Activity Diagram: Activity and Actions, Initial and Final Activity, Activity Edge, Decision and Merge Points, Fork and Join, Input and Output Pins, Activity Group, Activity Partitions, Constraints on Action, Swim Lanes. Sequence Diagram: Context, Objects and Roles, Links, Object Life Line, Message or stimulus, Activation/Focus of Control, Modelling Interactions.

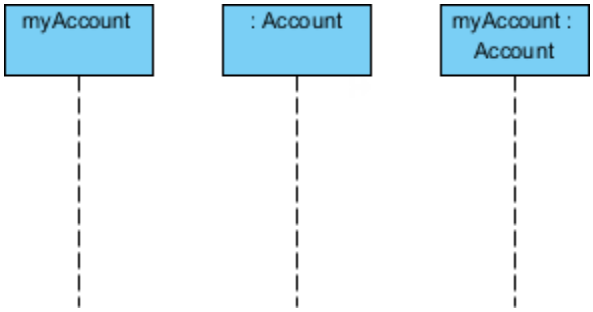
Objects and Roles

Object

In the UML, an object in a sequence diagram is drawn as a rectangle containing the name of the object, underlined. An object can be named in one of three ways: the object name, the object name and its class, or just the class name (anonymous object). The three ways of naming an object are shown in Figure below.

Lifeline

Entities of participants in a collaboration (scenario) are written horizontally across the top of the diagram. A lifeline is represented by dashed vertical line drawn below each object. These indicate the existence of the object.



Object names can be specific (e.g., myAccount) or they can be general (e.g., myAccount :Account). Often, an anonymous object (:Account) may be used to represent any object in the class. Each object also has its timeline represented by a dashed line below the object. Messages between objects are represented by arrows that point from sender object to the receiver object.

Everything in an object-oriented system is accomplished by objects. Objects take on the responsibility for things like managing data, moving data around in the system, responding to inquiries, and protecting the system. Objects work together by communicating or interacting with one another.

Message

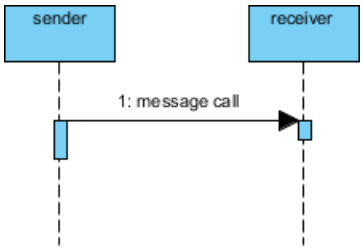
Messages depict the invocation of operations are shown horizontally. They are drawn from the sender

to the receiver. Ordering is indicated by vertical position, with the first message shown at the top of

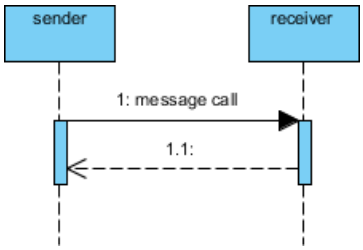
the diagram, and the last message shown at the bottom. As a result, sequence numbers is optional The line type and arrowhead type indicates the type of message being used:

1. A **synchronous message** (typically an operation call) is shown as a solid line with a filled

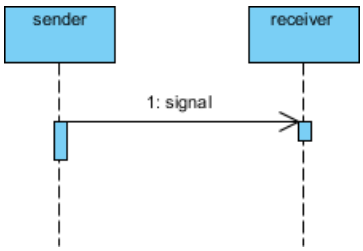
arrowhead. It is a regular message call used for normal communication between sender and receiver.



2. A **return message** uses a dashed line with an open arrowhead.



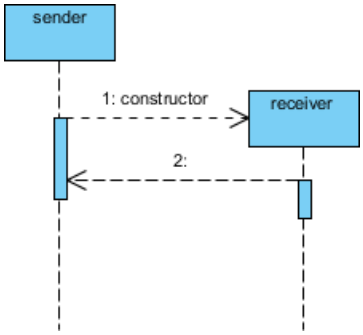
3. An **asynchronous message** has a solid line with an open arrowhead. A signal is an asynchronous message that has no reply.



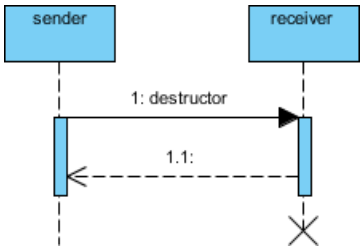
Participants do not necessarily live for the entire duration of a sequence diagram's interaction.

Participants can be created and destroyed according to the messages that are being passed.

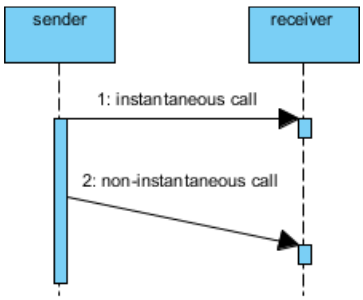
A **constructor message** creates its receiver. The sender that already exist at the start of the interaction are placed at the top of the diagram. Targets that are created during the interaction by a constructor call are automatically placed further down the diagram.



A **destructor message** destroys its receiver. There are other ways to indicate that a target is destroyed during an interaction. Only when a target's destruction is set to 'after destructor' do you have to use a destructor.



Messages are often considered to be instantaneous, thus, the time it takes to arrive at the receiver is negligible. The messages are drawn as a horizontal arrow. To indicate that it takes a certain while before the receiver actually receives a message, a **slanted arrow is used**.



Focus of Control

Focus of Control represents the period during which an element is performing an operation. The

top and the bottom of the of the rectangle are aligned with the initiation and the completion time

respectively

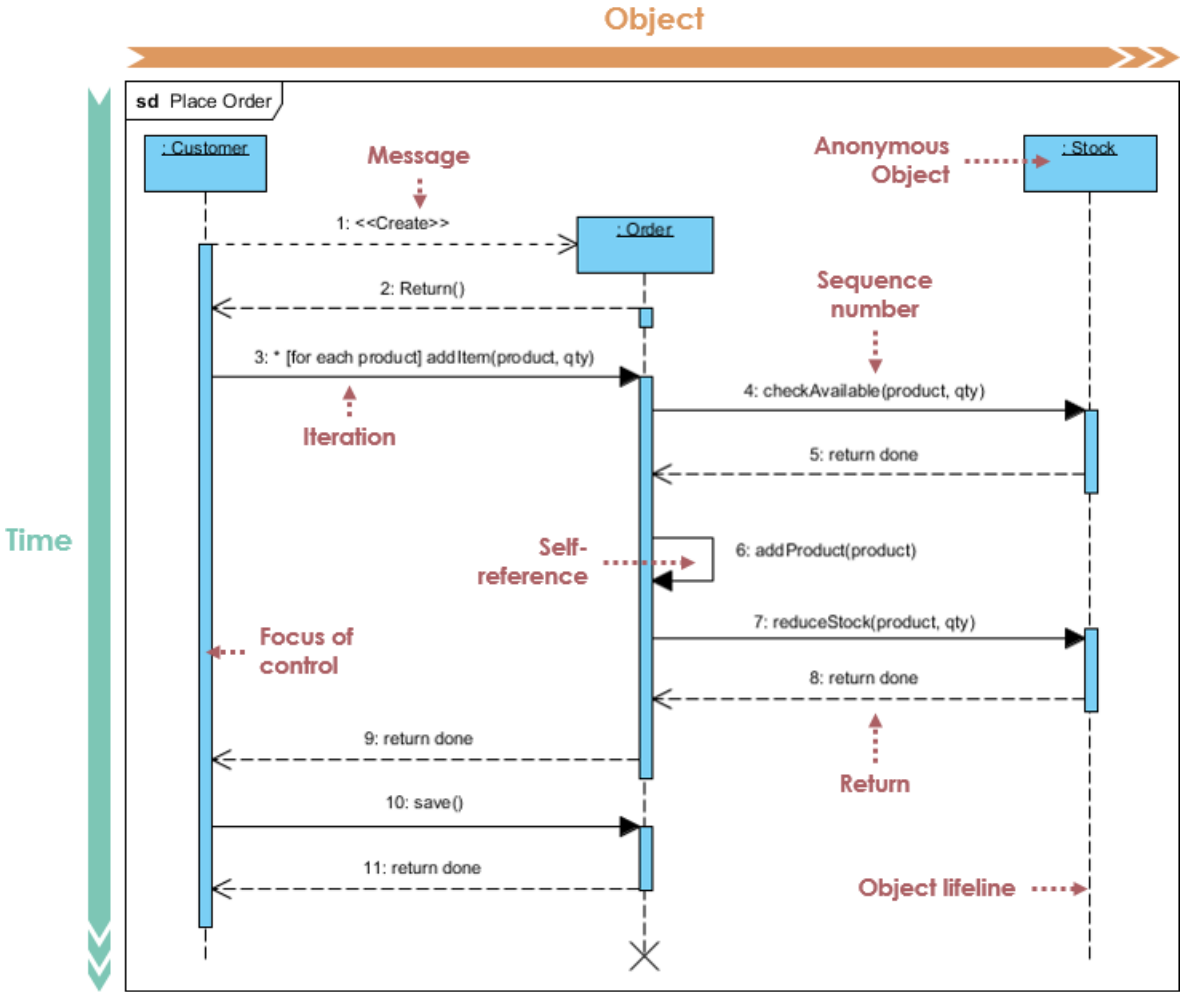
Iteration notation

Iteration notation represents a message is sent many times to multiple receiver objects, as would happen when you are iterating over a collection. You can show the basis of the iteration within brackets, such as *[for all order lines].

Example: Place Order

The example shows a Sequence diagram with three participating objects: Customer, Order, and the Stock. Without even knowing the notation formally, you can probably get a pretty good idea of what is going on.

1. Step 1 and 2: Customer creates an order.
2. Step 3: Customer add items to the order.
3. Step 4, 5: Each item is checked for availability in inventory.
4. Step 6, 7, 8 : If the product is available, it is added to the order.
5. Step 9 return
6. Step 10, 11: save and destroy order



Sequence Fragments

In a UML sequence diagram, combined fragments let you show loops, branches, and other

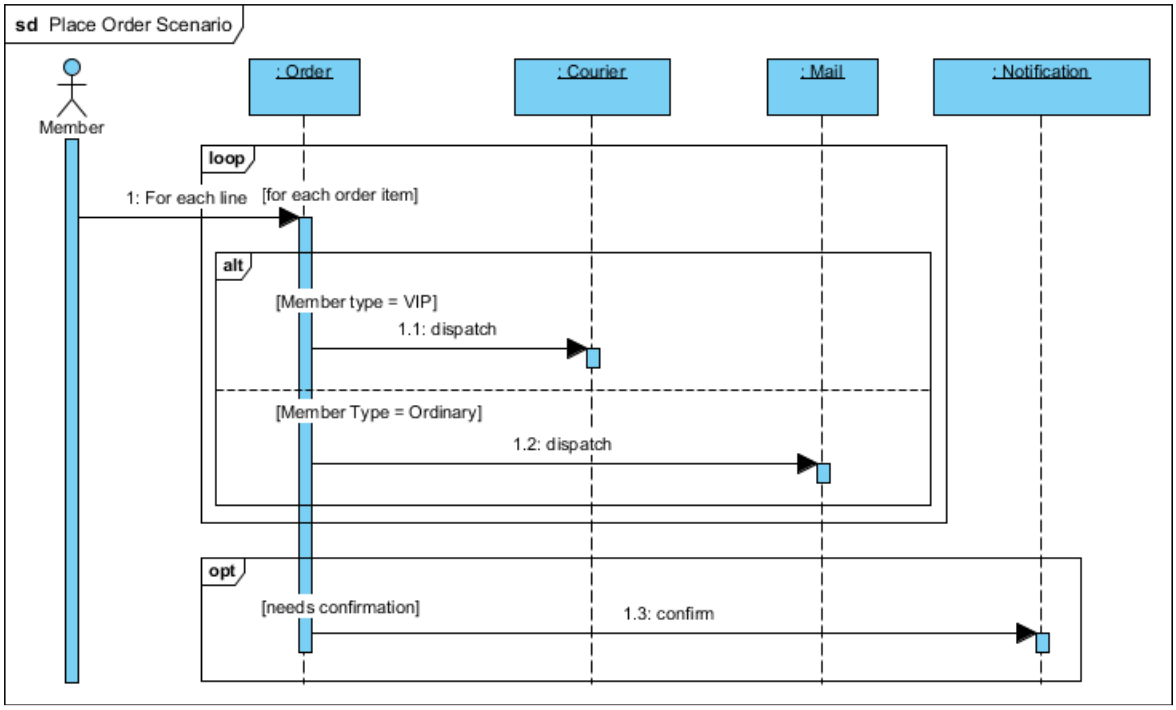
alternatives. A combined fragment consists of one or more interaction operands, and each of these encloses one or more messages, interaction uses, or combined fragments.

A sequence fragment is represented as a box called a combined fragment, which encloses a portion of the interactions within a sequence diagram. The fragment operator (in the top left cornet) indicates the type of fragment. Fragment types include ref, assert, loop, break, alt, opt and neg, ref, sd.

Operator	Meaning
alt	Alternative multiple fragments: only the one whose condition is true will execute.
opt	**Optional**: the fragment executes only if the supplied condition is true. Equivalent to an alt only with one trace.
par	**Parallel**: each fragment is run in parallel.
loop	**Loop**: the fragment may execute multiple times, and the guard indicates the basis of iteration.
critical	**Critical region**: the fragment can have only one thread executing it at once.
neg	**Negative**: the fragment shows an invalid interaction.
ref	**Reference**: refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value.
sd	**Sequence diagram**: used to surround an entire sequence diagram.

