Component Diagrams

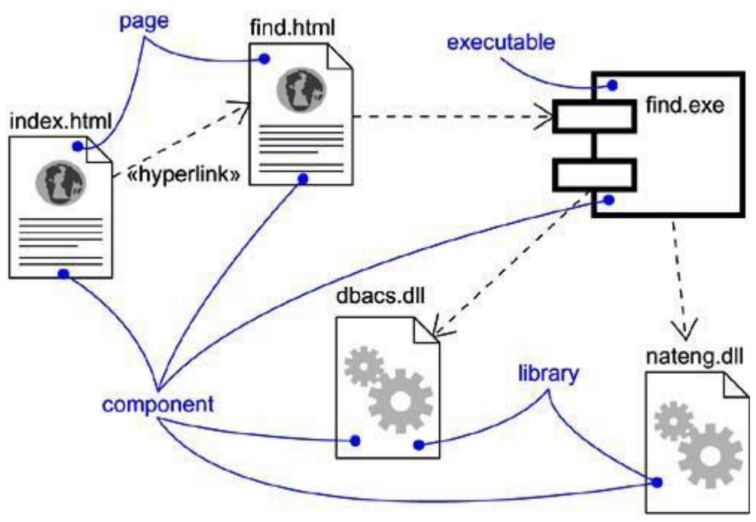


Figure : Component Diagram

Component Diagrams

- Component diagrams are used in modeling the physical aspects of object-oriented systems.
- A component diagram shows the organization and dependencies among a set of components.
- Component diagrams are used to model the static implementation view of a system.
- Component diagrams are essentially class diagrams that focus on a system's components.
- Graphically, a Component diagram is a collection of vertices and arcs.
- Component diagrams are used for visualizing, specifying, and documenting component-based systems and also for constructing executable systems through forward and reverse engineering.
- Component diagrams commonly contain Components, Interfaces and Dependency, generalization, association, and realization relationships. It may also contain notes and constraints.

Common Modeling Techniques

Modeling Source Code

To model a system's source code,

- Either by forward or reverse engineering identifies the set of source code files of interest and model them as components stereotyped as files.
- For larger systems, use packages to show groups of source code files.
- Consider exposing a tagged value indicating such information as the version number of the source code file, its author, and the date it was last changed. Use tools to manage the value of this tag.
- Model the compilation dependencies among these files using dependencies. Again, use tools to help generate and manage these dependencies.

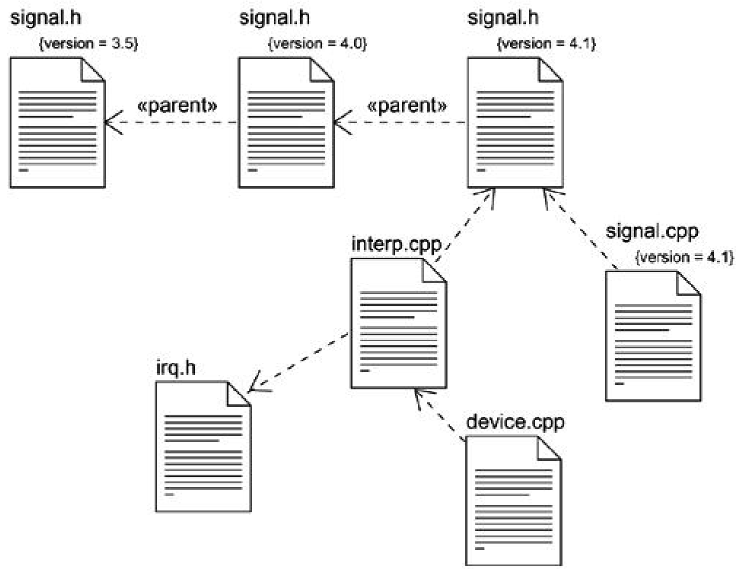


Figure : Modeling Source Code

Modeling an Executable Release

To model an executable release,

- Identify the set of components you'd like to model. Typically, this will involve some or all the components that live on one node, or the distribution of these sets of components across all the nodes in the system.
- Consider the stereotype of each component in this set. For most systems, you'll find a small number of different kinds of components (such as executables, libraries, tables, files, and documents). You can use the UML's extensibility mechanisms to provide visual cues(clues) for these stereotypes.
- For each component in this set, consider its relationship to its neighbors. Most often, this will involve interfaces that are exported (realized) by certain components and then imported (used) by others. If you want to expose the seams in your system, model these interfaces explicitly. If you want your model at a higher level of abstraction, elide these relationships by showing only dependencies among the components.

