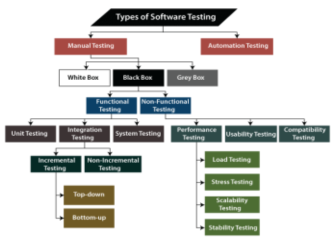
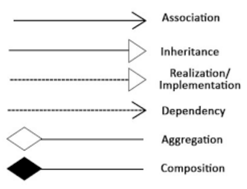
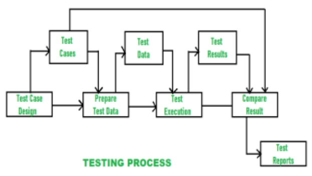
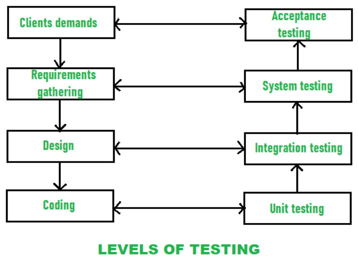






Software Design and Testing

Architectural Modeling



Outline

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

Patterns

- Software development process can be improved if a system can be analyzed, designed and built from prefabricated and predefined system components
- First prerequisite is to have a vocabulary for expressing its concepts and a language for relating them to each other
- Pattern is a body of literature to help software developers resolve common, difficult problem and a vocabulary for communicating insight and experience about problems and their solutions
- A pattern is an instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces

3

Patterns

- The design pattern identifies key aspects of a common design structure that make it useful for creating a reusable OO design
- It also identifies the participating classes and instances, their roles and collaborations and description of responsibilities
- The main idea behind creating and using patterns is to provide documentation to help, categorize and communicate solutions to recurring problems
- A "**pattern in waiting**", which is not yet known to recur, is called a proto-pattern
- The pattern has a name to facilitate discussion and the information it represents
- Its inappropriate to decisively call something a pattern until it has gone through peer scrutiny or review

4

Patterns

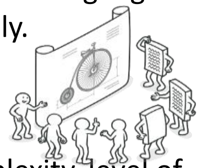
- Patterns are largely used for software architecture and design and also for other aspects of s/w development like s/w development process, requirement engineering, s/w configuration management, etc.
- Use of design patterns was originated by architect Christopher Alexander in 1070s
- A good pattern will do the following:
  - *It solves a problem*. Patterns capture solutions, not just abstract principles or strategies
  - *It is a proven concept*. Patterns capture solutions with a track record, not theories or speculation
  - *The solution is not obvious*. The best patterns generate a solution to a problem indirectly—a necessary approach for the most difficult problems of design

Patterns

  - *It describes a relationship*. Patterns do not just describe modules, but describe deeper system structures and mechanisms
  - *The pattern has a significant human component*. All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility
- Every pattern must be expressed “in the form of a rule (template)” which establishes a relationship between a context, forces affecting that context and a configuration, which allows these forces to resolve themselves in that context

Patterns

- **Design patterns** are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.
- **Benefits of patterns** are a toolkit of solutions to common problems in software design. They define a common language that helps your team communicate more efficiently.



- **Classification** Design patterns differ by their complexity, level of detail and scale of applicability. In addition, they can be categorized by their intent and divided into three groups.

7

Patterns Template

- Following are some components which should be clearly recognizable on reading a pattern
  - Name
    - Good pattern names form a vocabulary for discussing conceptual abstractions
    - Single word or short phrase can refer to pattern, knowledge or structure
    - Some patterns may have more than one commonly used names
  - Problem
    - What you really want to solve- or a statement of problem to describe its intent
    - The goals and objectives it wants to reach within given context and forces

8

Patterns Template

- Context
  - The preconditions under which the problem and its solution seem to recur and for which a solution is desirable
  - Hence define patterns applicability
  - One can use it to assume initial configuration of the system before the pattern is applied to it
- Forces
  - A description of relevant forces and constraints and how they interact with each other and with goals to be achieved
  - They reveal the intricacies of a problem and define the trade-offs that must be considered in the present of the tension they create
  - Good description should cover all forces that have impact on pattern

