Chapter 11

Designing the System Architecture

- The Need for Architecture
- The Architecture Team
- The "4+1" View of Architecture
- The Logical View
- The Component View
- The Process View
- The Deployment View
- The Use Case View
- Summary
THE NEED FOR ARCHITECTURE

OVER THE YEARS, I have heard many definitions of software architecture that range from "software architecture is what software architects do" to "software architecture is politics." I have come to the conclusion that software architecture is very difficult to define. It is a range of artifacts that is used to specify the strategic decisions about the structure and behavior of the system, the collaborations among the system elements, and the physical deployment of the system.

"Establishing a sound architectural foundation is absolutely essential to the success of an object-oriented project. Some teams try to ignore this phase, either because they are in such a rush to get a product out quickly they feel they don't have time to architect, or because they don't believe that architecting provides them any real value. Either way, the resulting head-long rush to code is always disastrous: fail to carry out this step properly, and your project will likely experience software meltdown."\(^1\)

Architecture development is a very complicated issue. The architecture of the system is developed iteratively in the elaboration phase of development: "The architecture of a proposed system does not appear in a flash. It takes exploration of the use cases, a proof-of-concept prototype, an architectural baseline, and other efforts during the Inception and Elaboration phases.\(^2\) Executable prototypes of the architecture are built to verify that the design decisions made are correct. "Building something executable is absolutely essential, because it forces the development team to validate their design assumptions in the harsh light of reality.\(^3\)"

---


3 Ibid.
THE ARCHITECTURE TEAM

Each project should have a chief architect who may be assisted by a small team of people. "The main activities of the architect include the definition of the architecture of the software, the maintenance of the architectural integrity of the software, the assessment of the technical risks of the project, the definition of the order and content of the successive iterations along with the planning of each iteration, providing consulting to various design, implementation, integration, and quality assurance teams and assisting in providing future market directions."  

THE "4 + 1" VIEW OF ARCHITECTURE

Software architecture is not a one-dimensional thing—it is made up of concurrent multiple views. Figure 11-1 shows the different views of software architecture. The rest of this chapter addresses the elements in each view of the architecture, along with the UML notation used to represent the architectural decisions made for the system.

THE LOGICAL VIEW

This view of architecture addresses the functional requirements of the system—what the system should provide in terms of services to its users. The logical architecture is captured in class diagrams that contain the classes and relationships that represent the key abstractions of the system under development. Most of the UML notation that has been addressed so far is contained within this view of architecture (e.g., classes, associations, aggregations, generalization, and packages).

---


5 Ibid.
This view of architecture is addressed early in the elaboration phase with the creation of classes and packages that represent the major abstractions of the domain. As time moves on, more classes and packages are added to the model to reflect the decisions made concerning the key mechanisms of the system. A key mechanism is a decision regarding common standards, policies, and practices. The selection of the key mechanisms for a system is often referred to as tactical design. "Poor tactical design can ruin even the most profound architecture, and so the team must mitigate this risk by explicitly identifying the project’s key policies." Some common key mechanisms involve the selection of an implementation language, persistent data storage, the look and feel of the user interface, error handling, communication mechanisms, object distribution and migration, and networking.

Today, many patterns exist that may be used to implement the key mechanism decisions made for your system. I strongly recommend looking into patterns before you try to “roll your own.”

---

Additionally, the concepts of cohesion, closure, and reuse will affect the choices that you make. Robert Martin discusses some of the ramifications of the choice of packages for your system in his book, *Designing Object-Oriented C++ Applications Using the Booch Method*. Although this book uses the Booch notation and process, it is still applicable to the Rational Objectory Process and the UML. The bottom line is: The UML may be used to communicate the strategic decisions made for your system by adding packages and classes to the model to communicate, implement, and document these decisions.

**Sample Key Mechanisms for the ESU Course Registration Problem**

Since most of the development team had prior experience using the C++ language, and because this system eventually will be expanded to include other university functionality, C++ is the language of choice by the architecture team. The architecture team also decided that a particular set of graphical user interface (GUI) controls should be used to control the look and feel of the user interface, and therefore, a package called GUI Controls was added to the model. The database persistence strategy chosen by the architecture team is the use of a corresponding database class (shadow class) for each persistent class in the system. Although other strategies that mainly involve the use of inheritance could have been chosen by the team, this strategy was chosen due to the fact that expertise in implementing this method of persistence already existed and the team felt that it had the least amount of risk. A database package containing the shadow classes was added to the model at this time. Additionally, it was decided to make use of the C++ features of catch and throw for exceptions. Rather than have each class be responsible for knowing how to catch and throw exceptions, a package called Error Handling was added to the model. Finally, a set of commercial foundation classes was chosen for this system. The packages representing the key mechanism decisions made for the course registration system are shown in Figure 11-2.

