ANNAMACHARYA UNIVERSITY
EXCELLENCE IN EDUCATION; SERVICE TO SOCIETY
(ESTD, UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATION) ACT, 2016)
Rajampet, Annamayya District, A.P – 516126, INDIA

BSC – COMPUTER SCIENCE(Honors), AIML, AI & DA, Programming & Data Science

Lecture Notes
on

Object Oriented Modeling and Design

Written by

D. Siva Sanjeev Kumar
Asst. Professor
Department of Computers

Title of the Course :	**Object Oriented Modeling and Design**
Category:	**PC**
Course Code:	**24BCOM33T**
Branch/es:	**Computer Science (Honors), AIML, AI&DA, Programming & Data Science**
Semester:	**III Semester**

Lecture Hours	Tutorial Hours	Practice Hours	Credits
3	0	0	3

Course Objectives: The course aims to help students understand the role and significance of modeling in software development. It aims to teach the principles of object-oriented modeling and how to apply them using UML. The course aims to cover the various building blocks and architecture of UML to enable effective software system modeling. It aims to develop the ability to design and interpret UML structural diagrams such as class, object, and component diagrams. Additionally, it aims to provide practical knowledge of behavioural modeling through use case, sequence, collaboration, and state chart diagrams. The course also aims to foster understanding of deployment and architectural diagrams and their importance in system implementation. Ultimately, it aims to equip students with the skills to apply UML modeling techniques to real-world software problems, bridging the gap between system requirements and implementation.

Course Outcomes:

At the end of the course, the student will be able to

1. Understand the fundamental concepts of modeling, its importance in software engineering, and the principles of object-oriented modeling.
2. Analyze and construct detailed structural models using advanced class features, relationships, interfaces, roles, and packages.
3. Identify and describe system functionality using use cases and actors.
4. Demonstrate understanding of dynamic system behavior using activity diagrams, state machines, and state chart diagrams.
5. Design and interpret component diagrams to illustrate system components and their interfaces.

Unit 1 Introduction to UML 12

Importance of modeling, principles of modeling, object-oriented modeling, An overview of UML, Building Blocks of UML, Architecture, SDLC, Classes – Terms and Concepts, Common Modeling Techniques, Relationships.

Unit 2 Structural Modeling 8

Advanced classes, advanced relationships, Interfaces, Types and Roles, Packages, Instances**,** **Class Diagrams** – Terms, concepts for Class Diagram.

Unit 3 Basic Behavioural Modeling 8

Use cases – Terms and Concepts, **Use case Diagrams** – Terms and Concepts, **Sequence Diagra** – Terms and Concepts, Collaboration Diagrams – Terms and Concepts.

Unit 4 Advanced Behavioral Modeling 10

Activity Diagrams – Terms and Concepts, Events and Signals, State Machines, Processes and Threads, Time and Space, State Chart Diagrams - Terms and Concepts.

Unit 5 Architectural Modeling 10

Component Diagrams – Terms and Concepts, Deployment Diagrams – Terms and Concepts.

Prescribed Textbooks:

1. Grady Booch, James Ram Baugh and Ivar Jacobson, The Unified Modeling Language User Guide. Pearson Education, 1999.

Reference Books:

1. John W. Satzinger, Robert B. Jackson and Stephen D. Burd, Object-Oriented Analysis and Design with the Unified Process. Cengage Learning, 2004.
2. Hans-Erik Eriksson, Magnus Penker, Brian Lyons and David Fado, UML 2: Toolkit. Wiley India Pvt. Ltd., 2004.

UNIT – I

Unit 1 Introduction to UML

Importance of modeling, principles of modeling, object-oriented modeling, An overview of UML, Building Blocks of UML, Architecture, SDLC, Classes – Terms and Concepts, Common Modeling Techniques, Relationships.

The UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML gives you a standard way to write a system's blueprints, covering conceptual things, such as business processes and system functions, as well as concrete things, such as classes written in a specific programming language, database schemas, and reusable software components.

Model

A model is a simplification of reality. A model provides the blueprints of a system. A model may be structural, emphasizing the organization of the system, or it may be behavioral, emphasizing the dynamics of the system.

Why do we model

We build models so that we can better understand the system we are developing.

Through modeling, we achieve four aims.

1. Models help us to visualize a system as it is or as we want it to be.
2. Models permit us to specify the structure or behavior of a system.
3. Models give us a template that guides us in constructing a system.
4. Models document the decisions we have made.

We build models of complex systems because we cannot comprehend such a system in its entirety.

Principles of Modeling :-

There are four basic principles of model

1. The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.
2. Every model may be expressed at different levels of precision.
3. The best models are connected to reality.
4. No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

Object Oriented Modeling

In software, there are several ways to approach a model. The two most common ways are

1. Algorithmic perspective
2. Object-oriented perspective

Algorithmic Perspective

The traditional view of software development takes an algorithmic perspective.

In this approach, the main building block of all software is the procedure or function.

This view leads developers to focus on issues of control and the decomposition of larger algorithms into smaller ones.

As requirements change and the system grows, systems built with an algorithmic focus turn out to be very hard to maintain.

Object-oriented perspective

The contemporary view of software development takes an object-oriented perspective.

In this approach, the main building block of all software systems is the object or class.

A class is a description of a set of common objects.

Every object has identity, state, and behavior.

Object-oriented development provides the conceptual foundation for assembling systems out of components using technology such as Java Beans or COM+.

An Overview of UML

- The Unified Modeling Language is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system.
- The UML is appropriate for modeling systems ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded systems. It is a very expressive language, addressing all the views needed to develop and then deploy such systems.

The UML is a language for

- Visualizing
- Specifying
- Constructing
- Documenting

- **Visualizing** The UML is more than just a bunch of graphical symbols. Rather, behind each symbol in the UML notation is a well-defined semantics. In this manner, one developer can write a model in the UML, and another developer, or even another tool, can interpret that model unambiguously
- **Specifying** means building models that are precise, unambiguous, and complete.
- **Constructing** the UML is not a visual programming language, but its models can be directly connected to a variety of programming languages
- **Documenting** a healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include
  - Requirements
  - Architecture
  - Design
  - Source code
  - Project plans
  - Tests
  - Prototypes
  - Releases

To understand the UML, you need to form a **conceptual model of the language**, and this requires learning three major elements:

1. Things
2. Relationships
3. Diagrams

Things in the UML

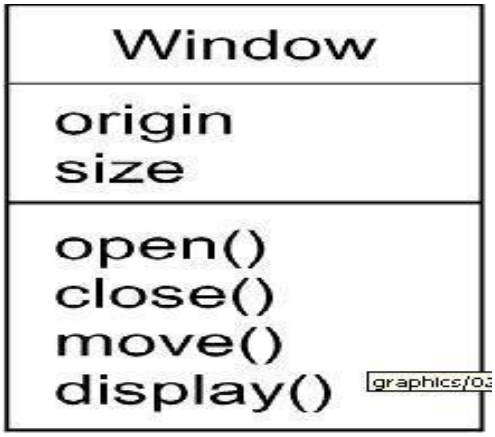
There are four kinds of things in the UML:

1.Structural things 2.Behavioral things 3.Grouping things 4.Annotational things

Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things.

1. Classes
2. Interfaces
3. Collaborations
4. Use cases
5. Active classes
6. Components
7. Nodes

Class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations.



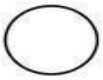
Interface

Interface is a collection of operations that specify a service of a class or component.

An interface therefore describes the externally visible behavior of that element.

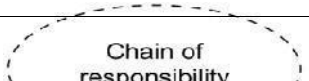
An interface might represent the complete behavior of a class or component or only a part of that behavior.

An interface is rendered as a circle together with its name. An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface



Collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Therefore, collaborations have structural, as well as behavioral, dimensions. A given class might participate in several collaborations.

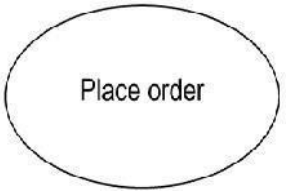
Graphically, a collaboration is rendered as an ellipse with dashed lines,



usually including only its name

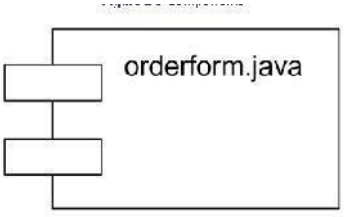
Usecase

- Use case is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor
- Use case is used to structure the behavioral things in a model.
- A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name



Active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations

Component is a physical and replaceable 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



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, usually including only its name

Behavioral Things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are two primary kinds of behavioral things

Interaction
state
machine

Interaction

Interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose

An interaction involves a number of other elements, including messages, action sequences and links

Graphically a message is rendered as a directed line, almost always including the name of its operation



State Machine

State machine is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events

State machine involves a number of other elements, including states, transitions, events and activities

Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates

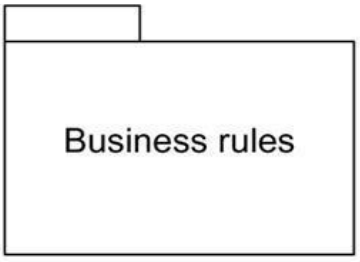


Grouping Things:-

1. are the organizational parts of UML models. These are the boxes into which a model can be decomposed
2. There is one primary kind of grouping thing, namely, packages.

Package:-

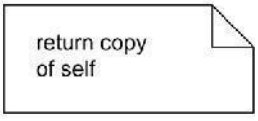
- A package is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package
- Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents



Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe about any element in a model.

A note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.

Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment



Relationships in the UML: There are four kinds of relationships in the UML:

1. Dependency
2. Association
3. Generalization
4. Realization

Dependency:- Dependency is a semantic relationship between two things in which a change to one thing may affect the semantics of the other thing.

Class diagram

A class diagram shows a set of classes, interfaces, and collaborations and their relationships.

Class diagrams that include active classes address the static process view of a system.

Object diagram

- Object diagrams represent static snapshots of instances of the things found in class diagrams
- These diagrams address the static design view or static process view of a system
- An object diagram shows a set of objects and their relationships

Use case diagram

- A use case diagram shows a set of use cases and actors and their relationships
- Use case diagrams address the static use case view of a system.
- These diagrams are especially important in organizing and modeling the behaviors of a system.

Interaction Diagrams

Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams

Interaction diagrams address the dynamic view of a system

A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages

A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages

Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other

Statechart diagram

- A statechart diagram shows a state machine, consisting of states, transitions, events, and activities
- Statechart diagrams address the dynamic view of a system
- They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object

Activity diagram

An activity diagram is a special kind of a statechart diagram that shows the flow from activity to activity within a system

Activity diagrams address the dynamic view of a system

They are especially important in modeling the function of a system and emphasize the flow of control among objects

Component diagram

- A component diagram shows the organizations and dependencies among a set of components.
- Component diagrams address the static implementation view of a system.

- They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations

Deployment diagram

- A deployment diagram shows the configuration of run-time processing nodes and the components that live on them
- Deployment diagrams address the static deployment view of an architecture

Rules of the UML

The UML has semantic rules for

1. Names What you can call things, relationships, and diagrams
2. Scope The context that gives specific meaning to a name
3. Visibility How those names can be seen and used by others
4. Integrity How things properly and consistently relate to one another
5. Execution What it means to run or simulate a dynamic model

Models built during the development of a software-intensive system tend to evolve and may be viewed by many stakeholders in different ways and at different times. For this reason, it is common for the development team to not only build models that are well-formed, but also to build models that are

1. Elided Certain elements are hidden to simplify the view
2. Incomplete Certain elements may be missing
3. Inconsistent The integrity of the model is not guaranteed

Common Mechanisms in the UML

UML is made simpler by the presence of four common mechanisms that apply consistently throughout the language.

1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

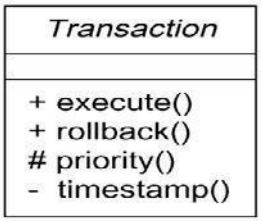
Specification that provides a textual statement of the syntax and semantics of that building block. The UML's specifications provide a semantic backplane that contains all the parts of all the models of a system, each part related to one another in a consistent fashion

Adornments Most elements in the UML have a unique and direct graphicalnotation that provides a visual representation of the most important aspects of the element. A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations. Many of these details can be rendered as graphical or textual adornments to the class's basic rectangular notation.

Extensibility Mechanisms

The UML's extensibility mechanisms include

1. Stereotypes
2. Tagged values
3. Constraints



Stereotype

- Stereotype extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem
- A tagged value extends the properties of a UML building block, allowing you to create new information in that element's specification
- A constraint extends the semantics of a UML building block, allowing you to add new rules or modify existing ones

Architecture

A system's architecture is perhaps the most important artifact that can be used to manage these different viewpoints and so control the iterative and incremental development of a system throughout its life cycle.

Architecture is the set of significant decisions about

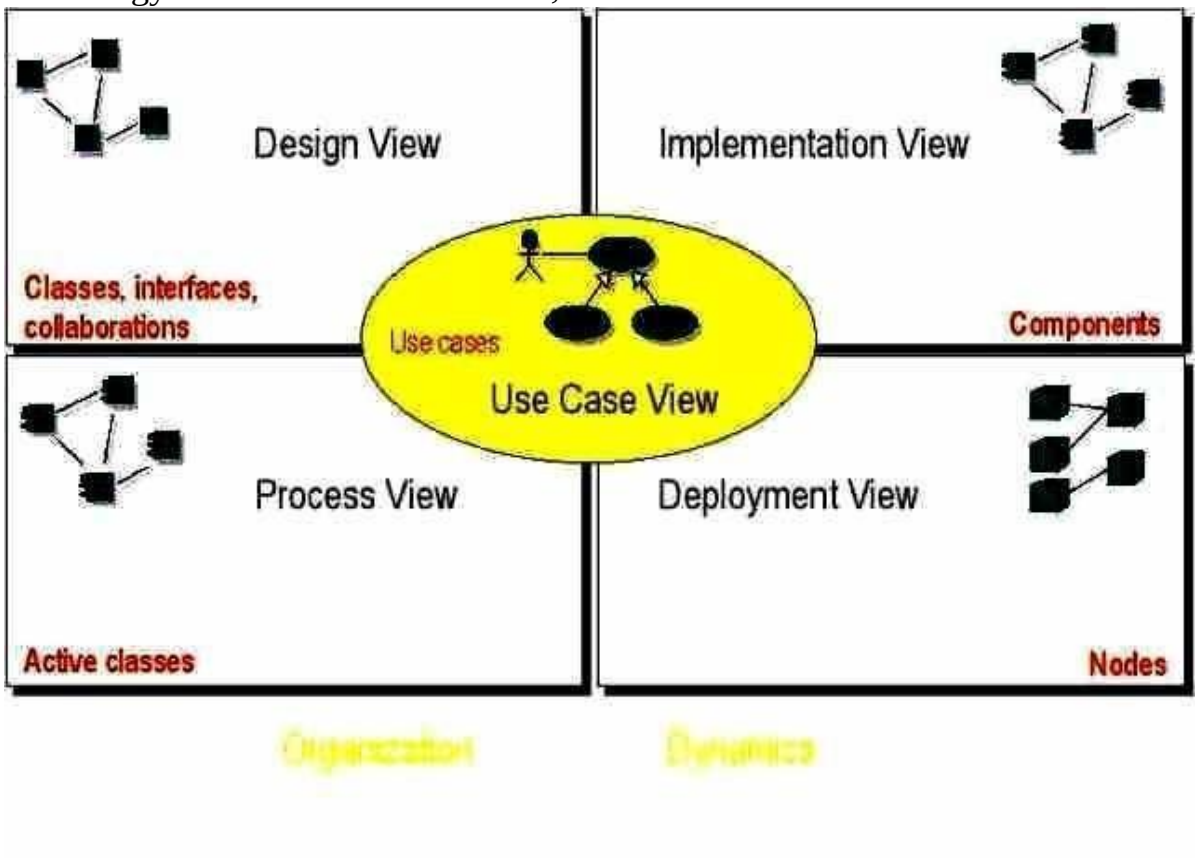
The organization of a software system

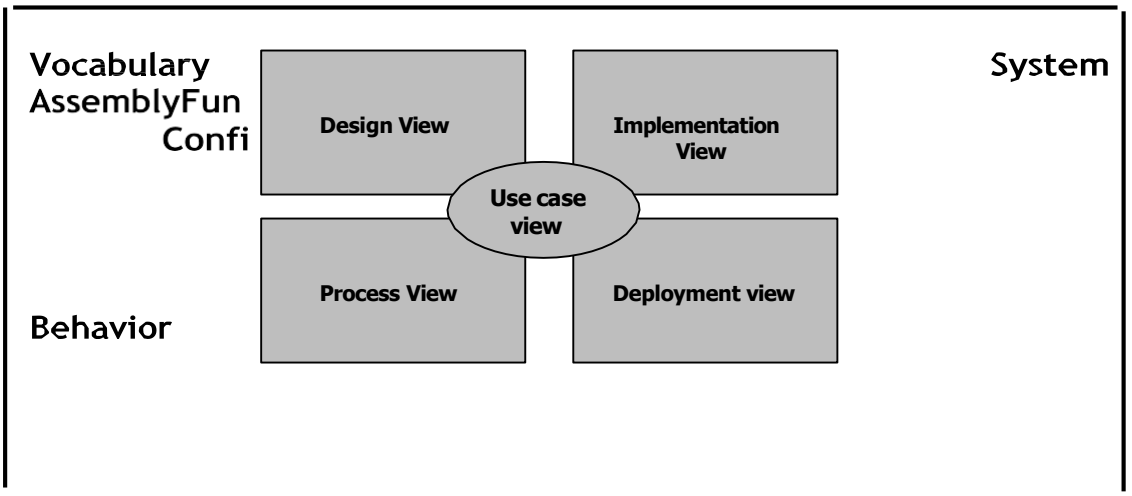
The selection of the structural elements and their interfaces by which the system is composed

Their behavior, as specified in the collaborations among those elements The composition of these structural and behavioral elements into progressively larger subsystems

The architectural style that guides this organization: the static and dynamic elements and their interfaces, their collaborations, and their composition.

Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns.





Modeling a System's Architecture

Use case view

The use case view of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers.

With the UML, the static aspects of this view are captured in use case diagrams

The dynamic aspects of this view are captured in interaction diagrams, state chart diagrams, and activity diagrams.

Design View

- The design view of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution.
- This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users.

Process View

- The process view of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms.
- This view primarily addresses the performance, scalability, and throughput of the system

Implementation View

The implementation view of a system encompasses the components and files that are used to assemble and release the physical system.

This view primarily addresses the configuration management of the system's releases, made up of somewhat independent components and files that can be assembled in various ways to produce a running system.

Deployment Diagram

The deployment view of a system encompasses the nodes that form the system's hardware topology on which the system executes.

This view primarily addresses the distribution, delivery, and installation of the

parts that make up the physical system.

Software Development Life Cycle(SDLC):-

The UML is largely process-independent, meaning that it is not tied to any particular software development life cycle. However, to get the most benefit from the UML, it should consider a process that is

- Use case driven
- Architecture-centric
- Iterative and incremental

Use case driven means that use cases are used as a primary artifact for establishing the desired behavior of the system, for verifying and validating the system's architecture, for testing, and for communicating among the stakeholders of the project.

Architecture-centric means that a system's architecture is used as a primary artifact for conceptualizing, constructing, managing, and evolving the system under development.

An ***iterative process*** is one that involves managing a stream of executable releases. An is one that involves the continuous integration of the system's architecture to produce these releases, with each new release embodying incremental improvements over the other. Together, an iterative and incremental process is *risk-driven,* meaning that each new release is focused on attacking and reducing the most significant risks to the success of the project.

This use case driven, architecture-centric, and iterative/incremental process can be broken into phases. A *phase* is the span of time between two major milestones of the process, when a welldefined set of objectives are met, artifacts are completed, and decisions are made whether to move into the next phase.

There are four phases in the software development life cycle:

- Inception,
- Elaboration,
- Construction
- Transition.

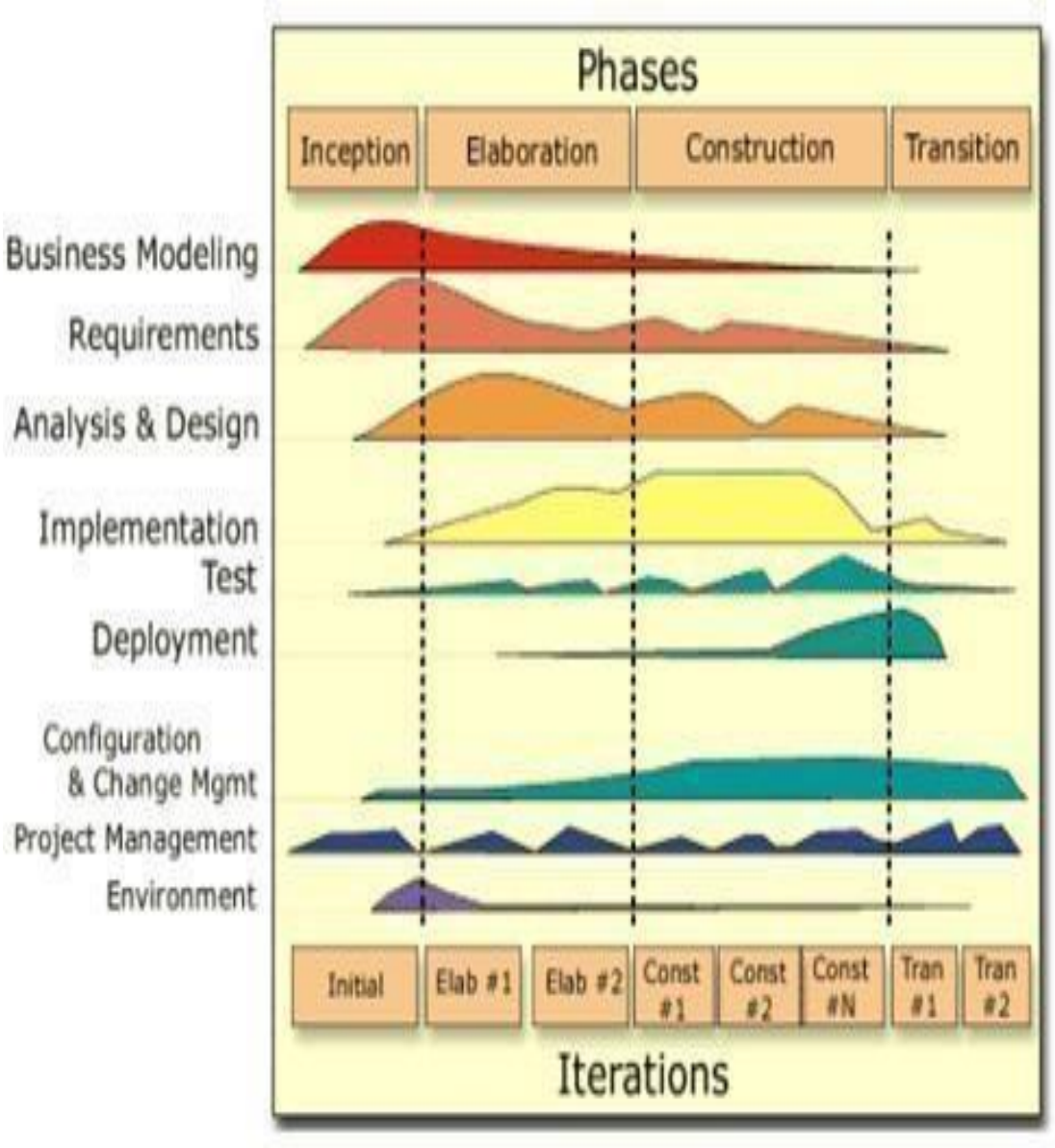
Inception is the first phase of the process, when the seed idea for the development is brought up to the point of being at least internally sufficiently well-founded to warrant entering into the elaboration phase.

Elaboration is the second phase of the process, when the product vision and its

architecture are defined. In this phase, the system's requirements are articulated, prioritized, and baselined. A system's requirements may range from general vision statements to precise evaluation criteria, each specifying particular functional or nonfunctional behavior and each providing a basis for testing.

Construction is the third phase of the process, when the software is brought from an executable architectural baseline to being ready to be transitioned to the user community. Here also, the system's requirements and especially its evaluation criteria are constantly reexamined against the business needs of the project, and resources are allocated as appropriate to actively attack risks to the project.

Transition is the fourth phase of the process, when the software is turned into the hands of the user community. Rarely does the software development process end here, for even during this phase, the system is continuously improved, bugs are eradicated, and features that didn't make an earlier release are added.



Unit II
Advanced Structural Modeling

Unit 2 Structural Modeling

Advanced classes, advanced relationships, Interfaces, Types and Roles, Packages, Instances**,**

Class Diagrams – Terms, concepts for Class Diagram

- A relationship is a connection among things. In object-oriented modeling, the four most important relationships are dependencies, generalizations, associations, and realizations.
- Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the different relationships.

Dependency :-

- A dependency is a using relationship, specifying that a change in the specification of one thing may affect another thing that uses it, but not necessarily the reverse. Graphically, a dependency is rendered as a dashed line
- A plain, unadorned dependency relationship is sufficient for most of the using relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines a number of stereotypes that may be applied to dependency relationships.
- There are 17 such stereotypes, all of which can be organized into six groups.

- First, there are eight stereotypes that apply to dependency relationships among classes and objects in class diagrams.

1	**bind**	Specifies that the source instantiates the target template using the given actual parameters
2	**derive**	Specifies that the source may be computed from the target
3	**friend**	Specifies that the source is given special visibility into the target
4	**instanceOf**	Specifies that the source object is an instance of the target classifier
5	**instantiate**	Specifies that the source creates instances of the target
6	**powertype**	Specifies that the target is a powertype of the source; a powertype is a classifier whose objects are all the children of a given parent
7	**refine**	Specifies that the source is at a finer degree of abstraction than the target
8	**use**	Specifies that the semantics of the source element depends on the semantics of the public part of the target

bind:

bind includes a list of actual arguments that map to the formal arguments of the template.

derive

When you want to model the relationship between two attributes or two associations, one of which is concrete and the other is conceptual**.**

friend

When you want to model relationships such as found with C++ friend classes.

instanceOf

When you want to model the relationship between a class and an object in the same diagram, or between a class and its meta class.

instantiate

when you want to specify which element creates objects of another.

Power type

when you want to model classes that cover other classes, such as you'll find when modeling databases

refine

when you want to model classes that are essentially the same but at different levels of abstraction.

use

when you want to explicitly mark a dependency as a using relationship

* There are two stereotypes that apply to dependency relationships among packages.

9	**access**	Specifies that the source package is granted the right to reference the elements of the target package
10	**import**	A kind of access that specifies that the public contents of the target package enter the flat namespace of the source, as if they had been declared in the source

* Two stereotypes apply to dependency relationships among use cases:

11	**extend**	Specifies that the target use case extends the behavior of the source
12	**include**	Specifies that the source use case explicitly incorporates the behavior of another use case at a location specified by the source

* There are three stereotypes when modeling interactions among objects.

13	**become**	Specifies that the target is the same object as the source but at a later point in time and with possibly different values, state, or roles
14	**call**	Specifies that the source operation invokes the target operation
15	**copy**	Specifies that the target object is an exact, but independent, copy of the source

* We'll use become and copy when you want to show the role, state, or attribute value of one object at

different points in time or space

* You'll use call when you want to model the calling dependencies among operations.

* One stereotype you'll encounter in the context of state machines is

16	**send**	Specifies that the source operation sends the target event

* We'll use send when you want to model an operation dispatching a given event to a target object.

* The send dependency in effect lets you tie independent state machines together.

Finally, one stereotype that you'll encounter in the context of organizing the elements of your system into subsystems and models is

17	**trace**	Specifies that the target is an historical ancestor of the source

* We'll use trace when you want to model the relationships among elements in different models

Generalization

A generalization is a relationship between a general thing (called the superclass or parent) and a more specific kind of that thing(called the subclass or child).

In a generalization relationship, instances of the child may be used anywhere instances of the parent apply—meaning that the child is substitutable for the parent.

A plain, unadorned generalization relationship is sufficient for most of the inheritance relationships you'll encounter. However, if you want to specify a shade of meaning,

The UML defines one stereotype and four constraints that may be applied to generalization relationships.

1	**implementation**	Specifies that the child inherits the implementation of the parent but does not make public nor support its interfaces, thereby violating substitutability

implementation

- We'll use implementation when you want to model private inheritance, such as found in C++.

Next, there are four standard constraints that apply to generalization relationships

1	**complete**	Specifies that all children in the generalization have been specified in the model and that no additional children are permitted
2	**incomplete**	Specifies that not all children in the generalization have been specified (even if some are elided) and that additional children are permitted
3	**disjoint**	Specifies that objects of the parent may have no more than one of the children as a type
4	**overlapping**	Specifies that objects of the parent may have more than one of the children as a type

complete

- We'll use the complete constraint when you want to show explicitly that you've fully specified a hierarchy in the model (although no one diagram may show that hierarchy);

incomplete

- We'll use incomplete to show explicitly that you have not stated the full specification of the hierarchy in the model (although one diagram may show

everything in the model).

Disjoint & overlapping

- These two constraints apply only in the context of multiple inheritance.

- We'll use disjoint and overlapping when you want to distinguish between static classification (disjoint) and dynamic classification (overlapping).

Association

An association is a structural relationship, specifying that objects of one thing are connected to objects of another. We use associations when you want to show structural relationships.

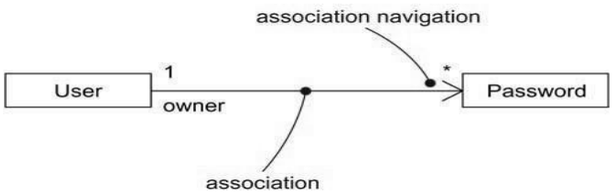
There are four basic adornments that apply to an association: a name, the role at each end of the association, the multiplicity at each end of the association, and aggregation.

For advanced uses, there are a number of other properties you can use to model subtle details, such as Navigation Vision Qualification various flavors of aggregation.

