Describing Software Architecture with UML

C. Hofmeister, R. L. Nord, D. Soni
Siemens Corporate Research, Princeton, New Jersey, USA
{chofmeister, rnord, dsoni}@scr.siemens.com

Key words: Software architecture, UML, architecture descriptions, multiple views

Abstract: This paper describes our experience using UML, the Unified Modeling Language, to describe the software architecture of a system. We found that it works well for communicating the static structure of the architecture: the elements of the architecture, their relations, and the variability of a structure. These static properties are much more readily described with it than the dynamic properties. We could easily describe a particular sequence of activities, but not a general sequence. In addition, the ability to show peer-to-peer communication is missing from UML.

1. INTRODUCTION

UML, the Unified Modeling Language, is a standard that has wide acceptance and will likely become even more widely used. Although its original purpose was for detailed design, its ability to describe elements and the relations between them makes it potentially applicable much more broadly. This paper describes our experience using UML to describe the software architecture of a system.

For these architecture descriptions, we wanted a consistent, clear notation that was readily accessible to architects, developers, and managers. It was not our goal to define a formal architecture description language. The notation could be incomplete, but had to nevertheless capture the most important aspects of the architecture. In this paper we start by giving an overview of the kinds of information we want to capture in a software architecture description. Then we give an example of a software architecture

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35563-4_35

P. Donohoe (ed.), *Software Architecture*

description for part of particular system: the image processing portion of a real-time image acquisition system. The final section discusses the strengths and weaknesses of UML for describing architecture.

We separate software architecture into four views: conceptual, module, execution, and code. This separation is based on our study of the software architectures of large systems, and on our experience designing and reviewing architectures (Soni, 1995). The different views address different engineering concerns, and separation of such concerns helps the architect make sound decisions about design trade-offs.

The notion of this kind of separation is not unique: most of the work in software architecture to date either recognizes different architecture views or focuses on one particular view in order to explore its distinct characteristics and distinguish it from the others (Bass, 1998). The 4+1 approach separates architecture into multiple views (Kruchten, 1995). The Garlen and Shaw work focuses on the conceptual view (Shaw, 1996). Over the years there has been a great deal of work on the module view (Prieto-Diaz, 1986). There is other work that focuses on the execution view, and in particular explores the dynamic aspects of a system (Kramer, 1990; Purtilo, 1994). The code view has been explored in the context of configuration management and system building.

The conceptual view describes the architecture in terms of domain elements. Here the architect designs the functional features of the system. For example, one common goal is to organize the architecture so that functional features can be added, removed, or modified. This is important for evolution, for supporting a product line, and for reuse across generations of a product.

The module view describes the decomposition of the software and its organization into layers. An important consideration here is limiting the impact of a change in external software or hardware. Another consideration is the focusing of software engineers' expertise, in order to increase implementation efficiency.

The execution view is the run-time view of the system: it is the mapping of modules to run-time images, defining the communication among them, and assigning them to physical resources. Resource usage and performance are key concerns in the execution view. Decisions such as whether to use a link library or a shared library, or whether to use threads or processes are made here, although these decisions may feed back to the module view and require changes there.

The code view captures how modules and interfaces in the module view are mapped to source files, and run-time images in the execution view are mapped to executable files. The partitioning of these files and how they are

organized into directories affect the buildability of a system, and become increasingly important when supporting multiple versions or product lines.

Each of the four views has particular elements that need to be described. The elements must be named, and their interface, attributes, behavior, and relations to each other must be described. Some of the views also have a configuration, which constrains the elements by defining what roles they can play in a particular system. In the configuration, the architect may want to describe additional attributes or behavior associated with the elements, or to describe the behavior of the configuration as a whole.

In the next four sections, we show how we used UML to describe each of these four views, starting with the conceptual view and ending with the code view. To make the explanation clearer, we use an example from an image acquisition system.

