Unit- VI

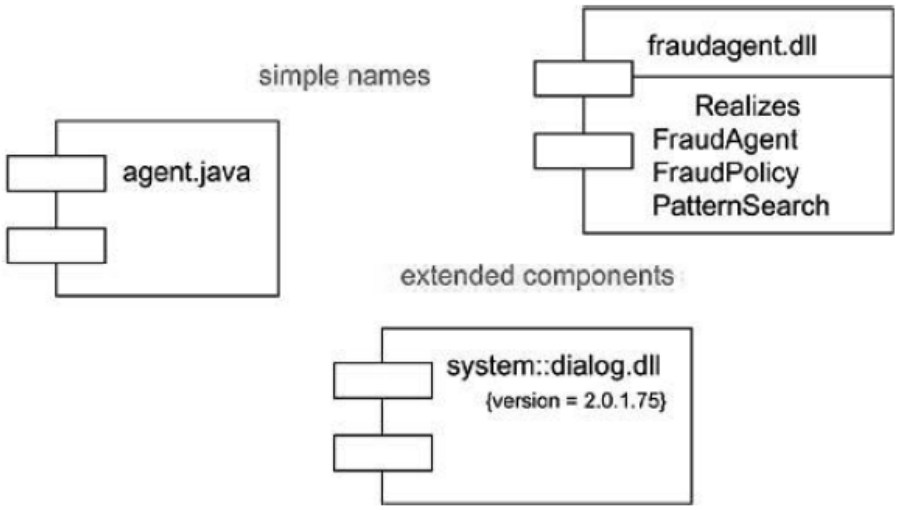
Chapter 1 Component

Introduction

A component is a physical replaceable part of a system that complies with and provides the realization of a set of interfaces. We use components to model the physical things that may reside on a node, such as executables, libraries, tables, files and documents.

A component typically represents the physical packaging of otherwise logical elements such as classes, interfaces and collaborations. We do logical modeling to visualize, specify, and document our decisions about the vocabulary of our domain and the structural and behavioral way those things collaborate.

We do physical modeling to construct the executable system. Object libraries, executables, COM+ components and Enterprise Java Beans are all examples of components.



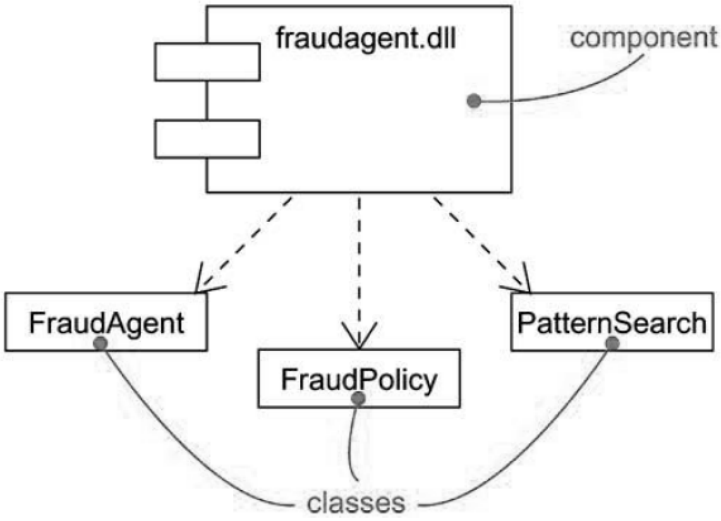
Components and Classes

In many ways, components are like classes. Both have names, both may realize a set of interfaces, both may participate in dependency, generalization and association relationships, both may be nested, and both may have instances. However, there are some significant differences between components and classes:

- Classes represent logical abstractions, components represent physical things that live in the world of bits. In short, components may live on nodes, classes may not.

- Components represent the physical packaging of otherwise logical components and are at a different level of abstraction.
- Classes may have attributes and operations directly. In general, components only have operations that are reachable only through their interfaces.

The relationship between a component and the classes it implements can be shown explicitly by using a dependency relationship as shown below:



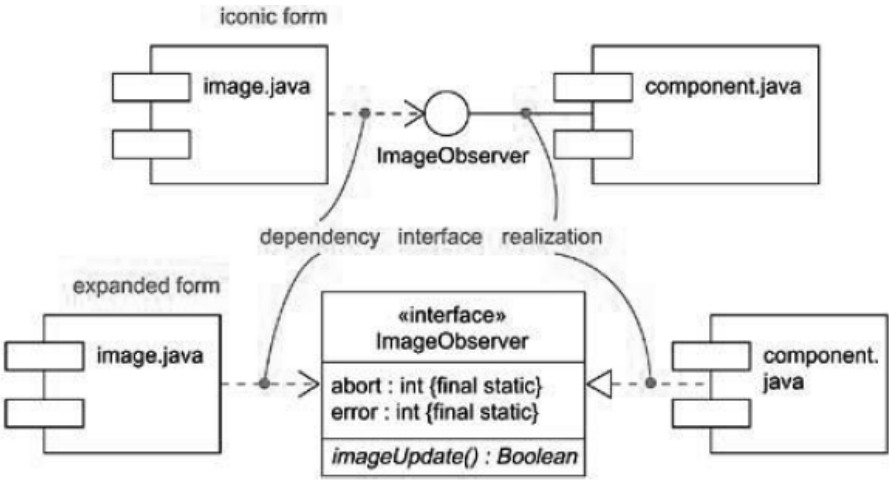
Components and Interfaces

An interface is a collection of operations that are used to specify a service of a class or a component. We can show the relationship between a component and its interfaces in one of the two ways.

The first style renders the interfaces in its elided, iconic form. The component that realizes the interfaces is connected to the interface using an elided realization relationship.

The second style renders the interface in its expanded form, perhaps revealing its operations. The component that realizes the interface is connected to the interface using a full realization relationship.

In both cases, the component that accesses the services of the other component through the interfaces is connected to the interface using a dependency relationship.



Binary Replaceability

The basic intent of every component-based operating system facility is to permit the assembly of systems from binary replaceable parts. This means that we can create a system out of components and then evolve the system by adding new components and replacing the old ones, without rebuilding the system.

First, a component is physical. It lives in the world of bits, not concepts.

Second, a component is replaceable. A component is substitutable means it is possible to replace a component with another that conforms to the sane interfaces.

Third, a component is part of a system. A component rarely stands alone. Rather, a given component collaborates with other components and in so doing exists in the architectural or technology context in which it is intended to be used.

Fourth, a component conforms to and provides the realization of a set of interfaces.

Kinds of Components

Three kinds of components may be distinguished.

First, there are deployment components. These are the components necessary and sufficient to form an executable system, such as dynamic libraries (DLLs) and executables (EXEs).

Second, there are work product components. These components are generally the residue of the development process, consisting of things such as the source code files and data files from which deployment components are created.

Third are execution components. These components are created as a consequence of an executing system, such as COM+ object, which is instantiated from a DLL.

Standard Elements

All the UML's extensibility mechanisms apply to components. Most often, we'll use tagged values to extend the component properties and stereotypes to specify new kind of components.

The UML defines five standard stereotypes that apply to components:

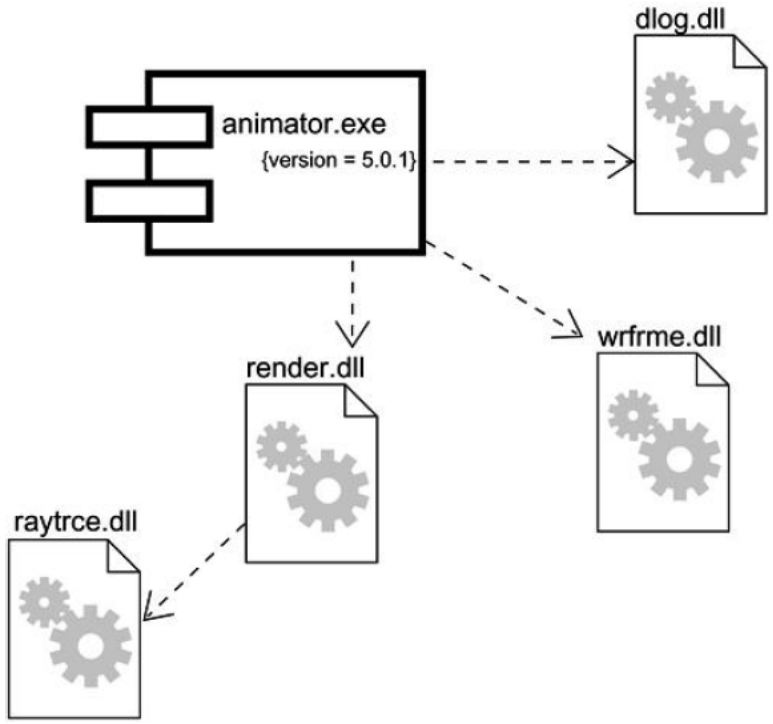
executable	Specifies a component that may be executed on a node
library	Specifies a static or dynamic object library
table	Specifies a component that represents a database table
file	Specifies a component that represents a document containing code or data
document	Specifies a component that represents a document

Common Modeling Techniques

Modeling Executables and Libraries

To model executables and libraries:

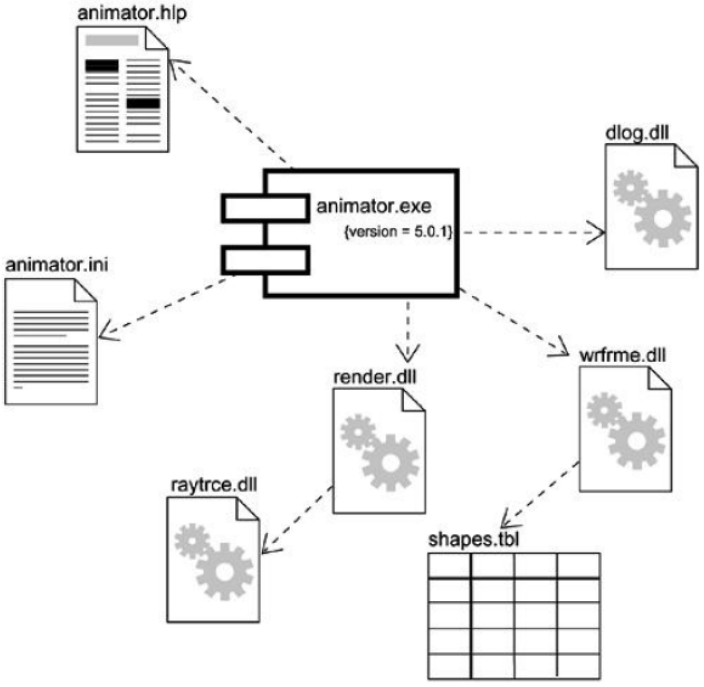
- Identify the partitioning of the physical system. Consider the impact of the technical, configuration management and reuse issues.
- Model any executables and libraries as components, using the appropriate standard elements.
- Model the significant interfaces that some components use and others realize.
- As necessary to communicate your intent, model the relationships among these executables, libraries and interfaces.



Modeling Tables, Files and Documents

To model tables, files and documents:

- Identify the components that are part of the physical implementation of your system.
- Model these things as components.
- As necessary to communicate your intent, model the relationships among these components and other executables, libraries and interfaces in the system**.**

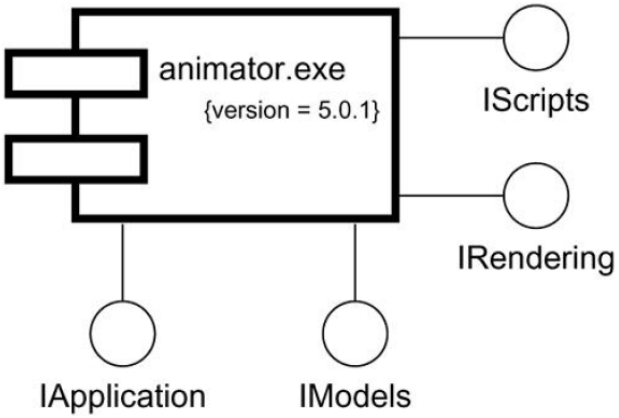


6-tables-files-and-documents

Modeling an API

To model an API:

- Identify the programmatic seems in the system and model each seem as an interface.
- Expose only those properties of the interface that are important to visualize the given context. Otherwise, hide these properties, keeping them in the interface's specification for reference, as necessary.
- Model the realization of each API only as it is important to show the configuration of a specific implementation.

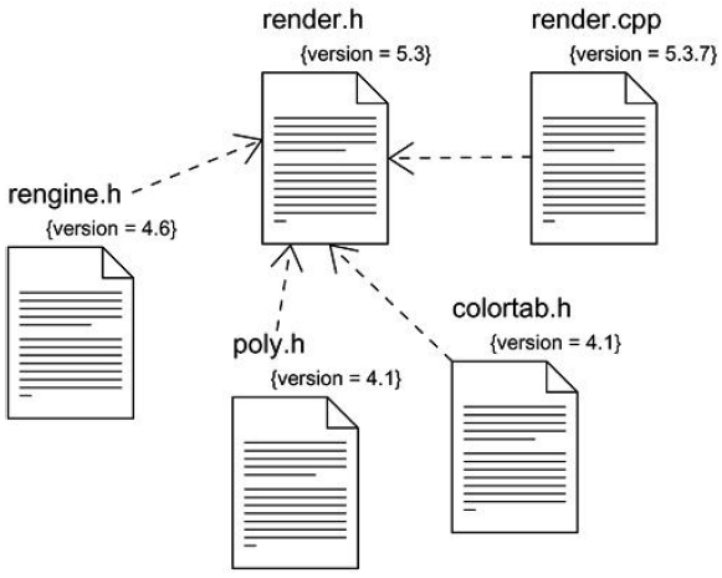


7-modeling-an-API

Modeling Source Code

To model source code:

- Depending on the constraints imposed by your development tools, model the files used to store the details of all your logical elements, along with their compilation dependencies.
- Use tagged values if you want to use configuration management and version control tools.
- As far as possible, let your development tools manage the relationships among these files, and use the UML only to visualize and document these relationships**.**

