Architectural Modeling

Introduction

- Patterns and Frameworks
- Component Diagrams
- Deployment Diagrams
- A Detailed Case Study on System Analysis and Design using Unified Approach.

Patterns and Frameworks

- All well-structured systems are full of patterns. A pattern provides a common solution to a common problem in a given context.
- A mechanism is a design pattern that applies to a society of classes; a framework is typically an architectural pattern that provides an extensible template for applications within a domain.
- You use patterns to specify mechanisms and frameworks that shape the architecture of your system. You make a pattern approachable by clearly identifying the slots, tabs, knobs, and dials that a user of that pattern may adjust in order to apply the pattern in a particular context.
- A pattern is a common solution to a common problem in a given context. A mechanism is a design pattern that applies to a society of classes.
- A framework is an architectural pattern that provides an extensible template for applications within a domain.

Patterns and Frameworks

- In all well-structured systems, you'll find lots of patterns at various levels of abstraction.
- **Design patterns** specify the structure and behavior of a society of classes; **architectural patterns** specify the structure and behavior of an entire system.
- In practice, there are two kinds of patterns of interest- design patterns and frameworks- and the UML provides a means of modeling both.
- When you model either pattern, you'll find that it typically stands alone in the context of some larger package, except for dependency relationships bind them to other parts of your system.

Design Patterns

- Patterns support reuse of software architecture and design.
  - Patterns capture the static and dynamic structures and collaborations of successful solutions to problems that arise when building applications in a particular domain.
- Frameworks support reuse of detailed design and code.
  - A framework is an integrated set of components that collaborate to provide a reusable architecture for a family of related applications.
- Together, design patterns and frameworks help to improve software quality and reduce development time.
  - e.g., reuse, extensibility, modularity, performance.
-

Design Patterns

- Design patterns represent solutions to problems that arise when developing software within a particular context
  - i.e., “Patterns == problem/solution pairs in a context”
- Patterns capture the static and dynamic structure and collaboration among key participants in software designs
  - They are particularly useful for articulating how and why to resolve non-functional forces
- Patterns facilitate reuse of successful software architectures and designs

Modeling Design Patterns

- To model a design pattern,
  - Identify the common solution to the common problem and reify it as a mechanism.
  - Model the mechanism as a collaboration, providing its structural, as well as its behavioral, aspects.
  - Identify the elements of the design pattern that must be bound to elements in a specific context and render them as parameters to the collaboration.

Frameworks

- 1. Frameworks are semi-complete applications
- Complete applications are developed by inheriting from, and instantiating parametrized framework components.
- 2. Frameworks provide domain-specific functionality
- e.g., business applications, telecommunication applications, window systems, databases, distributed applications, OS kernels
- 3. Frameworks exhibit inversion of control at run-time
- i.e., the framework determines which objects and methods to invoke in response to events

Modeling Architectural Patterns

- To model an Architectural pattern,
  - Harvest the framework from an existing, proven architecture.
  - Model the framework as a stereotyped package, containing all the elements (and especially the design patterns) that are necessary and sufficient to describe the various views of that framework.
  - Expose the slots, tabs, knobs, and dials necessary to adapt the framework in the form of design patterns and collaborations. For the most part, this means making it clear to the user of the pattern which classes must be extended, which operations must be implemented, and which signals must be handled.

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.
- Component diagrams are used 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 are not only important for visualizing, specifying, and documenting component-based systems, but also for constructing executable systems through forward and reverse engineering.*

Component Diagrams

- A component diagram is just a special kind of diagram and shares the same common properties as do all other diagrams- a name and graphical contents that are a projection into a model.
- What distinguishes a component diagram from all other kinds of diagrams is its particular content.

