LECTURE NOTES

ON

OBJECT ORIENTED ANALYSIS AND DESIGN

B.Tech V Semester

Mr. G Chandra Sekhar
Assistant Professor
Mr. N Bhaswanth
Assistant Professor

INFORMATION TECHNOLOGY

INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)
DUNDIGAL, HYDERABAD - 500 043

UNIT-I

Introduction to UML: Importance of modeling, principles of modeling, object oriented modeling, conceptual model of the UML, architecture, software development life cycle; Classes, relationships, common mechanisms and diagrams.

What is the UML?

- "The Unified Modeling Language is a family of graphical notations, backed by a single meta-model, that help in describing and designing software systems, particularly software systems built using the object-oriented style."
- UML first appeared in 1997
- UML is standardized. Its content is controlled by the Object Management Group (OMG), a consortium of companies.

- Unified
  - UML combined the best from object-oriented software modeling methodologies that were in existence during the early 1990's.
  - Grady Booch, James Rumbaugh, and Ivor Jacobson are the primary contributors to UML.
- Modeling
  - Used to present a simplified view of reality in order to facilitate the design and implementation of object-oriented software systems.
  - All creative disciplines use some form of modeling as part of the creative process.
  - UML is a language for documenting design
  - Provides a record of what has been built.
  - Useful for bringing new programmers up to speed.

- Language
  - UML is primarily a graphical language that follows a precise syntax.
  - UML 2 is the most recent version
  - UML is standardized. Its content is controlled by the Object Management Group (OMG), a consortium of companies.

Why We Model

- The importance of modeling
- Four principles of modeling
- Object-oriented modeling

The Importance of Modeling

- A successful software organization is one that consistently deploys quality software that meets the needs of its users.
- An organization that can develop such software in a timely and predictable fashion, with an efficient and effective use of resources, both human and material, is one that has a sustainable business

What, then, is a model? Simply put,

- *A model is a simplification of reality.*

- A model provides the blueprints of a system.
- A good model includes those elements that have broad effect and omits those minor elements that are not relevant to the given level of abstraction.

Why do we model? There is one fundamental reason.

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.

Modeling is not just for big systems. Even the software equivalent of a dog house can benefit from some modeling.
We build models of complex systems because we cannot comprehend such a system in its entirety.

Principles of Modeling

Four principles of modeling:

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

Two Approaches:

- Traditional Approach
- Objected-Oriented Approach
- TRADITIONAL APPROACH
- Collection of programs or functions.
- A system that is designed for performing certain actions.
- Algorithms + Data Structures = Programs.

TRADITIONAL APPROACH	OBJECT ORIENTED SYSTEM DEVELOPMENT
Collection of procedures(functions)	Combination of data and functionality
Focuses on function and procedures, different styles and methodologies for each step of process	Focuses on object, classes, modules that can be easily replaced, modified and reused.
Moving from one phase to another phase is complex.	Moving from one phase to another phase is easier.
Increases duration of project	decreases duration of project
Increases complexity	Reduces complexity and redundancy

An Overview of the UML

The UML is a language for

• Visualizing
• Specifying
• Constructing
• Documenting

The UML Is a Language for Documenting

A healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include (but are not limited to)
• Requirements
• Architecture
• Design
• Source code
• Project plans
• Tests
• Prototypes
• Releases

Where Can the UML Be Used?

The UML is intended primarily for software-intensive systems. It has been used effectively for such domains as

• Enterprise information systems
• Banking and financial services
• Telecommunications
• Transportation
• Defense/aerospace
• Retail
• Medical electronics
• Scientific
• Distributed Web-based services

A Conceptual Model of the UML

- A conceptual model needs to be formed by an individual to understand UML.
- UML contains three types of building blocks: things, relationships, and diagrams.
- Things
  - Structural things
    - Classes, interfaces, collaborations, use cases, components, and nodes.
  - Behavioral things
    - Messages and states.
  - Grouping things
    - Packages
  - Annotational things
    - Notes

- Relationships: Dependency, Association, Generalization and Realization.
- Diagrams: class, object, use case, sequence, collaboration, statechart, activity, component and deployment.

Building Blocks of the UML:

The vocabulary of the UML encompasses three kinds of building blocks:
1. Things
2. Relationships
3. Diagrams

Things in the UML

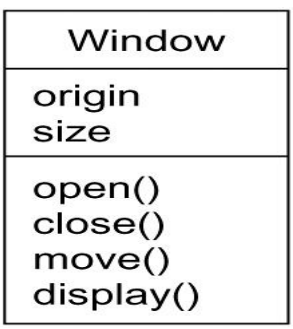
There are four kinds of things in the UML:
1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things

Structural Things

- *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.
- Classes
- Interface
- Cases
- Active Classes
- Components
- Nodes
- Collaborations

Classes:
a *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

Figure : Classes



Interfaces

- an *interface* is a collection of operations that specify a service of a class or component. An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface

Figure :Interfaces

Collaborations:

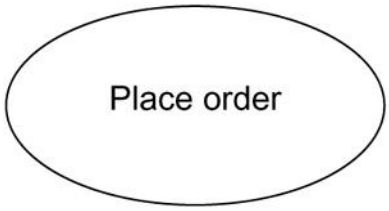
- *A collaboration* defines an interaction. These collaborations therefore represent the implementation of patterns that make up a system. Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its name

Figure: Collaborations

Use Cases:

- 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

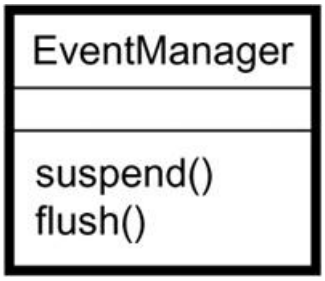
Figure :Use Cases



Active Classes:

- An active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations

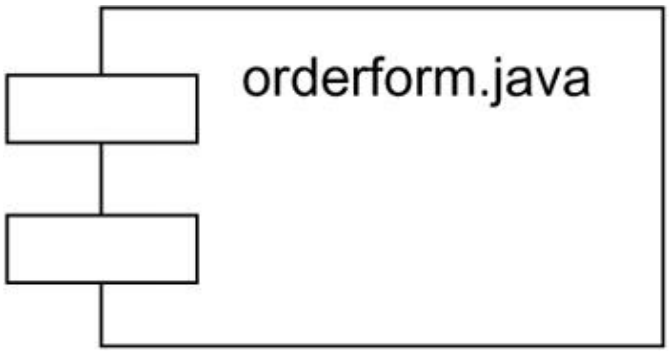
Figure :Active Classes



Components:

- A component typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations. Graphically, a component is rendered as a rectangle with tabs, usually including only its name.

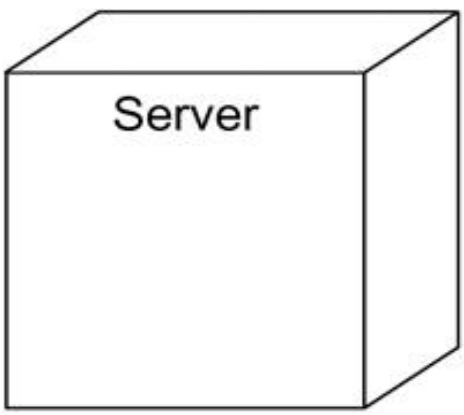
Figure :Components



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.
- A set of components may reside on a node and may also migrate from node to node. Graphically, a node is rendered as a cube, usually including only its name.

Figure :Nodes



Behavioral Things:

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.

1. Messages
2. States

Messages:

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

States:

- A *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.

Grouping Things:

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

Packages:

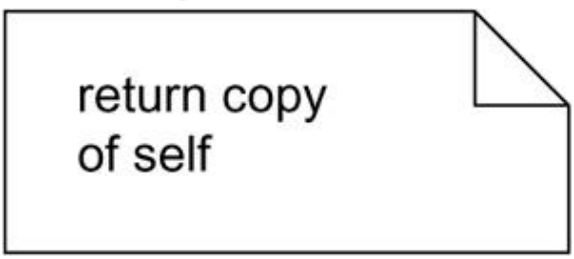
- A *package* is a general-purpose mechanism for organizing elements into groups. Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents

Figure: Packages

Annotational Things:

- *Annotational things* are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model.
- There is one primary kind of annotation thing, called a note. A *note* is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.

Figure: Notes



Relationships in the UML

There are four kinds of relationships in the UML:

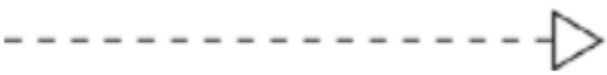
1. Dependency 2. Association 3. Generalization 4. Realization

Dependency is a semantic relationship between two model elements in which a change to one element (the independent one) may affect the semantics of the other element (the dependent one). Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label.

Association is a structural relationship among classes that describes a set of links, a link being a connection among objects that are instances of the classes.
Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and end names

Generalization is a specialization/generalization relationship in which the specialized element (the child) builds on the specification of the generalized element (the parent). The child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent.

Realization is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. generalization and a dependency relationship.



UML Diagrams

- A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and paths (relationships).
- A diagram represents an elided view of the elements that make up a system.
- In theory, a diagram may contain any combination of things and relationships.
- In practice, a small number of common combinations arise, which are consistent with the five most useful views that comprise the architecture of a software intensive system

The UML includes Nine kinds of diagrams:

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

1. Class diagram shows a set of classes, interfaces, and collaborations and their relationships. These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.

2. Object diagram shows a set of objects and their relationships. 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 as do class diagrams.
3. Use case diagram shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system.
4. Sequence diagram is an interaction diagram that emphasizes the time-ordering of messages;
5. Collaboration diagram a communication diagram is an interaction diagram that emphasizes the structural organization of the objects or roles that send and receive messages.
6. Statechart diagram shows a state machine, consisting of states, transitions, events, and activities. A state diagrams shows the dynamic view of an object.
7. Activity diagram shows the structure of a process or other computation as the flow of control and data from step to step within the computation. Activity diagrams address the dynamic view of a system.
8. Component diagram is shows an encapsulated class and its interfaces, ports, and internal structure consisting of nested components and connectors. Component diagrams address the static design implementation view of a system.
9. 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

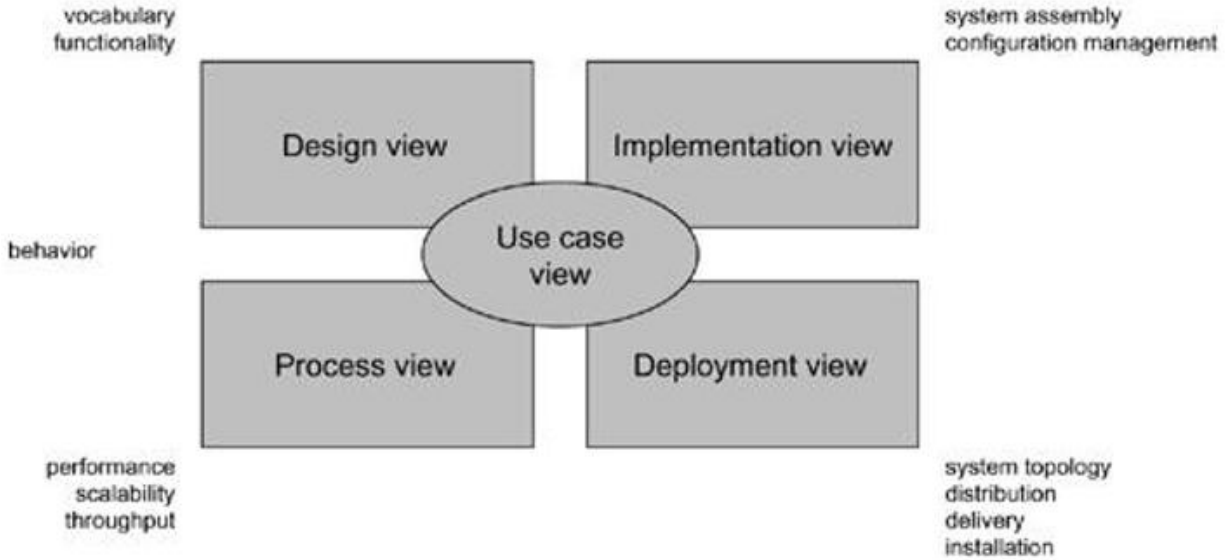
What is Legal UML?

The UML has syntactic and semantic rules for

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

Architecture

- Architecture refers to the different perspectives from which a complex system can be viewed.
- Visualizing, specifying, constructing, and documenting a software-intensive system demands that the system be viewed from a number of perspectives.
- The architecture of a software-intensive system is best described by five interlocking views:
  - Use case view: system as seen by users, analysts and testers.
  - Design view: classes, interfaces and collaborations that make up the system.
  - Process view: active classes (threads).
  - Implementation view: files that comprise the system.
  - Deployment view: nodes on which SW resides.
- Each view is a projection into the organization and structure of the system, focused on a particular aspect of that system.

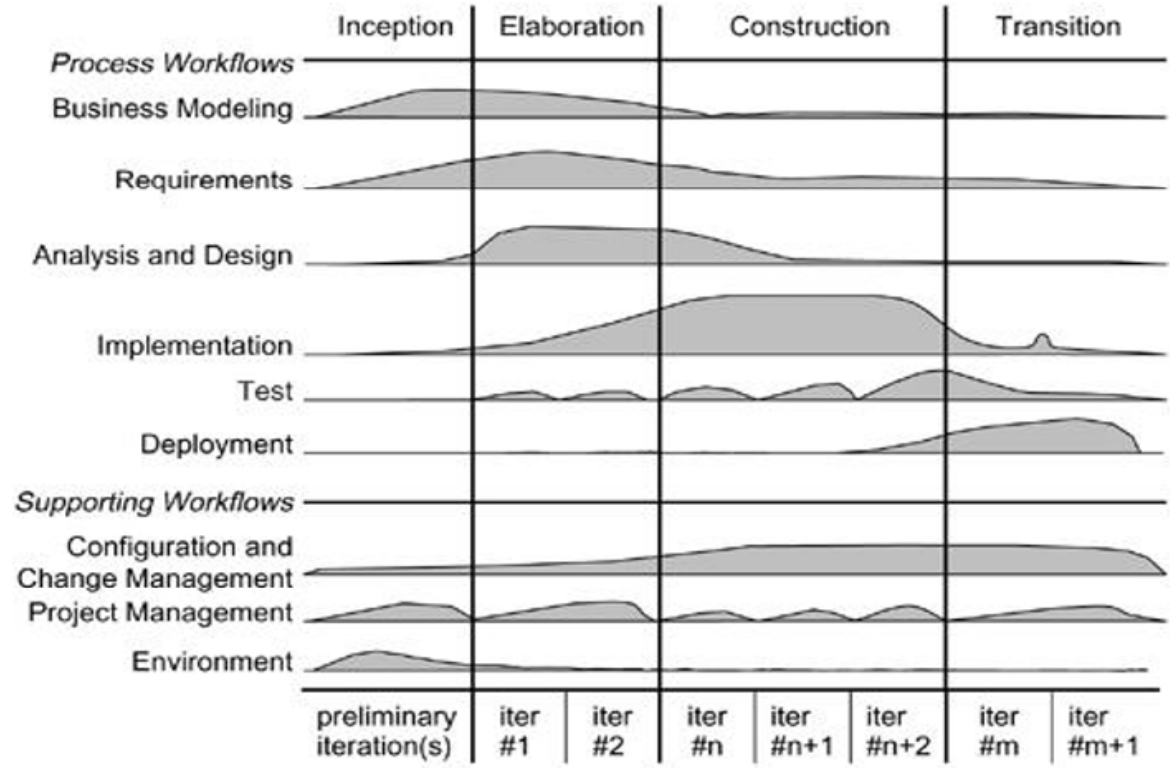


Each of these five views can stand alone so that different stakeholders can focus on the issues of the system's architecture that most concern them.

Software Development Life Cycle

- UML is involved in each phase of the software development life cycle.
- The UML development process is
  - Use case driven
  - 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
  - Architecture-centric means that a system's architecture is used as a primary artifact for conceptualizing, constructing, managing, and evolving the system under development.
  - Iterative and incremental
  - 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.

Software Development Life Cycle



- **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.

CLASSES

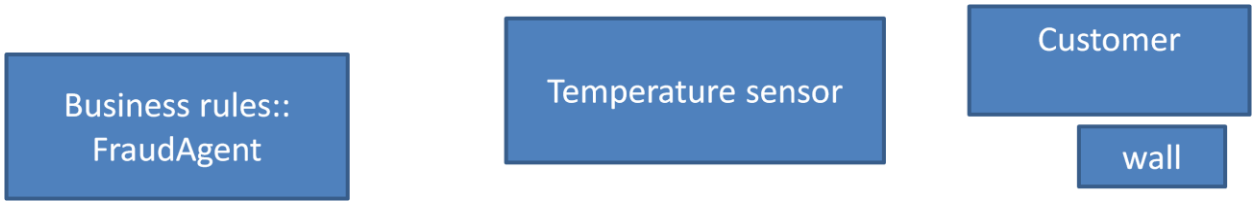
Classes

- A Class is a description of set of objects that share same attributes, operations, relationships and semantics .
- Graphically, a class is rendered as a rectangle

Name

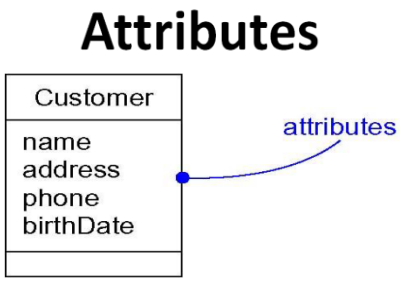
- Every class must have a name that distinguishes it from other classes. A *name is a textual string.*
- That name alone is known as a *simple name;*

- a *path name is the class name prefixed by the* name of the package in which that class lives.