The image acquisition system acquires a set of digitized images. The user controls the acquisition by selecting an acquisition procedure from a set of predefined procedures, then starting the procedure and perhaps adjusting it during acquisition. The raw data for the images is captured by a hardware device, a "camera", and is then sent to an image pipeline where it is converted to images. The image pipeline does this conversion, first composing the raw data into discrete images, and then running one or more standard imaging transformations to improve the viewability of the images. The image pipeline is the portion of the system that we will use as an example.

2. CONCEPTUAL ARCHITECTURE VIEW

The basic elements in the conceptual view are components with ports through which all interactions occur, and connectors with roles to define how they can be bound to ports. The components and connectors are bound together to form a configuration. In order to bind together a port and role in a configuration, the port and role protocols must be compatible. Components can be decomposed into other components and connectors. These elements, their associated behavior, and the relations of the conceptual view are summarized in Table 1.

Table 1. Elements of a conceptual architecture view

Elements	*Behavior*	*Relations*
component	component functionality	component decomposition
port	port protocol	port-role binding (for
connector	connector protocol	configuration)
role	role protocol	

Figure 1 is a UML diagram that describes much of the conceptual view for the image pipeline. It is represented by the ImagePipeline component, which has ports acqControl for controlling the acquisition, packetIn for the incoming raw data, and framedOutput for the resulting images.

The ImagePipeline is decomposed into a set of components and connectors that are bound together to form a configuration. The components, ports, and connectors are a stereotype of Class[1], but we use the convention of special symbols for ports and connectors (and leave off the stereotype for components) in order to make the diagrams easier to read. Roles are shown as labels on the port-connector associations. We also use the convention that when an association's multiplicity is not specified, it is assumed to be one.

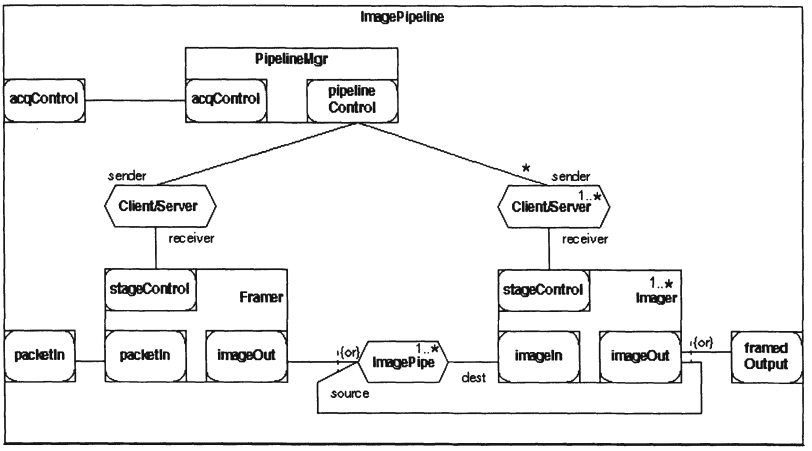

Figure 1. Conceptual configuration

The multiplicities on the components, connectors, and bindings show the set of allowable configurations. Each acquisition procedure has a distinct set of processing steps, represented by the Imager component. So the diagram shows the general structure of an image pipeline, which all acquisition procedures adhere to.

The first stage of the pipeline is the Framer, followed by one or more subsequent stages, represented by the Imager. Each of the stages is connected to

"A stereotype is, in effect, a new class of modeling element that is introduced at modeling time. It represents a subclass of an existing modeling element with the same form (attributes and relationships) but with a different intent... To permit limited graphical extension of the UML notation as well, a graphic icon or a graphic marker (such as texture or color) can be associated with a stereotype." (UML, 1997)

the pipelineControl port via a Client/Server connector. The Imager component has a multiplicity of "1..*", meaning that an acquisition procedure has one or more of these later stages.