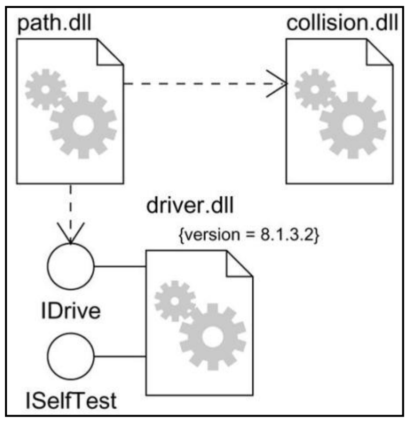


Figure : Modeling an Executable Release

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 will also want to consider the physical distribution of your databases. Your mapping strategy will be affected by the location in which you want your data to live on your deployed system.

- 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.

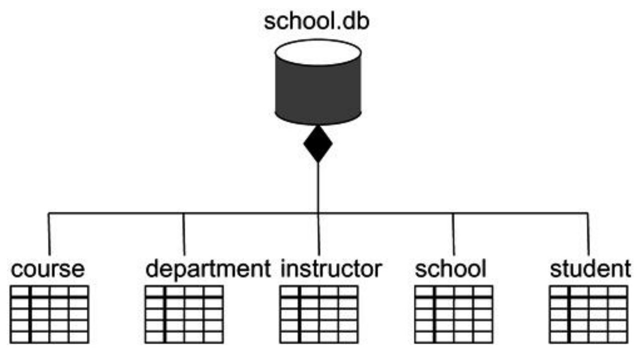


Figure : Modeling a Physical Database

Modeling Adaptable Systems

To model an adaptable system,

- Consider the physical distribution of the components that may migrate from node to node. You can specify the location of a component instance by marking it with a location tagged value, which you can then render in a component diagram (although, technically speaking, a diagram that contains only instances is an object diagram).
- If you want to model the actions that cause a component to migrate, create a corresponding interaction diagram that contains component instances. You can illustrate a change of location by drawing the same instance more than once, but with different values for its location tagged value.

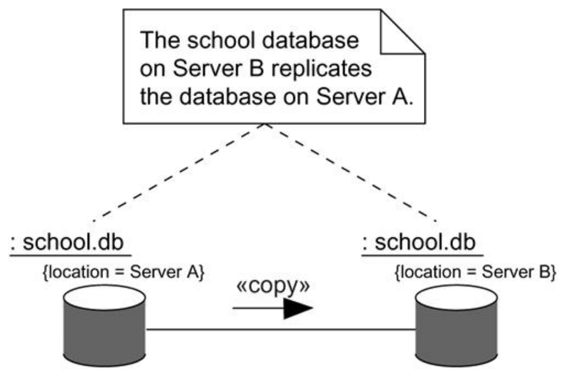
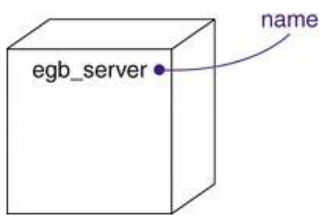


Figure : Modeling Adaptable Systems

Deployment

The UML provides a graphical representation of node. This canonical notation permits you to visualize a node apart from any specific hardware. Using stereotypes one of the UML's extensibility mechanisms you can (and often will) tailor this notation to represent specific kinds of processors and devices.

