UNIT 5

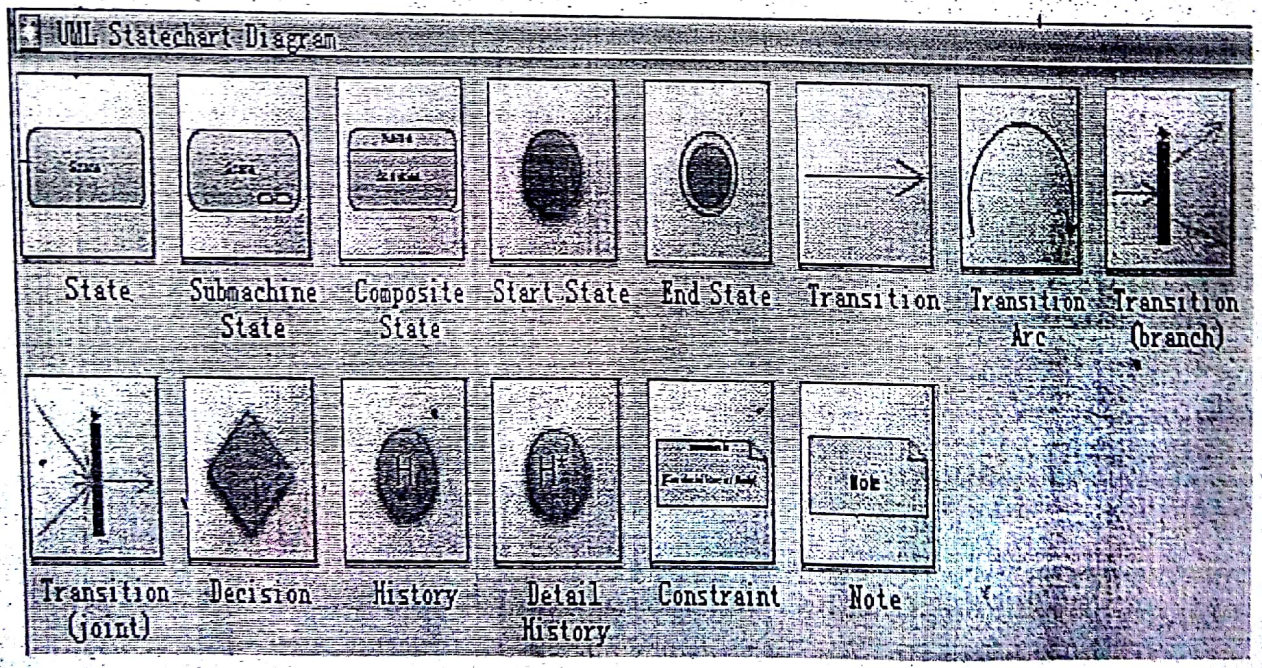
STATE CHART DIAGRAMS

- State chart diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of the systems.
- State chart diagram shows a state machine.
- An activity diagram is a special case of a state chart diagram.
- Both activity and state chart diagrams are useful in modeling the life type of an object.
- Activity diagram shows flow of control from activity to activity.
- State chart diagram shows flow of control from state to state.

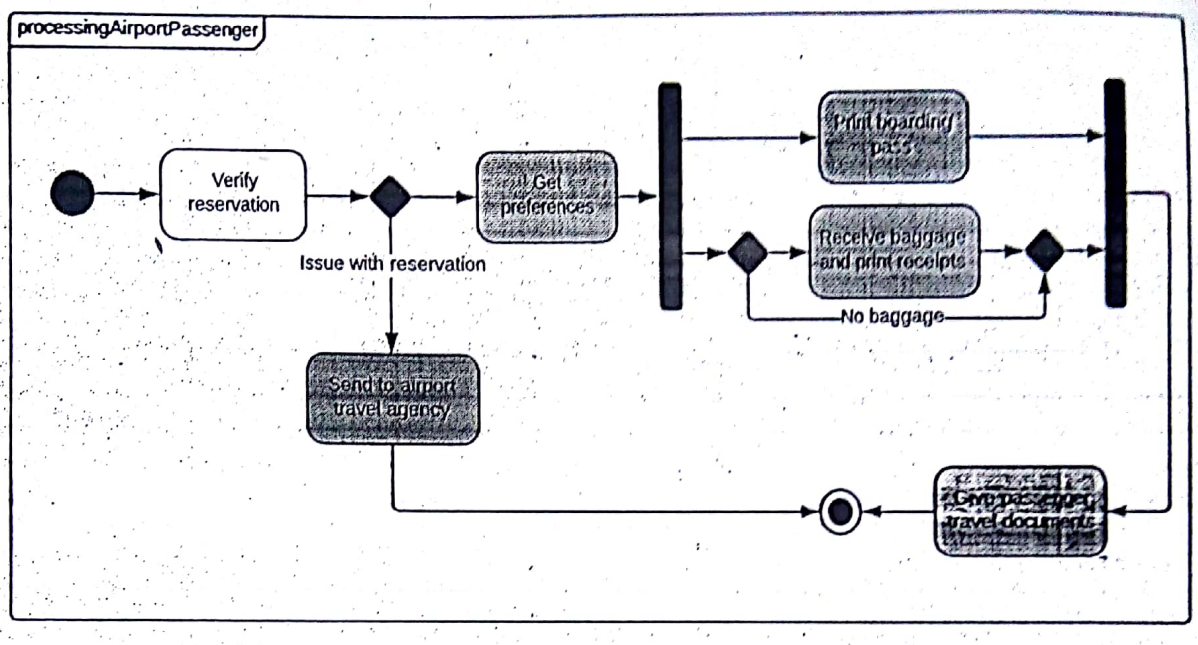
Contents:

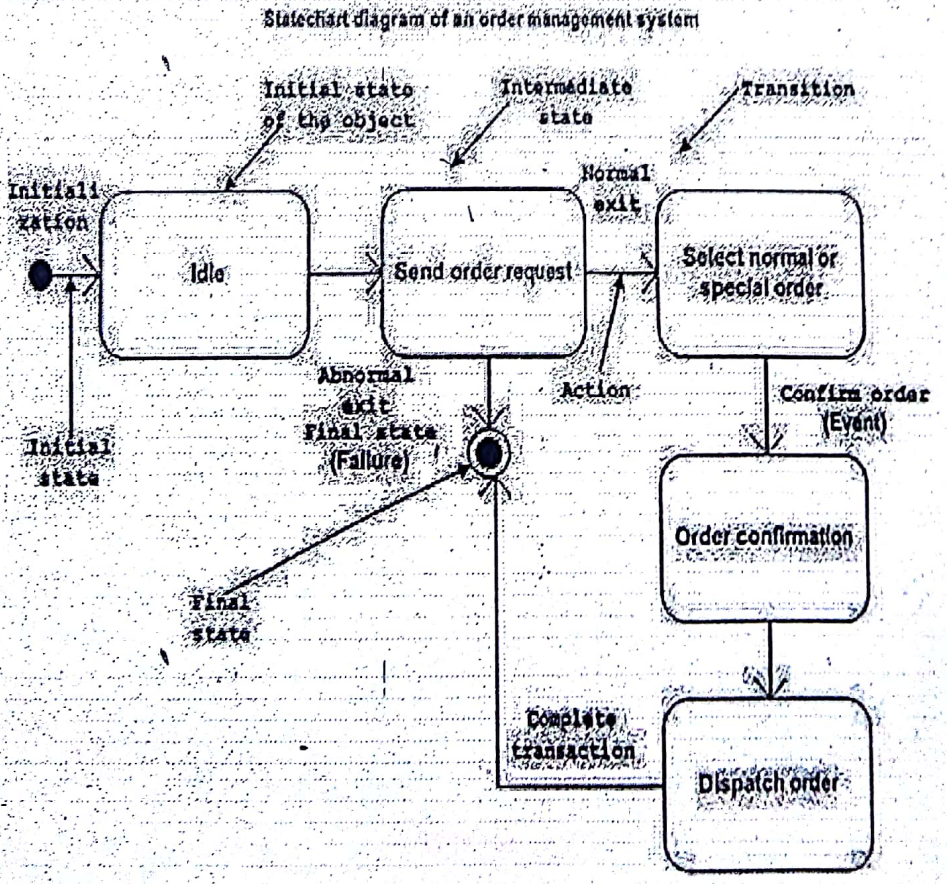
- State chart diagrams commonly contain
  - » Set of states & Complex states
  - » Transitions including events and actions

