UNIT-VI
UML Learning Material

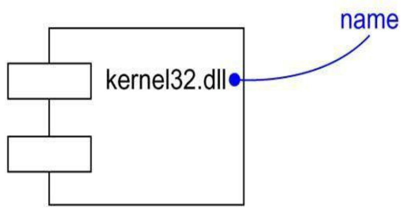
Syllabus:

Architectural Modeling: Components, Deployment, Component diagrams, Deployment diagrams, Common Modeling Techniques for Component and Deployment Diagrams. Case Study: The Unified Library application.

Component:

Components live in the material world of bits and therefore are an important building block in modeling the physical aspects of a system.

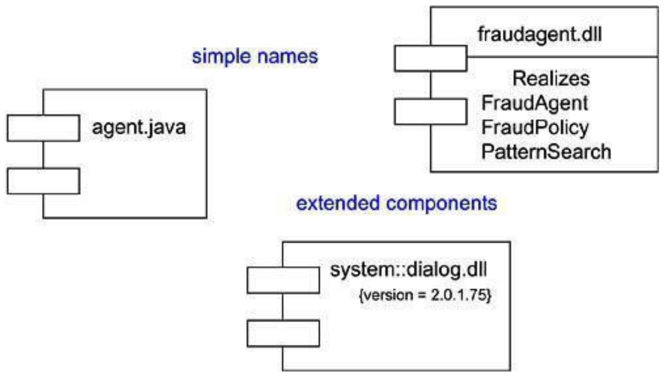
A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.



Names:

Every component must have a name that distinguishes it from other components. A *name* is a textual string.

That name alone is known as a *simple name;* a *path name* is the component name prefixed by the name of the package in which that component lives.

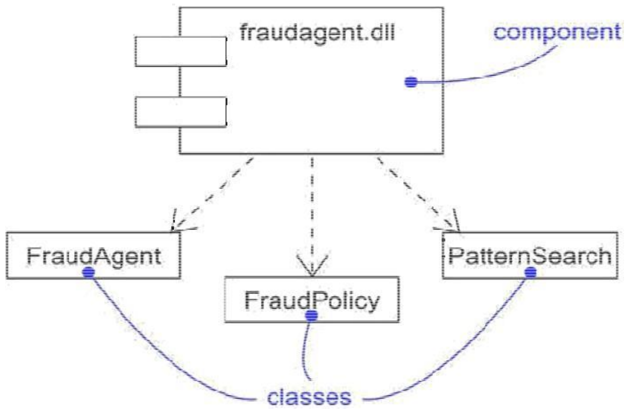


Components and Classes:

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; both may have instances; both may be participants in interactions.

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.

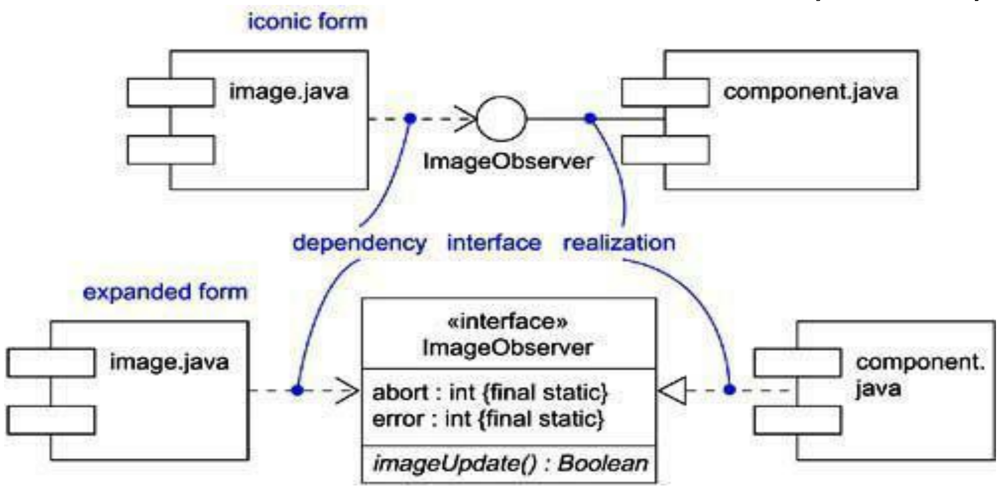


Components and Classes

<u>Components and Interfaces:</u>

An interface is a collection of operations that are used to specify a service of a class or a component.
The relationship between component and interface is
important. Component and its interfaces in one of two ways:

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



<u>Binary Replaceability:</u>

We can create a system out of components and then evolve that system by adding new components and replacing 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 substitutableit is possible to replace a component with another that conforms to the same 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.

<u>Kinds of Components:</u>

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

Work product components*:* These components are essentially the residue of the development process, consisting of things such as source code files and data files from which deployment components are created.

E*xecution components:* These components are created as a consequence of an executing system, such as a COM+ object, which is instantiated from a DLL.

<u>Standard Elements</u>

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 source code or data

Document : Specifies a component that represents a document

Common Modeling Techniques

1). <u>Modeling Executables and Libraries</u>: The most common purpose for which you'll use components is to model the deployment components that make up your implementation. For most systems, these deployment components are drawn from the decisions we make about how to segment the physical implementation of our system. These decisions will be affected by a number of technical issues, configuration management issues, and reuse issues.