The Imager is bound to "1..*" Client/Server connectors, but the association is one-to-one, so each Imager instance is bound to exactly one Client/Server instance. Each Client/Server instance is bound to the pipelineControl port of exactly one PipelineMgr, but pipelineControl is bound to all Client/Server instances in the pipeline. Similarly the "1..*" ImagePipe connectors have a one-to-one association with the Imagers. Because the bindings also have multiplicities, we can conclude that there are the same number of Client/Server, Imager, and ImagePipe elements bound together in a legal configuration.

We use the "{or}" annotation at the source side of the ImagePipe to show that an ImagePipe is either bound to the output of the first stage or a later stage. But while the output of the first stage (the Framer) is always bound to the ImagePipe, the later stages could be bound to framedOutput. When a later stage is bound to framedOutput, it is necessarily the last stage in the pipeline.

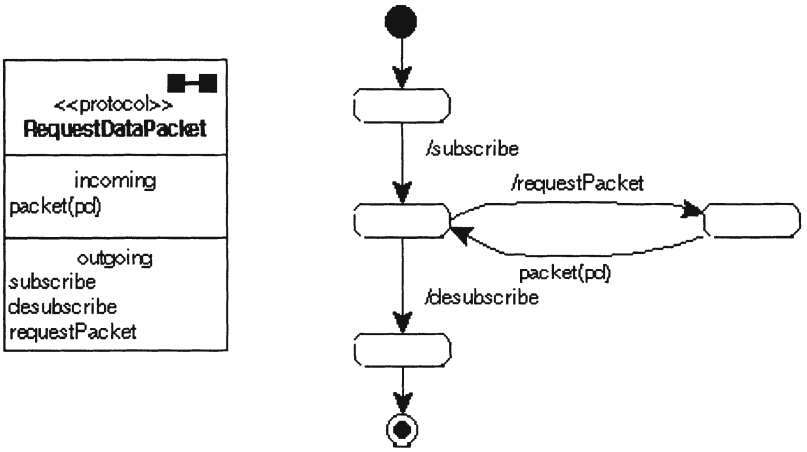

Figure 2. Protocol for packetIn Port

Figure 2 shows the protocol RequestDataPacket, which the packetIn ports on the ImagePipeline and Framer follow. We have adopted the ROOM notation here, showing the incoming and outgoing messages, then either a sequence diagram or state diagram to show the legal sequences of these messages (Selic, 1994; Selic, 1998).

The resource budgets are attributes of the components and connectors. They can be described in the attribute box of the appropriate class in a UML diagram, in a table, or in text.

For the conceptual view, we represent components, ports, and connectors as stereotyped classes. Decomposition is shown with nesting (association), and bindings are shown by association. We use:

- UML Class Diagrams for showing the static configuration.
- ROOM protocol declarations and UML Sequence Diagrams or State Diagrams for showing the protocols that ports adhere to.
- UML Sequence Diagrams for showing a particular sequence of interactions among a group of components.

3. MODULE ARCHITECTURE VIEW

In the module architecture view, subsystems are decomposed into modules, and modules are assigned to layers in accordance with their use-dependencies (Table 2). There is no configuration for the module view because it defines the modules and their inherent relations to each other, but not how they will be combined into a particular product.

Table 2. Elements of the module architecture view

Elements	*Behavior*	*Relations*
module	interface protocol	module implements
subsystem		conceptual component
layer		subsystem decomposition
		module use-dependency

Table 3 shows how the image pipeline's conceptual elements are mapped to module elements. Notice that ports, connectors, and components are sometimes combined into one module. This information could also be shown in a UML class diagram, with the mapping between conceptual and module elements shown as an explicit association.

Table 3. Mapping between conceptual and module architecture views

Conceptual element	*Subsystem or Module*
ImagePipeline	SPipeline
acqControl, pipelineControl	MPipelineAPI
PipelineMgr, ImagePipe, Client/Server	MPipelineControl, MImageBuffer
stageControl, imageIn, imageOut	MImageMgrAPI
Framer	MFramer
Imager	MImager

The SPipeline subsystem is decomposed into the six modules shown in Figure 3. This decomposition is dictated by the modules' correspondence to the conceptual elements, and their decomposition. Again we use nesting to

show the decomposition, and we use stereotypes for each different type of element.

