UNIT-IV

UML stands for Unified Modeling Language. It is a pictorial language used to make software blueprints. UML was created by Object Management Group (OMG). The UML 1.0 specification draft was proposed to the OMG in January 1997. It serves as a standard for software requirement analysis and design documents which are the basis for developing a software.

UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document a software system. Although UML is generally used to model software system, it is not limited within this boundary. It is also used to model non software systems such as process flows in a manufacturing unit.

The elements are like components which can be associated in different ways to make a complete UML picture, which is known as a **diagram**. So, it is very important to understand the different diagrams to implement the knowledge in real-life systems. We have two broad categories of diagrams and they are further divided into sub-categories i.e. **Structural Diagrams** and **Behavioral Diagrams**.

Structural Diagrams

Structural diagrams represent the static aspects of a system. These static aspects represent those parts of a diagram which forms the main structure and is therefore stable.

These static parts are represented by classes, interfaces, objects, components and nodes. Structural diagrams can be sub-divided as follows −

- Class diagram
- Object diagram
- Component diagram
- Deployment diagram
- Package diagram
- Composite structure

The following table provides a brief description of these diagrams −

Sr.No.	Diagram & Description
1	**Class** Represents the object orientation of a system. Shows how classes are statically related.
2	**Object** Represents a set of objects and their relationships at runtime and also represent the static view of the system.
3	**Component** Describes all the components, their interrelationship, interactions and interface of the system.
4	**Composite structure** Describes inner structure of component including all classes, interfaces of the component, etc.
5	**Package** Describes the package structure and organization. Covers classes in the package and packages within another package.
6	**Deployment** Deployment diagrams are a set of nodes and their relationships. These nodes are physical entities where the components are deployed.

Behavioral Diagrams

Behavioral diagrams basically capture the dynamic aspect of a system. Dynamic aspects are basically the changing/moving parts of a system. UML has the following types of behavioral diagrams −

- Use case diagram
- Sequence diagram
- Communication diagram
- State chart diagram
- Activity diagram

- Interaction overview
- Time sequence diagram

The following table provides a brief description of these diagram −

Sr.No.	Diagram & Description
1	**Use case** Describes the relationships among the functionalities and their internal/external controllers. These controllers are known as actors.
2	**Activity** Describes the flow of control in a system. It consists of activities and links. The flow can be sequential, concurrent, or branched.
3	**State Machine/state chart** Represents the event driven state change of a system. It basically describes the state change of a class, interface, etc. Used to visualize the reaction of a system by internal/external factors.
4	**Sequence** Visualizes the sequence of calls in a system to perform a specific functionality.
5	**Interaction Overview** Combines activity and sequence diagrams to provide a control flow overview of system and business process.
6	**Communication**

	Same as sequence diagram, except that it focuses on the object’s role. Each communication is associated with a sequence order, number plus the past messages.
7	**Time Sequenced** Describes the changes by messages in state, condition and events.

Architecture View Model

A model is a complete, basic, and simplified description of software architecture which is composed of multiple views from a particular perspective or viewpoint.

A view is a representation of an entire system from the perspective of a related set of concerns. It is used to describe the system from the viewpoint of different stakeholders such as end-users, developers, project managers, and testers.

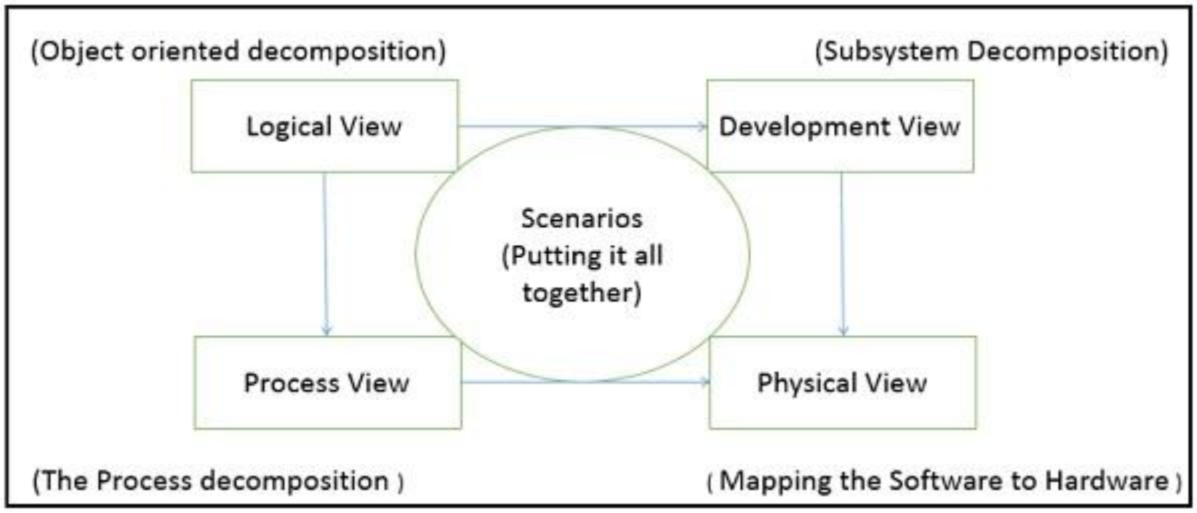
4+1 View Model

The 4+1 View Model was designed by Philippe Kruchten to describe the architecture of a software–intensive system based on the use of multiple and concurrent views. It is a multiple view model that addresses different features and concerns of the system. It standardizes the software design documents and makes the design easy to understand by all stakeholders.

It is an architecture verification method for studying and documenting software architecture design and covers all the aspects of software architecture for all stakeholders. It provides four essential views −

- **The logical view or conceptual view** − It describes the object model of the design.
- **The process view** − It describes the activities of the system, captures the concurrency and synchronization aspects of the design.
- **The physical view** − It describes the mapping of software onto hardware and reflects its distributed aspect.
- **The development view** − It describes the static organization or structure of the software in its development of environment.

This view model can be extended by adding one more view called **scenario view** or **use case view** for end-users or customers of software systems. It is coherent with other four views and are utilized to illustrate the architecture serving as “plus one” view, (4+1) view model. The following figure describes the software architecture using five concurrent views (4+1) model.


(Object oriented decomposition)
(Subsystem Decomposition)
Logical View
Development View
Scenarios (Putting it all together)
Process View
Physical View
(The Process decomposition)
(Mapping the Software to Hardware)

Why is it called 4+1 instead of 5?

The **use case view** has a special significance as it details the high level requirement of a system while other views details — how those requirements are realized. When all other four views are completed, it’s effectively redundant. However, all other views would not be possible without it. The following image and table shows the 4+1 view in detail −

	Logical	**Process**	**Development**	**Physical**	**Scenario**
Description	Shows the component (Object) of system as well as their interaction	Shows the processes / Workflow rules of system and how those processes communicate, focuses on dynamic view of system	Gives building block views of system and describe static organization of the system modules	Shows the installation, configuration and deployment of software application	Shows the design is complete by performing validation and illustration

Viewer / Stake holder	End-User, Analysts and Designer	Integrators & developers	Programmer and software project managers	System engineer, operators, system administrators and system installers	All the views of their views and evaluators
Consider	Functional requirements	Non Functional Requirements	Software Module organization (Software management reuse, constraint of tools)	Nonfunctional requirement regarding to underlying hardware	System Consistency and validity
UML – Diagram	Class, State, Object, sequence, Communication Diagram	Activity Diagram	Component, Package diagram	Deployment diagram	Use case diagram

Deployment or Physical View (Mapping Software to Hardware)

- This view encompasses the nodes that form the system's hardware topology on which the system executes; it focuses on distribution, communication and provisioning.
- The software executes on a network of computers, or processing nodes.
- The various elements such as processes, tasks and objects need to be mapped to the nodes on which they execute.
- These physical configurations can differ between production, development and testing environments.
- The software should be built to be flexible to scale across these hardware changes. Hence, this view accommodates the non-functional requirements such as availability, reliability, performance, throughput and scalability.

Benefits of component diagrams

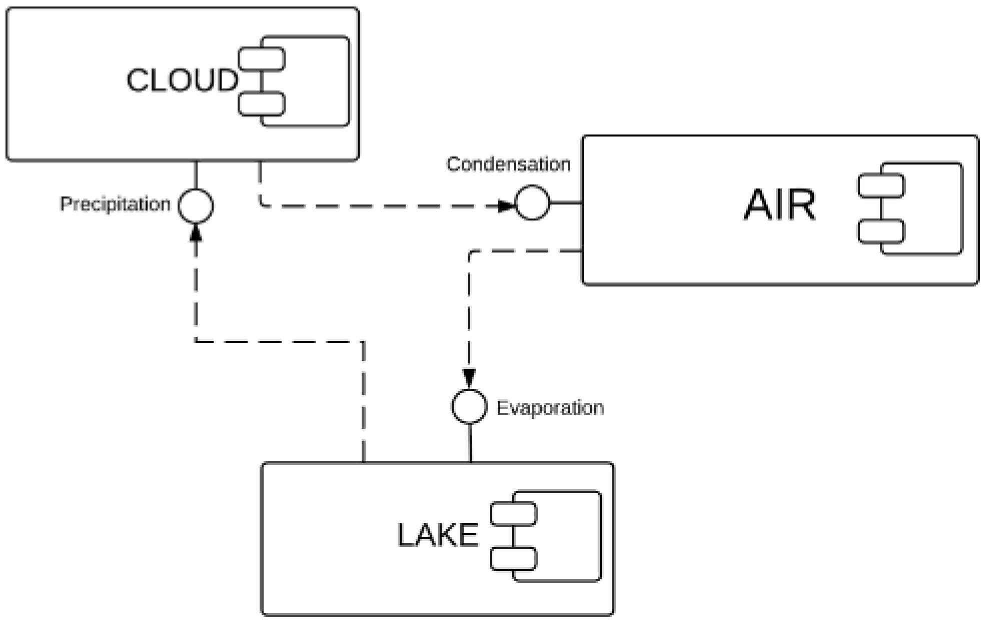
Though component diagrams may seem complex at first glance, they are invaluable when it comes to building your system. Component diagrams can help your team:

- Imagine the system’s physical structure.
- Pay attention to the system’s components and how they relate.
- Emphasize the service behavior as it relates to the interface.

How to use component diagrams

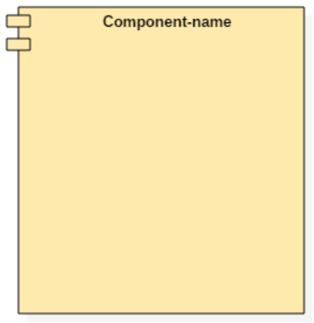
A component diagram in UML gives a bird’s-eye view of your software system. Understanding the exact service behavior that each piece of your software provides will make you a better developer. Component diagrams can describe software systems that are implemented in any programming language or style.