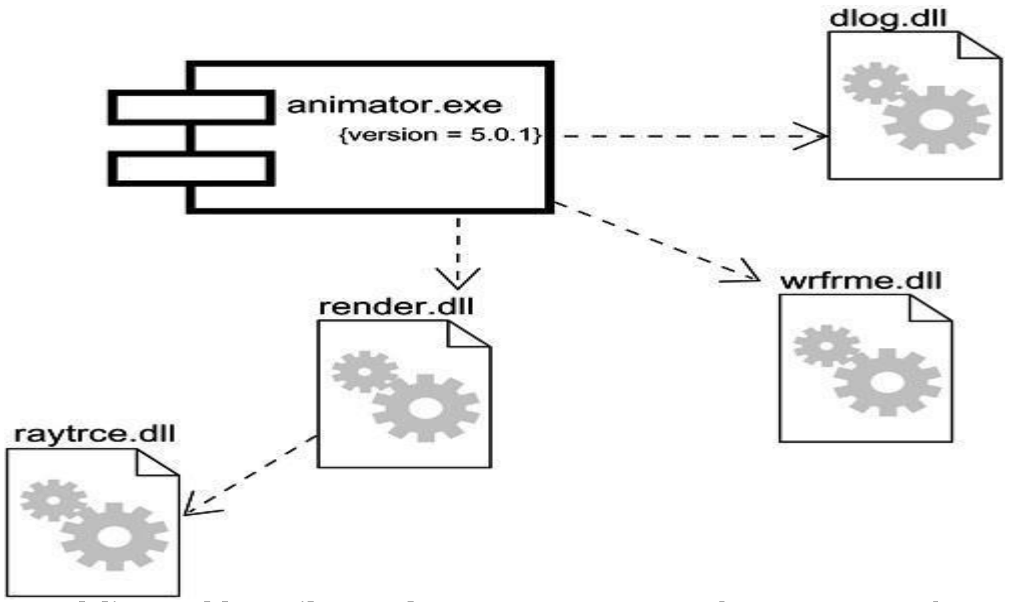
To model executables and libraries,

Identify the partitioning of your physical system. Consider the impact of technical, configuration management, and reuse issues.

Model any executables and libraries as components, using the appropriate standard elements. To implement new kinds of components, introduce a new appropriate stereotype.

If it's important for to manage the seams in the system, 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. Most often,to model the dependencies among these parts in order to visualize the impact of change.



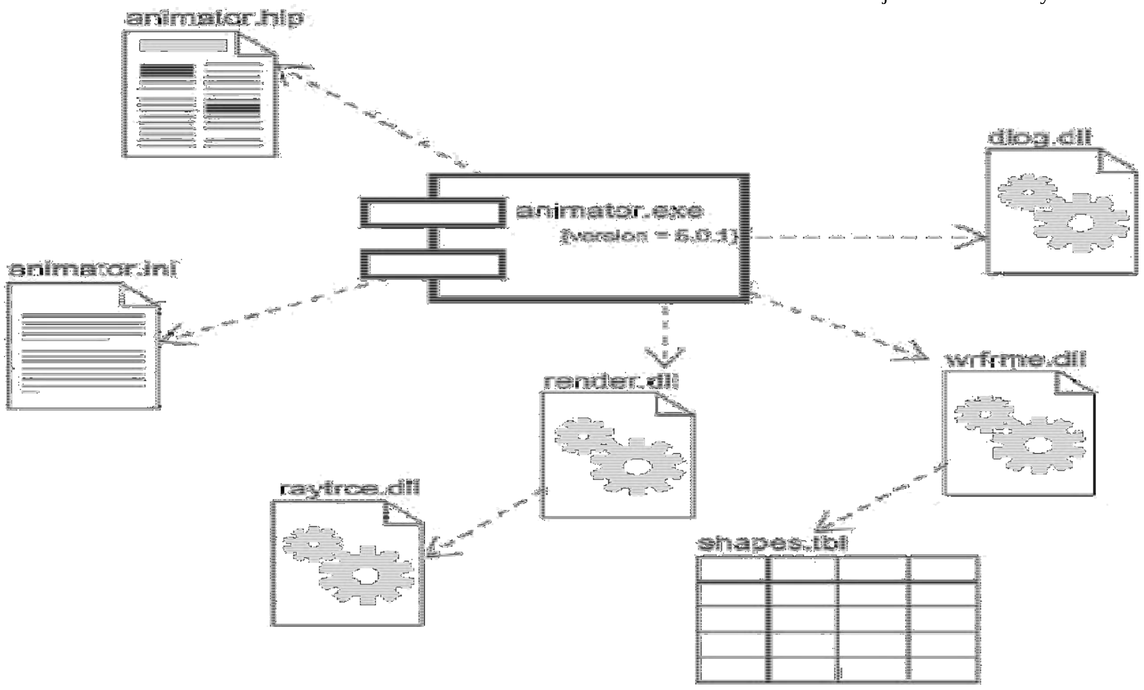
2). <u>Modeling Tables, Files, and Documents:</u> Our implementation might include data files, help documents, scripts, log files, initialization files, and installation/removal files. Modeling these components is an important part of controlling the configuration of your system. Fortunately, you can use UML components to model all of these artifacts.

To model tables, files, and documents,

Identify the ancillary components that are part of the physical implementation of the system.

Model these things as components. If the implementation introduces new kinds of artifacts, introduce a new appropriate stereotype.

As necessary to communicate intent, model the relationships among these ancillary components and the other executables, libraries, and interfaces in the system. Most often, you'll want to model the dependencies among these parts in order to visualize the impact of change



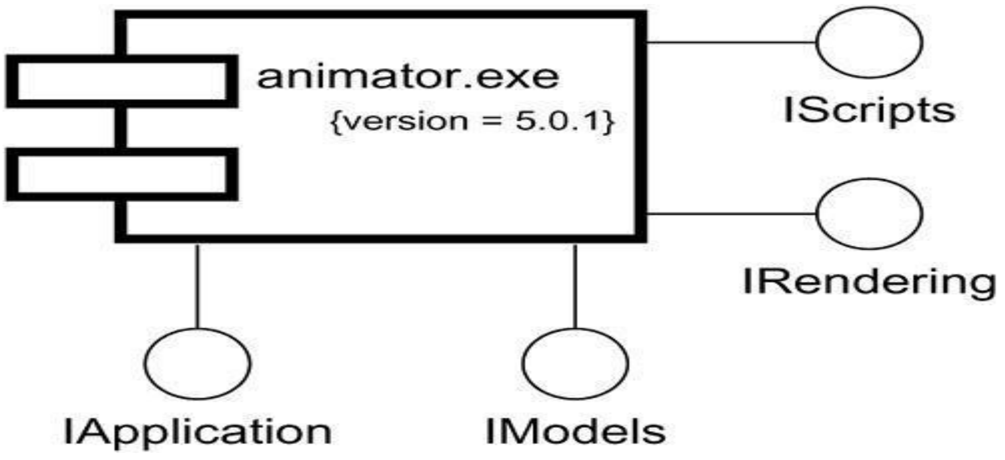
3). **Modeling an API:** An API is essentially an interface that is realized by one or more components. As a developer, you'll really care only about the interface itself; which component realizes an interface's operations is not relevant as long as *some* component realizes it.

To model an API,

Identify the programmatic seams in the system and model each seam as an interface, collecting the attributes and operations that form this edge.

Expose only those properties of the interface that are important to visualize in 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 insofar as it is important to show the configuration of a specific implementation.