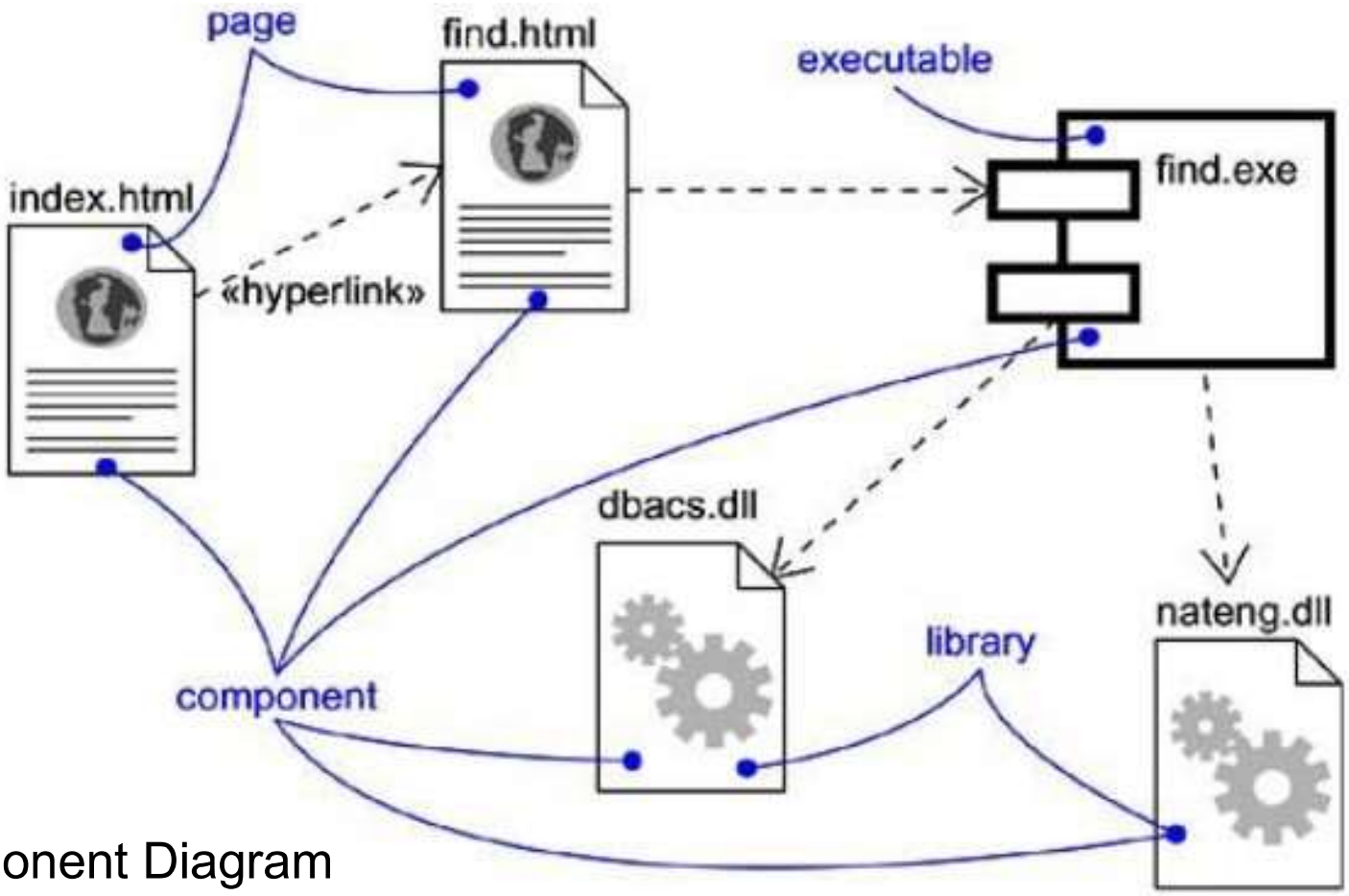


Figure 29-1 A Component Diagram

Component Diagrams

- You create use case diagrams to reason about the desired behavior of your system. You specify the vocabulary of your domain with class diagrams.
- You create sequence diagrams, collaboration diagrams, statechart diagrams, and activity diagrams to specify the way the things in your vocabulary work together to carry out this behavior.
- Eventually, you will turn these logical blueprints into things that live in the world of bits, such as executables, libraries, tables, files, and documents. You'll find that you must build some of these components from scratch, but you'll also end up reusing older components in new ways.
- With the UML, you use component diagrams to visualize the static aspect of these physical components and their relationships and to specify their details for construction, as in Figure 29-1.