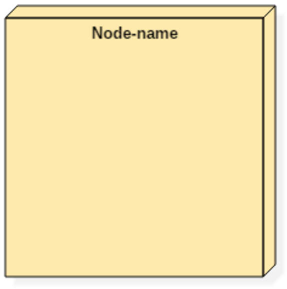
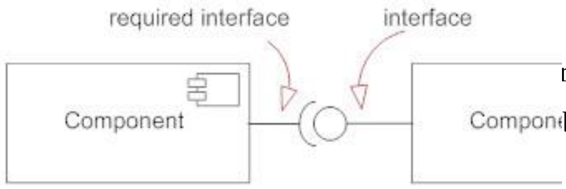
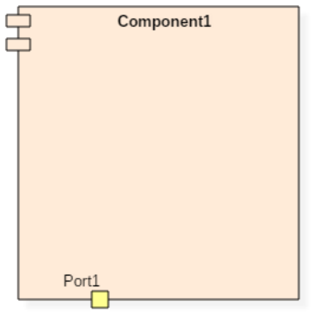
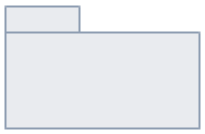
UML is a set of conventions for object-oriented diagrams that has a wide variety of applications. In component diagrams, the Unified Modeling Language dictates that components and packages are wired together with lines representing assembly connectors and delegation connectors. To learn more about UML and its uses, check out our guide, "What Is UML?"

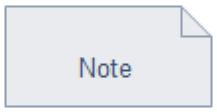
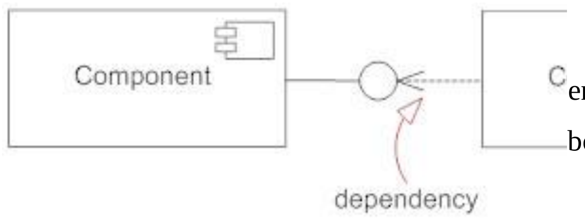


Component diagram shapes and symbols

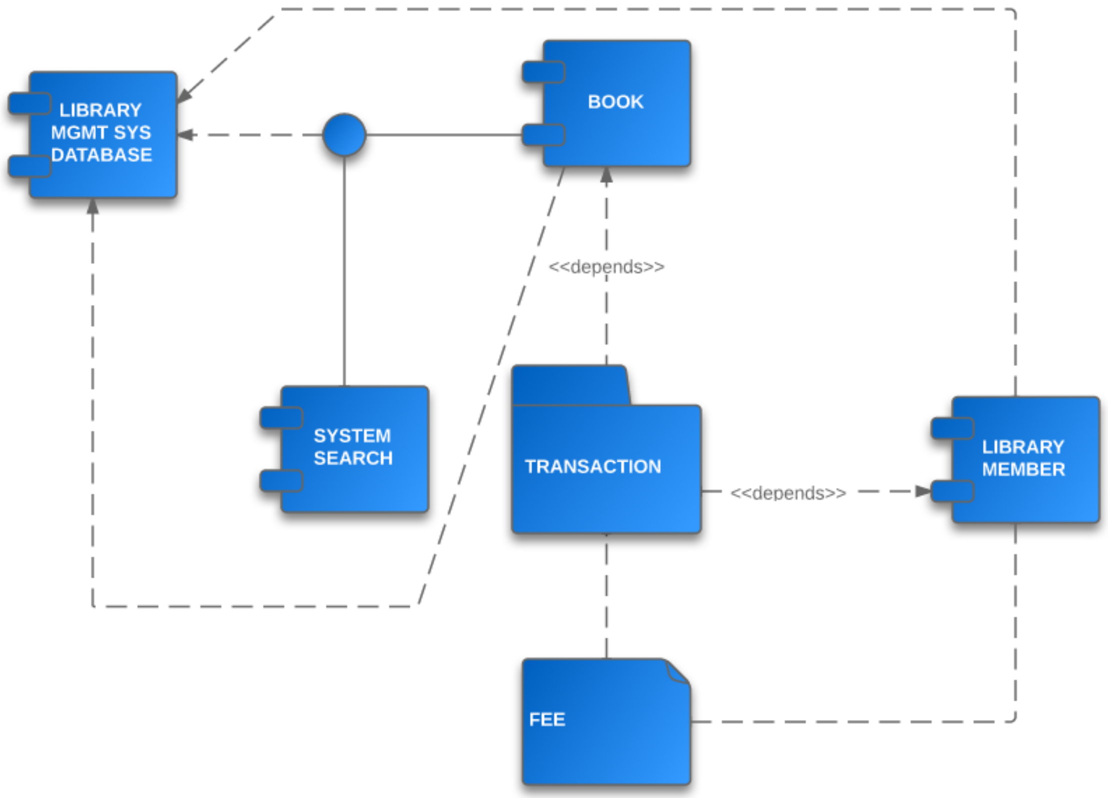
Component diagrams range from simple and high level to detailed and complex. Either way, you'll want to familiarize yourself with the appropriate UML symbols. The following are shape types that you will commonly encounter when reading and building component diagrams:

Symbol	Name	Description
	Component symbol	An entity required to execute a stereotype function. A component provides and consumes behavior through interfaces, as well as through other components. Think of components as a type of class. In UML 1.0, a component is modeled as a rectangular block with two smaller rectangles protruding from the side. In UML 2.0, a component is modeled as a rectangular block with a small image of the old component diagram shape.

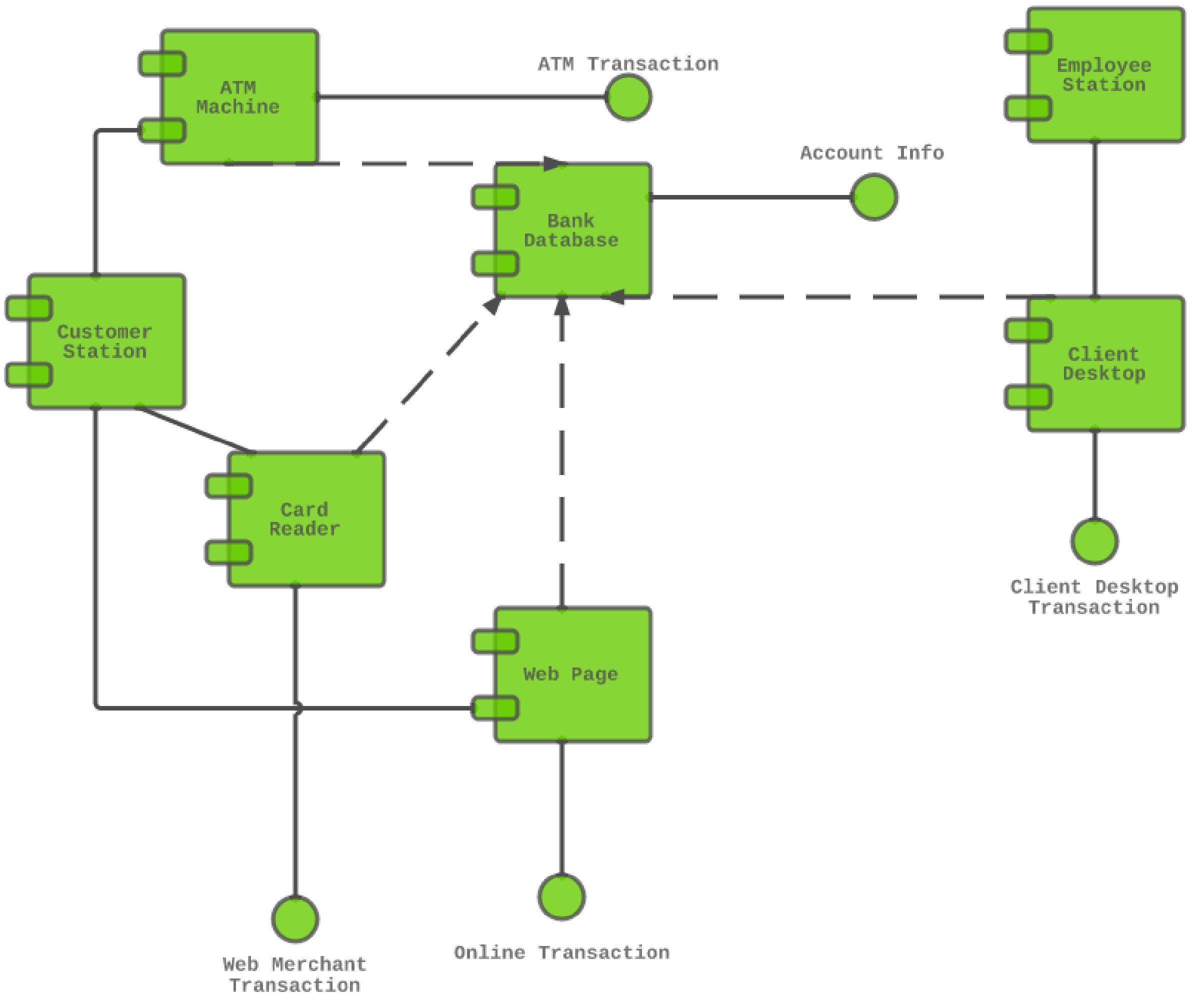
Symbol	Name	Description
	Node symbol	Represents hardware or software objects, which are of a higher level than components.
	rface bol	Shows input or materials that a component either receives or provides. Interfaces can be represented with textual notes or symbols, such as the lollipop, socket, and ball-and-socket shapes.
	Port symbol	Specifies a separate interaction point between the component and the environment. Ports are symbolized with a small square.
	Package symbol	Groups together multiple elements of the system and is represented by file folders in Lucidchart. Just as file folders group together multiple sheets, packages can be drawn around several components.

Symbol	Name	Description
	Note symbol	Allows developers to affix a meta-analysis to the component diagram.
	endency bol	Shows that one part of your system depends on another. Dependencies are represented by dashed lines linking one component (or element) to another.

Component diagram for a library management system



LIBRARY MGMT SYS DATABASE
BOOK
<<depends>>
SYSTEM SEARCH
TRANSACTION
<<depends>>
LIBRARY MEMBER
FEE


ATM Machine
ATM Transaction
Employee Station
Account Info
Bank Database
Customer Station
Client Desktop
Card Reader
Client Desktop Transaction
Web Page
Web Merchant Transaction
Online Transaction

DEPLOYMENT DIAGRAM

A UML deployment diagram is a diagram that shows the configuration of run time processing nodes and the components that live on them. Deployment diagrams is a kind of structure diagram used in modeling the physical aspects of an object-oriented system. They are often be used to model the static deployment view of a system (topology of the hardware).

Purpose of Deployment Diagrams

- They show the structure of the run-time system
- They capture the hardware that will be used to implement the system and the links between different items of hardware.
- They model physical hardware elements and the communication paths between them
- They can be used to plan the architecture of a system.
- They are also useful for Document the deployment of software components or nodes

Deployment Diagram at a Glance

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.