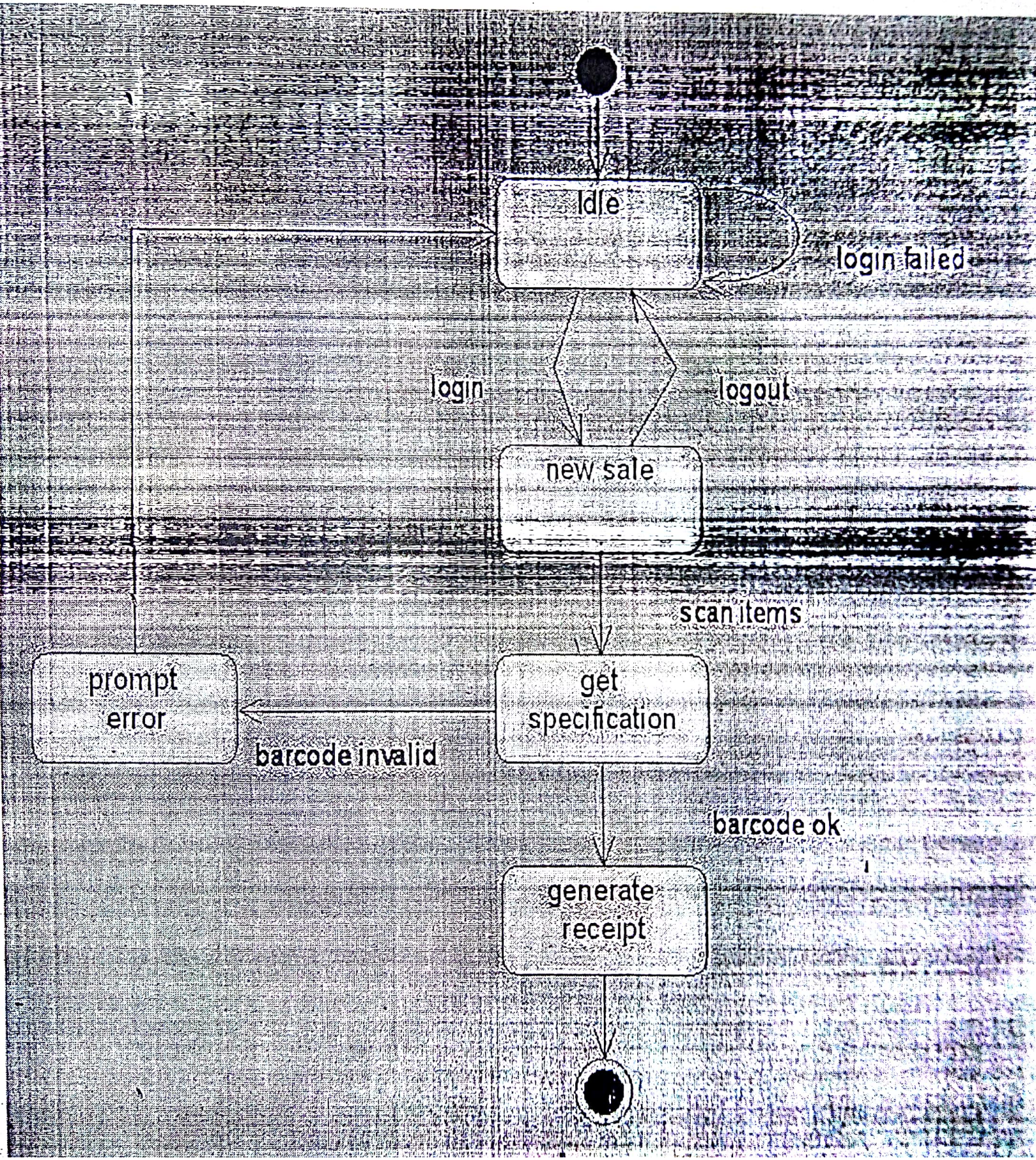
State chart Diagram Symbols:



Examples:

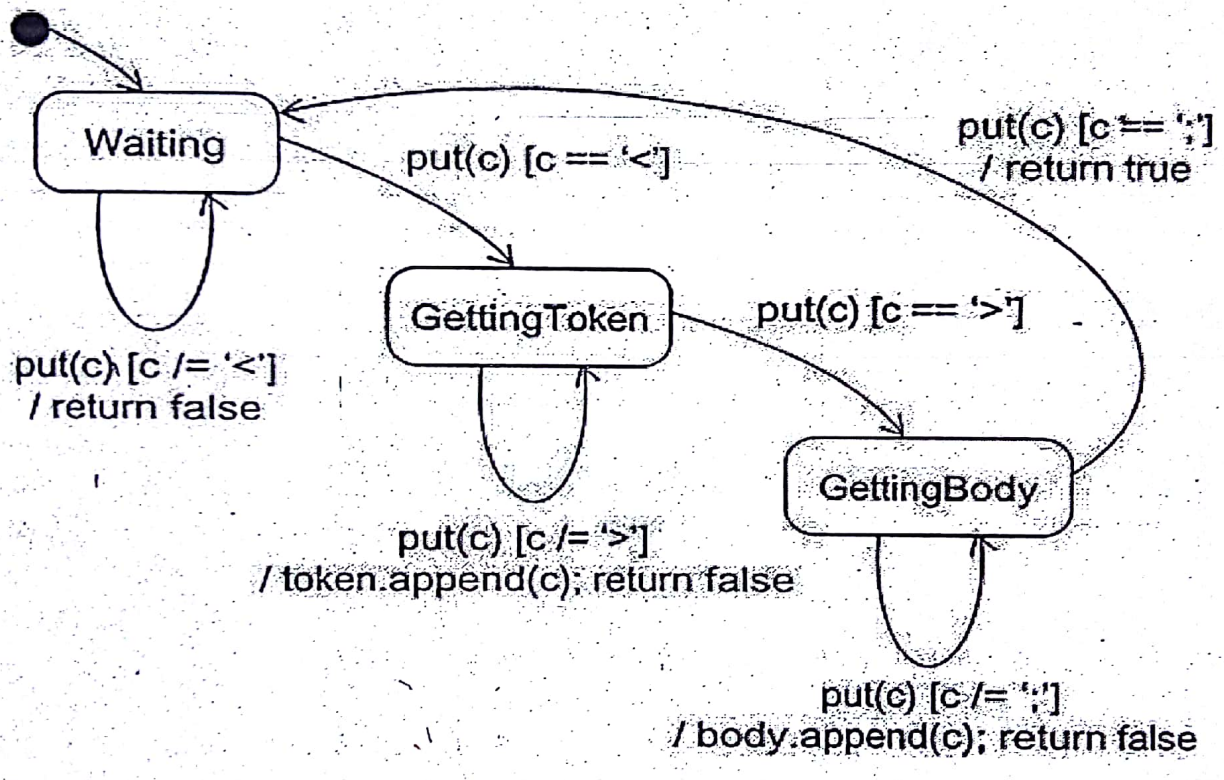

processingAirportPassenger
Verify reservation
Issue with reservation
Get preferences
Print boarding pass
Receive baggage and print receipts
No baggage
Send to airport travel agency


Statechart diagram of an order management system
Initial state of the object
Intermediate state
Transition
Initialization
Idle
Send order request
Normal exit
Select normal or special order
Abnormal exit
Final state (Failure)
Action
Confirm order (Event)
Initial state
Order confirmation
Final state
Complete transaction
Dispatch order


idle
login failed
login
logout
new sale
scan items
prompt error
get specification
barcode invalid
barcode ok
generate receipt

Modeling Reactive Objects:

- To model a reactive object
- For example the state chart diagram for parsing a simple context-free language message : '<'string '>'string ';'
- The first string represents a tag, the second string represents the body of the message.