Component Diagrams

- Component diagrams commonly contain
  - Components
  - Interfaces
  - Dependency, generalization, association, and realization relationships
- Like all other diagrams, component diagrams may contain notes and constraints.
- Component diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks.
- Sometimes, you'll want to place instances in your component diagrams, as well, especially when you want to visualize one instance of a family of component-based systems.
- **Note**
- In many ways, a component diagram is just a special kind of class diagram that focuses on a system's components.

Component Diagrams

- Static implementation view represented by component diagrams primarily supports the configuration management of a system's parts, made up of components that can be assembled in various ways to produce a running system.
- When you model the static implementation view of a system, you'll typically use component diagrams in one of four ways.
- 1. To model source code
- 2. To model executable releases
- 3. To model physical database
- 4. To model adaptable systems

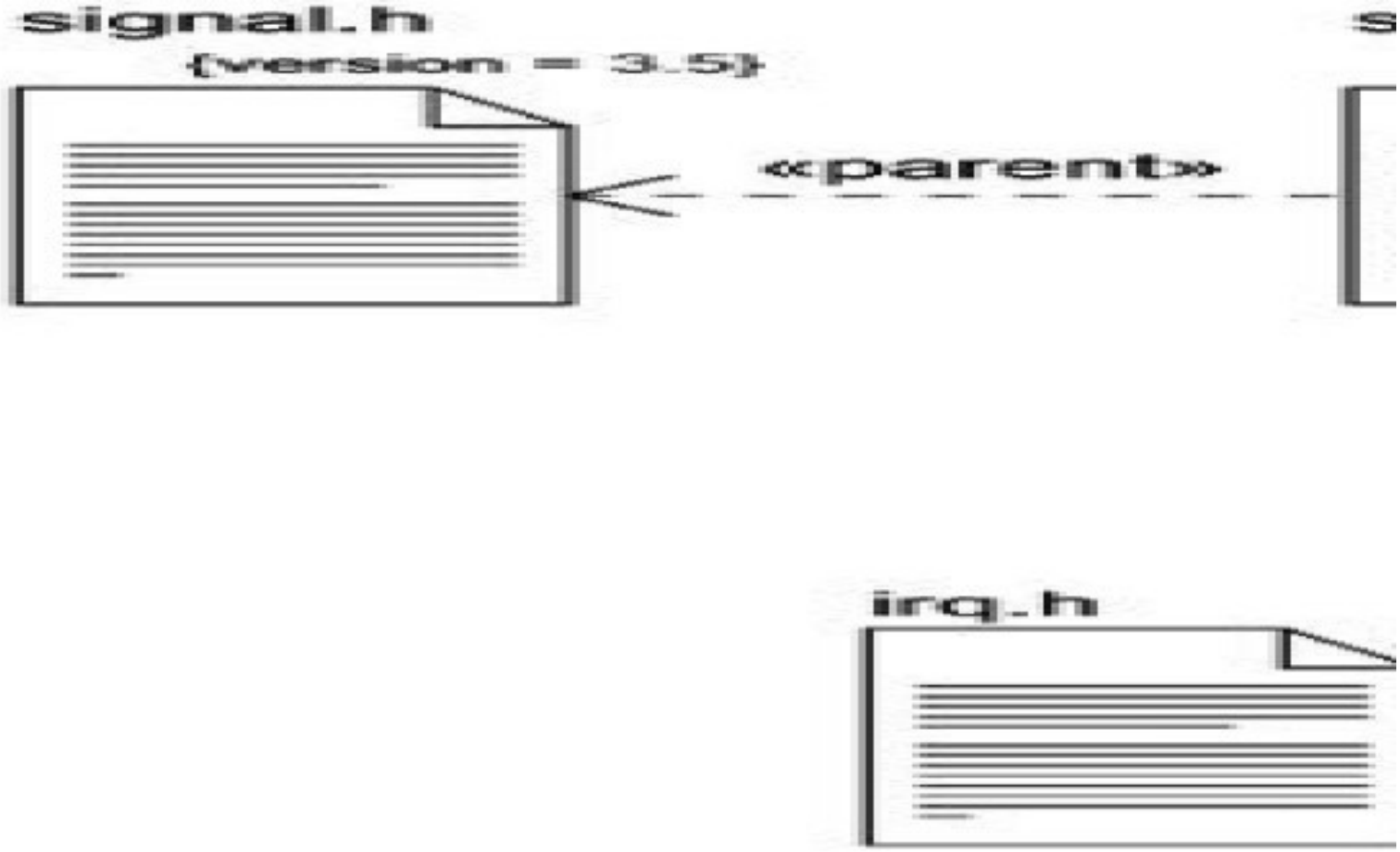
Component Diagrams

- **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 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

Component Diagrams

- **Modeling Source Code**
-

Figure 29-2 Modeling Source Code



Component Diagrams

- **Modeling an Executable Release**
- To model an executable release,
  - Identify the set of components you'd like to model. 1) some or all the components that live on one node, or 2) 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).
  - 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.