Navigation

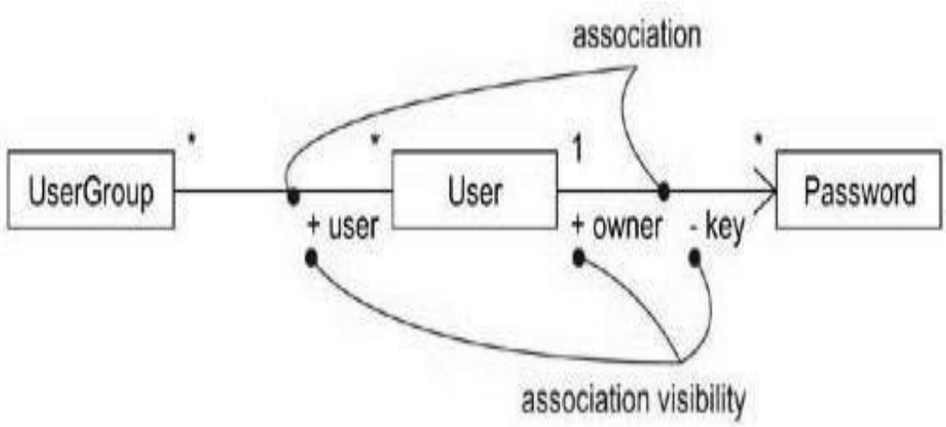
- unadorned association between two classes, such as Book and Library, it's possible to navigate from objects of one kind to objects of the other kind. Unless otherwise specified, navigation across an association is bidirectional.
- However, there are some circumstances in which you'll want to limit navigation to just

one direction.



Navigation Visibility

- Given an association between two classes, objects of one class can see and navigate to objects of the other, unless otherwise restricted by an explicit statement of navigation.
- However, there are circumstances in which you'll want to limit the visibility across that association relative to objects outside the association.
- In the UML, you can specify three levels of visibility for an association end, just as you can for a class's features by appending a visibility symbol to a role name the visibility of a role is public.
- rivate visibility indicates that objects at that end are not accessible to any objects outside the association.
- Protected visibility indicates that objects at that end are not accessible to any objects outside the association, except for children of the other end.

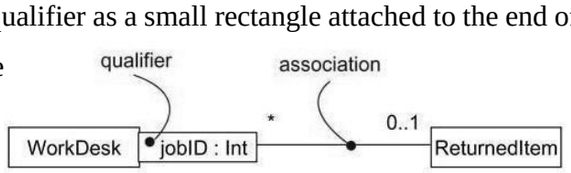


Visibility Qualification

In the context of an association, one of the most common modeling idioms you'll encounter is the problem of lookup. Given an object at one end of an association, how do you identify an object or set of objects at the other end?

In the UML, you'd model this idiom using a qualifier, which is an association attribute whose values partition the set of objects related to an object across an association.

You render a qualifier as a small rectangle attached to the end of an association, placing the attributes in the rectangle



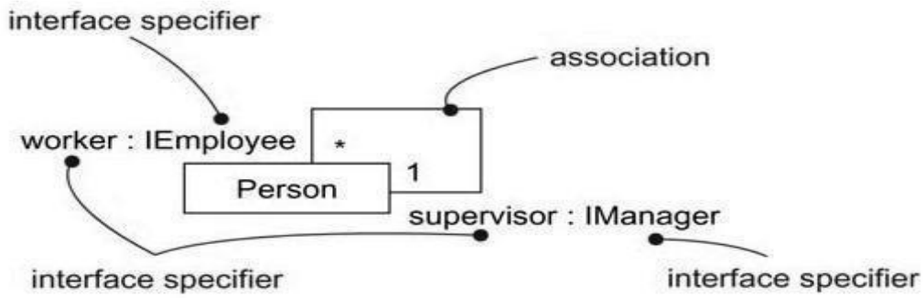
Qualification Interface Specifier

An interface is a collection of operations that are used to specify a service of a class or

a component

Collectively, the interfaces realized by a class represent a complete specification of the behavior of that class.

However, in the context of an association with another target class, a source class may choose to present only part of its face to the world



a Person class may realize many interfaces: IManager, IEmployee, IOfficer, and so on

you can model the relationship between a supervisor and her workers with a one-to- many association, explicitly labeling the roles of this association as supervisor and worker

- In the context of this association, a Person in the role of supervisor presents only the IManager face to the worker; a Person in the role of worker presents only the IEmployee face to the supervisor. As the figure shows, you can explicitly show the type of role using the syntax rolename : iname, where iname is some interface of the other classifier.

Composition

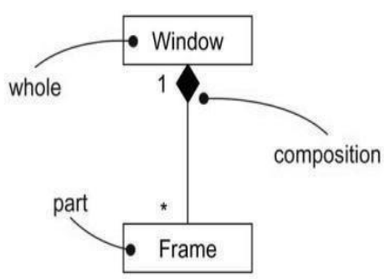
* Simple aggregation is entirely conceptual and does nothing more than distinguish a "whole" from a "part."

* Composition is a form of aggregation, with strong ownership and coincident lifetime as part of the whole.

* Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it. Such parts can also be explicitly removed before the death of the composite.

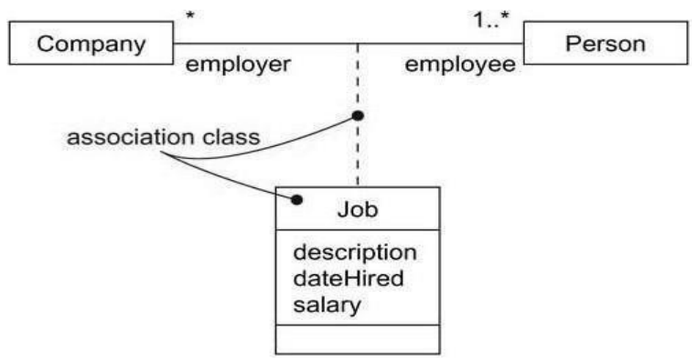
* This means that, in a composite aggregation, an object may be a part of only one composite at a time.

In addition, in a composite aggregation, the whole is responsible for the disposition of its parts, which means that the composite must manage the creation and destruction of its parts



Composition Association Classes

* In an association between two classes, the association itself might have properties.
  * An association class can be seen as an association that also has class properties, or as a class that also has
  association properties.
  * We render an association class as a class symbol attached by a dashed line to an association



Association Classes

Constraints

* UML defines five constraints that may be applied to association relationships.

1	**implicit**	Specifies that the relationship is not manifest but, rather, is only conceptual
2	**ordered**	Specifies that the set of objects at one end of an association are in an explicit order
3	**changeable**	Links between objects may be added, removed, and changed freely
4	**addOnly**	New links may be added from an object on the opposite end of the association
5	**frozen**	A link, once added from an object on the opposite end of the association, may not be modified or deleted

implicit

* if you have an association between two base classes, you can specify that same association between two children of those base classes

* you can specify that the objects at one end of an association (with a multiplicity greater than one) are ordered or unordered.

ordered

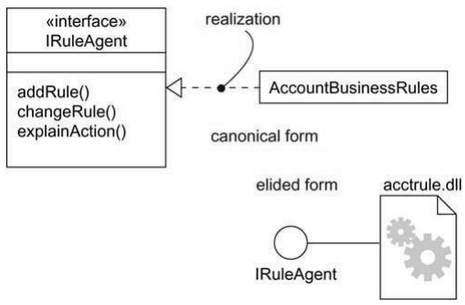
* For example, in a User/Password association, the Passwords associated with the User might be kept in a
least-recently used order, and would be marked as ordered.

Finally, there is one constraint for managing related sets of associations:

1	**xor**	Specifies that, over a set of associations, exactly one is manfest for each
		associated object

Realization

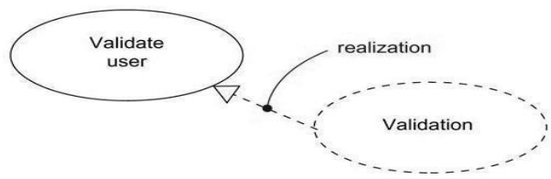
1. Realization is sufficiently different from dependency, generalization, and association relationships that it is treated as a separate kind of relationship.
2. A realizationis a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.
3. Graphically, a realization is rendered as a dashed directed line with a large open arrowhead pointing to the classifier that specifies the contract.
4. You'll use realization in two circumstances: in the context of interfaces and in the context of collaborations
5. Most of the time, you'll use realization to specify the relationship between an interface



and the class or component that provides an operation or service for it

6. You'll also use realization to specify the relationship between a use case and the collaboration that realizes that use case

Realization of an Interface



Realization of a Use Case

Common Modeling Techniques

Modeling Webs of Relationships

1. When you model the vocabulary of a complex system, you may encounter dozens, if not hundreds or thousands, of classes, interfaces, components, nodes, and use cases.
2. Establishing a crisp boundary around each of these abstractions is hard
3. This requires you to form a balanced distribution of responsibilities in the system as a whole, with individual abstractions that are tightly cohesive and with relationships that are expressive, yet loosely coupled
4. When you model these webs of relationships,

- Don't begin in isolation. Apply use cases and scenarios to drive your discovery of the relationships among a set of abstractions.
- In general, start by modeling the structural relationships that are present. These reflect the static view of the system and are therefore fairly tangible.
- Next, identify opportunities for generalization/specialization relationships; use multiple inheritance sparingly.
- Only after completing the preceding steps should you look for dependencies; they generally represent more-subtle forms of semantic connection.
- For each kind of relationship, start with its basic form and apply advanced features only as absolutely necessary to express your intent.
- Remember that it is both undesirable and unnecessary to model all relationships among a set of abstractions in a single diagram or view. Rather, build up your system's relationships by considering different views on the system. Highlight interesting sets of relationships in individual diagrams.

Interfaces, type and roles Interface

- An interface is a collection of operations that are used to specify a service of a class or a component type
- A type is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object.

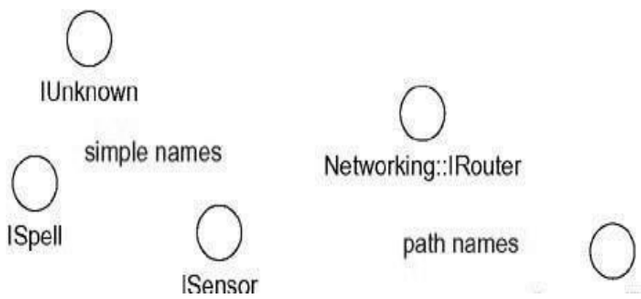
role

- A role is the behavior of an entity participating in a particular context.

an interface may be rendered as a stereotyped class in order to expose its operations and other properties.

Names

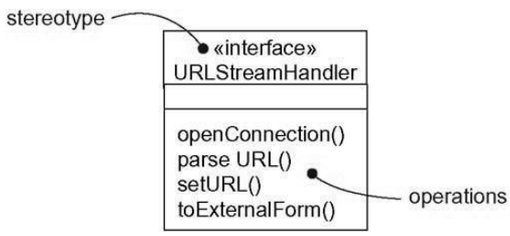
- Every interface must have a name that distinguishes it from other interfaces.
- A name is a textual string. That name alone is known as a simple name;
- A path name is the interface name prefixed by the name of the package



simple and Path Names

Operations

- An interface is a named collection of operations used to specify a service of a class or of a component.
- Unlike classes or types, interfaces do not specify any structure (so they may not include any attributes), nor do they specify any implementation
- These operations may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.
- you can render an interface as a stereotyped class, listing its operations in the appropriate compartment. Operations may be drawn showing only their name, or they may be augmented to show their full signature and other properties

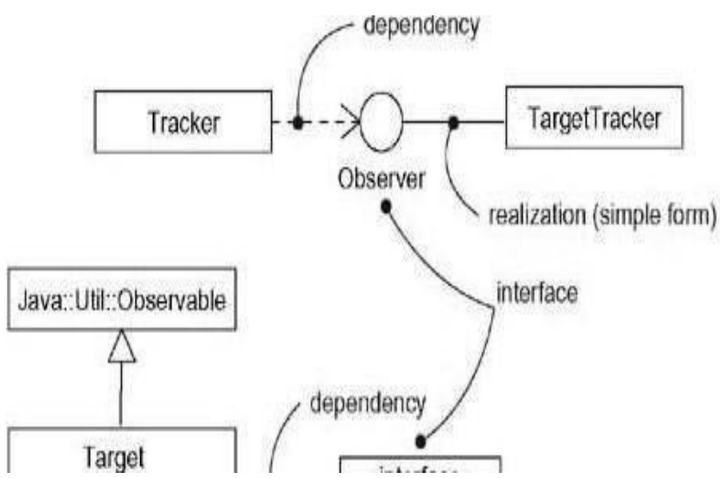


Operations Relationships

- Like a class, an interface may participate in generalization, association, and dependency relationships. In addition, an interface may participate in realization relationships.
- An interface specifies a contract for a class or a component without dictating its implementation. A class or component may realize many interfaces

  .We can show that an element realizes an interface in two ways.

  - First, you can use the simple form in which the interface and its realization relationship are rendered as a lollipop sticking off to one side of a class or component.
  - Second, you can use the expanded form in which you render an interface as a stereotyped class, which allows you to visualize its operations and other properties, and then draw a realization relationship from the classifier or component to the interface.



Realizations

Understanding an Interface

- In the UML, you can supply much more information to an interface in order to make it understandable and approachable.
- First, you may attach pre- and postconditions to each operation and invariants to the class or component as a whole. By doing this, a client who needs to use an interface will be able to understand what the interface does and how to use it, without having to dive into an implementation.
- We can attach a state machine to the interface. You can use this state machine to specify the legal partial ordering of an interface's operations.
- We can attach collaborations to the interface. You can use collaborations to specify the expected behavior of the interface through a series of interaction diagrams.

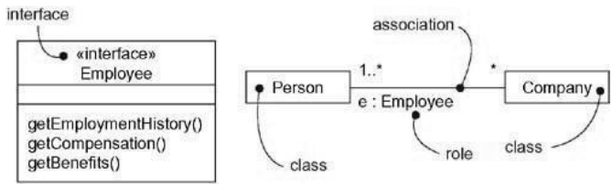
Types and Roles

A role names a behavior of an entity participating in a particular context. Stated another way, a role is the face that an abstraction presents to the world.

For example, consider an instance of the class Person. Depending on the context, that Person instance may play the role of Mother, Comforter, PayerOfBills, Employee, Customer, Manager, Pilot, Singer, and so on. When an object plays a particular role, it presents a face to the world, and clients that interact with it expect a certain behavior depending on the role that it plays at the time.

an instance of Person in the role of Manager would present a different set of properties than if the instance were playing the role of Mother.

In the UML, you can specify a role an abstraction presents to another abstraction by adorning the name of an association end with a specific interface.



Roles

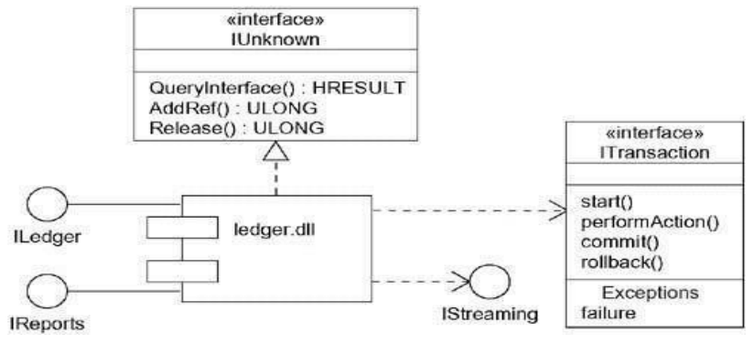
- A class diagram like this one is useful for modeling the static binding of an abstraction to its interface. You can model the dynamic binding of an abstraction to its interface by using the become stereotype in an interaction diagram, showing an object changing from one role to another.
- If you want to formally model the semantics of an abstraction and its conformance to a specific interface, you'll want to use the defined stereotype type
- Type is a stereotype of class, and you use it to specify a domain of objects, together with the operations (but not the methods) applicable to the objects of that type. The concept of type is closely related to that of interface, except that a type's definition may include attributes while an interface may not.

Common Modeling Techniques

- **Modeling the Seams in a Systemeling the Seams in a System**

The most common purpose for which you'll use interfaces is to model the seams in a system composed of software components, such as COM+ or Java Beans.

- Identifying the seams in a system involves identifying clear lines of demarcation in your architecture. On either side of those lines, you'll find components that may change independently, without affecting the components on the other side,



Modeling the Seams in a System

- The above Figure shows the seams surrounding a component (the library ledger.dll) drawn from a financial system. This component realizes three interfaces: IUnknown, ILedger, and IReports. In this diagram, IUnknown is shown in its expanded form; the other two are shown in their simple form, as lollipops. These three interfaces are realized by ledger.dll and are exported to other components for them to build on.
- As this diagram also shows, ledger.dll imports two interfaces, IStreaming and ITransaction, the latter of which is shown in its expanded form. These two interfaces are required by the ledger.dll component for its proper operation. Therefore, in a running system, you must supply components that realize these two interfaces.
- By identifying interfaces such as ITransaction, you've effectively decoupled the components on either side of the interface, permitting you to employ any component that conforms to that interface.

- **Modeling Static and Dynamic Typeseling Static and Dynamic Types**

- Most object-oriented programming languages are statically typed, which means that the type of an object is bound at the time the object is created.
- Even so, that object will likely play different roles over time.
- Modeling the static nature of an object can be visualized in a class diagram. However, when you are modeling things like business objects, which naturally change their roles throughout a workflow,
- To model a dynamic type

  - Specify the different possible types of that object by rendering each type as a class stereotyped as

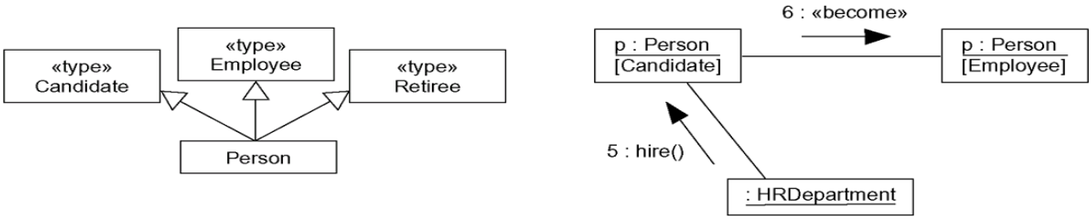
type (if the abstraction requires structure and behavior) or as interface (if the abstraction requires only behavior).

- Model all the roles the of the object may take on at any point in time. You can do so in two ways:

  1.) First, in a class diagram, explicitly type each role that the class plays in its association with Other classes. Doing this specifies the face instances of that class put on in the context of the associated object.

  2.) Second, also in a class diagram, specify the class-to-type relationships using generalization.

- In an interaction diagram, properly render each instance of the dynamically typed class. Display the role of the instance in brackets below the object's name.
- To show the change in role of an object, render the object once for each role it plays in the interaction, and connect these objects with a message stereotyped as become.



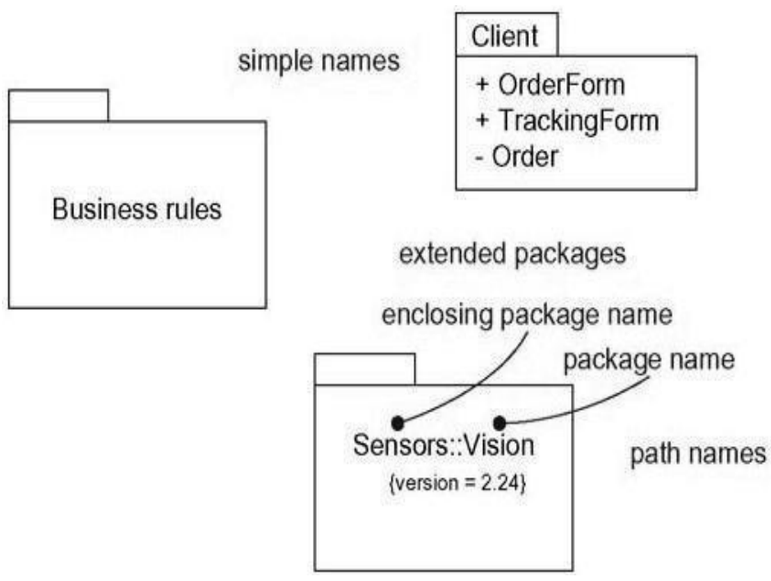
Modeling Static Types Modeling Dynamic Types

Package

"A package is a general-purpose mechanism for organizing elements into groups." Graphically, a package is rendered as a tabbed folder.

Names

- Every package must have a name that distinguishes it from other packages. A name is a textual string.
- That name alone is known as a simple name; a path name is the package name prefixed by the name of the package in which that package lives
- We may draw packages adorned with tagged values or with additional compartments to expose their details.



Simple and Extended Package Owned Elements

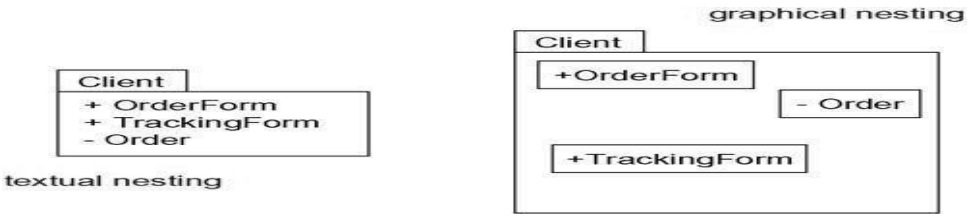
A package may own other elements, including classes, interfaces, components, nodes, collaborations, use cases, diagrams, and even other packages.

Owning is a composite relationship, which means that the element is declared in the package. If the package is destroyed, the element is destroyed. Every element is uniquely owned by exactly one package.

Elements of different kinds may have the same name within a package. Thus, you can have a class named Timer, as well as a component named Timer, within the same package.

Packages may own other packages. This means that it's possible to decompose your models hierarchically.

We can explicitly show the contents of a package either textually or graphically.



Owned Elements Visibility

You can control the visibility of the elements owned by a package just as you can control the visibility of the attributes and operations owned by a class.

Typically, an element owned by a package is public, which means that it is visible to the contents of any package that imports the element's enclosing package.

Conversely, protected elements can only be seen by children, and private elements cannot be seen outside the package in which they are declared.

We specify the visibility of an element owned by a package by prefixing the element's name with an appropriate visibility symbol.

Importing and Exporting

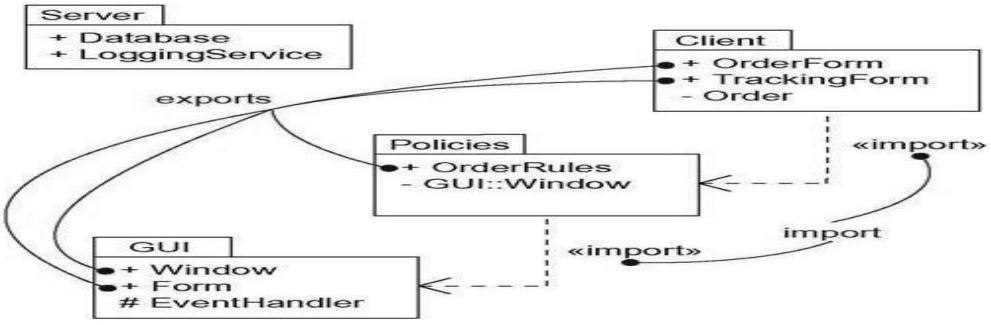
In the UML, you model an import relationship as a dependency adorned with the stereotype import.

Actually, two stereotypes apply here—import and access— and both specify that the source package has access to the contents of the target.

Import adds the contents of the target to the source's namespace Access does not add

the contents of the target The public parts of a package are called its exports.

The parts that one package exports are visible only to the contents of those packages that explicitly import the package. Import and access dependencies are not Transitive Importing and Exporting

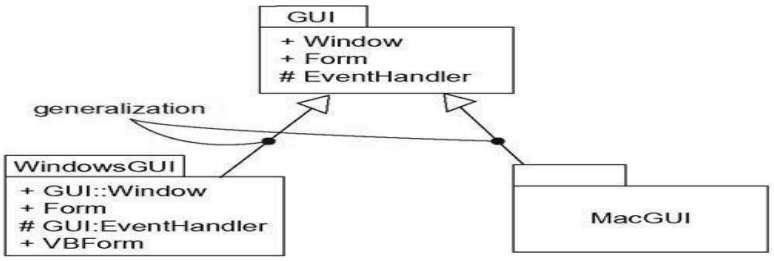


Generalization

There are two kinds of relationships you can have between packages: import and access dependencies used to import into one package elements exported from another and generalizations, used to specify families of packages

Generalization among packages is very much like generalization among classes Packages involved in generalization relationships follow the same principle of

substitutability as do classes. A specialized package (such as WindowsGUI) can be used anywhere a more general package (such as GUI) is used Generalization Among Packages



Standard Elements

- All of the UML's extensibility mechanisms apply to packages. Most often, you'll use tagged values to add new package properties (such as specifying the author of a package) and stereotypes to specify new kinds of packages (such as packages that encapsulate operating system services).
- The UML defines five standard stereotypes that apply to packages

1. facade	Specifies a package that is only a view on some other package
2. framework	Specifies a package consisting mainly of patterns
3. stub	Specifies a package that serves as a proxy for the public contents of another package
4. subsystem	Specifies a package representing an independent part of the entire system being modeled
5. system	Specifies a package representing the entire system being modeled

The UML does not specify icons for any of these stereotypes

Common Modeling Techniques Modeling Groups of Elements

The most common purpose for which you'll use packages is to organize modeling elements into groups that you can name and manipulate as a set. There is one important distinction between classes and packages:

Packages have no identity (meaning that you can't have instances of packages, so they are invisible in the running system);
classes do have identity (classes have instances, which are elements of a running system). To model groups of elements,

Scan the modeling elements in a particular architectural view and look for clumps defined by elements that are conceptually or semantically close to one another. Surround each of these clumps in a package.

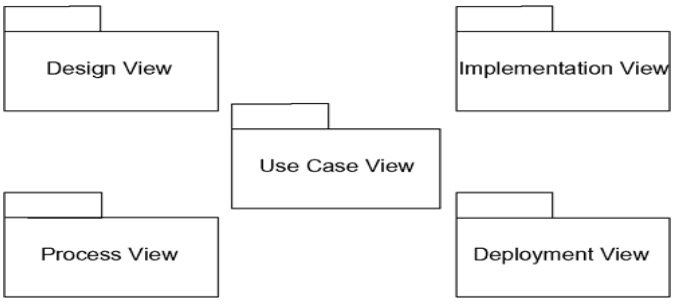
For each package, distinguish which elements should be accessible outside the package. Mark them public, and all others protected or private. When in doubt, hide the element.

Explicitly connect packages that build on others via import dependencies

In the case of families of packages, connect specialized packages to their more general part via generalizations

Modeling Architectural Views

- We can use packages to model the views of an architecture.
- Remember that a view is a projection into the organization and structure of a system, focused on a particular aspect of that system.
- This definition has two implications. First, you can decompose a system into almost orthogonal packages, each of which addresses a set of architecturally significant decisions.(design view, a process view, an implementation view, a deployment view, and a use case view)
- Second, these packages own all the abstractions germane to that view.(Implementation view)
- To model architectural views,
- Identify the set of architectural views that are significant in the context of your problem.
  In practice, this typically includes a design view, a process view, an implementation view, a deployment view, and a use case view.
- Place the elements (and diagrams) that are necessary and sufficient to visualize, specify, construct, and document the semantics of each view into the appropriate package.
- As necessary, further group these elements into their own packages.
- There will typically be dependencies across the elements in different views. So, in general, let each view at the top of a system be open to all others at that level.



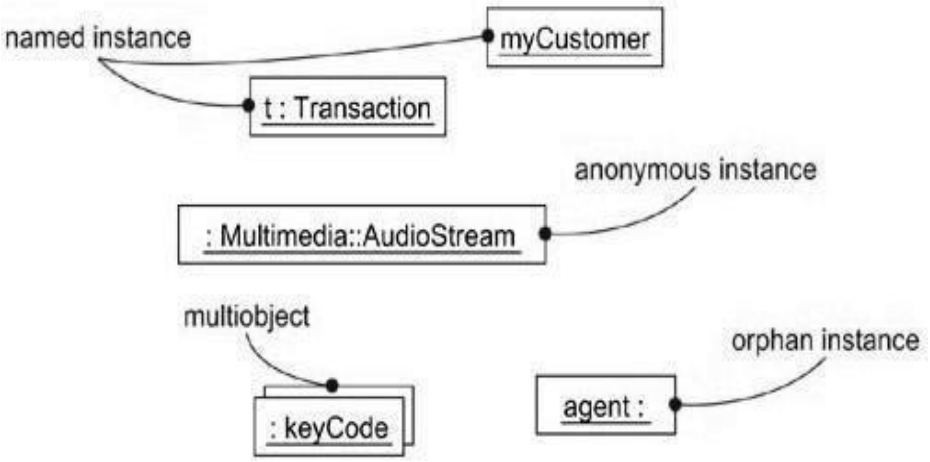
Instances

- An instance is a concrete manifestation of an abstraction to which a set of operations can be applied and which has a state that stores the effects of the operations.
- Graphically, an instance is rendered by underlining its name.

Abstractions and Instances

- Most instances you'll model with the UML will be instances of classes although you can have instances of other things, such as components, nodes, use cases, and associations
- In the UML, an instance is easily distinguishable from an abstraction. To indicate an instance, you underline its name.

  We can use the UML to model these physical instances, but you can also model things that are not so concrete.



Named, Anonymous, Multiple, and Orphan Instances

Names

- Every instance must have a name that distinguishes it from other instances within its context.
- Typically, an object lives within the context of an operation, a component, or a node.
- A name is a textual string. That name alone is known as a simple name. or it may be a path name

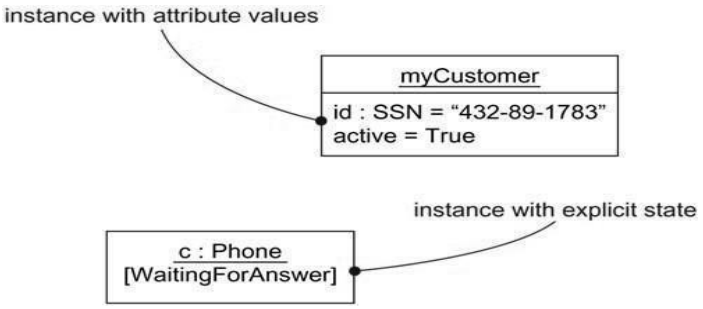
Operations

- The operations you can perform on an object are declared in the object's abstraction
- For example, if the class Transaction defines the operation commit, then given the instance t : Transaction, you can write expressions such as t. commit()

State

An object also has state. An object's state is therefore dynamic. So when you visualize its state, you are really specifying the value of its state at a given moment in time and space.