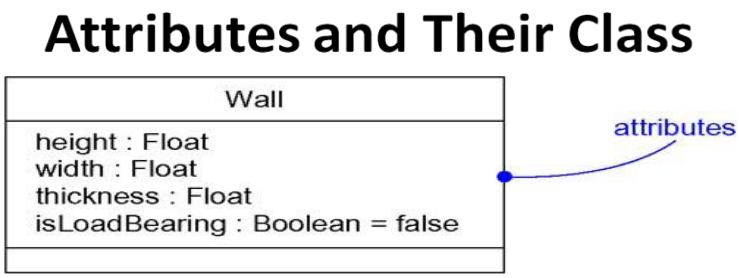


Attributes

- An attribute is a named property of a class that describes range of values that instances of the property may hold.
- A class may have any number of attributes or no attributes at all. An attribute
- represents some property of the thing you are modeling that is shared by all objects of that class.
- Graphically, attributes are listed in a compartment just below the class name.
- Attributes may be drawn showing only their names,

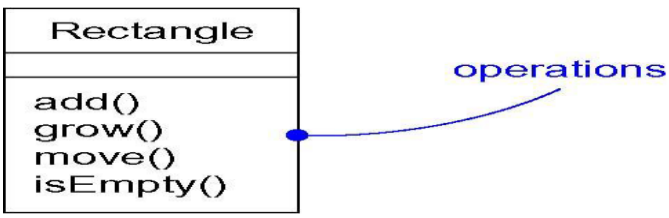


further specify an attribute by stating its class and possibly a default initial value



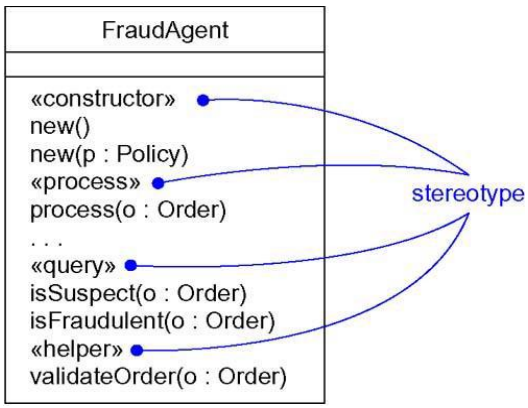
Operations

- An operation is the implementation of a service that can be requested from any object of the class to affect behavior.
- An operation is an abstraction of something you can do to an object and that is shared by all objects of that class.
- A class may have any number of operations or no operations at all.
- Operations may be drawn showing only their names.



Organizing attributes and relationships

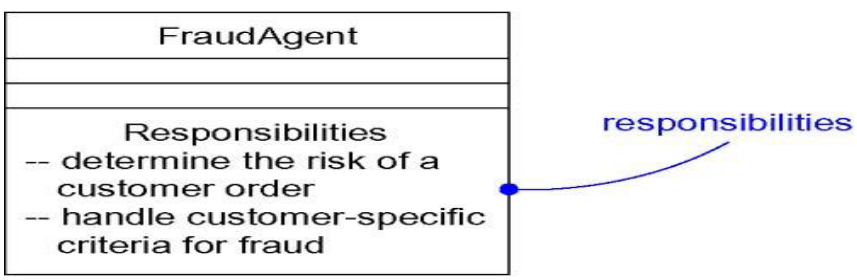
- When drawing a class, you don't have to show every attribute and every operation at once.
- Meaning that you can choose to show only some or none of a class's attributes and operations
- Explicitly specify that there are more attributes or properties than shown by ending each list with an ellipsis ("...").
- To better organize long lists of attributes and operations you prefix each group with descriptive category by using stereotypes



Responsibilities

- A responsibility is a contract or an obligation of a class.
- When you create a class you are making a statement that all objects of that class have the same kind of state and behavior .
- Ex: A Wall class is responsible for knowing about height, width, and thickness; a FraudAgent class, as you might find in a credit card application, is responsible for processing orders and determining if they are legitimate, suspect, or fraudulent; a TemperatureSensor class is responsible for measuring temperature and raising an alarm if the temperature reaches a certain point.
- Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon

Fig: Responsibilities



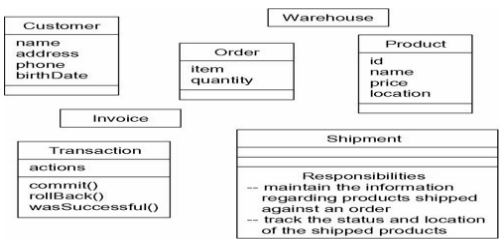
Common modeling techniques

1. Modeling the vocabulary of a system

To model the vocabulary of a system

1) Identify those things that users to describe the problem .use crc cards and usecase based analysis to help find these abstractions.
2) For each abstraction, identify a set of responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all your classes.
3) Provide the attributes and operations that are needed to carry out these responsibilities for each class

- Fig shows a set of classes drawn from a retail system, including Customer, Order, and Product. It also includes a few other related abstractions drawn from the vocabulary of the problem, such as Shipment (used to track orders), Invoice (used to bill orders), and Warehouse (where products are located prior to shipment). There is also one solution-related abstraction, Transaction, which applies to orders and shipments.

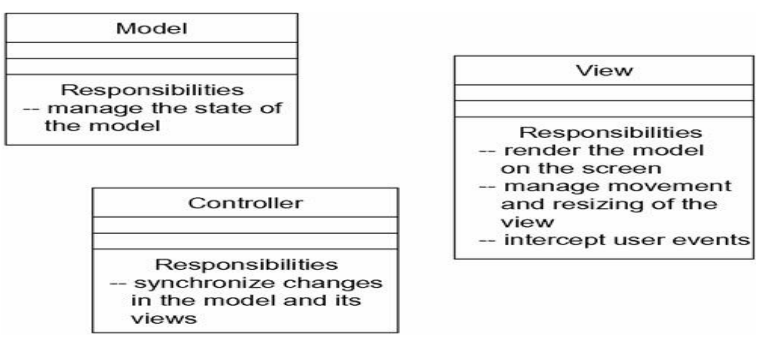


2. Modeling the Distribution of responsibilities in a System

To model the distribution of responsibilities in a System

- Identify a set of classes that work together closely to carryout some behavior.
- Identify a set of responsibilities for each of these classes.
- Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions, collapse tiny classes that have trivial responsibilities into larger ones, and reallocate responsibilities so that each abstraction reasonably stands on its own.
- Consider the ways in which those classes collaborate with one another, and redistribute their responsibilities accordingly so that no class within a collaboration does too much or too little.

Modeling the Distribution of responsibilities in a System



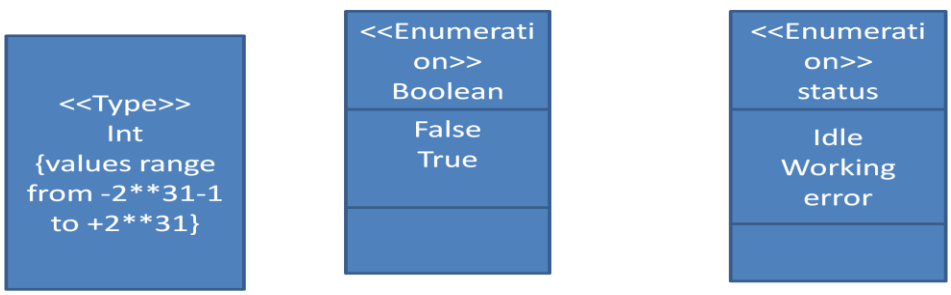
3. Modeling Non software things

To model the distribution of responsibilities in a System
Model the thing you are abstracting as a class.

- If you want to distinguish these things from the UML's defined building blocks, create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.
- If the thing you are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node, as well, so that you can further expand on its structure.

Modeling Non software Things

Modeling Non Software things



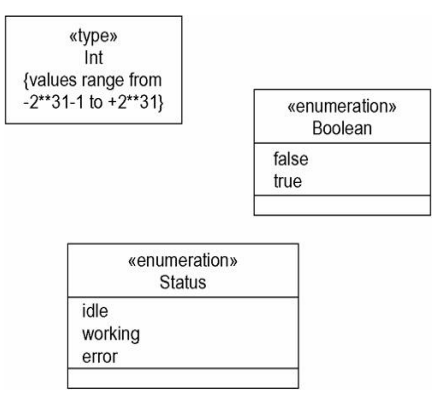
4. Modeling Primitive Types

- At the other extreme, the things you model may be drawn directly from the programming language you are using to implement a solution.
- Typically, these abstractions involve primitive types, such as integers, characters, strings, and even enumeration types, that you might create yourself.

To model primitive types,

- Model the thing you are abstracting as a type or an enumeration, which is rendered using class notation with the appropriate stereotype.
- If you need to specify the range of values associated with this type, use constraints.

Fig: Modeling Primitive Types



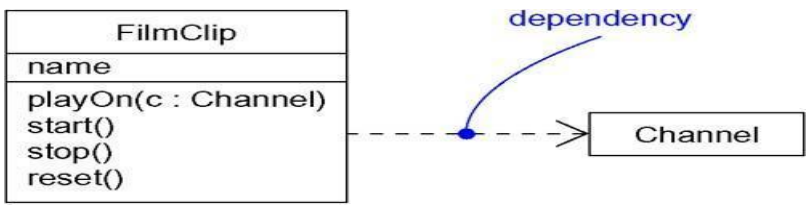
Relationships

- A *relationship is a connection among things.*
- *The three most* important relationships are dependencies, generalizations, and associations.
- Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships.

Dependency

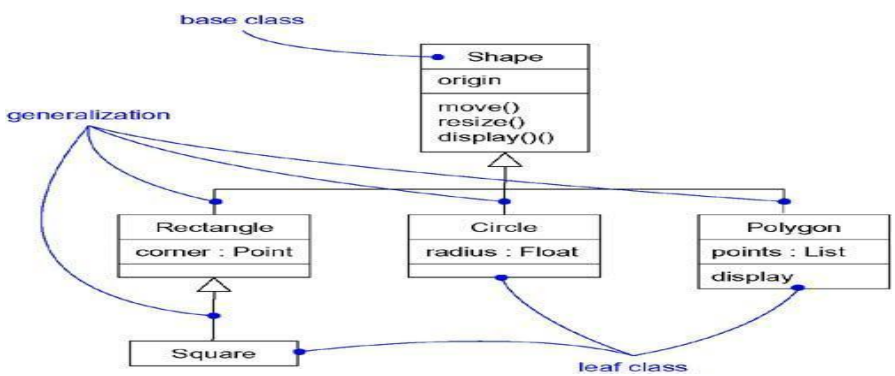
A *dependency* is a using relationship that states that a change in specification of one thing (for example, class **Event**) may affect another thing that uses it (for example, class **Window**), but not necessarily the reverse.

- Graphically, a dependency is rendered as a dashed directed line.



Generalization

- A *generalization* is a relationship between a general thing (called the super class or parent)and a more specific kind of that thing (called the subclass or child).
- Generalization is sometimes called an "is-a-kind-of" relationship: one thing (like the class **BayWindow**) is-a-kind-of a more general thing (for example, the class **Window**).
- Generalization means that objects of the child may be used anywhere the parent may appear, but not the reverse.
- In other words, generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations.



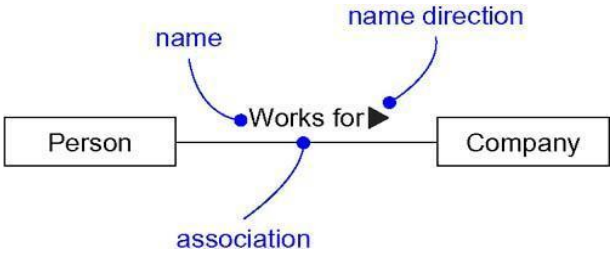
Association

- An *association* is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can navigate from an object of one class to an object of the other class, and vice versa.
- It's quite legal to have both ends of an association circle back to the same class. This means that, given an object of the class, you can link to other objects of the same class

There are four adornments that apply to associations.

Name

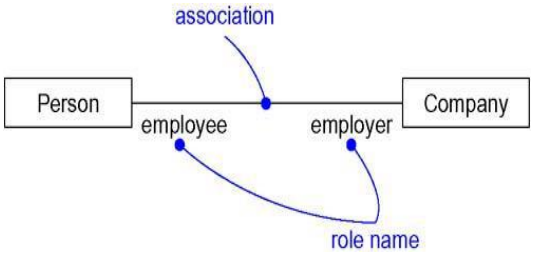
An association can have a name, and you use that name to describe the nature of the relationship. So that there is no ambiguity about its meaning, you can give a direction to the name by providing a direction triangle that points in the direction you intend to read the name, as shown in Figure



There are four adornments that apply to associations.

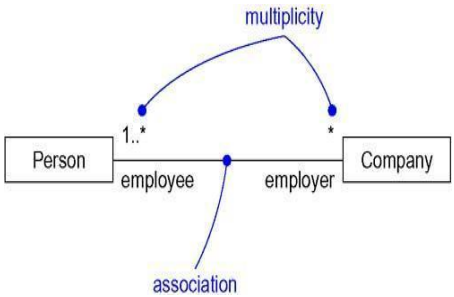
Role

- When a class participates in an association, it has a specific role that it plays in that relationship;
- A role is just the face the class at the near end of the association presents to the class at the other end of the association.



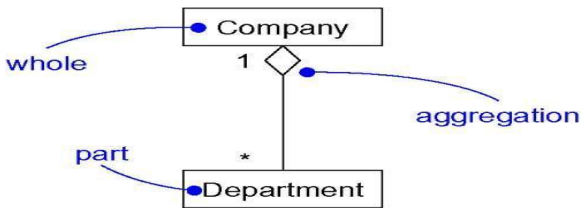
Multiplicity

- An association represents a structural relationship among objects. In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association.
- This "how many" is called the multiplicity of an association's role, and is written as an expression that evaluates to a range of values or an explicit value as in Figure



Aggregation

- A plain association between two classes represents a structural relationship between peers, meaning that both classes are conceptually at the same level, no one more important than the other.
- Sometimes, you will want to model a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts"). This kind of relationship is called aggregation, which represents a "has-a" relationship.

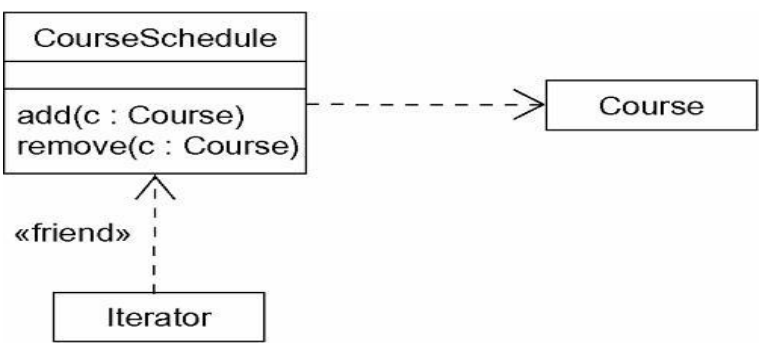


Common Modeling Techniques

1. Modeling simple dependencies

-To model this using relationship

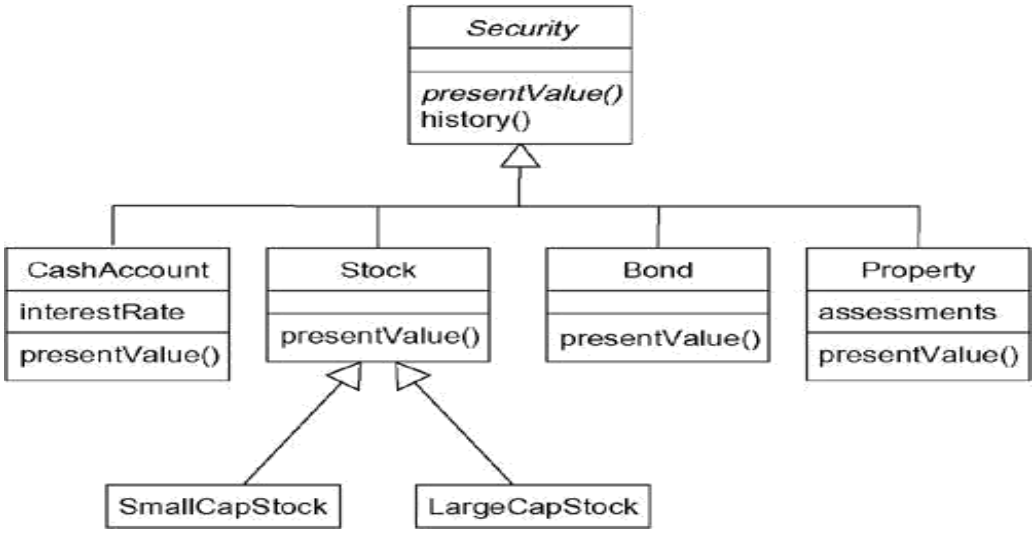
1) Create a dependency pointing from the class with the operation to the class used as a parameter in the operation.