Component Diagrams

- **Modeling an Executable Release**
-

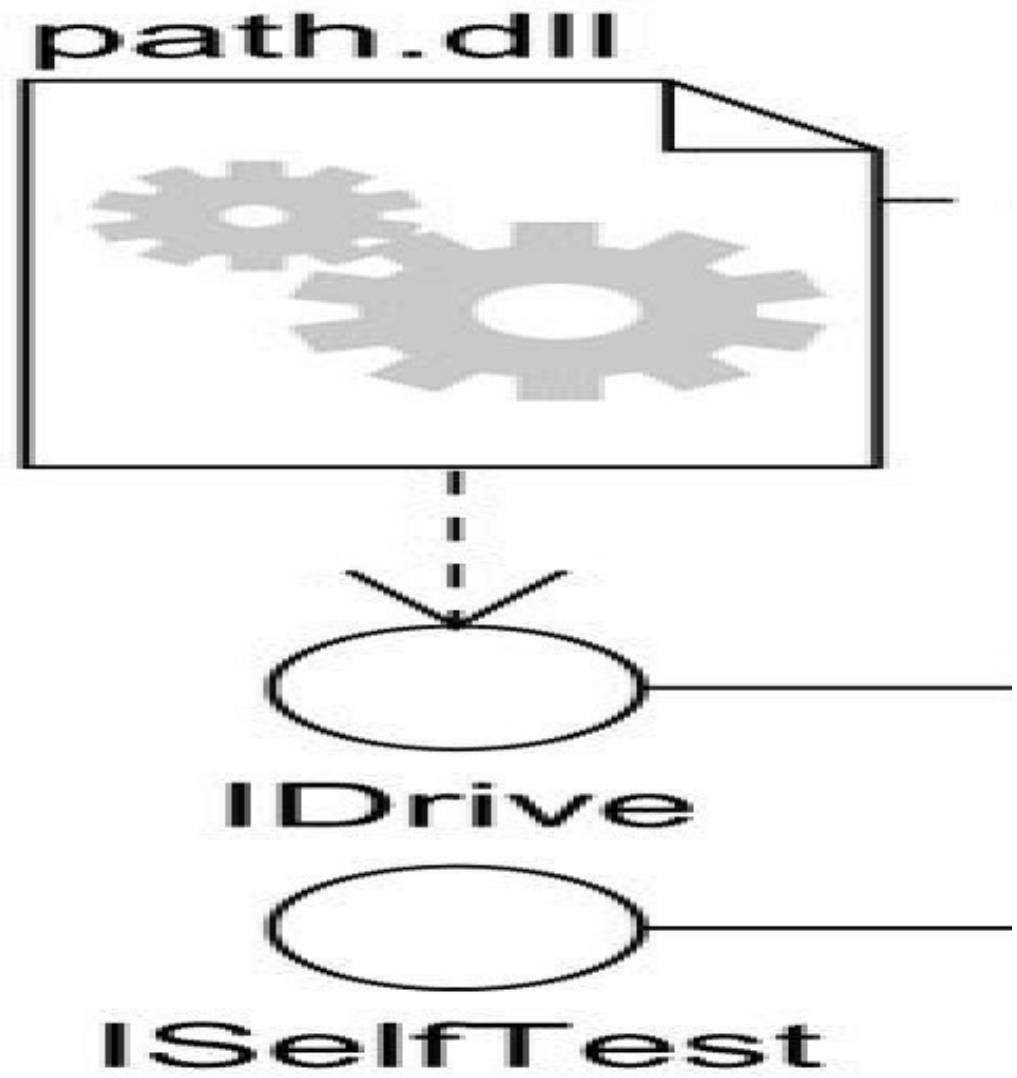


Figure 29-3 Modeling an Executable Release

Component Diagrams

- **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, an object-oriented one, or a hybrid object/relational database.
- The UML is well suited to modeling physical databases, as well as logical database schemas.
- Mapping a logical database schema to an object-oriented database is straightforward because even complex inheritance lattices can be made persistent directly.
- Mapping a logical database schema to a relational database is not so simple, however.
-

Component Diagrams

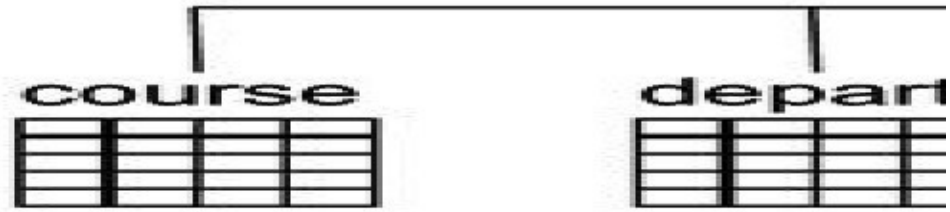
- In the presence of inheritance, you have to make decisions about how to map classes to tables. Typically, you can apply one or a combination of three strategies.
- 1. Define a separate table for each class. This is a simple but naive approach because it introduces maintenance headaches when you add 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.

Component Diagrams

- When designing a physical database, you also have to make decisions about how to map operations defined in your logical database schema.
- Object- oriented databases make the mapping fairly transparent.
- But, with relational databases, you have to make some decisions about how these logical operations are implemented. Again, you 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.

Component Diagrams

- Given these general guidelines, 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.



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.
- You use deployment diagrams to model the static deployment view of a system. For the most part, this involves 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 are not only important for visualizing, specifying, and documenting embedded, client/server, and distributed systems, but also for managing executable systems through forward and reverse engineering.*

