of the processing should be put into the PostProcessing component.

The processed images are retained for as long as 24 hours, and the ImageCollection component is responsible for storing this information. This information also needs to be available to other systems over the network. The Export component is responsible for moving data to other systems, and Comm is responsible for the domain-specific communication of the image data. Here we applied the strategy *Encapsulate domain-specific image data* to guide the decomposition of the Exporting component into the Image Collection, Export, and Comm components.

All of these decisions are reflected in Figure 4.6, but this is not the final configuration: Not all the ports have been named, and the connectors are only roughly defined. As we continue with the design of each of the inner components, the missing details of the ports and connectors will be pinned down.

The ImageProcessing Component

Next let's look at the design of the ImageProcessing component in more detail. Image processing is responsible for framing raw data (arriving at the rawDataInput port) into images (sent out via the framedOutput port). The DataCollection component will do some buffering, but ImageProcessing must keep up with the buffered data to meet its real-time requirements. Image processing is controlled according to the acquisition procedure through the acqControl port.

ImageProcessing is decomposed into a Packetizer and possibly multiple ImagePipelines. Here we can apply the strategy *Use a flexible pipeline model for image processing.* The Packetizer bundles the incoming data into packets suitable for additional processing, which takes place in the ImagePipelines. This decomposition is shown in Figure 4.7.

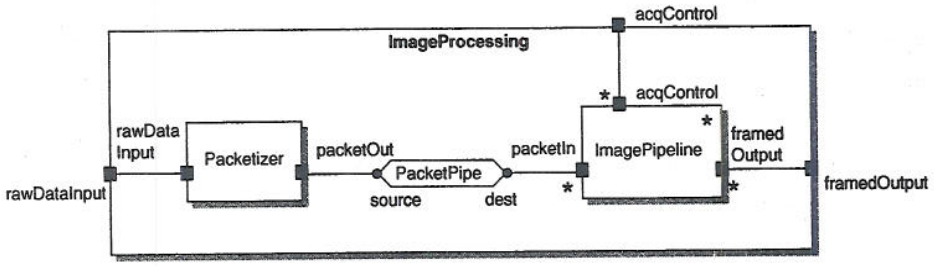

Figure 4.7. ImageProcessing decomposition

The fact that there can be zero or more ImagePipelines is shown with the UML notation for multiplicity. We use the convention that when the multiplicity of a class or association is not given, it is 1. So in Figure 4.7, there is exactly one Packetizer and one PacketPipe in ImageProcessing. The asterisk in UML means a multiplicity of 0 or more, so there can be zero or more ImagePipelines in ImageProcessing. The multiplicity on the binding between acqControl ports shows that the acqControl of ImageProcessing is bound to each of the ImagePipelines (a one-to-many binding). Similarly, the dest role of PacketPipe is connected to each of the ImagePipelines. All pipelines receive the same data from the Packetizer and filter out what they don't need.

You may have noticed that the names of components and connectors in Figure 4.6 are underlined and preceded by colons, but in Figure 4.7 they're not. In UML this signifies the distinction between an object (an instance) and a class. Here, in the conceptual configuration diagrams, we use this notation to distinguish between a component or connector instance (underlined and preceded by a colon) and a component or connector type. So Figure 4.6 shows exactly one configuration, and Figure 4.7 shows a set of configurations.

Whenever an aspect of the structure is reused, either in a different place of the configuration or in a different product, it is easier to describe it as a component, port, connector, or role type, with properties that apply to all instances of that type. In Figure 4.7, the properties that apply to instances of the components, ports, connectors, and roles are the association constraints—specifically, the decomposition, bindings, and connections.

Other properties of components, ports, connectors, and roles that we haven't yet discussed are their functional behavior and protocols. In the next section, we show how you can model these aspects using UML. If you'd prefer not to get into this topic yet, you can skip ahead to the next section, and read about describing behavior and protocols at a later time.

Defining Protocols and Behavior

The upper part of Figure 4.8 repeats some of the information that was in Figure 4.7. For example, the Packetizer component contains the rawDataInput port and the packetOut port, which is connected to the source role of the PacketPipe connector. The lower part of

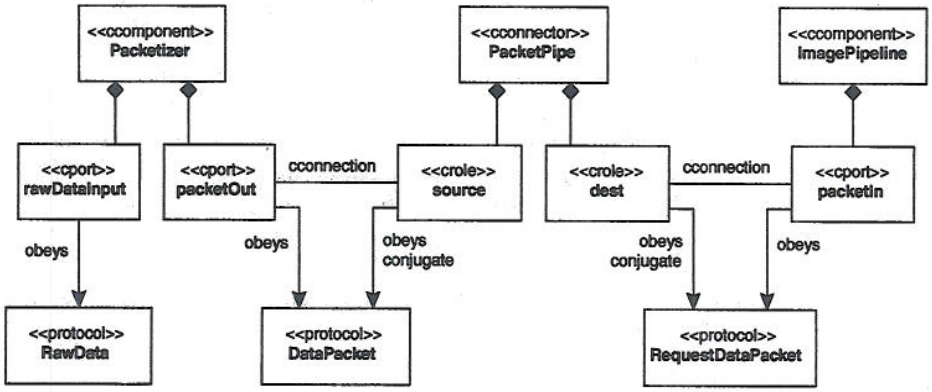

Figure 4.8. Protocols for Packetizer and PacketPipe

this figure shows the protocols obeyed by each of the ports or roles. In this section we use UML to describe these three protocols and the behavior of the Packetizer component and the PacketPipe connector.