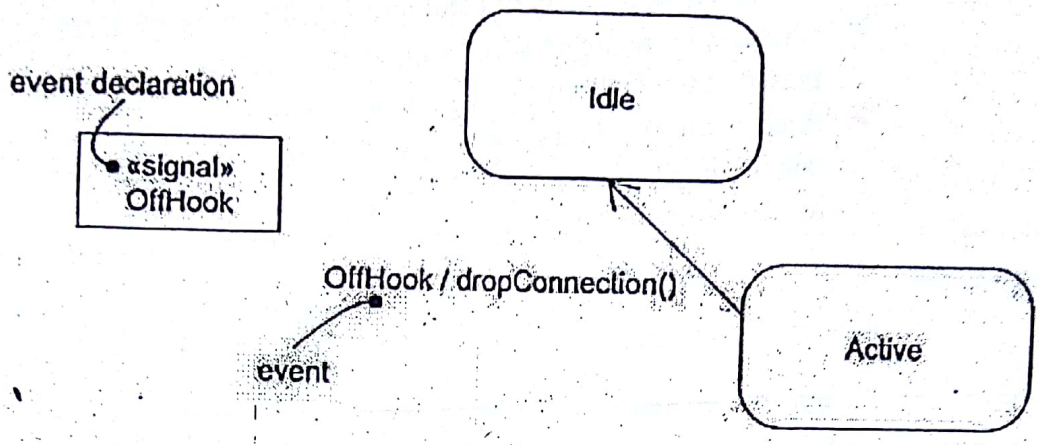
Events

- An event is the specification of a significant occurrence that has a location in time and space.
- Anything that happens is modeled as an event in UML.
- In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition
- Four kinds of events – signals, calls, the passing of time, and a change in state.

Figure 1: Events

- Events may be external or internal and asynchronous or synchronous.
- Asynchronous events are events that can happen at arbitrary times eg:- signal, the passing of time, and a change of state. synchronous events, represents the invocation of an operation eg:- Calls
- External events are those that pass between the system and its actors. Internal events are those that pass among the objects that live inside the system.

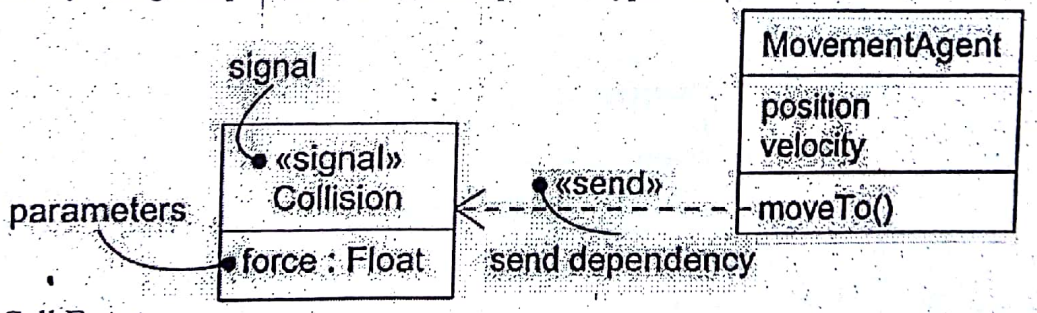
- A signal is an event that represents the specification of an asynchronous stimulus communicated between instances.



Kinds of events

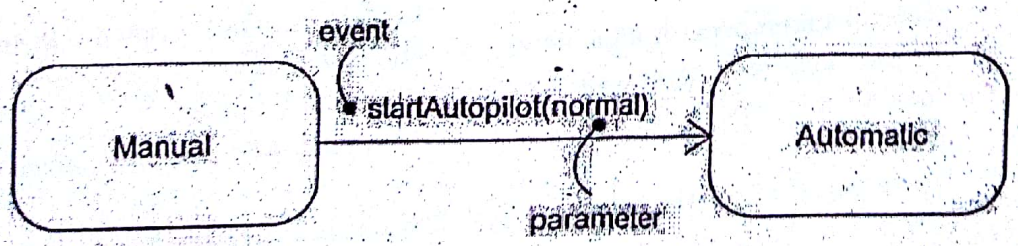
Signal Event

- a signal event represents a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another. Exceptions are an example of internal signal
- a signal event is an asynchronous event
- Signal events may have instances, generalization relationships, attributes and operations. Attributes of a signal serve as its parameters
- A signal event may be sent as the action of a state transition in a state machine or the sending of a message in an interaction
- signals are modeled as stereotyped classes and the relationship between an operation and the events by using a dependency relationship, stereotyped as send



Call Event:

- a call event represents the dispatch of an operation
- a call event is a synchronous event

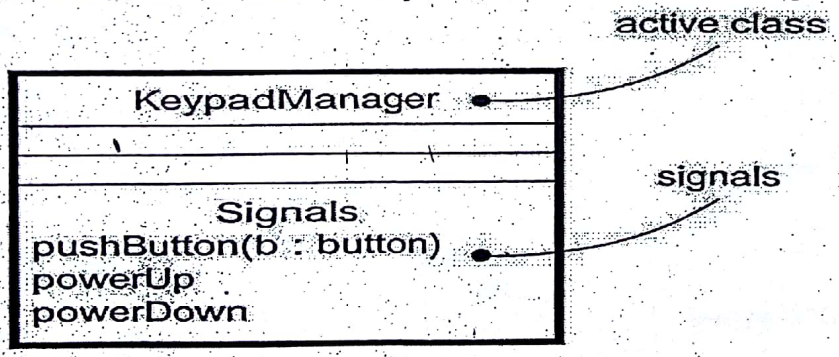


Time and Change Events:

- A time event is an event that represents the passage of time.
- Modeled by using the keyword 'after' followed by some expression that evaluates to a period of time which can be simple or complex.
- A change event is an event that represents a change in state or the satisfaction of some condition.
- Modeled by using the keyword 'when' followed by some Boolean expression.

Sending and Receiving Events

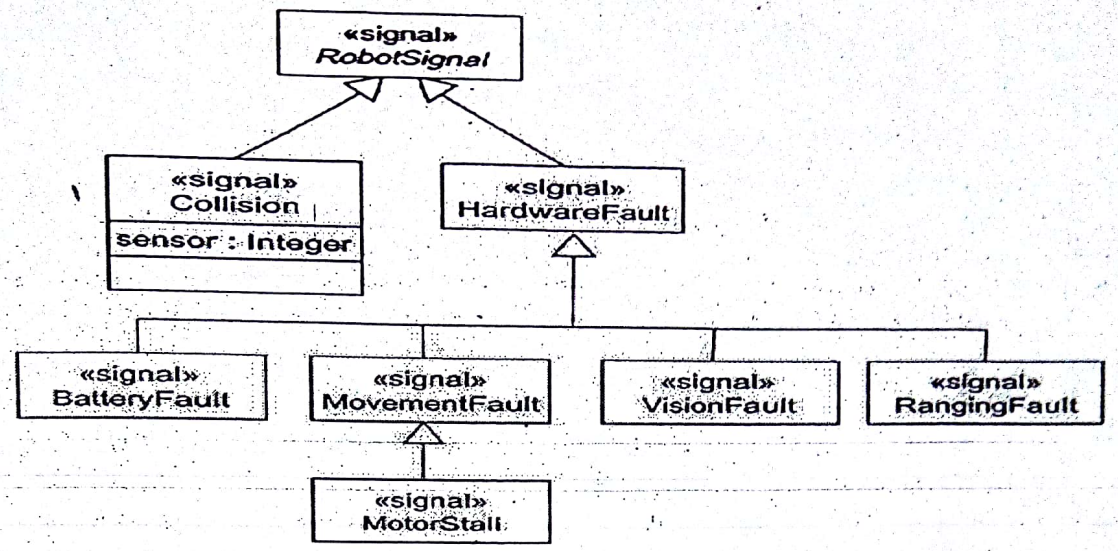
- For synchronous events (Sending or Receiving) like call event, the sender and the receiver are in a rendezvous (the sender dispatches the signal and wait for a response from the receiver) for the duration of the operation.
- when an object calls an operation, the sender dispatches the operation and then waits for the receiver.
- For asynchronous events (Sending or Receiving) like signal event, the sender and receiver do not rendezvous either sender dispatches the signal but does not wait for a response from the receiver.
- When an object sends a signal, the sender dispatches the signal and then continues along its flow of control, not waiting for any return from the receiver.
- call events can be modeled as operations on the class of the object.