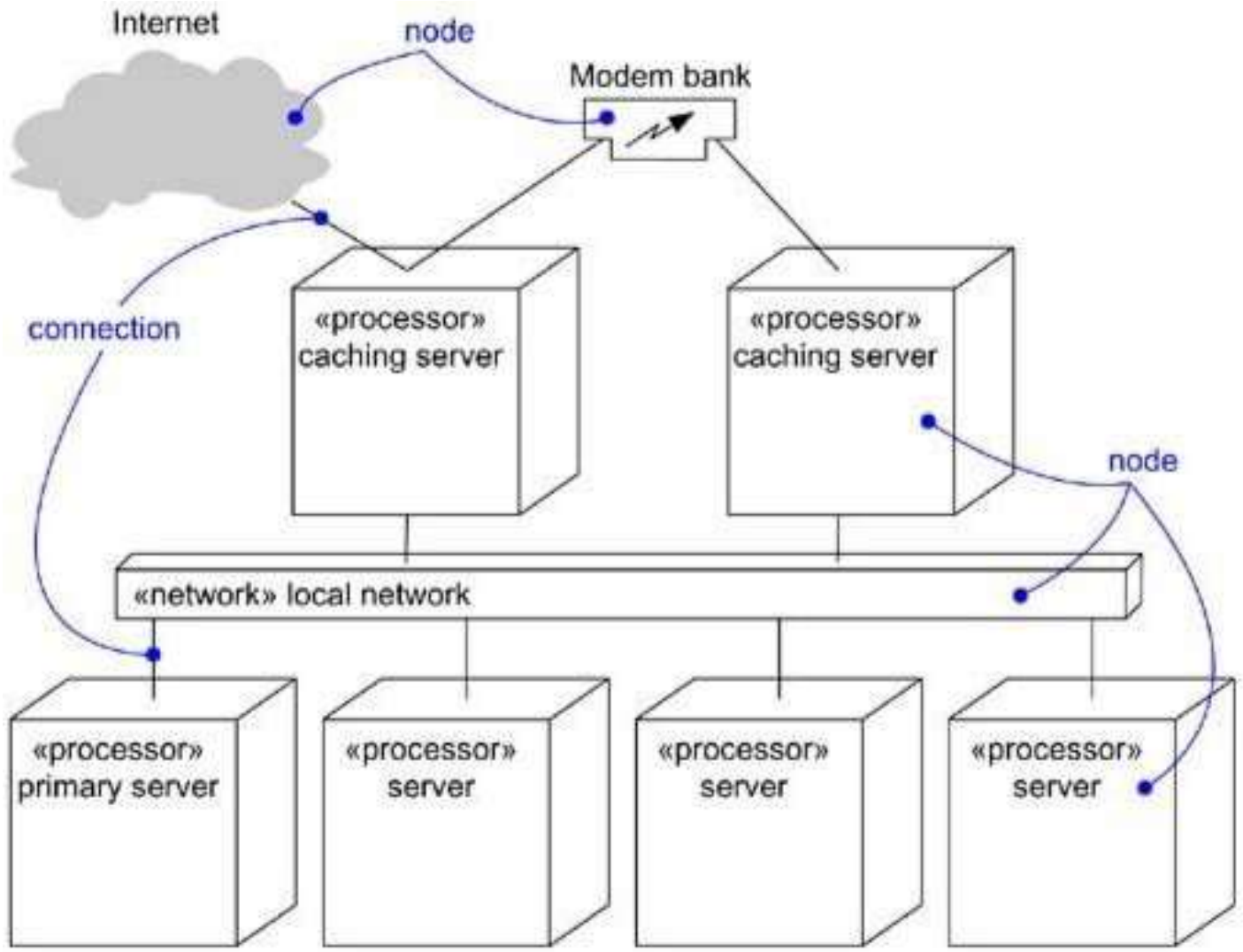
Deployment Diagrams

- The UML is designed to model many of the hardware aspects of a system sufficient for a software engineer to specify the platform on which the system's software executes and for a systems engineer to manage the system's hardware/software boundary.
- We use deployment diagrams to reason about the topology of processors and devices on which software executes.

Deployment Diagrams

- A deployment diagram is a diagram that shows the configuration of run time processing nodes and the components that live on them.
- Graphically, a deployment diagram is a collection of vertices and arcs.

Figure 30-1 A Deployment Diagram



Deployment Diagrams

- 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.
- Deployment diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place instances in your deployment diagrams, as well, especially when you want to visualize one instance of a family of hardware topologies.
- **Note**
- In many ways, a deployment diagram is just a special kind of class diagram, which focuses on a system's nodes.

Deployment Diagrams

- Deployment diagrams are used 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.
- There are some kinds of systems for which deployment diagrams are unnecessary.
- If you are developing a piece of software that lives on one machine and interfaces only with standard devices on that machine that are already managed by the host operating system (for example, a personal computer's keyboard, display, and modem), you can ignore deployment diagrams.
- On the other hand, if you are developing a piece of software that interacts with devices that the host operating system does not typically manage or that is physically distributed across multiple processors, then using deployment diagrams will help you reason about your system's software-to-hardware mapping.