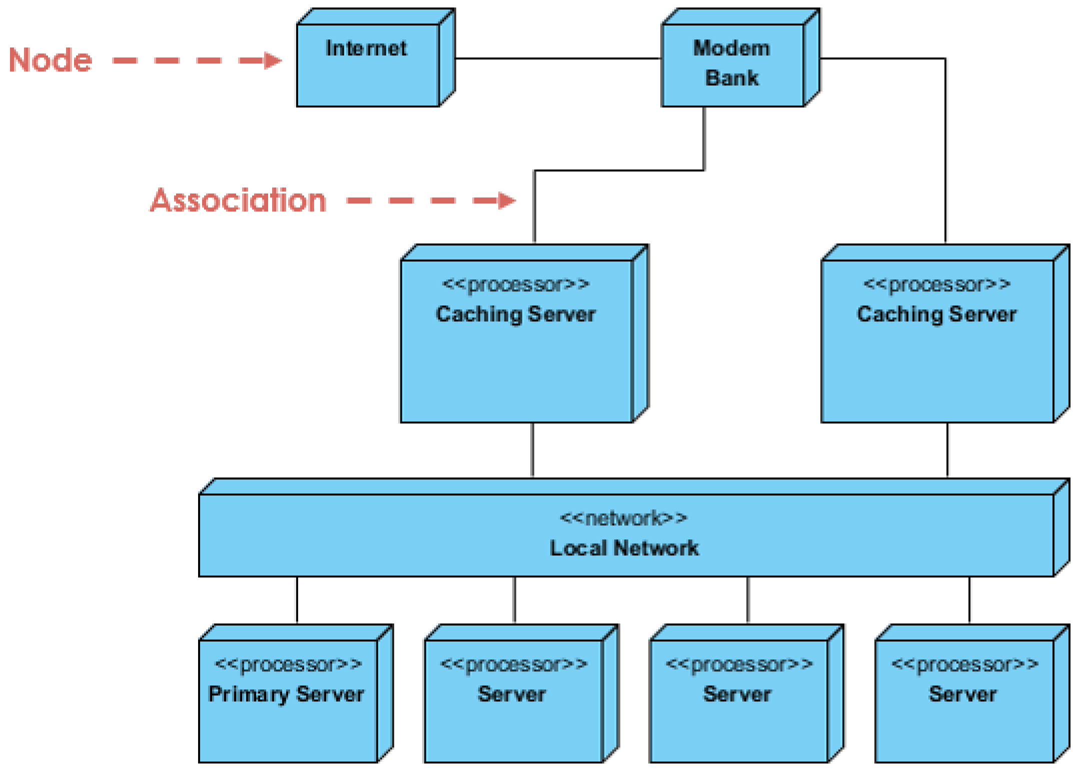
A deployment diagram is just a special kind of class diagram, which focuses on a system's nodes. Graphically, a deployment diagram is a collection of vertices and arcs. Deployment diagrams commonly contain:

Nodes

- 3-D box represents a node, either software or hardware
- HW node can be signified with <<stereotype>>
- Connections between nodes are represented with a line, with optional <<stereotype>>
- Nodes can reside within a node

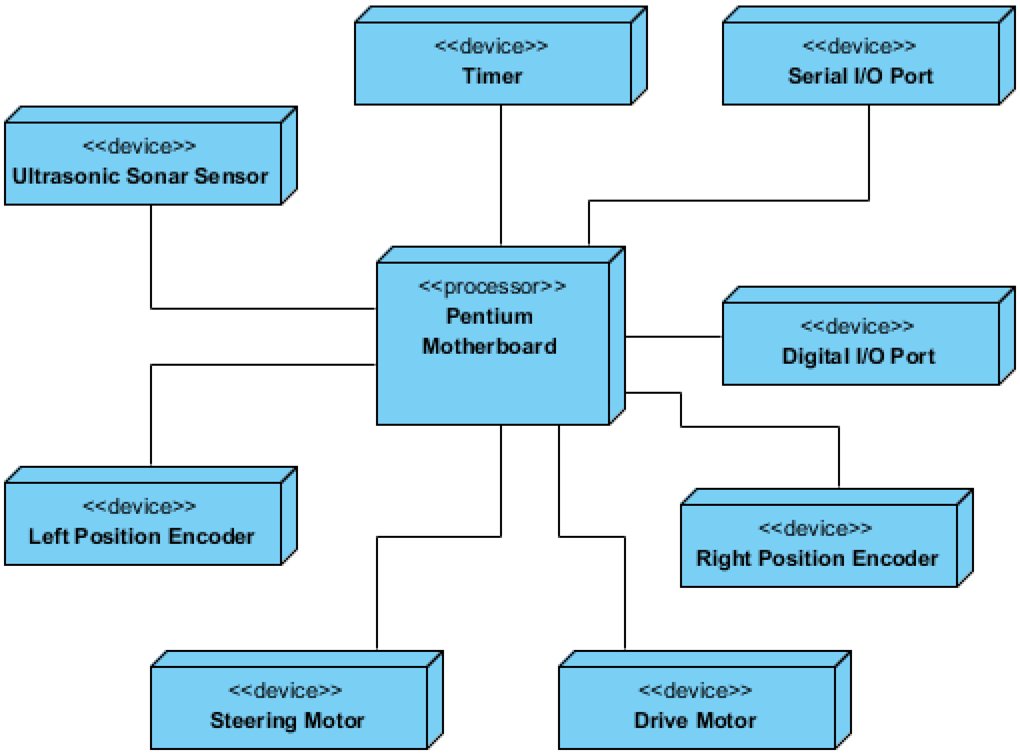
Other Notations

- Dependency
- Association relationships.
- May also contain notes and constraints.


Node
Internet
Modem Bank
Association
<<processor>> Caching Server
<<processor>> Caching Server
<<network>> Local Network
<<processor>> Primary Server
<<processor>> Server
<<processor>> Server
<<processor>> Server

Steps for Modeling an Embedded System

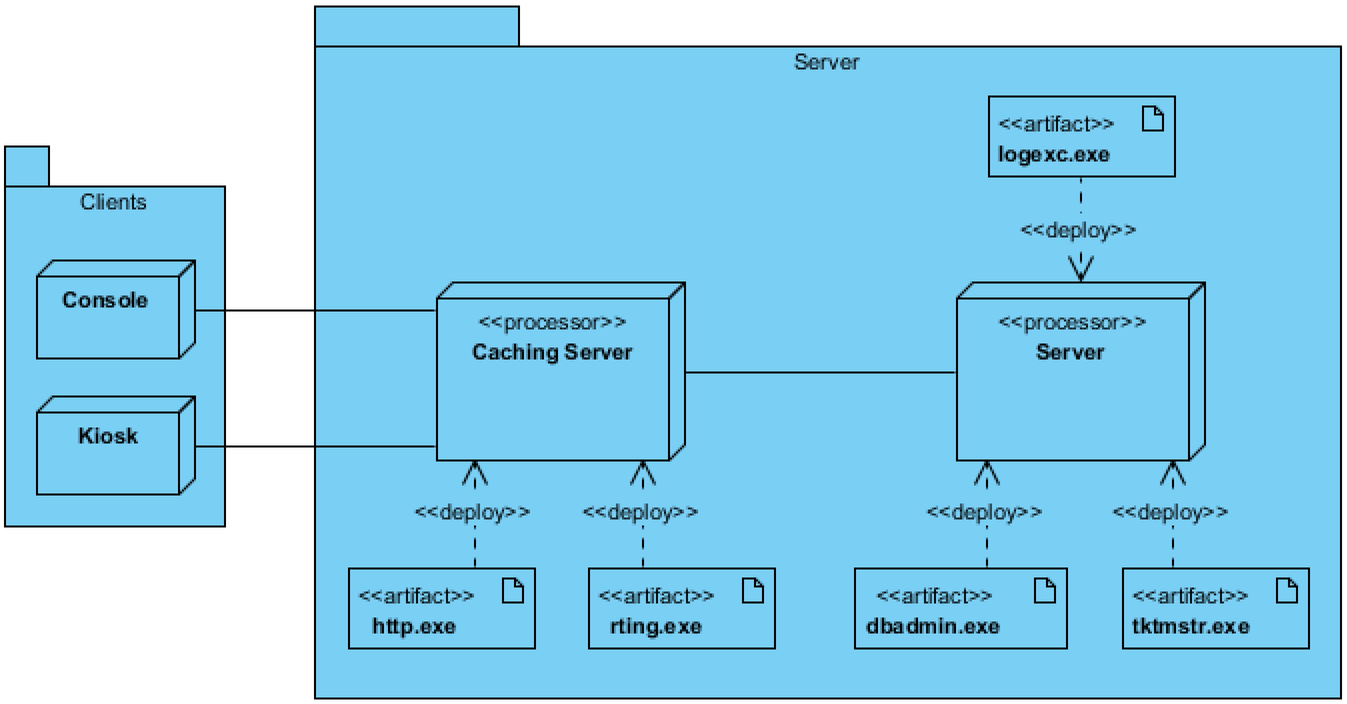
1. Identify the devices and nodes that are unique to your system.
2. Provide visual cues, especially for unusual devices, by using the UML's extensibility mechanisms to define system-specific stereotypes with appropriate icons. At the very least, you'll want to distinguish processors (which contain software components) and devices (which, at that level of abstraction, don't directly contain software).
3. 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.
4. As necessary, expand on any intelligent devices by modeling their structure with a more detailed deployment diagram.



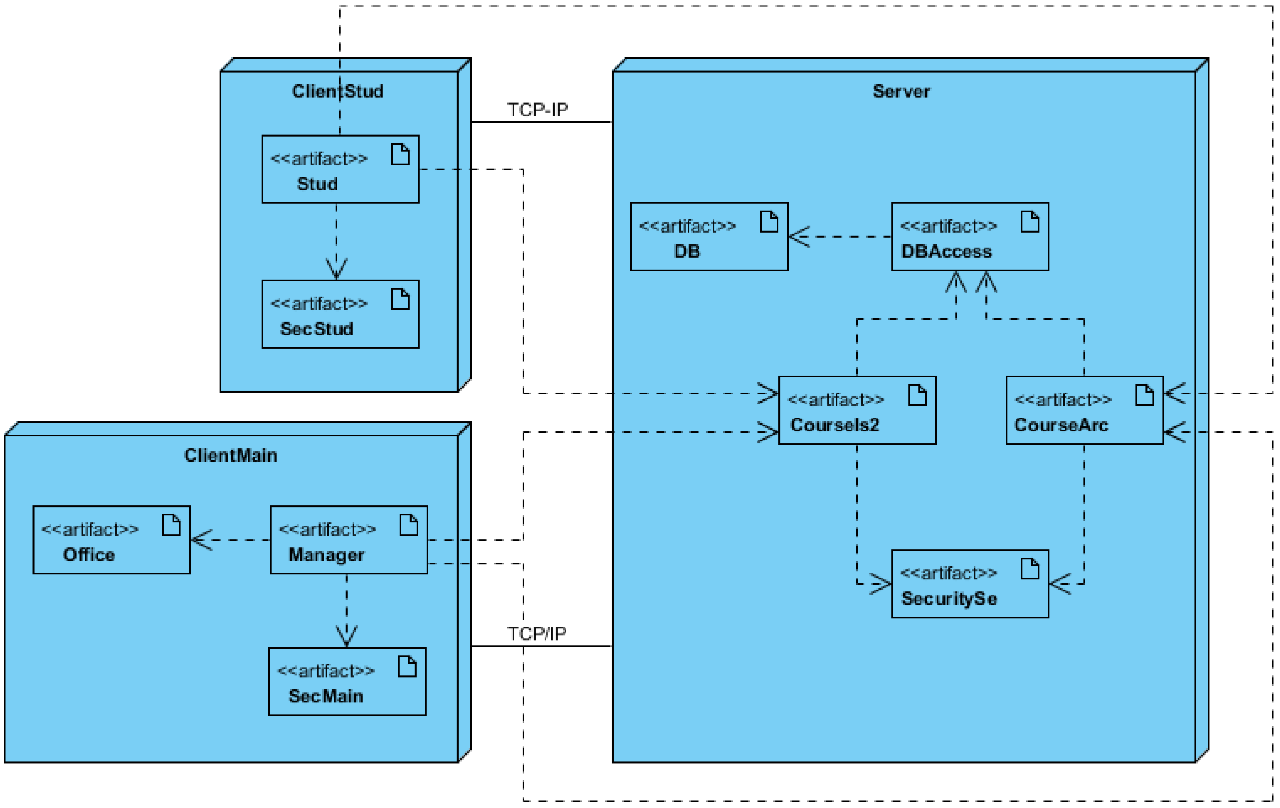
Steps for Modeling a Client/Server System

5. Identify the nodes that represent your system's client and server processors.
6. 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.
7. Provide visual cues for these processors and devices via stereotyping.
8. 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.