A protocol is defined as a set of incoming message types, outgoing message types, and the valid message exchange sequences. The rawDataInput port of the Packetizer component obeys the RawData protocol specified in Figure 4.9. The declaration on the left side of the figure shows that the Packetizer receives two incoming messages and sends one outgoing message. The rawData message contains the raw data, rd. The sequence diagram on the right side of the figure shows the interleaving of the messages. The Packetizer receives a notice that the data is ready. It makes a request for the data and then receives it.

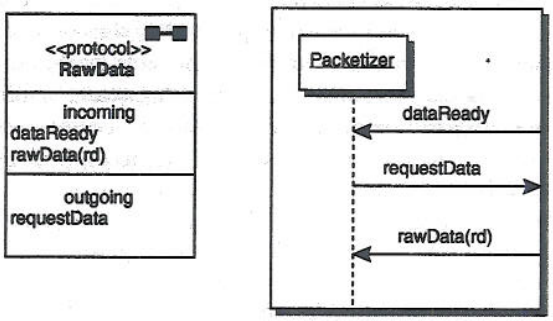

Figure 4.9. RawData protocol

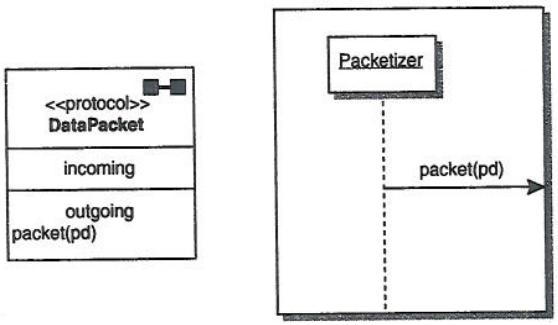

Figure 4.10. DataPacket protocol

The reason for the three-phase protocol is to give the Packetizer the opportunity to finish what it is doing and put itself into a state in which it is prepared to receive the data.[2]

The packetOut port adheres to the DataPacket protocol shown in Figure 4.10. The port sends the packets as they are available.

Since the beginning of the conceptual view design, we have kept decomposing components into sets of interconnected components and connectors. The Packetizer is not decomposed further, so it is time to define its behavior. Figure 4.11 uses a UML Statechart Diagram to define the behavior of the Packetizer in terms of the messages it sends and receives over its ports.

Transitions between the states have three parts, of the form Event [Guard] /Action, all of which are optional. Event represents an incoming message received by the component, and it triggers a state change. Action here is the component sending an outgoing message through one of its ports. Guard is a logical condition. A guarded transition occurs only if the guard resolves to true.

In the UML Statechart Diagram, the Packetizer initializes an empty packet, then waits. When data becomes available it adds this to the packet. When there is enough data to complete the packet, it sends out a message containing the packet and returns to the initial state.

The packetIn port of the ImagePipeline component adheres to the RequestDataPacket protocol shown in Figure 4.12. First it issues a subscription request. When it is ready for work it requests a packet and waits to receive one. Because ImagePipeline only requests a packet when it is ready to process one, no buffering is necessary. When it is notified to shut

2. The format of these protocol declarations comes from real-time object-oriented modeling, which is discussed at the end of the chapter.

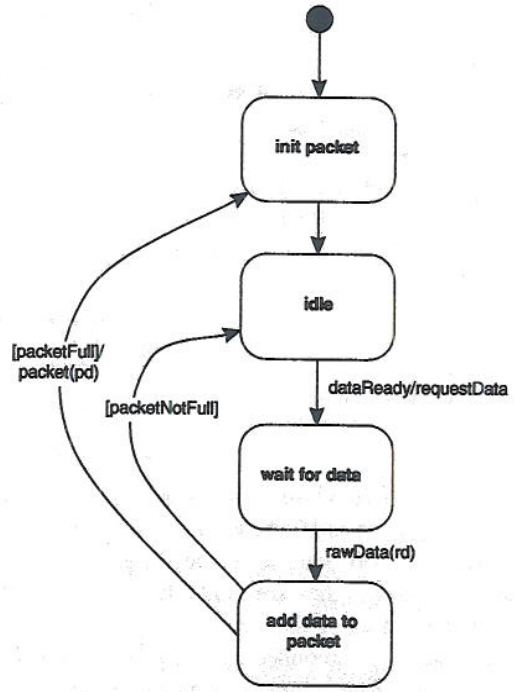

Figure 4.11. Packetizer behavior

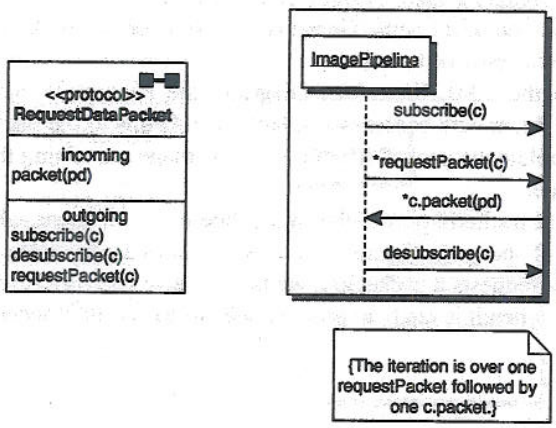

Figure 4.12. RequestDataPacket protocol

down, ImagePipeline cancels its subscription. Because of the indeterminate number of requests that can be generated and packets that can be received, let's add an asterisk and a constraint to express the message interleaving.