2. Modeling Single Inheritance

To model inheritance relationships,

1. Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.
2. Elevate these common responsibilities, attributes, and operations to a more general class. If necessary, create a new class to which you can assign these elements (but be careful about introducing too many levels).
3. Specify that the more-specific classes inherit from the more-general class by placing a generalization relationship that is drawn from each specialized class to its more-general parent.



3. Modeling Structural Relationships

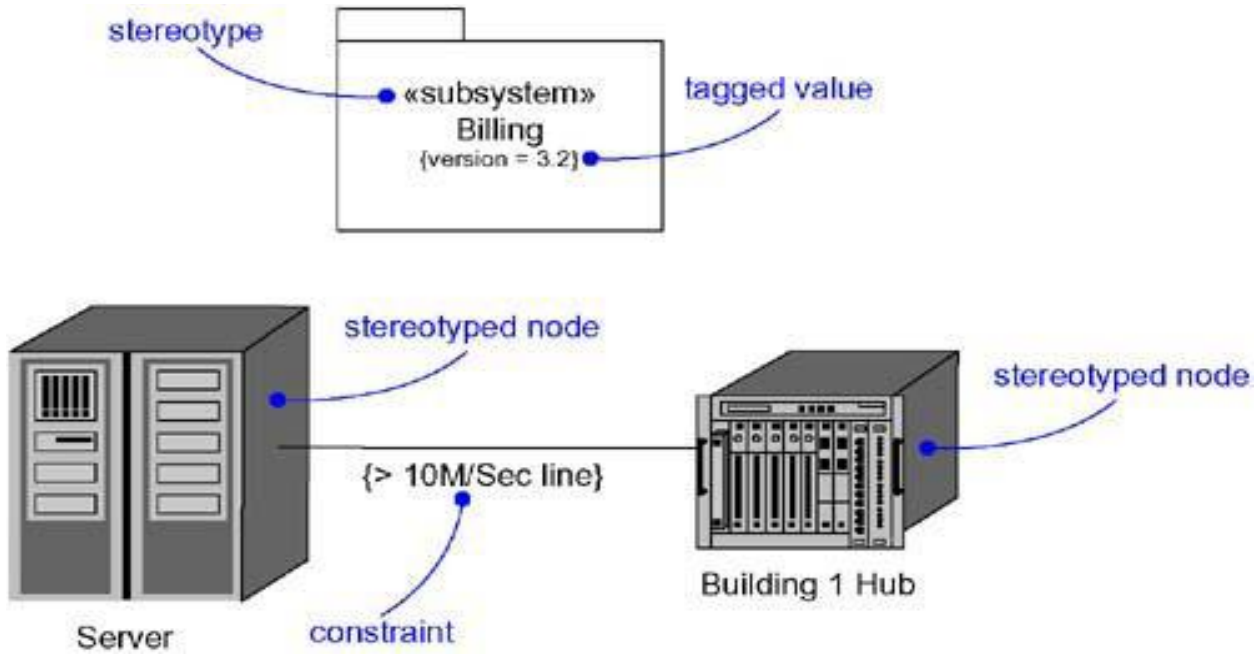
To model structural relationships,

1. For each pair of classes, if you need to navigate from objects of one to objects of another, specify an association between the two. This is a data-driven view of associations.
2. For each pair of classes, if objects of one class need to interact with objects of the other class other than as parameters to an operation, specify an association between the two. This is more of a behavior-driven view of associations.
3. For each of these associations, specify a multiplicity (especially when the multiplicity is not *****, which is the default), as well as role names (especially if it helps to explain the model).
4. If one of the classes in an association is structurally or organizationally a whole compared with the classes at the other end that look like parts, mark this as an aggregation by adorning the association at the end near the whole

COMMON MECHANISMS

- Stereotypes, tagged values, and constraints are the mechanisms provided by the UML to add new building blocks, create new properties, and specify new semantics.
- For example, if you are modeling a network, you might want to have symbols for routers and hubs; then use stereotyped nodes to make these things appear as primitive building blocks.
- Similarly, if you are part of your project's release team, responsible for assembling, testing, and then deploying releases, you might want to keep track of the version number and test results for each major subsystem.
- Then use tagged values to add this information to your models.

Fig: Stereotypes, Tagged Values, and constraints



A ***note*** is a graphical symbol for rendering constraints or 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.

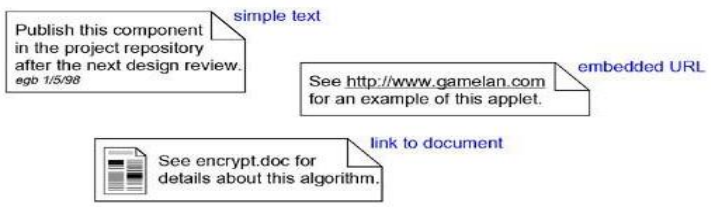
A ***stereotype*** is an extension of the vocabulary of the UML, allowing to create new kinds of building blocks similar to existing ones but specific to problem. Graphically, a stereotype is rendered as a name enclosed by guillemets (<< >>) and placed above the name of another element.

A ***tagged value*** is an extension of the properties of a UML element, allowing you to create new information in that element's specification. Graphically, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element.

A ***constraint*** is an extension of the semantics of a UML element, allowing you to add new rules or to modify existing ones. Graphically, a constraint is rendered as a string enclosed by brackets and placed near the associated element or connected to that element or elements by dependency relationships.

Notes

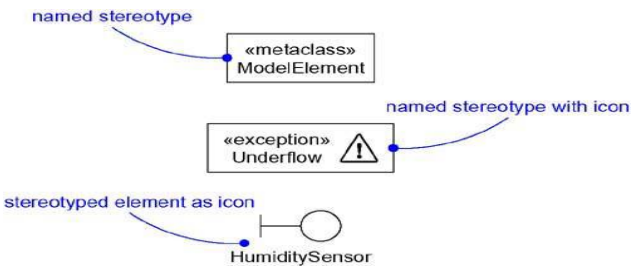
- A note that renders a comment has no semantic impact, meaning that its contents do not alter the meaning of the model to which it is attached. Notes are used to specify things like requirements, observations, reviews, and explanations, in addition to rendering constraints.
- A note may contain any combination of text or graphics. you can put a live URL inside a note, or even link to or embed another document



Other Adornments

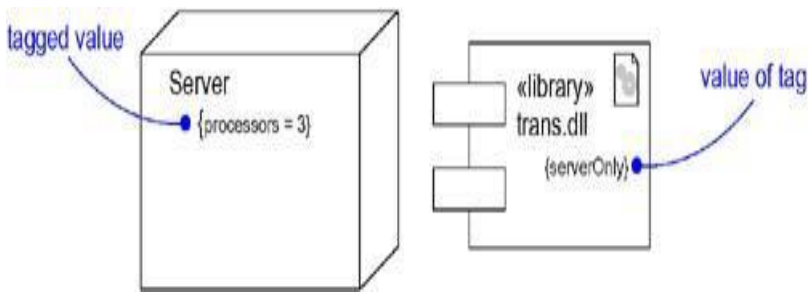
Adornments are textual or graphical items that are added to an element's basic notation and are used to visualize details from the element's specification.

- For example, the basic notation for an association is a line, but this may be adorned with such details as the role and multiplicity of each end.
- A stereotype is rendered as a name enclosed by guillemets (for example, <<name>>) and placed above the name of another element.
- You may define an icon for the stereotype and render that icon to the right of the name or use that icon as the basic symbol for the stereotyped item.



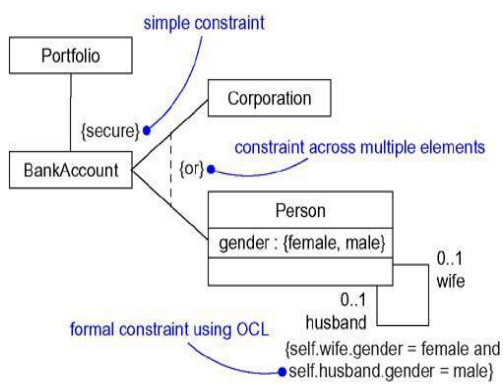
Tagged Values

A tagged value is rendered as a string enclosed by brackets and placed below the name of another element. That string includes a name (the tag), a separator (the symbol =), and a value (of the tag). Specify just the value if its meaning is unambiguous, such as when the value is the name of enumeration.



Constraints

- A constraint is rendered as a string enclosed by brackets and placed near the associated element.
- This notation is also used as an adornment to the basic notation of an element to visualize parts of an element's specification that have no graphical cue.
- For example, some properties of associations (order and changeability) are rendered using constraint notation.



Standard Elements

- The UML defines a number of standard stereotypes for classifiers, components, relationships and other modeling elements.
- There is one standard stereotype, mainly of interest to tool builders, that lets you model stereotypes themselves.

Stereotype

- Specifies that the classifier is a stereotype that may be applied to other elements
- The UML also specifies one standard tagged value that applies to all modeling elements.

Documentation

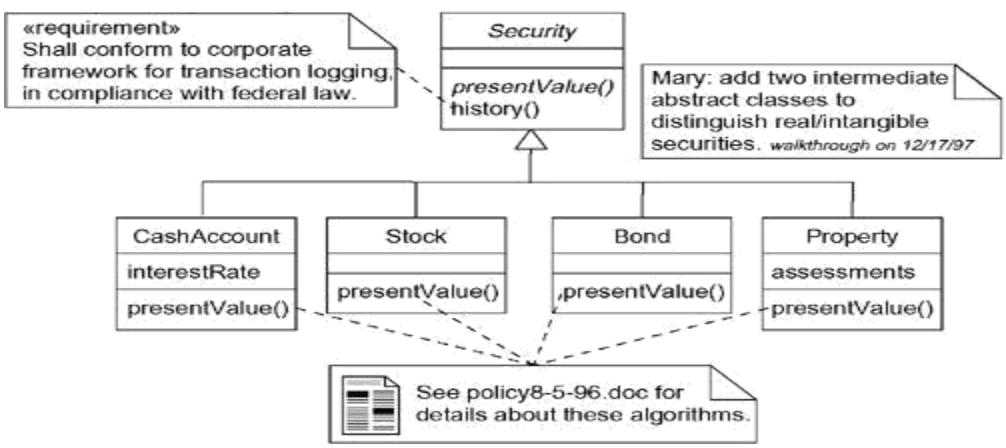
- Specifies a comment, description, or explanation of the element to which it is attached

Common Modeling Techniques

1. Modeling Comments

- To model a comment

1) Put your comment as text in a note and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a note to its elements using a dependency relationship.
2) Remember that you can hide or make visible the elements of your model as you see fit. This means that you don't have to make your comments visible everywhere the elements to which it is attached are visible. Rather, expose your comments in your diagrams only insofar as you need to communicate that information in that context.
3) If your comment is lengthy or involves something richer than plain text, consider putting your comment in an external document and linking or embedding that document in a note attached to your model.
4) As your model evolves, keep those comments that record significant decisions that cannot be inferred from the model itself, and unless they are of historic interest discard the others.



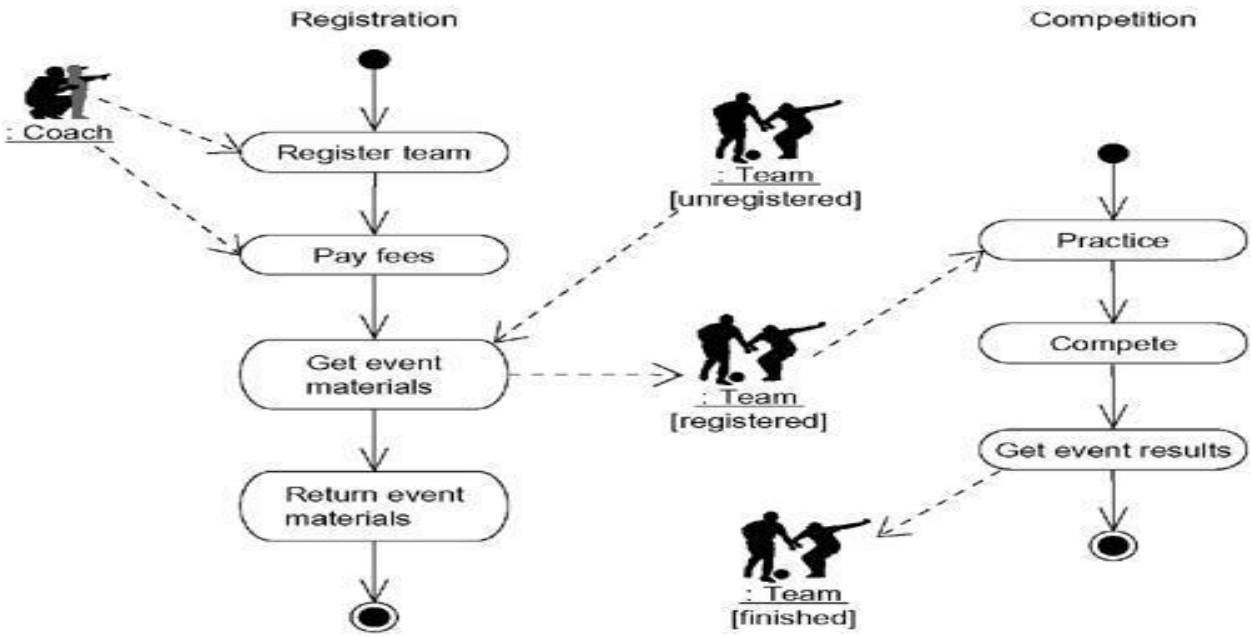
2. Modeling New Building Blocks

1) Make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are there's already some standard stereotype that will do what you want.

2) If you're convinced there's no other way to express these semantics, identify the primitive thing in the UML that's most like what you want to model (for example, class, interface, component, node, association, and so on) and define a new stereotype for that thing.

3) Specify the common properties and semantics that go beyond the basic element being stereotyped by defining a set of tagged values and constraints for the stereotype.

4) If you want these stereotype elements to have a distinctive visual cue, define a new icon for the stereotype.

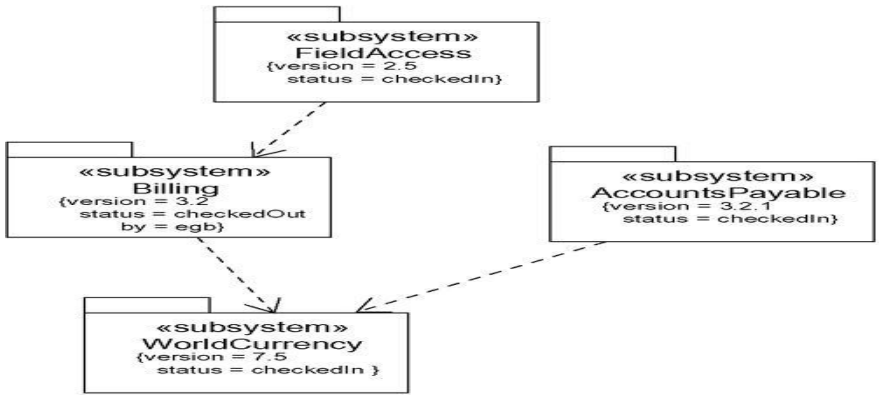


3) Modeling New Properties

1) First, make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there's already some standard tagged value that will do what you want.

2) If you're convinced there's no other way to express these semantics, add this new property to an individual element or a stereotype. The rules of generalization apply -- tagged values defined for one kind of element apply to its children.

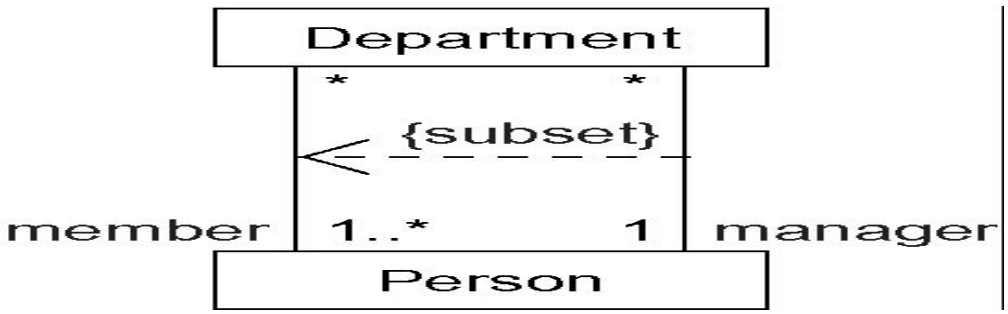
Fig: Modeling New Properties



4) To model new semantics

1. First, make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there's already some standard constraint that will do what you want.
2. If you're convinced there's no other way to express these semantics, write your new semantics as text in a constraint and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a constraint to its elements using a dependency relationship.
3. If you need to specify your semantics more precisely and formally, write your new semantics using Object Constraint Language (OCL).