Deployment Diagrams

- When you model the static deployment view of a system, you'll typically use deployment diagrams in one of three ways.
- 1. To model embedded systems
- 2. To model client/server systems
- 3. To model fully distributed systems

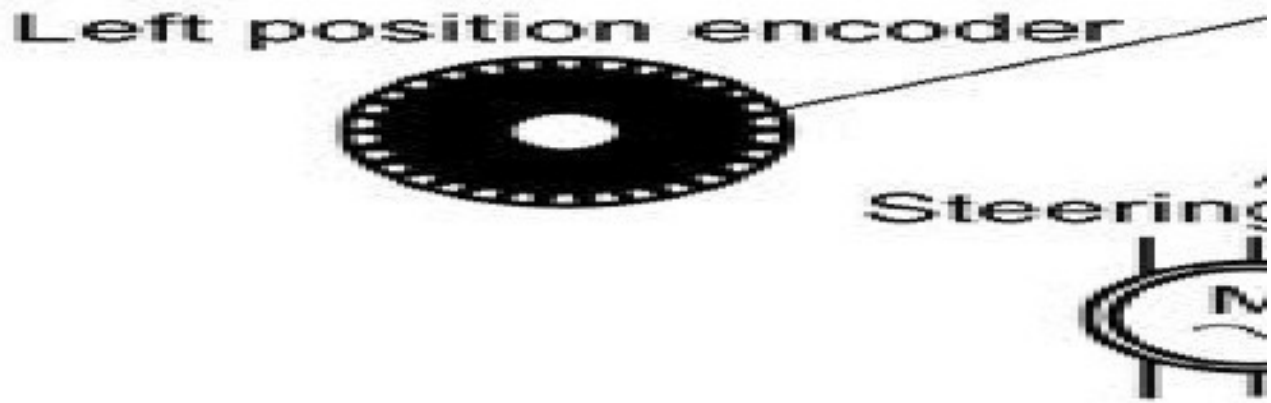
Deployment Diagrams

- **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.

Deployment Diagrams

- **Modeling an Embedded System**
-

Figure 30-2 Modeling an Embedded System



Deployment Diagrams

- **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.

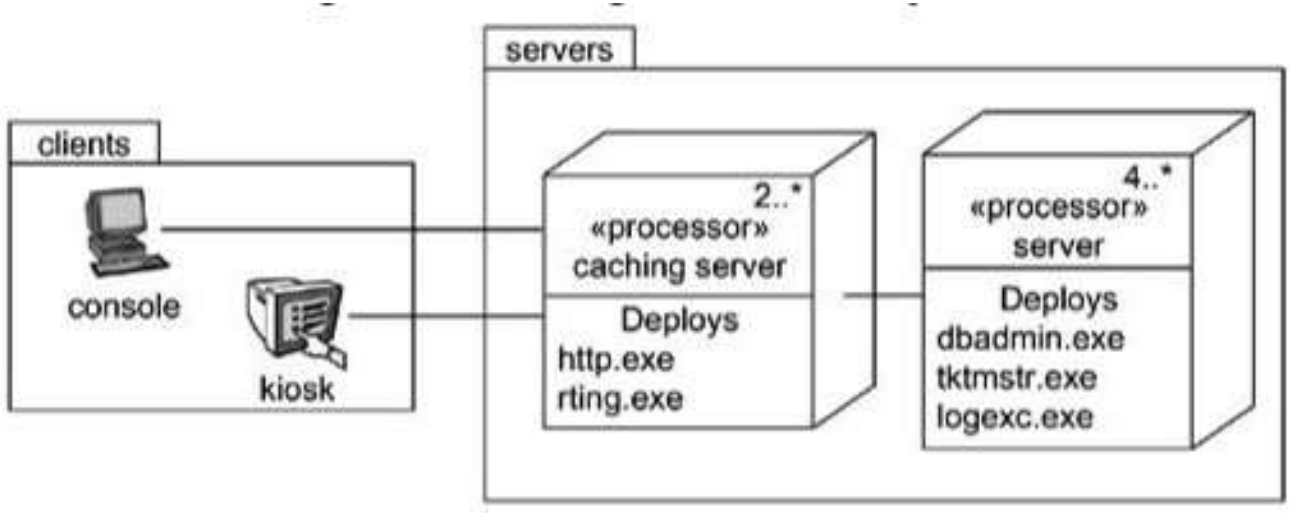


Figure 30-3
Modeling a Client/
Server System

Deployment Diagrams

- **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.