9

Patterns Template

- Solution
  - Static relationships and dynamic rules describing how to realize the desired outcome
  - Description may include pictures, diagrams etc. to show how the problem is solved
  - The description may indicate guideline to keep in mind while implementing the concrete implementation of the solution
- Examples
  - One or more sample applications of pattern that shows initial context, how pattern is applied and transforms that context to resulting context is given
  - Helps in understanding use of patterns
  - Examples can also be supplemented by sample implementations to show how solution is realized

10

Patterns Template

- Resulting Context
  - The state or configuration of the system after the pattern has been applied, its consequences, and other problems and patterns that may arise
  - It describes the post conditions and side effects of the patterns
  - Documenting the resulting context helps you in identifying initial context of other patterns
- Rationale
  - A step-by-step description of the rules in the pattern in terms of how and why it resolves it forces in order to achieve the desired goals, obey principles and philosophies
  - It explains how the forces and constraints can be orchestrated in concert to achieve resonant harmony

Patterns Template

- Related Patterns
  - The static and dynamic relationships between this pattern and others within the same pattern language
  - Related patterns often share common forces
  - It has an initial or resulting context that is compatible to resulting and initial context of other patterns
  - Such patterns might be
    - The predecessor patterns whose application leads to this pattern
    - Successor patterns whose applications follows from this pattern
    - Alternative patterns that describe a different solution to same problem under different forces and constraints

Patterns Template

- Known Uses
  - The known occurrences and applications within the existing system
  - So you can validate a pattern by verifying that it indeed is a proven solution to a recurring problem

13

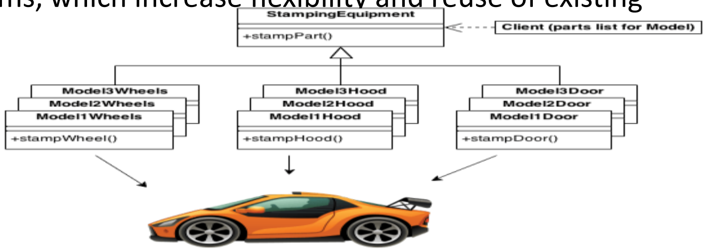
Design Patterns

- Creational Design patterns
- Structural design patterns
- Behavioral design patterns

14

Creational Design Patterns

- Creational patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.



1. Factory Method : Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

2. Abstract Factory : Lets you produce families of related objects without specifying their concrete classes.

Creational Design Patterns

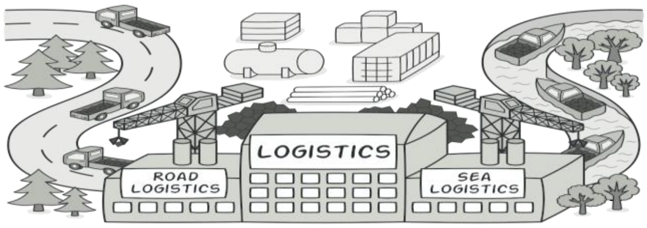
3. Prototype : Lets you copy existing objects without making your code dependent on their classes.

4. Singleton : Lets you ensure that a class has only one instance, while providing a global access point to this instance

5. Builder : Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

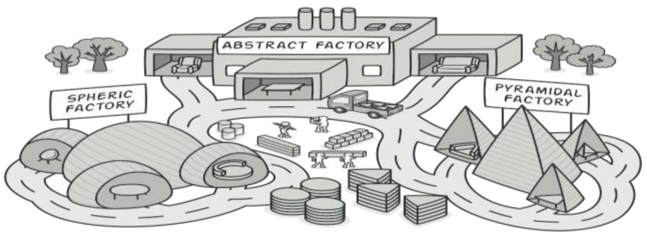
Factory Method

- It is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
- Problem : Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the Truck class.
- After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.