PacketPipe connects the Packetizer to one or more ImagePipelines. Its interfaces are called roles and, like ports, they adhere to protocols. In Figure 4.7, PacketPipe has two roles, source and dest. The source role obeys the conjugate of the DataPacket protocol. Being a conjugate means that the ordering of the messages is reversed so that incoming messages are now outgoing and vice versa. The source role of PacketPipe is connected to the packetOut port of Packetizer. Only ports and roles with compatible protocols can be connected to each other, and a common way of ensuring compatibility is to use a protocol and its conjugate. Protocol compatibility can be defined more broadly, and this is an area of active research.

The dest role obeys the conjugate of the RequestDataPacket protocol, so it is compatible with the packetIn port of ImagePipeline.

The behavior of the PacketPipe connector defines how its two roles interact; that is, how the outgoing messages are dependent on and interleaved with the incoming messages. The state machine in Figure 4.13 shows the behavior of the PacketPipe connector. Each of the image pipelines first subscribes for data. Each of them requests packets when it is ready for work, but no pipeline can process the next packet until all the other pipelines are also ready to receive it. In this way the pipelines stay synchronized. The PacketPipe connector must handle this synchronization among pipelines and provide some buffering to support it.

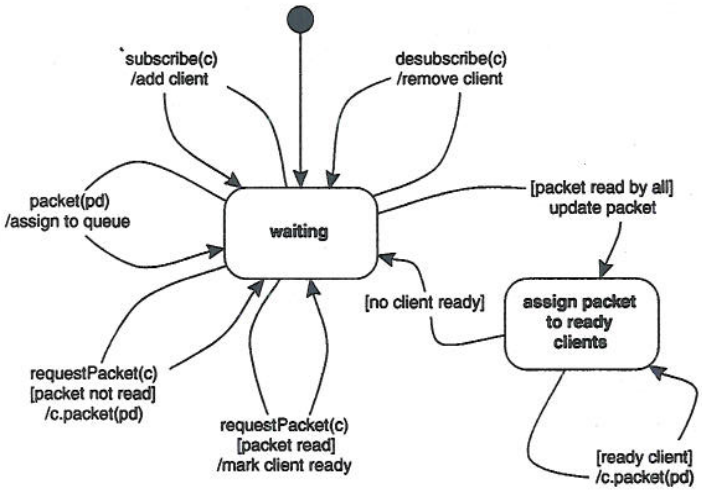

Figure 4.13. PacketPipe connector behavior

The ImagePipeline Component

With the definition of the Packetizer component and the PacketPipe connector complete, let's turn to the definition of ImagePipeline to complete the definition of image processing. The image pipeline has packets of raw data coming in (packetIn port) and framed images going out (framedOutput port). Its acqControl port responds to control messages from AcquisitionManagement.

Again, using the strategy *Use a flexible pipeline model for image processing*, let's decompose ImagePipeline into a number of stages and a pipeline manager (Figure 4.14). This makes it easier to reuse a stage across multiple pipelines, add new stages to a pipeline, or create new pipelines for new kinds of acquisition procedures. There is one PipelineMgr and one Framer per ImagePipeline, and one or more Imager components.

The Framer is always the first stage of the pipeline. It is connected to ImagePipe, which is connected to the imageIn port of an Imager. The imageOut port of an Imager is either connected to another ImagePipe or, when it is the last stage in the pipeline, is bound to framedOutput. The number of stages depends on the acquisition procedure selected by the user.

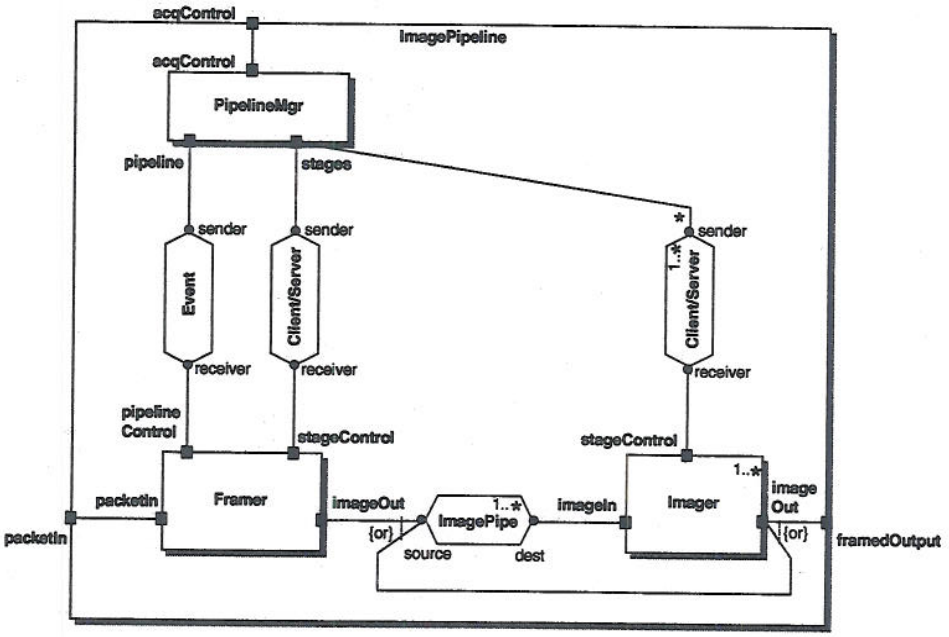

Figure 4.14. ImagePipeline decomposition

Although the specific processing is different, the stages share common features. They act as filters in the sense that they process a continuous stream of data, and pass it on to the next stage. They also all respond to control messages to start or stop processing. Almost all have ports for stageControl, imageIn, and imageOut. The first stage in the pipeline is the exception. It accepts packets of raw data rather than images and responds to messages for controlling the pipeline in addition to those for controlling the stage.