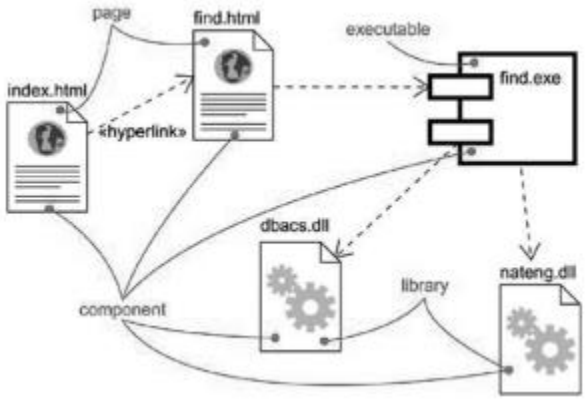


8-modeling-source-code

Chapter 2 Component Diagrams

Introduction

Component diagrams are one of the two kinds of diagrams for modeling the physical aspects of object-oriented software systems. A component diagram shows the organization and dependencies among a set of components. We use component diagrams to model the static implementation view of a software system.



Common Properties

A component is just a special kind of a diagram and shares the same common properties as the other diagrams like: a name and graphical contents. What distinguishes a component diagram from the rest of the diagrams is its content.

Common Uses

When modeling the static implementation view of a system, we will typically use component diagrams in one of four ways:

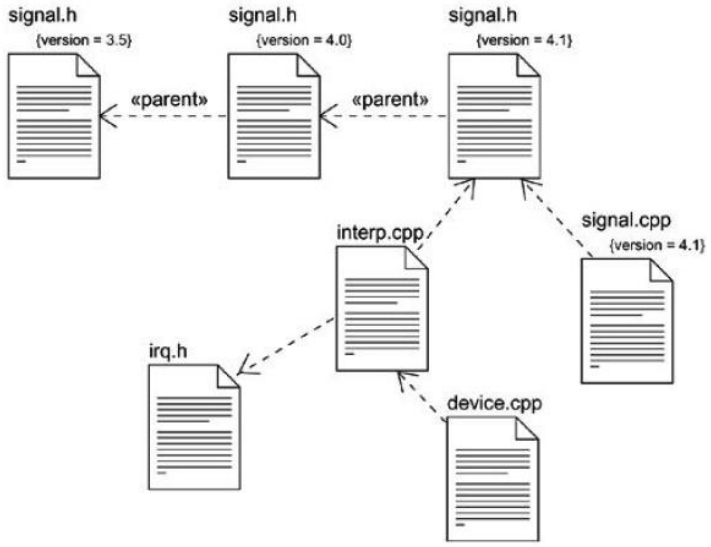
1. To model source code.
2. To model executable releases.
3. To model physical databases.
4. To model adaptable systems.

Common Modeling Techniques

Modeling source code

To model a system's source code:

- Either by forward or reverse engineering, identify the set of source code files of interest and model them as components stereotypes as files.
- For larger systems, use packages to show groups of source code files.
- Consider using tagged values indicating such information as the version number of the source code file, its author, and the date it was last changed.
- Model the compilation dependencies among these files using dependencies.

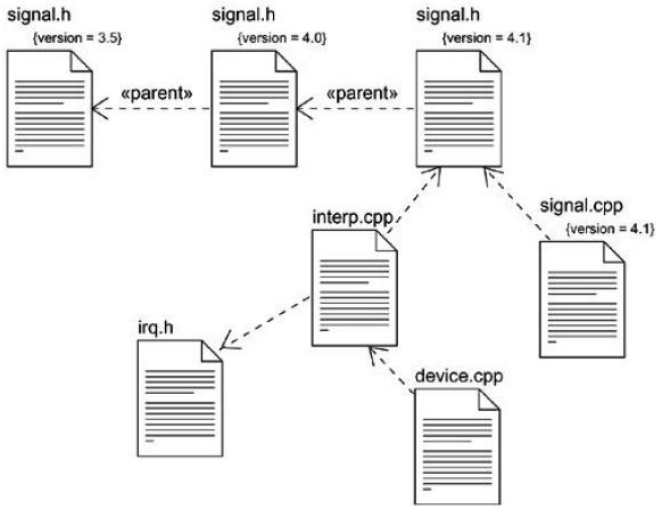


Modeling an executable release

To model an executable release:

- Identify the set of components you'd like to model.