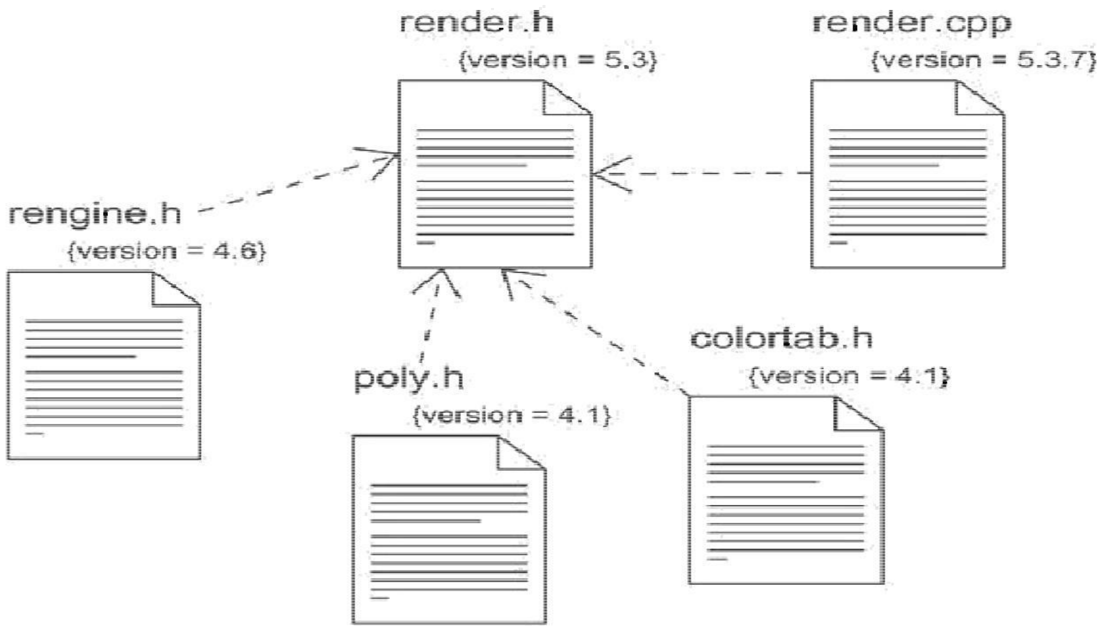
4). <u>Modeling Source Code :</u> Modeling source code graphically is particularly useful for visualizing the compilation dependencies among source code files and for managing the splitting and merging of groups of these files when fork and join development paths. In this manner, UML components can be the graphical interface to configuration management and version control tools.

To model source code,

Depending on the constraints imposed by the development tools, model the files used to store the details of all logical elements, along with their compilation dependencies.

If it's important for to bolt these models to configuration management and version control tools, to include tagged values, such as version, author, and check in/check out information, for each file that's under configuration management.

As far as possible, let development tools manage the relationships among these files, and use the UML only to visualize and document these relationships



Component Diagrams

Component diagrams are one of the two kinds of diagrams found in modeling the physical aspects of object-oriented systems.

“A component diagram shows the organization and dependencies among a set of components”.

Use component diagrams to model the static implementation view of a system. This involves modeling the physical things that reside on a node, such as executables, libraries, tables, files, and documents.

Component diagrams are essentially class diagrams that focus on a system's components.

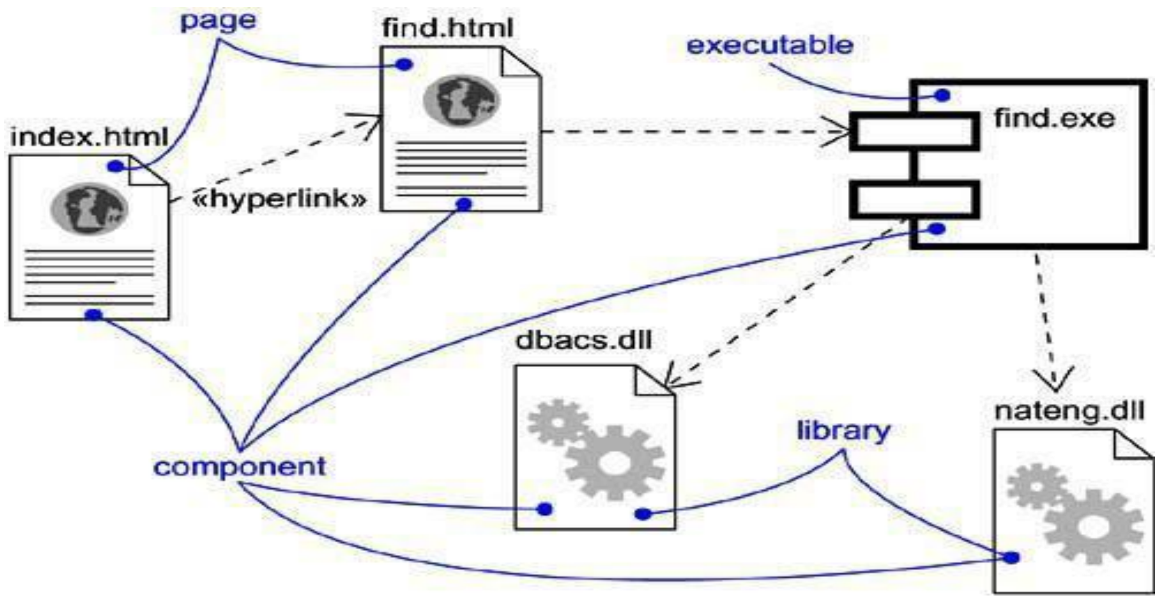
Component diagrams comprise of:

Components

Interfaces

Relationships

Packages and Subsystems (optional)



Component diagrams are used for:

Constructing systems through forward and reverse engineering.

Modeling configuration management of source code files while developing a system using an object-oriented programming language.

Representing schemas in modeling databases.

Modeling behaviors of dynamic systems.

Common Modeling Techniques of Component diagrams:

1) <u>Modeling Source Code :</u>

We will develop software in Java, you'll usually save your source code in .java files. If you develop software using C++, you'll typically store your source code in header files (.h files) and bodies (.cpp files).

To visualize the source code files and their relationships using component diagrams. Component diagrams used in this way typically contain only work-product components stereotyped as files, together with dependency relationships.