Example - Place Order Scenario

A member of a ship who would like to place an order online. The item ordered will be sent to the member either send by courier or by ordinary mail depending on she member status (VIP, Ordinary membership). Optionally, the shop will send the member a confirmation notification if the member opted for the notification option in the order.


sd Place Order Scenario
Member
: Order
: Courier
: Mail
: Notification
loop
1: For each line
[for each order item]
alt
[Member type = VIP]
1.1: dispatch
[Member Type = Ordinary]
1.2: dispatch
opt
[needs confirmation]
1.3: confirm

UNIT-IV

Collaboration Diagram: Objects and Links, Messages and stimuli, Active Objects, Communication Diagram,Iteration Expression, Parallel Execution, Guard Expression, Timing Diagram. Design Using UML Activity Diagram ,Introduction to Patterns General Responsibility Assignment Software Patterns (GRASP) : Introduction, Creator , Information Expert, Low coupling, Controller, High Cohesion, Polymorphism , Pure fabrication, Indirection, Protected Variations. Gang of Four (GoF): Introduction, Categories of Patterns (Creational, Structural and Behavioral Patterns), Singleton, Adapter, State, and Strategy.

Collaboration diagrams (known as Communication Diagram in UML 2.x) are used to show how objects interact to perform the behavior of a particular use case, or a part of a use case. Along with sequence diagrams, collaboration are used by designers to define and clarify the roles of the objects that perform a particular flow of events of a use case. They are the primary source of information used to determining class responsibilities and interfaces.

What is a Collaboration?

- A Collaboration is a collection of named objects and actors with links connecting them. They collaborate in performing some task.
- A Collaboration defines a set of participants and relationships that are meaningful for a given set of purposes
- A Collaboration between objects working together provides emergent desirable functionalities in Object-Oriented systems
- Each object (responsibility) partially supports emergent functionalities
- Objects are able to produce (usable) high-level functionalities by working together
- Objects collaborate by communicating (passing messages) with one another in order to work together

Why Collaboration Diagram?

Unlike a sequence diagram, a collaboration diagram shows the relationships among the objects. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways.

Because of the format of the collaboration diagram, they tend to better suited for analysis activities (see Activity: Use-Case Analysis). Specifically, they tend to be better suited to depicting simpler interactions of smaller numbers of objects. However, if the number of objects and messages grows, the diagram becomes increasingly hard to read. In addition, it is difficult to show

additional descriptive information such as timing, decision points, or other unstructured information that can be easily added to the notes in a sequence diagram. So, here are some use cases that we want to create a collaboration diagram for:

- Model collaborations between objects or roles that deliver the functionalities of use cases and operations
- Model mechanisms within the architectural design of the system
- Capture interactions that show the messages passing between objects and roles within the collaboration
- Model alternative scenarios within use cases or operations that involve the collaboration of different objects and interactions
- Support the identification of objects (hence classes) that participate in use cases
- Each message in a collaboration diagram has a sequence number.
- The top-level message is numbered 1. Messages sent during the same call have the same decimal prefix but suffixes of 1, 2, etc. according to when they occur.

Notations of Collaboration Diagram

Objects

An object is represented by an object symbol showing the name of the object and its class underlined, separated by a colon:

Object_name : class_name

You can use objects in collaboration diagrams in the following ways:

- Each object in the collaboration is named and has its class specified
- Not all classes need to appear
- There may be more than one object of a class
- An object's class can be unspecified. Normally you create a collaboration diagram with objects first and specify their classes later.
- The objects can be unnamed, but you should name them if you want to discriminate different objects of the same class.

Actors

Normally an actor instance occurs in the collaboration diagram, as the invoker of the interaction. If you have several actor instances in the same diagram, try keeping them in the periphery of the diagram.

- Each Actor is named and has a role

- One actor will be the initiator of the use case

Links

Links connect objects and actors and are instances of associations and each link corresponds to an association in the class diagram

Links are defined as follows:

- A link is a relationship among objects across which messages can be sent. In collaboration diagrams, a link is shown as a solid line between two objects.
- An object interacts with, or navigates to, other objects through its links to these objects.
- A link can be an instance of an association, or it can be anonymous, meaning that its association is unspecified.
- Message flows are attached to links, see Messages.

Messages

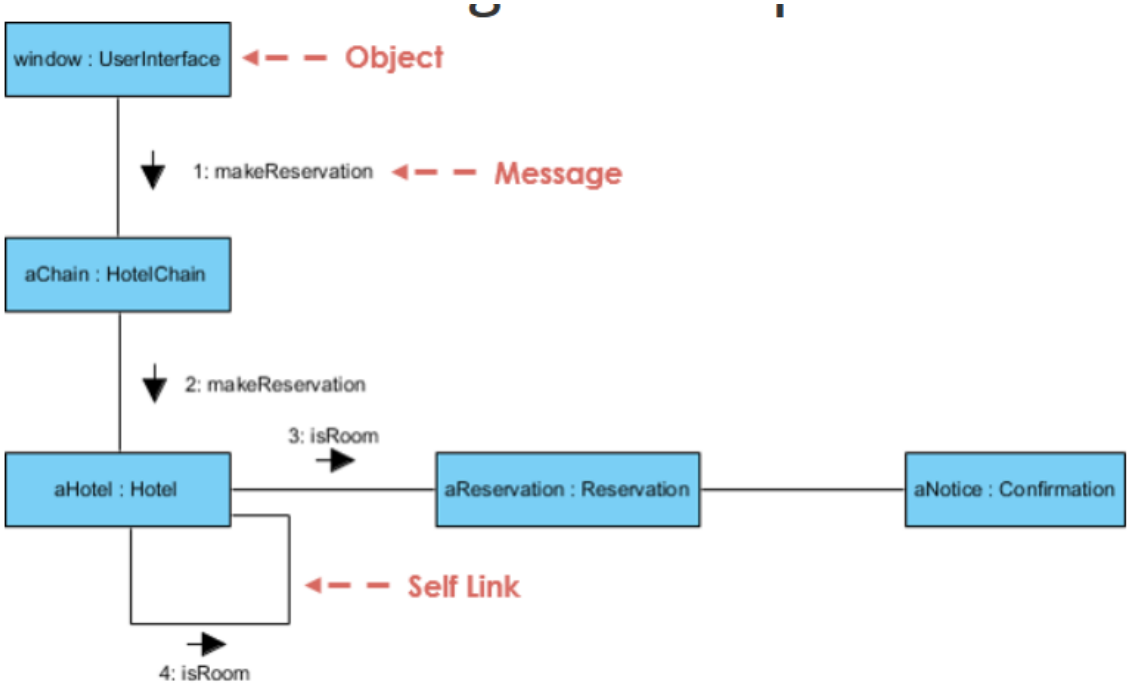
A message is a communication between objects that conveys information with the expectation that activity will ensue. In collaboration diagrams, a message is shown as a labeled arrow placed near a link.

- The message is directed from sender to receiver
- The receiver must understand the message
- The association must be navigable in that direction

Steps for Creating Collaboration Diagrams

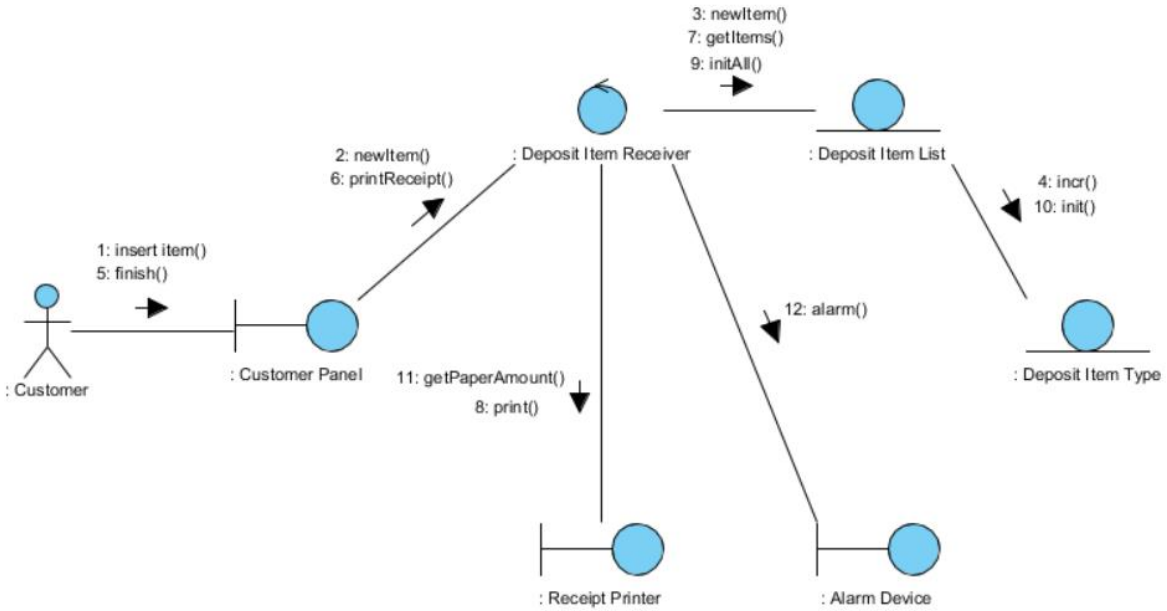
1. Identify behavior whose realization and implementation is specified
2. Identify the structural elements (class roles, objects, subsystems) necessary to carry out the functionality of the collaboration
    - Decide on the context of interaction: system, subsystem, use case and operation
3. Model structural relationships between those elements to produce a diagram showing the context of the interaction
4. Consider the alternative scenarios that may be required
    - Draw instance level collaboration diagrams, if required.
    - Optionally draw a specification level collaboration diagram to summarize the alternative scenarios in the instance level sequence diagrams

Collaboration Diagram Example



Collaboration Diagram in Robustness Diagram Format

You can have objects and actor instances in collaboration diagrams, together with links and messages describing how they are related and how they interact. The Receive Deposit Item in the Recycling-Machine System diagram shown below describes what takes place in the participating objects, in terms of how the objects communicate by sending messages to one another. You can make a collaboration diagram for each variant of a use case's flow of events.



Communication Diagram

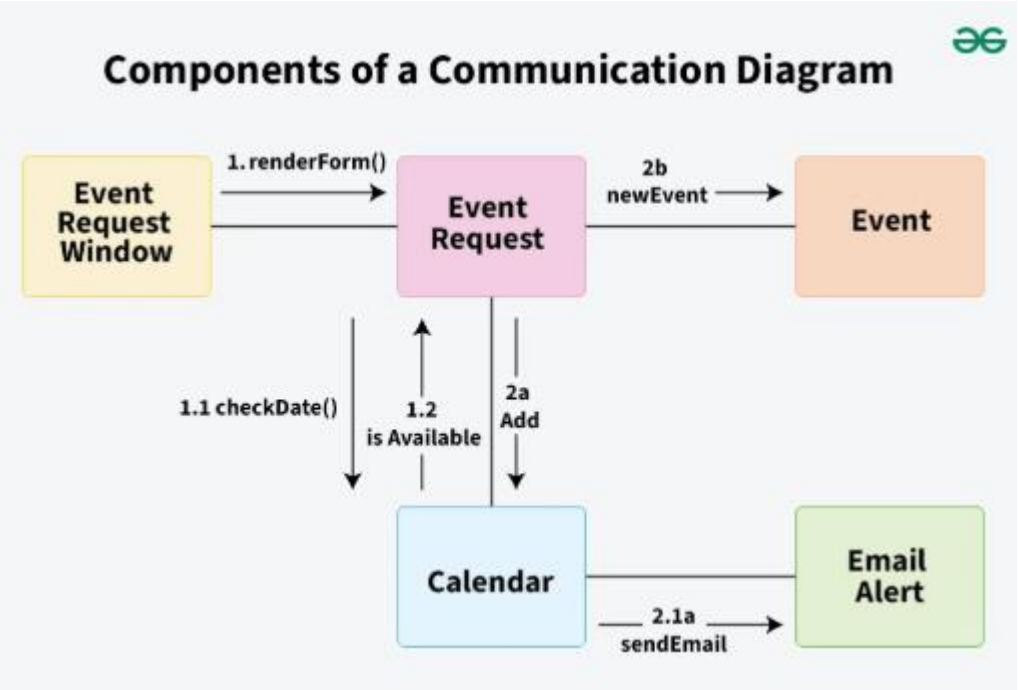
A Communication Diagram in Unified Modeling Language (UML) visually represents the interactions between objects or components in a system. It focuses on how messages are exchanged between these elements, highlighting the flow of information in a sequence. By illustrating both the structural and behavioral aspects of a system, communication diagrams offer a clear understanding of object relationships and message paths, making them essential for modeling dynamic interactions in software design and development. This article explores the structure, significance, and practical applications of communication diagrams in UML.

What are Communication Diagrams?

Speaking of communication diagrams, they are also called collaboration diagrams, which are UML structures used for the definition of interacting objects. They are concerned with the relationship between two objects and the sequence of messages passed between the two to realize a specific functionality.

- While the sequence diagrams concentrate more on the time variations in the interactions, the communication diagrams focus on the structural facet and connections between objects.
- They are also helpful in compounding and decomposing the behavior of a system during design and analysis by tooting their cogs and wheels as to how they work together to achieve the ultimate aim of the system.

Components of a Communication Diagram
A communication diagram consists of several key components:



1. Objects: Objects are described as rectangles with the name of the object on them. They are the very subjects that perform actions with regard to each other.
2. Links: Several straight lines depict the association and transmission of information between various items.
3. Messages: Arrows are to be placed on the link to show the direction of the communication, often accompanied by the message name and sequence number.
4. Interaction Occurrences: Such may refer to situations or cases where certain social interactions are done.
5. Roles: Usually drawn as threads or individuals that interconnect to demonstrate how one part or various users relate to another.