We do not use the UML "component" notation for a module, because in the module view the modules are abstract, not the physical modules of source code.

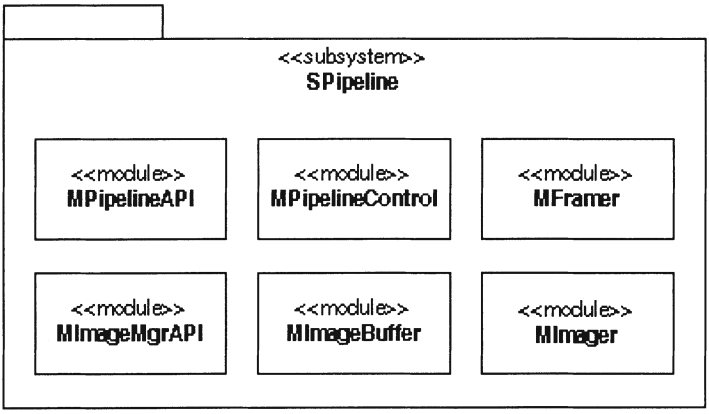

Figure 3. Decomposition of SPipeline

The use-dependencies among the pipeline modules are also derived from the conceptual elements' associations. These are shown in Figure 4. The MClient and MDataMgrAPI are not part of the SPipeline subsystem, but we included them in order to show all use-dependencies of the SPipeline subsystem. We use the UML "lollipop" notation to show the interface(s) of each module, and to make it clear that the modules are dependent on the interface of another module, not the module itself.

Figure 4 also shows some of the layers of the system. These are based on the use-dependencies among modules and subsystems, so we often show use-dependencies between and within layers in the same diagram, as we did here.

For the interface definition, we use a simple list of the interface methods. This information could be put inside the class definition in a UML diagram. We generally prefer to list it separately, using the class diagrams to focus on the relations among modules rather than a complete description of the modules. In the module view, we represent modules with a stereotyped class, and subsystems and layers with stereotyped packages. Decomposition is shown by nesting (association), and the use-dependency is a UML dependency.

We use:

- tables for describing the mapping between the conceptual and module views.

- UML Package Diagrams for showing subsystem decomposition dependencies.
- UML Class Diagrams for showing use-dependencies between modules.
- UML Package Diagrams for showing use-dependencies among layers and the assignment of modules to layers.

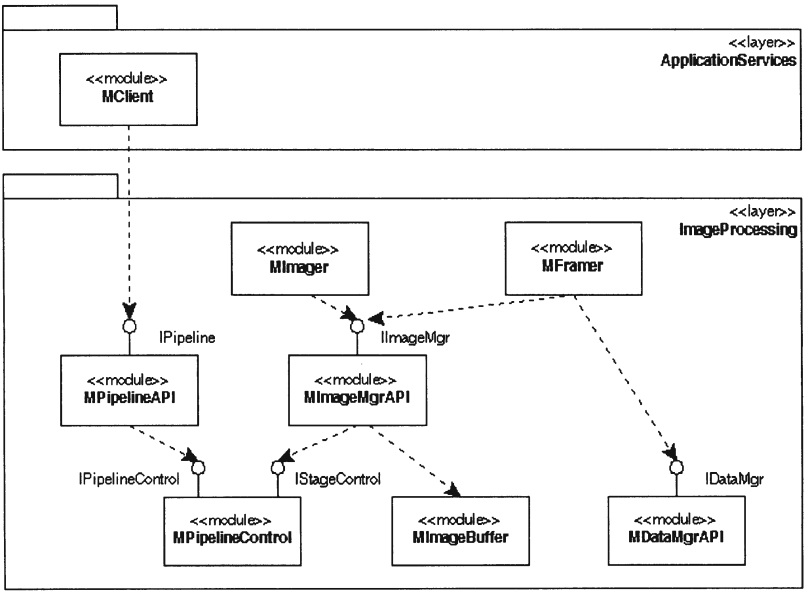

Figure 4. Use-dependencies of SPipeline

4. EXECUTION ARCHITECTURE VIEW

The execution architecture view describes how modules will be combined into a particular product by showing how they are assigned to run-time images. Here the run-time images and communication paths are bound together to form a configuration. Table 4 lists the elements, behavior, and relations of the execution view.