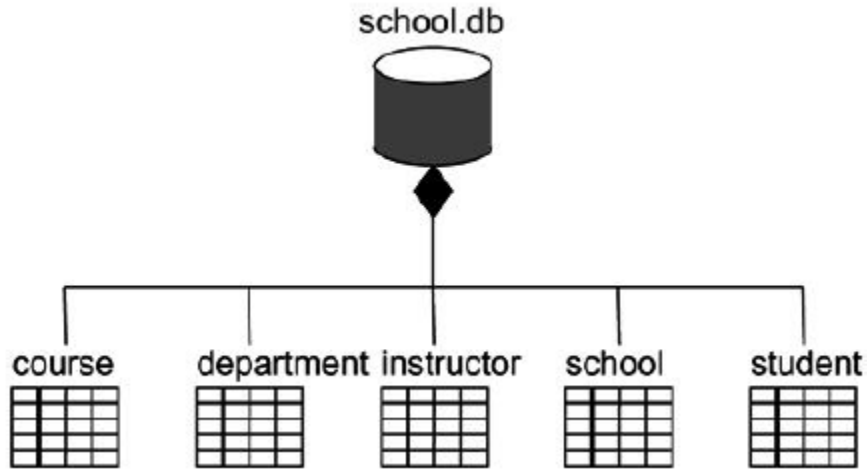
- Consider the stereotype of each component in this set.
- For each component in this set, consider its relationship to its neighbors. Most, often this will involve interfaces that are realized by certain components and then imported by others



Modeling a physical database

To model a physical database:

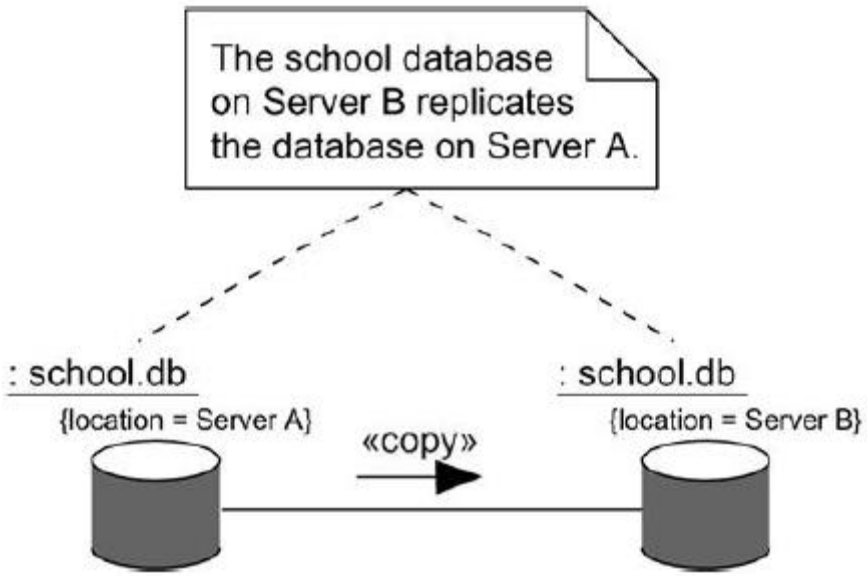
- Identify the classes in your model that represent your logical database schema.
- Select a strategy for mapping these classes to tables. You have to also consider the physical distribution of your databases.
- To visualize, specify, construct and document your mapping, create a component diagram that contains components stereotyped as tables.
- Where possible, use tools to help you transform your logical design into a physical design.



Modeling adaptable systems

To model an adaptable system:

- Consider the physical distribution of the components that may migrate from node to node. We can specify the location of a component instance by marking it with a location tagged value.
- If you want to model the actions that cause a component to migrate, create a corresponding interaction diagram that contains component instances. We can illustrate a change of location by drawing the same instance more than once, but with different values for its location tagged value.