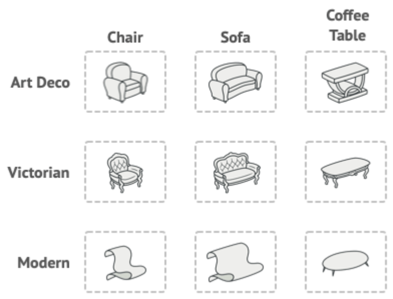


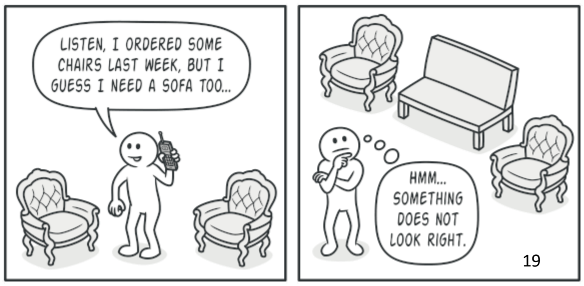
Abstract Factory Method

- It is a creational design pattern that lets you produce families of related objects without specifying their concrete classes
- Problem : Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:
  - A family of related products, say: Chair + Sofa + CoffeeTable.
  - Several variants of this family. For example, products Chair + Sofa + CoffeeTable are available in these variants: Modern, Victorian, ArtDeco



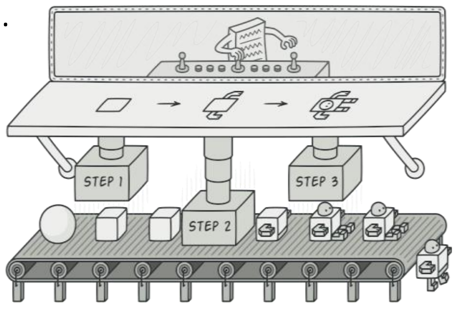
Abstract Factory Method




LISTEN, I ORDERED SOME CHAIRS LAST WEEK, BUT I GUESS I NEED A SOFA TOO...
HMM... SOMETHING DOES NOT LOOK RIGHT.

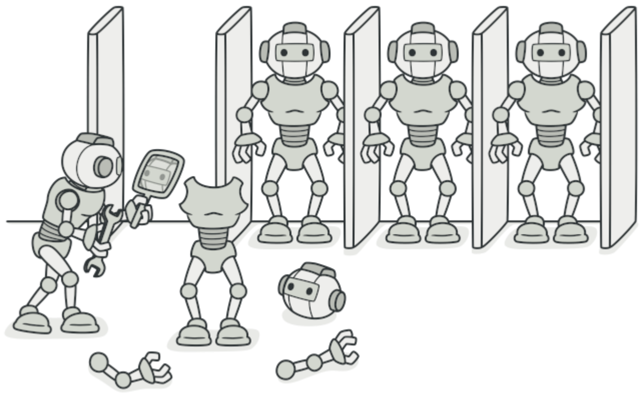
Builder

- **Builder** is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



Prototype

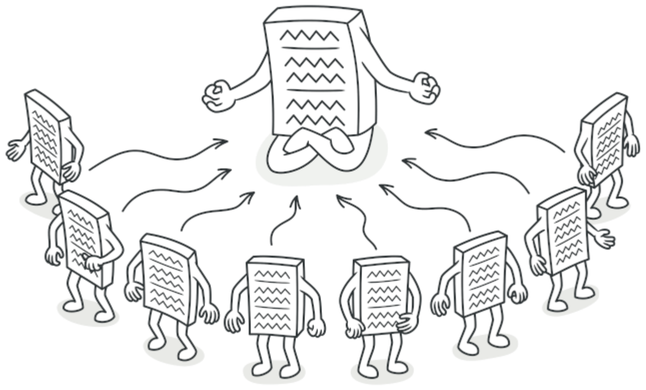
- **Prototype** is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.