Fig: Modeling New Semantics



The above diagram shows that each Person may be a member of zero or more Departments and that each Department must have at least one Person as a member. This diagram goes on to indicate that each Department must have exactly one Person as a manager and every Person may be the manager of zero or more Departments. All of these semantics can be expressed using simple UML.

DIAGRAMS

- A system is a collection of subsystems organized to accomplish a purpose and described by a set of models, possibly from different viewpoints.
- A subsystem is a grouping of elements, of which some constitute a specification of the behavior offered by the other contained elements.
- A diagram is just a graphical projection into the elements that make up a system.

Static parts of a system	Dynamic parts of a system.
Class diagram	Use case diagram
Object diagram	Sequence diagram
Component diagram	Collaboration diagram
Deployment diagram	Statechart diagram
	Activity diagram

The UML's structural diagrams are roughly organized around the major groups of things you'll find when modeling a system.

Class diagram	Classes, interfaces, and collaborations
Object diagram	Objects
Component diagram	Components
Deployment diagram	Nodes

The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.

Class diagram	Classes, interfaces, and collaborations
Use case diagram	Organizes the behaviors of the system
Sequence diagram	Focused on the time ordering of messages
Collaboration diagram	Focused on the structural organization of objects that send and receive messages
Statechart diagram	Focused on the changing state of a system driven by events
Activity diagram	Focused on the flow of control from activity to activity

Common Modeling Techniques

1. Modeling Different Views of a System

- Decide which views you need to best express the architecture of your system and to expose the technical risks to your project. The five views of an architecture described earlier are a good starting point.
- For each of these views, decide which artifacts you need to create to capture the essential details of that view. For the most part, these artifacts will consist of various UML diagrams.
- As part of your process planning, decide which of these diagrams you'll want to put under some sort of formal or semi-formal control. These are the diagrams for which you'll want to schedule reviews and to preserve as documentation for the project.
- Allow room for diagrams that are thrown away. Such transitory diagrams are still useful for exploring the implications of your decisions and for experimenting with changes.

2) Modeling Different Levels of Abstraction

- Consider the needs of your readers, and start with a given model.
- If your reader is using the model to construct an implementation, she'll need diagrams that are at a lower level of abstraction, which means that they'll need to reveal a lot of detail. The model to present a conceptual model to an end user, then use the diagrams that are at a higher level of abstraction, which means that they'll hide a lot of detail.
- Depending on where you land in this spectrum of low-to-high levels of abstraction, create a diagram at the right level of abstraction by hiding or revealing the following four categories of things from the model.

Building blocks and relationships:

- Hide those that are not relevant to the intent of the diagram or the needs of the reader.

Adornments:

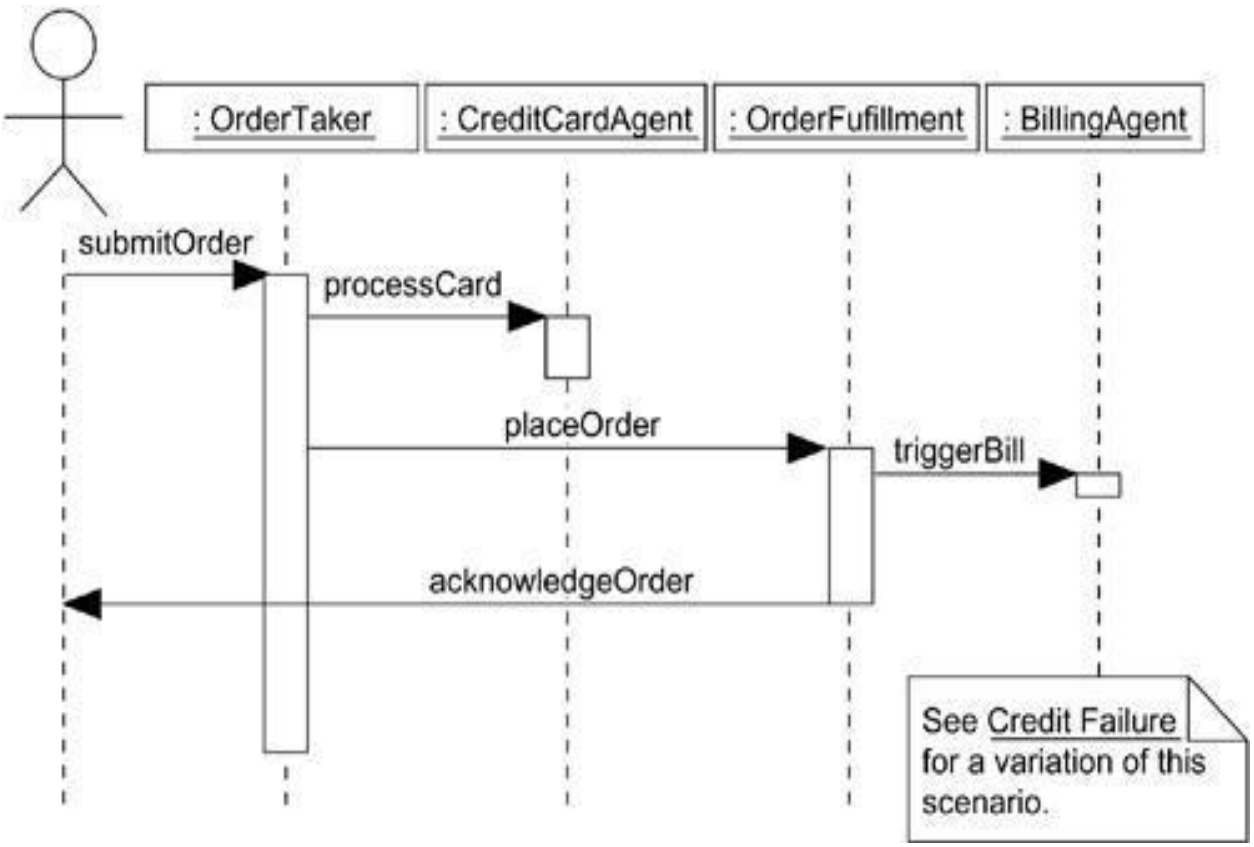
- Reveal only the adornments of these building blocks and relationships that are essential to understanding the intent.

Flow:

- In the context of behavioral diagrams, expand only those messages or transitions that are essential to understanding the intent.

Stereotypes:

- In the context of stereotypes used to classify lists of things, such as attributes and operations, reveal only those stereotyped items that are essential to understanding the intent.



3) Modeling Complex Views

To model complex views,

- First, convince yourself there's no meaningful way to present this information at a higher level of abstraction, perhaps eliding some parts of the diagram and retaining the detail in other parts.
- If you've hidden as much detail as you can and your diagram is still complex, consider grouping some of the elements in packages or in higher level collaborations, then render only those packages or collaborations in your diagram.
- If your diagram is still complex, use notes and color as visual cues to draw the reader's attention to the points you want to make.
- If your diagram is still complex, print it in its entirety and hang it on a convenient large wall. You lose the interactivity an online version of the diagram brings, but you can step back from the diagram and study it for common patterns.

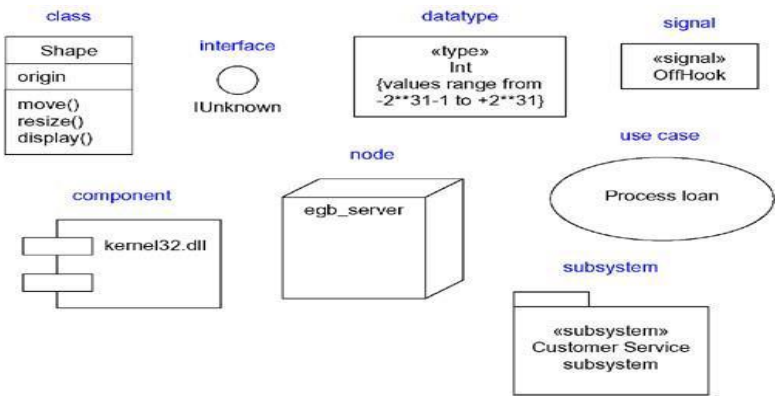
******

UNIT – II

Advanced classes, advanced relationships, interfaces, types and roles, packages, terms, concepts, modeling techniques for class and object diagrams; Interactions: Interaction diagrams; Use cases: Use case diagrams, activity diagrams.

Advanced Classes

- A *classifier* is a mechanism that describes structural and behavioral features.
- Classifiers include classes, interfaces, datatypes, signals, components, nodes, use cases, and subsystems.
- **The UML provides a number of other kinds of classifiers to help you model.**
- **Interface**
  A collection of operations that are used to specify a service of a class or a component
- **Data type**
  A type whose values have no identity, including primitive built-in types (such as numbers and strings), as well as enumeration types (such as Boolean)
- **Signal**
  The specification of an asynchronous stimulus communicated between instances
- **component**
  A physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces
- **Node**
  A physical element that exists at run time and that represents a computational resource, generally having at least some memory and often processing capability
- **Use case**
  A description of a set of a sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor
- **Subsystem**
  A grouping of elements of which some constitute a specification of the behavior offered by the other contained elements

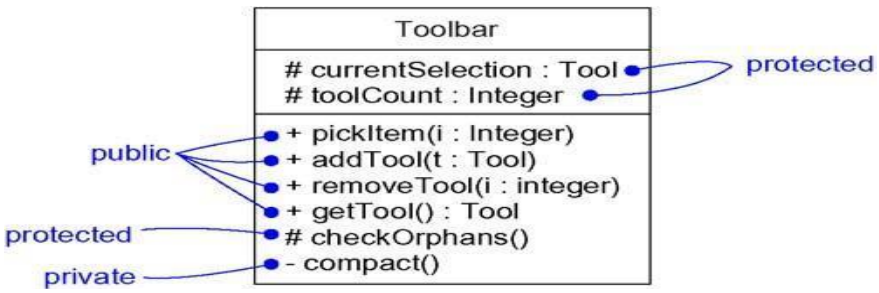


Visibility - UML, you can specify any of three levels of visibility.

1. **public**
   Any outside classifier with visibility to the given classifier can use the feature specified by prepending the symbol **+.**

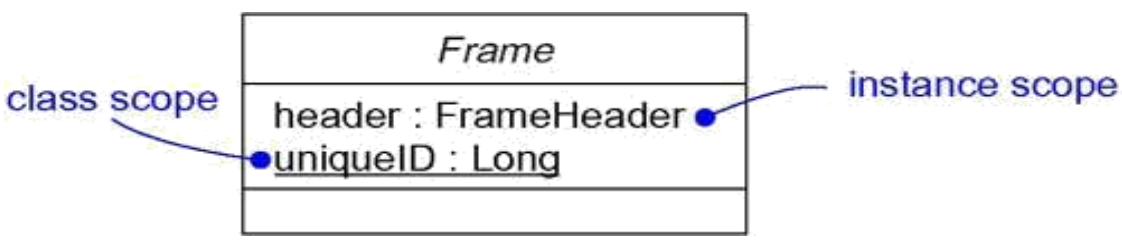
2. **protected**
   Any descendant of the classifier can use the feature; specified by prepending the symbol **#.**

3. **private**
   Only the classifier itself can use the feature; specified by prepending the symbol **-.**



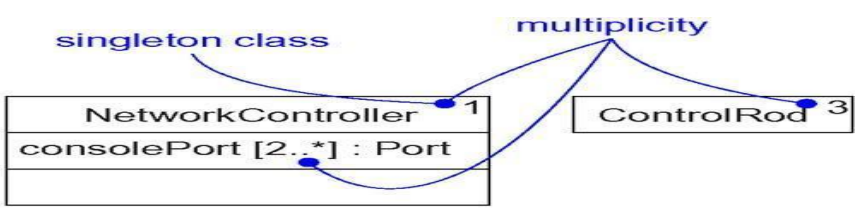
Scope- It specifies whether the feature appears in each instance of the classifier or whether there is just a single instance of the feature for all instances of the classifier. In the UML, you can specify two kinds of owner scope.

1. **Instance -** Each instance of the classifier holds its own value for the feature.
2. **Classifier -** There is just one value of the feature for all instances of the classifier.
   - Figure shows, a feature that is classifier scoped is rendered by underlining the feature's name



Multiplicity

- The number of instances a class may have is called its multiplicity. Multiplicity is a specification of the range of allowable cardinalities an entity may assume.
- Specify the multiplicity of a class by writing a multiplicity expression in the upper-right corner of the class icon.
- Multiplicity applies to attributes, as well. You can specify the multiplicity of an attribute by writing a suitable expression in brackets just after the attribute name. For example, in the figure, there are two or more consolePort instances in the instance of NetworkController.



Attributes

- You can Specify the visibility, scope, and multiplicity of each attribute. There's still more. You can also
- specify the type, initial value, and changeability of each attribute.
- In its full form, the syntax of an attribute in the UML is
- **[visibility] name [multiplicity] [: type] [= initial-value] [{property-string}].**
-

There are three defined properties that you can use with attributes.

changeable	**There are no restrictions on modifying the attribute's value.**
addOnly	**For attributes with a multiplicity greater than one, additional values may be added, but once created, a value may not be removed or altered**
frozen	**The attribute's value may not be changed after the object is initialized.**

Operations

- you can also specify the visibility and scope of each operation.
- You can also specify the parameters, return type, concurrency semantics, and other properties of each operation.
- The name of an operation plus its parameters (including its return type, if any) is called the operation's signature.
- In its full form, the syntax of an operation in the UML is

[visibility] name [(parameter-list)] [: return-type] [{property-string}]

- In an operation's signature, you may provide zero or more parameters, each of which follows the syntax
- **[direction] name : type [= default-value]**
- Direction may be any of the following values

1) in 2) out 3) inout

Template Classes

- A template is a parameterized element. In such languages as C++ and Ada, you can write template classes, each of which defines a family of classes.
- A template includes slots for classes, objects, and values, and these slots serve as the template's parameters.
- The most common use of template classes is to specify containers that can be instantiated for specific elements, making them type-safe

```
template<class Item, class Value, int Buckets> class Map {
        public:
   virtual Boolean bind(const Item&, constValue&);
   virtual Boolean isBound(const Item&) const;
    ... };
```

Standard Elements - The UML defines four standard stereotypes that apply to classes.

1. metaclass - Specifies a classifier whose objects are all classes
2. powertype - Specifies a classifier whose objects are the children of a given parent
3. stereotype - Specifies that the classifier is a stereotype that may be applied to other elements
4. utility - Specifies a class whose attributes and operations are all class scoped

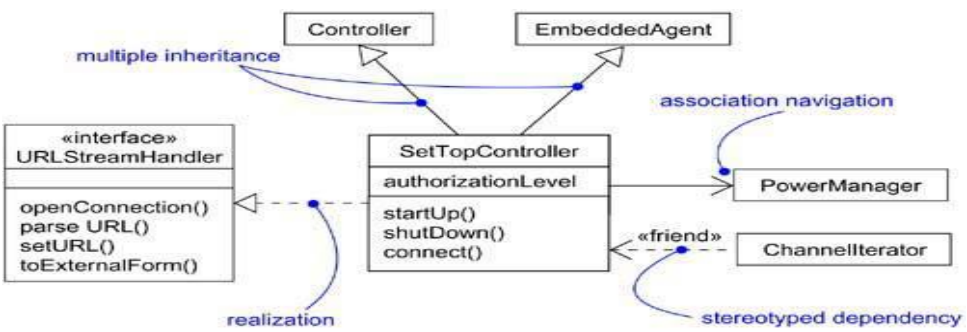
Common Modeling Techniques

1. Modeling semantics of class

- Specify the responsibilities of the class. A responsibility is a contract or obligation of a type or class and is rendered in a note (stereotyped as **responsibility**) attached to the class, or in an extra compartment in the class icon.
- Specify the semantics of the class as a whole using structured text, rendered in a note (stereotyped as **semantics**) attached to the class.
- Specify the body of each method using structured text or a programming language, rendered in a note attached to the operation by a dependency relationship.
- Specify the pre- and post conditions of each operation, plus the invariants of the class as a whole, using structured text. These elements are rendered in notes (stereotyped as **precondition, post condition,** and **invariant**) attached to the operation or class by a dependency relationship.
- Specify a state machine for the class. A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- Specify a collaboration that represents the class. A collaboration 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. A collaboration has a structural part, as well as a dynamic part, so you can use collaborations to specify all dimensions of a class's semantics.
- Specify the pre- and postconditions of each operation, plus the invariants of the class as a whole, using a formal language such as OCL.

Advanced Relationships

- 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, directed to the thing that is depended on.