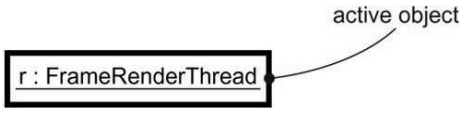
It's possible to show the changing state of an object by showing it multiple times in the same interaction diagram, but with each occurrence representing a different state. When you operate on an object, you typically change its state; when you query an object, you don't change its state



Other Features

Processes and threads are an important element of a system's process view, so the UML provides a visual cue to distinguish elements that are active from those that are passive.

You can declare active classes that reify a process or thread, and in turn you can distinguish an instance of an active class

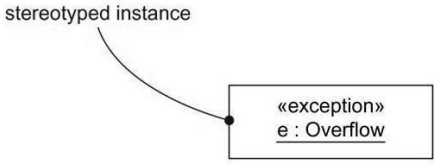


Active Objects

- There are two other elements in the UML that may have instances
- The first is a link. A link is a semantic connection among objects. An instance of an association is therefore a link. A link is rendered as a line
- The second is a class-scoped attribute and operation. A class-scoped feature is in effect an object in the class that is shared by all instances of the class.

Standard Elements

- All of the UML's extensibility mechanisms apply to objects. Usually, however, you don't stereotype an instance directly, nor do you give it its own tagged values. Instead, an object's stereotype and tagged values derive from the stereotype and tagged values



of its associated abstraction. Stereotyped

Objects

The UML defines two standard stereotypes that apply to the dependency relationships among objects and among classes:

1. instanceOf	Specifies that the client object is an instance of the supplier classifier
2. instantiate	Specifies that the client class creates instances of the supplier class

There are also two stereotypes related to objects that apply to messages and transitions:

1. become	Specifies that the client is the same object as the supplier, but at a later time and with possibly different values, state, or roles
2. copy	Specifies that the client object is an exact but independent copy of the supplier

The UML defines a standard constraint that applies to objects:

transient	Specifies that an instance of the role is created during execution of the enclosing interaction but is destroyed before completion of execution

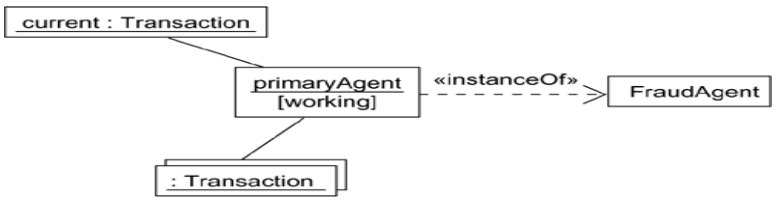
Common Modeling Techniques

Modeling Concrete Instances

- When you model concrete instances, you are in effect visualizing things that live in the real world

- One of the things for which you'll use objects is to model concrete instances that exist in the real world
- To model concrete instances,

- Identify those instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.
- Render these objects in the UML as instances. Where possible, give each object a name. If there is no meaningful name for the object, render it as an anonymous object.
- Expose the stereotype, tagged values, and attributes (with their values) of each instance necessary and sufficient to model your problem.
- Render these instances and their relationships in an object diagram or other diagram appropriate to the kind of the instance.



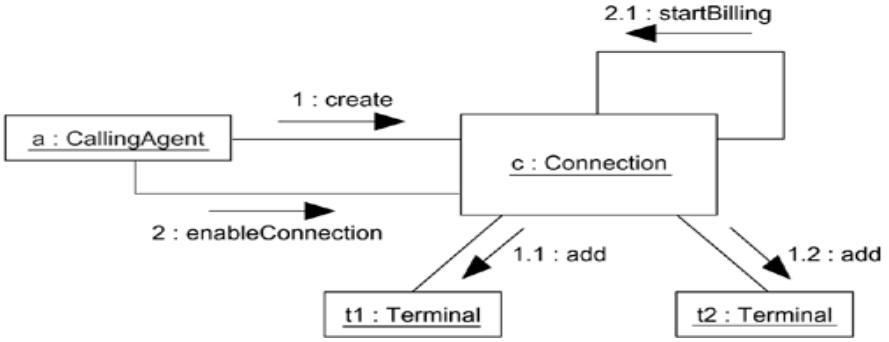
Modeling Concrete Instances

Modeling Prototypical Instances

- Perhaps the most important thing for which you'll use instances is to model the dynamic interactions among objects. When you model such interactions, you are generally not modeling concrete instances that exist in the real world.
- These are prototypical objects and, therefore, are roles to which concrete instances conform.
- Concrete objects appear in static places, such as object diagrams, component diagrams, and deployment diagrams.
- Prototypical objects appear in such places as interaction diagrams and activity diagrams.
- To model prototypical instances,

- Identify those prototypical instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.
- Render these objects in the UML as instances. Where possible, give each object a name. If there is no meaningful name for the object, render it as an anonymous object.

- Expose the properties of each instance necessary and sufficient to model your problem.
- Render these instances and their relationships in an interaction diagram or an activity diagram.



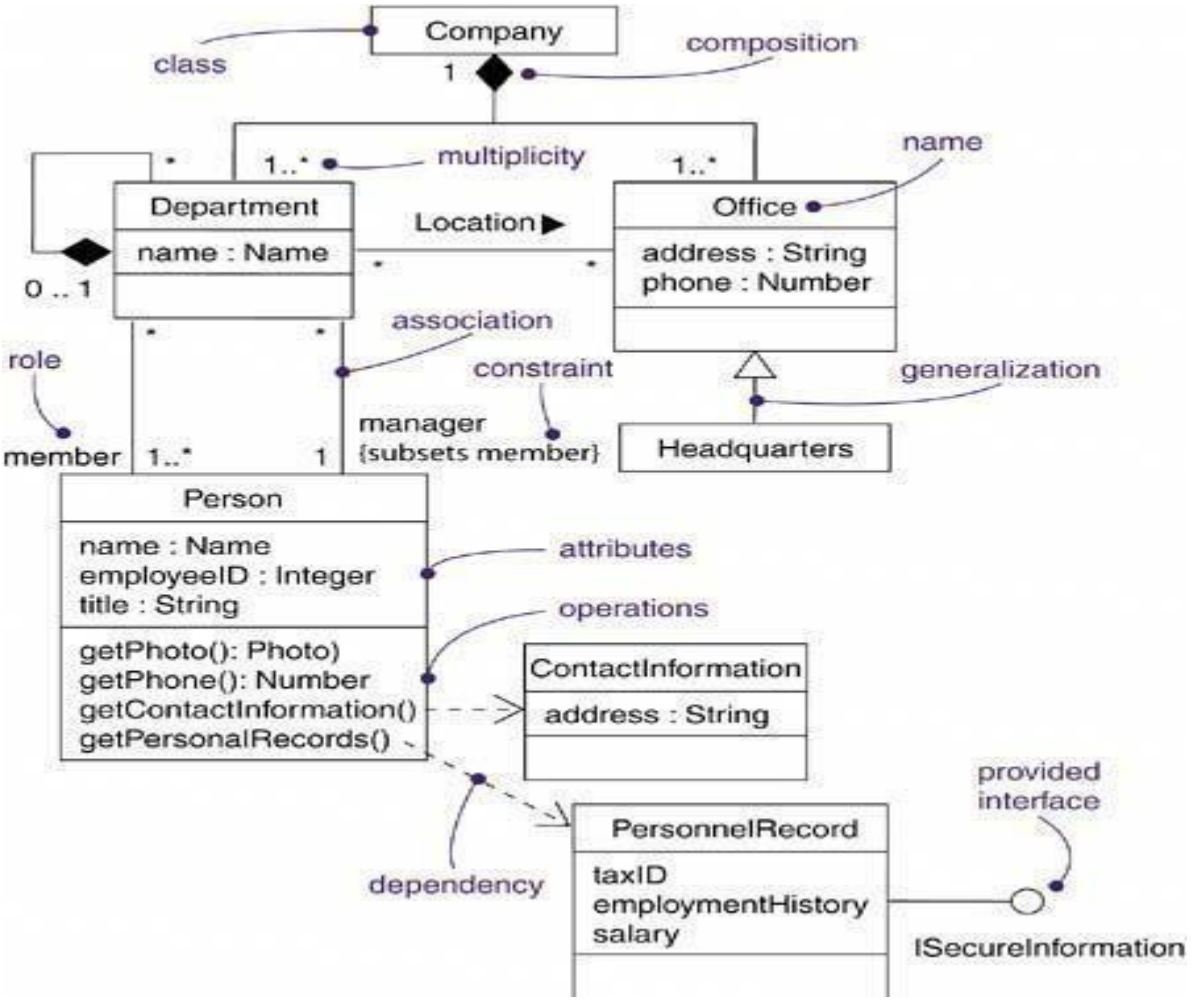
CLASS DIAGRAMS

Class diagrams are the most common diagram found in modeling object-oriented systems. A class diagram shows a set of classes, interfaces, and collaborations and their relationships.

Class diagrams are important not only for visualizing, specifying, and documenting structural models, but also for constructing executable systems through forward and reverse engineering.

Building software has much the same characteristics except that, given the fluidity of software, we have the ability to define your own basic building blocks from scratch. With the UML, you

use class diagrams to visualize the static aspects of these building blocks and their relationships and to specify their details for construction, as you can see in Figure



Common Properties

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

Class diagrams commonly contain the following things:

- Classes
- Interfaces
- Collaborations
- Dependency, generalization, and association relationships Like all other diagrams, class diagrams

may contain notes and constraints.

Class diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes you'll want to place instances in class diagrams as well, especially when we want to visualize the (possibly dynamic) type of an instance.

Common Uses

Use class diagrams to model the static design view of a system. This view primarily supports

the functional requirements of a system the services the system should provide to its end users.

When you model the static design view of a system, you'll typically use class diagrams in one of three ways.

1. To model the vocabulary of a system

   Modeling the vocabulary of a system involves making a decision about which abstractions are a

   part of the system under consideration and which fall outside its boundaries. You use class diagrams to specify these abstractions and their responsibilities.

2. To model simple collaborations

   A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. For example,

   when you re modeling the semantics of a transaction in a distributed system, you can't just stare at a single class to understand what's going on. Rather, these semantics are carried out by a set of classes that work together. You use class diagrams to visualize and specify this set of classes and their relationships.

3. To model a logical database schema

   Think of a schema as the blueprint for the conceptual design of a database. In many domains, we want to store persistent information in a relational database or in an object-oriented database. We can model schemas for these databases using class diagrams.

COMMON MODELING TECHNIQUES

Modeling Simple Collaborations

No class stands alone. Rather, each works in collaboration with others to carry out some semantics greater than each individual. Therefore, in addition to capturing the vocabulary of your system, you'll also need to turn your attention to visualizing,

specifying, constructing, and documenting the various ways these things in your vocabulary work together. Weuse class diagrams to represent such collaborations.

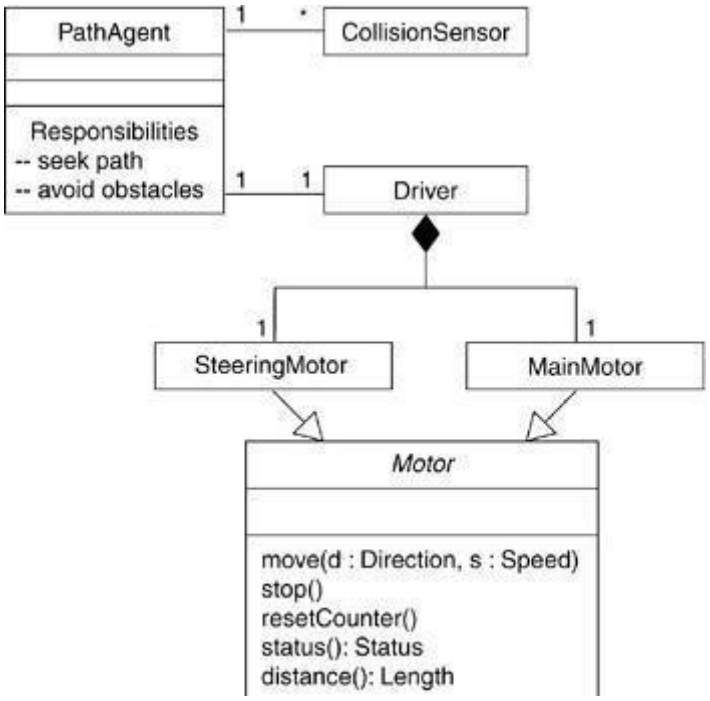
To model a collaboration,

- Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.

- For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration. Identify the relationships among these things as well.
- Use scenarios to walk through these things. Along the way, you'll discover parts of your model that were missing and parts that were just plain semantically wrong.
- Be sure to populate these elements with their contents. For classes, start with getting a good balance of responsibilities. Then, over time, turn these into concrete attributes and operations.

For example, Figure shows a set of classes drawn from the implementation of an autonomous robot. The figure focuses on the classes involved in the mechanism for moving the robot along

a path. You'll find one abstract class (Motor) with two concrete children, Steering Motor and Main Motor. Both of these classes inherit the five operations of their parent, Motor. The two classes are, in turn, shown as parts of another class, Driver. The class Path Agent has a one-to-one association to Driver and a one-to-many association to Collision Sensor. No attributes or operations are shown for Path Agent, although its responsibilities are given.



MODELING A LOGICAL DATABASE SCHEMA

Many of the systems you'll model will have persistent objects, which means that they can be stored in a database for later retrieval. Most often, you'll use a relational database, an object oriented database, or a hybrid object/relational database for persistent storage. The UML is well suited to modeling logical database schemas, as well as physical databases themselves.

The UML's class diagrams are a superset of entity-relationship (E-R) diagrams, a common modeling tool for logical database design. Whereas classical E-R diagrams focus only on data, class diagrams go a step further by permitting the modeling of behavior as well. In the physical database, these logical operations are generally turned into triggers or stored procedures.

To model a schema,

Identify those classes in your model whose state must transcend the lifetime of their applications.

- Create a class diagram that contains these classes. You can define your own set of stereotypes and tagged values to address database-specific details.
- Expand the structural details of these classes. In general, this means specifying the details of their attributes and focusing on the associations and their multiplicities that

relate these classes.

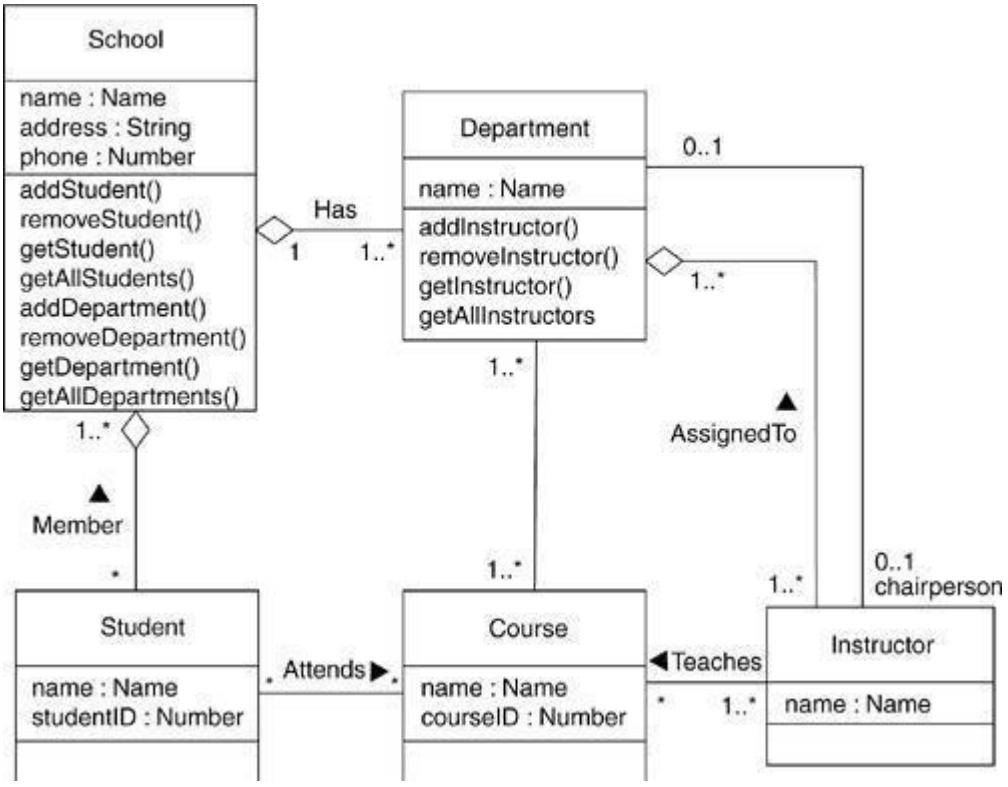
- Watch for common patterns that complicate physical database design, such as cyclic associations and one-to-one associations. Where necessary, create intermediate abstractions to simplify your logical structure.
- Consider also the behavior of these classes by expanding operations that are important for data access and data integrity.

The Figure shows a set of classes drawn from an information system for a school.

We find the classes named Student, Course, and Instructor. There's an association between Student and Course, specifying that students attend courses. Furthermore, every student

may attend any number of courses, and every course may have any number of students.

Figure: Modeling a Schema



FORWARD AND REVERSE ENGINEERING

For some uses of the UML, the models we create will never map to code. For example, if you are modeling a business process using activity diagrams, many of the activities we model will involve people, not computers. In other cases, you'll want to model systems whose parts are, from your level of abstraction, just a piece of hardware.

Forward engineering is the process of transforming a model into code through a mapping to an implementation language. Forward engineering results in a loss of information, because models written in the UML are semantically richer than any current object- oriented programming language. In fact, this is a major reason why you need models in addition to code.

Structural features, such as collaborations, and behavioral features, such as interactions, can be visualized clearly in the UML, but not so clearly from raw code.

To forward engineer a class diagram,

- Identify the rules for mapping to your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Depending on the semantics of the languages you choose, you may want to constrain your use of certain UML features. For example, the UML permits you to model multiple inheritance, but Smalltalk permits only single inheritance.
- Use tagged values to guide implementation choices in your target language. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages. Use tools to generate code.

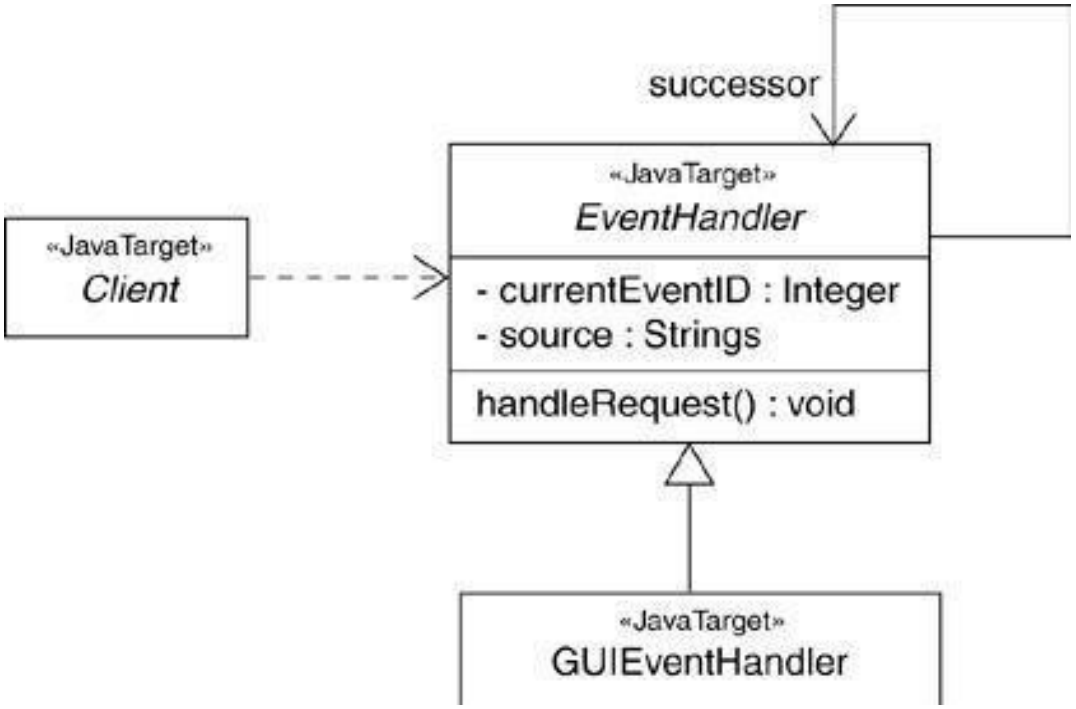


Figure: Forward Engineering

Reverse engineering is the process of transforming code into a model through a mapping from a specific implementation language. Reverse engineering results in a flood of information, some of which is at a lower level of detail than you'll need to build useful models. At the same time, reverse engineering is incomplete.

To reverse engineer a class diagram,

- Identify the rules for mapping from your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered. It is unreasonable to expect to reverse engineer a single concise model from a large body of code. You need to select portion of the code and build the model from the bottom.
- Using your tool, create a class diagram by querying the model. For example, you might start with one or more classes, then expand the diagram by following specific relationships or other neighboring classes. Expose or hide details of the contents of this class diagram as necessary to communicate your intent.

- Manually add design information to the model to express the intent of the design that is missing or hidden in the code.

OBJECT DIAGRAMS

Object diagrams model the instances of things contained in class diagrams. An object diagram shows a set of objects and their relationships at a point in time.

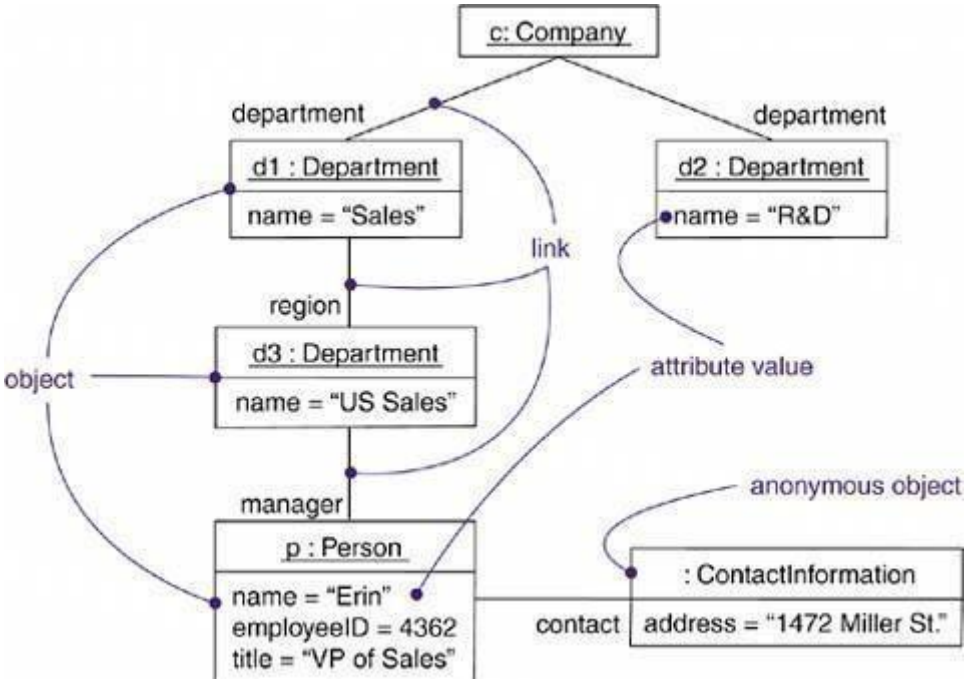
We use object diagrams to model the static design view or static process view of a system. This involves modeling a snapshot of the system at a moment in time and rendering a set of objects, their state, and their relationships.

Object diagrams are not only important for visualizing, specifying, and documenting structural models, but also for constructing the static aspects of systems through forward and reverse engineering.

An object diagram covers a set of instances of the things found in a class diagram. An object diagram,

therefore, expresses the static part of an interaction, consisting of the objects that collaborate but without any of the messages passed among them. In both cases, an object diagram freezes a moment in time, as in Figure .

Figure. An Object Diagram



An *object diagram* is a diagram that shows a set of objects and their relationships at a point in time. Graphically, an object diagram is a collection of vertices and arcs.

Common Properties

An object diagram is a special kind of diagram and shares the same common properties as all other diagrams that is, a name and graphical contents that are a projection into a model.

What distinguishes an object diagram from all other kinds of diagrams is its particular content.

Contents

Object diagrams commonly contain

- Objects
- Links

Like all other diagrams, object diagrams may contain notes and constraints.

COMMON MODELING TECHNIQUES:-

Modeling Object Structures

When you construct a class diagram, a component diagram, or a deployment diagram, what we are doing is capturing a set of abstractions that are interesting to you as a group and, in that context, exposing their semantics and their relationships to other abstractions in the group.

If class A has a one-to-many association to class B, then for one instance of A there might be five instances of B; for another instance of A there might be only one instance of B. Furthermore, at a given moment in time, that instance of A, along with the related instances of B, will each have certain values for their attributes and state machines.

To model an object structure, Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.

- Create a collaboration to describe a mechanism.
- For each mechanism, identify the classes, interfaces, and other elements that participate in this collaboration; identify the relationships among these things as well.
- Consider one scenario that walks through this mechanism. Freeze that scenario at a moment in time, and render each object that participates in the mechanism.
- Expose the state and attribute values of each such object, as necessary, to understand the scenario.
- Similarly, expose the links among these objects, representing instances of associations among them.

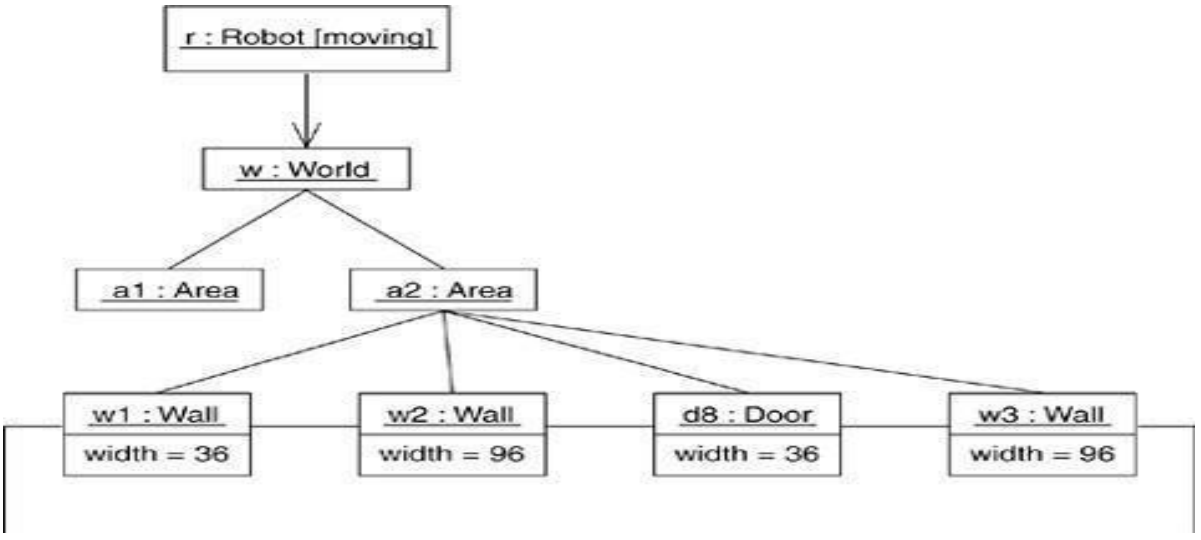


Figure : Modeling Object Structures

FORWARD AND REVERSE ENGINEERING

Forward engineering (the creation of code from a model) an object diagram is theoretically possible but pragmatically of limited value. In an object-oriented system, instances are things that are created and destroyed by the application during run time. Therefore, you cannot exactly instantiate these objects from the outside.

Reverse engineering (the creation of a model from code) an object diagram can be useful.In fact, while you are debugging your system, this is something that you or your tools will do all the time.

To reverse engineer an object diagram,

- Chose the target you want to reverse engineer. Typically, you'll set your context inside an operation or relative to an instance of one particular class.
- Using a tool or simply walking through a scenario, stop execution at a certain moment in time.
- Identify the set of interesting objects that collaborate in that context and render them in an object diagram.
- As necessary to understand their semantics, expose these object's states.
- As necessary to understand their semantics, identify the links that exist among these objects.
- If your diagram ends up overly complicated, prune it by eliminating objects that are not germane to the questions about the scenario you need answered. If your diagram is too simplistic, expand the neighbors of certain interesting objects and expose each object's state more deeply.

----------**END**----------

UNIT- 3

Interactions

Unit 3 Basic Behavioural Modeling

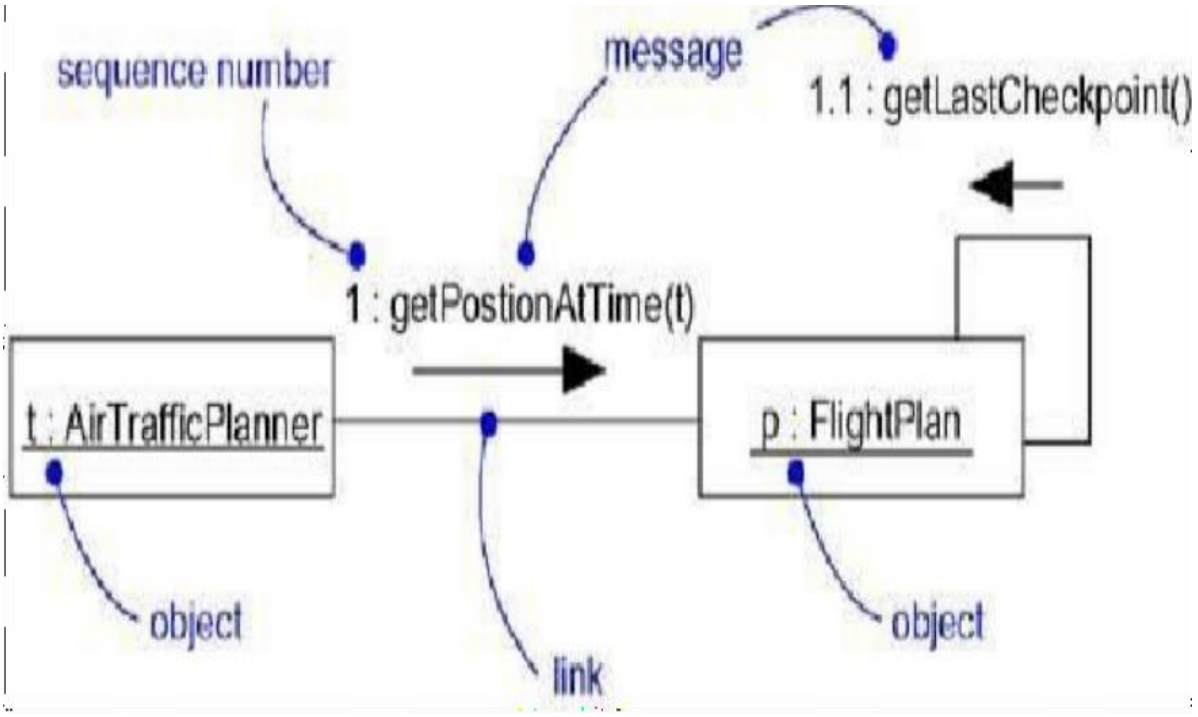
Use cases – Terms and Concepts, **Use case Diagrams** – Terms and Concepts, **Sequence Diagra** – Terms and Concepts, Collaboration Diagrams – Terms and Concepts.