Use Cases of Communication Diagram

Here are five use cases for Communication Diagrams in the context of Unified Modeling Language (UML):

- System Design: Communication Diagrams assist the designers in being able to have a graphical representation of objects that make up its system and more to that, how they interconnect. These are constructional, used for creating the system's blueprint and guaranteeing that elements are integrated optimally.
- Interaction Analysis: These diagrams are used in the analysis and documentation of how objects in the system interact with each other to give an adequate view of the messages that are exchanged.
- Requirements Validation: Communication Diagrams help in the validation of requirements since the use cases are depicted in the manner in which objects interact and all the requirements are fulfilled.
- Troubleshooting: It aids in giving out possible problems in system exchanges thus it locates the interactions and potential problems in communication.
- Documentation: While developing Communication Diagrams are useful because they give detailed documentation to the developers and stakeholders on how the objects interact and helps in comprehending the system.

Timing Diagram

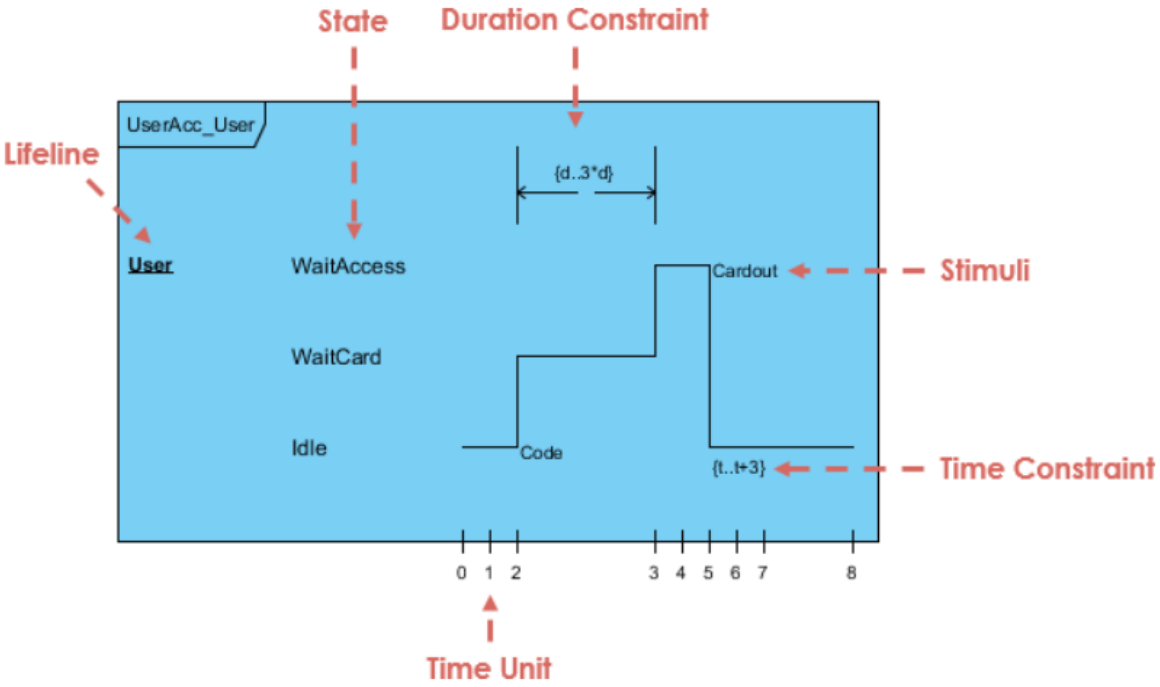
Timing diagrams focus on conditions changing within and among lifelines along a linear time axis. Timing Diagrams describe behavior of both

individual classifiers and interactions of classifiers, focusing attention on time of occurrence of events causing changes in the modeled conditions of the Lifelines.

State Timeline Representation

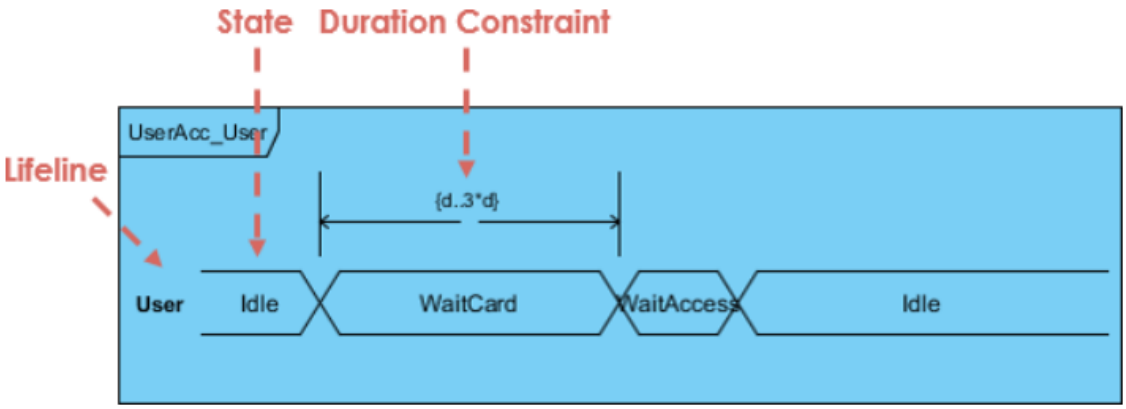
Changes from one state to another are represented by a change in the level of the lifeline. For the period of time when the object is a given state, the timeline runs parallel to that state. A change in state appears as a vertical change from one level to another. The cause of the change, as is the case in a state or sequence diagram, is the receipt of a message, an event that causes a change, a condition within the system, or even just the passage of time.

,…



Value lifeline Representation

The figure below shows an alternative notation of UML Timing diagram. It shows the state of the object between two horizontal lines that cross with each other each time the state changes.

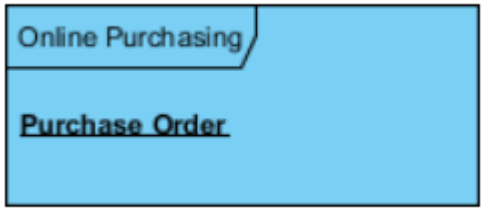


Basic Concepts of Timing Diagrams

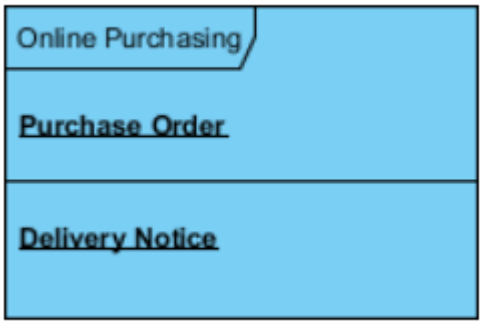
Major elements of timing UML diagram - lifeline, timeline, state or condition, message, duration constraint, timing ruler.

Lifeline

A lifeline in a Timing diagram forms a rectangular space within the content area of a frame. Lifeline is a named element which represents an individual participant in the interaction. It is typically aligned horizontally to read from left to right.

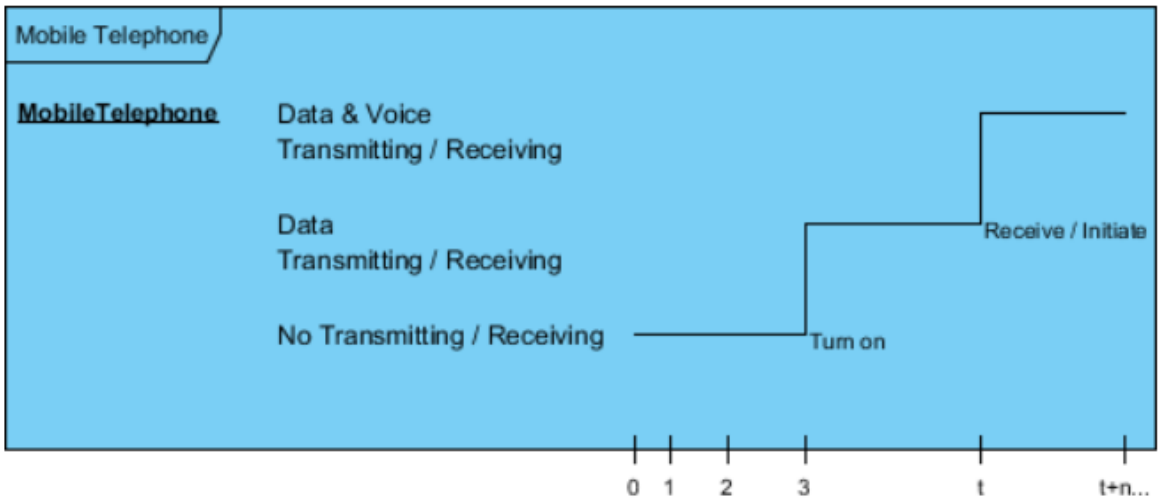


Multiple lifelines may be stacked within the same frame to model the interaction between them.



State Timeline in Timing Diagram

A state or condition timeline represents the set of valid states and time. The states are stacked on the left margin of the lifeline from top to bottom.



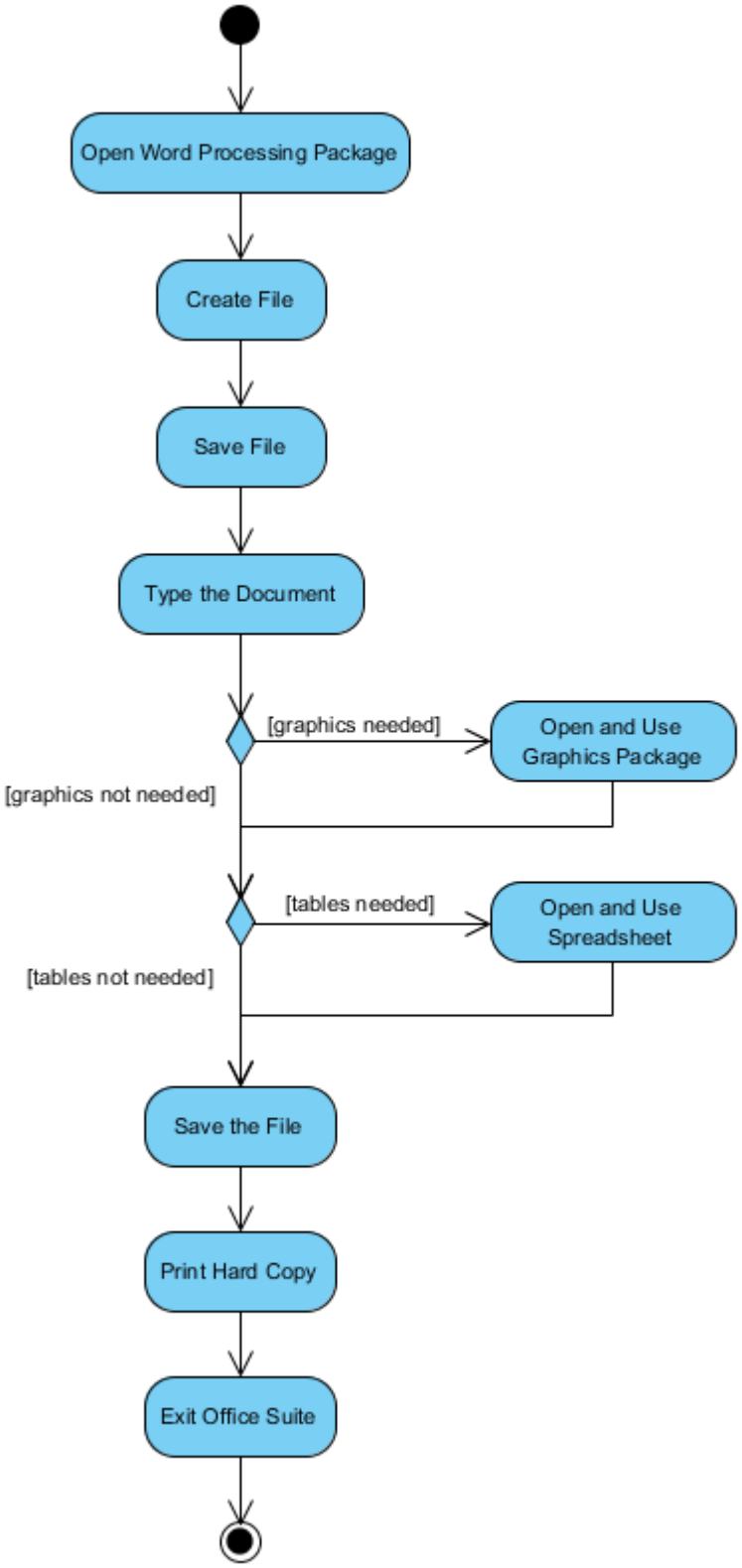
The cause of the change, as is the case in a state or sequence diagram, is the receipt of a message, an event that causes a change, a condition within the system, or even just the passage of time.

Design Using UML Activity Diagram

Activity Diagram - Modeling a Word Processor

The activity diagram example below describes the workflow for a word process to create a document through the following steps:

- Open the word processing package.
- Create a file.
- Save the file under a unique name within its directory.
- Type the document.
- If graphics are necessary, open the graphics package, create the graphics, and paste the graphics into the document.
- If a spreadsheet is necessary, open the spreadsheet package, create the spreadsheet, and paste the spreadsheet into the document.
- Save the file.
- Print a hard copy of the document.
- Exit the word processing package.