Eight Stereotypes that apply to dependency relationships among classes and objects

bind	the source instatiates the target template
derive	the source may be computed from target
friend	the source is given special visibility into target
instanceOf	source object is an instance of the target classifier
instantiate	source object creates instance of the target
powertype	target is a powertype of the source
refine	source is at a finer degree of abstraction than target

use	the semantics of the source element depends on the semamtics of the public part of the target

- Two stereotypes that apply to dependency relationships among packages.
  - access – source package is granted the right to reference the elements of the target package.
  - import – a kind of access, but only public content.
- Two stereotypes that apply to dependency relationships among use case.
  - extend – target use case extends the behavior of source.
  - include – source use case explicitly incorporates the behavior of another use case at a location specified by the source
- Three stereotypes when modeling interactions among objects.
  - become – target is the same object of source at later time
  - call – source operation invoke the target operation
  - copy – target is an exact, but different, copy of source
- In the context of state machine
  - send – source operation sends the target event
- In the context of organizing the elements of your system into subsystem and model
  - trace – target is an historical ancestor of the source *(model relationship 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).

There is the one stereotype.

- **Implementation:** Specifies that the child inherits the implementation of the parent but does not make public nor support. Its interfaces, thereby violating substitutability

The four constraints that may be applied to generalization relationships.

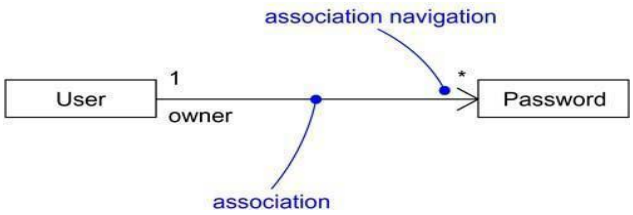
1.Complete - Specifies that all children in the generalization have been specified in the model and that no additional children are permitted
2.Incomplete - Specifies that not all children in the generalization have been specified and that additional children are permitted
3.Disjoint - Specifies that 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

Association

- An association is a structural relationship, specifying that objects of one thing are connected to object of another.
- Basic adornments: name, role, multiplicity, aggregation.
- Advanced adornments: navigation, qualification, various flavors of aggregation

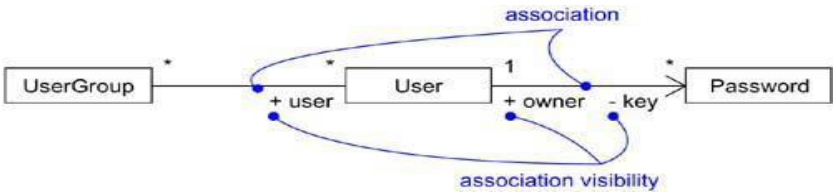
Navigation

- Given a plain, 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.

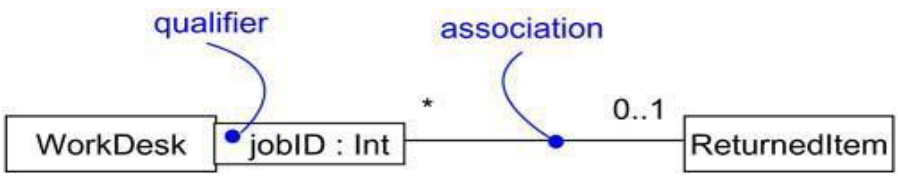


Visibility

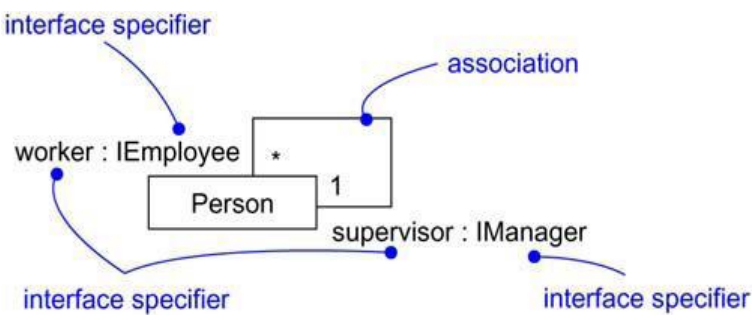
- An association between two classes, objects of one class can see and navigate to objects of the other, unless otherwise restricted by an explicit statement of navigation



Qualification: A form of aggregation with strong ownership and coincident lifetime of the parts by the whole.

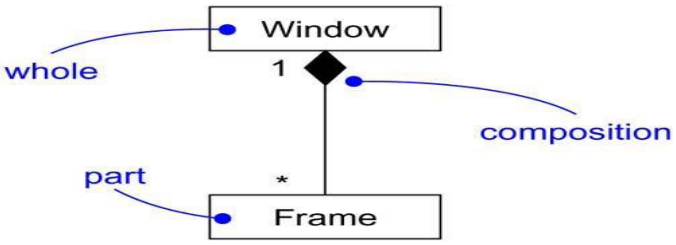


- **Interface specifier :** An interface is a collection of operations that are used to specify a service of a class or a component; every class may realize many interfaces.



Composition

- In a composite aggregation, an object may be a part of only one composite at a time.
- For example, in a windowing system, a Frame belongs to exactly one Window. 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.

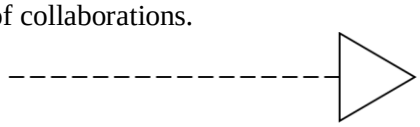


Constraints

1. implicit: The relationship is not manifest but, rather, is only conceptual.
2. ordered: the set of objects at one end of an association are in an explicit order.
3. changeable: links between objects may be changed.
4. add Only: new links may be added from an object on the opposite end of association.
5. frozen: a link added may not be modified or deleted.
6. Xor: over a set of associations, exactly one is man fest for each associated object.

Realization

- A realization is a semantic relationship between classifiers in which one classifier specifies a contract that anther classifier guarantees to carry out.
- Use in two circumstances:
  - In the context of interfaces.
  - In the context of collaborations.
  - Rendering as:



Common Modeling Techniques

1. Modeling Webs of Relationships

- 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, Types and Roles

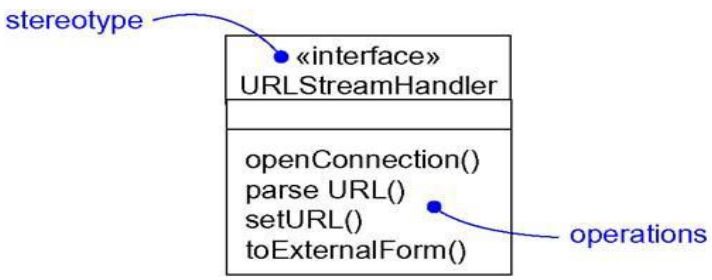
- An interface is a collection of operations that are used to specify a service of a class or a component. Graphically, an interface is rendered (represented) as a circle; in its expanded form, an interface may be rendered as a stereotyped class(a class with stereotype interface)

Names

- Every interface must have a name that distinguishes it from other interfaces
- Two naming mechanism:
- A simple name (only name of the interface).
- A path name is the interface name prefixed by the name of the package in which that interface lives represented.

Operations:

- To distinguish an interface from a class, prepend an 'I' to every interface name.
- Operations in an interface may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.
- Interface don't have attributes. interfaces span model boundaries and it doesn't have direct instances.



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.

Interface relationships

- An interface may participate in generalization, association, dependency and realization relationships. Realization is a semantic relationship between two classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.

Types and Roles

Type:

- A type is a stereotype of a class used to specify a domain of objects, together with the operations applicable to the object of that type.
- To distinguish a type from an interface or a class, prepend a 'T' to every type.

Role

- A role names(indicates) a behavior of an entity participating in a particular context. Or, 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.

Common Modeling Techniques

1. Modeling the Seams in a Systemeling the Seams in a System

- Within the collection of classes and components in your system, draw a line around those that tend to be tightly coupled relative to other sets of classes and components.
- Refine your grouping by considering the impact of change. Classes or components that tend to change together should be grouped together as collaborations.
- Consider the operations and the signals that cross these boundaries, from instances of one set of classes or components to instances of other sets of classes and components.
  Package logically related sets of these operations and signals as interfaces.
- For each such collaboration in your system, identify the interfaces it relies on (imports) and those it provides to others (exports). You model the importing of interfaces by dependency relationships, and you model the exporting of interfaces by realization relationships.
- For each such interface in your system, document its dynamics by using pre- and postconditions for each operation, and use cases and state machines for the interface as a whole.

2. Modeling Static and Dynamic Types

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 class of the object may take on at any point in time. You can do so in two ways:
  - ✓ 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.
  - ✓ 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.**
- For example, Figure shows the roles that instances of the class Person might play in the context of a human resources system.

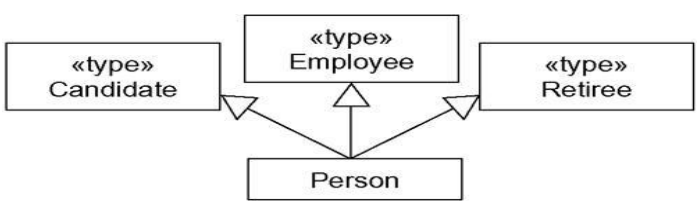
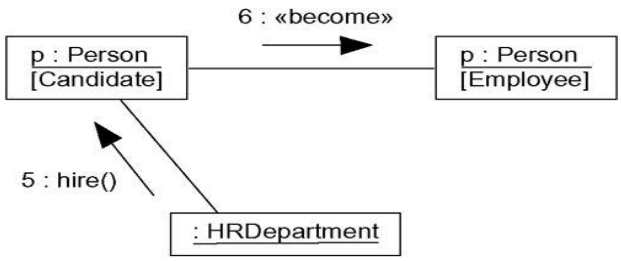


Fig: Modeling Static Types

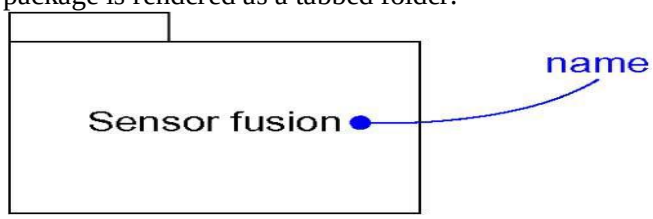
Figure shows the dynamic nature of a person's type. In this fragment of an interaction diagram, p (the Person object) changes its role from Candidate to Employee.

Fig: Modeling Dynamic Types



Packages

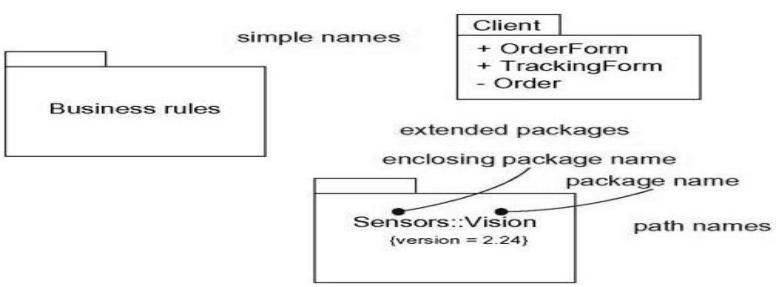
"**A package** is a general-purpose mechanism for organizing elements into groups." Graphically, a package is rendered as a tabbed folder.



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.

Fig: Simple and Extended Package



Visibility

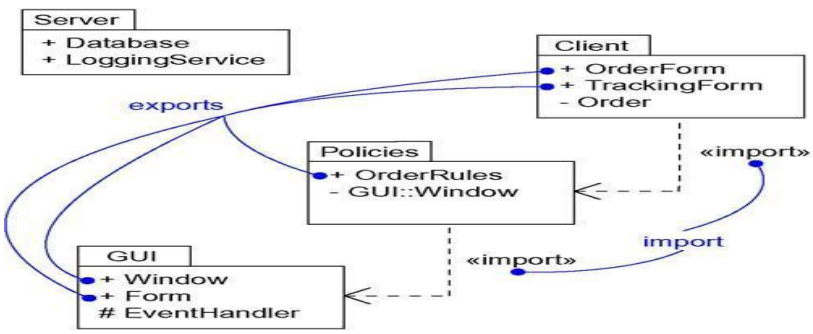
- You can control the visibility of the elements owned by a package just as you can control the visibility of the attributes and operations owned by a class.
- Typically, an element owned by a package is public, which means that it is visible to the contents of any package that imports the element's 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

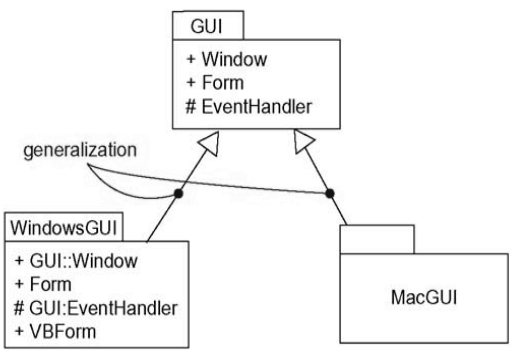
- Suppose you have two classes named **A** and **B** sitting side by side. Because they are peers, **A** can see **B** and **B** can see **A,** so both can depend on the other. Just two classes makes for a trivial system, so you really don't need any kind of packaging.
- In the UML, you model an import 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

Fig: 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.



- 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).

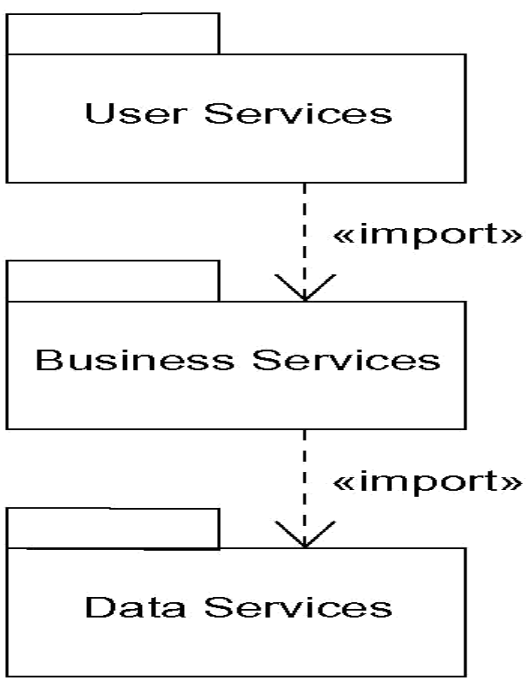
1. facade : : Specifies a package that is only a view on some other package
2. framework : Specifies a package consisting mainly of patterns
3. stub : Specifies a package that serves as a proxy for the public contents of another package
4. subsystem : Specifies a package representing an independent part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

Common Modeling Techniques

1. **Modeling 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

Fig: Modeling Groups of Elements

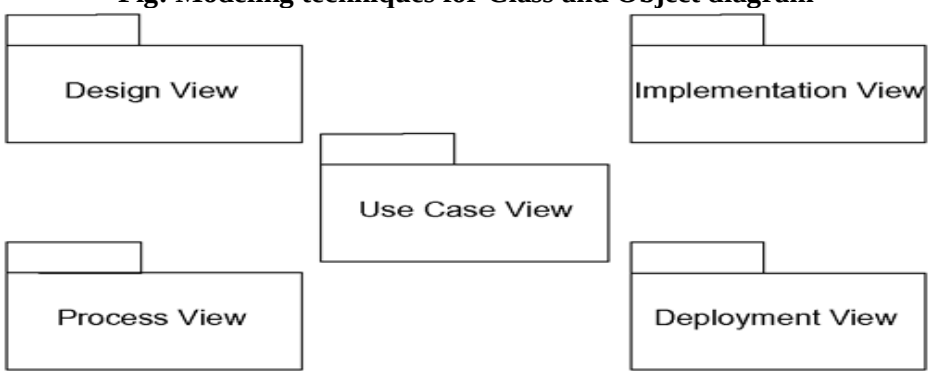


2. Modeling Architectural Views

- 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

Fig: Modeling Architectural Views

Fig: Modeling techniques for Class and Object diagram



Modeling techniques for Class and Object diagram

Class diagram

- It's a diagram that shows set of classes ,interfaces ,collaboration and either relationships .

Common properties

- It shows the same common properties as all other diagrams .

Contents

Class diagram contain the following things

1. Classes
2. Interfaces
3. Collaboration
4. Dependency ,Generalization, association

1. To model the vocabulary of a system

1. Modeling the vocabulary of a system involves making a decision about which abstractions
2. are a part of the system under consideration and which fall outside its boundaries. You use class
3. diagrams to specify these abstractions and their responsibilities.

2. Modeling simple collaboration

To model a collaboration .

1. Identify the mechanism you want to model .
2. For each mechanism identify the classes ,interfaces and collaboration .
3. Use scenarios to walk through these things .