The connectors between stages are ImagePipes that transmit a stream of data from one stage to another. They have a source role for accepting input and a dest role for providing output. The imageOut ports and dest roles obey the same protocol, and the source roles and imageIn ports obey its conjugate. An ImagePipe has the property that it preserves the order of the data: The first item in is the first item out (FIFO queue).

PipelineMgr handles initialization and control messages from outside the system. Note that this high-level management or "control" is different from the control of the data flowing between the pipeline stages. The first provides the kind of acquisition control seen by the user, and it should be considered in the conceptual view. The second is part of the behavior of the connectors between the pipeline stages, and the details of its implementation are handled in the module view. Although these two kinds of control could end up in the same module in the module view, having an explicit conceptual view gives us the advantage of separating these two concepts in the design.

Control of the pipeline is through the pipeline and stages ports. There are commands to set up and to shut down the stages for an acquisition procedure, and to adjust the processing of the individual stages. The connector between the PipelineMgr and each of the stages is called Client/Server. On the sender side, control requests are accepted, and on the receiver side the control request is sent and a corresponding reply is received. On the pipeline port, the PipelineMgr sends commands to signal the start, stop, pause, and end of images, and no reply is expected.

After initialization, the Framer checks for any messages from the PipelineMgr. When a message is received to start framing images, the Framer requests packets from the Packetizer and processes them repeatedly until notified of the end of the image. It then passes the framed image on to the next stage in the pipeline before checking for new messages and repeating the cycle.

The Imager component's functionality is a subset of that of the Framer. It does not have a pipelineControl port. It receives images rather than packets, and thus it does not have the added complexity of constructing the framed image.

We've summarized the behavior of the ImageProcessing component in Figure 4.15. This is a UML Sequence Diagram, so it can only show a particular sequence of interactions among instances. Here, to emphasize the relationships among the components, we show only component instances, not ports, connectors, or roles.

The acquisition manager (not shown) starts an acquisition by sending a message to PipelineMgr. PipelineMgr is created for this particular acquisition. It in turn creates the stages to do the necessary processing, here shown as instances of Framer and Imager. The Packetizer has already been initialized at system start-up. When there is enough raw data to make up a packet, it gets sent to the first stage in the image pipeline—the Framer. The Framer does its processing and then passes the image along to the next stage and so on

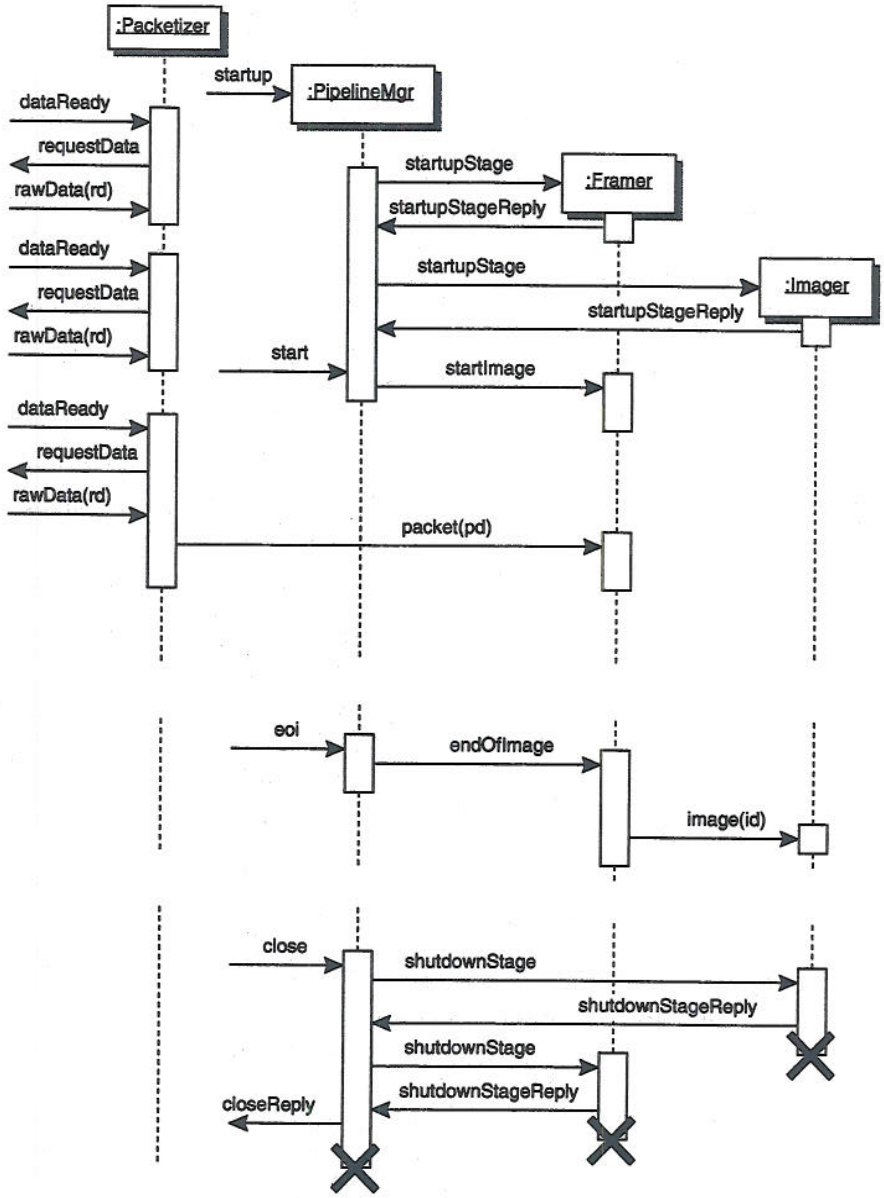

Figure 4.15. Sequence Diagram for the ImageProcessing component

Pipeline	Stages
pipeline1: ImagePipeline	stage1: Framer, stage2: Imager, stage3: Imager, stage4: Imager

Table 4.1. Sample Pipeline Instances

down the line (if there are any additional stages). The acquisition manager controls the acquisition and sends messages to signal the start and end of images and to signal when it is time to shut down the acquisition.