Modeling family of signals:
To model a family of signals

- Consider all the different kinds of signals to which a given set of active objects may respond.
- Look for the common kinds of signals and place them in a generalization/specialization hierarchy using inheritance.
- Elevate more general ones and lower more specialized ones.
- Look for the opportunity for polymorphism in the state machines of these active objects.
- Where you find polymorphism, adjust the hierarchy as necessary by introducing intermediate abstract signals.

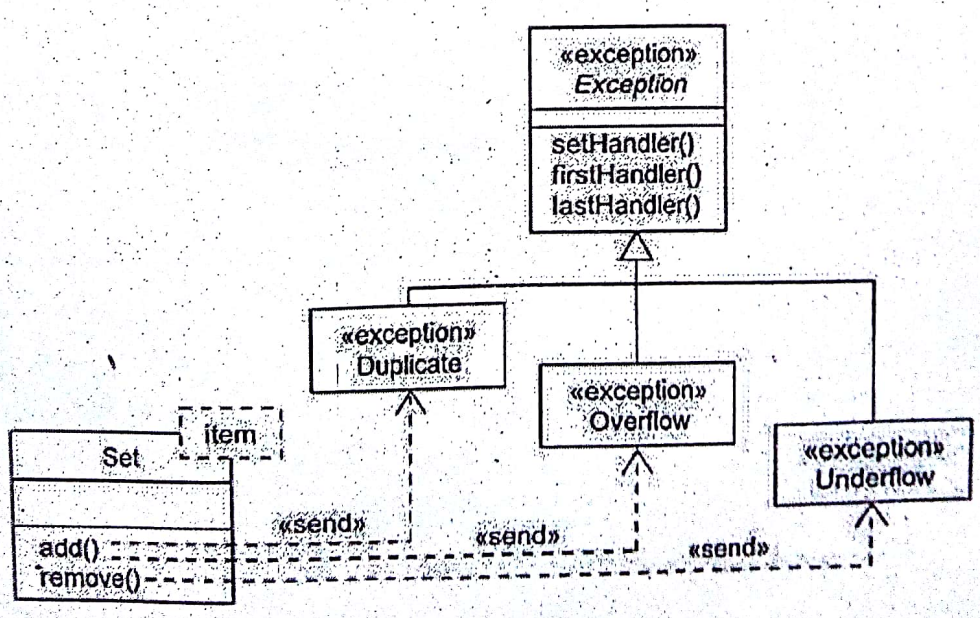
- 6 models a family of signals that may be handled by an autonomous robot.



Modeling Exceptions:

To model exceptions

- For each class and interface, and for each operation of such elements, consider the exceptional conditions that may be raised.
- Arrange these exceptions in a hierarchy. Elevate general ones, lower specialized ones, and introduce intermediate exceptions, as necessary.
- For each operation, specify the exceptions that it may raise.
- You can do so explicitly (by showing send dependencies from an operation to its exceptions) or you can put this in the operation's specification.



UNIT - VI

ADVANCED BEHAVIOURAL MODELING

Architectural Modeling:

Components:

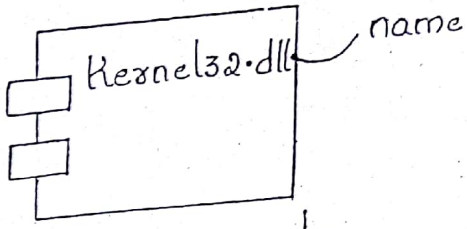


Figure : Components

Terms and Concepts:

- A component is a physical and replacable part of a system that conforms to and provides the realization of a set of interfaces.
- Graphically, a component is rendered as a rectangle with tabs.

Names:

- Every component must have a name that distinguishes it from oth components.

That name alone is known as a simple name.
A pathname is the component name prefixed by the name of the package in which that component lives.

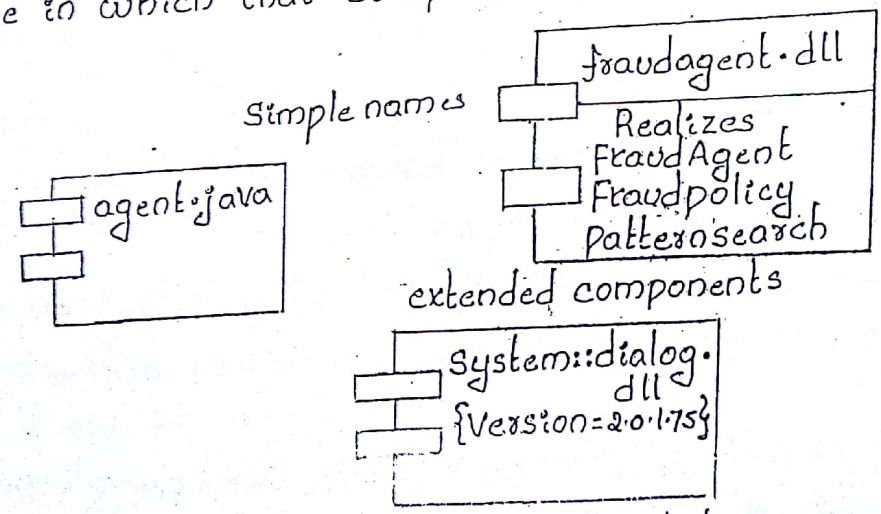


Figure : simple and Extended components

Components and classes:

In many ways, components are like classes : Both have names; both may realize a set of interfaces; both may participate in dependency generalization and association relationships; both may be nested; both may have instances; both may be participants in interactions.

wever, there are some significant differences between compared classes.

Classes	Components
)classes represent logical bstractions.)classes may have attributes nd operations directly.	(1) In short, components may live on nodes, classes may not. (2) In general, components only have operations that are reachable only through their interfaces.

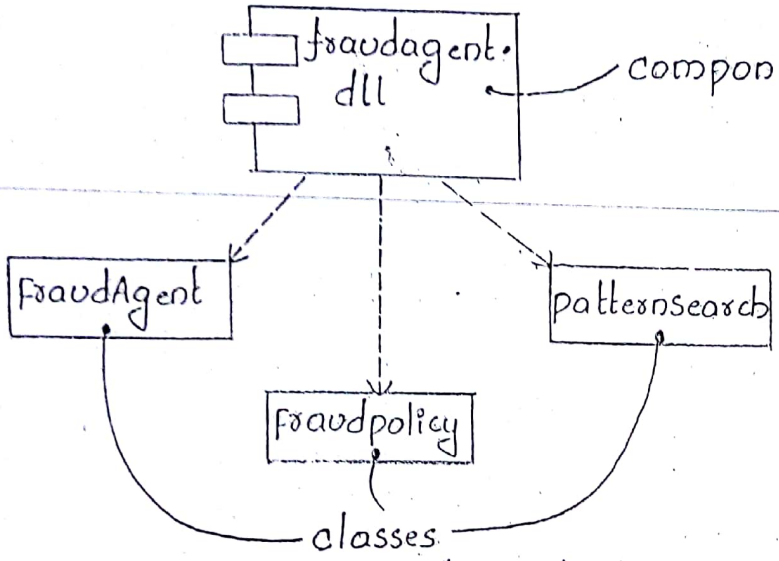


Figure: components and classes

Components and Interfaces:

n interface is a collection of operations that are used to specify a ervice of a class or a component.

he relationship between component and interface is important. he component that realizes the interface is connected to the interfa sing a full realization relationship.

n both cases, the component that accesses the services of the ther component through the interface is connected to the interfa sing a dependency relationship.

n interface that a component realizes is called an export interfac t means ~~an interface~~ an interface that the component provides a service to other components. A component may provide many xport interfaces.

he interface that a component uses is called an import interface.

it means an interface that the component conforms to and so builds on.