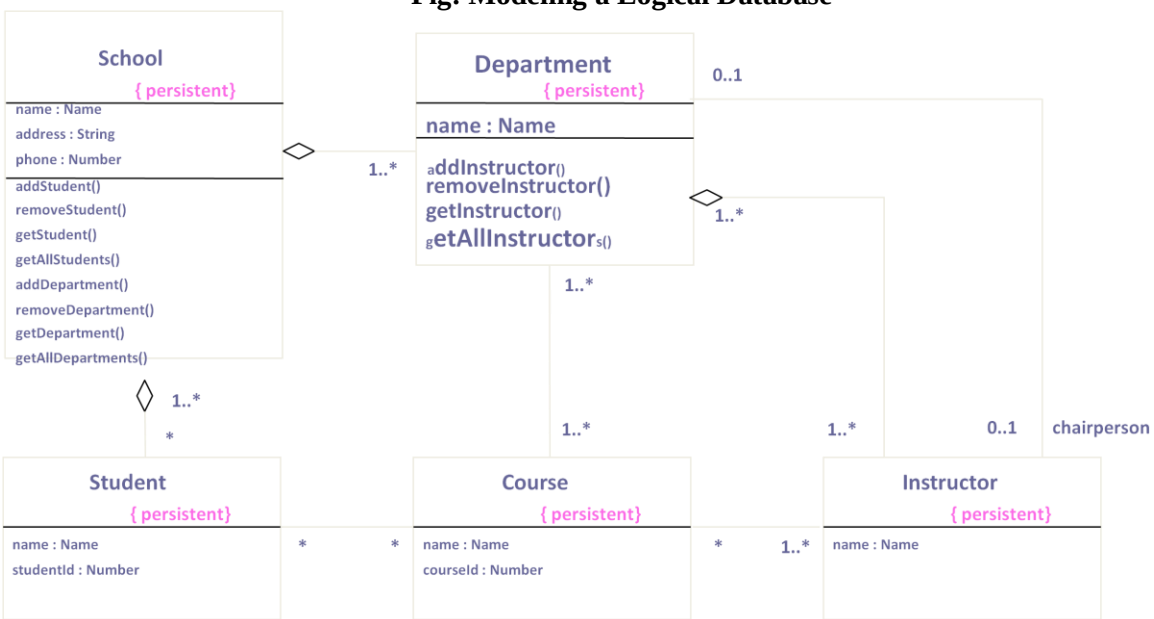
4. Be sure to populate these elements with their contents .

3. Modeling a logical database schema

To model a schema

1. Identify those classes in the model whose state must transcend the lifetime of their application.
2. Create class diagram that contain these classes and mark them as persistent .
3. Explain structural details of these classes .
4. Watch for common pattern that complicate physical database design .
5. Consider the behavior of these classes by expending operations .
6. Use tools to transform logical design to physical design .

Fig: Modeling a Logical Database



Forward Engineering

- It is the process of transforming a model into code through a mapping to an implementation language .

To forward engineer a class diagram

1) Identify the rules for mapping to your implantation language.
2) Depending upon the semantics of the language you have to constrain.
3) Use tagged values to specify your tagged values.
4) Use tools to forward engineer your models.

Reverse Engineering

-transforming code to uml model.

To reverse engineer a class diagram

1) Identify the rules for mapping from your language.
2) Use tools point to code you would like to reverse engineer.
3) Use tool, create a class diagram by querying the model

Object diagram

- It shows set of objects and their relationships at a point in time .
- Object diagrams are used to model the static design view or static process view of a system.

- 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.
- 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.*

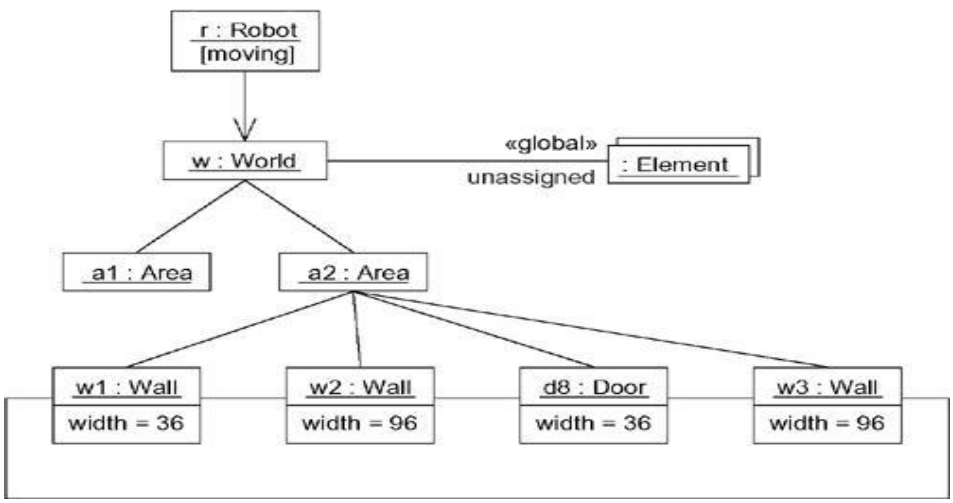
Contents: commonly it contains

1. objects
2. links
3. Object diagram contains notes and constraints .

1. Modeling object structures

To model object structures

1) Identify the mechanism you would like to model.
2) For each mechanism, identify classes, interfaces, other elements.
3) Consider one scenario that work through this mechanism .
4) Expose the state and attribute value of each such object to understand .
5) Similarly expose the links, instances, associations among them



Forward and Reverse Engineer

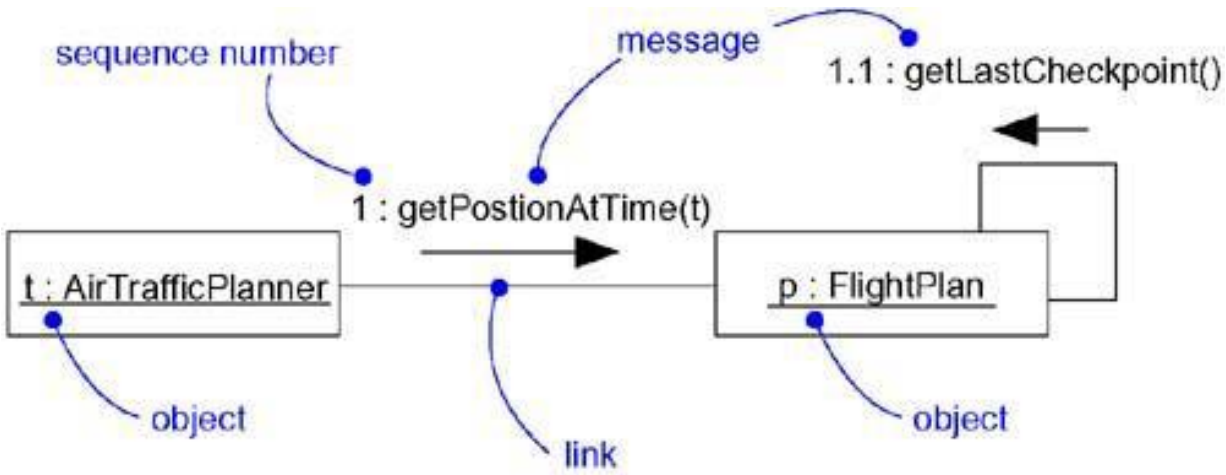
To reverse engineer an object diagram

1) Choose the target you want to reverse engineer.
2) Use tool or simply walkthrough a scenario.
3) Identify the set of objects that collaborate in the context.
4) As necessary to understand their semantics expose these objects.
5) Identify links among objects.
6) If your diagrams end up complicated, prune it by eliminating objects that are not germane.

INTERACTIONS

An *interaction* is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose. A *message* is a specification of a communication between objects that conveys information with the expectation that activity will ensue.

Fig: Messages, Links, and Sequencing



Context

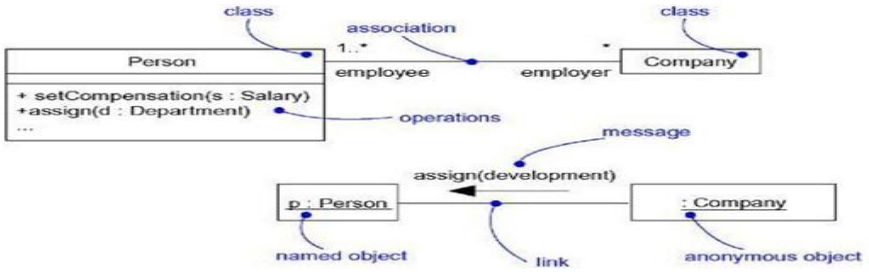
- Interaction can find wherever objects are linked to one another.
- Interaction can find in the collaboration of objects that exist in the context of your system or subsystem.
- It also find interactions in the context of an operation.
- Finally, you'll find interactions in the context of a class.

Object and Roles:

- The objects that participate in an interaction are either concrete things or prototypical things.
- A concrete thing, an object represents something in the real world. For example, p, an instance of the class Person, might denote a particular human.
- A prototypical thing, p might represent any instance of Person.

Links

- A link is a semantic connection among objects.
- In general, a link is an instance of an association.
- Following fig. shows, wherever a class has an association to another class, there may be a link between the instances of the two classes; wherever there is a link between two objects, one object can send a message to the other object.



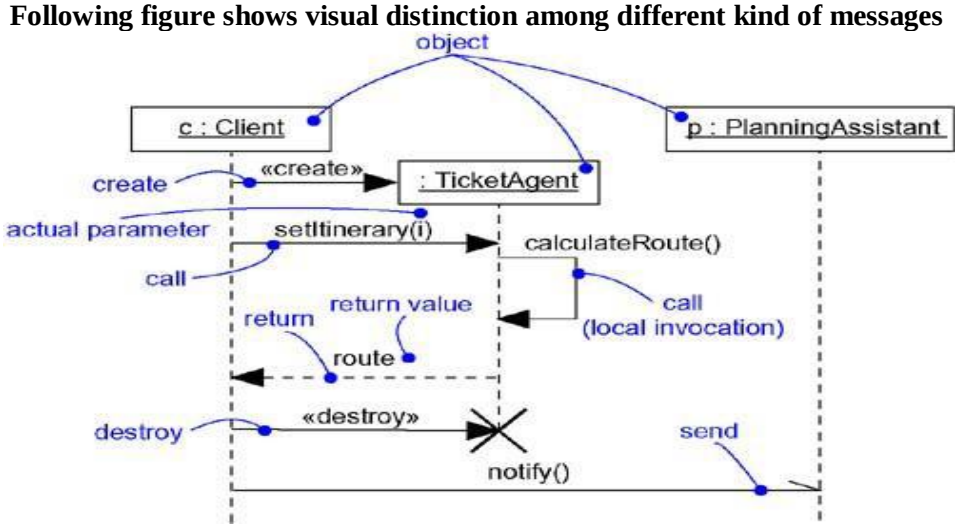
Following five standard stereotypes you can use

- association – corresponding object is visible by association.
- self – dispatches of operation.
- global – represents enclosing scope.
- local – local scope
- parameter – parameter visibility.

Messages

- A message is the specification of a communication among objects that conveys information with the expectation that activity will ensue.
- The receipt of a message instance may be considered an instance of an event.
- When you pass a message, the action that results is an executable statement that forms an abstraction of a computational procedure.
- An action may result in a change in state.
- **UML can model several kind of actions**:
- **call** - invoke an operation **Return** - return a value to the caller
- **Send** - send signal to an object **Create** - creates an object
- **Destroy** - destroys an object

Following figure shows visual distinction among different kind of messages



Sequencing

- When an object passes a message to another object (in effect, delegating some action to the receiver), the receiving object might in turn send a message to another object, which might send a message to yet a different object, and so on. This stream of messages forms a sequence. Any sequence must have a beginning.
- Most commonly, you can specify a procedural or nested flow of control, rendered using a filled solid arrowhead, as Figure shows. In this case, the message findAt is specified as the first message nested in the second message of the sequence (2.1).

Procedural Sequence

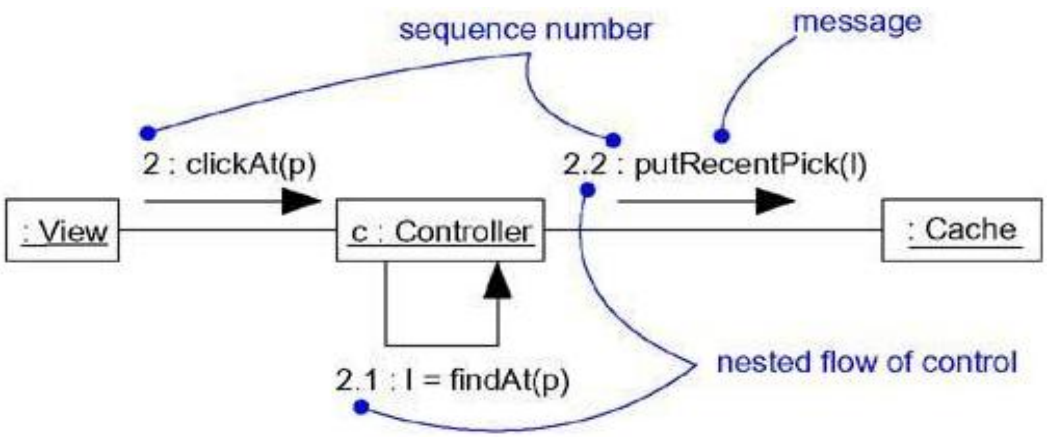
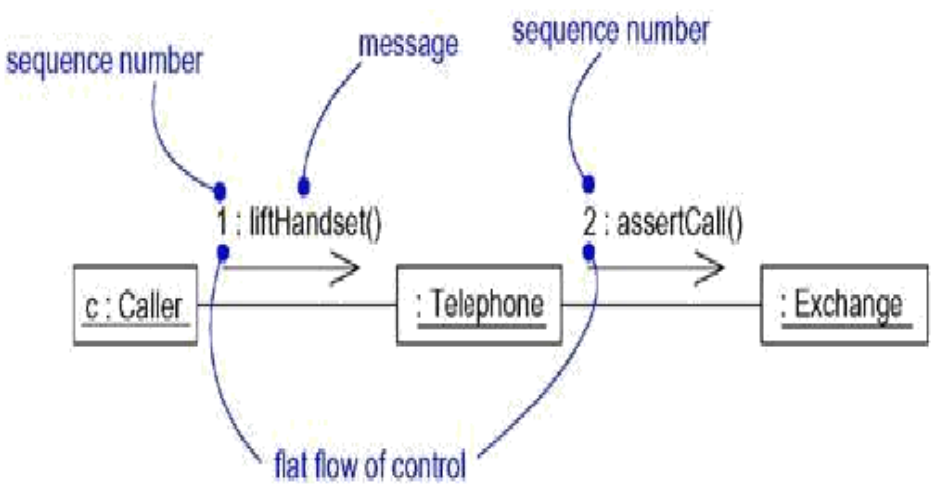


Figure below shows, a flat flow of control, rendered using a stick arrowhead, to model the nonprocedural progression of control from step to step.
In this case, the message assertCall is specified as the second message in the sequence.



Creation , Modification and Destruction :

- To specify if an object or link enters and/or leaves during an interaction you can attach one of 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

- When you model an interaction, you typically include both objects (each one playing a specific role) and messages (each one representing the communication between objects, with some resulting action).
- You can visualize those objects and messages involved in an interaction in two ways:
  1. by emphasizing the time ordering of its messages
  2. by emphasizing the structural organization of the objects that send and receive messages.

- In the UML, the first kind of representation is called a sequence diagram; the second kind of representation is called a collaboration diagram.
- Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams.

Common Modeling Techniques

1. Modeling a flow 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 your 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. Specify the nature of the links using the UML's standard stereotypes and constraints, as necessary.

Modeling a flow control contd..

- 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.

This figure is an example of a sequence diagram, which emphasizes the time order of messages.