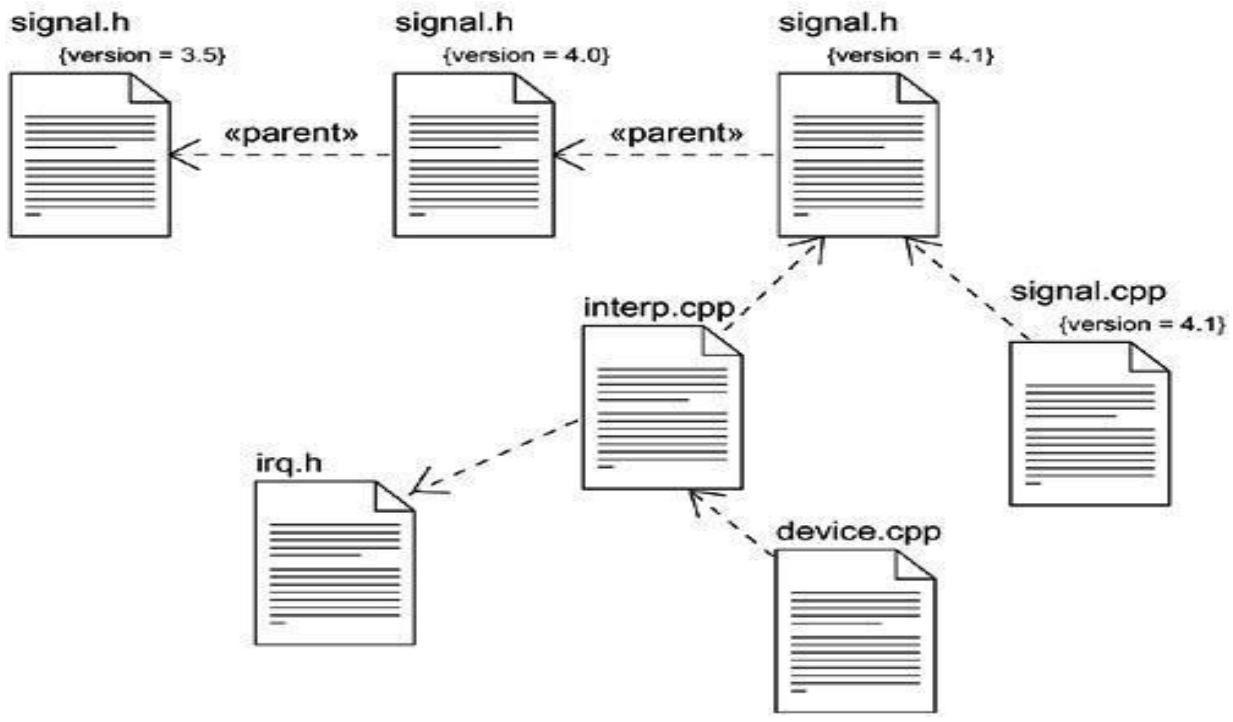
To model a system's source code,

Either by forward or reverse engineering, identify the set of source code files of interest and model them as components 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.



2). **Modeling an Executable Release :**

Releasing a simple application is easy: You throw the bits of a single executable file on a disk, and your users just run that executable.

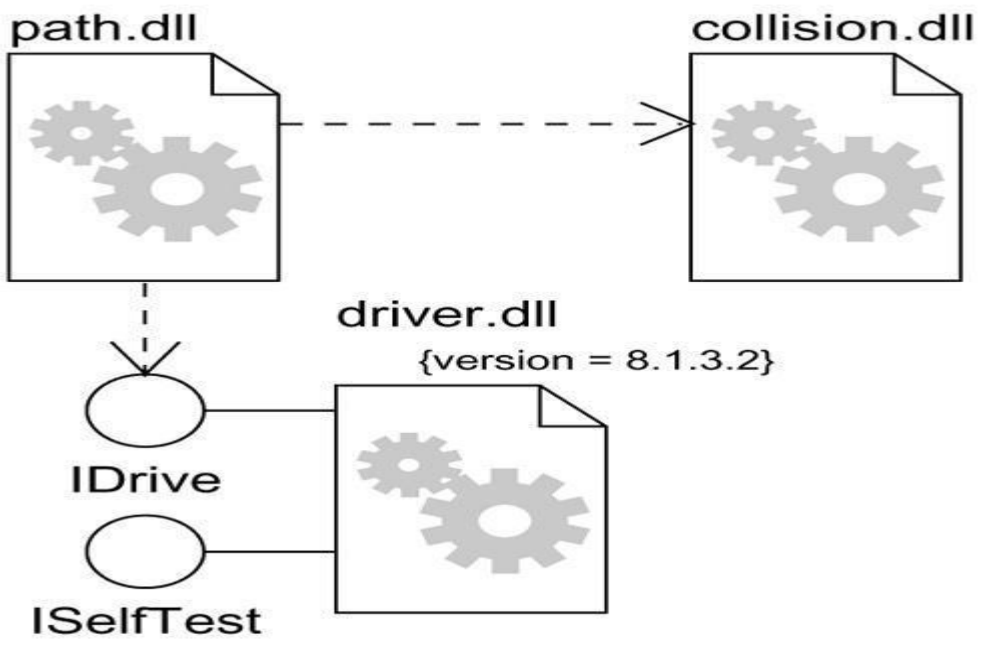
Releasing anything other than a simple application is not so easy. Need the main executable (usually, a .exe file), but you also need all its ancillary parts, such as libraries (commonly .dll).

To model an executable release,

Identify the set of components 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, find a small number of different kinds of components (such as executables, libraries, tables, files, and documents). We can use the UML's extensibility mechanisms to provide visual cues 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.



3). Modeling a Physical Database :

A logical database schema captures the vocabulary of a system's persistent data, along with the semantics of their relationships.

Physically, these things are stored in a database for later retrieval, either a relational database.

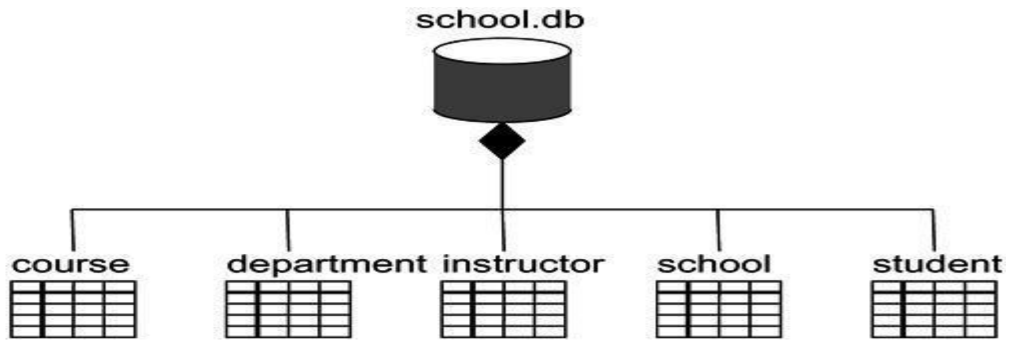
Mapping a logical database schema to an object-oriented database is straightforward because even complex inheritance lattices can be made persistent directly.