Table 4. Elements of the execution architecture view

Elements	*Behavior*	*Relations*
run-time image communication path	communication protocol	run-time image contains module binding (for configuration)

The execution configuration of the Image pipeline in Figure 5 indicates that there is always just one EClient process, but multiple pipelines can exist at one time. A pipeline has one process each for EPipelineMgr, EImageBuffer, and EFramer, and one process each for additional pipeline stages.

We again use a stereotype of the UML Class for run-time images. They are stereotyped with the name of the platform element, in this case <<process>> or <<shared data>>. We originally used the UML "active object" notation for a process, but now prefer to use a stereotyped class. One reason is that we often want to use classes rather than objects in a configuration diagram. A second reason is that active objects have a thread of control, whereas passive objects run only when invoked (UML, 1997). This distinction was not what we wanted to describe; we wanted to characterize the run-time image by its platform element (e.g. process, thread, dynamic link library, etc.) rather than convey control flow information about the elements.

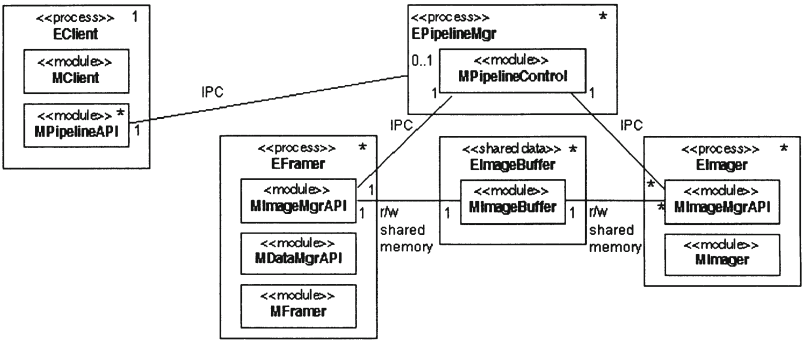

Figure 5. Execution configuration of the image pipeline

This diagram uses nesting to show the modules associated with each run-time image. The modules have a multiplicity that is assumed to be one if none is explicitly shown. In the configuration in Figure 5, there are multiple modules MImageMgrAPI, but at most one per process, and only in the EFramer and EImager processes. There are also multiple modules MPipelineAPI in the configuration, but all of these reside in process EClient.

The run-time images also have multiplicity, as do communication paths, which are labeled to show the communication mechanisms. This has the same implications as for the conceptual configuration, namely that with multiplicities on the run-time images, communication paths, and modules we can show all allowable configurations in a single diagram.

UML class diagrams cannot show dynamic behavior, so we use different diagrams to show the dynamic aspects of configurations. Figure 5 shows the configuration of the pipeline during an imaging procedure. The processes that implement the pipeline are created dynamically when the imaging procedure is requested, and are destroyed after the procedure has completed. A UML sequence diagram shows how the pipeline is created at the start of a procedure (Figure 6).

For the execution view, we represent the run-time images as stereotyped classes, and the communication paths as associations. Module containment is shown by nesting (association). We use:

- UML Class Diagrams for showing the static configuration.
- UML Sequence Diagrams for showing the dynamic behavior of a configuration, or the transition between configurations.
- UML State Diagrams or Sequence Diagrams for showing the protocol of a communication path.