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



TCP/IP Client / Server Example

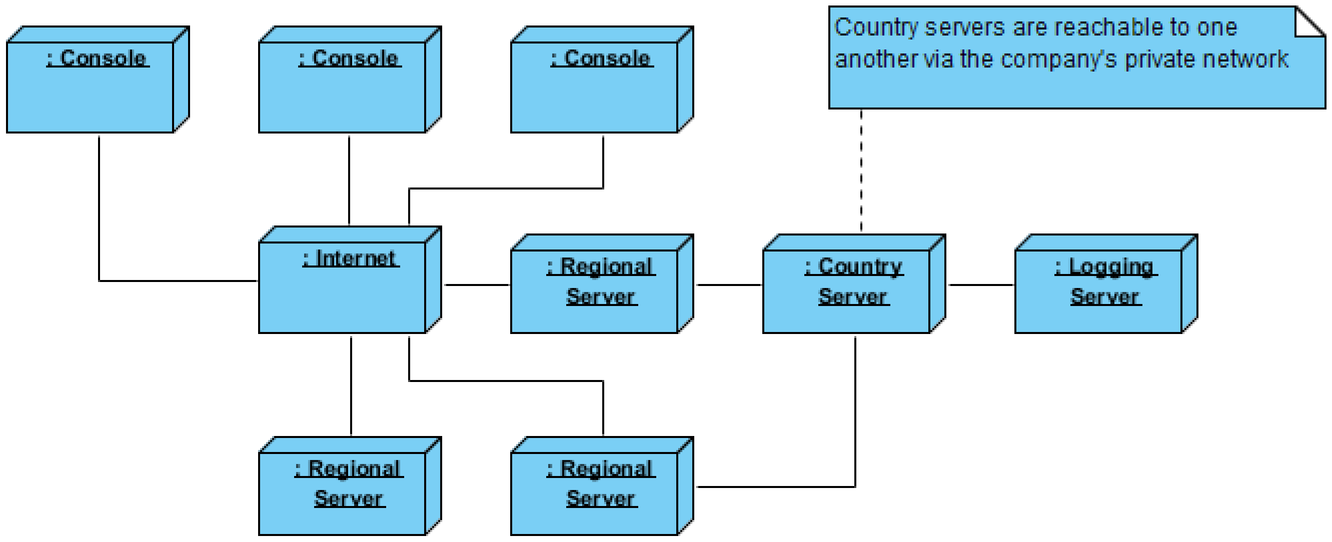


Deployment Diagram Example - Modeling a Distributed System

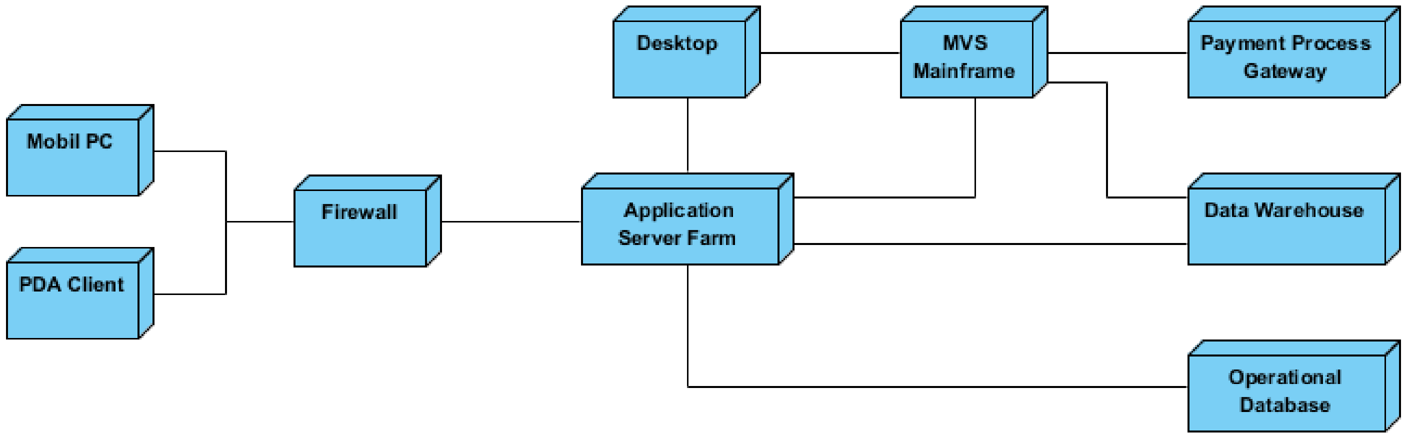
9. Identify the system's devices and processors as for simpler client/server systems.
10. If you need to reason about the performance of the system's network or the impact of changes to the network, be sure to model these communication devices to the level of detail sufficient to make these assessments.

11. Pay close attention to logical groupings of nodes, which you can specify by using packages.
12. 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.
13. 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.
14. When modeling a fully distributed system, it's common to reify the network itself as an node. i.e. Internet, LAN, WAN as nodes

The Example shows the topology of a fully distributed system.



Deployment Diagram Example - Corporate Distributed System



DEPLOYMENT DIAGRAM

Deployment Diagram is a type of diagram that specifies the physical hardware on which the software system will execute. It also determines how the software is deployed on the underlying hardware. It maps software pieces of a system to the device that are going to execute it.

The deployment diagram maps the software architecture created in design to the physical system architecture that executes it. In distributed systems, it models the distribution of the software across the physical nodes.

The software systems are manifested using various **artifacts**, and then they are mapped to the execution environment that is going to execute the software such as **nodes**. Many nodes are involved in the deployment diagram; hence, the relation between them is represented using communication paths.

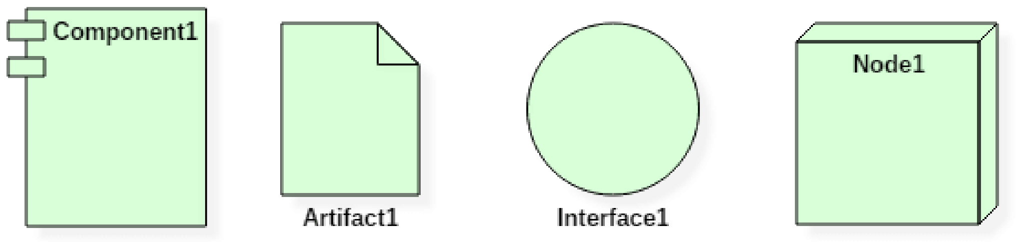
There are two forms of a deployment diagram.

- Descriptor form
  - It contains nodes, the relationship between nodes and artifacts.
- Instance form
  - It contains node instance, the relationship between node instances and artifact instance.
  - An underlined name represents node instances.

Purpose of a deployment diagram

Deployment diagrams are used with the sole purpose of describing how software is deployed into the hardware system. It visualizes how software interacts with the hardware to execute the complete functionality. It is used to describe software to hardware interaction and vice versa.

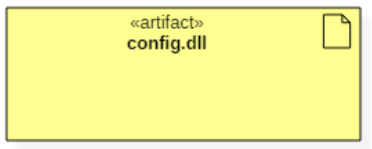
Deployment Diagram Symbol and notations



A deployment diagram consists of the following notations:

1. A node
2. A component
3. An artifact
4. An interface

What is an artifact?



An artifact represents the specification of a concrete real-world entity related to software development. You can use the artifact to describe a framework which is used during the software development process or an executable file. Artifacts are deployed on the nodes. The most common artifacts are as follows,

- Source files
- Executable files
- Database tables
- Scripts
- DLL files
- User manuals or documentation
- Output files

Artifacts are deployed on the nodes. It can provide physical manifestation for any UML element. Generally, they manifest components. Artifacts are labeled with the stereotype <<**artifact**>>**,** and it may have an artifact icon on the top right corner.

Each artifact has a filename in its specification that indicates the physical location of the artifact. An artifact can contain another artifact. It may be dependent on one another.

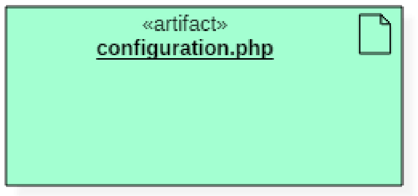
Artifacts have their properties and behavior that manipulates them.

Generally, an artifact is represented as follows in the unified modeling language.

Artifact Instances

An artifact instance represents an instance of a particular artifact. An artifact instance is denoted with same symbol as that of the artifact except that the name is underlined. UML diagram allows this to differentiate between the original artifact and the instance. Each physical copy or a file is an instance of a unique artifact.

Generally, an artifact instance is represented as follows in the unified modeling language.



What is a node?

Node is a computational resource upon which artifacts are deployed for execution. A node is a physical thing that can execute one or more artifacts. A node may vary in its size depending upon the size of the project.

Node is an essential UML element that describes the execution of code and the communication between various entities of a system. It is denoted by a 3D box with the node-name written inside of it. Nodes help to convey the hardware which is used to deploy the software.

An association between nodes represents a communication path from which information is exchanged in any direction.

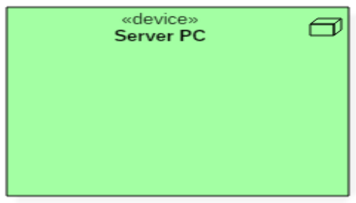
Generally, a node has two stereotypes as follows:

- **<< device >>**

It is a node that represents a physical machine capable of performing computations. A device can be a router or a server PC. It is represented using a node with stereotype **<<device>>.**

In the UML model, you can also nest one or more devices within each other.

Following is a representation of a device in UML:

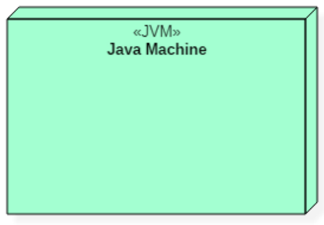


device node

- **<< execution environment >>**

It is a node that represents an environment in which software is going to execute. For example, Java applications are executed in java virtual machine (JVM). JVM is considered as an execution environment for Java applications. We can nest an execution environment into a device node. You can net more than one execution environments in a single device node.

Following is a representation of an execution environment in UML:



Execution environment node

How to draw a deployment diagram?

Deployment diagram visualizes the topological view of an entire system. It represents the deployment of a system.

A deployment diagram consists of nodes which describe the physical devices used inside the system. On these nodes, artifacts are deployed. We can also have node instances on which artifact instances are going to be implemented.

Node and artifacts of a system participate in the final execution of a system.