Typically, we can apply one or a combination of three strategies

1. Define a separate table for each class. This is a simple but naive approach because it introduces maintenance headaches when you add new child classes or modify your parent classes.
2. Collapse your inheritance lattices so that all instances of any class in a hierarchy has the same state. The downside with this approach is that you end up storing superfluous information for many instances.
3. Separate parent and child states into different tables. This approach best mirrors your inheritance lattice, but the downside is that traversing your data will require many cross-table joins.

When designing a physical database, you also have to make decisions about how to map operations defined in our logical database schema. We have some choices.

1. For simple CRUD (create, read, update, delete) operations, implement them with standard SQL or ODBC calls.
2. For more-complex behavior (such as business rules), map them to triggers or stored procedures.



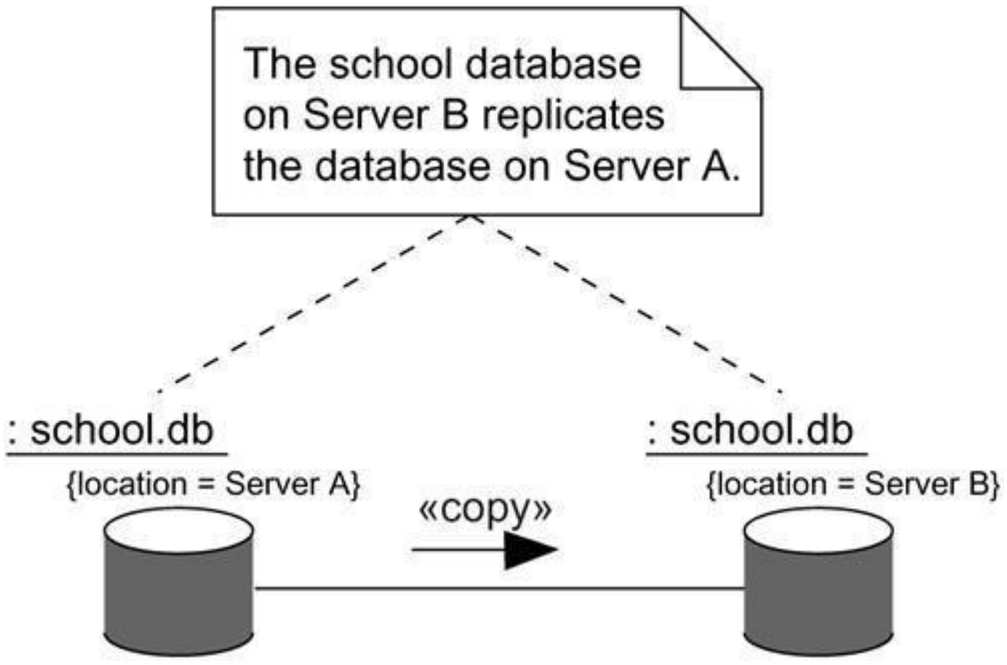
4). **Modeling Adaptable Systems :**

All the component diagrams shown thus far have been used to model static views. Their components spend their entire lives on one node. This is the most common situation you'll encounter, but especially in the domain of complex, distributed systems, we need to model dynamic views- need to use a combination of component diagrams, object diagrams, and interaction diagrams

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.



5). **Forward and Reverse Engineering :**

Forward engineering and reverse engineering components are pretty direct, because components are themselves physical things (executables, libraries, tables, files, and documents) that are close to the running system.

When you forward engineer a class or a collaboration, you really forward engineer to a component that represents the source code, binary library, or executable for that class or collaboration.

When you reverse engineer source code, binary libraries, or executables, you really reverse engineer to a component or set of components that, in turn, trace to classes or collaborations.

To forward engineer(the creation of code from a model) a component diagram,

For each component, identify the classes or collaborations that the component implements.

Choose the target for each component. Your choice is basically between source code or a binary library or executable.

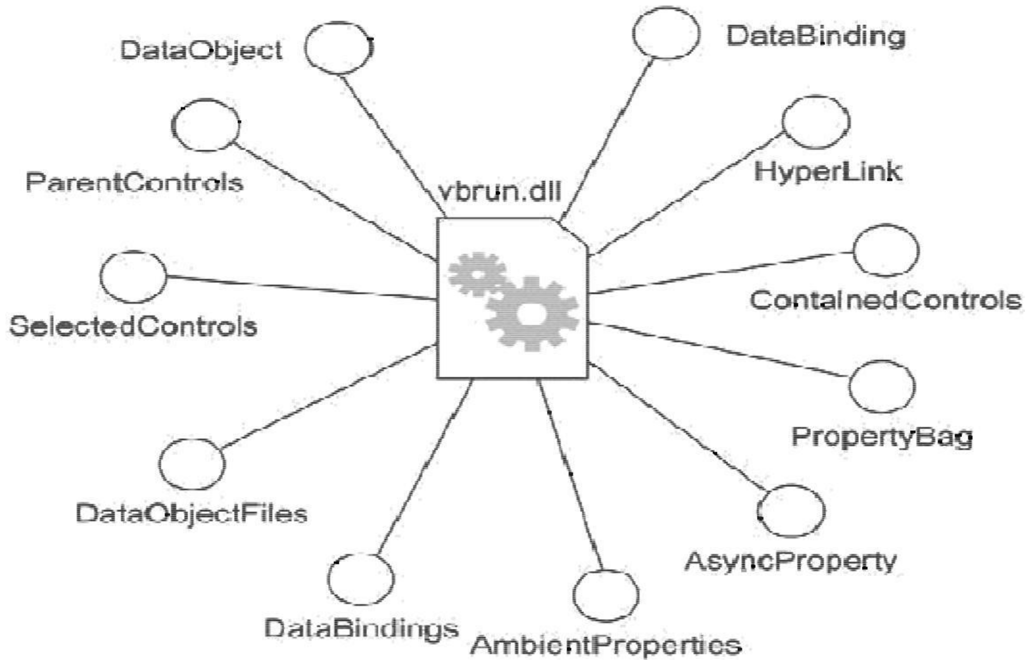
Use tools to forward engineer your models.