Since the Error Handling package and the Foundations package are used by every other package in the system, they are global packages.
THE COMPONENT VIEW

1. Right-click to select the package on a class diagram.
2. Select the Specification menu choice.
3. Select the Detail tab.
4. Click to select the global checkbox.
5. Click the OK button to close the specification.

THE COMPONENT VIEW

THIS VIEW OF architecture concerns itself with the actual software module organization within the development environment. The component view of architecture takes into account derived requirements related to ease of development, software management, reuse,
and constraints imposed by programming languages and development tools. The modeling elements in the component view of architecture are packages and components along with their connections.

A package in this view of architecture represents a physical partitioning of the system. Component view packages are often called subsystems. The packages are organized in a hierarchy of layers where each layer has a well-defined interface. The fact that an object-oriented system tends to be a layered system should not bring any surprises. This is due to the definition of an object—it should do one thing, and do it well! A drawing showing some typical layers of a system can be found in Figure 11-3.

<table>
<thead>
<tr>
<th>User Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Specific Packages</td>
</tr>
<tr>
<td>Reusable Business Packages</td>
</tr>
<tr>
<td>Key Mechanisms</td>
</tr>
<tr>
<td>Hardware and Operating System Packages</td>
</tr>
</tbody>
</table>

*Figure 11-3  System Layers*

The UML notation for a package in the component view is the same as a package in the logical view—a notched folder, as found in Figure 11-4.
CREATING COMPONENT VIEW PACKAGES IN RATIONAL ROSE

1. Right-click to select the Component View package on the browser and make the shortcut menu visible.
2. Select the New:Package menu choice. This will add an item called NewPackage to the browser.
3. With the NewPackage still selected, enter the name of the package.

The main component diagram typically is a view of the packages (subsystems) defined for the system.

THE MAIN COMPONENT DIAGRAM IN RATIONAL ROSE

1. Double-click on the Main Diagram under the Component View package on the browser to open the diagram.
2. Click to select a package and drag the package onto the diagram.
3. Repeat step 2 for each additional package.
4. Dependency relationships are added by selecting the dependency icon from the toolbar, clicking on the package representing the client, and dragging the arrow to the package representing the supplier.

The main component diagram for the ESU Course Registration problem is shown in Figure 11-5.

Information in the logical view of the model is connected to information in the component view of the model by mapping logical view packages to component view packages. In general, a logical view package corresponds directly to a physical view package. However, there are times where a one-to-one mapping is not possible. Some reasons for this are: a logical package may be divided for implementation purposes (possibly developed by different contractors); logical packages may be merged to keep closely communicating objects together; and, finally, physical view packages may be added to implement low-level functionality not represented during analysis (i.e., communication for distributed systems).
The mapping of logical packages to component packages for the ESU Course Registration problem is straightforward, and is shown in Table 11-1.

<table>
<thead>
<tr>
<th>Component Package</th>
<th>Logical Package(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interfaces</td>
<td>Interfaces, GUI Controls</td>
</tr>
<tr>
<td>University</td>
<td>UniversityArtifacts, People</td>
</tr>
<tr>
<td>Database</td>
<td>Database</td>
</tr>
<tr>
<td>Foundations</td>
<td>Foundations</td>
</tr>
<tr>
<td>Error Handling</td>
<td>Error Handling</td>
</tr>
</tbody>
</table>

*Table 11-1  Mapping of Logical Packages to Component Packages*
MAPPING LOGICAL PACKAGES TO
COMPONENT PACKAGES IN RATIONAL ROSE

1. Right-click to select the component package on the browser and make the shortcut menu visible.
2. Select the Specification menu choice.
3. Select the Packages tab.
4. Right-click to select the logical package and make the shortcut menu visible.
5. Select the Assign menu choice.
6. Click the OK button to close the Specification.

The mapping of logical packages to component packages for the ESU Course Registration problem is shown in Figure 11-6.

Source Code Components
In the component view of the model, a component represents a software file that is contained by a package (subsystem). The type of file is language dependent (e.g., in C++, software components represent .h and .cpp files, in Java they represent .java files, and in PowerBuilder, a software component is a .pbl). Classes in the logical
view are mapped to components in the component view. In C++, the mapping is typically one-to-one; that is, one class maps to one component. However, there are times that more than one class will be mapped to a component. This is usually done when there is very tight coupling between the classes. For example, a container and its iterator are contained within one .h and one .cpp file. In this case, the container and the iterator classes would be mapped to one component. I have also seen classes that represent a pattern of collaboration mapped to one physical file. The UML notation for a component is shown in Figure 11-7.