Interface:-

• An interaction is a behavior that is composed of a set of messages exchanged among a set of objects within a context to accomplish a purpose.

• A message specifies the communication between objects for an activity to happen. It has following parts: its name, parameters (if any), and sequence number.

• Objects in an interaction can be concrete things or prototypical things.



A link is a semantic connection(path) among objects through which a message/s can be send. A link is an instance of an association. The semantics of link can be enhanced by using following prototypes as adornments

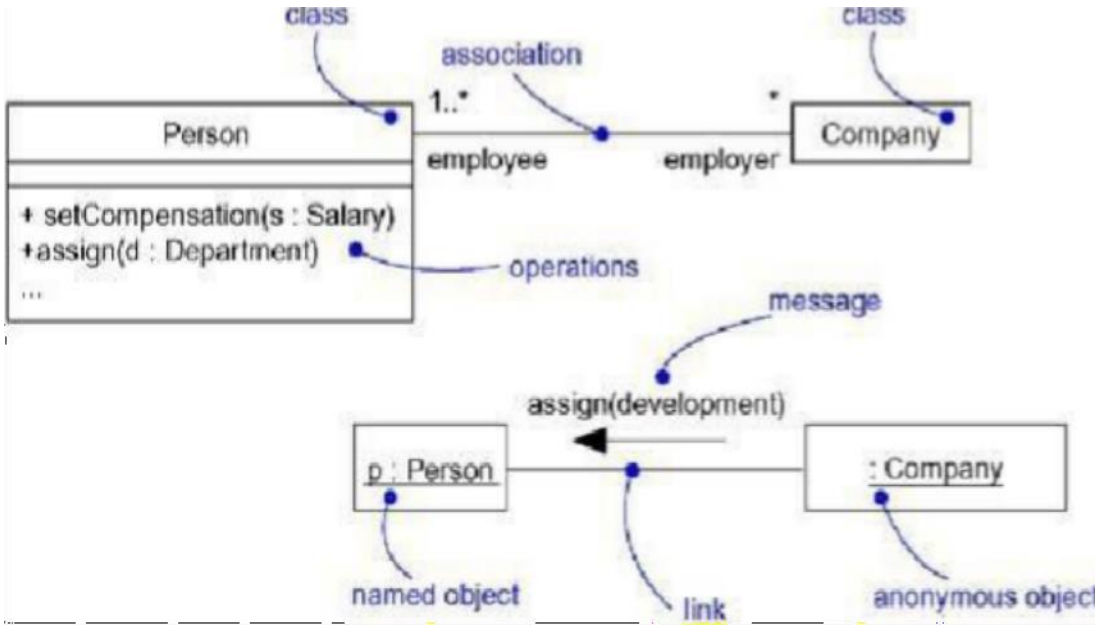
<<association>> – Specifies that the corresponding object is visible by association

<<self>> – Specifies that the corresponding object is visible because it is the dispatcher of the operation

<<global>> – Specifies that the corresponding object is visible because it is in an enclosing scope

<<local>> – Specifies that the corresponding object is visible because it is in a local scope

<<parameter>> – Specifies that the corresponding object is visible because it is a parameter

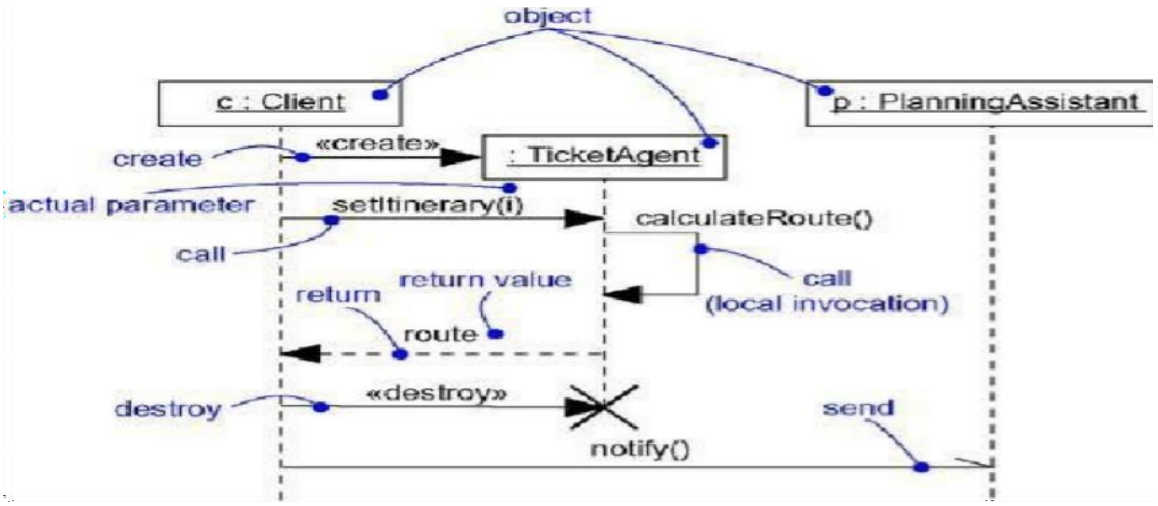


message – indicates an action to be done. Complex expressions can be written on arbitrary string of message. Different types of messages are:

- Call -Invokes an operation on an object; an object may send a message to itself, resulting in the local invocation of an operation
    - Return – Returns a value to the caller
    - Send – Sends a signal to an object
    - Create – Creates an object
    - Destroy – Destroys an object; an object may commit suicide by destroying itself

Sequencing

• a sequence is a stream of messages exchange between objects



• sequence must have a beginning and is rooted in some process or thread

• sequence will continue as long as the process or thread that owns it lives

Flow of control (2 types)

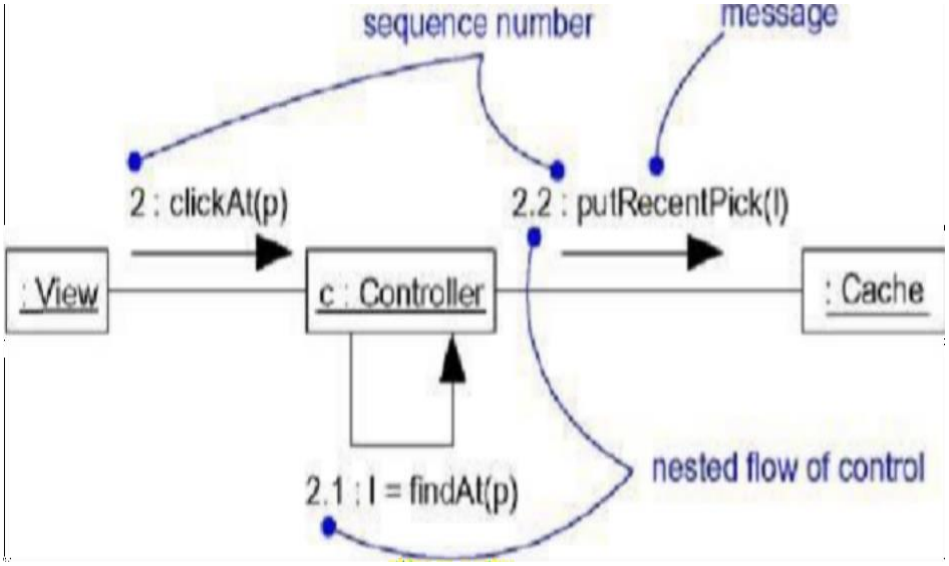
In each flow of control, messages are ordered in sequence by time and are visualized by prefixing the message with a sequence number set apart by a colon separator

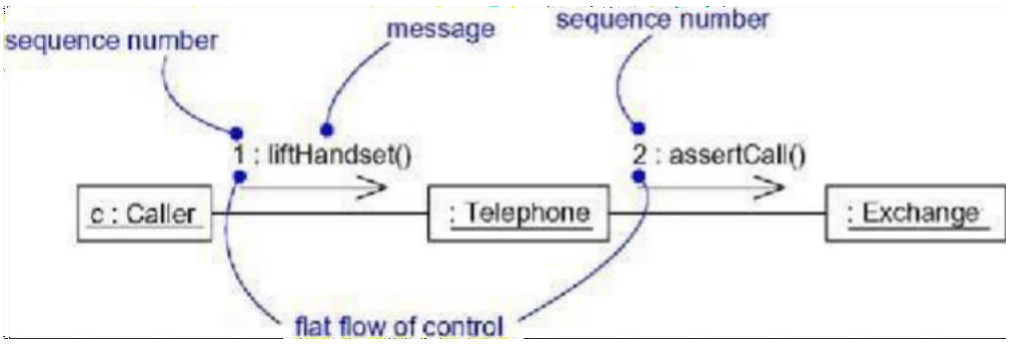
• A *procedural or nested flow of control* is rendered by using a filled solid arrowhead,

• A *flat flow of control* is rendered by using a stick arrowhead

• *Distinguishing one flow of control from another* by prefixing a message's sequence number with the name of the process or thread that sits at the root of the sequence.

• more-complex forms of sequencing, such as iteration, branching, and guarded messages can be modeled in UML.





Creation, Modification, and Destruction of links

Enabled by *adding the following constraints to the element*

- new – Specifies that the instance or link is created during execution of the enclosing interaction
- destroyed – Specifies that the instance or link is destroyed prior to completion of execution of the enclosing interaction
- transient – Specifies that the instance or link is created during execution of the enclosing interaction but is destroyed before completion of execution

Representation of interactions

interaction goes together with objects and messages.

represented by time ordering of its messages (sequence diagram), and by emphasizing the structural organization of these objects that send and receive messages (collaboration diagram)

Common Modeling Techniques

Modeling a Flow of Control

To model a flow of control,

- Set the context for the interaction, whether it is the system as a whole, a class, or an individual operation
- Set the stage for the interaction by identifying which objects play a role; set their initial properties, including their attribute values, state, and role
- If model emphasizes the structural organization of these objects, identify the links that connect them, relevant to the paths of communication that take place in this interaction
- In time order, specify the messages that pass from object to object As necessary, distinguish the different kinds of messages; include parameters and return values to convey the necessary detail of this interaction
- Also to convey the necessary detail of this interaction, adorn each object at every moment in time with its state and role.

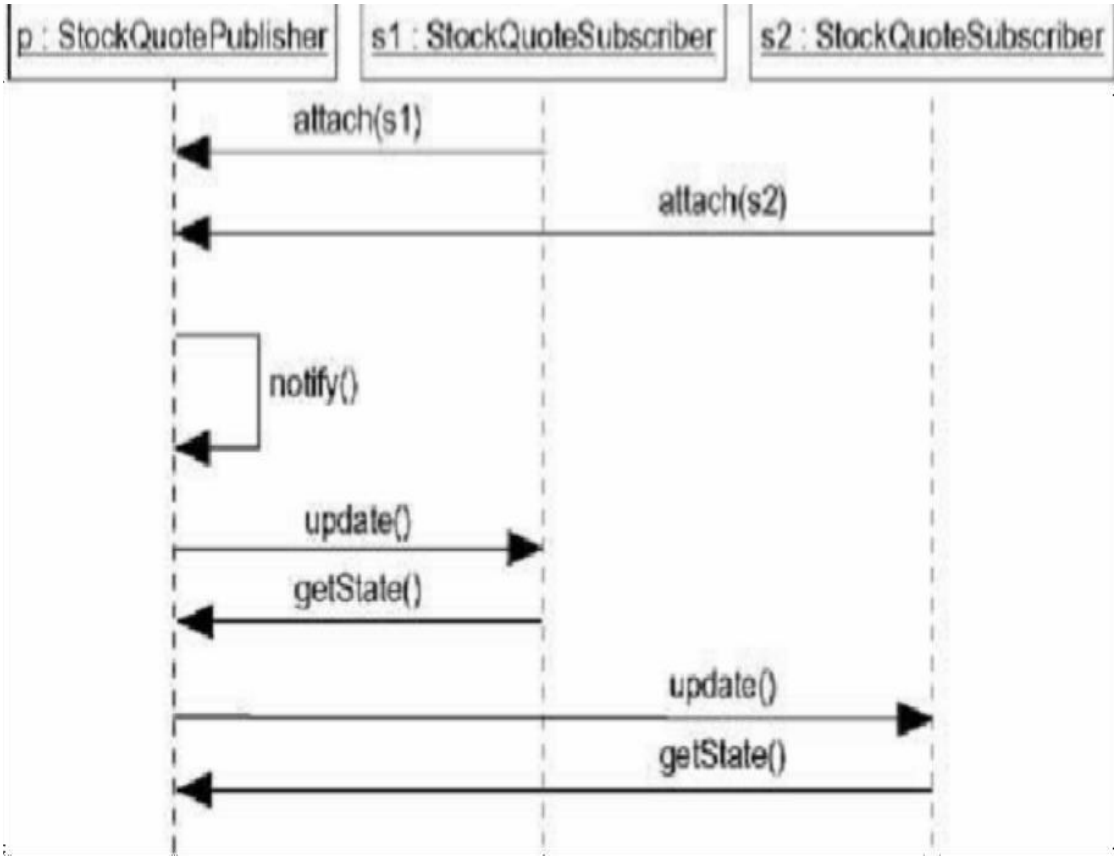


Figure 6: Flow of Control by time

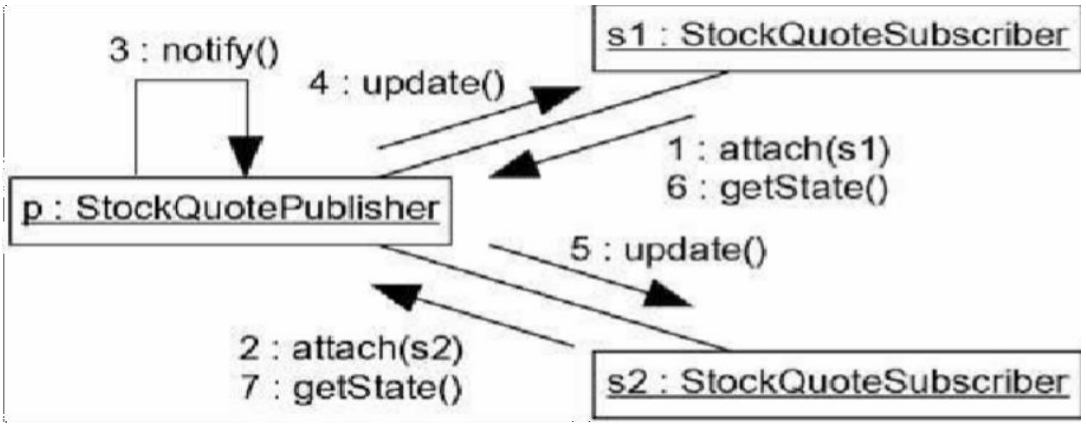


Figure 7: Flow of Control by Organization

Interaction Diagrams

- An interaction diagram shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them
- Interaction diagrams commonly contain Objects, Links, Messages
- interaction diagrams are used to model the dynamic aspects of a system
- An interaction diagram is basically a projection of the elements found in an interaction.
- It may contain notes and constraints.

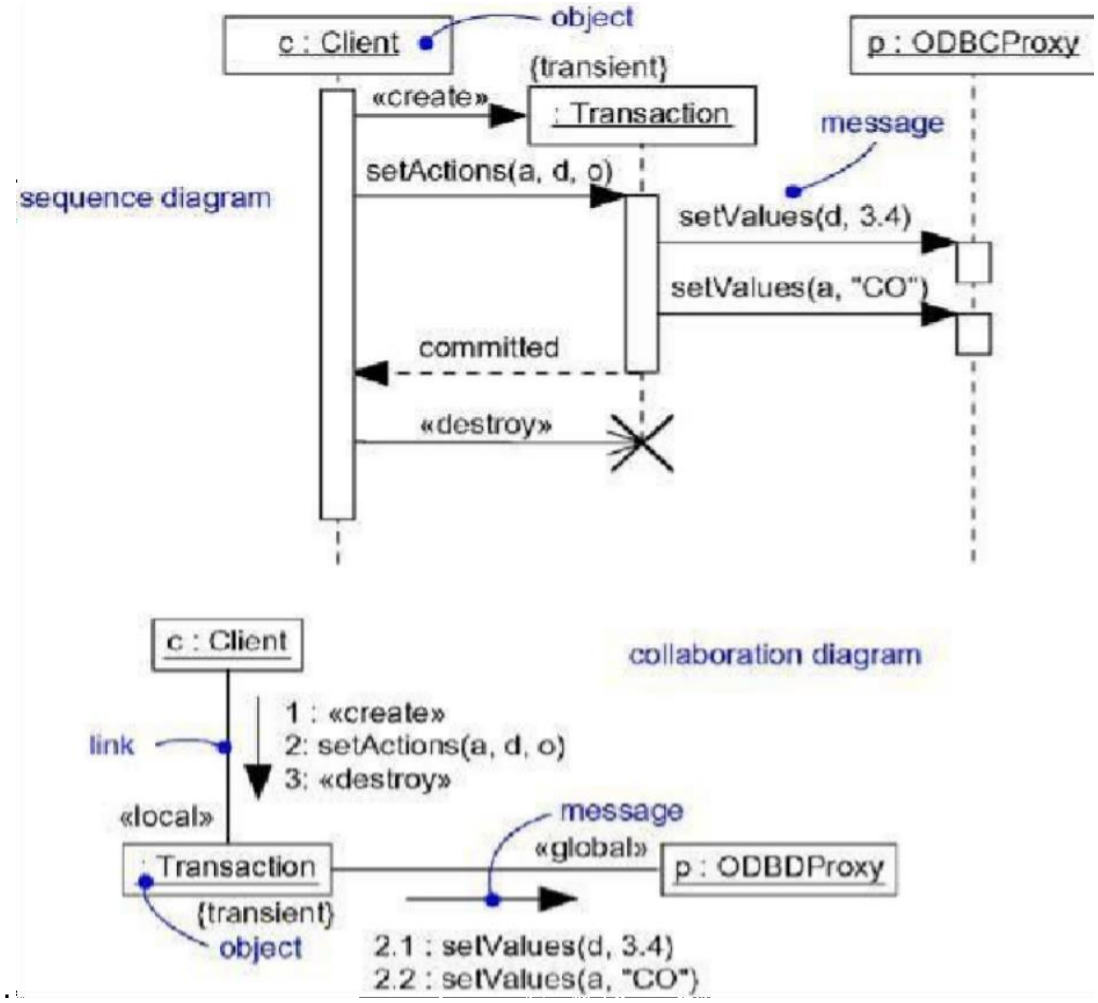


Fig:Interaction Diagrams

Sequence Diagrams

- A sequence diagram is an interaction diagram that emphasizes the time ordering of messages
- Graphically it is a table that shows objects arranged along the X axis and messages ordered in increasing time along the Y axis
- place the objects that participate in the interaction at the top of your diagram, across the X axis, object that initiates the interaction at the left, and increasingly more subordinate objects to the right
- place the messages that these objects send and receive along the Y axis, in order of increasing time from top to bottom

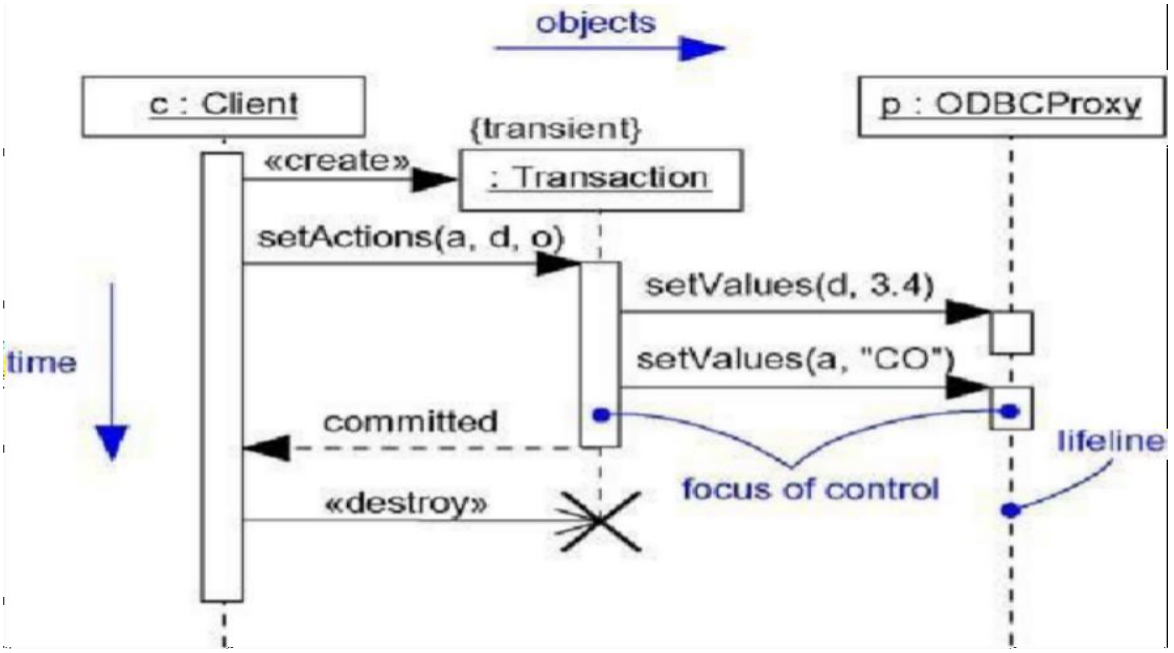


Figure: Sequence Diagram

Sequence diagrams have two features that distinguish them from collaboration diagrams

- First, there is the *object lifeline* which is a vertical dashed line that represents the existence of an object over a period of time
- Second, there is the *focus of control* which is a tall, thin rectangle that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure

Collaboration Diagrams

- A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages
- Graphically it is a collection of vertices and arcs
- more-complex flows, involving iterations and branching are modeled as [i := 1n] (or just), [x > 0]

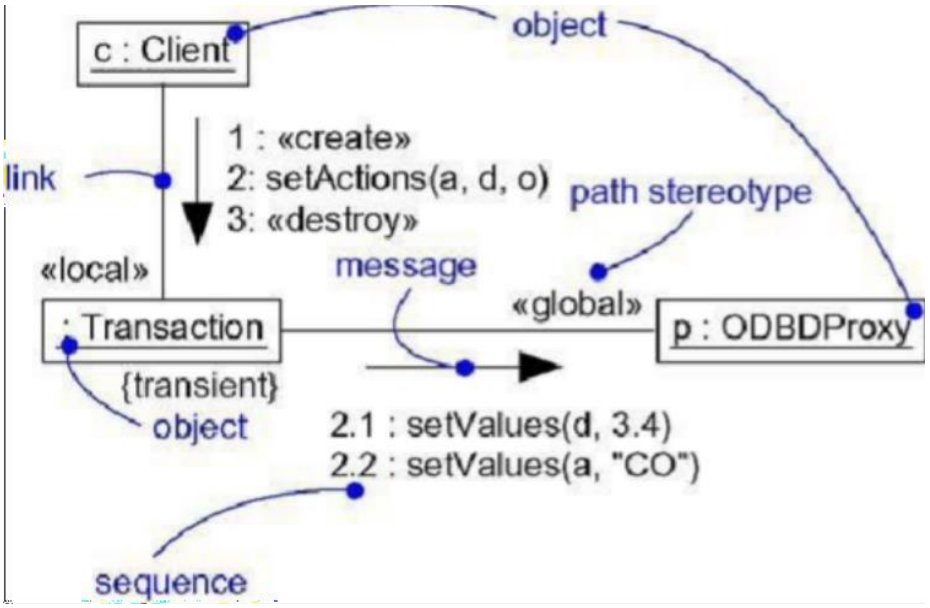


Figure : Collaboration Diagram

Collaboration diagrams* have two *features that distinguish* them from *sequence diagrams

- First, there is the *path* to indicate how one object is linked to another, attach a path stereotype to the far end of a link such as local, parameter, global, and self
- Second, there is the *sequence number* to indicate the time order of a message denoted by prefixing the message with a number, nesting is indicated by Dewey decimal numbering (eg:- 1 is the first message; 1.1 is the first message nested in message 1.)

Semantic Equivalence

sequence diagrams and collaboration diagrams are semantically equivalent that means conversion to the other is possible without any loss of information.

Common Modeling Techniques

Modeling Flows of Control by Time Ordering

To model a flow of control by time ordering,

- Set the context for the interaction, whether it is a system, subsystem, operation, or class or one scenario of a use case or collaboration
- Set the stage for the interaction by identifying which objects play a role in the

interaction.

- Set the lifeline for each object. Objects will persist through the entire interaction. For those objects that are created and destroyed during the interaction, set their lifelines, as appropriate, and explicitly indicate their birth and death with appropriately stereotyped messages
- Starting with the message that initiates this interaction, lay out each subsequent message from top to bottom between the lifelines, showing each message's properties .
- If you need to visualize the nesting of messages or the points in time when actual computation is taking place, adorn each object's lifeline with its focus of control
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints
- If you need to specify this flow of control more formally, attach pre- conditions and post-conditions to each message.

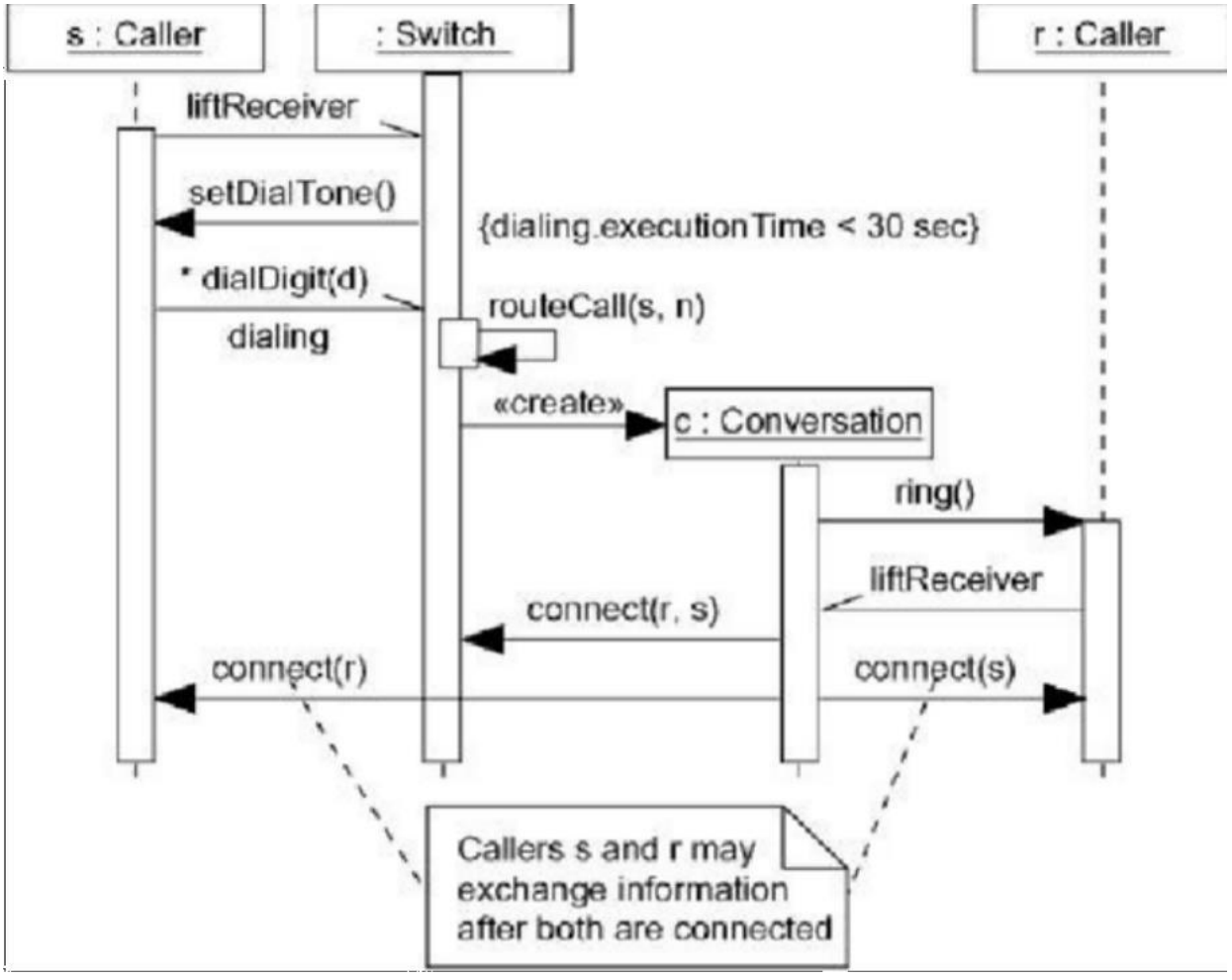


Figure : Modeling Flows of Control by Time Ordering

Modeling Flows of Control by Organization

To model a flow of control by organization

• Set the context for the interaction, whether it is a system, subsystem, operation, or class or one scenario of a use case or collaboration

• Set the stage for the interaction by identifying which objects play a role in the interaction.

• Set the initial properties of each of these objects If the attribute values, tagged values, state or role of any object changes in significant ways over the duration of the interaction, place a duplicate object on the diagram, update it with these new values, and connect them by a message stereotyped as become or copy .

• Specify the links among these objects, along which messages may pass

1. Lay out the association links first; these are the most important ones, because they represent structural connections
2. Lay out other links next, and adorn them with suitable path stereotypes (such as global and local) to explicitly specify how these objects are related to one another

• Starting with the message that initiates this interaction, attach each subsequent message to the appropriate link, setting its sequence number, as appropriate Show nesting by using Dewey decimal numbering.

• If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.