21

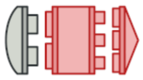
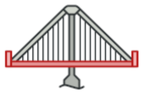
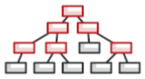
Singleton

- **Singleton** is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance



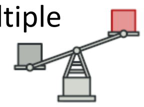
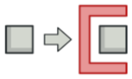
22

Structural Design Patterns

- Structural patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.
- Adapter : Allows objects with incompatible interfaces to collaborate. 
- Bridge :Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other. 
- Composite : Lets you compose objects into tree structures and then work with these structures as if they were individual objects 

23

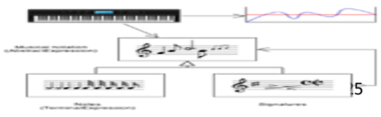
Structural Design Patterns

- Decorator :Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors
- Facade : Provides a simplified interface to a library, a framework, or any other complex set of classes.
- Flyweight : Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object 
- Proxy : Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object 

24

Behavioral Design Patterns

- Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects.
- Chain of Responsibility :Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.
- Command :Turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations

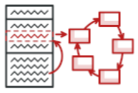


Behavioral Design Patterns

- Iterator :Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).
- Mediator :Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.
- Memento :Lets you save and restore the previous state of an object without revealing the details of its implementation
- Observer :Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing

Behavioral Design Patterns

- State :Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



- Strategy :Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

- Template Method :Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

- Visitor :Lets you separate algorithms from the objects on which they operate.

Frame Works

- A way of delivering application development patterns to support best practice sharing during application development- through an emerging framework market
- A *framework* is a way of presenting a generic solution to a problem that can be applied to all levels in a development
- A single framework typically encompasses several design patterns and can be viewed as the implementation of a system of design patterns
- An experienced programmer almost never codes a new program from scratch- but uses macros, copy libraries, and template like code fragments to make a start. New work begins by filling in new domain-specific data inside older structures.
- A seasoned business consultant who has worked on many consulting projects for data modeling almost never builds new models from scratch- but new domain specific terms will be substituted in his library models

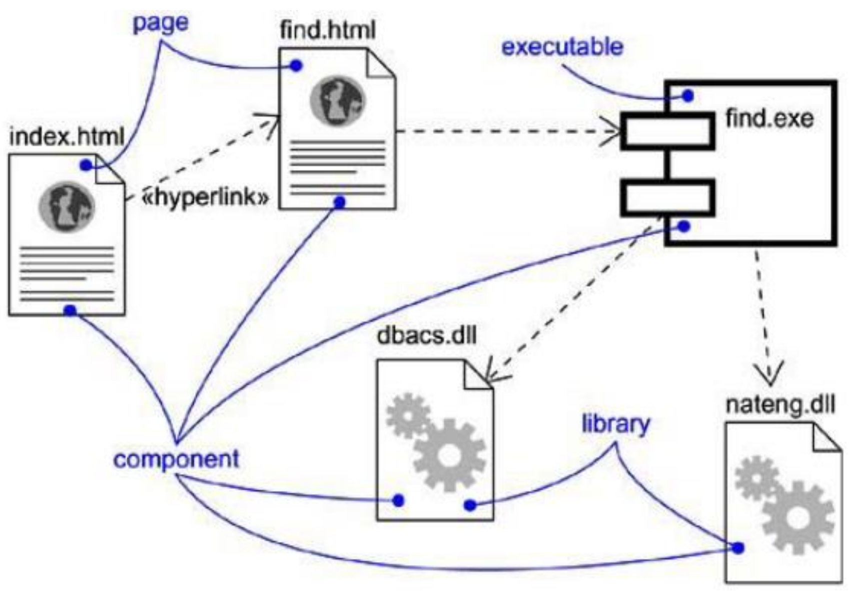
DIFFERENCES BETWEEN DESIGN PATTERNS AND Frame Works