Figure : Nodes

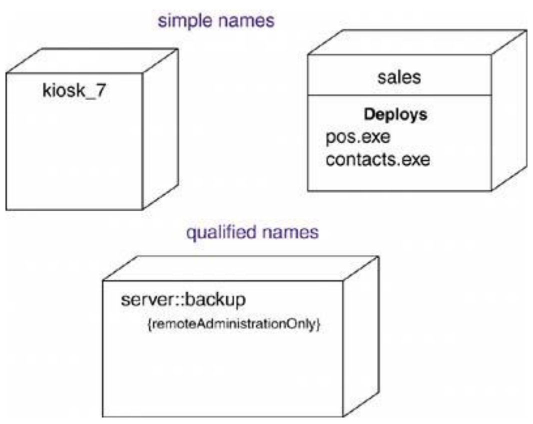


A *node* is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. Graphically, a node is rendered as a cube.

Names

Every node must have a name that distinguishes it from other nodes. A *name* is a textual string. That name alone is known as a *simple name*; a *qualified name* is the node name prefixed by the name of the package in which that node lives.

Figure Nodes with Simple and Qualified Names



Nodes and components

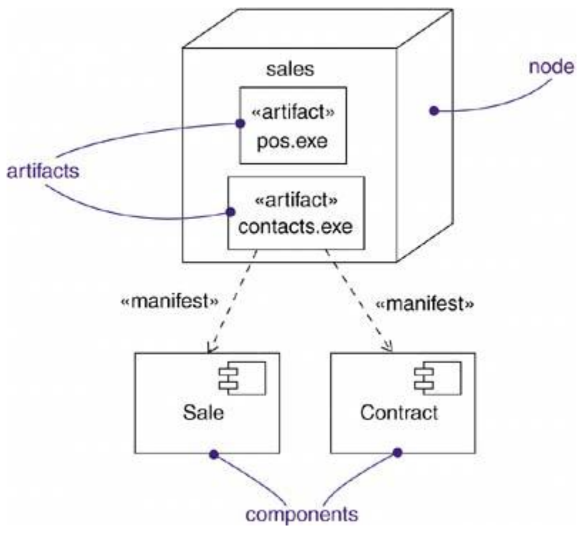
Nodes are a lot like components: Both have names; both may participate in dependency, generalization, and association relationships; both may be nested; both may have instances; both may be participants in interactions. significant differences between nodes and components are.

- Components are things that participate in the execution of a system; nodes are things that execute components.
- Components represent the physical packaging of otherwise logical elements; nodes represent the physical deployment of components.

This first difference is, nodes execute components; components are things that are executed by nodes.

The second difference suggests a relationship among classes, components, and nodes. A component is the manifestation of a set of logical elements, such as classes and collaborations, and a node is the location upon which components are deployed. A class may be manifested by one or more components, and, in turn, an component may be deployed on one or more nodes.

Figure: Nodes and Components



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

Organizing Nodes

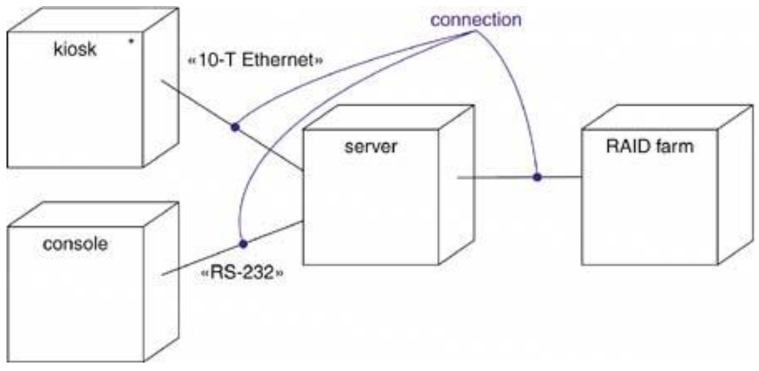
- You can organize nodes by grouping them in packages in the same manner in which you can organize classes and components.
- You can also organize nodes by specifying dependency, generalization, and association (including aggregation) relationships among them.

Connections

The most common kind of relationship 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.

We can include roles, multiplicity, and constraints.

Figure : Connections



Common Modeling Techniques

Modeling Processors and Devices

Modeling the processors and devices that form the topology of a stand-alone, embedded, client/server, or distributed system is the most common use of nodes.