5-modeling-adaptable-systems

Chapter 3 Deployment

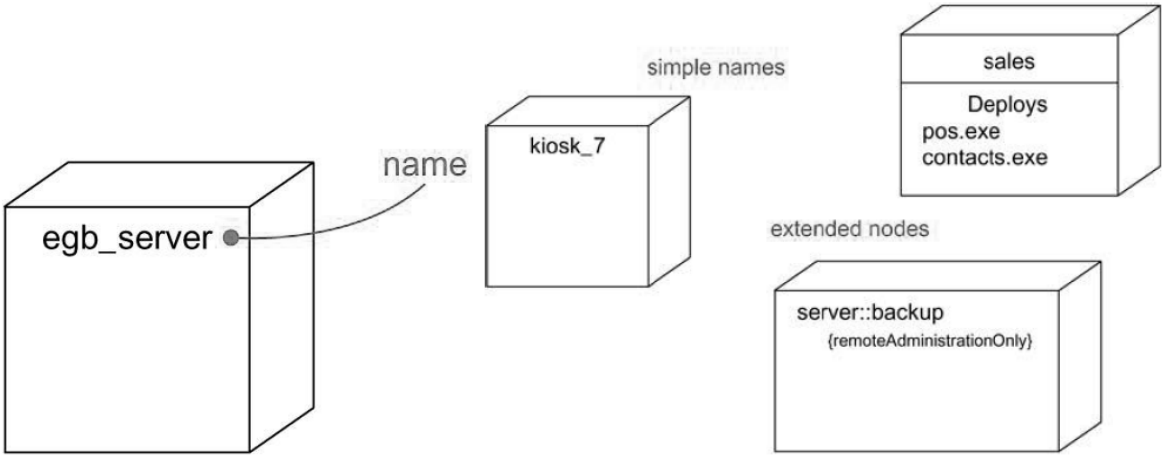
Introduction

A node is a physical element that exists at runtime and represents a computational resource, generally having atleast some memory and, often, processing capability.

We use nodes to model the topology of the hardware on which our system executes. A node typically represents a processor or a device on which components may be deployed.

When we architect a software-intensive system, we have to consider both its logical and physical dimensions. On the logical side, you'll find things such as classes, interfaces, collaborations, interactions and state machines. On the physical side you'll find components and nodes.

In UML, a node is represented as a cube as shown below. Using stereotypes we can tailor this notation to represent specific kinds of processors and devices.

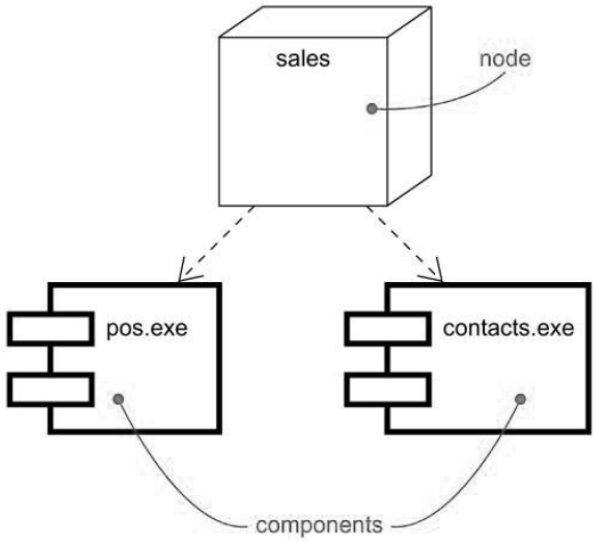


Nodes and Components

In many ways, nodes are like components: Both have names, both may participate in dependency, generalization and association relationships. Both may be nested, both may have instances, both may be participants in interactions. However, there are significant differences between nodes and components:

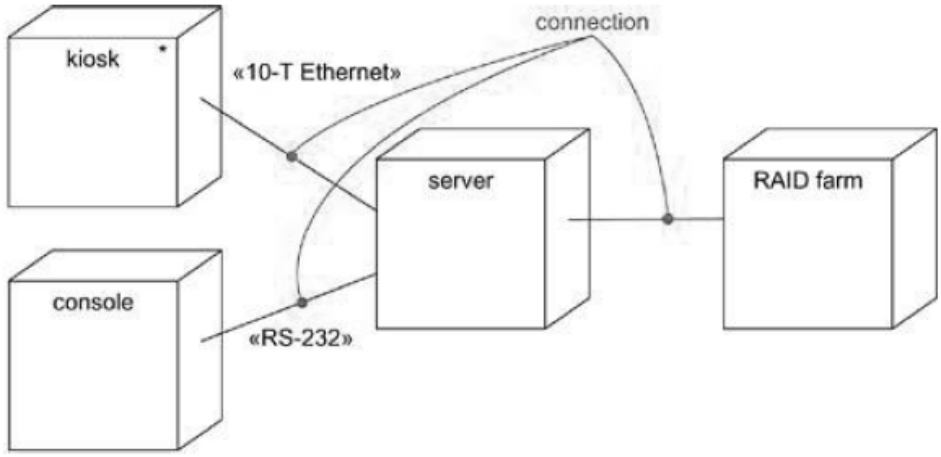
- Components are things that participate in the execution of a system. Nodes are things that execute components.
- Components represent the physical packaging of logical elements, nodes represent the physical deployment of components.
- This first difference is the most important. Simply put, nodes execute components; components are things that are executed by nodes.

- A set of objects or components that are allocated to a node as a group is called a distribution unit.



Connections

The most common kind of relationship we'll use among nodes is an association. In this context, an association represents a physical connection among nodes, such as an Ethernet connection, a serial line, or a shared bus as shown below. We can even use associations to model indirect connections, such as a satellite link between distant processors.