Deployment Diagrams

- **Modeling a Fully Distributed System**
-

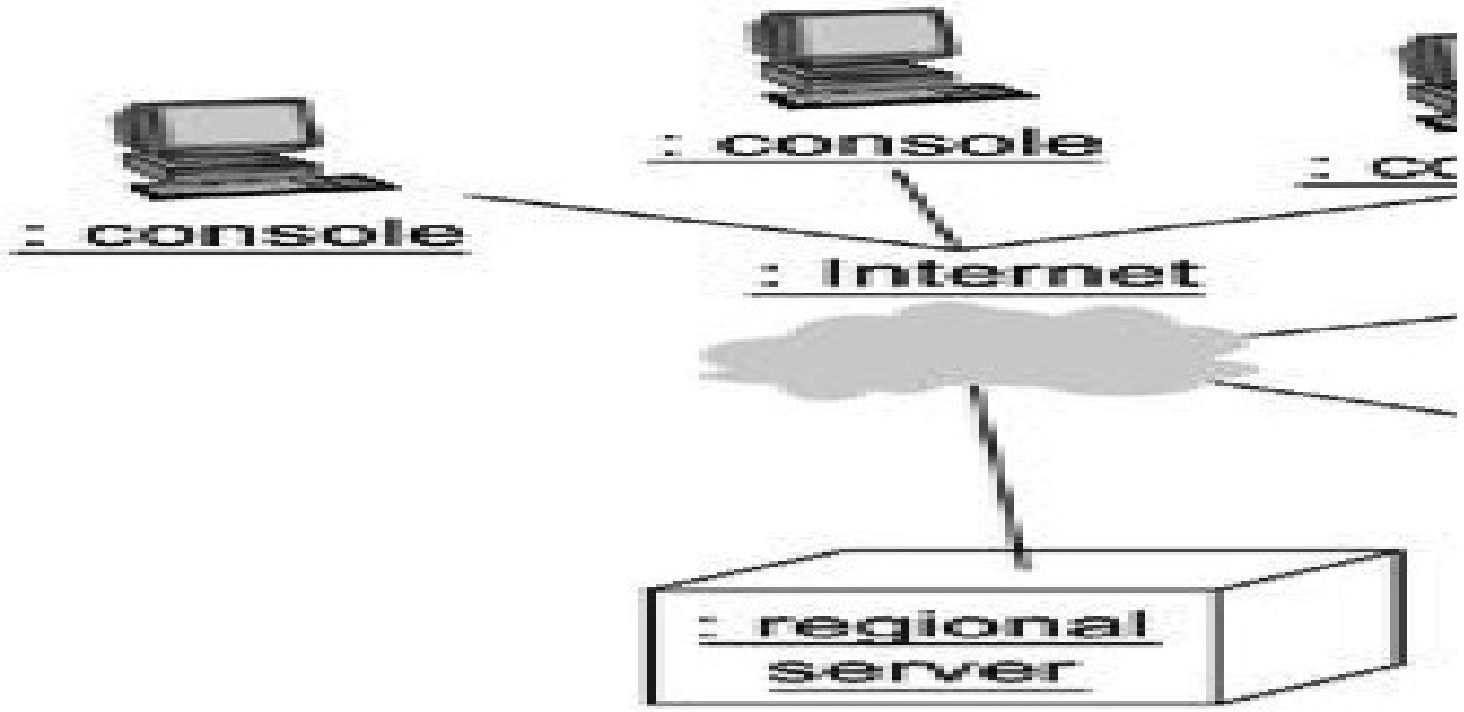


Figure 30-4 Modeling a Fully Distributed System