A deployment diagram plays a critical role during the administrative process, and it must satisfy the following parameters,

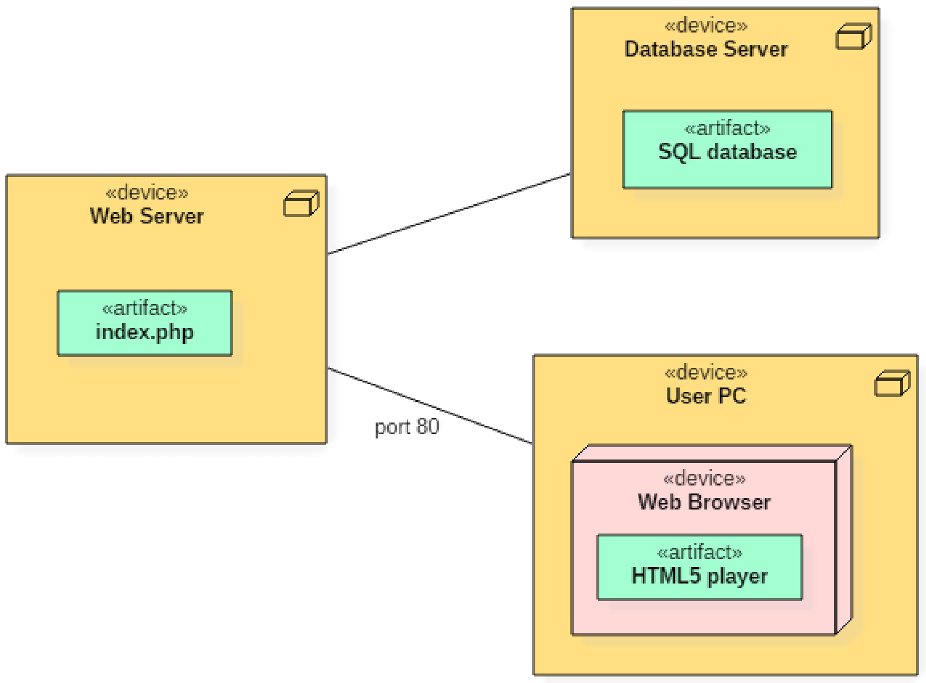
- High performance
- Maintainability
- Scalability
- Portability
- Easily understandable

Nodes and artifacts are the essential elements of deployment. Before actually drawing the deployment diagram, all nodes and the relationship between every node of the system must be identified.

You must know the architecture of a system, whether an application is a web application, cloud application, desktop application, or a mobile application. All these things are critical and plays a vital role during the development of a deployment diagram. **If all the nodes, relations, and artifacts are known, then it becomes easy to develop a deployment diagram.**

Example of a Deployment diagram

Following deployment diagram represents the working of HTML5 video player in the browser:



Deployment Diagram

When to use a deployment diagram?

Deployment diagrams are mostly used by system administrators, network engineers, etc. These diagrams are used with the sole purpose of describing how software is deployed into the hardware system. It visualizes how software interacts with the hardware to execute the complete functionality.

To make the software work efficiently and at a faster rate, the hardware also must be of good quality. It must be designed efficiently to make software work properly and produce accurate results in quick time.

Deployment diagrams can be used for,

1. Modeling the network topology of a system.
2. Modeling distributed systems and networks.
3. Forward and reverse engineering processes.

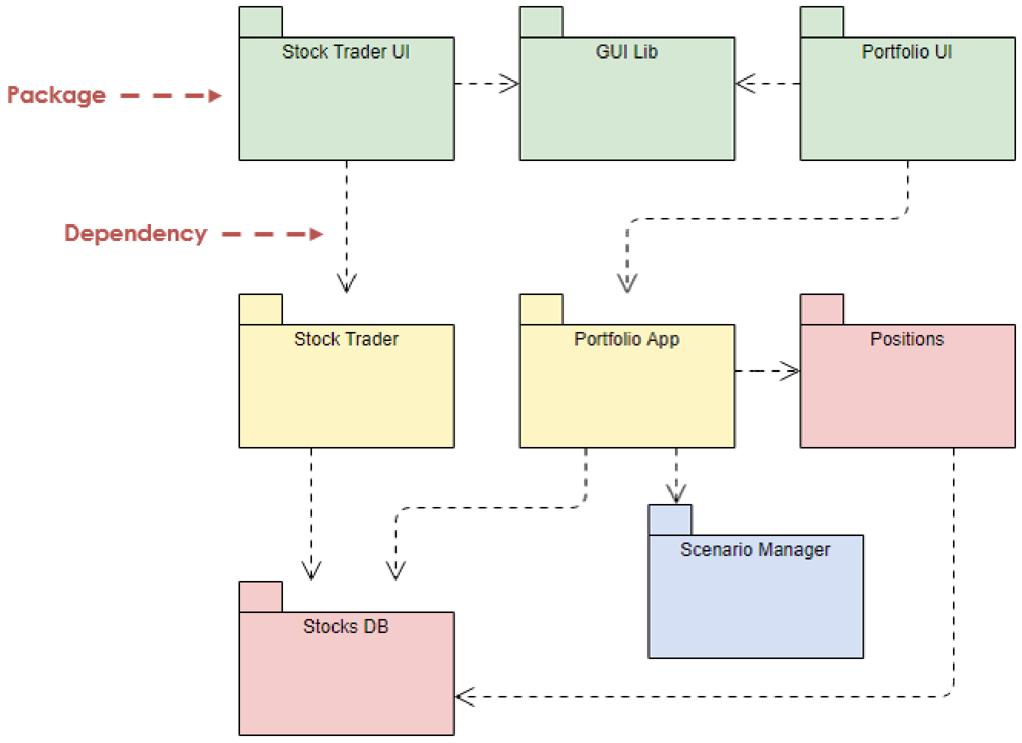
Package Diagram

Package diagram shows the arrangement and organization of model elements in middle to large scale project that can be used to show both structure and dependencies between sub-systems or modules.

Purpose of Package Diagrams

Package diagrams are used to structure high level system elements. Packages are used for organizing large system which contains diagrams, documents and other key deliverables.

- Package Diagram can be used to simplify complex class diagrams, it can group classes into packages.
- A package is a collection of logically related UML elements.
- Packages are depicted as file folders and can be used on any of the UML diagrams.

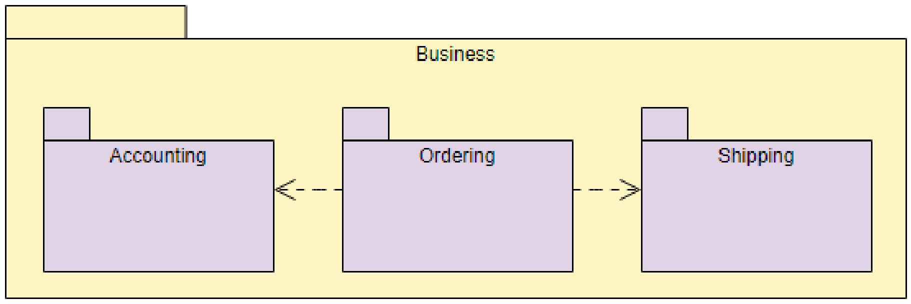


What is a Package Diagram in UML?

Big systems offer special challenges. Draw a class model for a large system, and it is too big to comprehend. There are too many links between classes to understand. A useful technique to handle this is that of UML's packages. A package in the Unified Modeling Language helps:

1. To group elements
2. To provide a namespace for the grouped elements
3. A package may contain other packages, thus providing for a hierarchical organization of packages.
4. UML elements can be grouped into packages.

The illustration below shows an example package diagram is used to represent the composition of a business.



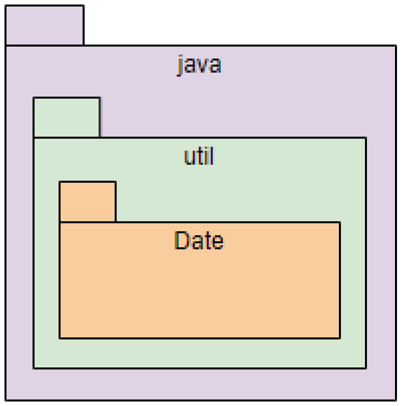
Package Diagram Notations

Package diagrams are used to structure high level systems. Packages are used for organizing large system which contains diagrams, documents and other key deliverables. In other words, packages can be used as a part of other diagrams also.

Nested and Hierarchical Packages

A package can be represented as a hierarchical structure with nested packages. Atomic module for nested package is usually class diagrams.

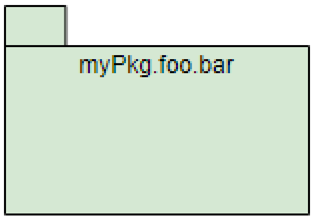
The figure below gives an example of package diagram that consists of several nested packages.



There are few constraints while using package diagrams, they are as follows.

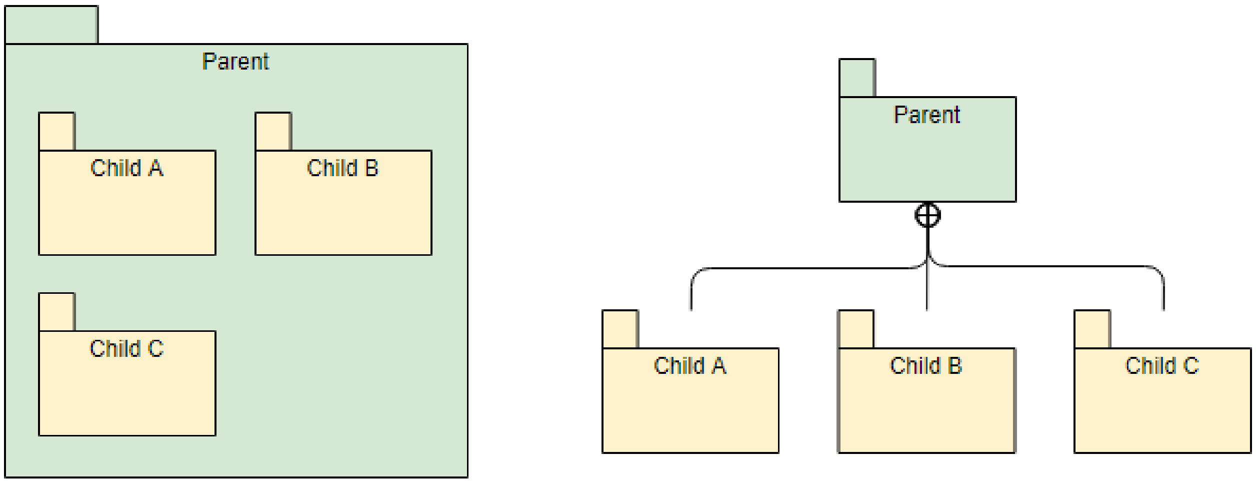
1. The name of packages should be unique within a system. However, it is allowed for classes inside different packages to have same name. For Example, Package::Product & Shipping::Product are allowed.
2. Users should avoid using package name delivered by the programming language. For Example, Java provides Date as a package. So, programmers should construct package with name Date.
3. Packages can include whole diagrams, name of components alone or no components at all.

A package can also has a fully qualified name. The figure below shows an example use of such a package.