Introduction to Patterns General Responsibility Assignment Software Patterns (GRASP)

In Object-Oriented Analysis and Design (OOAD), General Responsibility Assignment Software Patterns (GRASP) play a crucial role in designing effective and maintainable software systems. GRASP offers a set of

guidelines to aid developers in assigning responsibilities to classes and objects in a way that promotes low coupling, high cohesion, and overall robustness. By understanding and applying GRASP principles, developers can create software solutions that are flexible, scalable, and easier to maintain over time.

What are GRASP Principles?

GRASP, which stands for General Responsibility Assignment Software Patterns, includes several principles that guide the allocation of responsibilities in object-oriented design. These principles include:

1. Creator: Assign the responsibility of creating instances of a class to the class that has the most knowledge about when and how to create them.
2. Information Expert: Assign a responsibility to the class that has the necessary information to fulfill it, promoting high cohesion and minimizing coupling.
3. Low Coupling: Aim for classes to have minimal dependencies on each other, facilitating easier maintenance and flexibility in the system.
4. High Cohesion: Ensure that the responsibilities within a class are closely related and focused, enhancing readability, maintainability, and reusability.
5. Controller: Assign the responsibility of handling system events or coordinating activities to a controller class, promoting centralized control and avoiding cluttered classes.
6. Pure Fabrication: Introduce new classes to fulfill responsibilities without violating cohesion and coupling principles, promoting cleaner and more maintainable designs.

7. Indirection: Use intermediaries or abstractions to decouple classes and promote flexibility in design.
8. Polymorphism: Utilize inheritance and interfaces to enable multiple implementations of behaviors, allowing for flexible and extensible systems.

Importance in OOAD

In Object-Oriented Analysis and Design (OOAD), GRASP principles hold significant importance as they provide a framework for designing systems with clarity, flexibility, and maintainability. Here's why they are essential:

- Clarity of Design: GRASP principles help in organizing classes and responsibilities in a way that makes the design more understandable. Clear responsibilities assigned to classes make it easier for developers to comprehend the system's architecture.
- Low Coupling, High Cohesion: GRASP encourages low coupling between classes, meaning that classes are less dependent on each other. This leads to more modular and reusable code. Additionally, high cohesion ensures that each class has a clear and focused purpose, making the system easier to maintain and modify.
- Flexible Design: By following GRASP principles such as Indirection and Polymorphism, the design becomes more flexible and adaptable to changes. Indirection allows for the introduction of intermediaries, which can simplify complex interactions, while Polymorphism enables the use of multiple implementations for behaviors, facilitating extensibility.
- Scalability: GRASP principles contribute to the scalability of the system by promoting a design that can accommodate future changes and enhancements without significant refactoring. This scalability is vital as systems evolve and grow over time.
- Ease of Maintenance: With clear responsibilities assigned to classes and well-defined relationships between them, maintaining the system becomes more straightforward. Developers can quickly identify where changes need to be made and can do so without inadvertently affecting other parts of the system.

GRASP Principles and their Examples
stands for General Responsibility Assignment Software Patterns ,guides in assigning responsibilities to collaborating objects.

9 GRASP patterns

- Creator
- Information Expert
- Low Coupling

- Controller
- High Cohesion
- Indirection
- Polymorphism
- Protected Variations

Pure Fabrication

Responsibility

Responsibility can be: – accomplished by a single object. – or a group of object collaboratively accomplish a responsibility.

● GRASP helps us in deciding which responsibility should be assigned to which object/class.

● Identify the objects and responsibilities from the problem domain, and also identify how objects interact with each other.

● Define blue print for those objects – i.e. class with methods implementing those responsibilities.

Creator

Who creates an Object? Or who should create a new instance of some class?

● "Container" object creates "contained" objects.

● Decide who can be creator based on the objects association and their interaction.

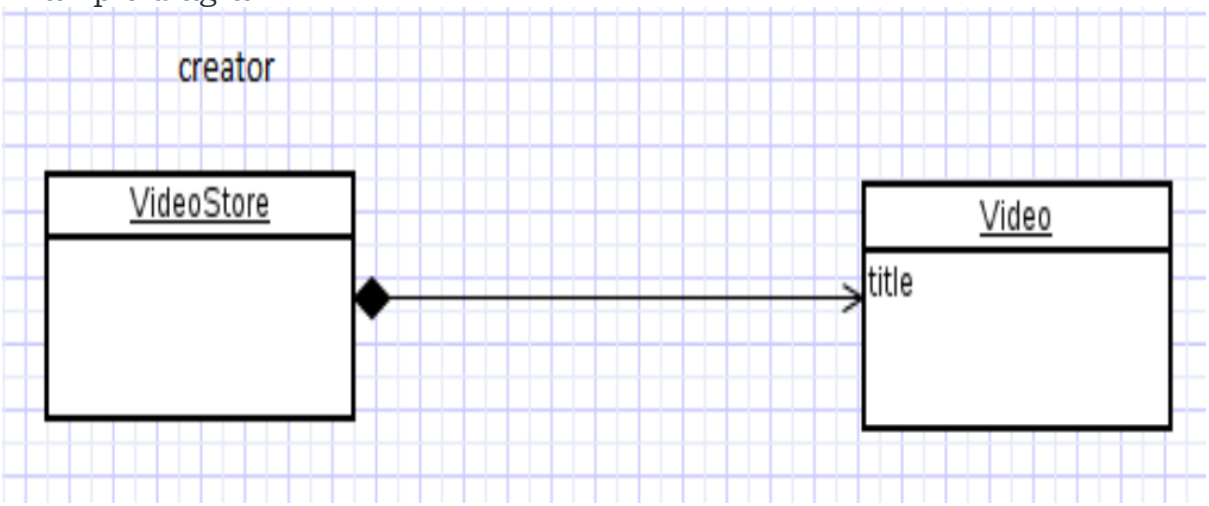
Example for Creator

Consider VideoStore and Video in that store.

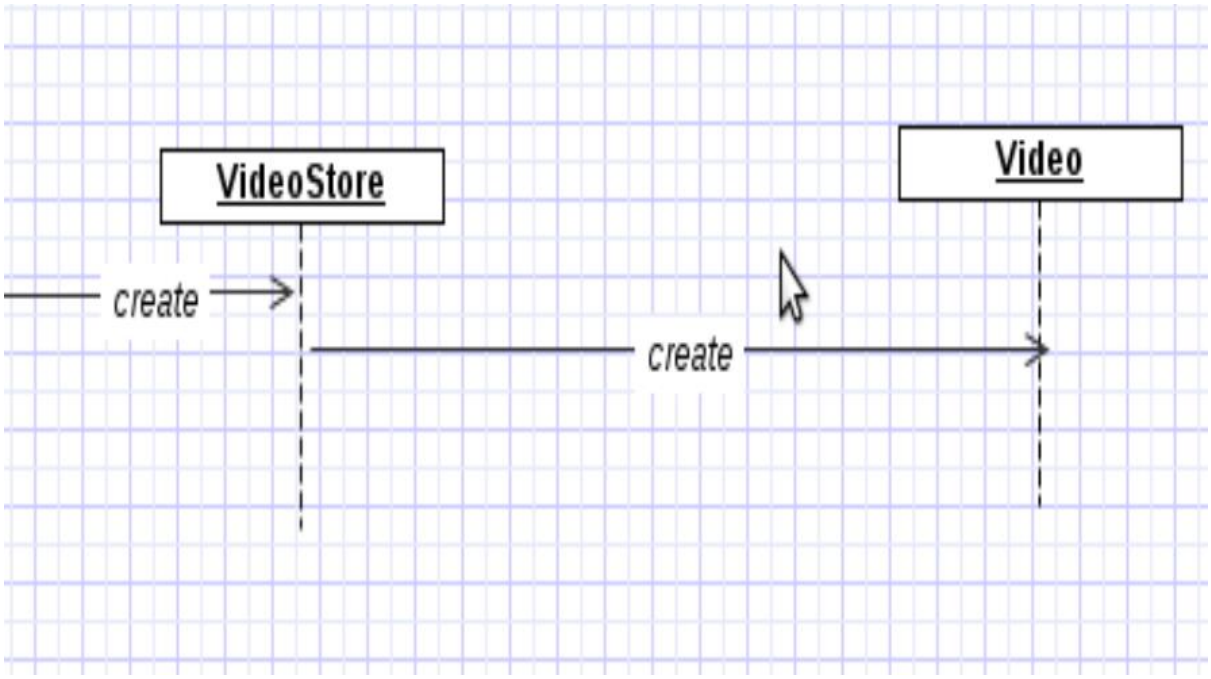
● VideoStore has an aggregation association with Video. I.e, VideoStore is the container and the Video is the contained object.

● So, we can instantiate video object in VideoStore class

Example diagram



Example for creator



Expert

● Given an object o, which responsibilities can be assigned to o?

● Expert principle says – assign those responsibilities to o for which o has the information to fulfill that responsibility.

● They have all the information needed to perform operations, or in some cases they collaborate with others to fulfill their responsibilities.

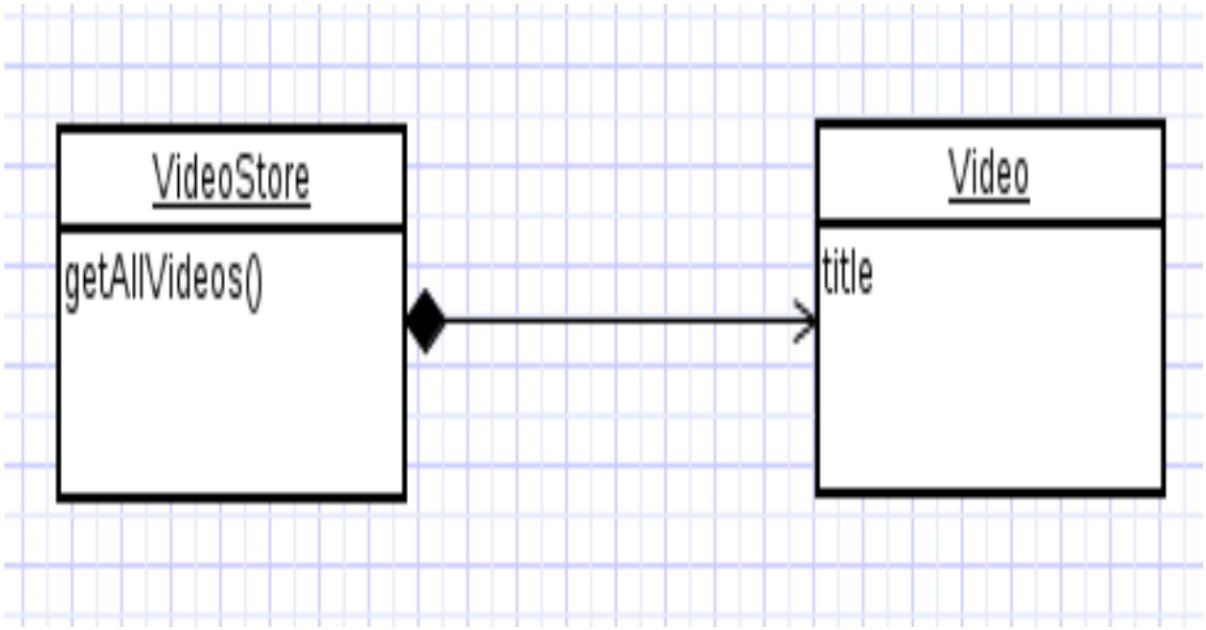
Example for Expert

● Assume we need to get all the videos of a VideoStore.

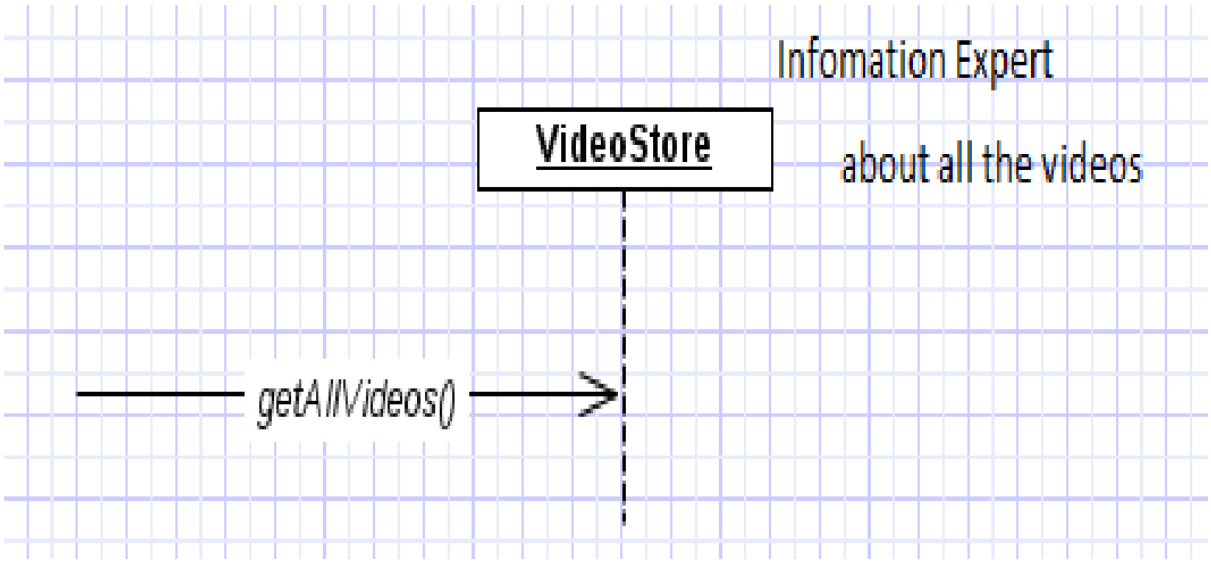
● Since VideoStore knows about all the videos, we can assign this responsibility of giving all the videos can be assigned to VideoStore class.