• If you need to specify this flow of control more formally, attach pre- and post-conditions to each message The Figure shows a collaboration diagram that specifies the flow of control involved in registering a new student at a school, with an emphasis on the structural relationships among these objects

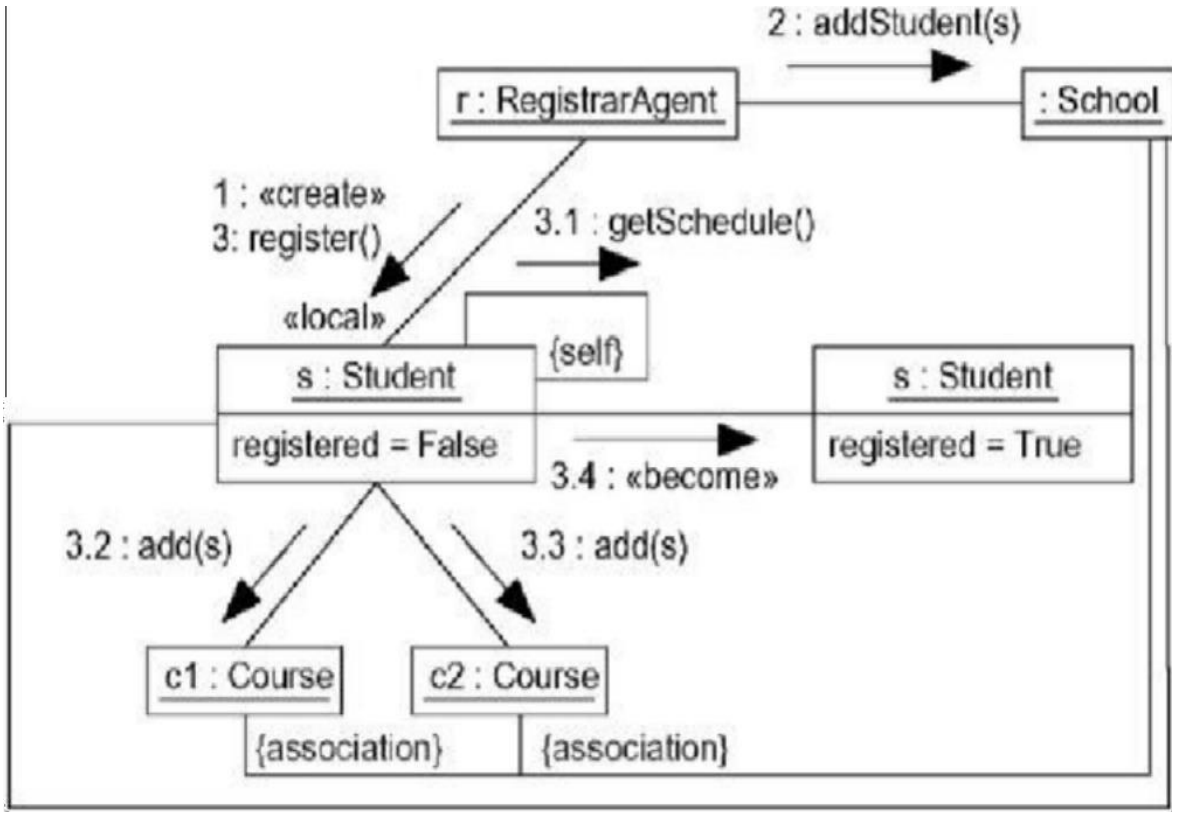


Figure 5: Modeling Flows of Control by Organization

Sequence diagram emphasizes on time sequence of messages and collaboration diagram emphasizes on the structural organization of the objects that send and receive messages.

So the purposes of interaction diagram can be describes as:

- To capture dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe structural organization of the objects.
- To describe interaction among objects.

Use Cases

• use case is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor

• use case captures the intended behavior of the system (or subsystem, class, or interface) without having to specify how that behavior is implemented

• a use case is represented as an ellipse

• Every use case must have a name that distinguishes it from other use cases: simple name and path name Figure 2: Simple and Path Names

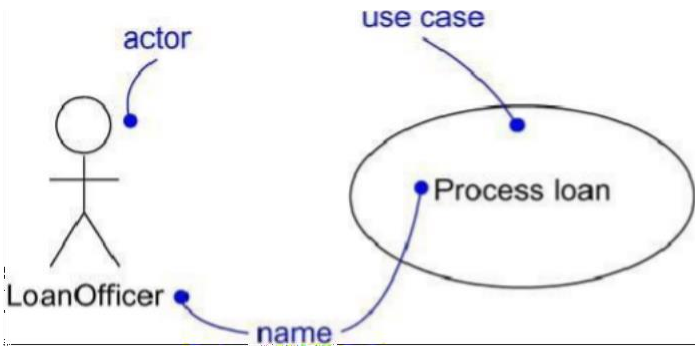


Figure 1: Actors and Use Cases

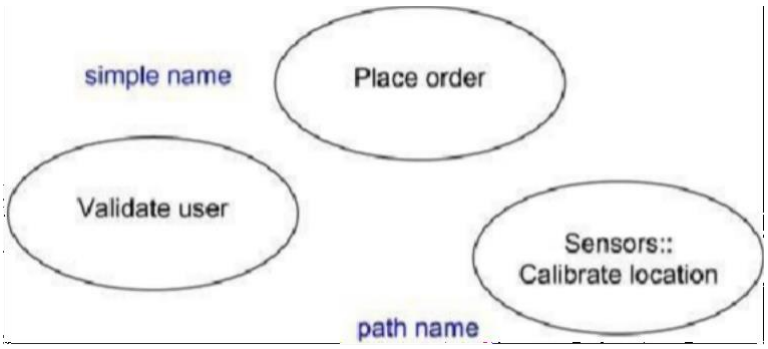


Figure 2: Simple and Path Names

Actors

• actor represents a coherent set of roles that users of use cases play when interacting with these use cases

• an actor represents a role that a human, a hardware device, or even another system plays with a system Figure 3: Actors

Actors may be connected to use cases by association

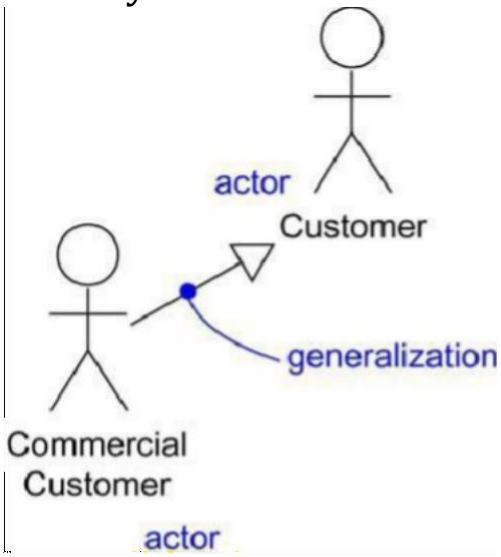


Figure 3: Actors

Use Cases & Flow of Events

flow of events include how and when the use case starts and ends when the use case interacts with the actors and what objects are exchanged, and the basic flow and alternative flows of the behavior.

The behavior of a use case can be specified by describing a flow of events in text. There can be Main flow of events and one or more Exceptional flow of events.

Use Cases and Scenarios

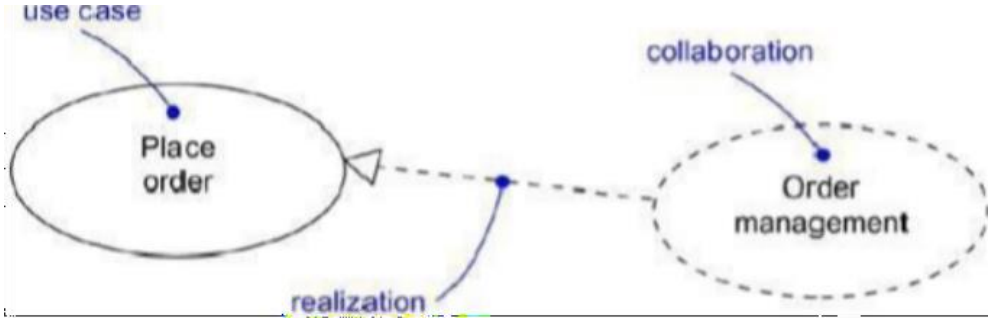
- A scenario is a specific sequence of actions that illustrates behavior
- Scenarios are to use cases, as instances are to classes means that scenario is basically one instance of a use case
- for each use case, there will be primary scenarios and secondary scenarios.

Use Cases and Collaborations

- Collaborations are used to implement the behavior of use cases with society of classes and other elements that work together

- It includes static and dynamic structure

Figure 4: Use Cases and Collaborations



Organizing Use Cases

- organize use cases by grouping them in packages
- organize use cases by specifying generalization, include, and extend relationships among them

Generalization, Include and Extend

An *include relationship* between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base.

An include relationship can be rendered as a dependency, stereotyped as include

An *extend relationship* between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case

An extend relationship can be rendered as a dependency, stereotyped as extend.

extension points are just labels that may appear in the flow of the base use case

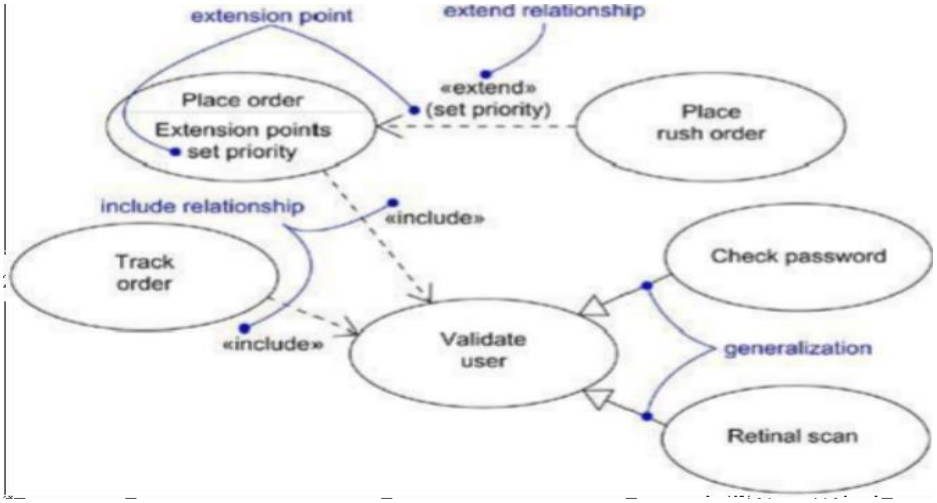


Figure 5: Generalization, Include and Extend

Modeling the Behavior of an Element

To model the behavior of an element,

- Identify the actors that interact with the element Candidate actors include groups that require certain behavior to perform their tasks or that are needed directly or indirectly to perform the element's functions
- Organize actors by identifying general and more specialized roles
- For each actor, consider the primary ways in which that actor interacts with the element Consider also interactions that change the state of the element or its environment or that involve a response to some event
- Consider also the exceptional ways in which each actor interacts with the element
- Organize these behaviors as use cases, applying include and extend

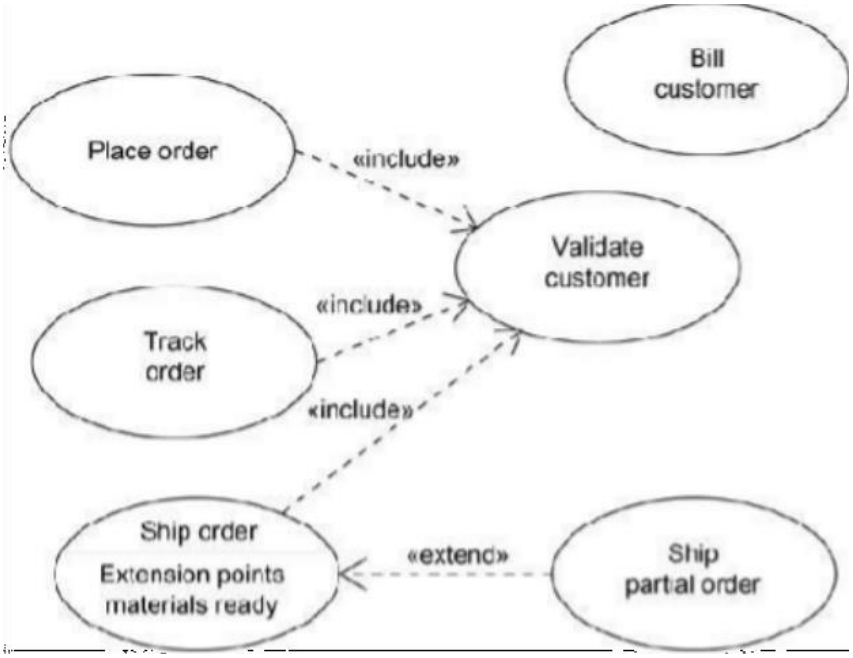


Figure 6: Modeling the Behavior of an Element

Use Case Diagrams

- A use case diagram is a diagram that shows a set of use cases and actors and their relationships

- Use case diagrams commonly contain Use cases, Actors, Dependency, generalization, and association relationships

    - use case diagrams may contain packages, certain times instances of use cases, notes and constraints

    - apply use case diagrams to model the static use case view of a system by modeling the context of a system and by modeling the requirements of a system

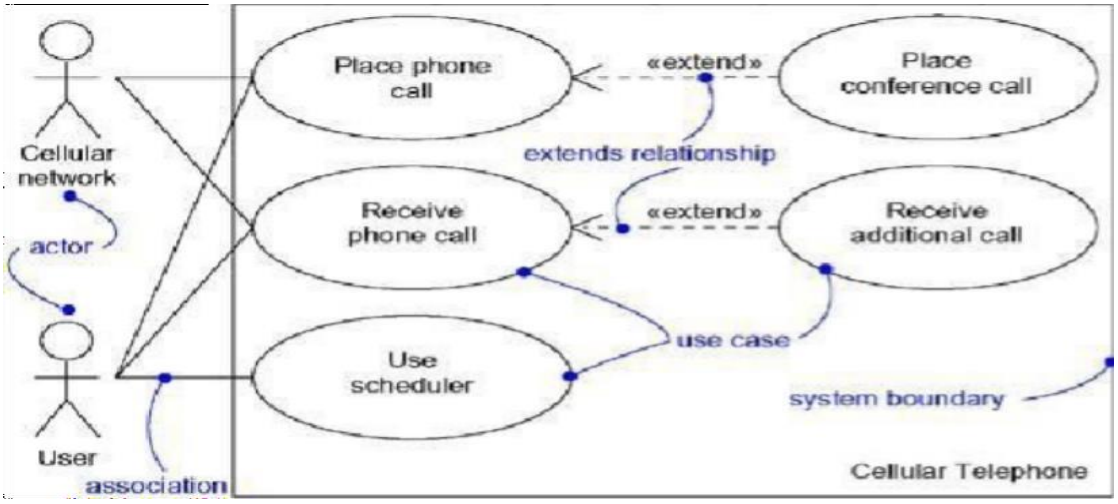


Figure 1: A Use Case Diagram

Modeling the Context of a System

To model the context of a system,

• Identify the actors that surround the system by considering which groups require help from the system to perform their tasks; which groups are needed to execute the system's functions; which groups interact with external hardware or other software systems; and which groups perform secondary functions for administration and maintenance.

- Organize actors that are similar to one another in a generalization / specialization hierarchy
- provide a stereotype for each such actor
- Populate a use case diagram with these actors and specify the paths of communication from each actor to the system's use cases

Figure 2 shows the context of a credit card validation system, with an emphasis on the actors that surround the system

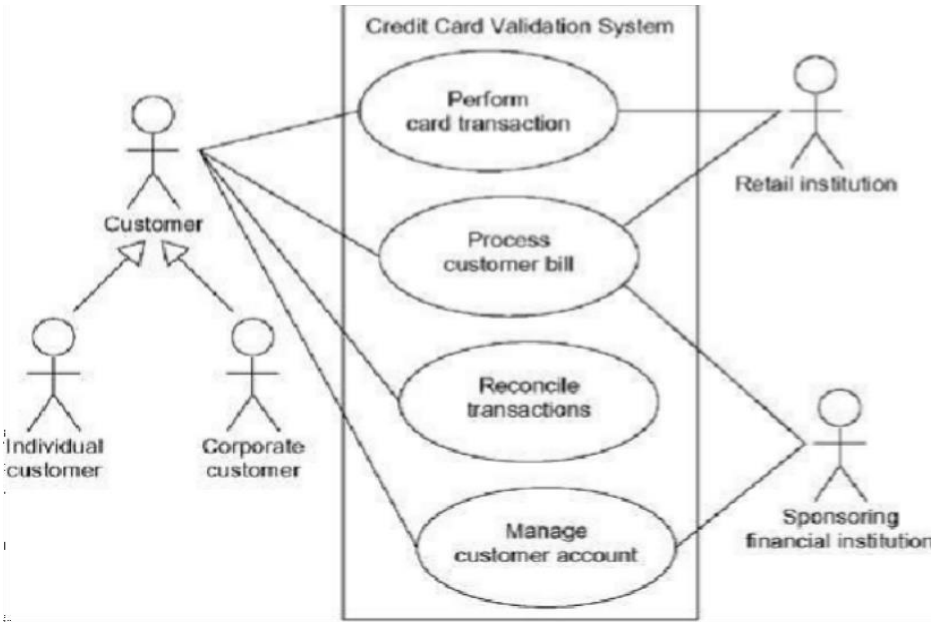


Figure 2: Modeling the Context of a System

Common Modeling Techniques

Modeling the Requirements of a System To model the requirements of a system,

- Establish the context of the system by identifying the actors that surround it
- For each actor, consider the behavior that each expects or requires the system to provide
- Name these common behaviors as use cases
- Factor common behavior into new use cases that are used by others; factor variant behavior into new use cases that extend more main line flows
- Model these use cases, actors, and their relationships in a use case diagram
- Adorn these use cases with notes that assert nonfunctional requirements; you may have to attach some of these to the whole system

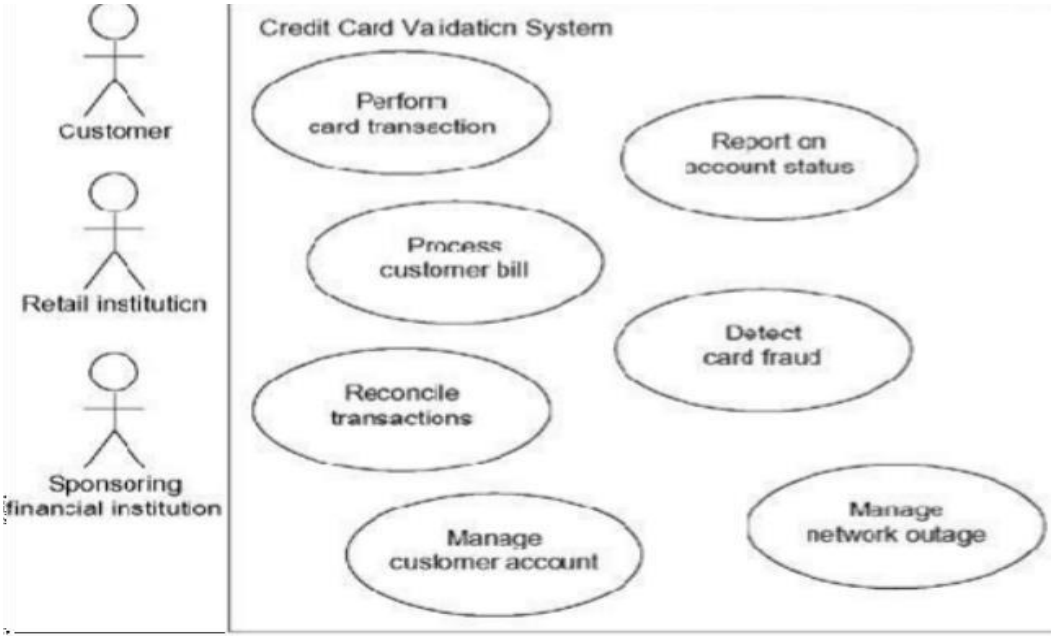


Figure 3: Modeling the Requirements of a System

Activity Diagrams

- An activity diagram shows the flow from activity to activity
- an activity diagram shows the flow of an object, how its role, state and attribute values changes
- activity diagrams is used to model the dynamic aspects of a system
- Activities result in some action (Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation, such as evaluating an expression)
- an activity diagram is a collection of vertices and arcs
- Activity diagrams commonly contain Activity states and action states, Transitions, Objects
- activity diagrams may contain simple and composite states, branches, forks, and joins
- the initial state is represented as a solid ball and stop state as a solid ball inside a circle

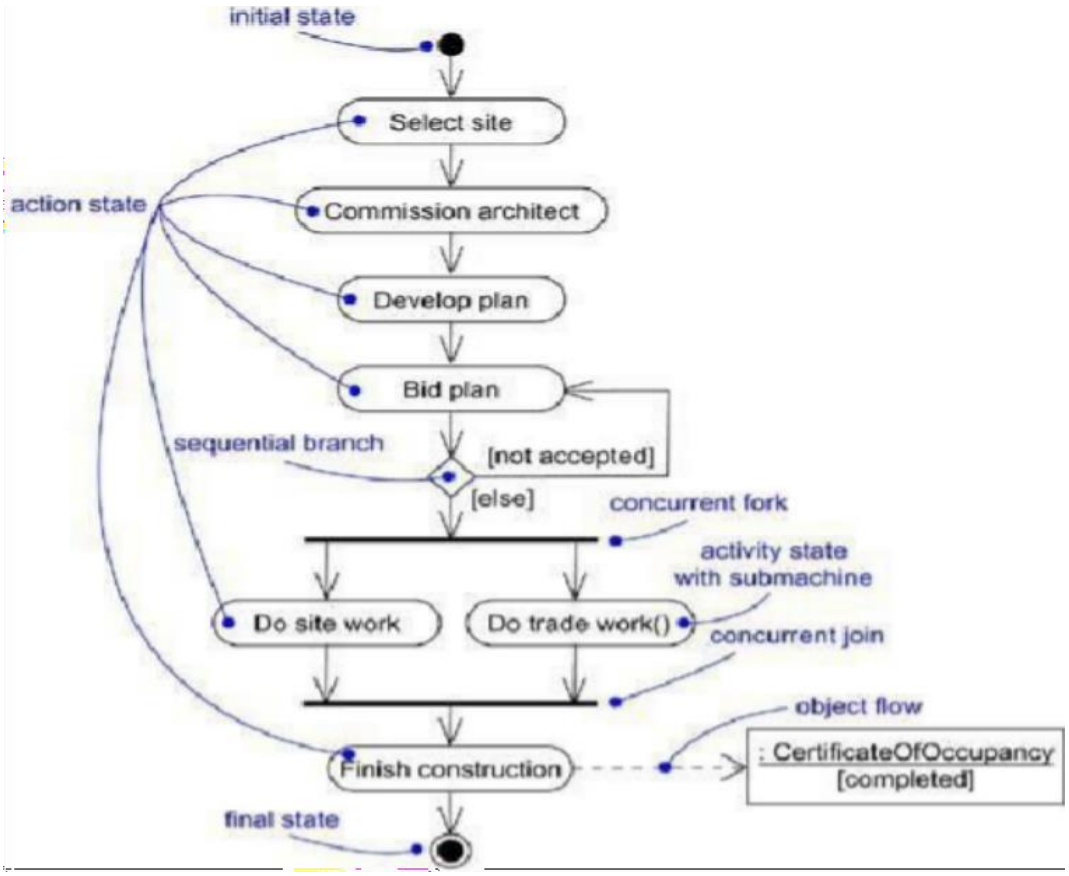


Figure 1: activity diagram

Action States

• The executable, atomic computations are called action states because they are states of the system, each representing the execution of an action

• Figure 2 Action States

• action states can't be decomposed

• action states are atomic, meaning that events may occur, but the work of the action state is not interrupted

• action state is considered to take insignificant execution time

• action states are special kinds of states in a state machine

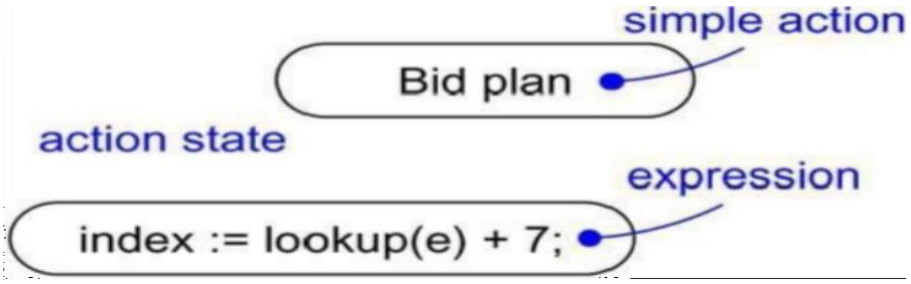


Figure 2: Action States

Activity States

- activity states can be further decomposed
- activity states are not atomic, meaning that they may be interrupted
- they take some duration to complete
- are just special kinds of states in a state machine

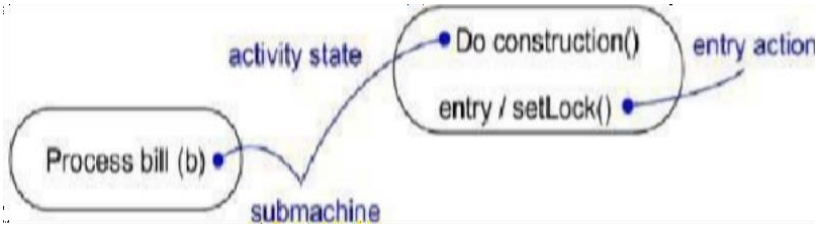


Figure 3: Activity States

Transitions

- transitions shows the path from one action or activity state to the next action or activity state
- a transition is represented as a simple directed line

Triggerless Transitions

- Triggerless Transitions are transitions where control passes immediately once the work of the source state is done

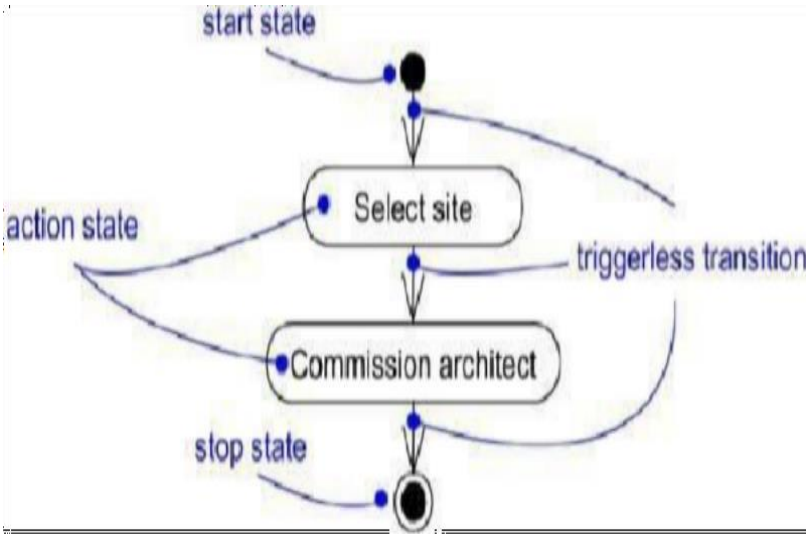


Figure 4: Triggerless Transitions

Branching

- represent a branch as a diamond
- A branch may have one incoming transition and two or more outgoing ones
- each outgoing transition contains a guard expression, which is evaluated only once on entering the branch

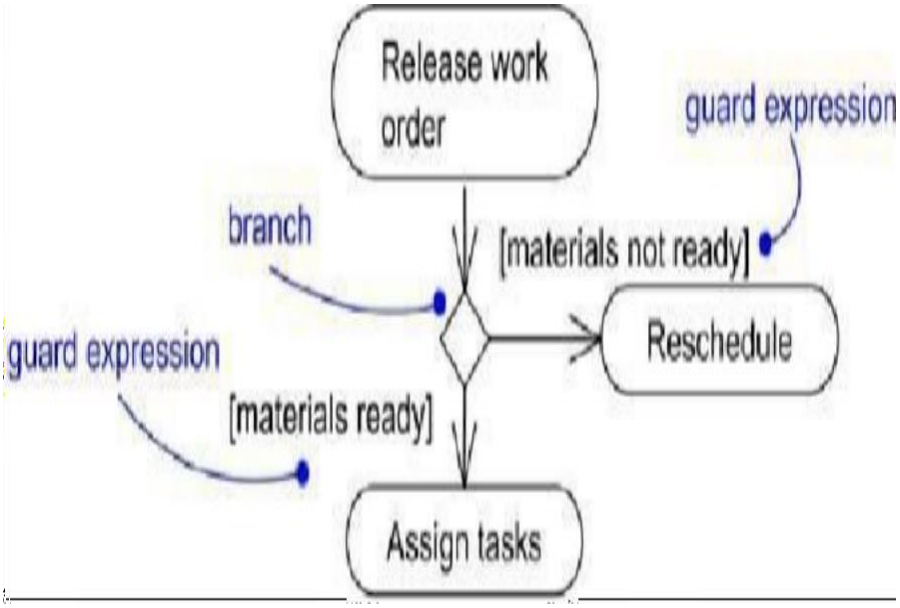


Figure 5: Branching

Forking and Joining

• A *fork* may have one incoming transition and two or more outgoing transitions each of which represents an independent flow of control

• a fork represents the splitting of a single flow of control into two or more concurrent flows of control

• Below the fork, the activities associated with each of these paths continues in parallel

• A *join* may have two or more incoming transitions and one outgoing transition

• Above the join, the activities associated with each of these paths continues in parallel

• At the join, the concurrent flows synchronize, meaning that each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join

• the forking and joining of the parallel flows of control are specified by a *synchronization bar*

• A synchronization bar is rendered as a thick horizontal or vertical line

Joins and forks should balance, meaning that the number of flows that leave a fork should match the number of flows that enter its corresponding join.

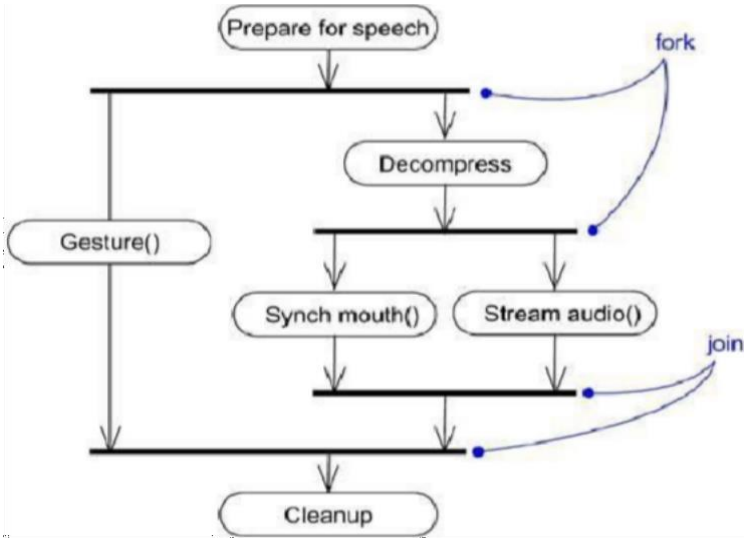


Figure 6: Forking and Joining

Swimlanes

- swimlanes partitions activity diagrams into groups having activity states where each group represents the business organization responsible for those activities
- Each swimlane has a name unique within its diagram
- swimlane represents a high-level responsibility for part of the overall activity of an activity diagram
- each swimlane is implemented by one or more classes.