To reverse engineering (the creation of a model from code) a component diagram

Choose the target you want to reverse engineer. Source code can be reverse engineered to components and then classes. Binary libraries can be reverse engineered to uncover their interfaces. Executables can be reverse engineered the least.

Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or to modify an existing one that was previously forward engineered.

Using the tool, create a component diagram by querying the model.



Deployments:

Nodes, just like components, live in the material world and are an important building block in modeling the physical aspects of a system. 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.

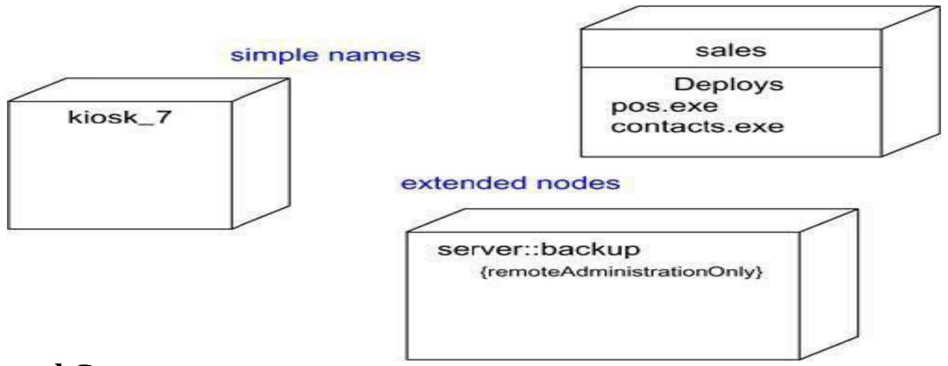
A node typically represents a processor or a device on which components may be deployed.

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.

Simple name: Name alone is known as a simple name.

*Path name: N*ode name prefixed by the name of the package in which that node lives.



Nodes and Components

Nodes are a lot like components:

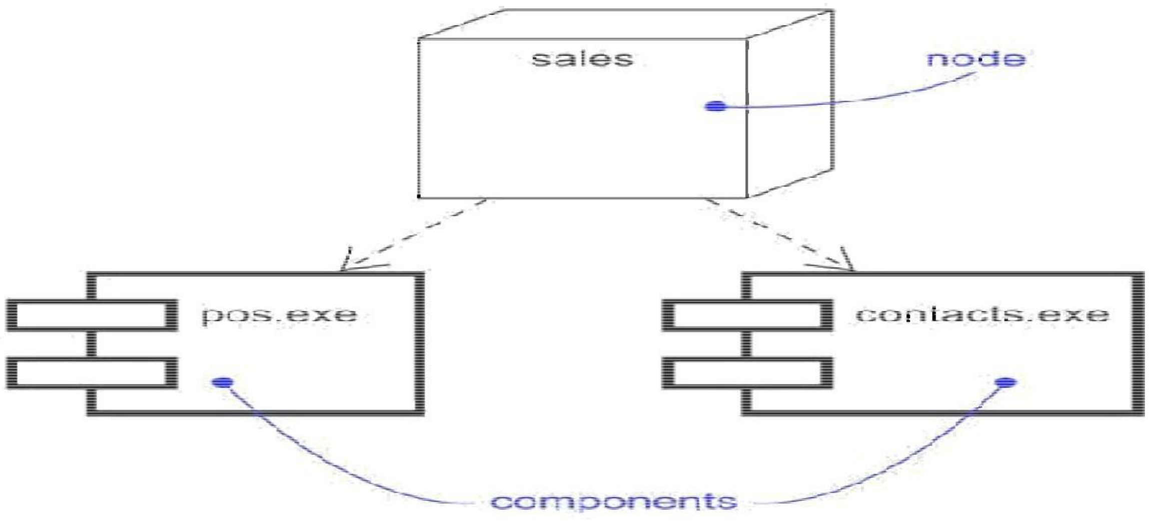
Both have names; both may participate in dependency, generalization, and association relationships; both may be nested; both may have instances; both may be participants in interactions.

There are some significant differences between nodes and components:

Components are things that participate in the execution of a system; nodes are things that execute components.

Components are things that participate in the execution of a system; nodes are things that execute components.

The relationship between a node and the components it deploys can be shown explicitly by using a dependency relationship.



The most common kind of relationship you'll use among nodes is an association. In this context, an association represents a physical connection among nodes.

Common Modeling Techniques of Deployments

1). <u>Modeling Processors and Devices:</u>

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.

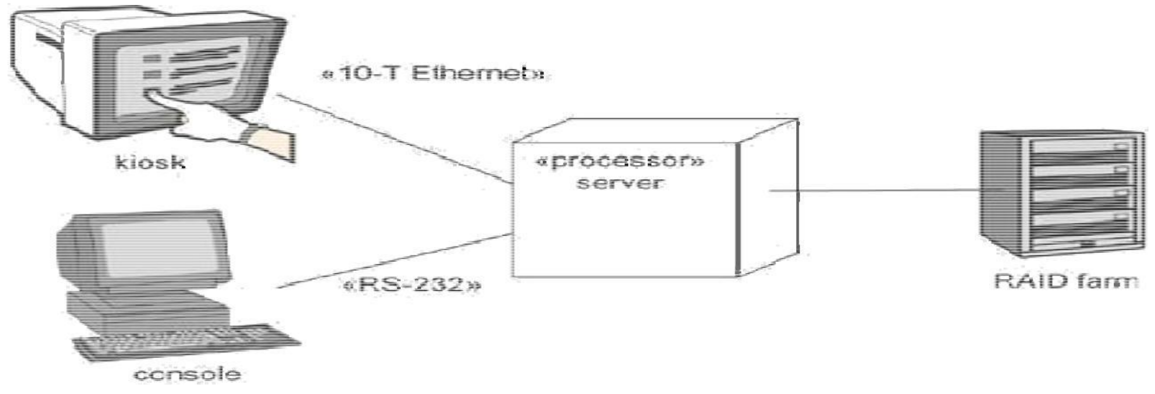
Nodes that you can use to represent specific kinds of processors and devices. A ***processor*** is a node that has processing capability, meaning that it can execute a component. A ***device*** is a node that has no processing capability and, in general, represents something that interfaces to the real 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.