Notice that ImagePipeline decomposition depicted in Figure 4.14 contains only classes (component and connector types), not instances. The final step would be to define exactly which component instances are used for each acquisition protocol. Because each acquisition procedure has its own image pipeline, we could draw a diagram similar to Figure 4.14 for each acquisition procedure, containing all the instances it uses. But here, because the structure of an image pipeline is relatively simple, we prefer to use a table that lists the instances in each pipeline. Table 4.1 shows an entry for the instances in a pipeline. It records the configuration information that can vary among pipelines, that is, the number and order of stages in each pipeline. Information about the pipeline manager and the connectors is not included because this information is derived from Figure 4.14.

Recovery and Diagnostics

Up to this point, our focus has been on introducing components to support the functional features. One exception was the global property for real-time processing, which influenced how we divided the system into components.

Now let's consider additional global properties, from the category of failure detection, reporting, recovery. Because we have not yet considered product factors in this category, let's return to the global analysis to add product factors, then develop additional strategies for handling them.

One product factor is that the image data must be recoverable in the event of a failure. Raw data that has been captured should be recovered and processed when the system is back on-line. Acquisitions may take a long time. If an acquisition consists of 100 images and the system fails during acquisition of the ninety-ninth image, the first 98 images should be recoverable. Recovery affects a number of components: During normal processing, components must gather the data needed to recover. After a failure, the system needs to process the recovered data, which includes recovering a possibly incomplete acquisition.

Depending on the available resources, recovery may have a major impact on the architecture design. If the data to be recovered is already in a persistent store (for example, a file or database) or if there are enough resources to take "snapshots" of the internal state

and input/output, then the job is practically done. If not, we need to design the architecture so that the data is accessible when it needs to be recovered. Once the architecture approach is determined, the impact of future changes can be localized to one or more of the recovery components. Table 4.2 shows the analysis of product factor P5.4, recovery.

Product Factor	Flexibility and Changeability	Impact
P5: Failure detection, reporting, recovery		
P5.1: Error classification		
System errors are classified according to their type and severity.	Error classification is likely to change during development. It is affected by functional features, the user interaction model, and the probe model.	There is a moderate impact on all components. Error logging is affected.
P5.2: Error logging		
Error logging is used to capture diagnostic, clinical, and software trace events.	Error logging is affected by error classification, error-handling policy, and the hardware platform. It is likely to change during development.	There is a moderate impact on all components. Acquisition performance may be affected.
P5.3: Support for use of error logs		
Error logs can be retrieved and viewed for the purpose of error tracking and diagnosis.	Requirements for the use of an error log are expected to be quite stable.	There is a minimal impact on components.
P5.4: Recovery		
Acquired imaged data must be recoverable in the event of a system failure.	Recovery requirements are stable. They are flexible to adapt to changes in acquisition size, data rate, format, and recovery support by the data storage system.	There is a moderate to large impact on all components involved in recovery.

Table 4.2. Factors Added During Design of Conceptual View

The other new product factors in this table are related to diagnostics: error classification, error logging, and support for using error logs in diagnosis. Error classification and logging affect all parts of the system and all its modes (for example, acquisition, recovery, diagnostics) of operation. Because performance is critical, any strategy for error logging must minimize its impact on performance. To reduce its scope, we can encapsulate it, but it cannot be localized. The distributed part of logging that interacts with the rest of the system should be easy to use and easy to incorporate.

From these product factors, we identify two new issues: Implementation of Recovery and Implementation of Diagnostics. The strategies for these issues need to show how we handle two requirements that have a global impact.

Implementation of Recovery

Recovery is the responsibility of many components. We need to ensure that support for recovery is uniformly implemented by all relevant components.

Influencing Factors

P5.4: Acquired imaged data must be recoverable in the event of a system failure. Recovery requirements are stable and can be flexible to reduce effort of implementation.

Solution

Apply the principle of separation of concerns to introduce a separate recovery mode of operation for each component. Ensure that the data to be recovered is accessible to all components.

Strategy: ***Introduce a recovery mode of operation.***

Introduce a recovery mode to localize change. This means introducing components for cleanup, shutdown, and crash recovery. The scope of impact can be reduced cost-effectively by introducing a recovery mode for processing the recovered data. Using a separate recovery model means that we don't have to worry about affecting the system's performance during an acquisition. However, users may wish to execute recovery in parallel with other activities in the future. Existing components involved in processing will now have an additional responsibility to provide an interface to run in recovery mode. We also want to encapsulate file system and database recovery support.

Strategy: ***Make all data at the point of recovery persistent and accessible.***

Make data at the point of recovery persistent and accessible through an interface at the architecture level for easier implementation of a recovery mechanism. If it is not accessible, then recovery may require snapshots of the internal state.