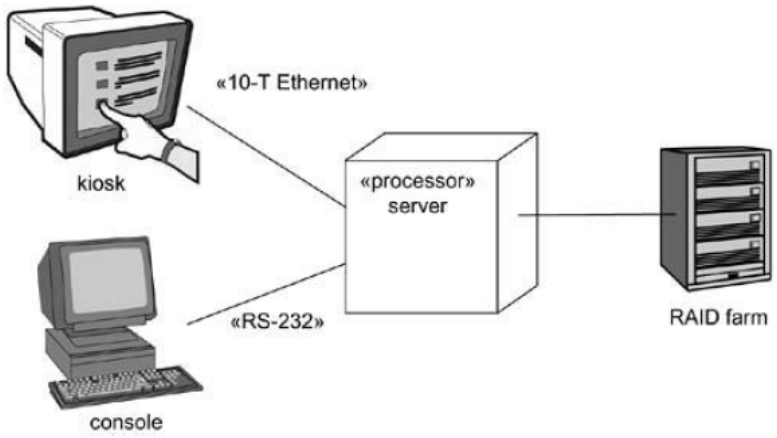


Common Modeling Techniques

Modeling processors and devices

To model processors and devices:

- Identify the computational elements of your system's deployment view and model each as a node.
- If these elements represent generic processors and devices, then stereotype them as such. If they are kinds of processors and devices that are part of the vocabulary of your domain, then specify an appropriate stereotype with an icon for each.
- As with class modeling, consider the attributes and operations that might apply to each node.

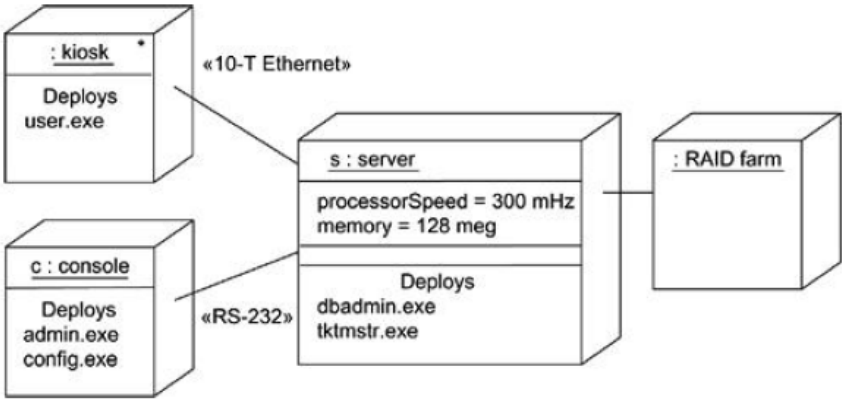


Modeling the distribution of components

To model the distribution of components:

- For each significant component in your system, allocate it to a given code.
- Consider duplicate locations for components.
- Render this allocation in one of the three ways:

1. Don't make the allocation visible, but leave it as part of the backplane of your model that is, in each node's specification.
2. Using dependency relationships, connect each node with the components it deploys.
3. List the components deployed on a node in an additional compartment.

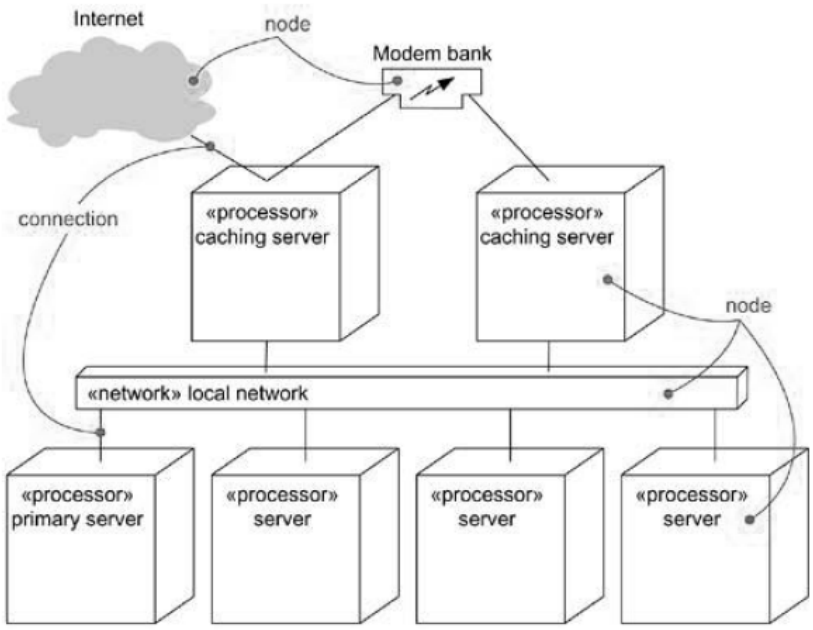


Chapter 4 –Deployment Diagram

Introduction:

Deployment diagrams are one of the two kinds of diagrams used in modeling the physical aspects of an object-oriented system. A deployment diagram shows the configuration of run time processing nodes and the components that live on them.