![UML Notation for a Component](image)

**Figure 11-7  UML Notation for a Component**

**Software Components in the ESU Course Registration Problem**
This is a relatively simple system, and the decision was made to provide a one-to-one mapping between classes and components—each class has its own header and .cpp file.

**CREATING COMPONENTS IN RATIONAL ROSE**
1. Open a component diagram.
2. Click to select the Package Specification icon on the toolbar.
3. Click on the diagram to place the component. This will also add the component to the Browser.
4. While the component is still selected, enter the name of the component.
A sample component diagram is shown in Figure 11-8.

![Software Components](image)

**Figure 11-8 Software Components**

**MAPPING CLASSES TO COMPONENTS IN RATIONAL ROSE**

1. Right-click to select the component on the browser and make the shortcut menu visible.
2. Select the Specification menu choice.
3. Select the Classes tab.
4. Right-click to select the class and make the shortcut menu visible.
5. Select the Assign menu choice.
6. Click the OK button to close the Specification.

The browser view of the mapping of classes to components for the ESU Course Registration System is shown in Figure 11-9.
THE PROCESS VIEW

This view of architecture focuses on the run-time implementation structure of the system. The process view of architecture takes into account requirements such as performance, reliability, scalability, integrity, system management, and synchronization. Components are also used in this view of architecture. Component diagrams are created to view the run-time and executable components created for the system. Components are related via dependency relationships. Run-time components show the mapping of classes to run-time libraries such as Java applets, Active X components, and dynamic libraries. Executable components show the interfaces and calling dependencies among executables.

In Rational Rose, the information for the process view of architecture is created as diagrams in the Component View of the tool. Components (either run-time or executable) are added to the package representing their own subsystem. Diagrams are created to show dependencies between the different types of components in the system.
For the ESU Course Registration System, the architecture team decided that there would be two DLLs—one containing course and course-offering information, and one containing database information. This allocation was chosen since it was felt that the course structure and the choice of database strategy was subject to change. By making them libraries, only the libraries would have to be replaced to implement future changes. There are three executables for the system—one for the Registrar to create and maintain the system, one for the Student to access the system, and one for the Professor to access the system. There is no communication between the executables. The component diagram for the Professor executable (professor.exe) is shown in Figure 11-10.

![Component Diagram: Component View / Professor Components](image)

Figure 11-10  Professor Executable

THE DEPLOYMENT VIEW

This view of architecture involves mapping software to processing nodes—it shows the configuration of run-time processing elements and the software processes living in them. The deployment view
takes into account requirements such as system availability, reliability, performance, and scalability. Deployment diagrams are created to show the different nodes along with their connections in the system. The deployment diagram visualizes the distribution of components across the enterprise. Run-time processing elements are represented as nodes. Nodes are connected by associations indicating communication paths between them. Software processes are illustrated as text attached to a node or group of nodes.

This diagram allows the architecture team to understand the system topology and aids in mapping subsystems to executable processes. Issues such as processor architecture, speed, and capacity, along with interprocess communication bandwidth/capacity, physical location of the hardware, and distributed processing techniques all come into play.

**Deployment Diagram for the ESU Course Registration System**

After studying the component packages [subsystems] defined for the problem, examining existing hardware, and estimating the load on the system during the course registration period, the architecture team decided that they will need five processors for the system—one to handle the professor executable, one for the database, and three for student registration.

**CREATING THE DEPLOYMENT DIAGRAM IN RATIONAL ROSE**

1. Rose automatically creates the deployment diagram. To open the diagram, double-click on the Deployment Diagram on the browser.
2. To create a node, click to select the Processor icon, and click on the diagram to place the node.
3. With the node still selected, enter the name of the node.
4. To create a connection between nodes, click to select the connection icon from the toolbar, click on one node on the deployment diagram, and drag the connection to the other node.
The deployment diagram for the ESU Course Registration problem is shown in Figure 11-11.

**Figure 11-11 Deployment Diagram**

**THE USE CASE VIEW**

THIS VIEW OF architecture demonstrates and validates the logical, process, component, and deployment views. Sequence diagrams and collaboration diagrams are created to show how the various design elements interact to produce the desired behavior.

**SUMMARY**

SOFTWARE ARCHITECTURE is not one-dimensional—it is made up of concurrent multiple views—logical, component, process, and deployment. Scenarios are developed to validate the different views. Good architectures are constructed in well-defined layers
Chapter 11 / Designing the System Architecture

of abstraction where there is a clear separation between the interface and implementation of each layer. Key mechanisms focus on decisions regarding common standards, policies, and practices. Packages are created to show the architectural layout of the system. Architecture also addresses the physical layout of the system. Component diagrams are created to show subsystems that are the physical implementations of the logical packages. Deployment diagrams are created to show the hardware configuration that is used for the system under development.