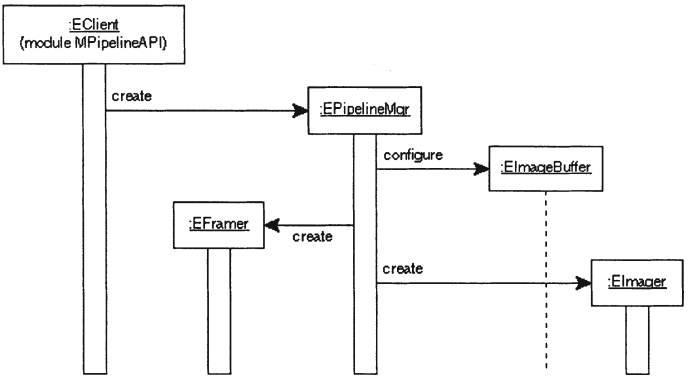

Figure 6. Image pipeline creation

5. CODE ARCHITECTURE VIEW

The code architecture view contains files and directories, and like the module view, does not have a configuration. The relations defined in the code view apply across all products, not just to a particular product. The code view elements and their relations are listed in Table 5. Modules and interfaces from the module view are partitioned into source files in a particular programming language.

Table 6 shows this mapping for the MPipelineControl module and its interfaces: the public interfaces are each mapped to a file, and we have created an additional file for the private interface to the module.

Table 5. Elements of code architecture view

Elements	*Relations*
source	source implements module
intermediate	source includes source
executable	intermediate compiled from run-time image
directory	executable implements run-time image
	executable linked from intermediate

Table 6. Source files for module MPipelineControl

Module or Interface	*Source File*
MPipelineControl	CPipelineControl.CPP, CPipelineControlPvt.H
IPipelineControl	CPipelineControl.H
IStageControl	CStageControl.H

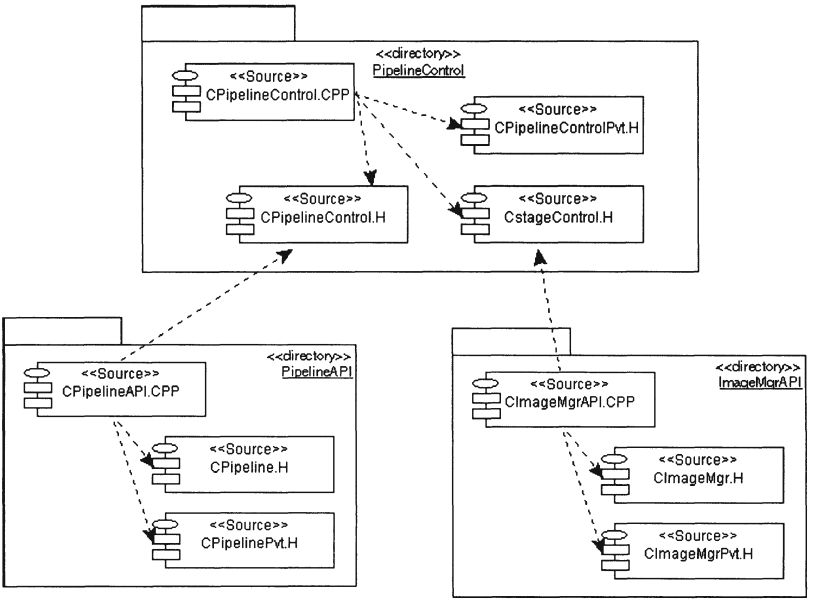

Figure 7. Include dependencies among source files

The source files are organized into directories, as shown in Figure 7. We use the UML "component" notation to represent the files, and the package notation for directories. Both files and directories have stereotypes to clarify their meaning. In UML, the component symbol is used for "source code components, binary code components, and executable components" (UML, 1997). We believe the intention of this symbol is closest to our notion of a file (whether source, intermediate, or executable).

Figure 7 also shows the include dependencies for the PipelineControl source files. We use the UML dependency notation for these relationships, with the stereotype <<include>> if the diagram contains more than one type of dependency. Source files can also have a "generate" dependency, for example when a preprocessor uses one source file to generate another.