2). Modeling the Distribution of Components :

It's often useful to visualize or specify the physical distribution of its components across the processors and devices that make up the system.

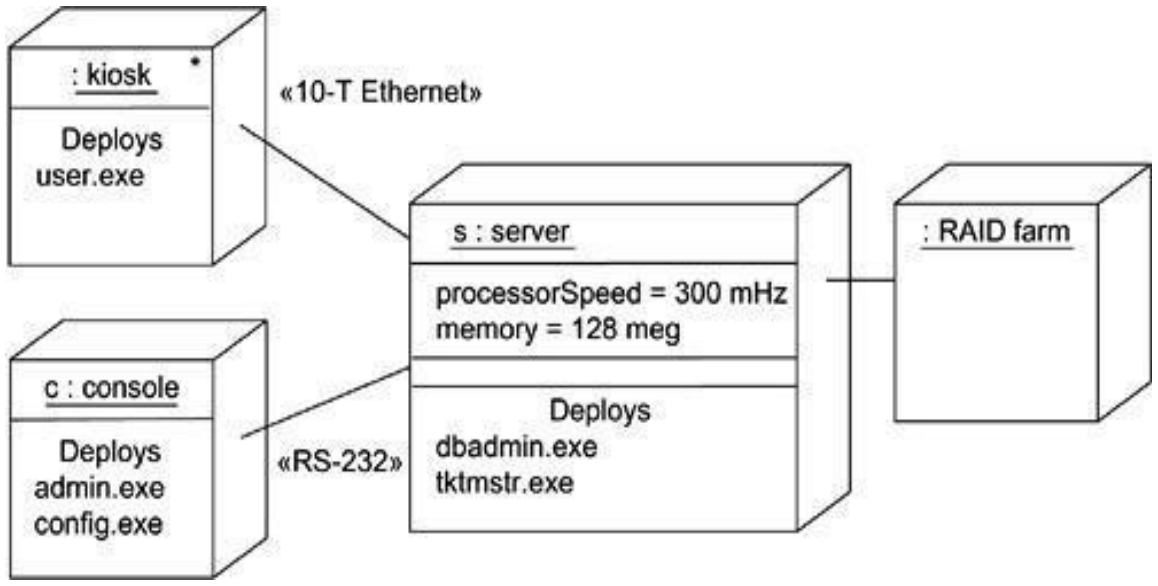
To model the distribution of components,

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



Deployment Diagrams:

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.

Deployment diagrams to model the static deployment view of a system. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system.

Deployment diagrams modeling the topology of the hardware on which your system executes

Deployment diagrams are essentially class diagrams that focus on a system's nodes.

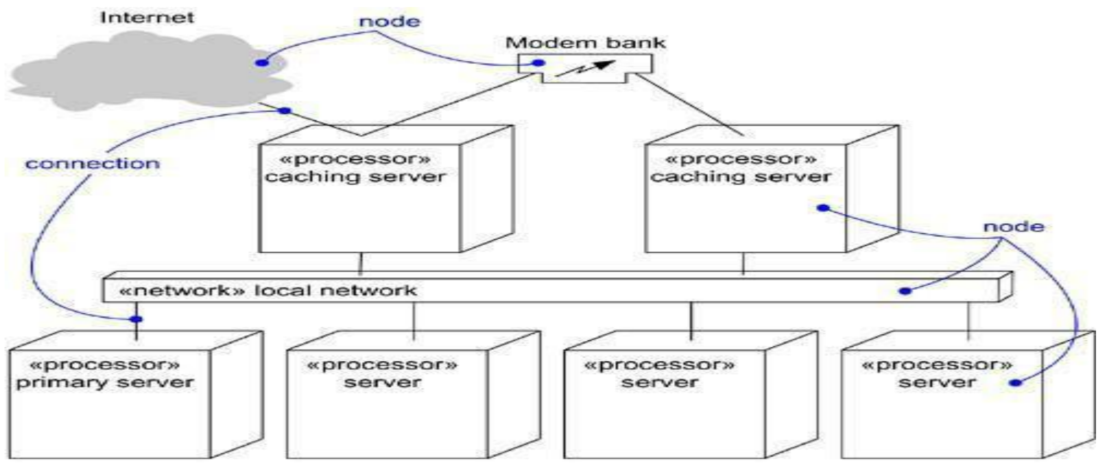
Deployment diagrams commonly contain

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.



Common Modeling Techniques of Deployment Diagram

1) .Modeling an Embedded System:

Developing an embedded system is far more than a software problem. To manage the physical world in which there are moving parts that break and in which signals are noisy and behavior is nonlinear.

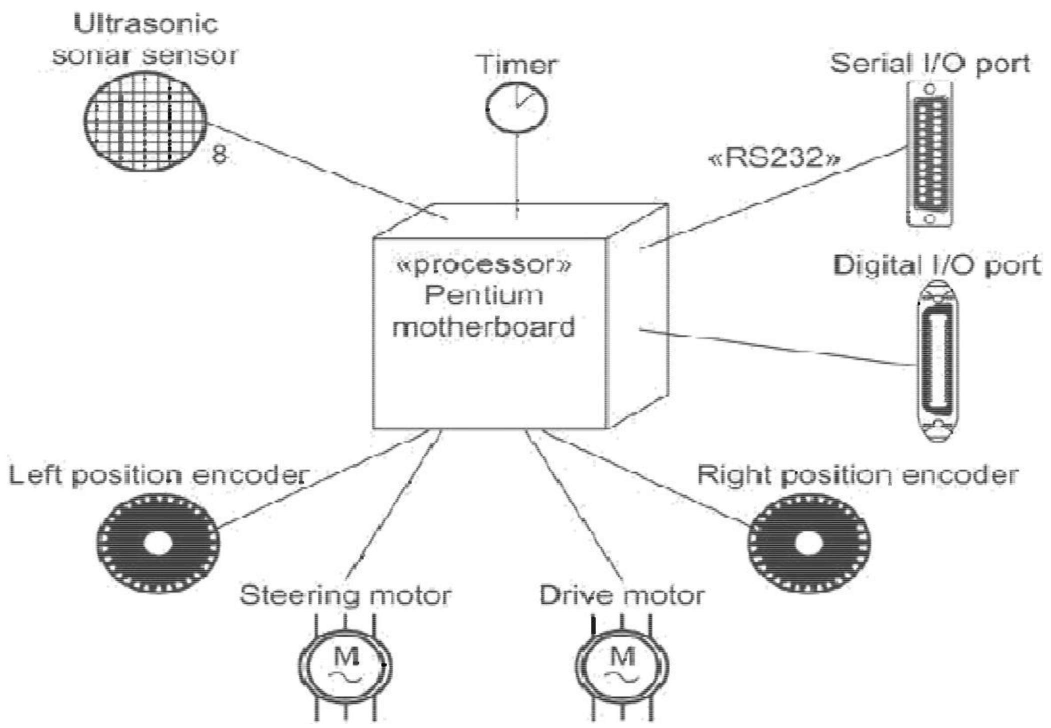
To model an embedded system,

Identify the devices and nodes that are unique to your system.