Eg. Flow of control by time

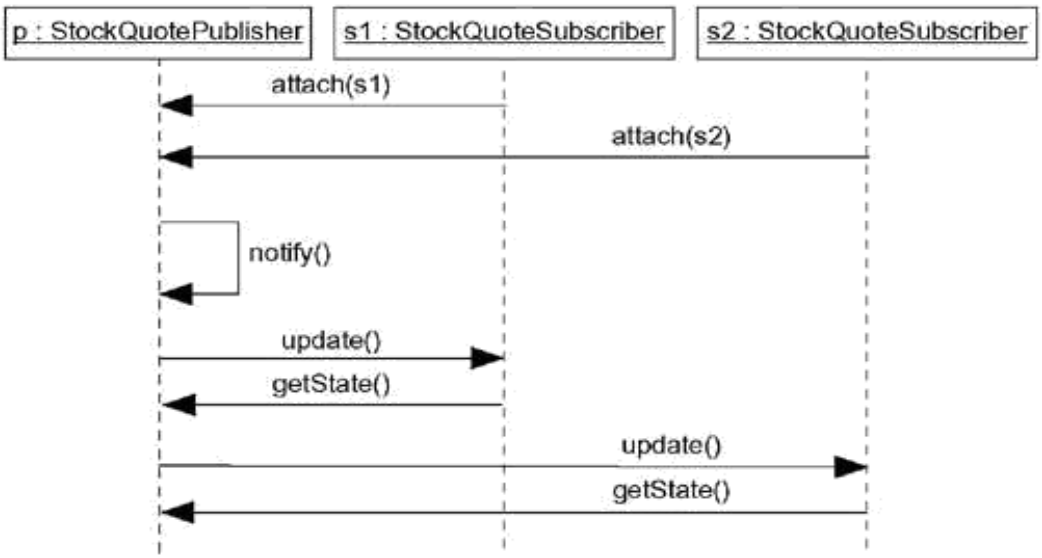
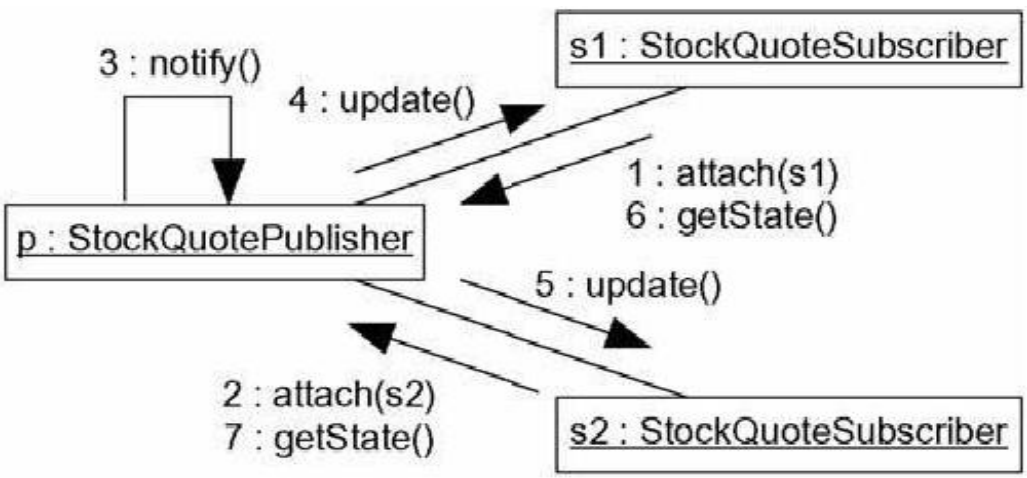


Figure is semantically equivalent to the previous one but it is drawn as a collaboration diagram, which emphasizes the structural organization of the objects. This figure shows the same flow of control, but it also provides a visualization of the links among these objects. **Eg. Flow of control by organization**



INTERACTION DIAGRAMS

- Interaction diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering.
- An *interaction diagram shows an interaction, consisting of a set of objects and their relationships,* including the messages that may be dispatched among them.
- A ***sequence diagram*** is an interaction diagram that emphasizes the time ordering of messages.
- A **C*ollaboration diagram*** is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.

Common Properties

- An interaction diagram is just a special kind of diagram and shares the same common properties as do all other diagrams
- A name and graphical contents that are a projection into a model.

Contents

Interaction diagrams commonly contain

- Objects
- Links
- Messages

Sequence Diagrams

- Describe the flow of messages, events, actions between objects
- Show concurrent processes and activations
- Show time sequences that are not easily depicted in other diagrams
- Typically used during analysis and design to document and understand the logical flow of your system.
- A sequence diagram emphasizes the time ordering messages
- **Sequence Diagram Key Parts**
- **participant:** object or entity that acts in the diagram
- – diagram starts with an unattached "found message" arrow
- **message:** communication between participant objects
- the **axes** in a sequence diagram:
- – horizontal: which object/participant is acting
- – vertical: time (down -> forward in time)

SEQUENCE DIAGRAM

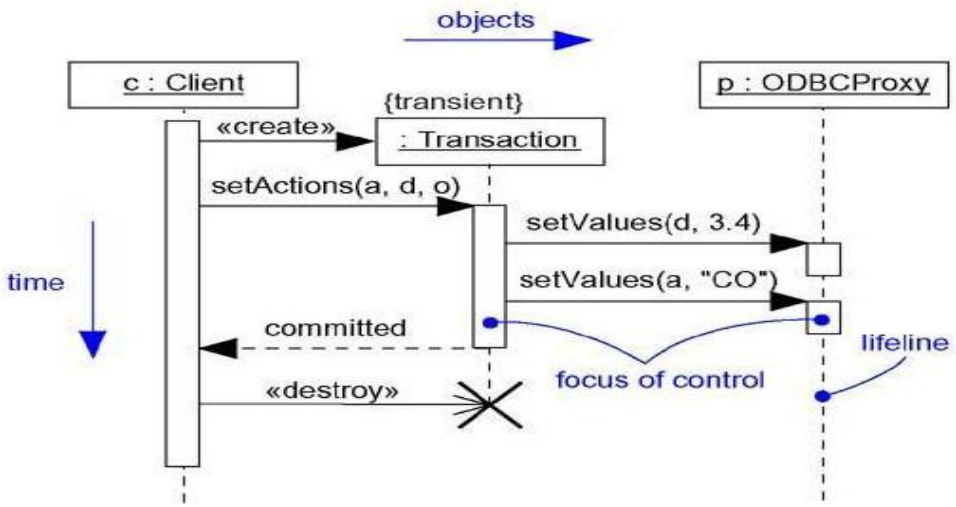
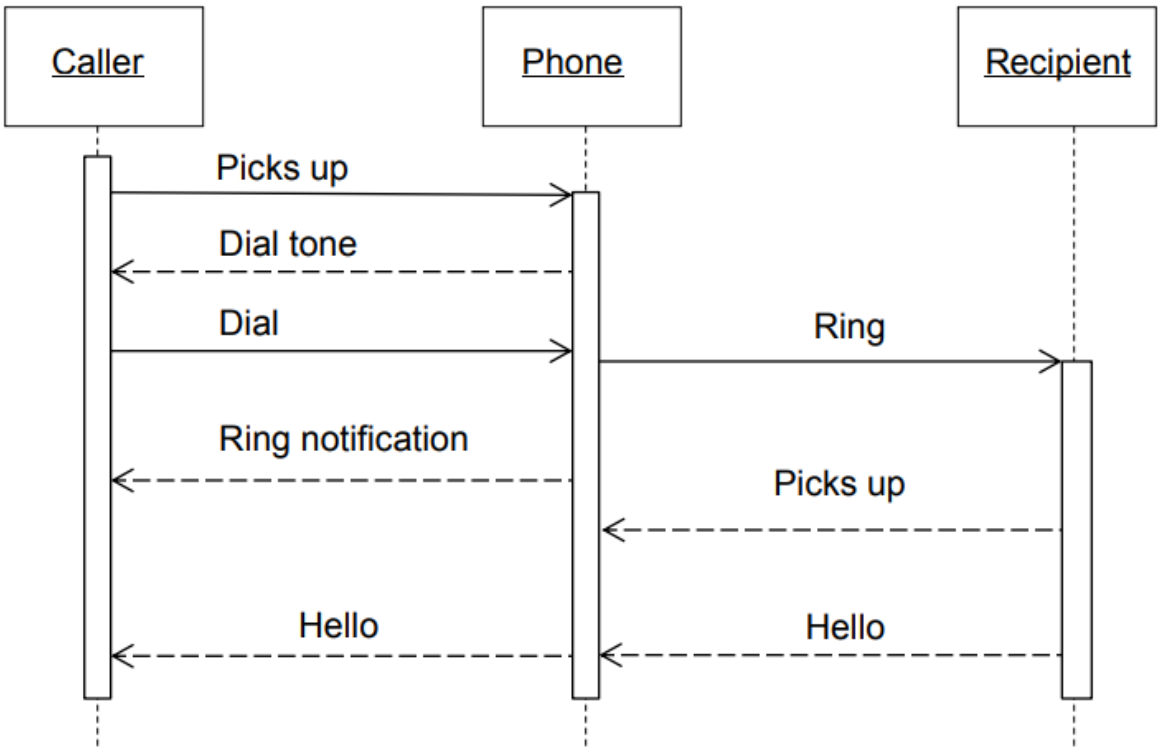


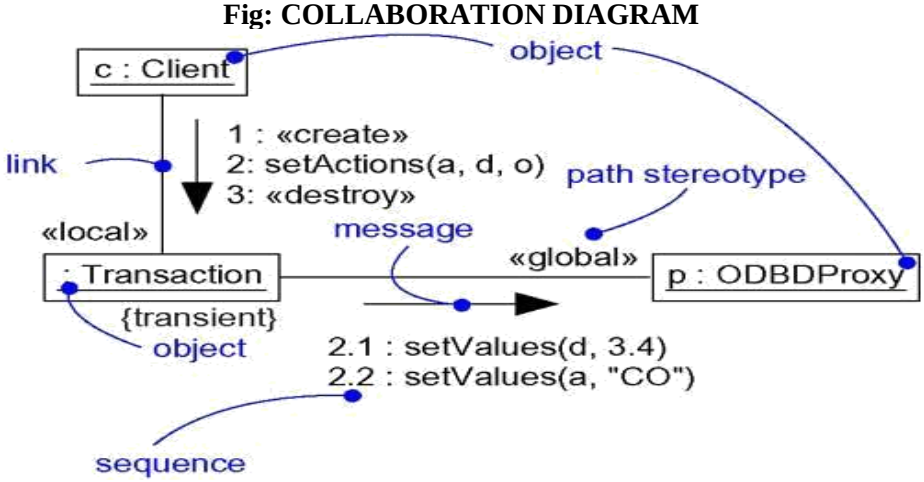
Fig: Sequence Diagram (make a phone call)



Collaboration Diagrams

- A collaboration diagram emphasizes the organization of the objects that participate in an interaction.
- In the collaboration diagram, the method call sequence is indicated by some numbering technique.
- The number indicates how the methods are called one after another.
- 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, you can attach a path stereotype to the far end of a link (such as <<local>>, indicating that the designated object is local to the sender).
- Second, there is the sequence number. To indicate the time order of a message, you prefix the message with a number (starting with the message numbered 1), increasing monotonically for each new message in the flow of control (2, 3, and so on).

Fig: COLLABORATION DIAGRAM

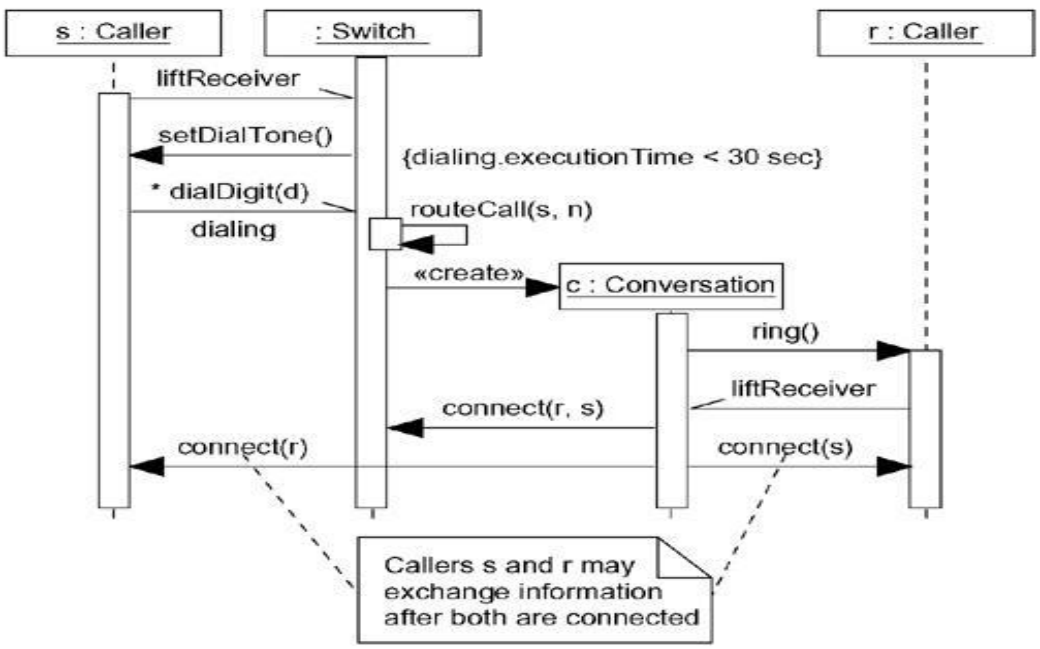


Common Modeling Techniques

1. Modeling flow 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. Lay them out on the sequence diagram from left to right, placing the more important objects to the left and their neighboring objects to the right.
- Set the lifeline for each object. In most cases, 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 (such as its parameters), as necessary to explain the semantics of the interaction.
- 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- and postconditions to each message.

Eg. Modeling Flows of Control by Time Ordering



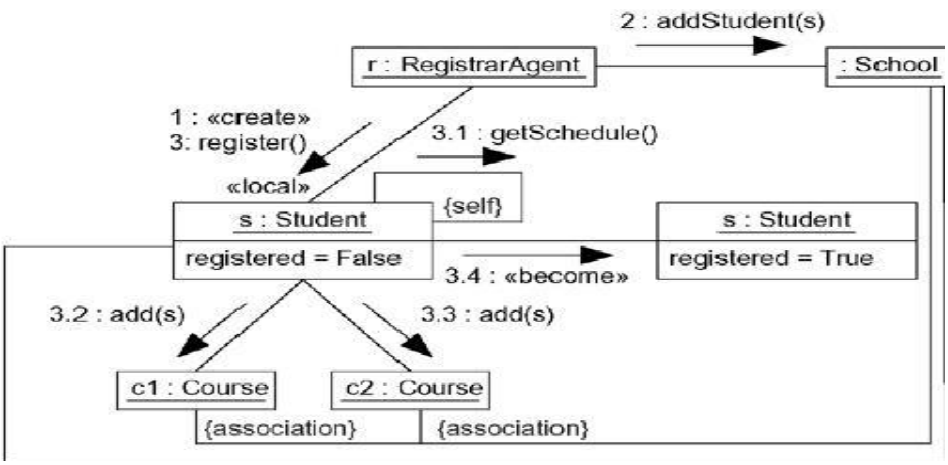
2. Modeling Flows of control by organization

- 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. Lay them out on the collaboration diagram as vertices in a graph, placing the more important objects in the center of the diagram and their neighboring objects to the outside.
- 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** (with a suitable sequence number).

- Specify the links among these objects, along which messages may pass.
  - Lay out the association links first; these are the most important ones, because they represent structural connections.
  - 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.

Fig: Modeling Flows of control by organization



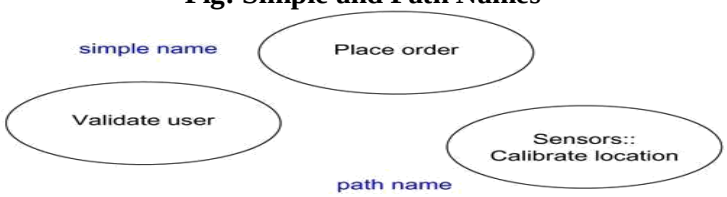
Use Cases

- A *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.
- Graphically, a use case is rendered as an ellipse.

Names

- Every use case must have a name that distinguishes it from other use cases. A *name* is a textual string.
- name alone is known as a *simple name;* a *path name* is the use case name prefixed by the name of the package in which that use case lives.
- A use case is typically drawn showing only its name.

Fig: Simple and Path Names



Note

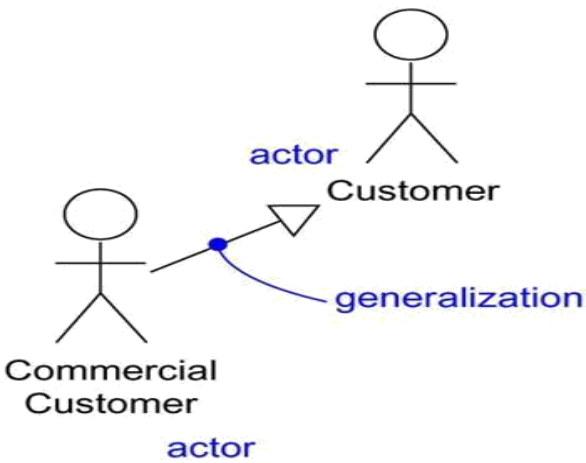
- A use case name may be text consisting of any number of letters, numbers, and most punctuation marks and may continue over several lines

Use Cases and Actors

- An 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 eve another system plays with a system.

As Figure indicates, actors are rendered as stick figures. You can define general kinds of actors (such as Customer) and specialize them (such as CommercialCustomer) using generalization relationships.

Fig: Actors



Use Cases and Flow of Events

A use case describes *what* a system (or a subsystem, class, or interface) does but it does not specify *how* it does it. When you model, it's important that you keep clear the separation of concerns between this outside and inside view.

For example, in the context of an ATM system, you might describe the use case ValidateUser in the following way:

Main flow of events:

- The use case starts when the system prompts the *Customer* for a PIN number. The *Customer* can now enter a PIN number via the keypad. The *Customer* commits the entry by pressing the Enter button. The system then checks this PIN number to see if it is valid. If the PIN number is valid, the system acknowledges the entry, thus ending the use case.

Exceptional flow of events:

- The *Customer* can cancel a transaction at any time by pressing the Cancel button, thus restarting the use case. No changes are made to the *Customer*'s account.