Note That:
1. UML, C++, Perl, Ruby myPkg::foo::bar
2. Java, C# myPkg.foo.bar

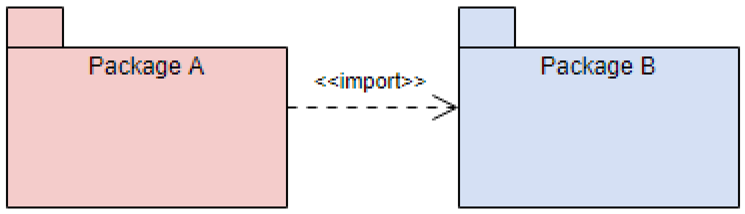
Package Containment
1. Packages are shown in static diagrams
2. Two equivalent ways to show containment:



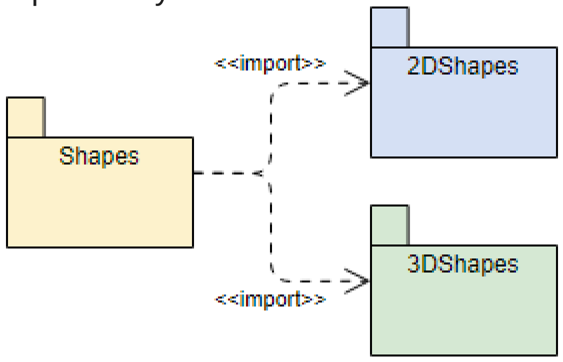
Dependency

There are two sub-types involved in dependency. They are <<access>> & <<import>>. Though there are two stereotypes users can use their own stereotype to represent the type of dependency between two packages.

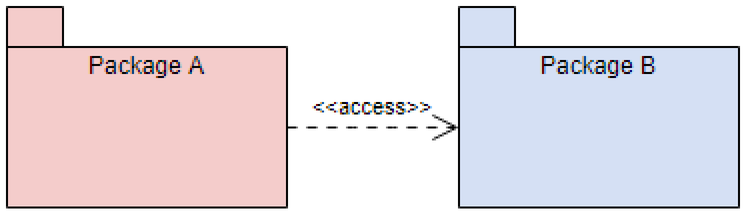
<<import>> - one package imports the functionality of other package



Example – <<import>> Dependency



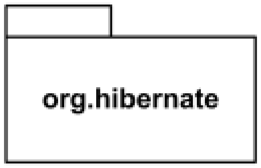
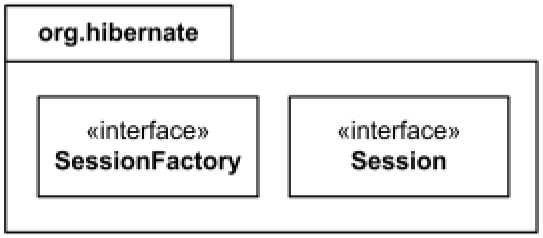
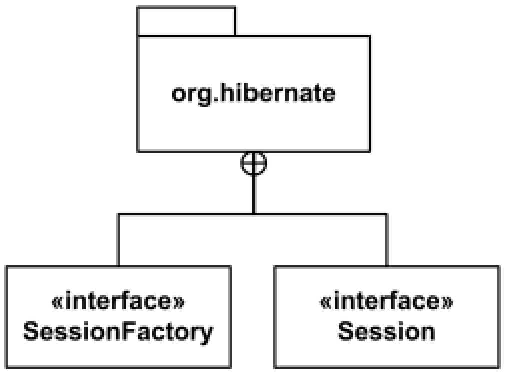
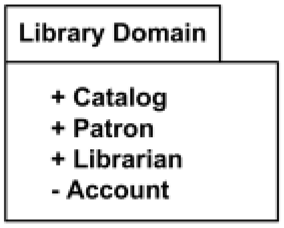
<<access>> - one package requires help from functions of other package

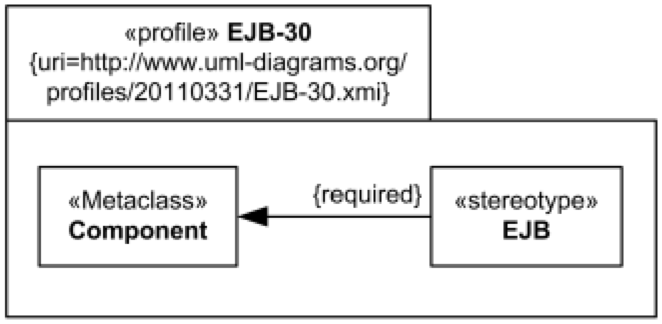


When to draw Package Diagram?

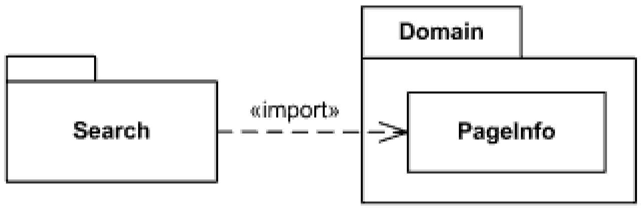
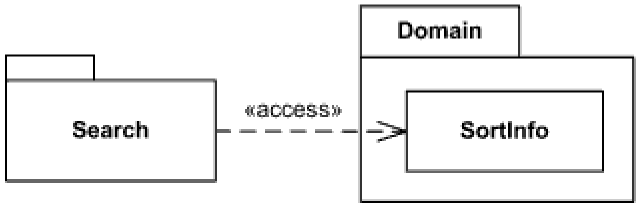
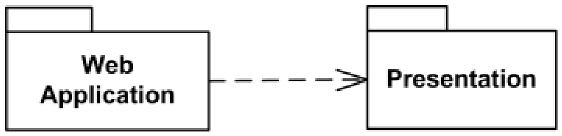
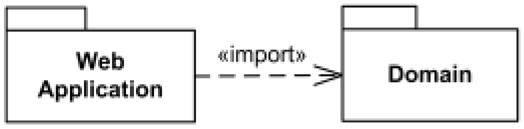
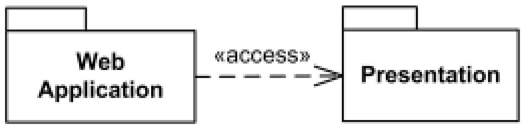
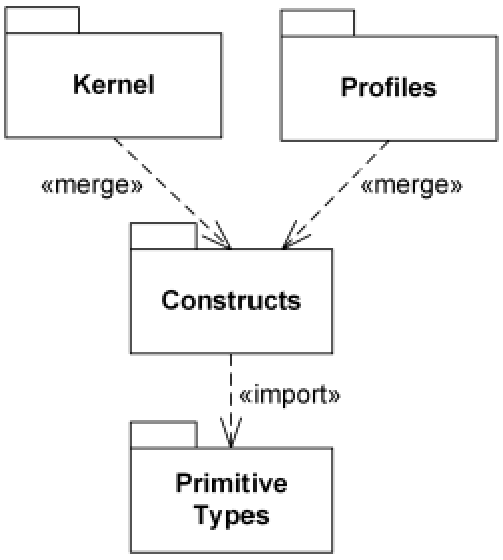
The UML does not treat package diagrams as a separate technique, It is often useful to combine them by grouping other model elements together into different packages on the same diagram. Package diagrams can be useful in many ways, such as:

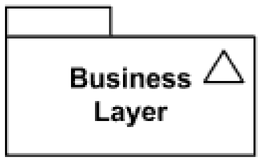
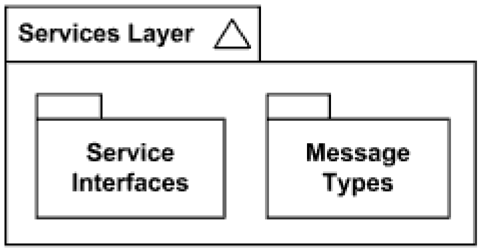
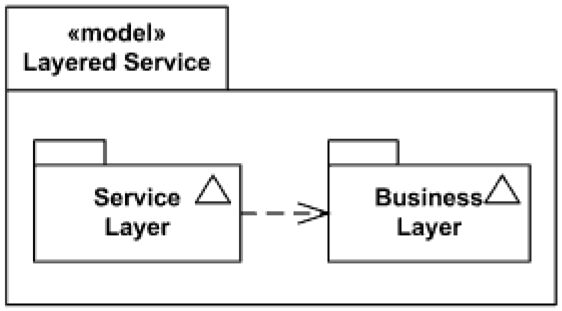
1. To create an overview of a large set of model elements
2. To organize a large model
3. To group related elements
4. To separate namespaces

Notation	Description
Package	
 *Package org.hibernate*	**Package** is a **namespace** used to group together elements that are semantically related and might change together. A package could be shown as a rectangle with a small tab attached to the left side of the top of the rectangle.
 org.hibernate «interface» SessionFactory «interface» Session *Package org.hibernate contains SessionFactory and Session.*	Members of the package may be shown within the boundaries of the package. In this case the name of the package should be placed on the tab.
 org.hibernate «interface» SessionFactory «interface» Session *Package org.hibernate contains interfaces SessionFactory and Session.*	Members of the package may be shown **outside** of the package by branching lines from the package to the members. A **plus sign (+) within a circle** is drawn at the end attached to the namespace (package). This notation for packages is semantically equivalent to **composition** (which is shown using solid diamond.)
 Library Domain + Catalog + Patron + Librarian - Account *All elements of Library Domain package are public except for Account.*	If an element that is owned by a package has **visibility**, it could be only **public** or **private** visibility. Protected or package visibility is not allowed. The visibility of a package element may be indicated by preceding the name of the element by a visibility symbol ("+" for public and "-" for private).
Package URI Attribute	

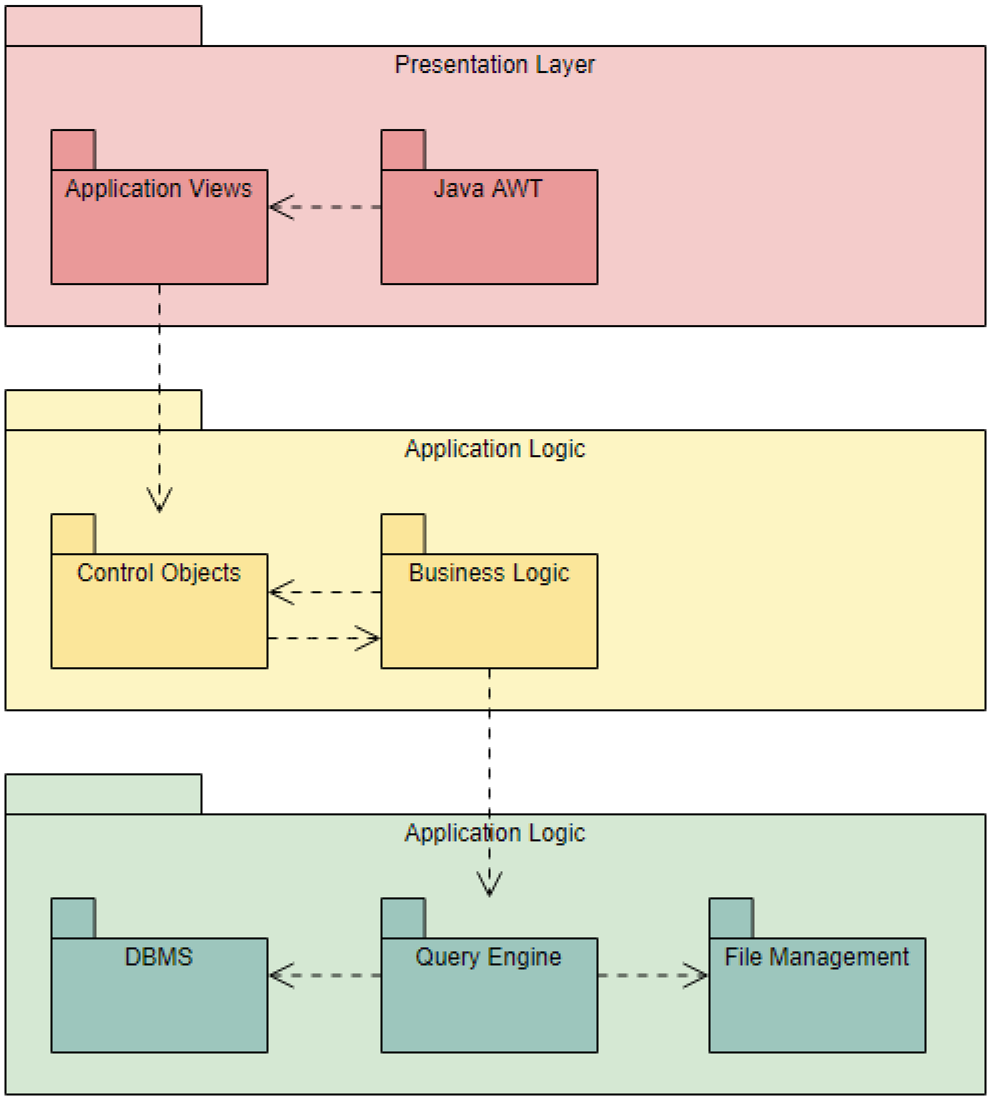
 «profile» EJB-30 {uri=http://www.uml-diagrams.org/ profiles/20110331/EJB-30.xmi}
«Metaclass» Component
{required}
«stereotype» EJB *EJB Profile shown as a package with URI attribute.*
	Package has optional **URI** attribute which serves as **unique identifier** of the package. This attribute was introduced in **UML 2.4** mostly to support exchange of **profiles** using XMI. **UML 2.4** requires this URI attribute to follow the rules and syntax of the IETF URI specification **RFC 2396** (while the more recent version of the URI syntax **RFC 3986** released in 2005 rendered the RFC 2396 obsolete). The URI attribute of a package may be rendered in the form {uri=<uri>} after the package name.