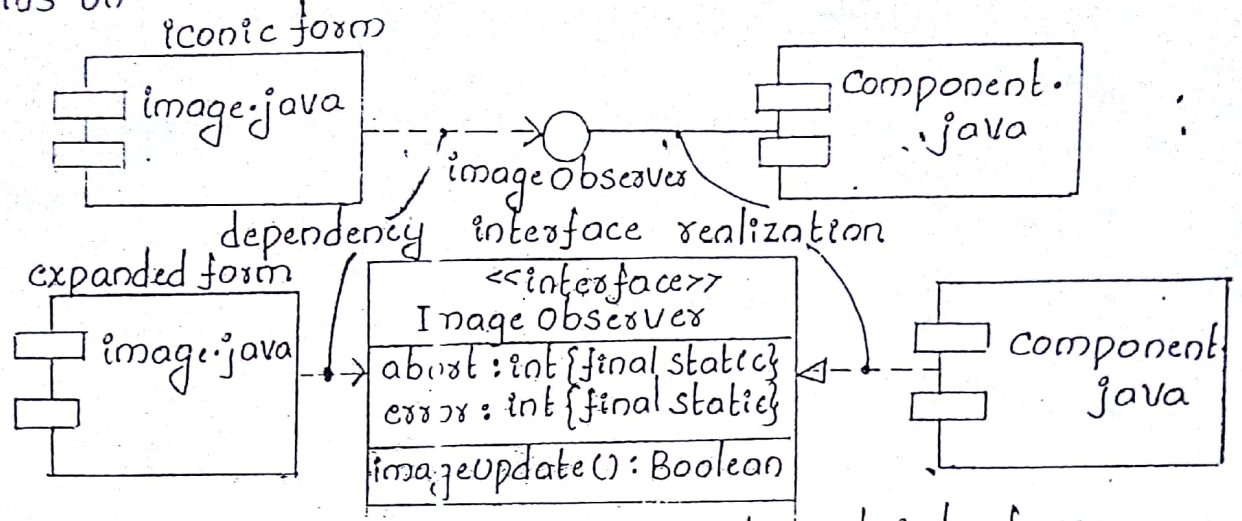


Figure: components and interfaces

Binary Replaceability:

- First, a component is physical. It lives in the world of bits, not concepts.
- Second, a component is replaceable.

Third, a component is part of a system.

Fourth, a component conforms to and provides the realization of a set of interfaces.

Kinds of Components:

Three kinds of components may be distinguised.

First, there are deployment components.

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

Second, there are work product components.

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

Third are execution components.

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

ndard Elements:

e UML defines five standard stereotypes that apply to component

executable → specifies a component that may be executed on a node.

library → specifies a static or dynamic object library.

table → specifies a component that represents a database table.

file → specifies a component that represents a document containing source code or data.

document → specifies a component that represents a document

eployment:

. node is a physical element that exists at run time and represent computational resource, generally having atleast some memory nd often processing capability.

raphically, a node is rendered as a cube.

lames:

very node must have a name that distinguishes it from other nodes. name is a textual string.

at name alone is known as a simple name; a path name is the ode name prefixed by the name of the package in which that node ves.

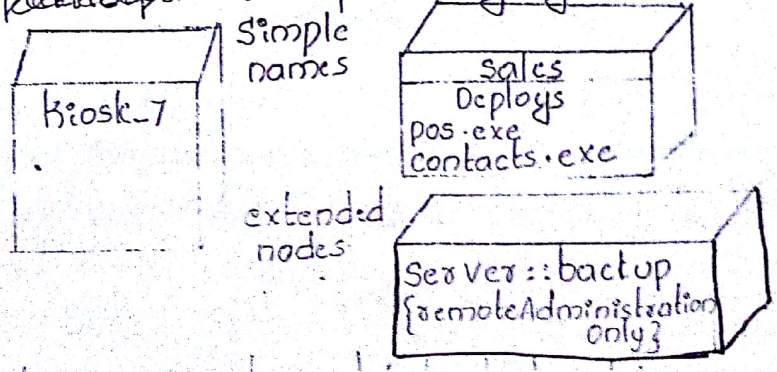
node is typically drawn showing only its name.

odes and Components:

n many ways, nodes are a lot like components:

oth have names.

oth may participate in dependency, generalization and association



odes and components:

In many ways, nodes are a lot like components.
Both have names.
Both may participate in dependency, generalization and associatio relationships; both may be nested; both may have instances; both may be participants in interactions.
However, there are some significant differences between nodes and components.

components	nodes
(1) components are things that participate in the execution of a system.	(1) nodes are things that execute components
(2) components represent the physical packaging of otherwise logical elements.	(2) nodes represent the physical deployment of components.

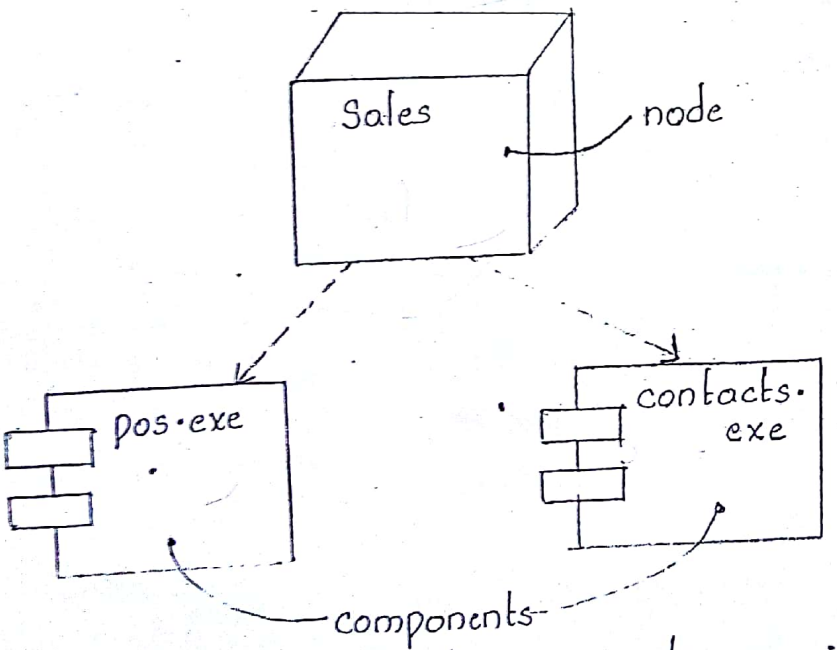
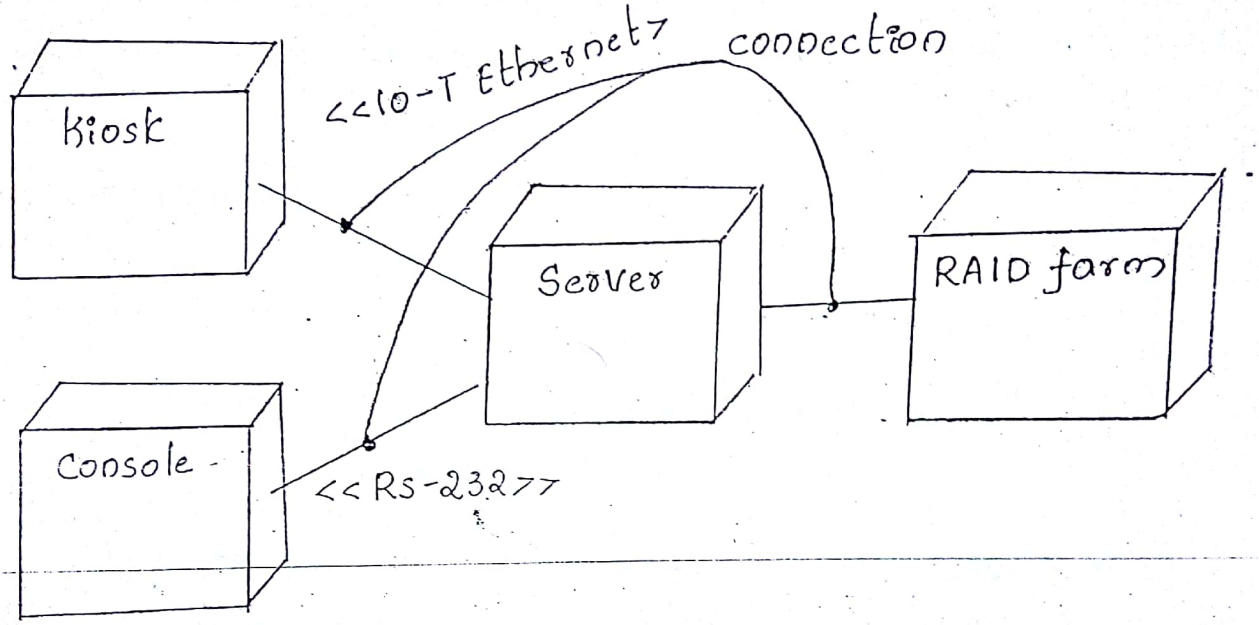