Implementation of Diagnostics

Error logging will be used to support diagnostics. Responsibility for diagnostics is shared by all components. We need to ensure that support for diagnostics is uniformly implemented by all components.

Influencing Factors

P5.1: System errors are classified according to their type and severity. Error classification is likely to change.

P5.2: Error logging is affected by error classification, error-handling policy, and the hardware platform. It is likely to change.

P5.3: Requirements for the use of error logs are expected to be stable.

Solution

Support for diagnostics is needed during development, for tracing, and at runtime. The design of the diagnostics has many elements. It includes the log itself (which is a repository of the error information), the types of errors being reported, and the kinds of information captured (for example, component, error, location). Components need to log information, and the user needs to be able to read the logs.

Strategy: *Define an error-handling policy.*

Define a policy to avoid, detect, and handle different classes of errors. This includes how to define and use exceptions, and how to select an exception mechanism. Provide guidelines on how to avoid errors, as well as how to detect and report them.

Strategy: *Reduce effort for error handling.*

Error handling is tedious, and many developers dislike it. Make their task easier by using tools to support the error-handling policy. For example, there are tools that take an error classification scheme as input and generate code templates for error reporting.

Strategy: *Encapsulate diagnostics components.*

Localize the impact of error logging by encapsulating one component for error logging and another for the use of logs. Define product-specific interfaces so that their implementation can be easily replaced.

Strategy: *Use standard logging services.*

To meet the schedule and to take advantage of industry standards, use standard services such as message catalogs and the file system to read and review log files.

With these strategies in place, let's turn first to designing the architecture to support recovery. The strategy *Make all data at the point of recovery persistent and accessible* influences where we draw the boundaries among the components involved in processing the image data.

Revisiting Figure 4.6 to examine the components involved in an acquisition, we see that data is available in three places: at the point where data is coming into ImageProcessing, between ImageProcessing and PostProcessing, and where data is leaving PostProcessing. If recovering data at one of these points is feasible and meets the requirements, then no structural changes to the architecture need to be made. If not, then we may need to divide one of the components to get access to the data at the appropriate point, or shift the boundary of responsibility between two components. Any potential changes must also be evaluated with respect to the strategies *Separate time-critical from nontime-critical components* and *Use a flexible pipeline model for image processing*, because these were the source of our earlier design decisions for this part of the system.

Now let's consider the implications of recovering the data at each of the three points. Recovering data at the point before ImageProcessing is not feasible because of the amount of raw data that needs to be stored. The point after ImageProcessing cuts down the volume to a manageable size, and ImageProcessing is so fast that we would only lose an image or two if we recovered data here. The point after PostProcessing doesn't reduce the volume of data much further. PostProcessing, however, takes much more time than ImageProcessing. If we recover at this point we would lose much of the image data in the beginning of the pipeline and not meet our recovery requirements. Thus we need to make image information recoverable at the point between ImageProcessing and PostProcessing.

Because the data must be available even if the system goes down, the connector between ImageProcessing and PostProcessing must have the property that the data it transfers is persistent. This is a new connector type, called PersistentDataPipe.

We also apply the strategy *Introduce a recovery mode of operation*. This causes us to add components that encapsulate the functionality for the recovery mode, and to add a new recovery port to existing components, such as PostProcessing, for when they run in recovery mode.

Next let's look at what is needed to support diagnostics. For error logging, let's introduce two new components: Logger and LogReader. The Logger is responsible for receiving and storing event messages in a log. To log an event with the Logger, existing components need a new logging port. The LogReader allows a user to review the logs.

4.2.3 Final Design Task: Resource Budgeting

The remaining task in the conceptual view is to budget resources for component and connector instances identified during the central design tasks. The resources we consider are those that are limited or are to be shared. If we can't come up with a balanced budget, meaning that we can't assign reasonable budgets to components and connectors without overrunning our available resources, then we must return to the central design tasks with additional information about resource constraints.

For example, we establish memory budgets for the components involved in data-intensive activities based on data flow rate. The FIFO queues for the pipes between the pipeline stages are given a budget based on predicted image size. The budget for PersistentDataPipe, the connector between ImageProcessing and PostProcessing, is based on the number of images that we predict will accumulate between the two, which is in turn based on their relative speed of processing an image.

We need computation time budgets for the components responsible for time-critical activities. These budgets are dictated by the end-to-end deadlines given in the requirements. For example, the time between detection of a real-time event (for example, stop) and its effect on event data must be less than 1 msec. This affects the budgets for DataCollection, ImageProcessing, and PostProcessing. This is an iterative process. Budgets are assigned to these top-level components. Budgets are then assigned to subcomponents and so on. A preliminary analysis can then be performed to ensure the latency requirement is met.

The system must be able to support a sustained data rate of 2MB per second generated by the probe hardware. This affects both the time budget and the buffer size for the DataCollector component.

4.2.4 Design Summary for IS2000 Conceptual View

To design the conceptual view, we identified components based on our global analysis and captured their responsibilities and dependencies. We refined the dependencies by introducing connectors, with details about their communication protocols. We used these building blocks to configure our particular system. Global properties such as error logging, recovery, and real-time processing caused us to introduce additional components or to make adjustments in the interfaces or boundaries of existing components. Table 4.3 summarizes the main design decisions we made in this chapter, and the rationale for them.