● VideoStore is the information expert.

Example for Expert



Example for Expert



Low Coupling

- How strongly the objects are connected to each other?
- Coupling – object depending on other object.
- When depended upon element changes, it affects the dependant also.
- Low Coupling – How can we reduce the impact of change in depended upon elements on dependant elements.
- Prefer low coupling – assign responsibilities so that coupling remain low.
- Minimizes the dependency hence making system maintainable, efficient and code reusable

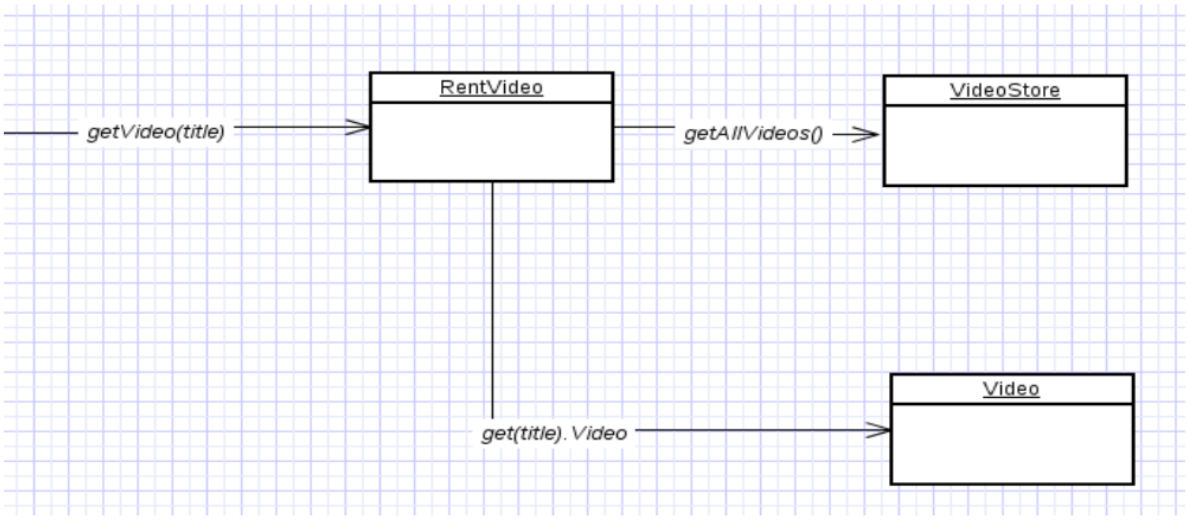
Low coupling

● Two elements are coupled, if

– One element has aggregation/composition association with another element.

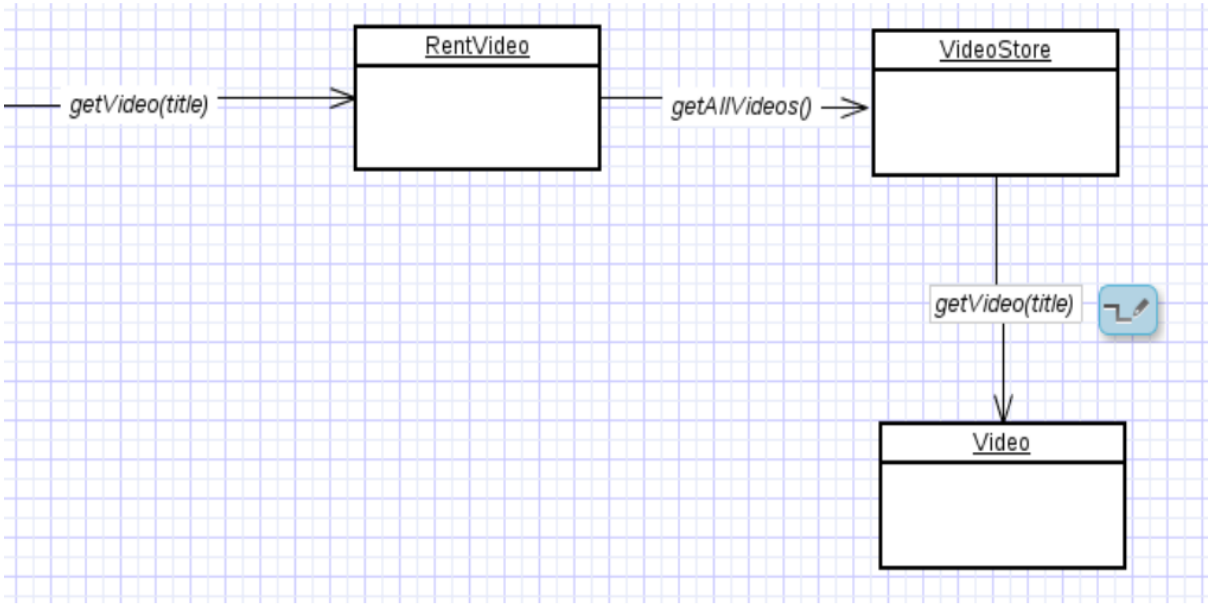
– One element implements/extends other element.

Example of poor coupling



here class Rent knows about both VideoStore and Video objects. Rent is depending on both the classes

Example for low coupling ● VideoStore and Video class are coupled, and Rent is coupled with VideoStore. Thus providing low coupling.



Controller

● Deals with how to delegate the request from the UI layer objects to domain layer objects.

● when a request comes from UI layer object, Controller pattern helps us in determining what is that first object that receive the message from the UI layer objects.

● This object is called controller object which receives request from UI layer object and then controls/coordinates with other object of the domain layer to fulfill the request.

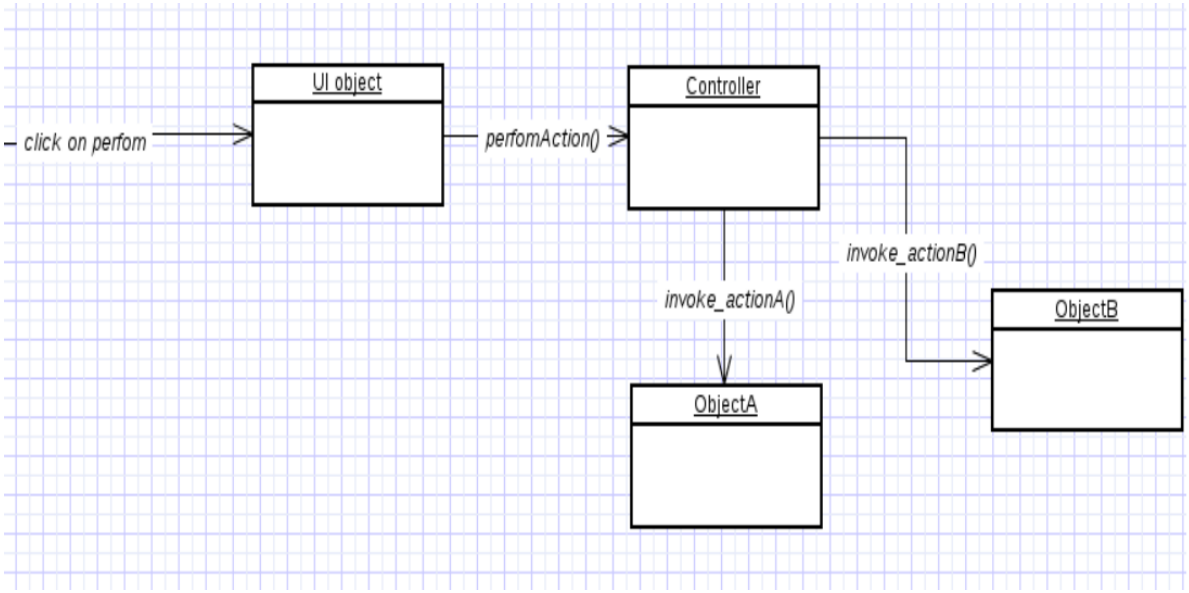
● It delegates the work to other class and coordinates the overall activity

Controller

● We can make an object as Controller, if – Object represents the overall system (facade controller) – Object represent a use case, handling a sequence of operations (session controller).

● Benefits – can reuse this controller class. – Can use to maintain the state of the use case. – Can control the sequence of the activities

Example for Controller



Bloated Controllers

● Controller class is called bloated, if

– The class is overloaded with too many responsibilities. Solution

– Add more controllers

– Controller class also performing many tasks instead of delegating to other class. Solution

– controller class has to delegate things to others.

High Cohesion

● How are the operations of any element are functionally related?

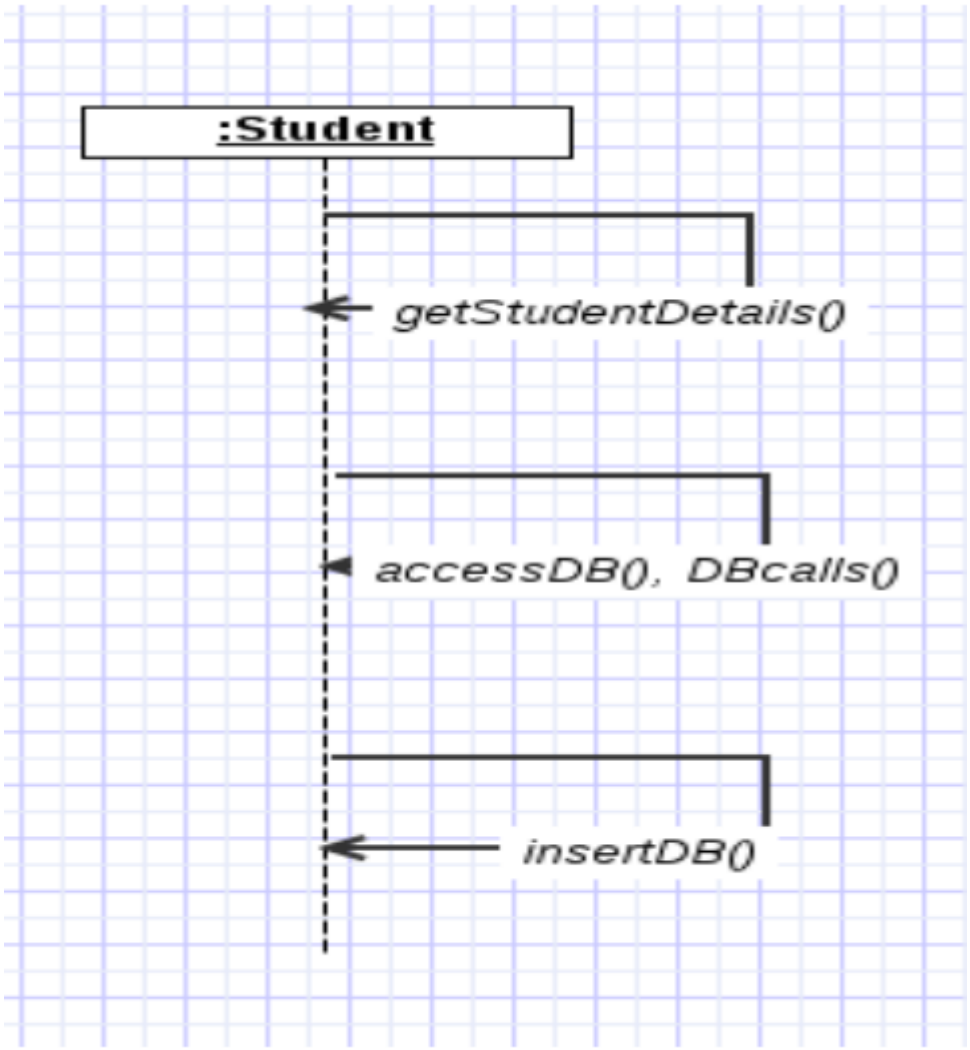
● Related responsibilities in to one manageable unit

● Prefer high cohesion

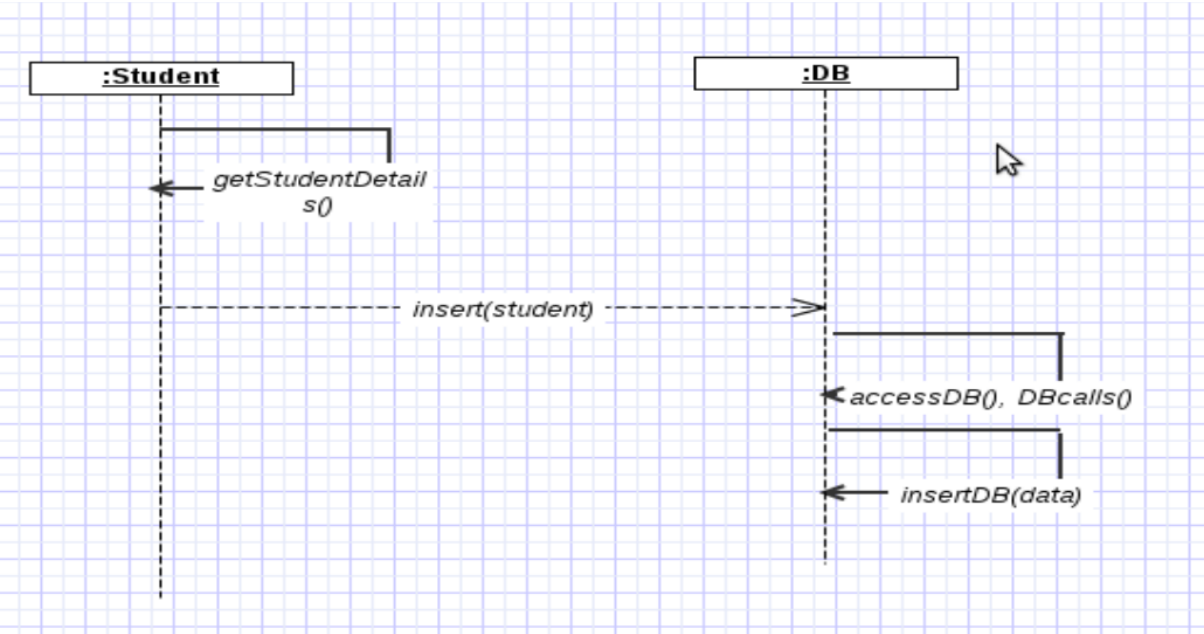
● Clearly defines the purpose of the element