Figure : Nodes and components

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

connections:
The most command kind of relationship you will use among

des is an association.
ou can even use associations to model indirect connections,
ch as a satellite link between distant processors.



Uses:
To model processors and devices.
To model the distributions of components.
To model Embedded systems.

ploym Component Diagrams: (14)

rms and Concepts:

A component diagram shows a set of components and their relations
Graphically, a component diagram is a collection of vertices and arc
A component diagram is just a special kind of diagram and shares th
same common properties as do all other diagrams - a name and
graphical contents that are a projection into a model.

Contents:

Component diagrams commonly contain

- ✓ components
- ✓ Interfaces
- ✓ Dependency, generalization, association and realization relationships.

Like all other diagrams, component diagrams may contain notes an
constraints.

Component diagrams may also contain packages or subsystems, both
of which are used to group elements of your model into larger chunks

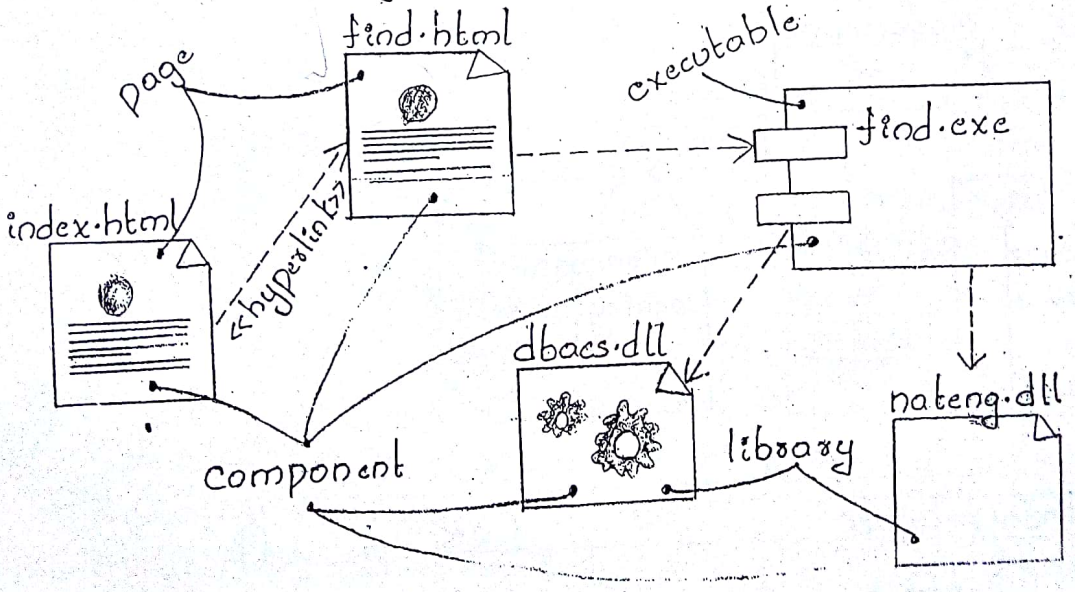


Figure: A component Diagram

Common uses:

- To model source code.
- To model executable releases.
- To model physical databases.
- To model adaptable systems.

loyment Diagrams:

ms and concepts:

leployment diagram is a diagram that shows the configuration of
time processing nodes and the components that live on them.
aphically, a deployment diagram is a collection of vertices and arcs.
deployment diagram is just a special kind of diagram and shares
same common properties as all other diagrams – a name and
phical contents that are a projection into a model.

ntents:

ployment diagrams commonly contain

- ✓ Nodes
- ✓ Dependency and association relationships

ke all other diagrams, deployment diagrams may contain notes
d constraints.

ployment diagrams may also contain components, each of which
ist live on some node.

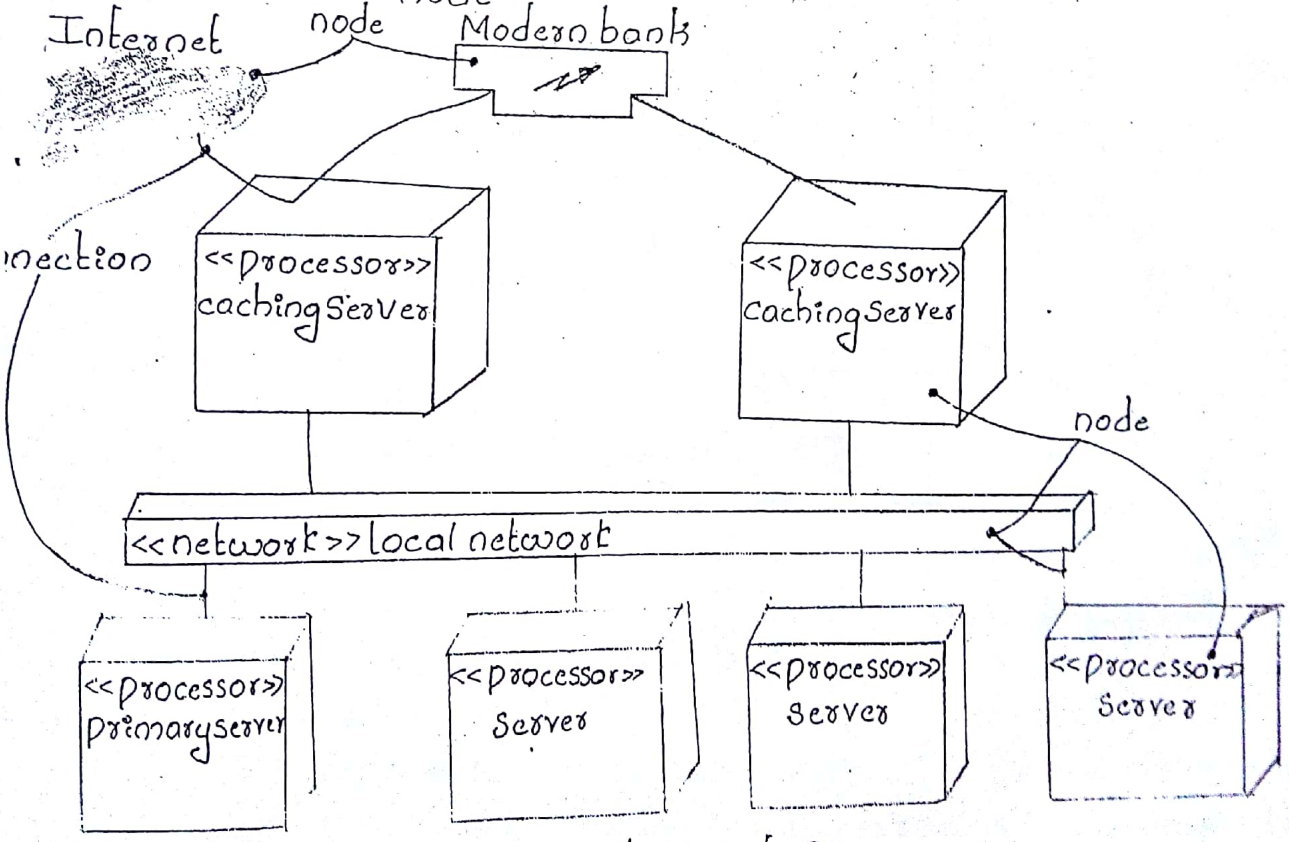


Figure: A Deployment Diagram

Common uses:

To model embedded systems.