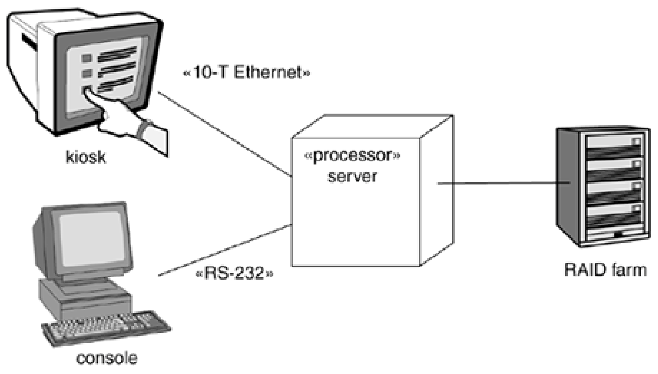
A *processor* is a node that has processing capability, meaning that it can execute a component.

A *device* is a node that has no processing capability (at least, none that are modeled at this level of abstraction) and, in general, represents something that interfaces to the real world.

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.

Figure : Processors and Devices

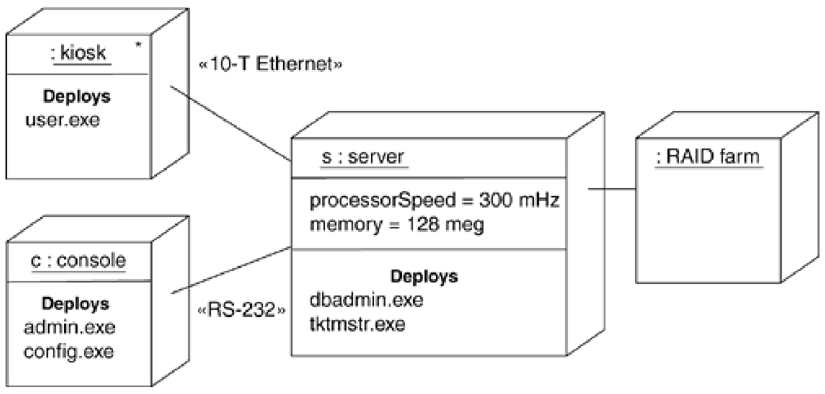


Modeling the Distribution of Components

To model the distribution of components,

- For each significant component in your system, allocate it to a given node.
- Consider duplicate locations for components. It's not uncommon for the same kind of component (such as specific executables and libraries) to reside on multiple nodes simultaneously.
- Render this allocation in one of three ways.
  1. Don't make the allocation visible, but leave it as part of the backplane of your modelthat 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.

Figure: Modeling the Distribution of Components



<u>Deployment Diagrams</u>

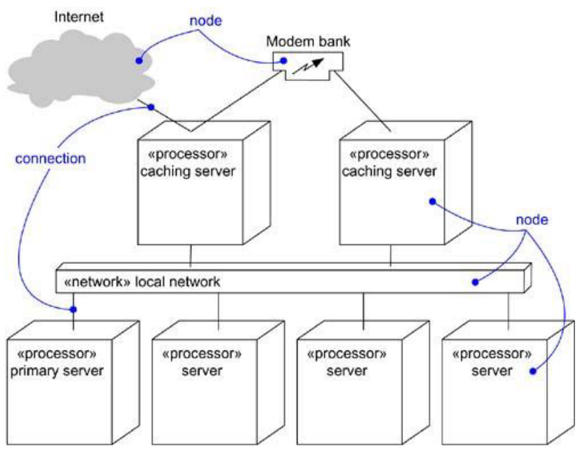


Figure : Deployment Diagram

Deployment Diagrams

- A deployment diagram is a diagram that shows the configuration of run time processing nodes and the components that live on them.
- Deployment diagrams are one of the two kinds of diagrams used in modeling the physical aspects of an object-oriented system.
- used to model the static deployment view of a system (topology of the hardware)
- A deployment diagram is just a special kind of class diagram, which focuses on a system's nodes.
- Graphically, a deployment diagram is a collection of vertices and arcs.
- Deployment diagrams commonly contain Nodes and Dependency & association relationships. It may also contain notes and constraints.
- Deployment diagrams are important for visualizing, specifying, and documenting embedded, client/server, and distributed systems and also for managing executable systems through forward and reverse engineering.