Design Decision	Rationale
Global analysis	
Add new issue: Easy Addition and Removal of Acquisition Procedures and Processing Algorithms.	Strategy: *Make it easier to add or remove features.* Strategy: *Encapsulate features into separate components.*
Add new issue: Real-Time Acquisition Performance.	Product factors for performance

Table 4.3. Sequence of Design Decisions for IS2000 Conceptual View

Design Decision	Rationale
The conceptual configuration	
Decompose IS2000 into three main components: Acquisition, Imaging, and Exporting.	Strategy: *Introduce components for acquisition and image processing.*
Introduce components that are an abstraction of the probe hardware (ProbeControl, DataCollection).	Strategy: *Encapsulate domain-specific hardware.*
Introduce components for the user's interactions with the system (Acquire, Monitor, Export).	Strategy: *Decouple the user interaction model.*
Create AcquisitionManagement component.	Strategy: *Separate components and modules along dimensions of concern.*
Split processing of images into two components: ImageProcessing and PostProcessing.	Strategy: *Separate time-critical components from nontime-critical components.*
Introduce ImageCollection and Comm components	Strategy: *Encapsulate domain-specific image data.*
The ImageProcessing component	
Decompose ImageProcessing into Packetizer and multiple ImagePipelines, connected by a PacketPipe.	Strategy: *Use a flexible pipeline model for acquisition and image processing.*
Defining protocols and behavior	
Add RawData, DataPacket, and RequestDataPacket protocols. Describe behavior of Packetizer and PacketPipe.	Describe behavior of ports and roles using protocols. When components and connectors are not decomposed further, describe their behavior using UML Statechart Diagrams.

Table 4.3. Sequence of Design Decisions for IS2000 Conceptual View *(continued)*

Design Decision	Rationale
The ImagePipeline component	
Decompose ImagePipeline into a Framer and multiple Imager stages, connected by ImagePipes.	Strategy: *Use a flexible pipeline model for acquisition and image processing.*
Put one PipelineMgr in each ImagePipeline.	Provide support for user-level acquisition control.
Recovery and diagnostics	
Add new factors to the product factor table: P5.1: Error classification, P5.2: Error logging, P5.3: Support for use of error logs, and P5.4: Recovery.	Identify new factors during the analysis of global properties related to failure detection, reporting, and recovery.
Add new issue: Implementation of Recovery.	Identify new issues during the analysis of product factors for recovery.
Add new issue: Implementation of Diagnostics.	Identify new issues during the analysis of product factors for error classification, logging, and use of logs.
Introduce a recovery mode and recover data at the connector between ImageProcessing and PostProcessing.	Strategy: *Introduce a recovery mode of operation.* Strategy: *Make all data at the point of recovery persistent and accessible.*
Introduce Logger and LogReader components.	Strategy: *Define an error-handling policy.* Strategy: *Encapsulate diagnostic components.*
Resource budgeting	
Assign data budgets to connectors.	Budgets depend on image size, number of images, and relative processing speed of ImageProcessing and PostProcessing.
Assign time budgets to the time-critical components.	Budgets depend on end-to-end deadlines given in requirements.

Table 4.3. Sequence of Design Decisions for IS2000 Conceptual View *(continued)*

For IS2000, we began with the conceptual configuration of the system, mapping functionality to components and determining the rough behavior of the connectors.

If you start with a domain-specific or reference architecture, it will most likely provide you with similar information. It will probably give you a more precise indication of the component and connector types, but not all the details of the configuration, such as exactly which instances of components and connectors exist.

If you use instead an architectural style for some part of the architecture, you will not have specific configuration information, but you will have precise information about component and connector types, and which interconnections are allowed. It is up to you to decide how to map the functionality of the system to the components and connectors in the style.

4.3 Summary of Conceptual Architecture View

Table 4.4 shows the elements, relations, and artifacts of the conceptual architecture view. The elements and relations are the building blocks, and the artifacts are the architecture descriptions or documentation. For the elements and relations, we show which UML meta-model class (the kind of UML element) is used to represent them, and the notation we use in the diagrams that describe the conceptual view.

Element	UML Element	New Stereotype	Notation	Attributes	Associated Behavior
CComponent	Active class	<<ccomponent>>		Resource budget	Component behavior
CPort	Class	<<cport>>		—	—
CConnector	Active class	<<cconnector>>		Resource budget	Connector behavior
CRole	Class	<<crole>>		—	—
Protocol	Class	<<protocol>>		—	Legal sequence of interactions

Table 4.4. Summary of Conceptual Architecture View

Relation	UML Element	Notation	Description
composition	Composition	Nesting (or )	A component or connector can be decomposed into a configuration of interconnected components and connectors.
cbinding	Association	————	A port can be bound to a port of the enclosing component. A role can be bound to a role of the enclosing connector.
cconnection	Association	————	A component's port can be connected to a connector's role when both are directly enclosed by the same element.
obeys	Association	obeys	A port or role obeys a protocol.
obeys conjugate	Association	obeys conjugate	A port or role obeys the conjugate of a protocol.
Artifact		**Representation**	
Conceptual configuration		UML Class Diagram	
Port or role protocol		ROOM protocol declaration (uses UML Sequence or Statechart Diagram)	
Component or connector behavior		Natural language description or UML Statechart Diagram	
Interactions among components		UML Sequence Diagram	

ROOM = real-time object-oriented modeling; UML = Unified Modeling Language.

Table 4.4. Summary of Conceptual Architecture View *(continued)*

Components and connectors can have a resource budget recorded as an attribute, and can have an associated description of their behavior. The description of this behavior is one kind of artifact: It can be represented by a natural language description or a UML Statechart Diagram.

For a protocol, the associated behavior describes the legal sequence of incoming and outgoing messages. A protocol description is another kind of artifact, represented by a real-time object-oriented modeling (ROOM) protocol declaration.