Element Import	

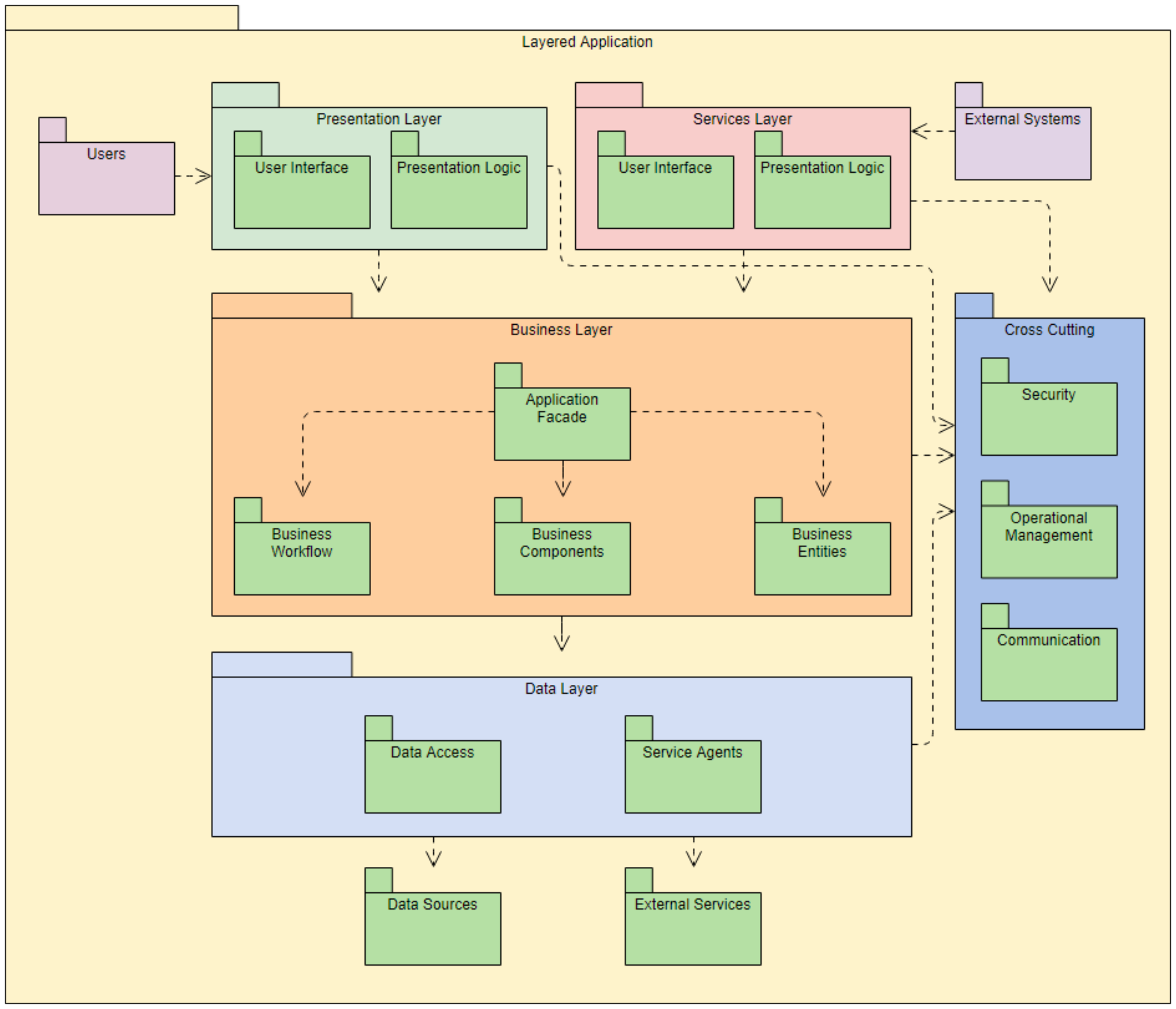
 *Public import of PageInfo element into Search namespace from Domain package.*	If element import is **public**, the imported element will be added to the namespace and made visible outside the namespace. Keyword **«import»** indicates **public** element import.
 *Private import of SortInfo element into Search namespace from Domain package.*	If element import is **private**, the imported element will be added to the namespace but will not be visible outside the namespace. Keyword **«access»** indicates **private** element import.
Package Import	
 *WebApplication imports Presentation package with default public visibility.*	**Package import** is shown using a dashed arrow with an open arrowhead from the importing namespace to the imported package. By default, the value of visibility is **public**, so it is the same as **«import»**.
 *Public import of Domain package into WebApplication.*	If the package import is **public**, the imported elements will be added to the namespace and made visible outside the namespace. Keyword **«import»** indicates **public** package import.
 *Private import of Presentation package into WebApplication.*	If the package import is **private**, the imported elements will be added to the namespace but will not be visible outside the namespace. Keyword **«access»** indicates **private** package import.
Package Merge	
 Kernel Profiles «merge» «merge» Constructs «import» Primitive Types *UML Kernel package merges Constructs package which imports Primitive Types.*	A **package merge** is a **directed relationship** between two packages that indicates that content of one package is extended by the contents of another package. **Package merge** is shown using a dashed line with an open arrowhead pointing from the receiving package to the merged package. Keyword «merge

Model	
 *Business layer model*	**Model** is a **package** which captures a view of a system. View is some abstraction of the system describing only those aspects of the system that are relevant to the purpose of the model, at the appropriate level of detail, describing logical or behavioral aspects of the system to a certain category of readers. Model is notated using the ordinary package symbol (a folder icon) with a small triangle in the upper right corner of the large rectangle.
 *Service Layer model contains service interfaces and message types*	If contents of the model are shown within the large rectangle, the triangle may be drawn to the right of the model name in the tab.
 *Stereotyped model Layered Service*	Model could be notated as a package with the keyword «model» placed above the name of the model.

Package Diagram Examples
1. MVC Structure



2.Package Diagram Example - Layering Structure


Layered Application
Users
Presentation Layer
User Interface
Presentation Logic
Services Layer
User Interface
Presentation Logic
External Systems
Business Layer
Application Facade
Business Workflow
Business Components
Business Entities
Cross Cutting
Security
Operational Management
Communication
Data Layer
Data Access
Service Agents
Data Sources
External Services

Modeling System and Sub-System

Model

A model captures a view of a system. It is an abstraction of the system, with a certain purpose. This purpose determines what is to be included in the model and what is irrelevant. Thus **the model completely describes those aspects of the system that are relevant to the purpose of the model, at the appropriate level of detail.**

Package

A package is used to group elements, and provides a namespace for the grouped elements.

Standard Stereotypes: Subsystem

A unit of hierarchical decomposition for large systems. A subsystem is commonly instantiated indirectly. Definitions of subsystems vary widely among domains and methods, and it is expected that domain and method profiles will specialize this construct. A subsystem may be defined to have specification and realization elements. **«specification»** and **«realization».**

Subsystems

A system is an organized collection of elements that may be recursively decomposed into smaller subsystems and eventually into **non-decomposable primitive elements**.

For example, the project management system may be decomposed into the following:

- A user interface subsystem responsible for providing a user interface through which users may interact with the system
- A business processing subsystem responsible for implementing business functionality
- A data subsystem responsible for implementing data storage functionality

The primitive elements would be the various classes that are used in these subsystems and ultimately in the whole system. While a **package** simply groups' elements, a **subsystem** groups elements that together provide services such that other elements may access only those services and none of the elements themselves. And while **packages** allow us to partition our system into logical groups and relate these logical groups, **subsystems** allow us to consider what services these logical groups provide to one another.

A subsystem is shown as a package marked with the subsystem keyword. The large package rectangle may have three standard compartments shown by dividing the rectangle with a vertical line and then dividing the area to the left of this line into two compartments with a horizontal line. Figure 3-42 shows how a Data subsystem for our project management system might look. The subsystem's operations, specification elements, and interfaces describe the services the subsystem provides, and are the only services accessible by other elements outside the subsystem.

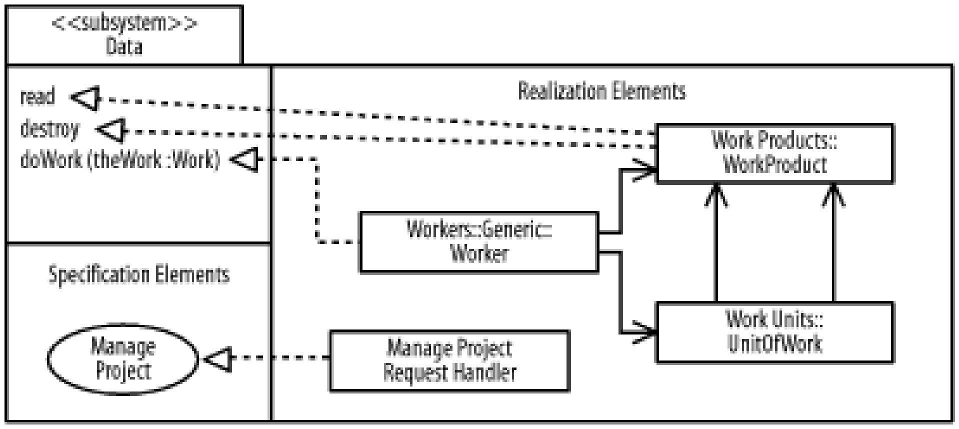


Figure 3-42. A subsystem's representation in the UML

The upper-left compartment shows a list of operations that the subsystem realizes. The lower-left compartment may be labeled "**Specification Elements**" and shows specification elements that the subsystem realizes. The right compartment may be labeled "**Realization Elements**" and shows elements inside the subsystem that realize the subsystem's operations and specification elements as well as any interfaces that the subsystem provides.