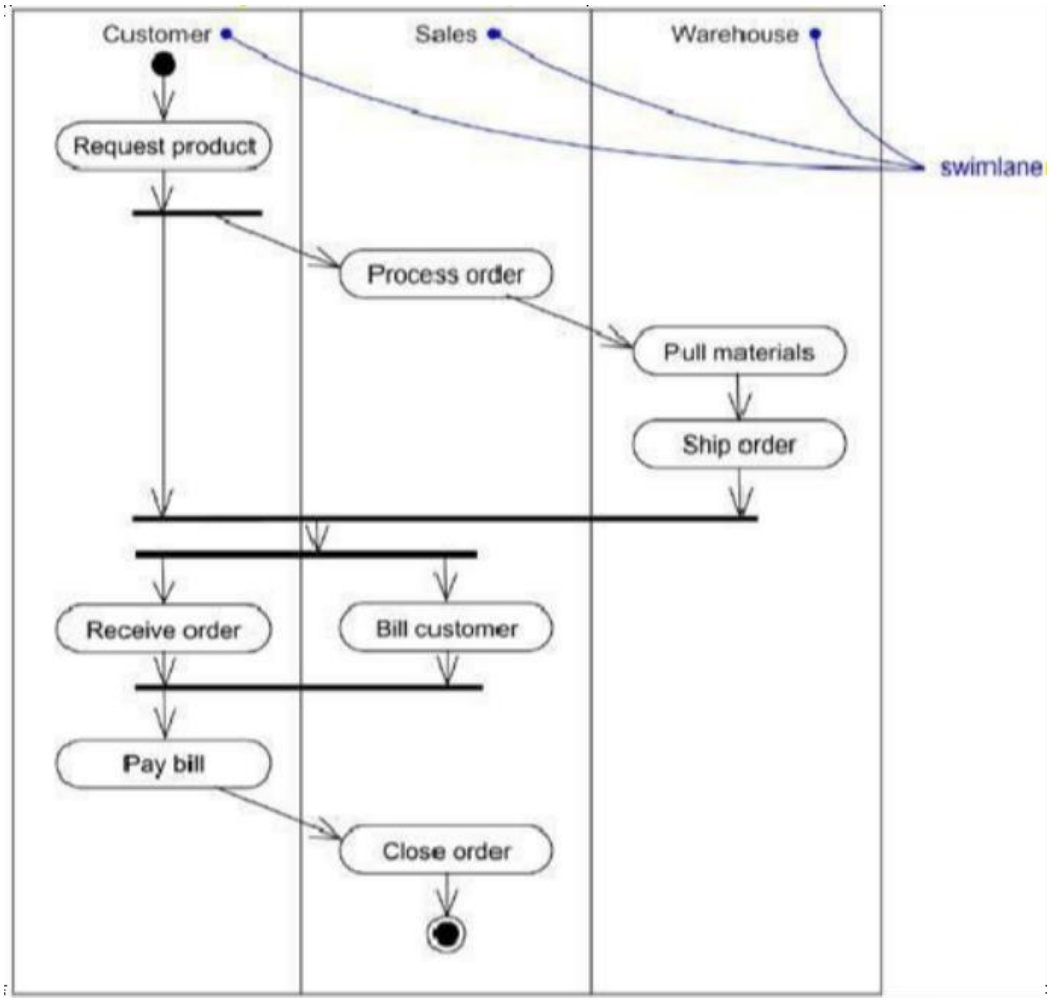


Figure 7: Swimlanes

Object Flow

• object flow indicates the participation of an object in a flow of control, it is represented with the help of dependency relationships.

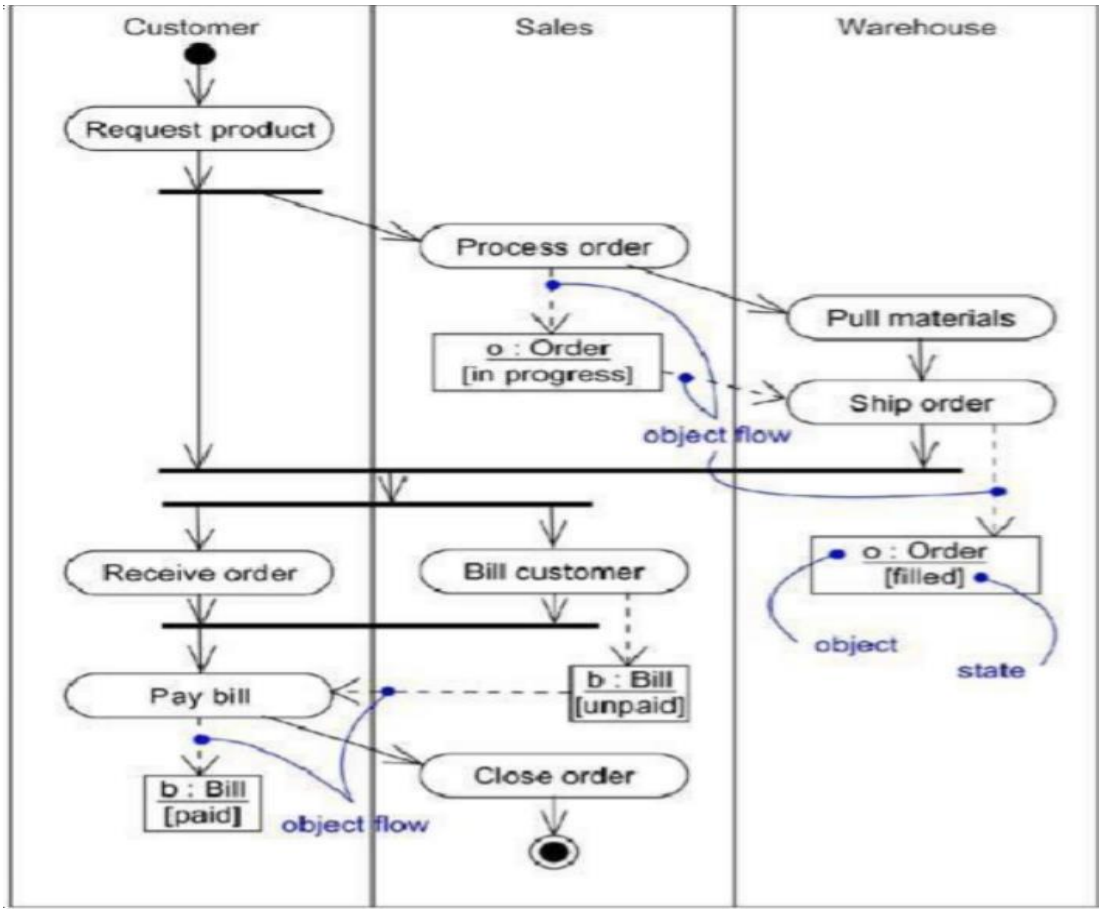


Figure 8: Object Flow

Common Modeling Techniques

Modeling a Workflow

To model a workflow,

• Establish a focus for the workflow. For nontrivial systems, it's impossible to show all interesting workflows in one diagram.

• Select the business objects that have the high-level responsibilities for parts of the overall workflow. These may be real things from the vocabulary of the system, or they may be more abstract. In either case, create a swimlane for each important business object.

• Identify the preconditions of the workflow's initial state and the postconditions of the

workflow's final state. This is important in helping you model the boundaries of the workflow.

•Beginning at the workflow's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.

- For complicated actions, or for sets of actions that appear multiple times, collapse these into activity states, and provide a separate activity diagram that expands on each.
- Render the transitions that connect these activity and action states. Start with the sequential flows in the workflow first, next consider branching, and only then consider forking and joining.
- If there are important objects that are involved in the workflow, render them in the activity diagram, as well. Show their changing values and state as necessary to communicate the intent of the object flow.

For example, Figure shows an activity diagram for a retail business, which specifies the workflow involved when a customer returns an item from a mail order.

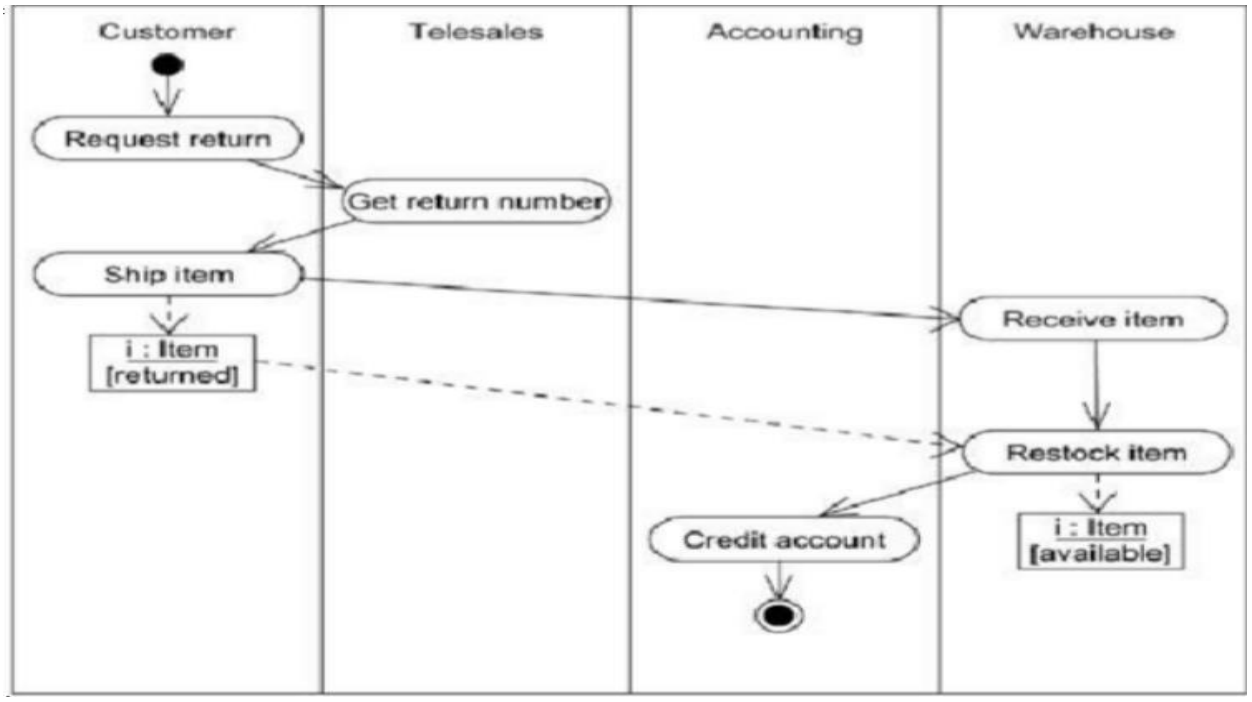


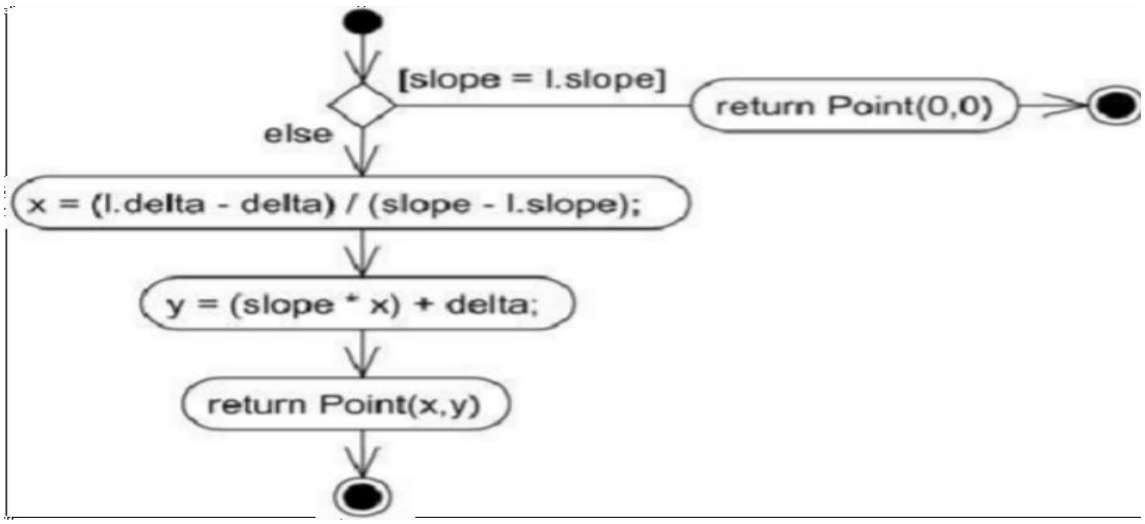
Figure : Modeling a Workflow

Modeling an Operation

To model an operation,

- Collect the abstractions that are involved in this operation. This includes the operation's parameters (including its return type, if any), the attributes of the enclosing class, and certain neighboring classes.
- Identify the preconditions at the operation's initial state and the post conditions at the operation's final state. Also identify any invariants of the enclosing class that must hold during the execution of the operation.
- Beginning at the operation's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
- Use branching as necessary to specify conditional paths and iteration.
- Only if this operation is owned by an active class, use forking and joining as necessary to specify parallel flows of control.

Figure shows an activity diagram that specifies the algorithm of the operation intersection b/w lines.



UNIT – IV

ADVANCED BEHAVIORAL MODELING: Events and signals, state machines, processes and Threads, time and space, state chart diagrams

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.
- 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.

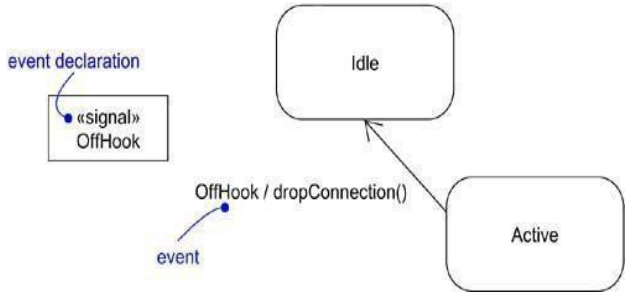


Figure 1: Events

kinds of events

1.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

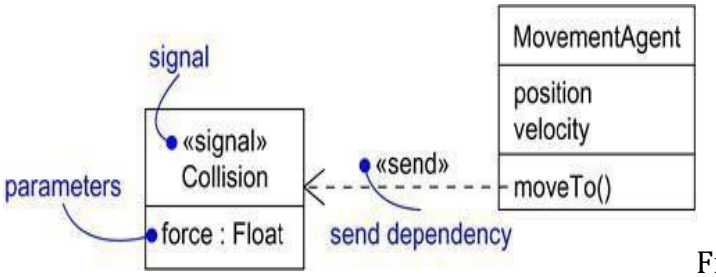


Figure: Signals

2.Call Event

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

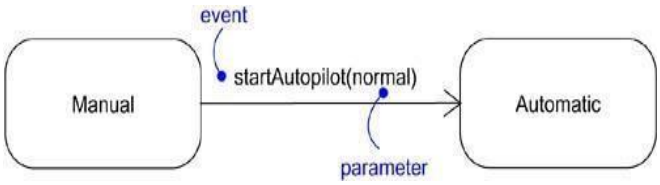


Figure: Call Events

3.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

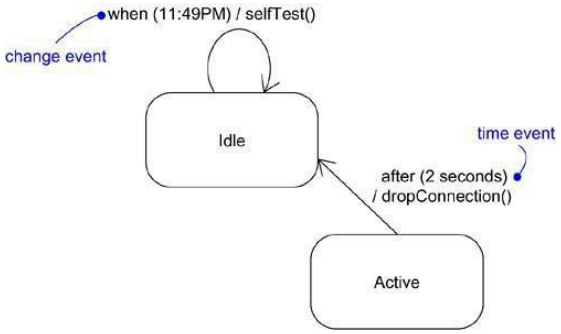


Figure : Time and Change Events

4.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 ie,the 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.

Named signals can be modeled by naming them in an extra compartment of the class

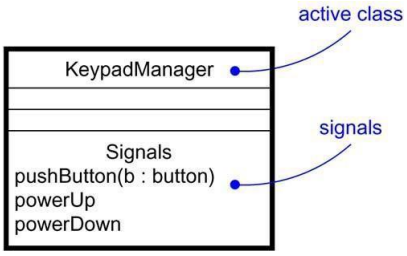


Figure : Signals and Active Classes

Common Modeling Techniques 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.

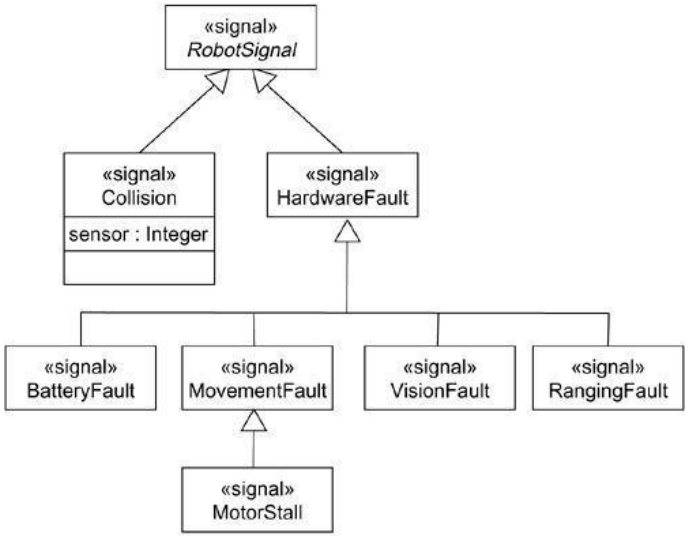
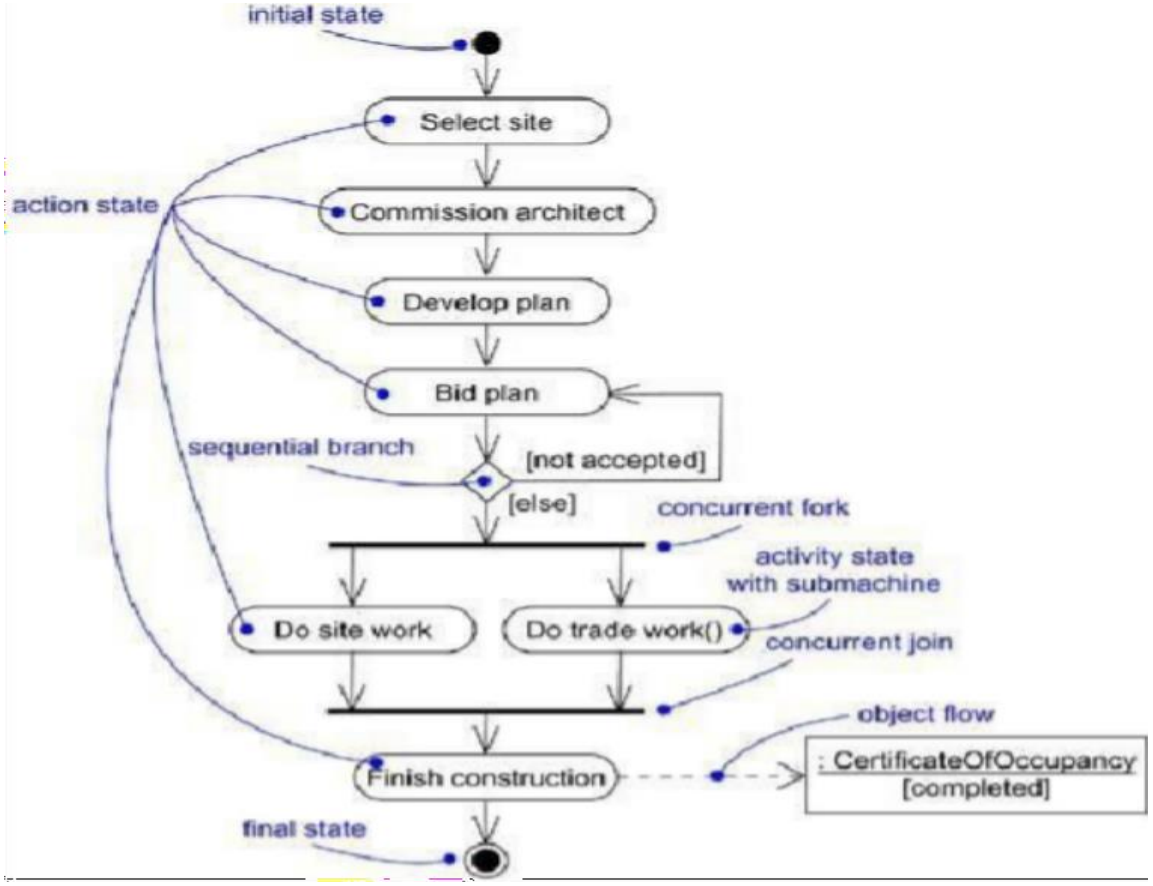


Figure : Modeling Families of Signals

Activity Diagrams

- An activity diagram shows the flow from activity to activity
- An activity diagram shows the flow of an object, how its role, state and attribute values changes.

- activity diagrams is used to model the dynamic aspects of a system
- Activities result in some action (Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation, such as evaluating an expression)
- An activity diagram is a collection of vertices and arcs
- Activity diagrams commonly contain Activity states and action states, Transitions, Objects.
- Activity diagrams may contain simple and composite states, branches, forks, and joins
  - The initial state is represented as a solid ball and stop state as a solid ball inside a circle.



- Figure 1: activity diagram

Action States

- The executable, atomic computations are called action states because they are states of the system, each representing the execution of an action.
- Action states can't be decomposed
- Action states are atomic, meaning that events may occur, but the work of the action state is not interrupted.
- Action state is considered to take insignificant execution time.
- Action states are special kinds of states in a state machine.

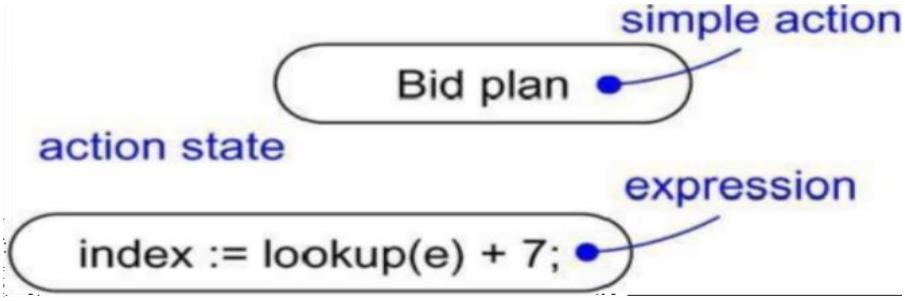


Figure 2: Action States

Activity States

- Activity states can be further decomposed.
- Activity states are not atomic, meaning that they may be interrupted.
- They take some duration to complete are just special kinds of states in a state machine.

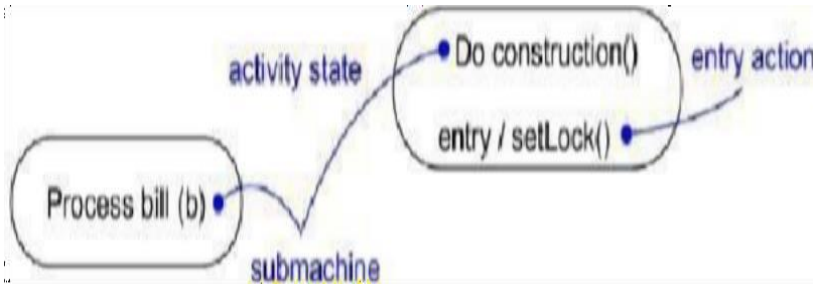


Figure 3: Activity States

Transitions

- Transitions shows the path from one action or activity state to the next action or activity state.
- A transition is represented as a simple directed line.

Triggerless Transitions

- Triggerless Transitions are transitions where control passes immediately once the work of the source state is done

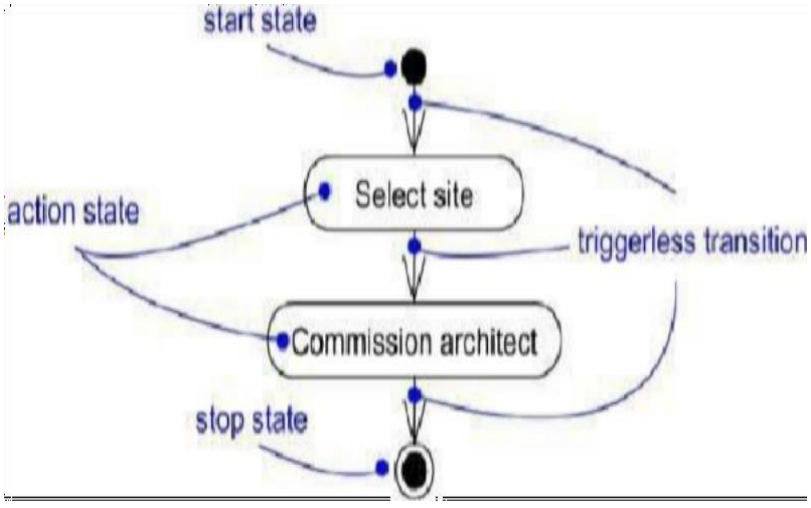


Figure 4: Triggerless Transitions

Branching

- It is represent a branch as a diamond.

- A branch may have one incoming transition and two or more outgoing ones.
- Each outgoing transition contains a guard expression, which is evaluated only once on entering the branch

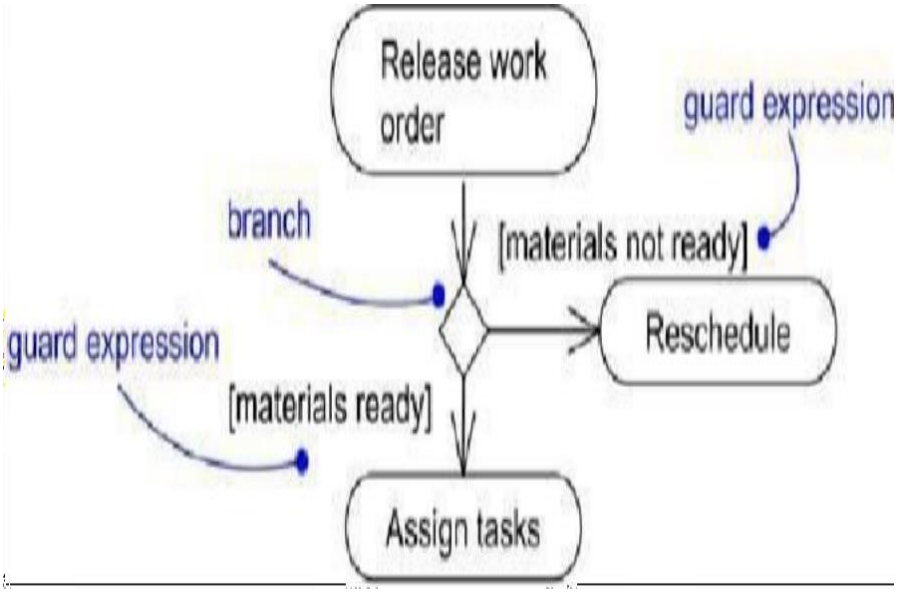


Figure 5: Branching

Forking and Joining

- A fork may have one incoming transition and two or more outgoing transitions each of which represents an independent flow of control.
- A fork represents the splitting of a single flow of control into two or more concurrent flows of control.
- Below the fork, the activities associated with each of these paths continues in parallel
- A join may have two or more incoming transitions and one outgoing transition
- Above the join, the activities associated with each of these paths continues in parallel
- At the join, the concurrent flows synchronize, meaning that each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join
- The forking and joining of the parallel flows of control are specified by a synchronization bar.
- A synchronization bar is rendered as a thick horizontal or vertical line
- Joins and forks should balance, meaning that the number of flows that leave a fork should match the number of flows that enter its corresponding join.