The run-time images from the execution view also have a relationship to elements in the code view, in this case to executable files. Table 7 shows how two of the run-time images in the image pipeline are mapped to executable files. Here the mapping is one-to-one, but if the run-time image contained dynamic link libraries, each of these libraries would be in a separate executable file.

Table 7. Mapping between run-time image and executable file

Run-time Image	*Executable File*
EPipelineMgr	EPipelineMgr.exe
EFramer	EFramer.exe

The executable files are also organized into directories (Figure 8). The relationship between executable files and source files is through intermediate files. An executable file has link dependencies to the object files it links in, and an object file has compile dependencies to the source files from which it is compiled. These dependencies are also shown in Figure 8.

For the code view, we represent the source, object, and executable files as stereotyped classes, and the directories as stereotyped packages. The include, compile, and link relationships are shown as stereotyped dependencies. We use:

- Tables to describe the mapping between elements in the module and execution views and elements in the code view.
- UML Component Diagrams for showing the dependencies among source, intermediate, and executable files.

6. DISCUSSION

Table 8 summarizes the elements of our four architecture views and their corresponding UML Metamodel Classes and stereotype names, if any. For relations among the architecture description elements, we use UML associations and dependencies. We generally create a separate diagram for each kind of relation, but sometimes we combine them (e.g. the execution configuration diagram).

We use UML Class/Object, Package, and Component Diagrams for the elements and their relations, sometimes including the interfaces and

attributes in these diagrams. Sequence Diagrams or State Diagrams are used to describe behavior.

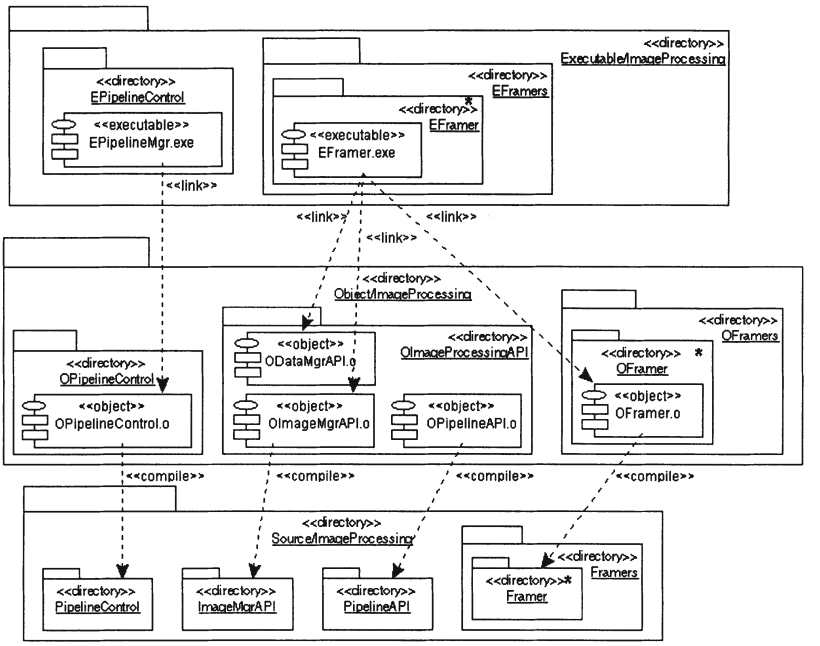

Figure 8. Dependencies among source, object, and executable files

The configuration diagrams in the conceptual and execution views are UML Class/Object Diagrams, but we added some conventions to help define the semantics and improve the readability of the diagrams.

One convention is to use nesting to indicate decomposition. This makes the structure easier to see, although it can make layout difficult for complex structures. With this convention we cannot show recursive or indefinite nesting, which could be easily described in a diagram that depicts decomposition as a labeled association (a line) between two objects.