- Any element may be used as a specification or realization element, because a realization simply indicates that the realization element supports at least all the operations of the specification element without necessarily having to support any attributes or associations of the specification element.

The Business Processing and Data packages are now subsystems. The Business Processing subsystem provides an interface that is used by the User Interface package. The Business Processing subsystem itself uses the Data subsystem and the IProducible interface provided by the Data subsystem. The Data subsystem realizes the IProducible interface, which is outside the subsystem itself, various operations, and the Manage Project use case that was discussed in Chapter 2. The use case is the oval in the specification element's compartment. The realization elements of the Data subsystem realize the read, destroy, and doWork operations, the use case, and the operations of the IProducible interface.

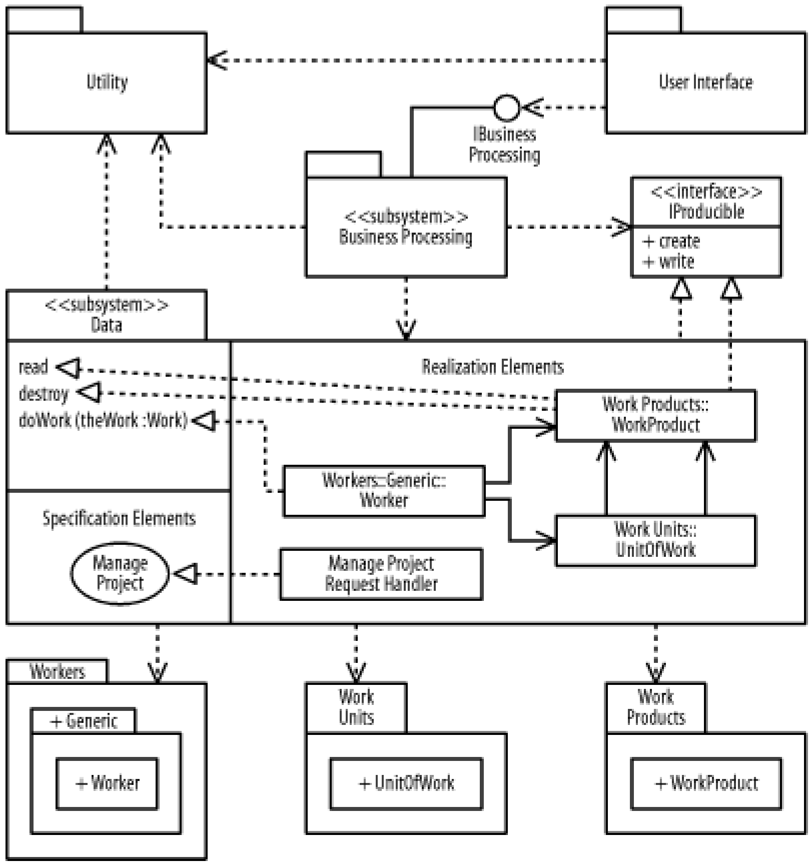


Figure 3-43. Subsystems

PATTERNS

Design Patterns

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

A Pattern has four elements namely:

1. A Pattern name
2. Problem
3. Solution
4. Consequences

Why Design Pattern?

- The DP satisfies this need for good, simple and reusable solutions.
- The DP catalog common interactions between objects that programmers have often found useful.

Uses of Design Patterns

Design Patterns have two main usages in software development.

Common platform for developers

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

Best Practices

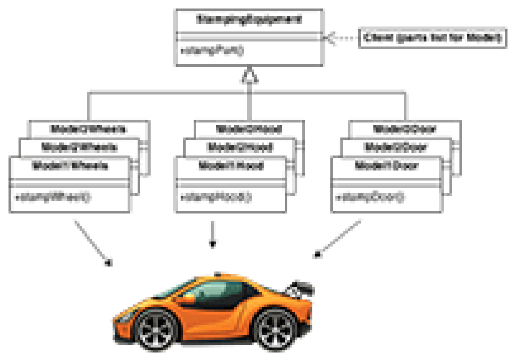
Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps un-experienced developers to learn software design in an easy and faster way.

Types Of Design Pattern

As per the design pattern reference book **Design Patterns - Elements of Reusable Object-Oriented Software** , there are 23 design patterns. These patterns can be classified in three categories: **Creational, Structural and behavioral patterns.**

1. **Creational design patterns**

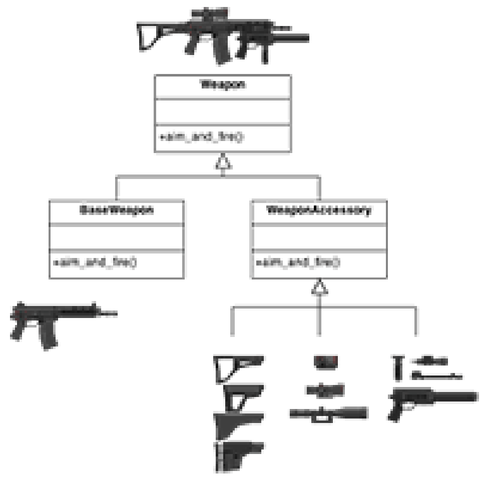
These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.



- **Abstract Factory**
  Creates an instance of several families of classes
- **Builder**
  Separates object construction from its representation
- **Factory Method**
  Creates an instance of several derived classes
- **Object Pool**
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- **Prototype**
  A fully initialized instance to be copied or cloned
- **Singleton**
  A class of which only a single instance can exist

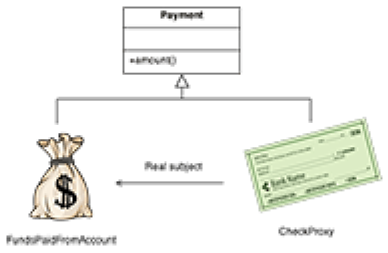
2. Structural design patterns

These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.



- **Adapter**
  Match interfaces of different classes
- **Bridge**
  Separates an object's interface from its implementation

- **Composite**
  A tree structure of simple and composite objects
- **Decorator**
  Add responsibilities to objects dynamically
- **Facade**
  A single class that represents an entire subsystem
- **Flyweight**
  A fine-grained instance used for efficient sharing

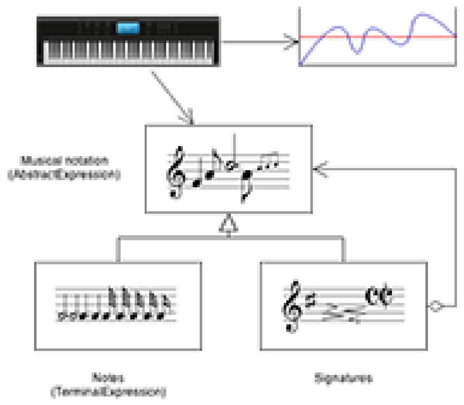


Private Class Data
Restricts accessor/mutator access

- **Proxy**
  An object representing another object

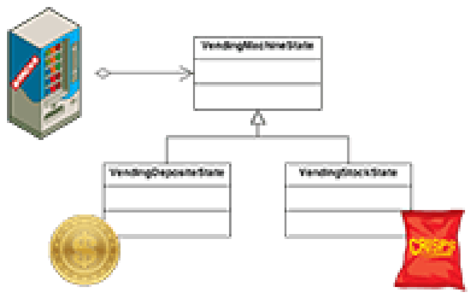
3. Behavioral design patterns

These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.


Musical notation (AbstractExpression)
Notes (TerminalExpression)
Signatures

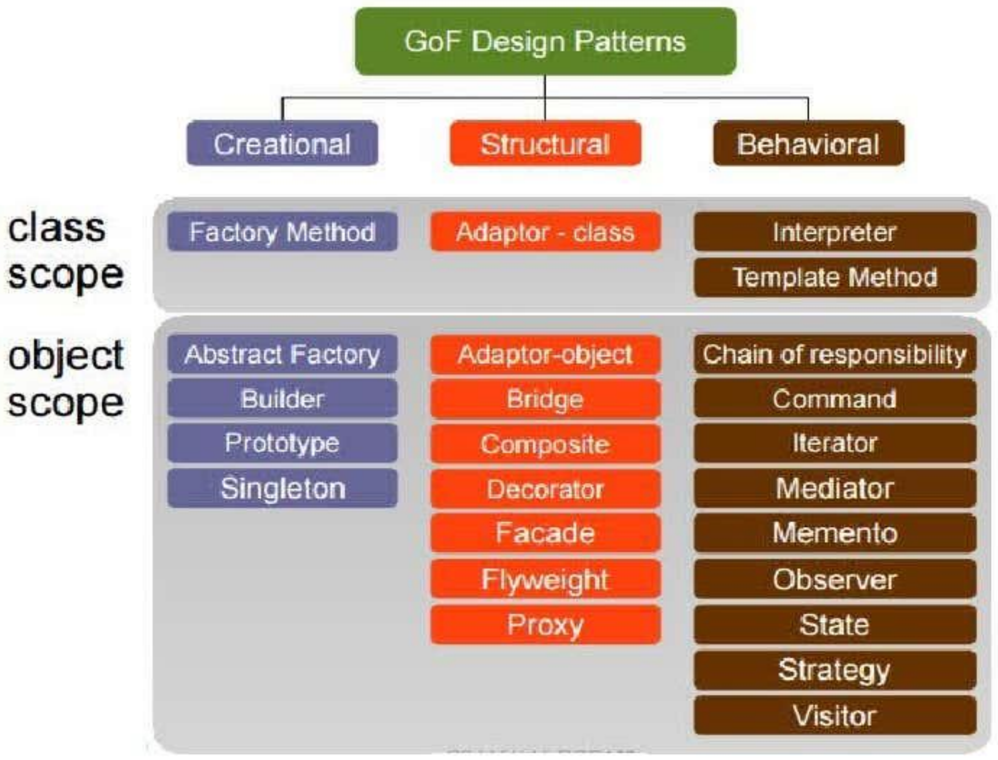
- **Chain of responsibility**
  A way of passing a request between a chain of objects
- **Command**
  Encapsulate a command request as an object
- **Interpreter**
  A way to include language elements in a program
- **Iterator**
  Sequentially access the elements of a collection

- **Mediator**
  Defines simplified communication between classes
- **Memento**
  Capture and restore an object's internal state
- **Null Object**
  Designed to act as a default value of an object
- **Observer**
  A way of notifying change to a number of classes



- **State**
  Alter an object's behavior when its state changes

- **Strategy**
  Encapsulates an algorithm inside a class
- **Template method**
  Defer the exact steps of an algorithm to a subclass
- **Visitor**
  Defines a new operation to a class without change


GoF Design Patterns
Creational
Structural
Behavioral
class scope
Factory Method
Adaptor - class
Interpreter
Template Method
object scope
Abstract Factory
Builder
Prototype
Singleton
Adaptor-object
Bridge
Composite
Decorator
Facade
Flyweight
Proxy
Chain of responsibility
Command
Iterator
Mediator
Memento
Observer
State
Strategy
Visitor