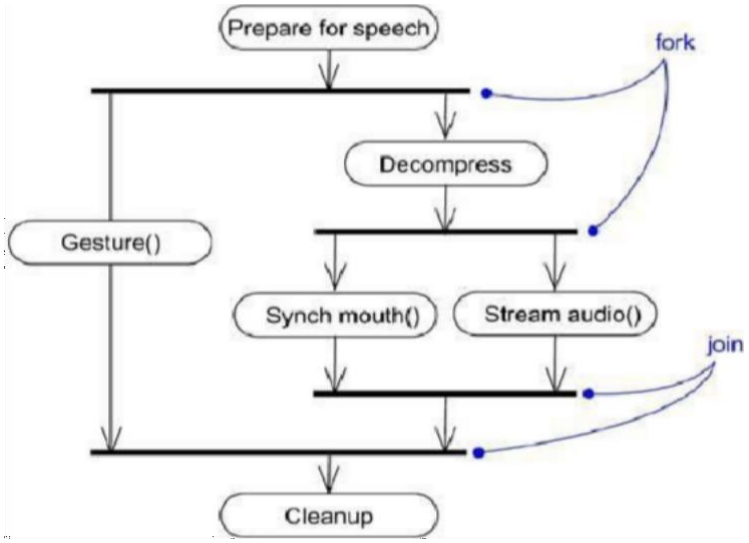
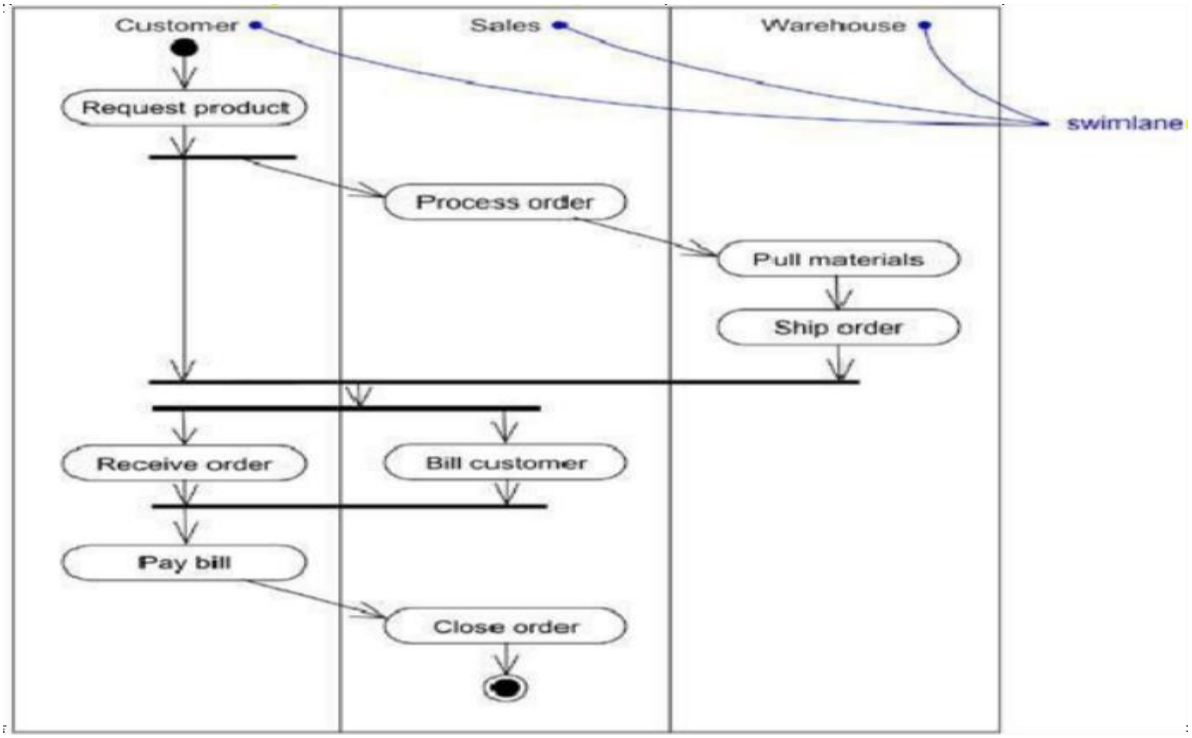


Figure 6: Forking and Joining

Swim lanes

- Swim lanes partitions activity diagrams into groups having activity states where each group represents the business organization responsible for those activities
- Each Swim lane has a name unique within its diagram
- Swim lane represents a high-level responsibility for part of the overall activity of an activity diagram.



- Each Swim lane is implemented by one or more classes.

Figure 7: Swim lanes

Object Flow

- Object flow indicates the participation of an object in a flow of control, it is represented with the

help of dependency relationships.

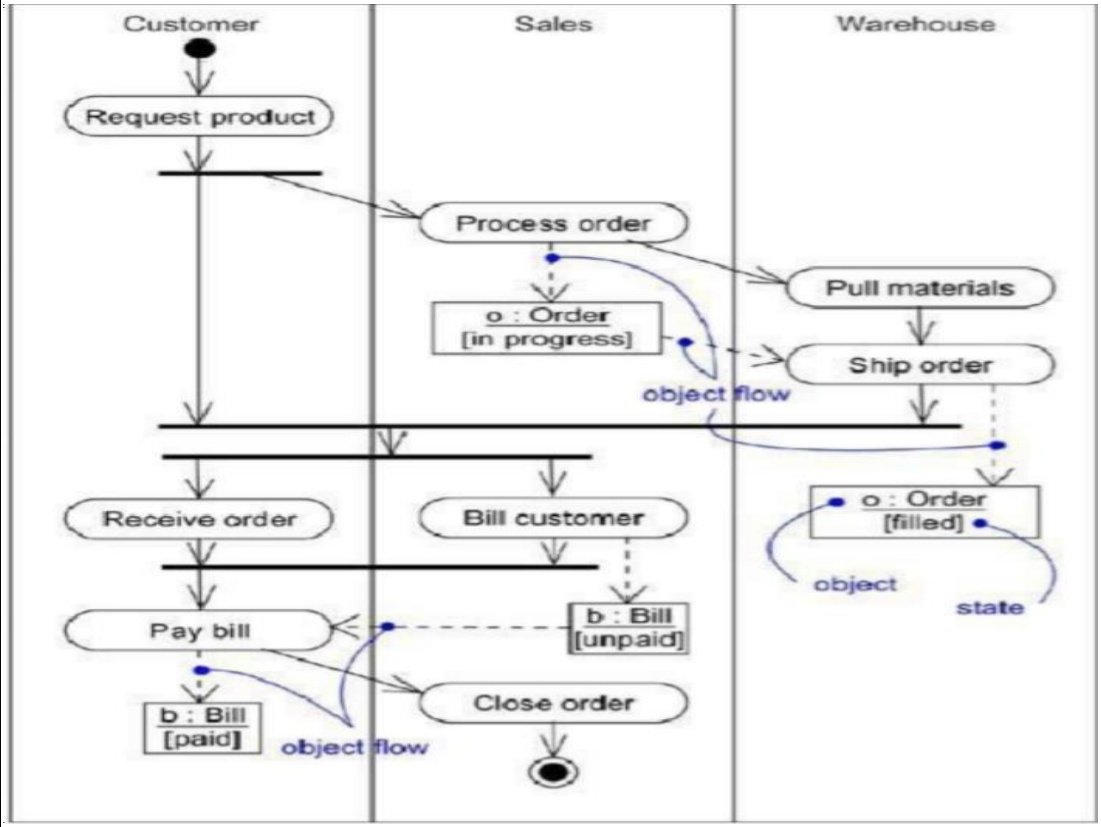


Figure 8: Object Flow

Common Modeling Techniques :-

To model a workflow:-

- Establish a focus for the workflow. For nontrivial systems, it's impossible to show all interesting workflows in one diagram.
- Select the business objects that have the high-level responsibilities for parts of the overall workflow. These may be real things from the vocabulary of the system, or they may be more abstract. In either case, create a Swim lane for each important business object.
- Identify the preconditions of the workflow's initial state and the post conditions of the work flow's final state. This is important in helping you model the boundaries of the workflow.
- Beginning at the workflow's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
- For complicated actions, or for sets of actions that appear multiple times, collapse these into activity states, and provide a separate activity diagram that expands on each.
- Render the transitions that connect these activity and action states. Start with the sequential flows in the workflow first, next consider branching, and only then consider forking and joining.
- If there are important objects that are involved in the workflow, render them in the activity diagram, as well. Show their changing values and state as necessary to communicate the intent of the object flow.
  - For example, Figure shows an activity diagram for a retail business, which specifies the workflow involved when a customer returns an item from a mail order.

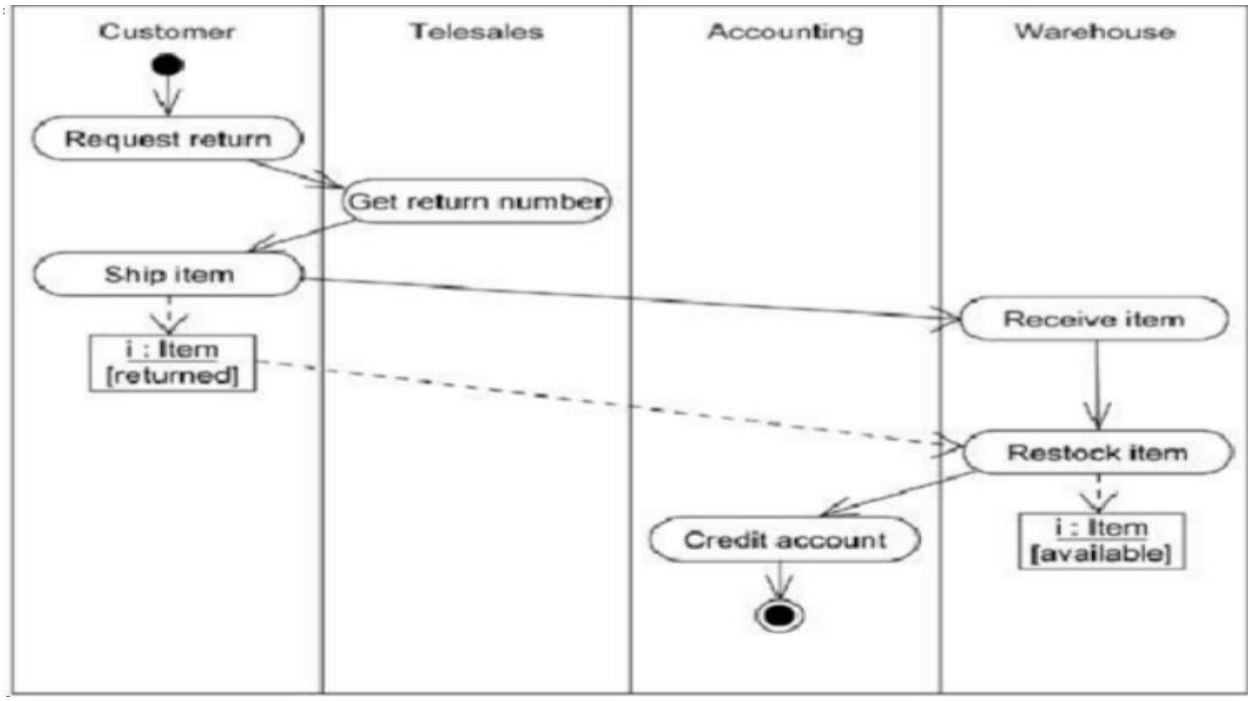


Figure : Modeling a Workflow

Modeling an Operation

To model an operation.

- Collect the abstractions that are involved in this operation. This includes the operation's parameters (including its return type, if any), the attributes of the enclosing class, and certain neighboring classes.
- Identify the preconditions at the operation's initial state and the post conditions at the operation's final state. Also identify any invariants of the enclosing class that must hold during the execution of the operation.
- Beginning at the operation's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
- Use branching as necessary to specify conditional paths and iteration.
- Only if this operation is owned by an active class, use forking and joining as necessary to specify parallel flows of control.

Figure shows an activity diagram that specifies the algorithm of the operation intersection b/w lines.

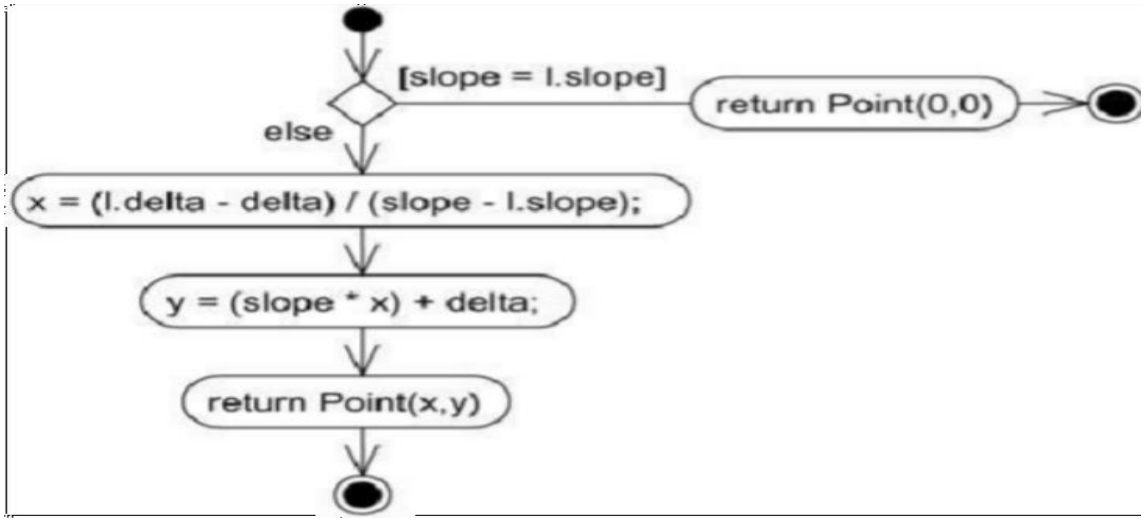


Figure 10: Modeling an Operation

Events and Signals

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.

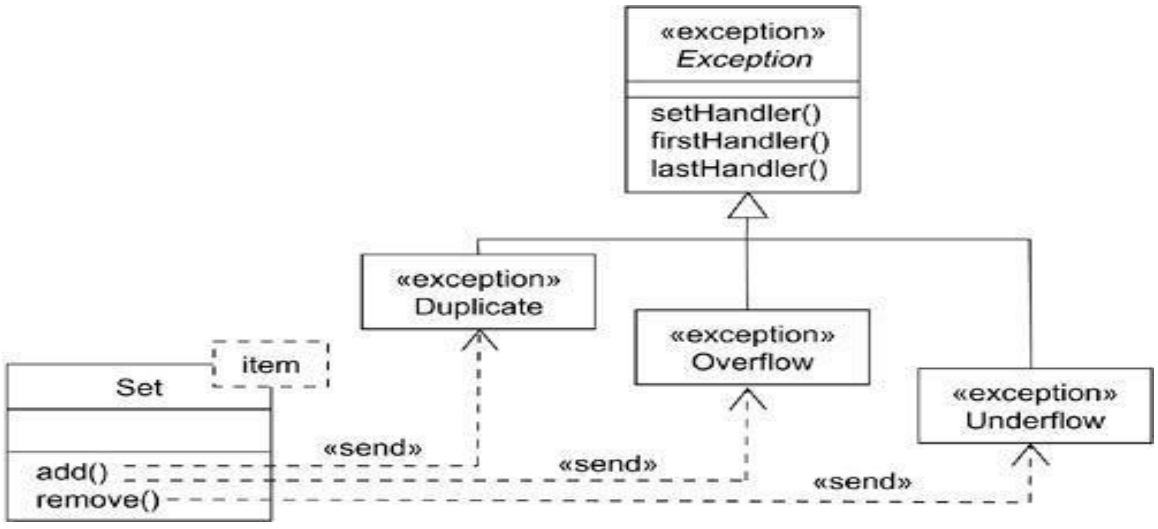


Figure : Modeling Exceptions

State Machine

State machine

- A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events.
- Graphically, a state is rendered as a rectangle with rounded corners. A transition is rendered as a solid directed line.
- State machines are used to specify the behavior of objects that must respond to asynchronous stimulus or whose current behavior depends on their past.
- state machines are used to model the behavior of entire systems, especially reactive systems, which must respond to signals from actors outside the system.

States

- A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- An object remains in a state for a finite amount of time. For example, a Heater in a home might be in any of four states: Idle, Activating, Active, and Shutting Down.
- a state name must be unique within its enclosing state
- A state has five parts: Name, Entry/exit actions, Internal transitions – Transitions that are handled without causing a change in state,
- Substates – nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) substates,
- Deferred events – A list of events that are not handled in that state but, rather, are postponed and queued for handling by the object in another state
- initial state indicates the default starting place for the state machine or substate and is represented as a filled black circle
- final state indicates that the execution of the state machine or the enclosing state has been completed and is represented as a filled black circle surrounded by an unfilled circle.

- Initial and final states are pseudo-states

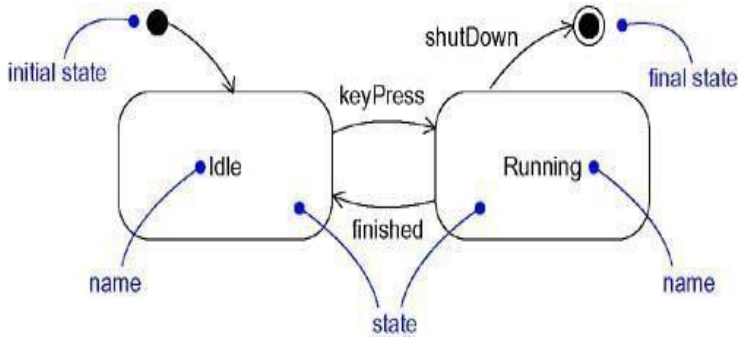


Figure : States

Transitions

- A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- Transition fires means change of state occurs. Until transition fires, the object is in the source state; after it fires, it is said to be in the target state.
- A transition has five parts:

Source state – The state affected by the transition,

Event trigger – a stimulus that can trigger a source state to fire on satisfying guard condition,

Guard condition – Boolean expression that is evaluated when the transition is triggered by the reception of the event trigger,

Action – An executable atomic computation that may directly act on the object that owns the state machine, and indirectly on other objects that are visible to the object,

Target state – The state that is active after the completion of the transition.

- A transition may have multiple sources as well as multiple targets
- A self-transition is a transition whose source and target states are the same

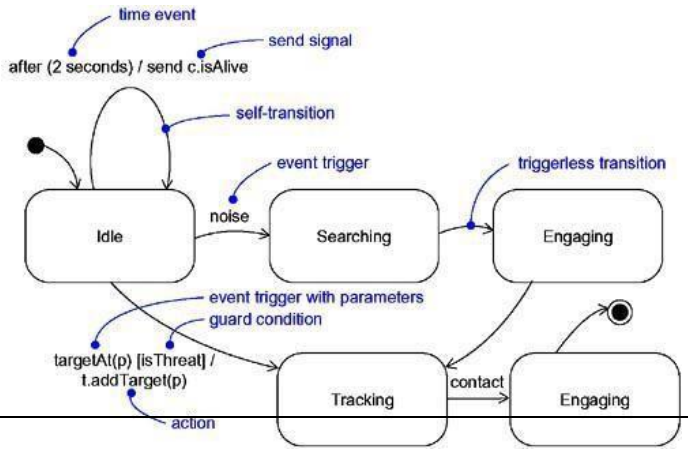


Figure : Transitions

Event Trigger

- An event in the context of state machines is an occurrence of a stimulus that can trigger a state transition.
- events may include signals, calls, the passing of time, or a change in state.
- An event – signal or a call – may have parameters whose values are available to the transition, including expressions for the guard condition and action.
- An event trigger may be polymorphic

Guard condition

- a guard condition is rendered as a Boolean expression enclosed in square brackets and placed after the trigger event
- A guard condition is evaluated only after the trigger event for its transition occurs
- A guard condition is evaluated just once for each transition at the time the event occurs, but it may be evaluated again if the transition is retriggered

Action

- An action is an executable atomic computation i.e, it cannot be interrupted by an event and runs to completion.
- Actions may include operation calls, the creation or destruction of another object, or the sending of a signal to an object

An activity may be interrupted by other events.

Advanced States and Transitions

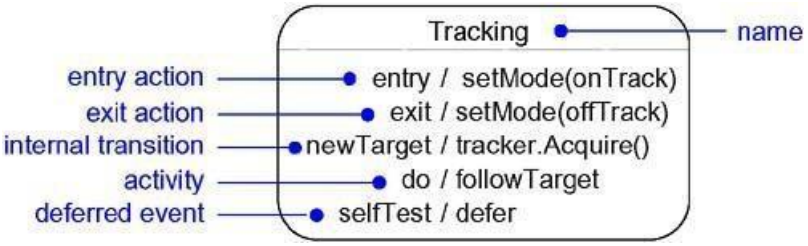


Figure : Advanced States and Transitions

Entry and Exit Actions

- Entry Actions are those actions that are to be done upon entry of a state and are shown by the keyword event 'entry' with an appropriate action
- Exit Actions are those actions that are to be done upon exit from a state marked by the keyword event 'exit', together with an appropriate action

Internal Transitions

- Internal Transitions are events that should be handled internally without leaving the state.
- Internal transitions may have events with parameters and guard conditions.

Activities

Activities make use of object's idle time when inside a state. 'do' transition is used to specify the work that's to be done inside a state after the entry action is dispatched.

Deferred Events

A deferred event is a list of events whose occurrence in the state is postponed until a state in which the listed events are not deferred becomes active, at which time they occur and may trigger transitions as if they had just occurred. A deferred event is specified by listing the event with the special action 'defer'.

Substates

- A Sub state is a state that's nested inside another one.
- A state that has Sub states is called a composite state.
- A composite state may contain either concurrent (orthogonal) or sequential (disjoint) sub states.
- Sub states may be nested to any level

Sequential Substates

- Sequential Sub states are those sub states in which an event common to the composite states can easily be exercised by each states inside it at any time
- sequential sub states partition the state space of the composite state into disjoint states
- A nested sequential state machine may have at most one initial state and one final state

History States

- A history state allows composite state that contains sequential sub states to remember the last sub state that was active in it prior to the transition from the composite state.
- a shallow history state is represented as a small circle containing the symbol H
- The first time entry to a composite state doesn't have any history
- the symbol H designates a shallow history, which remembers only the history of the immediate nested state machine.
- the symbol H* designates deep history, which remembers down to the innermost nested state at any depth.
- When only one level of nesting, shallow and deep history states are semantically equivalent.

Concurrent Sub states

- concurrent Sub states specify two or more state machines that execute in parallel in the context of the enclosing object
- Execution of these concurrent Sub states continues in parallel. These Sub states waits for each other to finish to joins back into one flow
- A nested concurrent state machine does not have an initial, final, or history state

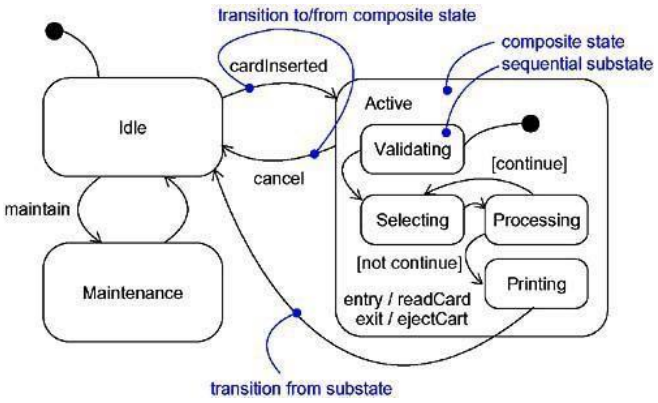


Figure : Sequential Sub tates

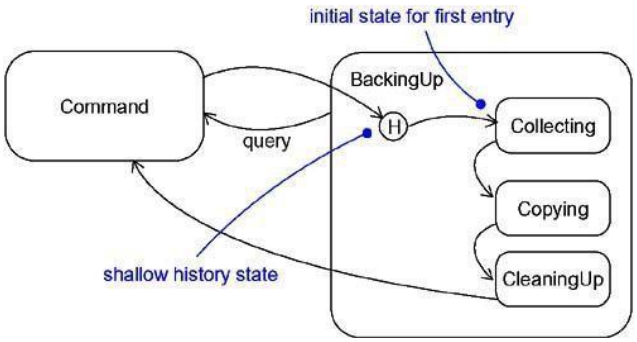


Figure : History State

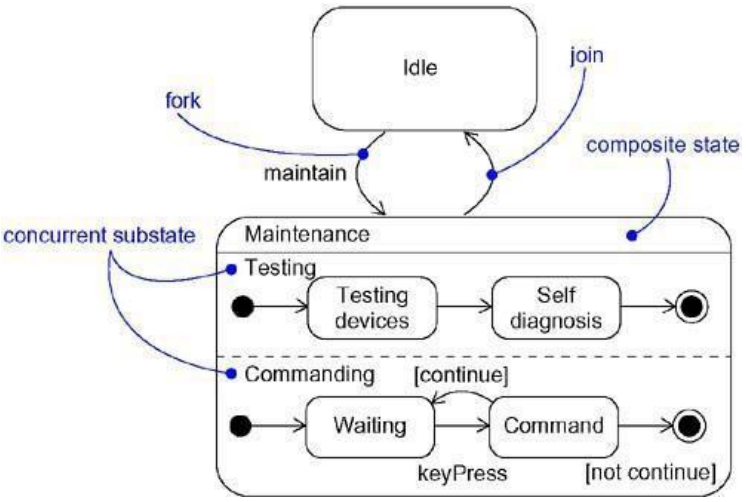


Figure : Concurrent Sub states

Common Modeling Techniques Modeling the Lifetime of an Object To model the lifetime of an object,

- Set the context for the state machine, whether it is a class, a use case, or the system as a whole.
- If the context is a class or a use case, collect the neighboring classes, including any parents of the class and any classes reachable by associations or dependences. These neighbors are candidate targets for actions and are candidates for including in guard conditions.
- If the context is the system as a whole, narrow your focus to one behavior of the system. Theoretically, every object in the system may be a participant in a model of the system's lifetime, and except for the most trivial systems, a complete model would be intractable.
- Establish the initial and final states for the object. To guide the rest of your model,

possibly state the pre- and post conditions of the initial and final states, respectively.

- Decide on the events to which this object may respond. If already specified, you'll find these in the object's interfaces; if not already specified, you'll have to consider which objects may interact with the object in your context, and then which events they may possibly dispatch.
- Starting from the initial state to the final state, lay out the top-level states the object may be in. Connect these states with transitions triggered by the appropriate events. Continue by adding actions to these transitions.
- Identify any entry or exit actions (especially if you find that the idiom they cover is used in the state machine).
- Expand these states as necessary by using sub states.
- Check that all events mentioned in the state machine match events expected by the interface of the object. Similarly, check that all events expected by the interface of the object are handled by the state machine. Finally, look to places where you explicitly want to ignore events.
- Check that all actions mentioned in the state machine are sustained by the relationships, methods, and operations of the enclosing object.
- Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses. Be especially diligent in looking for unreachable states and states in which the machine may get stuck.
- After rearranging your state machine, check it against expected sequences again to ensure that you have not changed the object's semantics.

For example, Figure shows the state machine for the controller in a home security system

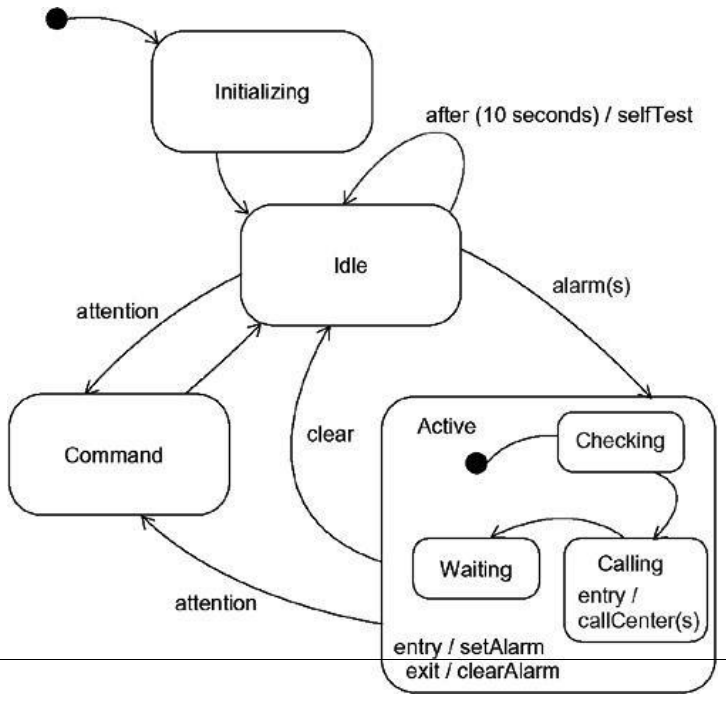


Figure : Modeling the Lifetime of An Object

Processes and Threads

- A process is a heavyweight flow that can execute concurrently with other processes.
- A thread is a lightweight flow that can execute concurrently with other threads within the same process.
- An active object is an object that owns a process or thread and can initiate control activity.
- An active class is a class whose instances are active objects.

- Graphically, an active class is rendered as a rectangle with thick lines. Processes and threads are rendered as stereotyped active classes.

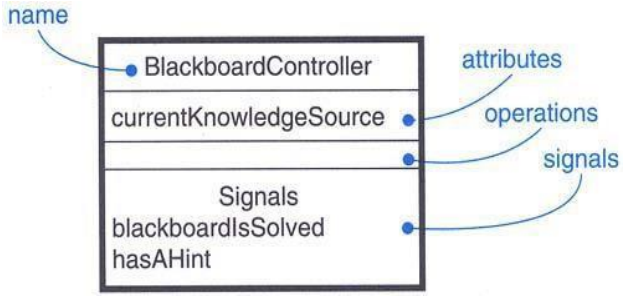


Figure : Active Class

Flow of Control

In a sequential system, there is a single flow of control. i.e, one thing, and one thing only, can take place at a time. In a concurrent system, there is multiple simultaneous flow of control i.e, more than one thing can take place at a time.

Classes and Events

- Active classes are just classes which represents an independent flow of control
- Active classes share the same properties as all other classes.
- When an active object is created, the associated flow of control is started; when the active object is destroyed, the associated flow of control is terminated
- Two standard stereotypes that apply to active classes are, <<**process**>> – Specifies a heavyweight flow that can execute concurrently with other processes. (heavyweight means, a thing known to the OS itself and runs in an independent address space) <<**thread**>> –

Specifies a lightweight flow that can execute concurrently with other threads within the same process (lightweight means, known to the OS itself.)

- All the threads that live in the context of a process are peers of one another

Communication

- In a system with both active and passive objects, there are *four possible combinations of interaction*
- First, a message may be passed from one passive object to another.
- Second, a message may be passed from one active object to another.
- In inter-process communication there are two possible styles of communication. *First*, one active object might synchronously call an operation of another. *Second*, one active object might asynchronously send a signal or call an operation of another object
- a synchronous message is rendered as a full arrow and an asynchronous message is rendered as a half arrow
- Third, a message may be passed from an active object to a passive object
- *Fourth*, a message may be passed from a passive object to an active one

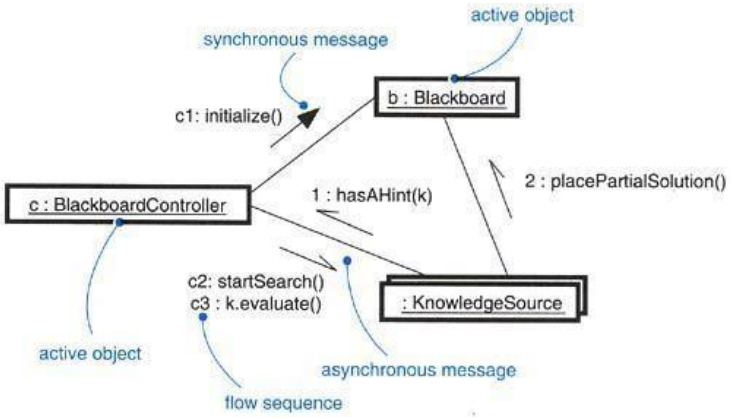


Figure 2: Communication

Synchronization

- Synchronization means arranging the flow of controls of objects so that mutual exclusion will be guaranteed.