To model client/server systems.

[illegible] distributed systems

Common Modeling Techniques for component Diagrams:

Modeling Source code:

To model a systems source code,

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

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

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

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

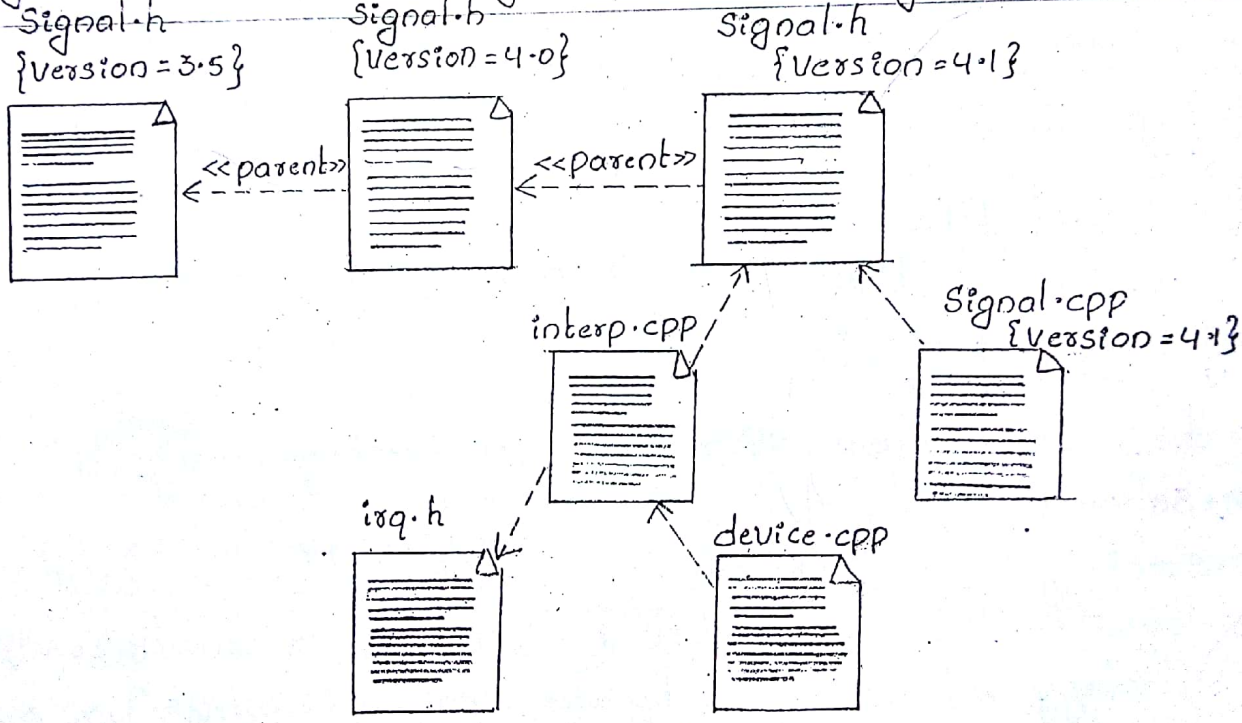


Figure: Modeling source code

Modeling an Executable Release:

To model an executable release,

Identify the set of components you did 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 will find a small number of different kinds of comp

can use the UML's extensibility mechanisms to provide visual
s for these stereotypes.

each component in this set, consider its relationship to its neigh-
s. Most often, this will involve interfaces that are exported (realized
certain components and then imported (used) by others. If you won't
expose the seams in your system, model these interfaces explicitly.
you want your model at a higher level of abstraction, elide these
tionships by showing only dependencies among the components.

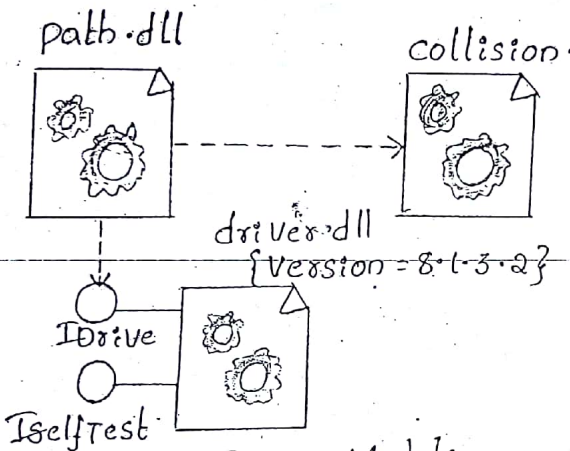


Figure: Modeling an Executable Release

Modeling a physical Database:

dentify the classes in your model that represent your logical
tabase schema.

lect a strategy for mapping these classes to tables. you will also
ant to consider the physical distribution of your databases. your
apping strategy will be affected by the location in which you want
ur data to live on your deployed system.

o visualize, specify, construct and document your mapping, create
component diagram that contains components stereotyped as table
here possible, use tools to help you transform your logical design
to a physical design.

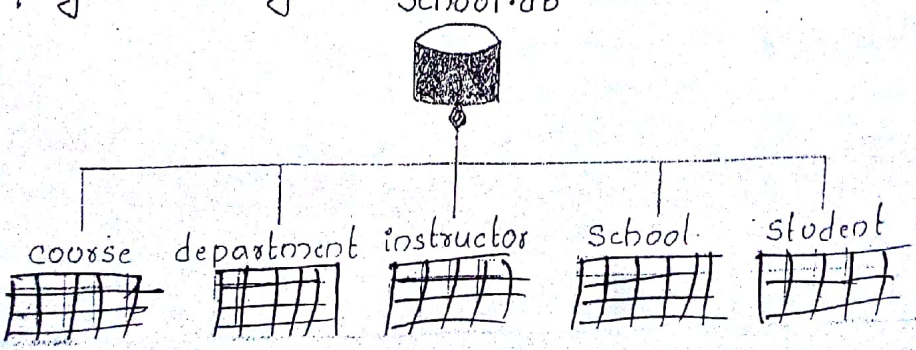


Figure: Modeling a physical Database

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 compone 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 diagra 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.