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 the UML's extensibility mechanisms to define system-specific stereotypes with appropriate icons. At the very least, you'll want to distinguish processors (which contain software components) and devices (which, at that level of abstraction, don't directly contain software).
- Model the relationships among these processors and devices in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.
- As necessary, expand on any intelligent devices by modeling their structure with a more detailed deployment diagram.

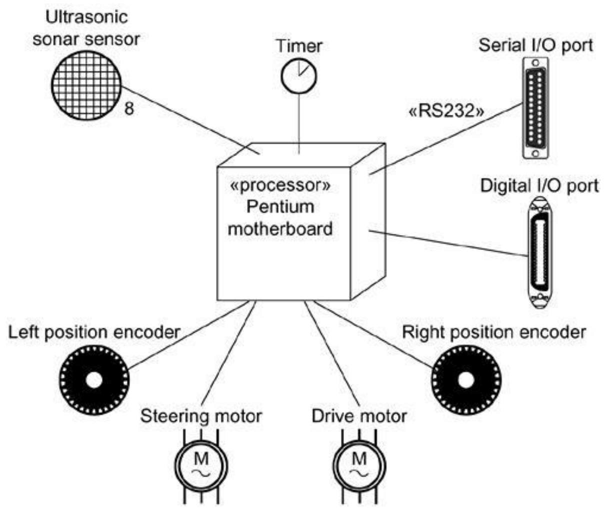


Figure : Modeling an Embedded System

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 germane to the behavior of your system. For example, you'll want to model special devices, such as credit card readers, badge readers, and display devices other than monitors, because their placement in the system's hardware topology are likely to be architecturally significant.
- Provide visual cues for these processors and devices via stereotyping.
- Model the topology of these nodes in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.

Figure 3 shows the topology of a human resources system, which follows a classical client/server architecture.

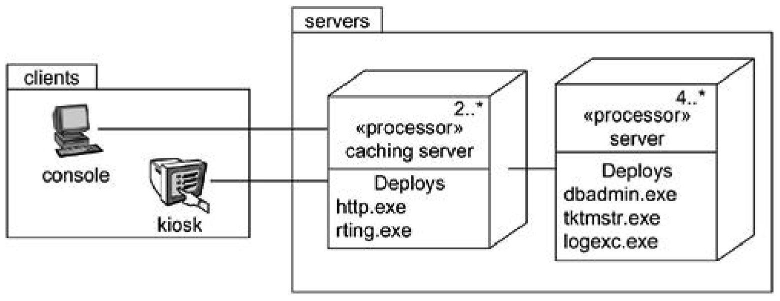


Figure : Modeling a Client/ Server System

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 these assessments.\
- Pay close attention to logical groupings of nodes, which you can specify by using packages.

- Model these devices and processors using deployment diagrams. Where possible, use tools that discover the topology of your system by walking your system's network.
  If you need to focus on the dynamics of your system, introduce use case diagrams to specify the kinds of behavior you are interested in, and expand on these use cases with interaction diagrams.
- When modeling a fully distributed system, it's common to reify the network itself as an node. eg:- Internet, LAN, WAN as nodes

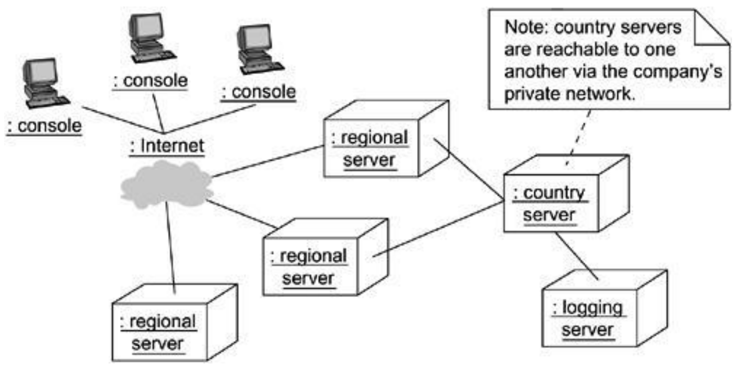


Figure : Modeling a Fully Distributed System