- *Design patterns are more abstract than frameworks :* Frameworks can be embodied in code, but only examples of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly. In contrast, design patterns have to be implemented each time they are used.
- *Design patterns are smaller architectural elements than frameworks* : A typical framework contains several design patterns but the reverse is never true.
- *Design patterns are less specialized than frameworks :* Frameworks always have a particular application domain. In contrast, design patterns can be used in nearly any kind of application. While more specialized design patterns are certainly possible, even these would not dictate an application architecture.

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.
- 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.

Component Diagrams



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.

32

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.
- Component diagrams commonly contain : Components , Interfaces , Dependency, generalization, association, and realization relationships

Component Diagrams

- 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.
- When you model the static implementation view of a system, you'll typically use component diagrams in one of four ways.
  - To model source code
  - To model executable releases
  - To model physical database
  - To model adaptable systems

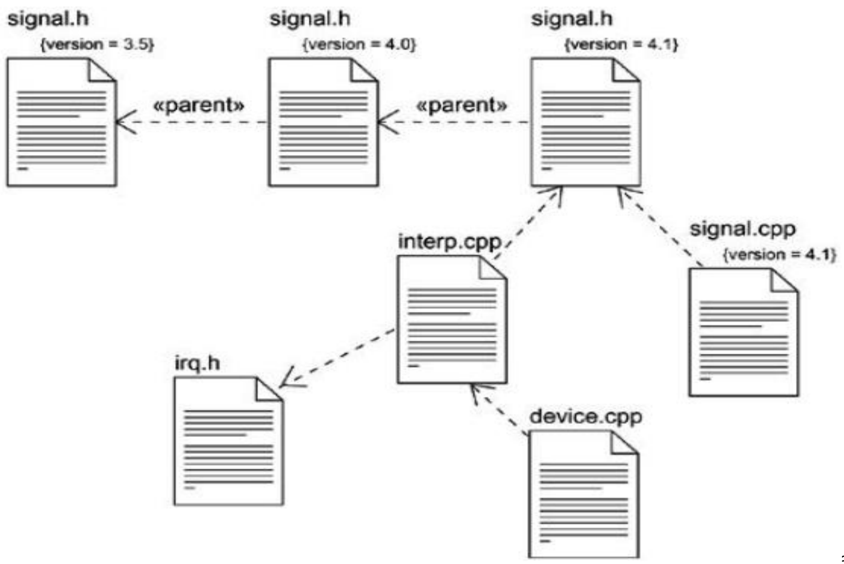
Component Diagrams

- **Modeling 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

35

Component Diagrams

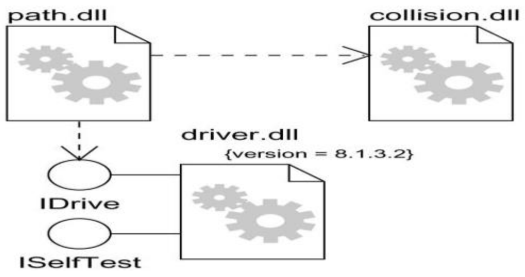
- **Modeling Source Code**



5

Component Diagrams

- **Modeling an Executable Release**
  - Identify the set of components you'd like to model.
    - 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.
  - 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.



37

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.
  - 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.
  - 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.

38

Component Diagrams

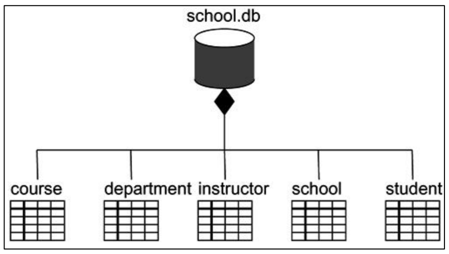
- **Modeling a Physical Database**
  1. Define a separate table for each class. This is a simple but naive approach because it introduces maintenance headaches when you add new child classes or modify your parent classes.
  2. Collapse your inheritance lattices so that all instances of any class in a 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.