● Benefits – Easily understandable and maintainable. – Code reuse – Low coupling

Example for low cohesion



Example for High Cohesion



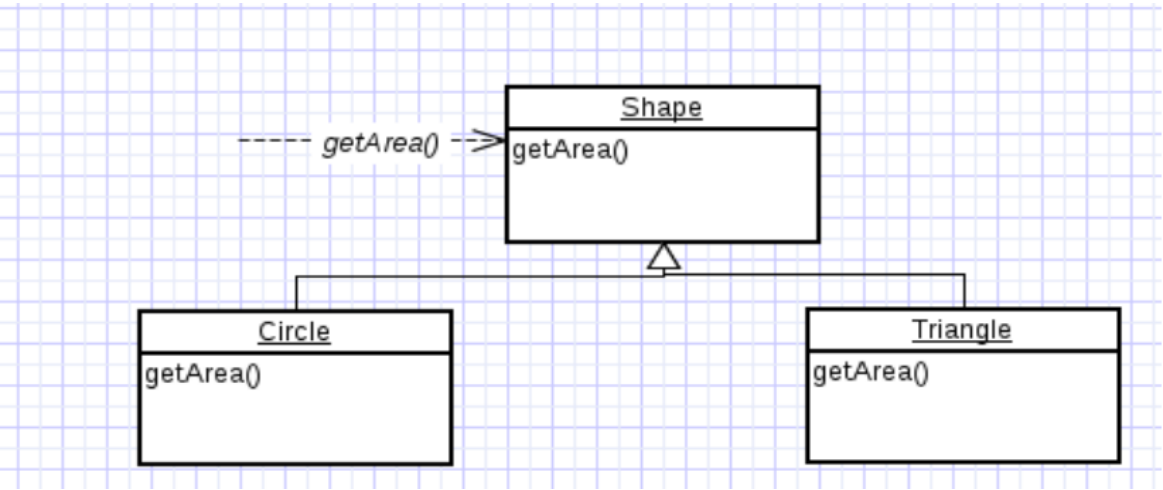
Polymorphism

How to handle related but varying elements based on element type?

● Polymorphism guides us in deciding which object is responsible for handling those varying elements.

● Benefits: handling new variations will become easy

Example for Polymorphism ● the getArea() varies by the type of shape, so we assign that responsibility to the subclasses.



By sending message to the Shape object, a call will be made to the corresponding sub class object – Circle or Triangle.

Pure Fabrication

Fabricated class/ artificial class – assign set of related responsibilities that doesn't represent any domain object.

● Provides a highly cohesive set of activities.

● Behavioral decomposed – implements some algorithm.

● Examples: Adapter, Strategy

● Benefits: High cohesion, low coupling and can reuse this class.

Example

● Suppose we Shape class, if we must store the shape data in a database.

● If we put this responsibility in Shape class, there will be many database related operations thus making Shape incohesive.

● So, create a fabricated class DBStore which is responsible to perform all database operations.

● Similarly logInterface which is responsible for logging information is also a good example for Pure Fabrication.

Indirection

How can we avoid a direct coupling between two or more elements.

● Indirection introduces an intermediate unit to communicate between the other units, so that the other units are not directly coupled.

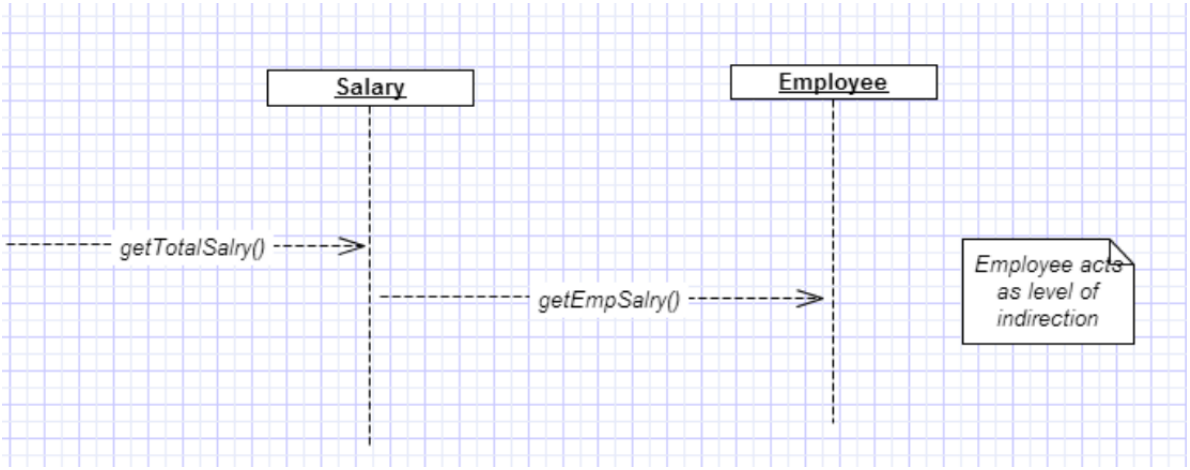
● Benefits: low coupling

● Example: Adapter, Facade, Obserever

Example for Indirection

● Here polymorphism illustrates indirection

● Class Employee provides a level of indirection to other units of the system.



Protected Variation

How to avoid impact of variations of some elements on the other elements.

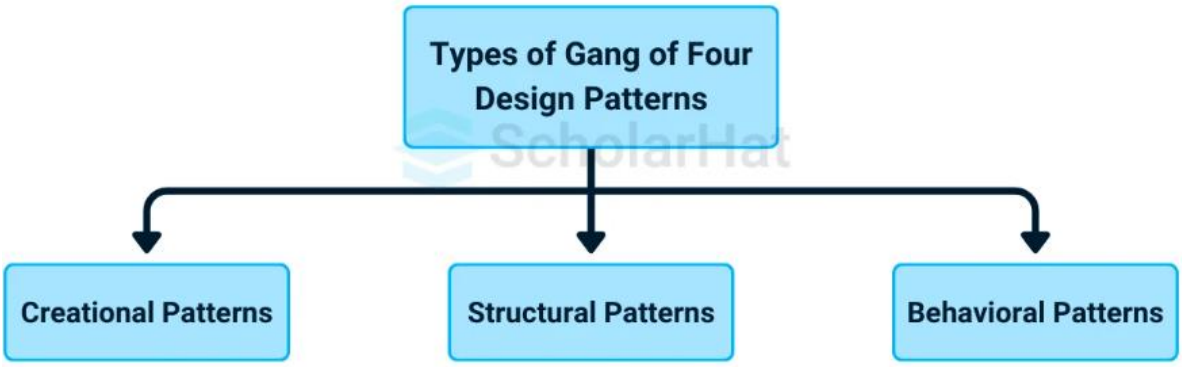
● It provides a well defined interface so that the there will be no affect on other units.

● Provides flexibility and protection from variations.

● Provides more structured design.

● Example: polymorphism, data encapsulation, interfaces

Gang of Four (GoF)



1. Creational Patterns: Attempt to generate things in a manner appropriate to the situation (e.g., Singleton, Factory Method, Abstract Factory, Builder, Prototype).
2. Structural Patterns: Concerned with object composition, specifying how to combine items to achieve additional functionality (e.g., adapter, composite, proxy, flyweight, facade, bridge, decorator).
3. Behavioral Patterns: These patterns describe how objects interact and how responsibilities are distributed (for example, Strategy, Observer, Command, Iterator, Mediator, Memento, State, Visitor, Template Method, Chain of Responsibility, Interpreter).

Creational Design Patterns

Creational design patterns are concerned with the generation of objects, ensuring that they are formed in a situationally appropriate manner and providing flexibility in how objects are instantiated.

1. Singleton Pattern: This design is focused on exclusivity. It assures that each class has only one instance, similar to having a VIP pass to a club. You may access that instance from anywhere, making it useful in cases where your application requires a single point of control or coordination.

2. Structural Design Patterns

- A Structural Design pattern is a formula for combining many objects and classes to create a larger structure.
- It's similar to building a house based on a blueprint.
- These patterns tell us how to incorporate the distinct components of a system in a way that is simple to update or grow without affecting the overall system.

Types of Structural Design Patterns

1.Adapter Pattern: The Adapter Pattern enables one class to interact with another that has a different interface. It serves as a bridge between two incompatible interfaces.

Types of Behavioral Design Patterns

1. State Pattern: The State Pattern allows an entity to adapt its behavior as its internal state changes.
2. Strategy Pattern: The Strategy Pattern entails establishing a set of various algorithms and making it easy to select and switch between them while a program is executing.

UNIT-V

Component Diagram, Interfaces and ports, Deployment diagrams, Need, purpose & application of above diagrams two, three tier architecture, Concept of Forward Engineering and Reverse Engineering of UML Diagrams Development stages, Development life cycle, devising a system concepts, Elaborating a concept. Preparing problem statements, Overview of analysis, Domain class models, Domain state model, Domain Interaction model.

A component is a physical thing that conforms to and realizes a set of interfaces. Interfaces therefore bridge your logical and physical models.
Component Diagrams

• Component diagrams are used in modeling the physical aspects of object-oriented systems.

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

• Component diagrams are used to model the static implementation view of a system.

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

• Graphically, a Component diagram is a collection of vertices and arcs.

• Component diagrams are used for visualizing, specifying, and documenting componentbased systems and also for constructing executable systems through forward and reverse engineering.

• Component diagrams commonly contain Components, Interfaces and Dependency, generalization, association, and realization relationships. It may also contain notes and constraints.

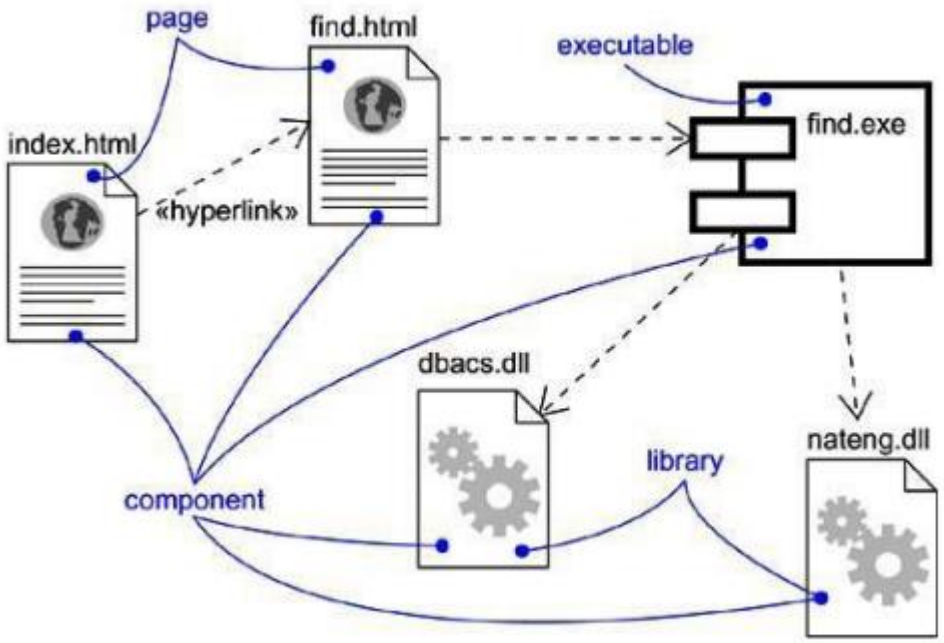


Figure : Component Diagram

Common Modeling Techniques Modeling Source Code To model a system's source code,

• Either by forward or reverse engineering identifies the set of source code files of interest and model them as components stereotyped as files.

• For larger systems, use packages to show groups of source code files.

• Consider exposing a tagged value indicating such information as the version number of the source code file, its author, and the date it was last changed. Use tools to manage the value of this tag.

• Model the compilation dependencies among these files using dependencies. Again, use tools to help generate and manage these dependencies.