- in object-oriented systems these objects are treated as a critical region
- *three approaches* are there to handle synchronization:
- *Sequential* – Callers must coordinate outside the object so that only one flow is in the object at a time
- *Guarded* – multiple flow of control is sequentialized with the help of object's guarded operations. in effect it becomes sequential.
- *Concurrent* – multiple flow of control is guaranteed by treating each operation as atomic
- synchronization are rendered in the operations of active classes with the help of constraints

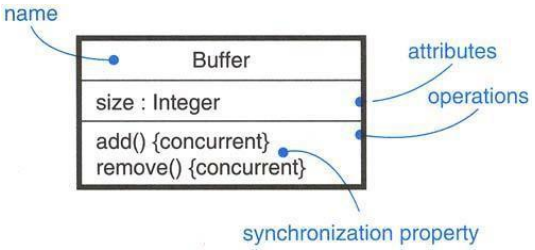


Figure: Synchronization

Process Views

- The process view of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms.
- This view primarily addresses the performance, scalability, and throughput of the system.

Common Modeling

Techniques Modeling

Multiple Flows of Control

To model multiple flows of

control,

- Identify the opportunities for concurrent action and reify each flow as an active class. Generalize common sets of active objects into an active class. Be careful not to over engineer the process view of your system by introducing too much concurrency.
- Consider a balanced distribution of responsibilities among these active classes, then examine the other active and passive classes with which each collaborates statically. Ensure that each active class is both tightly cohesive and loosely coupled relative to these

neighboring classes and that each has the right set of attributes, operations, and signals.

- Capture these static decisions in class diagrams, explicitly highlighting each active class.
- Consider how each group of classes collaborates with one another dynamically. Capture those decisions in interaction diagrams. Explicitly show active objects as the root of such flows. Identify each related sequence by identifying it with the name of the active object.
- Pay close attention to communication among active objects. Apply synchronous and asynchronous messaging, as appropriate.
- Pay close attention to synchronization among these active objects and the passive objects with which they collaborate. Apply sequential, guarded, or concurrent operation semantics, as appropriate.

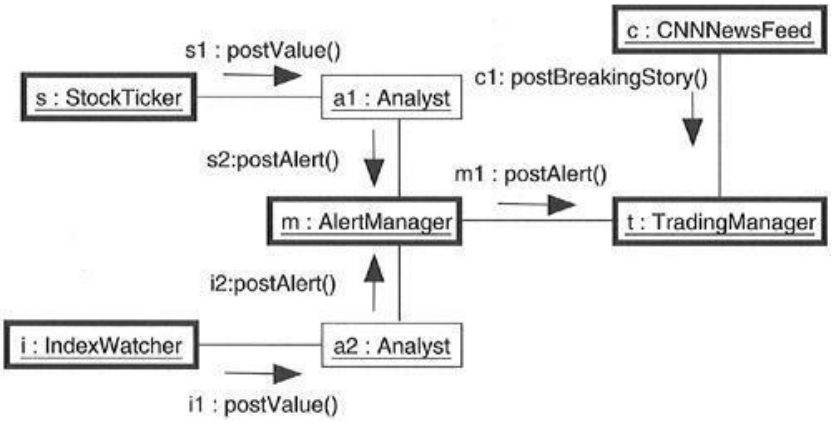


Figure : Modeling Flows of Control

Modeling Inter Process Communication

To model inter process communication,

- Model the multiple flows of control.
- Consider which of these active objects represent processes and which represent threads. Distinguish them using the appropriate stereotype.
- Model messaging using asynchronous communication; model remote procedure calls using synchronous communication.
- Informally specify the underlying mechanism for communication by using notes, or more formally by using collaborations.

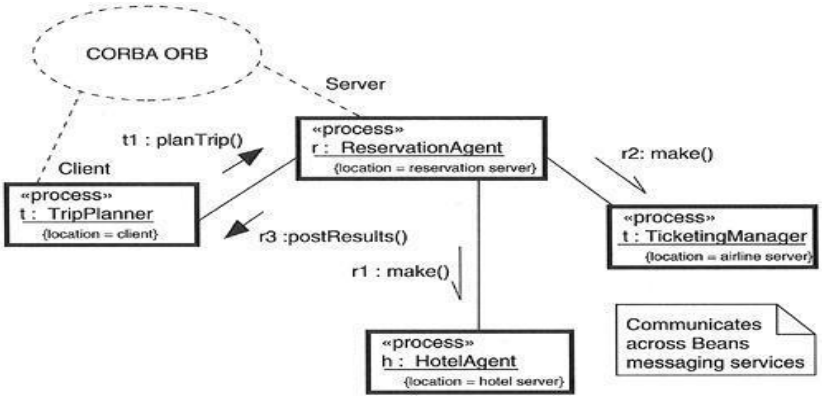


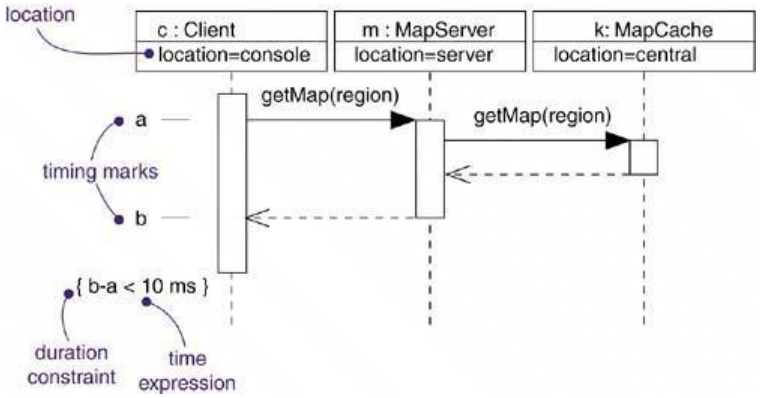
Figure : Modeling Inter process Communication

Time and Space

A distributed system is one in which components may be physically distributed across nodes. These nodes may represent different processors physically located in the same box, or they may even represent computers that are located half a world away from one another.

To represent the modeling needs of real time and distributed systems, the UML provides a graphic representation for timing marks, time expressions, timing constraints, and location.

Figure :Timing Constraints and Location



A timing mark is a denotation for the time at which an event occurs. Graphically, a timing mark is depicted as a small hash mark (horizontal line) on the border of a sequence diagram.

A time expression is an expression that evaluates to an absolute or relative value of time. A time expression can also be formed using the name of a message and an indication of a stage in its processing, for
example, request. Send Time or request. Receive Time.

A timing constraint is a semantic statement about the relative or absolute value of time. Graphically, a timing constraint is rendered as for any constraint-that is, a string enclosed by brackets and generally connected to an element by a dependency relationship.

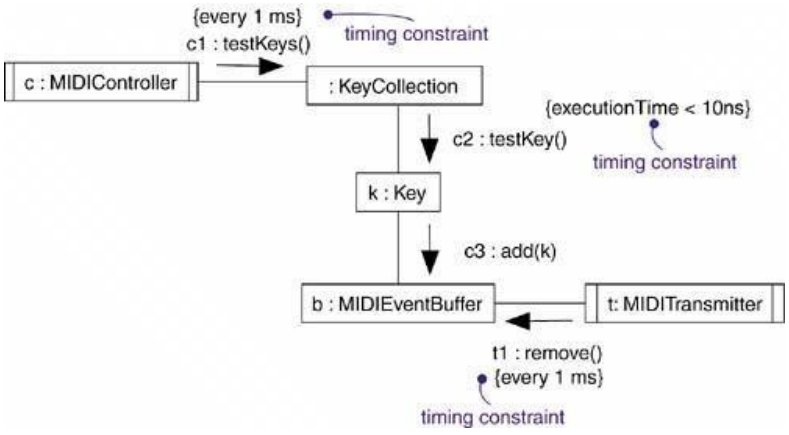
Location is the placement of a component on a node. Location is an attribute of an object.

Time

Real time systems are, time-critical systems. Events may happen at regular or irregular times; the response to an event must happen at predictable absolute times or at predictable times relative to the event itself.

The passing of messages represents the dynamic aspect of any system, They are mainly rendered with the name of an event, such as a signal or a call.

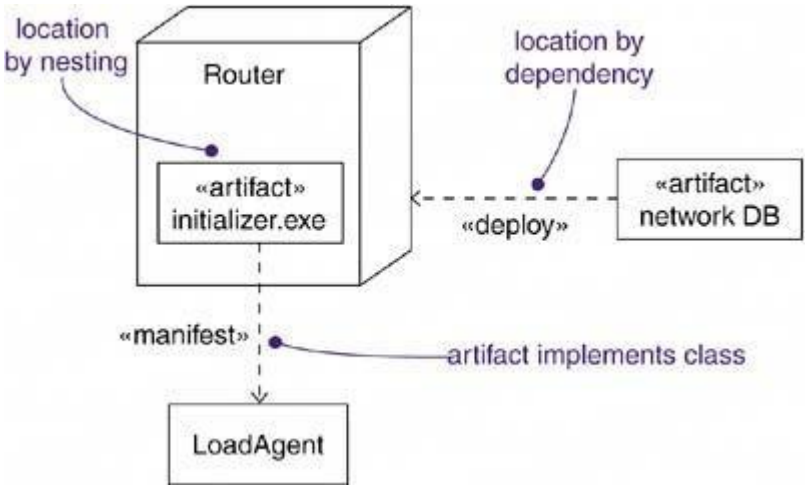
Figure : Time



Location

Distributed systems, encompass components that are physically scattered among the nodes of a system. For many systems, components are fixed in place at the time they are loaded on the system; in other systems, components may migrate from node to node.

Figure : Location



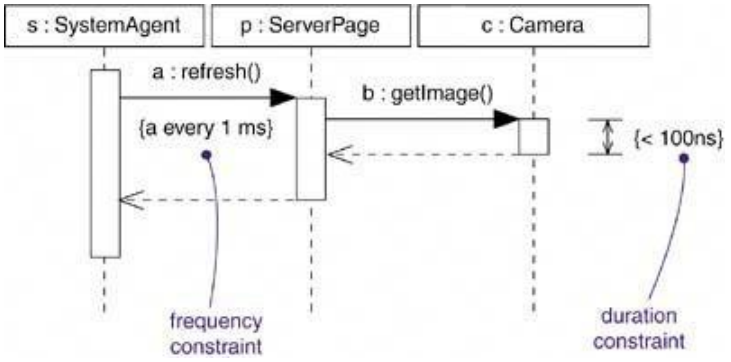
Common Modeling Techniques

Modeling Timing Constraints

To model timing constraints,

- For each event in an interaction, consider whether it must start at some absolute time. Model that real time property as a timing constraint on the message.
- For each interesting sequence of messages in an interaction, consider whether there is an associated maximum relative time for that sequence. Model that real time property as a timing constraint on the sequence.

Figure : Modeling Timing Constraint



Modeling the Distribution of Objects

To model the distribution of objects,

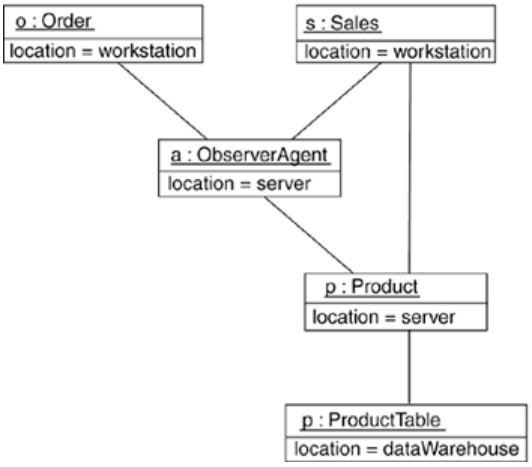
- For each interesting class of objects in your system, consider its locality of reference. In other

words, consider all its neighbours and their locations. A tightly coupled locality will have neighbouring objects close by; a loosely coupled one will have distant objects.

- Tentatively allocate objects closest to the actors that manipulate them.
- Next consider patterns of interaction among related sets of objects.
- Partition sets of objects that have low degrees of interaction.
- Next consider the distribution of responsibilities across the system. Redistribute your objects to balance the load of each node.
- Consider also issues of security, volatility, and quality of service, and redistribute your objects as appropriate.
- Assign objects to components so that tightly coupled objects are on the same component.
- Assign components to nodes so that the computation needs of each node are within capacity. Add additional nodes if necessary.
- Balance performance and communication costs by assigning tightly coupled components to the

same node.

Figure : Modeling the Distribution of Objects



State chart Diagrams

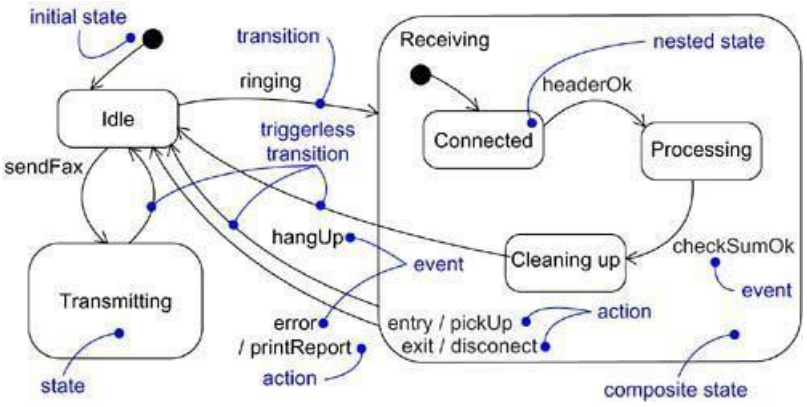


Figure: State chart Diagram

- State chart diagram is simply a presentation of a state machine which shows the flow of control from state to state.
- State chart diagrams are important for constructing executable systems through forward and reverse engineering.
- state chart diagrams are useful in modeling the lifetime of an object
- State chart diagrams commonly contain – Simple states and composite states, Transitions-including events and actions
- It is one of the five diagrams in UML for modeling the dynamic aspects of systems.
- Graphically, a state chart diagram is a collection of vertices and arcs.

A state is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
An event in the context of state machines is an occurrence of a stimulus that can trigger a state transition. A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
An activity is ongoing non atomic execution within a state machine.

An action is an executable atomic computation that results in a change in state of the model or the return of a value.
A reactive or event-driven object is one whose behavior is best characterized by its response to events dispatched from outside its context

Modeling Reactive Objects

To model a reactive object,

- Choose the context for the state machine, whether it is a class, a use case, or the system as a whole.
- Choose the initial and final states for the object. To guide the rest of your model, possibly state the pre- and post conditions of the initial and final states, respectively.
- Decide on the stable states of the object by considering the conditions in which the object may exist for some identifiable period of time. Start with the high-level states of the object and only then consider its possible sub states.
- Decide on the meaningful partial ordering of stable states over the lifetime of the object.

- Decide on the events that may trigger a transition from state to state. Model these events as triggers to transitions that move from one legal ordering of states to another.
- Attach actions to these transitions (as in a Mealy machine) and/or to these states (as in a Moore machine).
- Consider ways to simplify your machine by using sub states, branches, forks, joins, and history states.
- Check that all states are reachable under some combination of events.
- Check that no state is a dead end from which no combination of events will transition the object out of that state.
- Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses.

The first string represents a tag; the second string represents the body of the message.

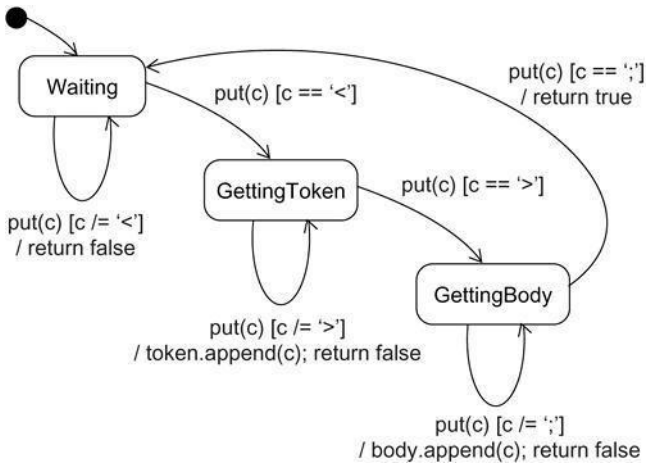


Figure : Modeling Reactive Objects

UNIT – V

ARCHITECTURAL MODELING: Component Diagrams – Terms and Concepts , Deployment Diagrams – Terms and Concepts

Component Diagrams

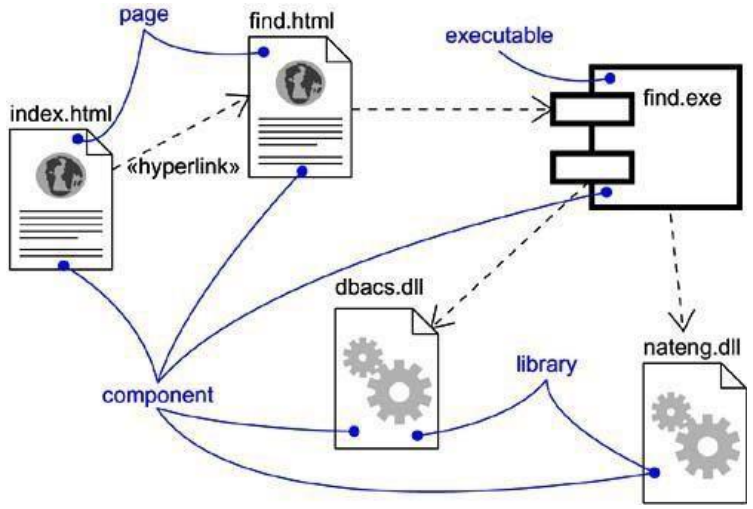


Figure : Component Diagram

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 component-based 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.

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.

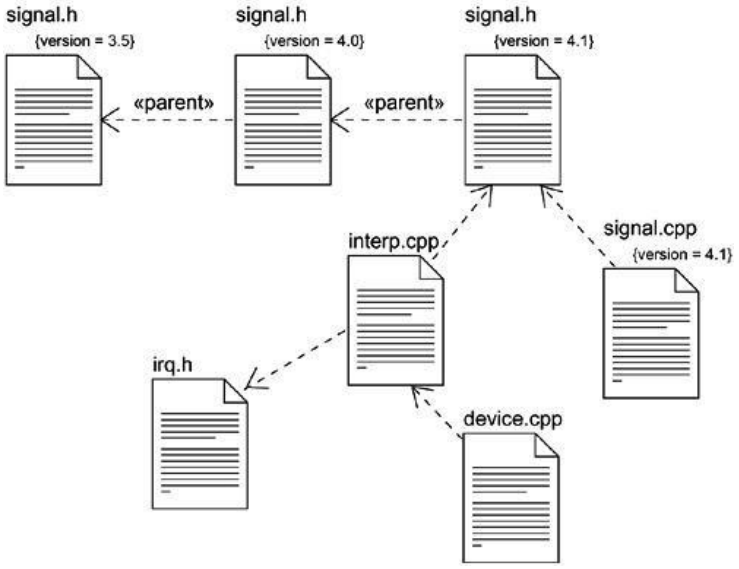


Figure : Modeling Source Code

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.

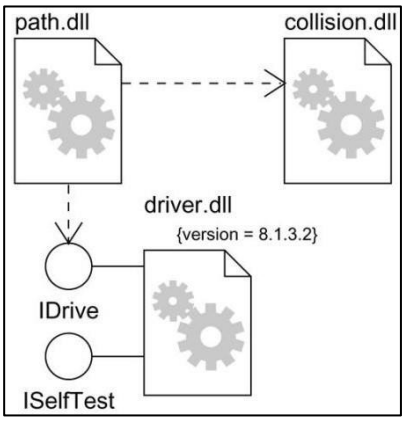
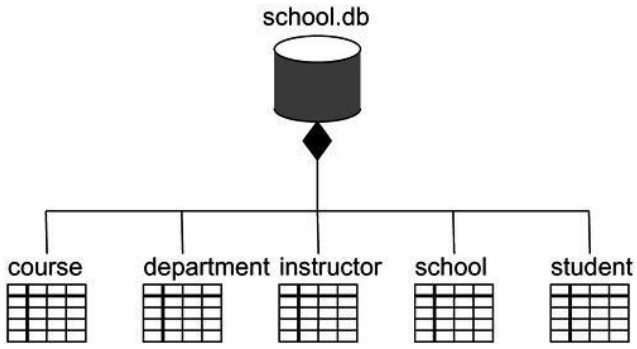


Figure : Modeling an Executable Release

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.



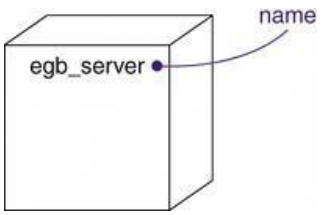
. Where possible, use tools to help you transform your logical design into a physical design.

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.

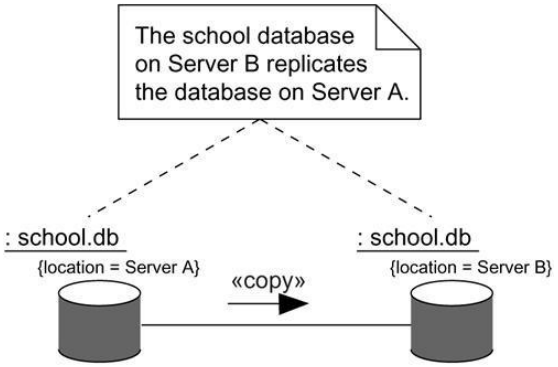


Figure : Modeling Adaptable Systems

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.

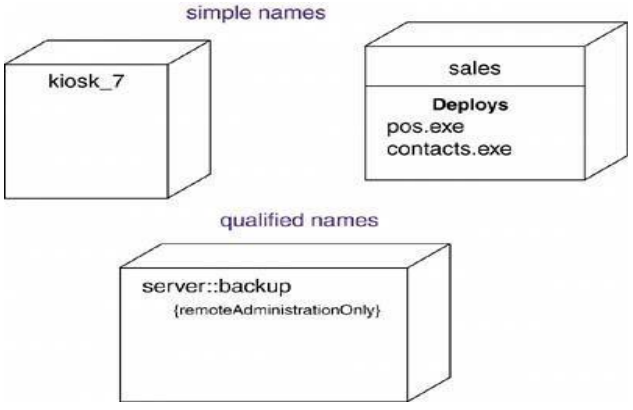


Figure Nodes with Simple and Qualified Names

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.

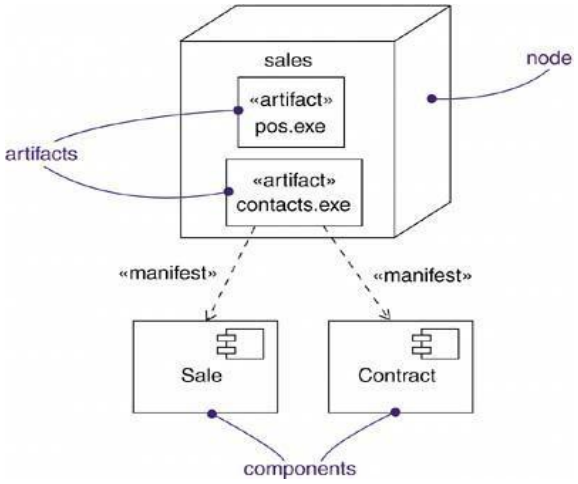


Figure: Nodes and Components

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.

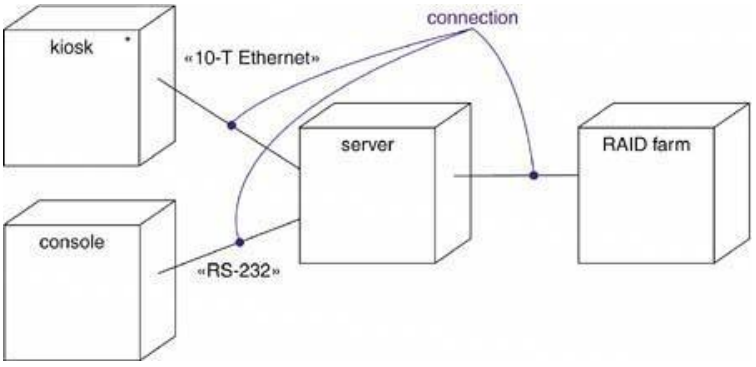


Figure : Connections

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.

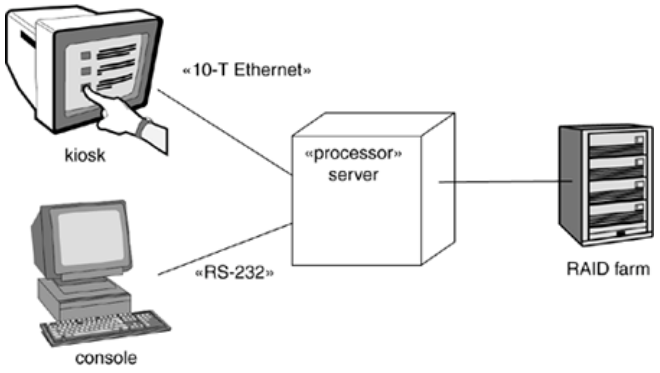


Figure : Processors and Devices

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 model that 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.

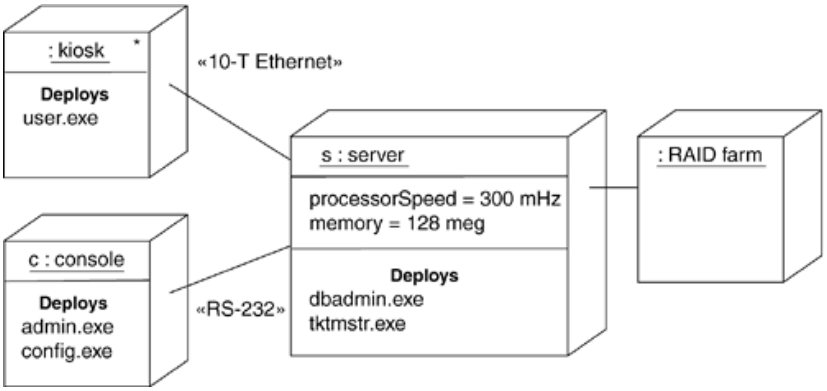


Figure: Modeling the Distribution of Components

Deployment Diagrams

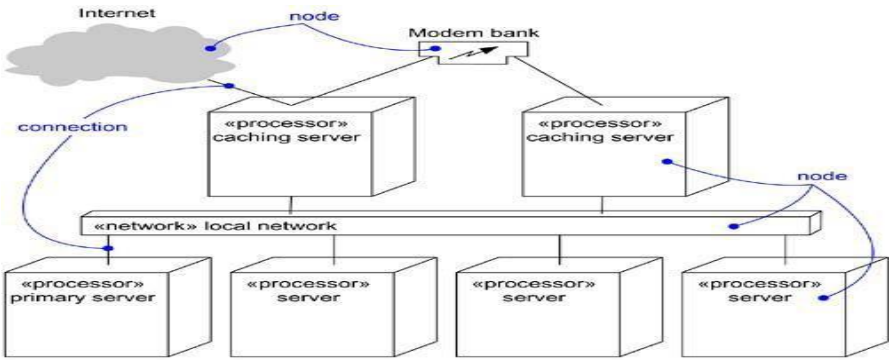


Figure : Deployment Diagram

- 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.

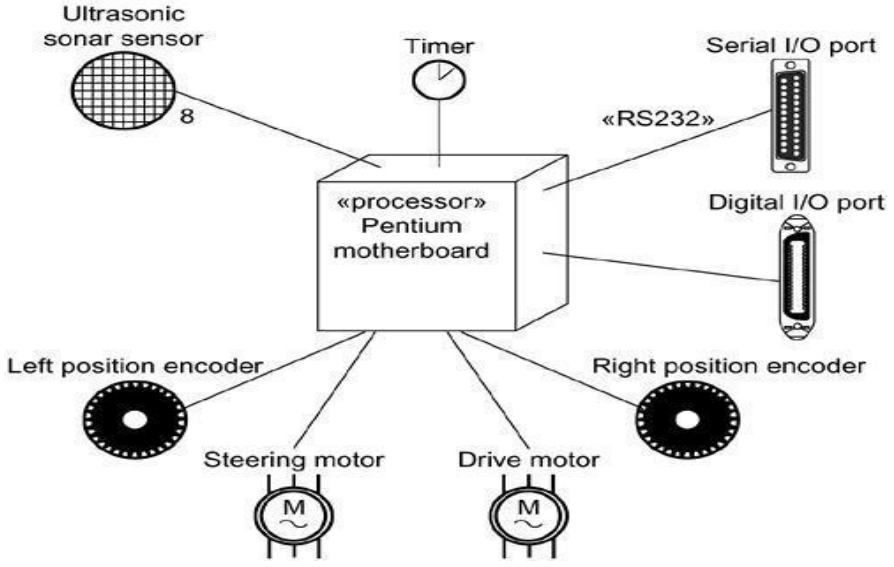


Figure : Modeling an Embedded System

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.

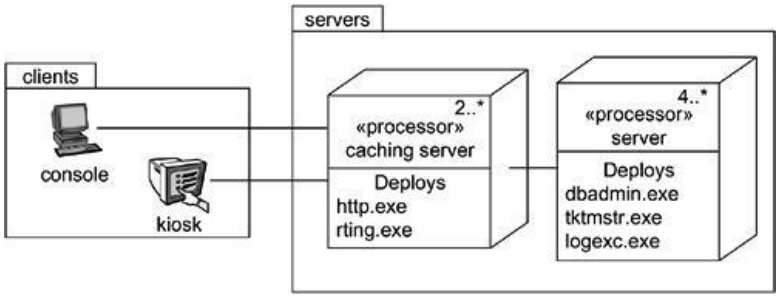


Figure : Modeling a Client/ Server System

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

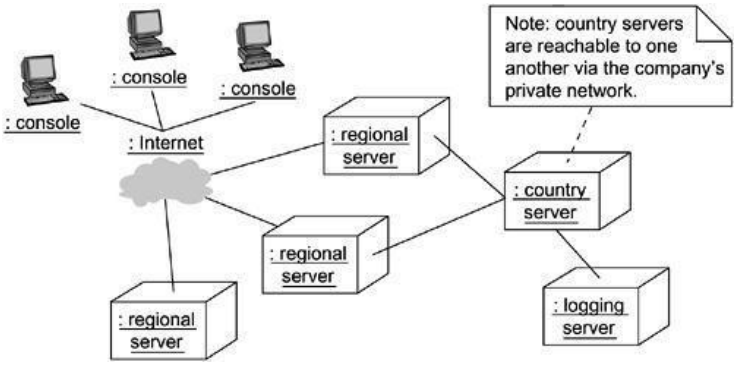


Figure : Modeling a Fully Distributed System

----------END----------