39

Component Diagrams

- **Modeling a Physical Database**
  - With relational databases, you have to make some decisions about how these logical operations are implemented. Again, you have some choices.

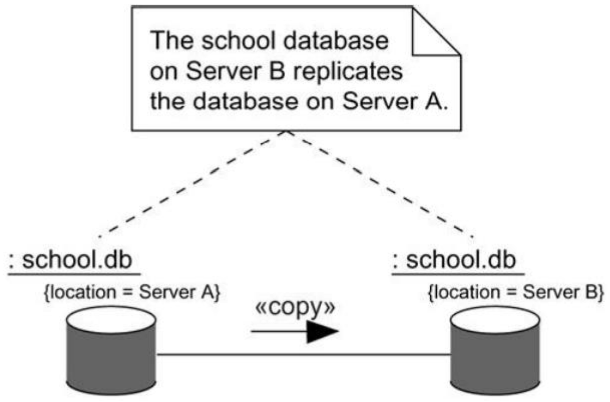
  1. For simple CRUD (create, read, update, delete) operations, implement them with standard SQL or ODBC calls.
  2. For more-complex behavior (such as business rules), map them to triggers or stored procedures.



40

Component Diagrams

- **Modelling Adaptable Systems**
  - For example, you might have a system that replicates its databases across several nodes, switching the one that is the primary database when a server goes down.



41

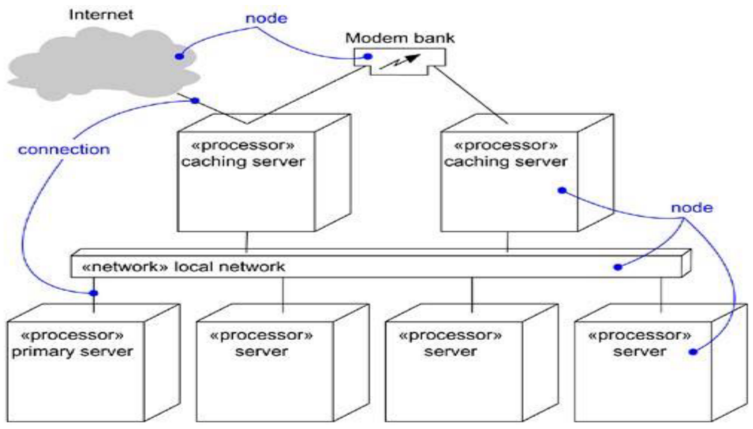
Deployment Diagrams

- 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.
- 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.

42

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.



Deployment Diagrams

- Deployment diagrams commonly contain : **Nodes & Dependency and association relationships.**
- Like all other diagrams, deployment diagrams may contain notes, constraints, packages or subsystems, hardware topologies and components that live on node.
- **Note** : In many ways, a deployment diagram is just a special kind of class diagram, which focuses on a system's nodes.
- 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.

Deployment Diagrams

- 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

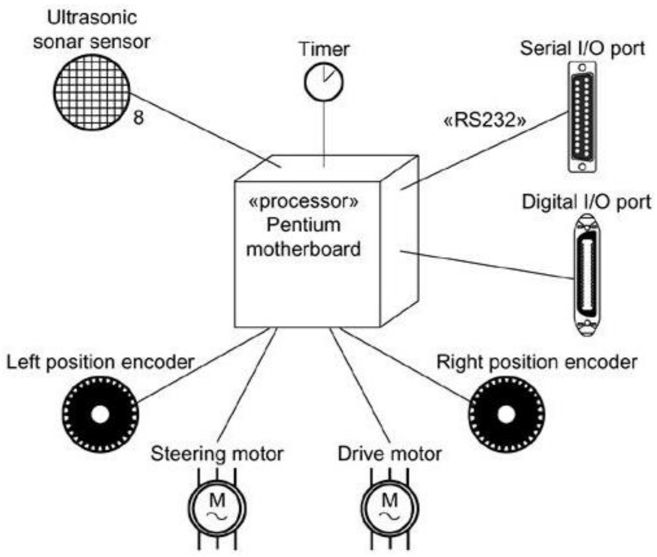
- When you model the static deployment view of a system, you'll typically use deployment diagrams in one of three ways.
  - To model embedded systems
  - To model client/server systems
  - To model fully distributed systems