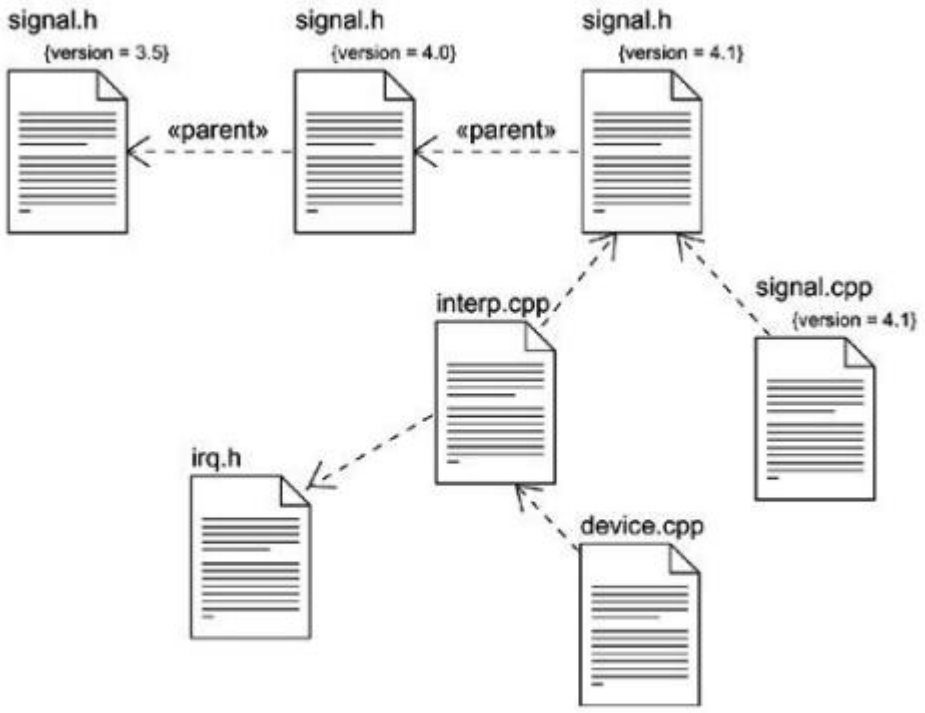


Figure : Modeling Source Code

Common Modeling Techniques

Modeling Source Code To model a system's source code,

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

 For larger systems, use packages to show groups of source code files.

 Consider exposing a tagged value indicating such information as the version number of the source code file, its author, and the date it was last changed. Use tools to manage the value of this tag.

 Model the compilation dependencies among these files using dependencies. Again, use tools to help generate and manage these dependencies.

Modeling an Executable Release

To model an executable release,

 Identify the set of components you'd like to model. Typically, this will involve some or all the components that live on one node, or the distribution of these sets of components across all the nodes in the system.

 Consider the stereotype of each component in this set. For most systems, you'll find a small number of different kinds of components (such as executables, libraries, tables, files, and documents). You can use the UML's extensibility mechanisms to provide visual cues(clues) for these stereotypes.

 For each component in this set, consider its relationship to its neighbors. Most often, this will involve interfaces that are exported (realized) by certain components and then imported (used) by others. If you want to expose the seams in your system, model these interfaces explicitly. If you want your model at a higher level of abstraction, elide these relationships by showing only dependencies among the components.

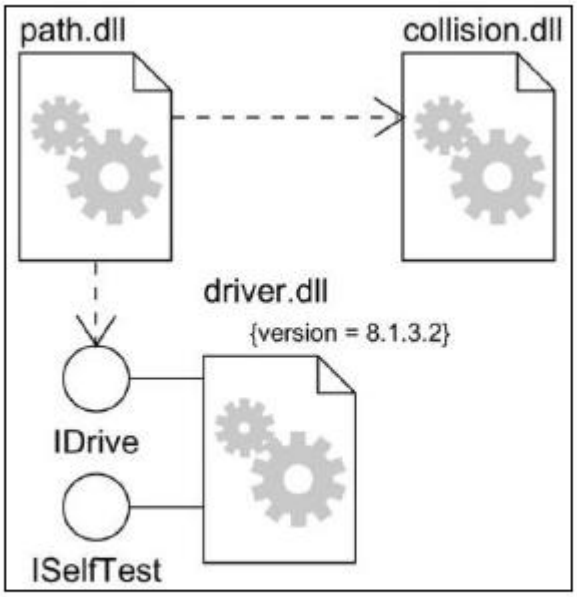


Figure : Modeling an Executable Release

Modeling a Physical Database

To model a physical database,

 Identify the classes in your model that represent your logical database schema.

 Select a strategy for mapping these classes to tables. You will also want to consider the physical distribution of your databases. Your mapping strategy

will be affected by the location in which you want your data to live on your deployed system.

 To visualize, specify, construct, and document your mapping, create a component diagram that contains components stereotyped as tables.

 Where possible, use tools to help you transform your logical design into a physical design.

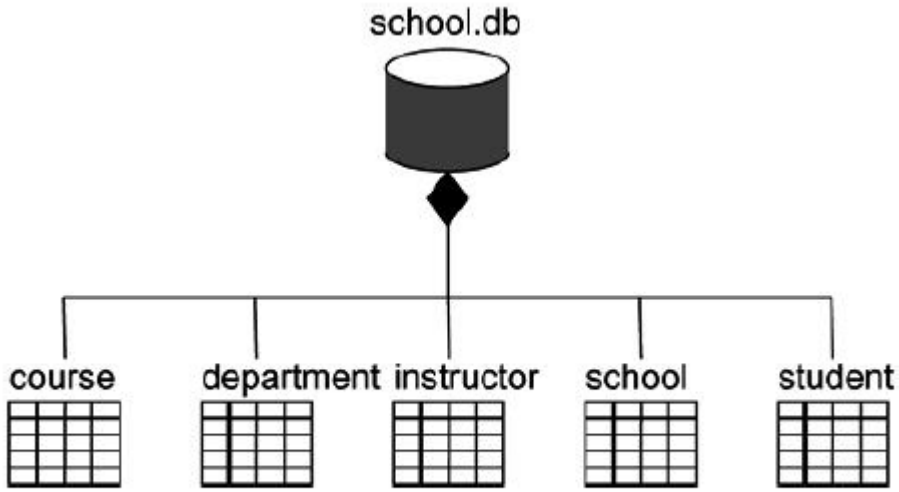


Figure : Modeling a Physical Database

Modeling Adaptable Systems

To model an adaptable system,

 Consider the physical distribution of the components that may migrate from node to node. You can specify the location of a component instance by marking it with a location tagged value, which you can then render in a component diagram (although, technically speaking, a diagram that contains only instances is an object diagram).

 If you want to model the actions that cause a component to migrate, create a corresponding interaction diagram that contains component instances. You can illustrate a change of location by drawing the same instance more than once, but with different values for its location tagged value.

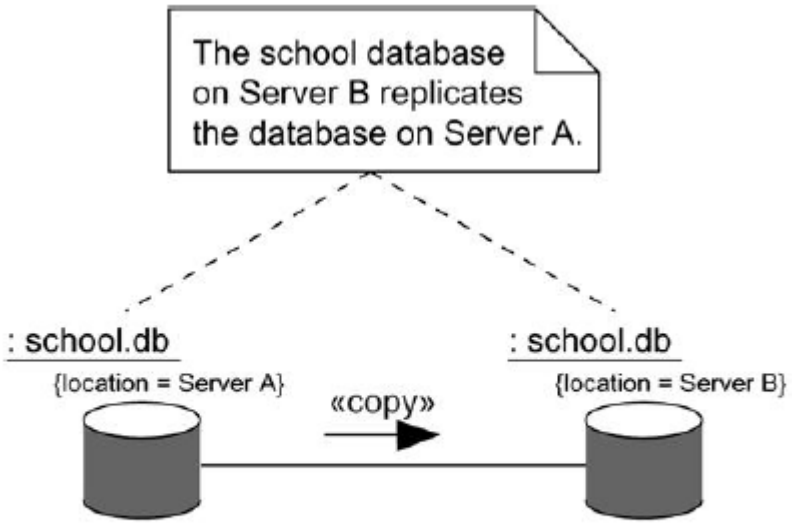
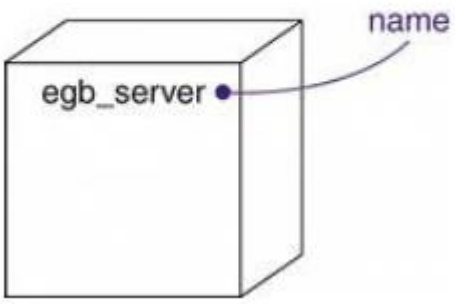


Figure : Modeling Adaptable Systems

Deployment

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

Figure : Nodes



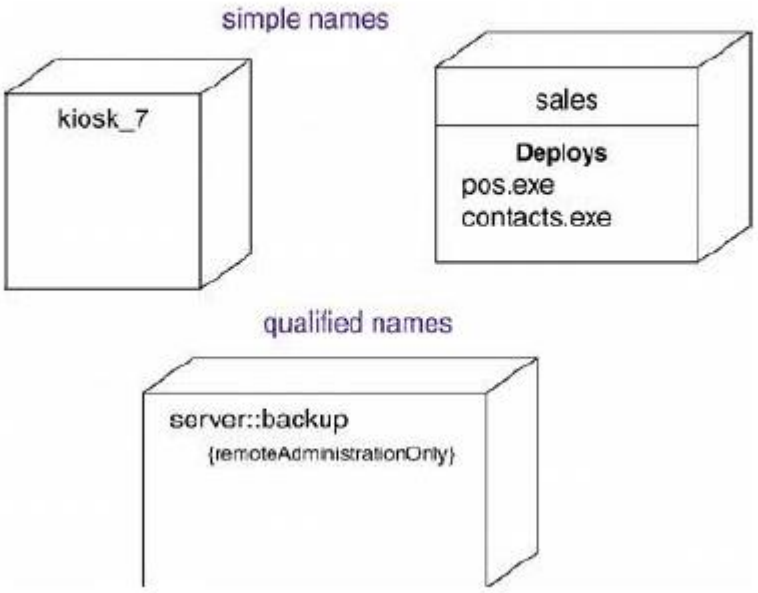
A *node* is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. Graphically, a node is rendered as a cube.

Names

Every node must have a name that distinguishes it from other nodes. A *name* is a textual string. That name alone is known as a *simple name*; a

qualified name is the node name prefixed by the name of the package in which that node lives.

Figure Nodes with Simple and Qualified Names



Nodes and components

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

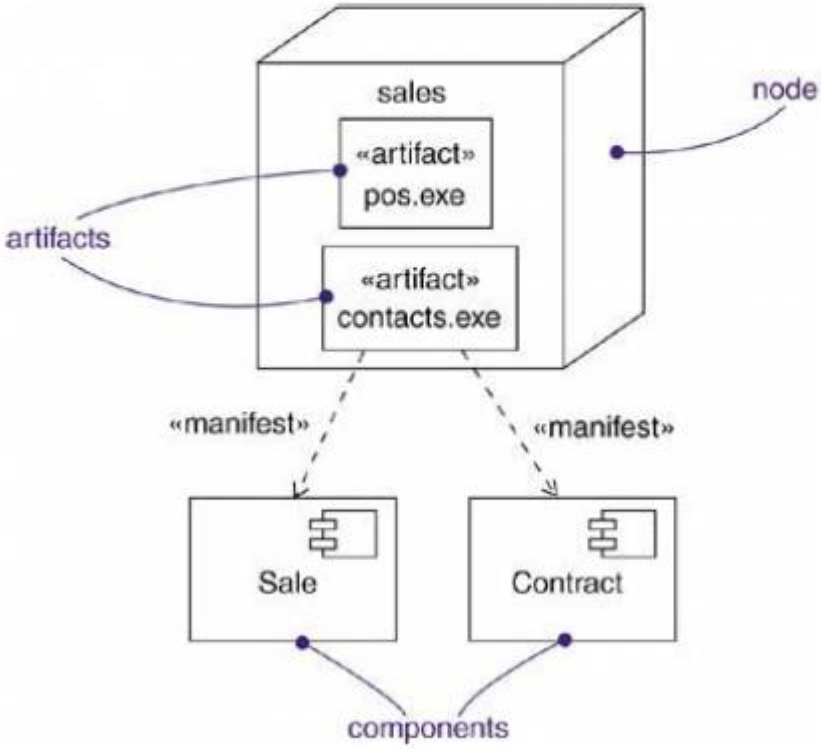
 Components are things that participate in the execution of a system; nodes are things that execute components.

 Components represent the physical packaging of otherwise logical elements; nodes represent the physical deployment of components.

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

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

Figure: Nodes and Components



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

Organizing Nodes

 You can organize nodes by grouping them in packages in the same manner in which you can organize classes and components.

 You can also organize nodes by specifying dependency, generalization, and association (including aggregation) relationships among them.

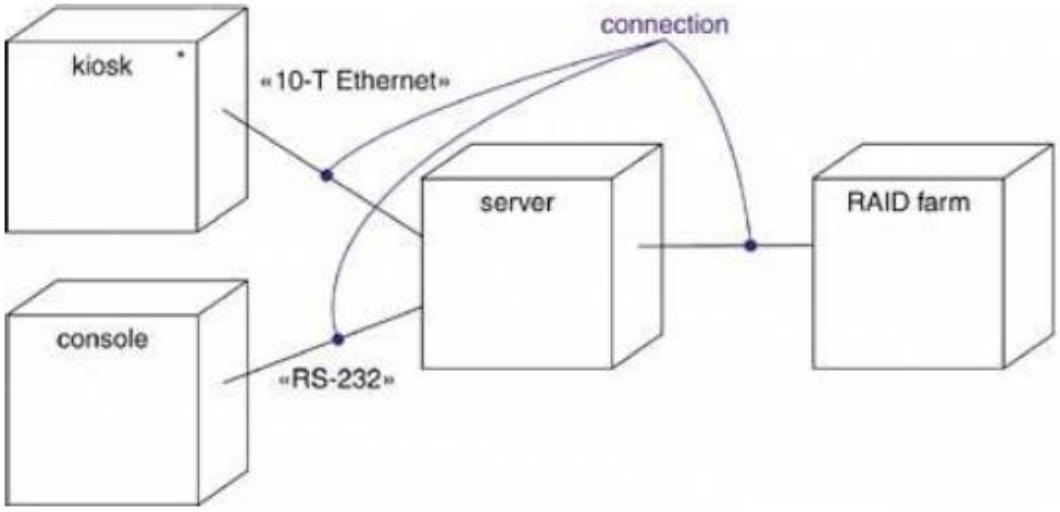
Connections

The most common kind of relationship use among nodes is an association. In this context, an association represents a physical connection among nodes, such as an Ethernet connection, a serial line, or a shared bus.