Povide 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 and devices.

Model the relationships among these processors and devices in a deployment diagram.

As necessary, expand on any intelligent devices by modeling their structure with a more detailed deployment diagram.



2) Modeling a Client/Server System :

The moment you start developing a system whose software no longer resides on a single processor, you are faced with a host of decisions: How do you best distribute your software components across these nodes? How do they communicate? How do you deal with failure and noise?

Ans: client/server systems, in which there's a clear separation of concerns between the system's user interface (typically managed by the client) and its data (typically managed by the server).

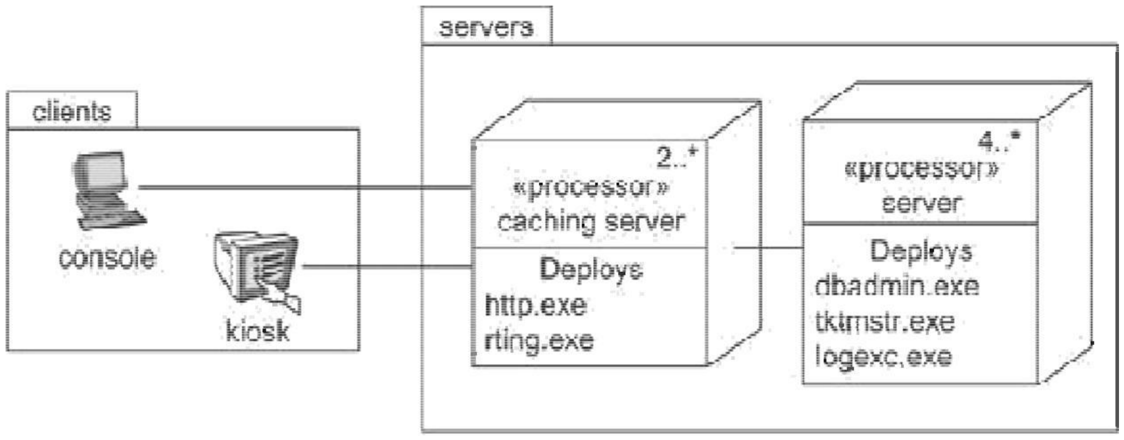
To model a client/server system,

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.



3) .Modeling a Fully Distributed System:

Distributed systems come in many forms, from simple two-processor systems to those that span many geographically dispersed nodes. The latter are typically never static. Nodes are added and removed as network traffic changes and processors fail; new and faster communication paths may be established in parallel with older

To model a fully distributed system,

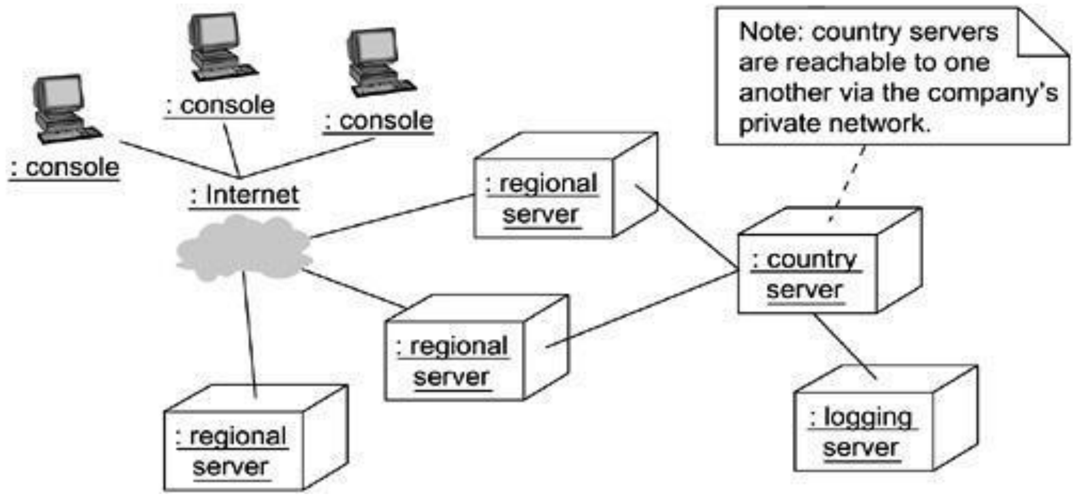
Identify the system's devices and processors as for simpler client/server systems.

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.



4) . Forward and Reverse Engineering

There's only a modest amount of forward engineering (the creation of code from models) that you can do with deployment diagrams.

For example, after specifying the physical distribution of components across the nodes in a deployment diagram, it is possible to use tools that then push these components out to the real world

Reverse engineering (the creation of models from code) from the real world back to deployment diagrams is of tremendous value, especially for fully distributed systems that are under constant change.

To reverse engineer a deployment diagram

Choose the target that you want to reverse engineer. In some cases, you'll want to sweep across your entire network; in others, you can limit your search.

Choose also the fidelity of your reverse engineering. In some cases, it's sufficient to reverse engineer just to the level of all the system's processors.

Use a tool that walks across your system, discovering its hardware topology. Record that topology in a deployment model.

You can use similar tools to discover the components that live on each node, which you can also record in a deployment model

Using modeling tools , create a deployment diagram by querying the model.

I) Descriptive Questions

)

1. Describe the common modeling Techniques of components?
2. Explain the common modeling Techniques for Components diagrams?
3. What are different types of components that are used in component diagram? List out different uses of component diagrams?
4. Elaborate the common modeling techniques of Deployment?
5. Explain the common modeling Techniques for Deployment diagrams?
6. Draw deployment diagrams to model a fully distributed system?

IT