A semantic convention we use is that a configuration diagram describes the set of possible configurations at a single point in time. Systems generally have defined modes, e.g. start-up, shut-down, operational, diagnosis, recovery, etc. Each of these modes can have a different configuration, so should have a different diagram. In some modes (in our example, the operational mode) the configuration changes over time (in our case, pipelines are created and destroyed with each acquisition procedure). The dynamic behavior should be described separately. A sequence diagram works well to describe start-up and shut-down behavior.

Table 8. Summary of architecture description elements

Element	*UML Metamodel Class*	*Stereotype Name*
component	Class	<<component>>
port	Class	<<port>>
connector	Class	<<connector>>
role	label on association	
port or role protocol	Class	<<protocol>>
module	Class	<<module>>
subsystem	Package	<<subsystem>>
layer	Package	<<layer>>
run-time image	Class	<<process>>, <<shared data>>, <<thread>>, etc.
communication path	association	
source	Component	<<source>>
intermediate	Component	<<object>>
executable	Component	<<executable>>
directory	Package	<<directory>>

An important concern we have about using UML to describe software architecture is that the same notation can have a wide range of semantics. We use the same basic diagram, the UML Class/Object diagram to show most of the aspects of the architecture. We use stereotypes and special symbols to minimize the confusion between different views.

The more traditional use of UML is for the design of implementation classes for a system. We are also concerned that by using the same notation to describe the software architecture, we run the risk of further blurring the distinction between the architecture and the implementation. This is another reason to consistently use particular conventions, stereotypes, and special symbols for these architecture diagrams.

In summary, we found UML deficient in describing:

- correspondences: A graphical notation is too cumbersome for straightforward mappings such as the correspondence between elements in different views. This information is more efficiently described in a table (e.g.Table 3).
- protocols: The ability to show peer-to-peer communication is missing from UML. We used ROOM to describe protocols (e.g. Figure 2).
- ports on components: We used nesting to show the relationship between ports and components, but this is visually somewhat misleading. We would prefer a notation more similar to the lollipop notation for the interfaces of a module.
- dynamic aspects of the structure
- a general sequence of activities

UML worked well for describing:

- the static structure of the architecture
- variability: e.g. the conceptual configuration in Figure 1 describes the structure of a set of pipelines.
- a particular sequence of activities: e.g. the start-up behavior of an Image Pipeline (Figure 6).

REFERENCES

Bass, L., Clements, P., and Kazman, R. (1998) Software Architecture in Practice. Addison-Wesley, Massachusetts.

Eriksson, H., and Penker, M. (1998) UML Toolkit. John Wiley and Sons, London.

Fowler, M., with Scott, K. (1997) UML Distilled. Applying the Standard Object Modeling Language. Addison-Wesley, Massachusetts.

Hofmeister, C., Nord, R., Soni, D. (to appear) Applied Software Architecture. Addison-Wesley, Massachusetts.

Kramer, J., and Magee, J. (1990) The Evolving Philosophers Problem: Dynamic Change Management. *ACM Transactions on Software Engineering*, **16(11)**, 1293-1306.

Kruchten, P. (1995) The 4+1 View Model of Architecture, *IEEE Software*, **12(6)**.

Prieto-Diaz, R., and Neighbors, J.M. (1986) Module Interconnection Languages. *The Journal of Systems and Software*, **6(4)**, 307-334.

Purtilo, J.M. (1994) The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems,* **16(1)**, 151-174.

Selic, B., Gullekson, G., and Ward, P.T. (1994) Real-Time Object-Oriented Modeling. John Wiley and Sons, New York.

Selic, B., and Rumbaugh, J. (1998) Using UML for Modeling Complex Real-Time Systems. http://www.objectime.com/uml/uml.html.

Shaw, M., and Garlan, D. (1996) Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall.

Soni, D., Nord, R.L., and Hofmeister, C. (1995) Software Architecture in Industrial Applications, in Proceedings of the 17th International Conference on Software Engineering, Seattle, WA.

UML (1997) UML Notation Guide, Version 1.1. http://www.rational.com/uml.