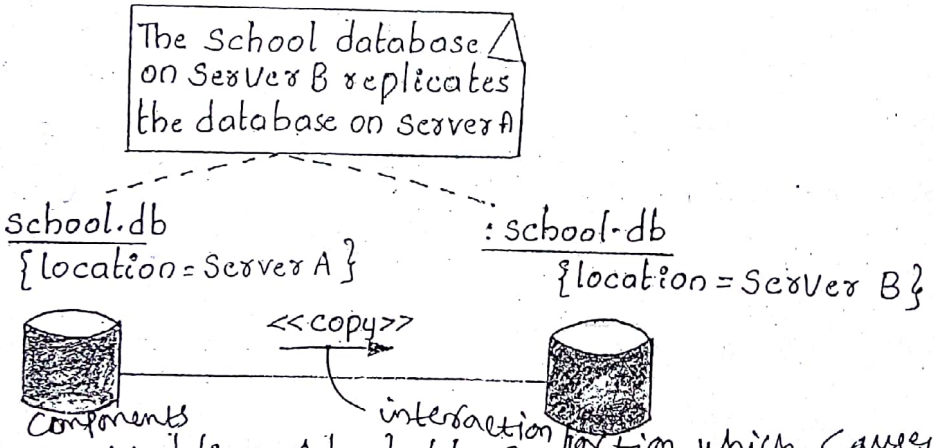


Figure: Modeling Adaptable Systems

Common modeling Techniques for Deployment Diagrams:

Modeling an Embedded system:

To model an embedded system,
Identify the devices and nodes that are unique to your system.
provide visual cues, especially for unusual devices, by using the UML extensibility mechanisms to define system-specific stereotypes with appropriate icons. At the very last you will want to distinguish proces (which contain software components) and devices (which, at that leve 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 implem

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

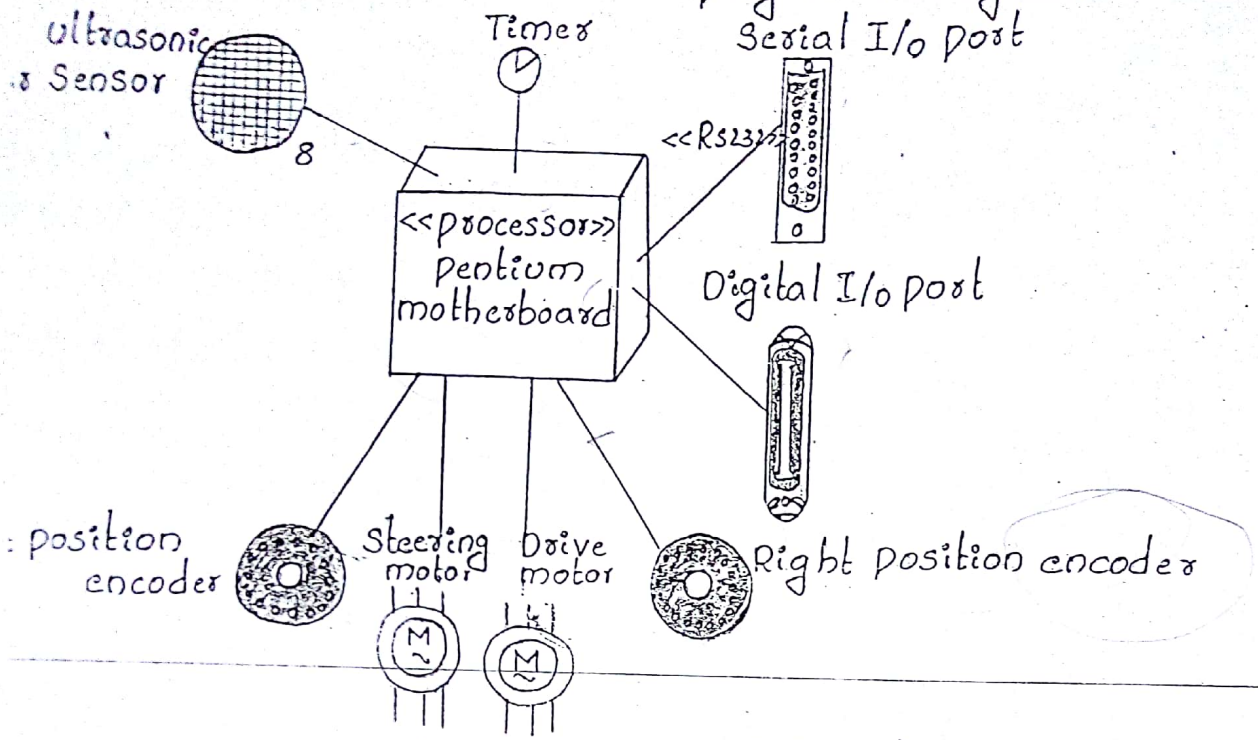


Figure: Modeling an Embedded System

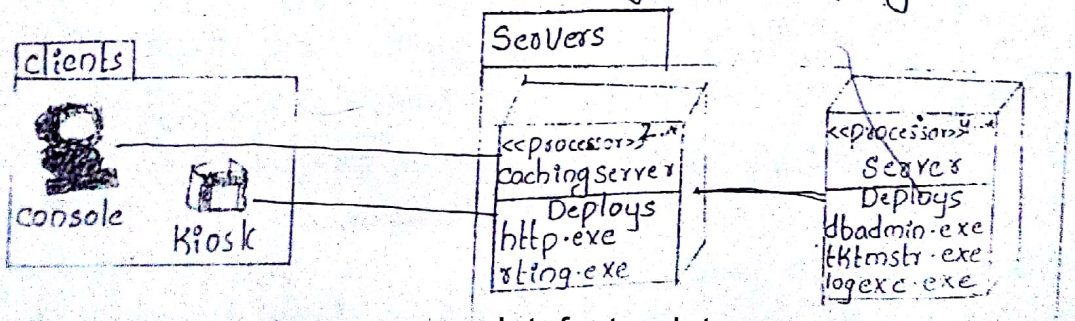
lodeling a client/server system:

o model a client/server system,

dentify the nodes that represent your system's client and server
rocessors.

lighlight those devices that are germane to the behavior of your
stem. for example, you will want to model special devices, such a
edit card readers, badge readers and display devices other than
onitors, because their placement in the systems hardware topolog
ie likely to be architecturally significant.

rovide visual cues for these processors and devices via stereotypi
lodel the topology of these nodes in a deployment diagram. similarly,
ecify the relationship between the components in your systems imple
tation view and the nodes in your systems deployment view.



Modeling a Fully Distributed System:

(7)

To model a fully distributed system,

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

If you need to reason about the performance of the systems 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 usecase diagrams to specify the kinds of behavior you are interested in, and expand on these usecases with interaction diagrams.

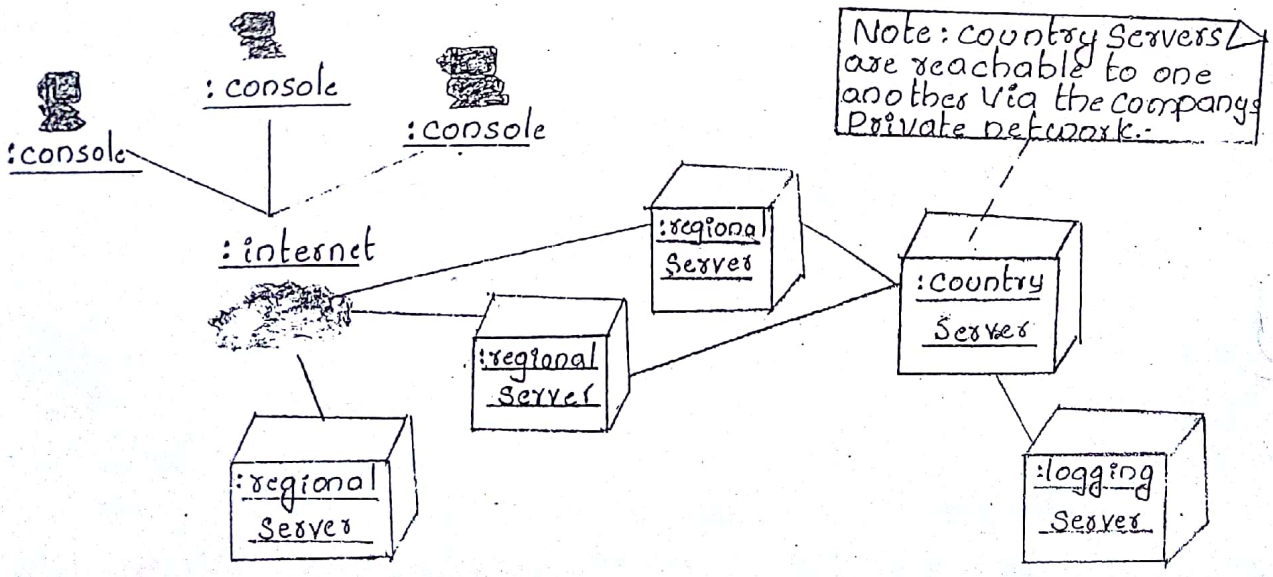


Figure: Modeling a Fully Distributed System