We can include roles, multiplicity, and constraints.

Figure : Connections

Figure : Connections



Common Modeling Techniques

Modeling Processors and Devices

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

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

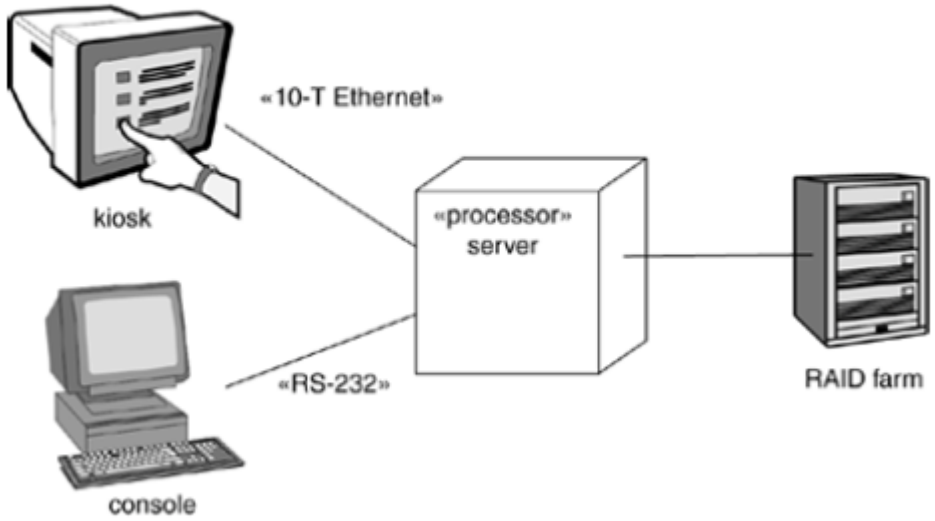
A *device* is a node that has no processing capability (at least, none that are modeled at this level of abstraction) and, in general, represents something that interfaces to the real world.

To model processors and devices,Identify the computational elements of your system's deployment view and model each as a node.

 If these elements represent generic processors and devices, then stereotype them as such. If they are kinds of processors and devices that are part of the vocabulary of your domain, then specify an appropriate stereotype with an icon for each.

 As with class modeling, consider the attributes and operations that might apply to each node.

Figure : Processors and Devices



Modeling the Distribution of Components

To model the distribution of components,

 For each significant component in your system, allocate it to a given node.

 Consider duplicate locations for components. It's not uncommon for the same kind of component (such as specific executables and libraries) to reside on multiple nodes simultaneously.

 Render this allocation in one of three ways.

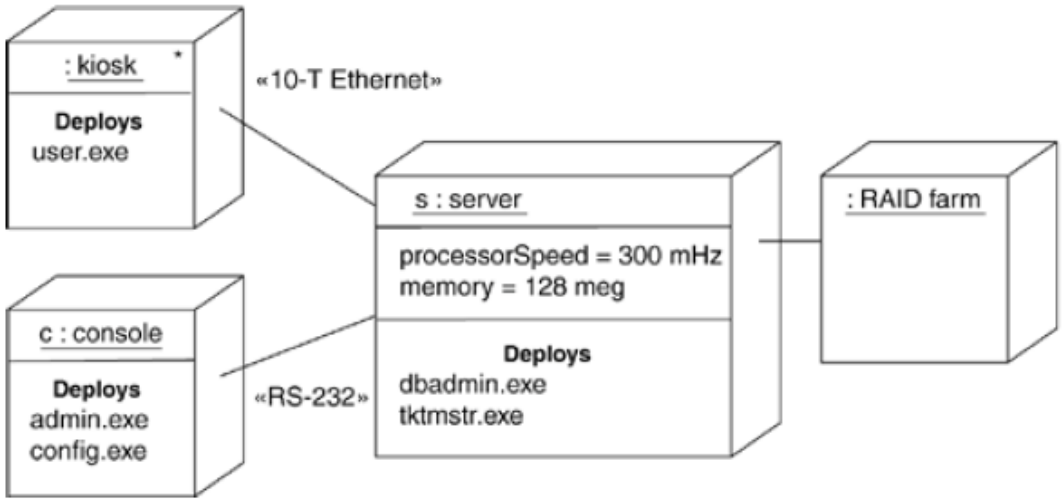
1. Don't make the allocation visible, but leave it as part of the backplane of your modelthat is, in each node's specification.

2. Using dependency relationships, connect each node with the components it deploys.

3. List the components deployed on a node in an additional compartment.

Figure: Modeling the Distribution of Components

Figure: Modeling the Distribution of Components



Deployment Diagrams

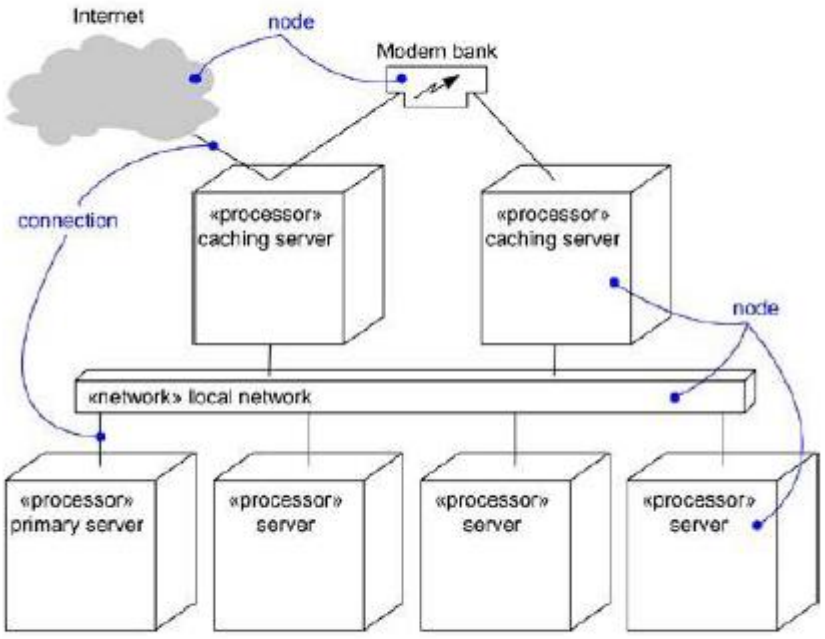


Figure : Deployment Diagram

Deployment Diagrams

 A deployment diagram is a diagram that shows the configuration of run time processing nodes and the components that live on them.

 Deployment diagrams are one of the two kinds of diagrams used in modeling the physical aspects of an object-oriented system.

 used to model the static deployment view of a system (topology of the hardware)

 A deployment diagram is just a special kind of class diagram, which focuses on a system's nodes.

 Graphically, a deployment diagram is a collection of vertices and arcs.

 Deployment diagrams commonly contain Nodes and Dependency & association relationships. It may also contain notes and constraints.

Deployment diagrams are important for visualizing, specifying, and documenting embedded, client/server, and distributed systems and also for managing executable systems through forward and reverse engineering.

Common Modeling Techniques

Modeling an Embedded System To model an embedded system,

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

 Provide visual cues, especially for unusual devices, by using the UML's extensibility mechanisms to define system-specific stereotypes with appropriate icons. At the very least, you'll want to distinguish processors (which contain software components) and devices (which, at that level of abstraction, don't directly contain software).

 Model the relationships among these processors and devices in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.

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

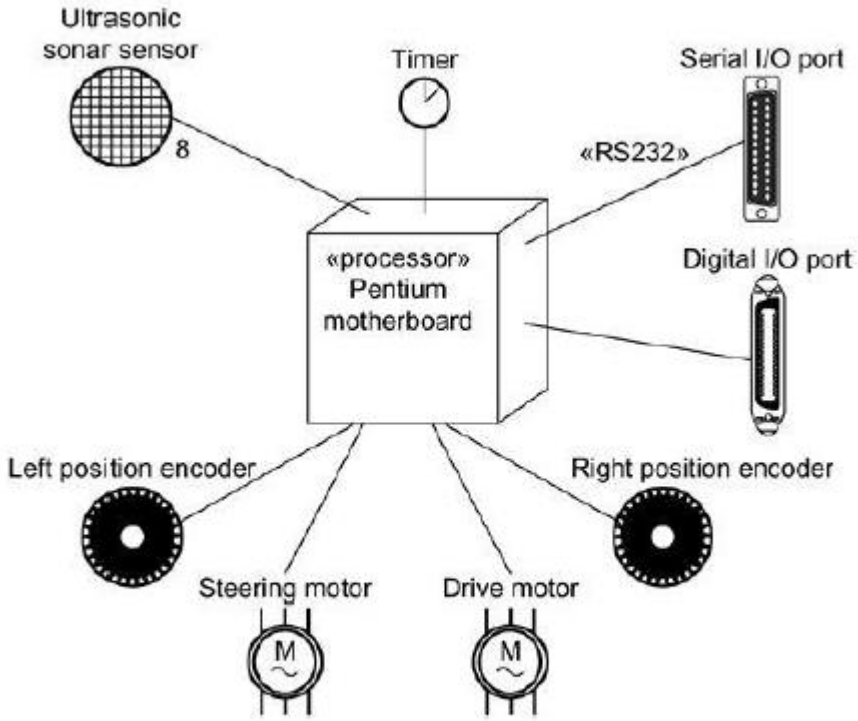


Figure : Modeling an Embedded System

Modeling a Client/Server System To model a client/server system,

 Identify the nodes that represent your system's client and server processors.

 Highlight those devices that are germane to the behavior of your system. For example, you'll want to model special devices, such as credit card readers, badge readers, and display devices other than monitors, because their placement in the system's hardware topology are likely to be architecturally significant.

 Provide visual cues for these processors and devices via stereotyping.

 Model the topology of these nodes in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.

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

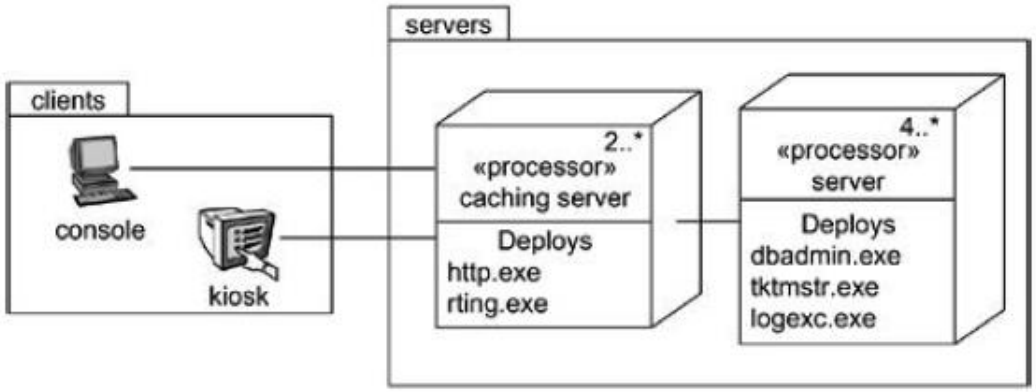


Figure : Modeling a Client/ Server System

Modeling a Fully Distributed System To model a fully distributed system,

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

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

Model these devices and processors using deployment diagrams. Where possible, use tools that discover the topology of your system by walking your system's network. If you need to focus on the dynamics of your system, introduce use case diagrams to specify the kinds of behavior you are interested in, and expand on these use cases with interaction diagrams.

 When modeling a fully distributed system, it's common to reify the network itself as an node. eg:- Internet, LAN, WAN as nodes

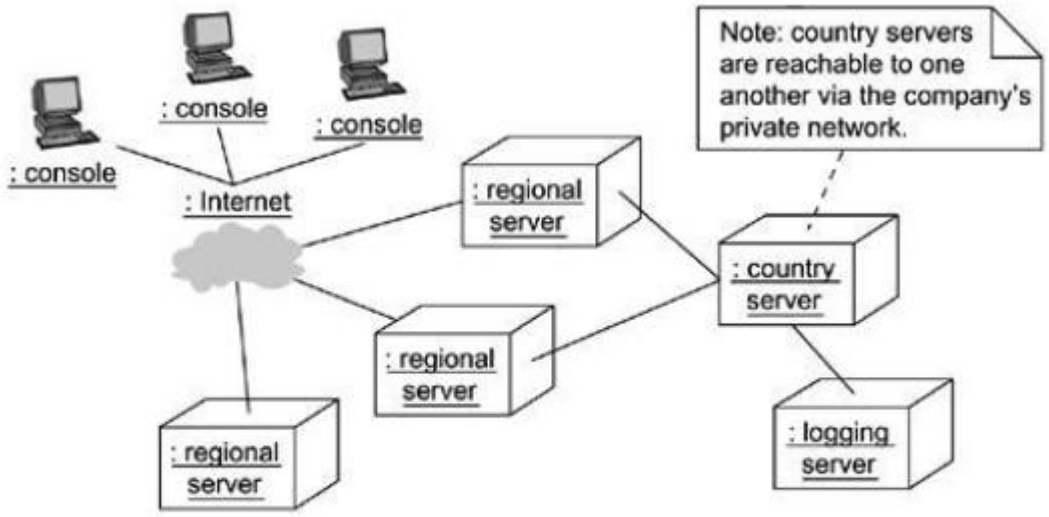


Figure : Modeling a Fully Distributed System