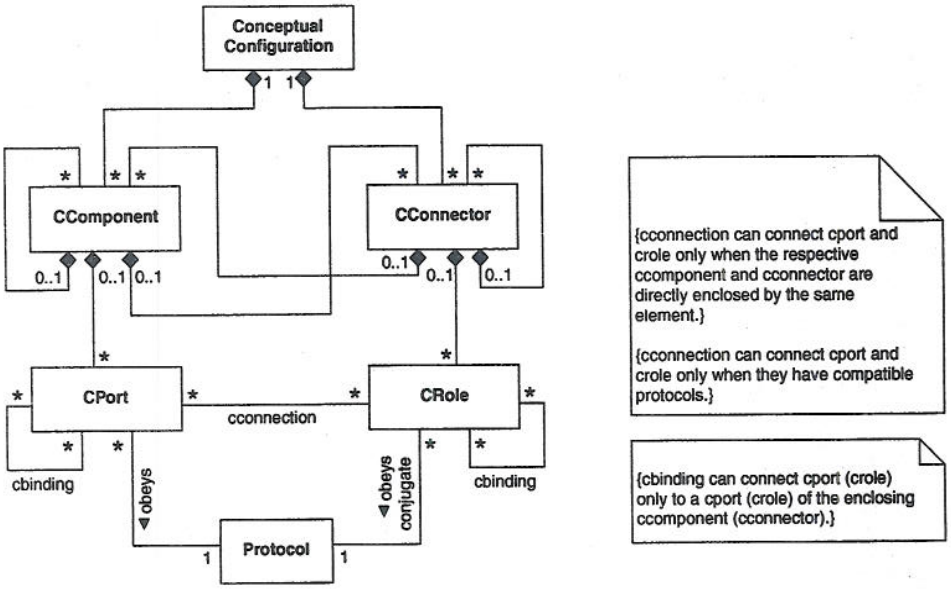

Figure 4.16. Meta-model of the conceptual architecture view

The main type of artifact is a conceptual configuration, which is represented by a UML Class Diagram that contains the elements and relations of the conceptual view. We define more precisely how the relations can be applied to the elements using the meta-model in Figure 4.16. Starting at the top, a conceptual configuration contains zero or more components and zero or more connectors.

Conceptual components contain ports that define how the component can interact with other elements of the configuration. Components can be decomposed further into a collection of components and connectors. When a component is decomposed, its ports can be bound to ports of the components it (directly) contains.

Conceptual connectors are analogous to conceptual components, except that their interaction points are called *roles* rather than *ports*. Connectors can also be decomposed into a set of components and connectors, and roles can be bound only to the role of an enclosing connector.

The connection relation is used to connect a component's port to a connector's role, but it is only allowed when the component and connector are directly enclosed by the same element. In addition, both ports and roles obey a protocol (or obey its conjugate), and a connection is allowed only when the associated protocols are compatible.

4.3.1 Traceability

Describing the relationships of the conceptual architecture view to requirements, external factors, and other architecture views provides traceability. During global analysis you create factor tables for characterizing the influencing factors, and you create issue cards for capturing strategies that accommodate the factors. A table such as Table 4.3 shows which strategies are used in each of the design decisions of the conceptual view. When one of the factors changes, its influence can be traced to the affected components. The following two items should be traceable in the architecture view:

1. *Critical requirements.* Product features and requirements can be traced to the conceptual elements. This helps you figure out the impact that existing or new requirements will have on the software architecture. Conversely, you can trace the design decisions to requirements, and determine how well they are met.
2. *Organizational and technological factors.* As mentioned at the beginning of the chapter, external factors and the strategies to deal with them have a direct impact on the conceptual view. Making them part of your recorded architecture description makes it easier to understand the implications of change during the design process and in the future.

As you continue the architecture design and introduce the module and execution architecture views, you will support traceability by recording how elements in the conceptual architecture view are mapped to modules or processes.

4.3.2 Uses for the Conceptual Architecture View

When the conceptual view is explicit, it can be used to reason about and/or predict important system properties. You may use the conceptual view for

- Use-cases and scenarios
- Performance estimation
- Safety and reliability analysis
- Target independent testing
- Understanding the static and dynamic configurability of the system
- Effort estimation (preliminary; not including the infrastructure)

Additional Reading

Shaw and Garlan (1996) provide a definition of software architecture in terms of components and connectors that is close to our conceptual view. They also provide additional information about work that addresses the conceptual view concerns of constructing systems as assemblies of components and first-class connectors, and information about protocol compatibility and interface checking.

Selic, Gullekson, and Ward (1994) describe a methodology for ROOM and strategies for dealing with complexity. ROOM is an example of an architecture description language that gives software engineers the ability to model aspects of the conceptual view, such as composing software systems using communicating objects. Our description for protocols is influenced by ROOM. Selic and Rumbaugh (1998) describe how certain ROOM constructs can be modeled in UML. Their notion of capsules, ports, and protocols is similar to our conceptual components, ports, and protocols.

Diagrams are useful for improving the understanding of normal operation or functionality of the system. Tables are useful for checking for completeness—to determine whether something is missing. To supplement the diagrams, tables of components, their responsibilities, and collaborations give additional details. CRC cards (Beck, 1991), documenting the component, its responsibility, and collaborations, are useful for capturing this information.

Drongowski (1993) describes the software architecture of a real-time system. Nord and Cheng (1994) perform an evaluation of this system to assess the performance properties based on the conceptual architecture and to demonstrate how this provides feedback to the architecture design decisions.