Deployment Diagrams

- **To model embedded systems**
  - 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. 47

Deployment Diagrams

- **To model embedded systems**



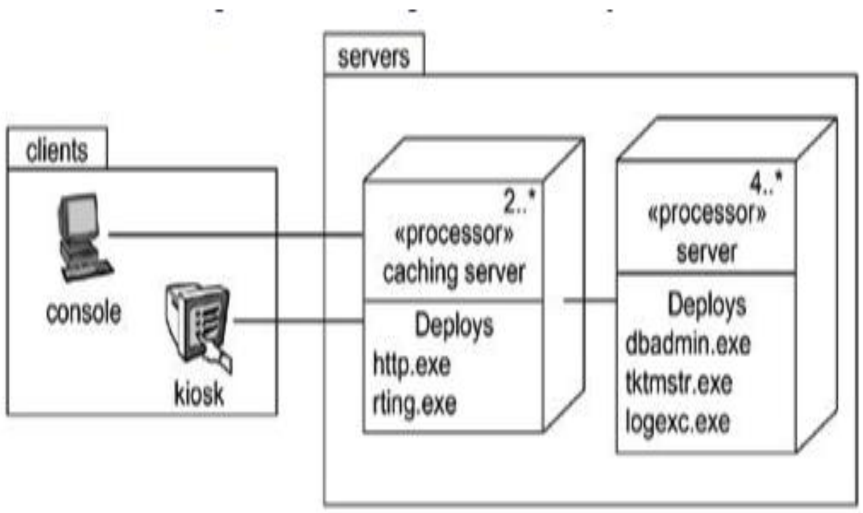
48

Deployment Diagrams

- **Modeling a Client/Server System**
  - Identify the nodes that represent your system's client and server processors.
  - Highlight those devices that are applicable 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.

Deployment Diagrams

- **Modeling a Client/Server System**



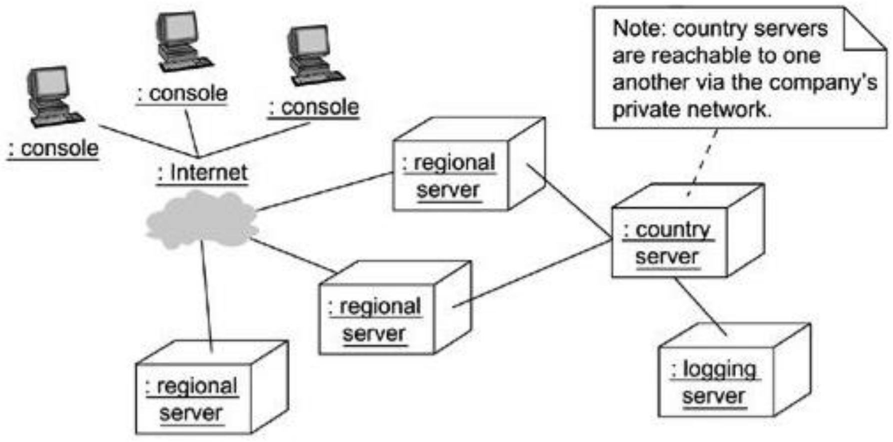
Deployment Diagrams

- **Modeling 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.

51

Deployment Diagrams

- **Modeling a Fully Distributed System**



52

53