We use deployment diagrams to model the static deployment view of a system. A deployment diagram is a diagram that shows the configuration of run time processing nodes and the components that live on them.



Common Properties

A deployment is just a special kind of diagram that shares the same properties as all other diagrams like: a name and graphical contents. What distinguishes a deployment diagram from the rest of the diagrams is its content.

A deployment diagram commonly contains:

- Nodes
- Dependency and association relationships

Like all other diagrams, deployment diagrams may contain notes and constraints.

Deployment diagrams may also contain components, each of which must live on some node. Deployment diagrams may also contain packages and subsystems.

Common Uses

When modeling the static deployment view of a system, we'll typically use deployment diagrams in one of the three ways:

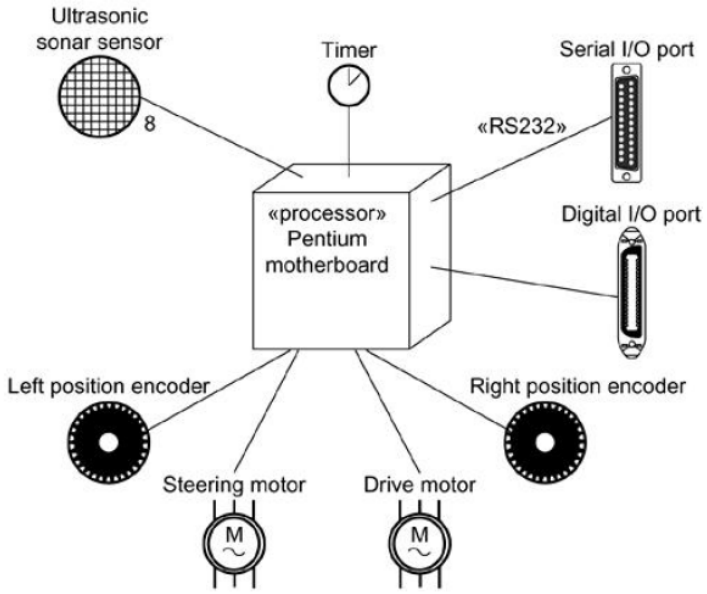
1. To model embedded systems.
2. To model client/server systems.
3. To model fully distributed systems.

Common Modeling Techniques

Modeling an embedded system

To model an embedded system:

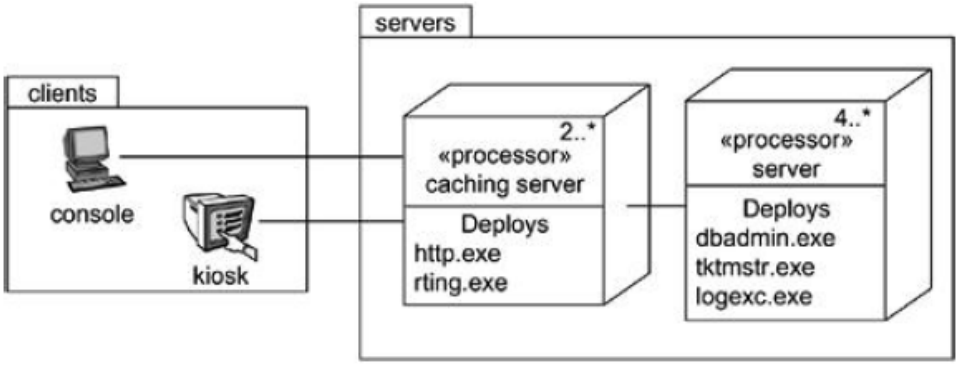
- Identify the devices and nodes that are unique to your system.
- Provide visual cues, especially for unusual devices, by using stereotypes.
- Model the relationships among these processors and devices in a deployment diagram. Similarly, specify the relationship between components and nodes.
- As necessary, expand on the intelligent devices by modeling their structure with a more detailed deployment diagram.



Modeling a client/server system

To model a client/server system:

- Identify the nodes that represent your system's client and server processors.
- Highlight those devices that are relevant to the behavior of your system.
- Provide visual cues for these processors and devices via stereotyping.
- Model the topology of these nodes in a deployment diagram



Modeling a fully distributed system

To model a fully distributed system:

- Identify the system's devices and processors as for simpler client/server systems.
- If you need to reason about the performance of the system's network or the impact of changes to the network, be sure to model these communication devices to the level of detail sufficient to make assessments.
- Pay close attention to logical groupings of nodes, which you can specify by using packages.
- Model these devices and processors using deployment diagrams.
- If you need to focus on the dynamics of the system, introduce use case diagrams to specify the kind of behavior you are interested in, and expand on these use cases with interaction diagrams.