Exceptional flow of events:

- The *Customer* can clear a PIN number anytime before committing it and reenter a new PIN number.

Exceptional flow of events:

- If the *Customer* enters an invalid PIN number, the use case restarts. If this happens three times in a row, the system cancels the entire transaction, preventing the *Customer* from interacting with the ATM for 60 seconds.

Use Cases and Scenarios

A Scenario is a specific sequence of actions that illustrates behavior.

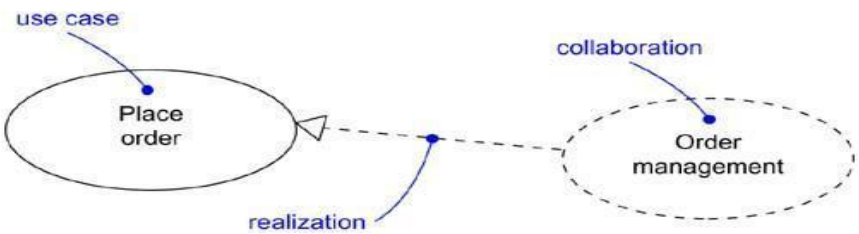
- Scenarios are to use cases as instances are to classes, meaning that a scenario is basically one instance of a use case.

Use Cases and Collaborations

- A use case captures the intended behavior of the system (or subsystem, class, or interface) you are developing, without having to specify how that behavior is implemented.
- That's an important separation because the analysis of a system (which specifies behavior) should, as much as possible, not be influenced by implementation issues (which specify how that behavior is to be carried out)

As Figure shows, you can explicitly specify the realization of a use case by a collaboration.

Fig: Use Cases and Collaborations

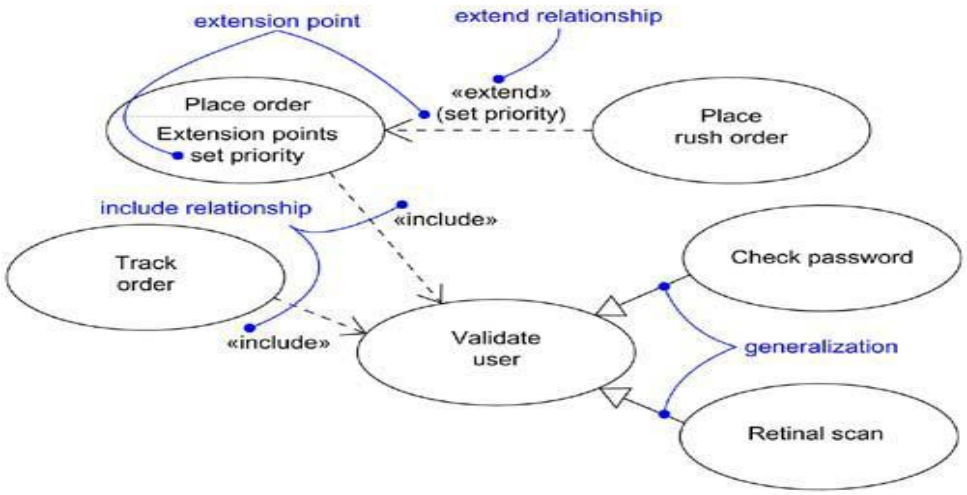


Organizing Use Cases

It is also possible to organize use cases by specifying generalization, include, and extend relationships among them.

- Apply these relationships in order to factor common behavior (by pulling such behavior from other use cases that it includes) and in order to factor variants (by pushing such behavior into other use cases that extend it).
- 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 **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.

Fig: Generalization, Include, and Extend



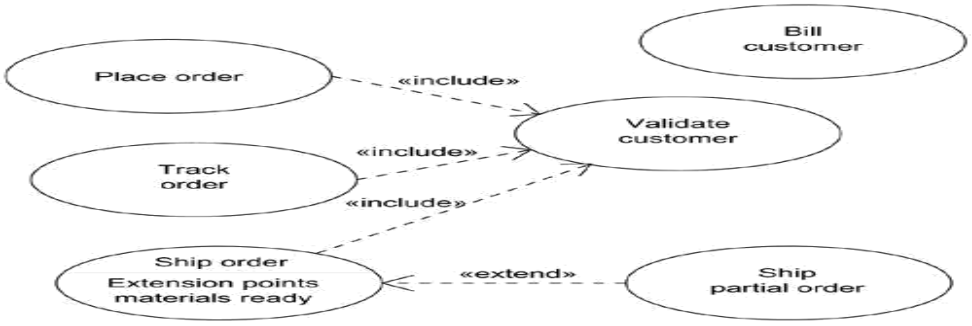
Common Modeling Techniques

1. 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 relationships to factor common behavior and distinguish exceptional behavior

Fig: 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.

Common Properties

- A use case diagram is just a special kind of diagram and shares the same common properties as do all other diagrams.
- a name and graphical contents that are a projection into a model.

Contents

Use case diagrams commonly contain

- Use cases
- Actors
- Dependency, generalization, and association relationships

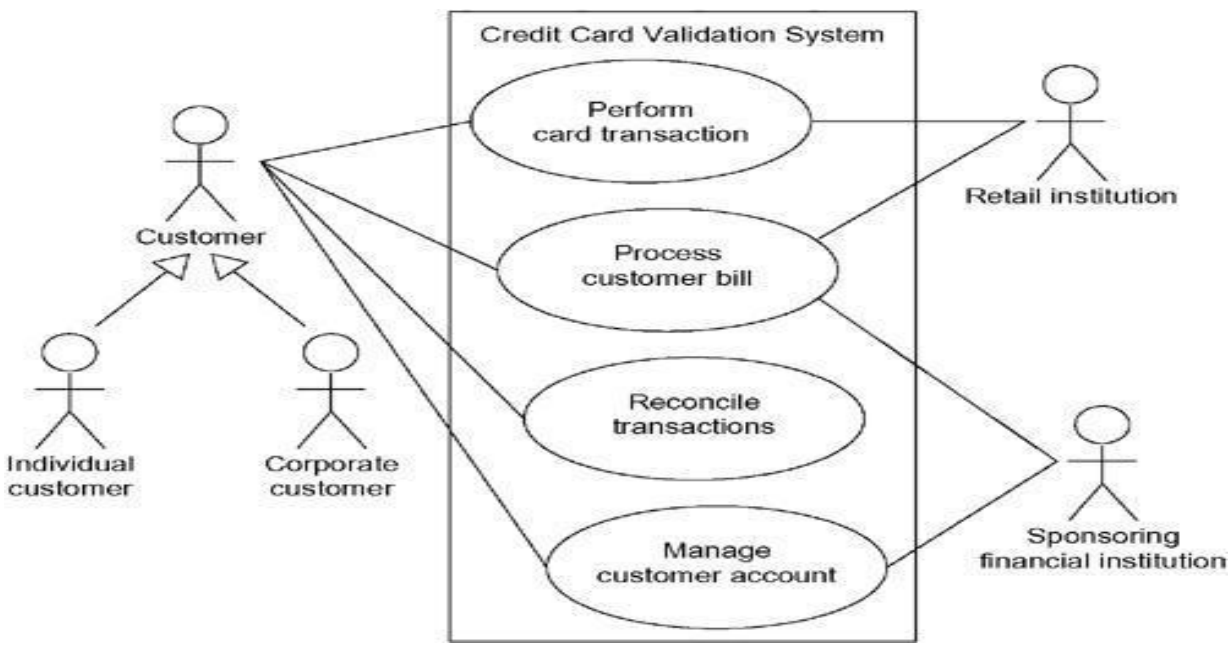
Like all other diagrams, use case diagrams may contain notes and constraints.

Common Modeling Techniques

1. 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.
- Where it aids understandability, 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.
- **Fig: Modeling the Context of a System – Credit card validation system**

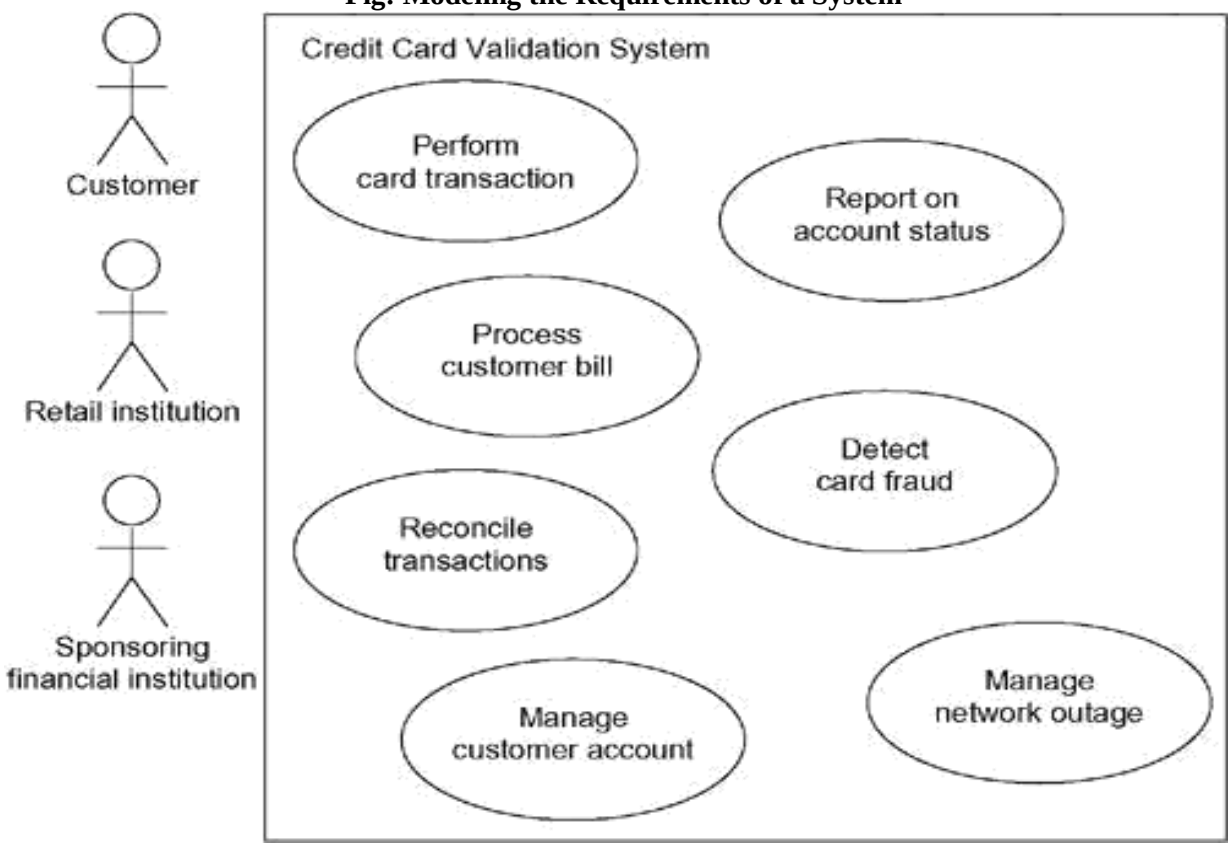


2. Modeling the Requirements of a System

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.

Fig: Modeling the Requirements of a System



3. Forward and Reverse Engineering

Forward engineering is the process of transforming a model into code through a mapping to an implementation language

To forward engineer a use case diagram

- For each use case in the diagram, identify its flow of events and its exceptional flow of events.
- Depending on how deeply you choose to test, generate a test script for each flow, using the flow's preconditions as the test's initial state and its postconditions as its success criteria.
- As necessary, generate test scaffolding to represent each actor that interacts with the use case. Actors that push information to the element or are acted on by the element may either be simulated or substituted by its real-world equivalent.

- Use tools to run these tests each time you release the element to which the use case diagram applies.

To reverse engineer a use case diagram,

- Identify each actor that interacts with the system.
- For each actor, consider the manner in which that actor interacts with the system, changes the state of the system or its environment, or responds to some event.
- Trace the flow of events in the executable system relative to each actor. Start with primary flows and only later consider alternative paths.
- Cluster related flows by declaring a corresponding use case. Consider modeling variants using extend relationships, and consider modeling common flows by applying include relationships.
- Render these actors and use cases in a use case diagram, and establish their relationships.

Activity Diagrams

- Activity diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems.
- An *activity diagram* shows the flow from activity to activity.
- Activity diagrams can use to model the dynamic aspects of a system. It involves modeling the sequential (and possibly concurrent) steps in a computational process.
- With an activity diagram, you can also model the flow of an object as it moves from state to state at different points in the flow of control.

Activity diagrams commonly contain

- Activity states and action states
- Transitions
- Objects

Action States and Activity States

- Action states are atomic and cannot be decomposed
  - meaning that events may occur, but the work of the action state is not interrupted. Finally, the work of an action state is generally considered to take insignificant execution time.Work of the action state is not interrupted.
- Activity states can be further decomposed
  - Their activity being represented by other activity diagrams
  - Activity states are not atomic, meaning that they may be interrupted and, in general, are considered to take some duration to complete. They may be interrupted.

Fig: Action States

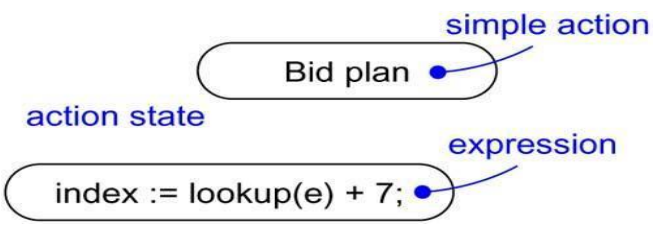
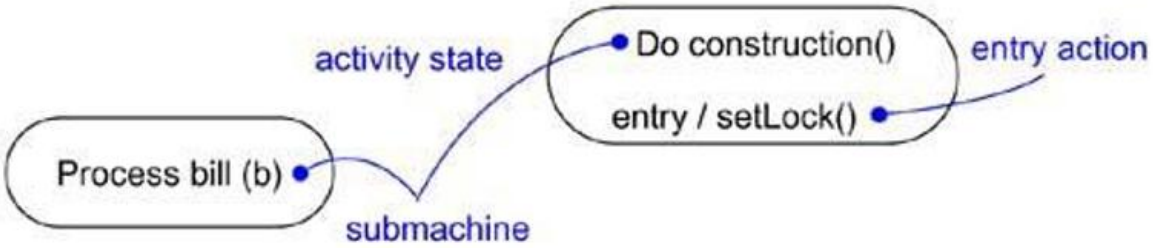
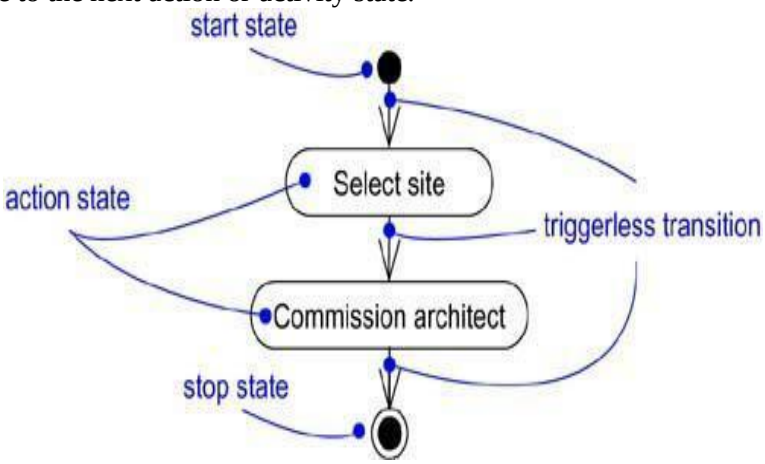


Fig: Activity States



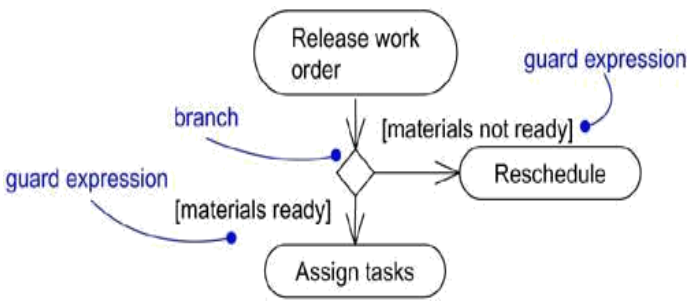
Transitions

- Triggerless transitions may have guard conditions, meaning that such a transition will fire only if that condition is met; guard conditions.
- When the action or activity of a state completes, flow of control passes immediately to the next action or activity state. You specify this flow by using transitions to show the path from one action or activity state to the next action or activity state.



Branching

- *Branches are a notational convenience, semantically equivalent to multiple transitions with guards.*
- *Include a branch, which specifies alternate paths taken based on some Boolean expression.*
- *A branch may have one incoming transition and two or more outgoing ones.*
- *On each outgoing transition, place a Boolean expression, which is evaluated only once on entering the branch.*



Forking and Joining

- Use a synchronization bar to specify the forking and joining of parallel flows of control
- A synchronization bar is rendered as a thick horizontal or vertical line.

Fork

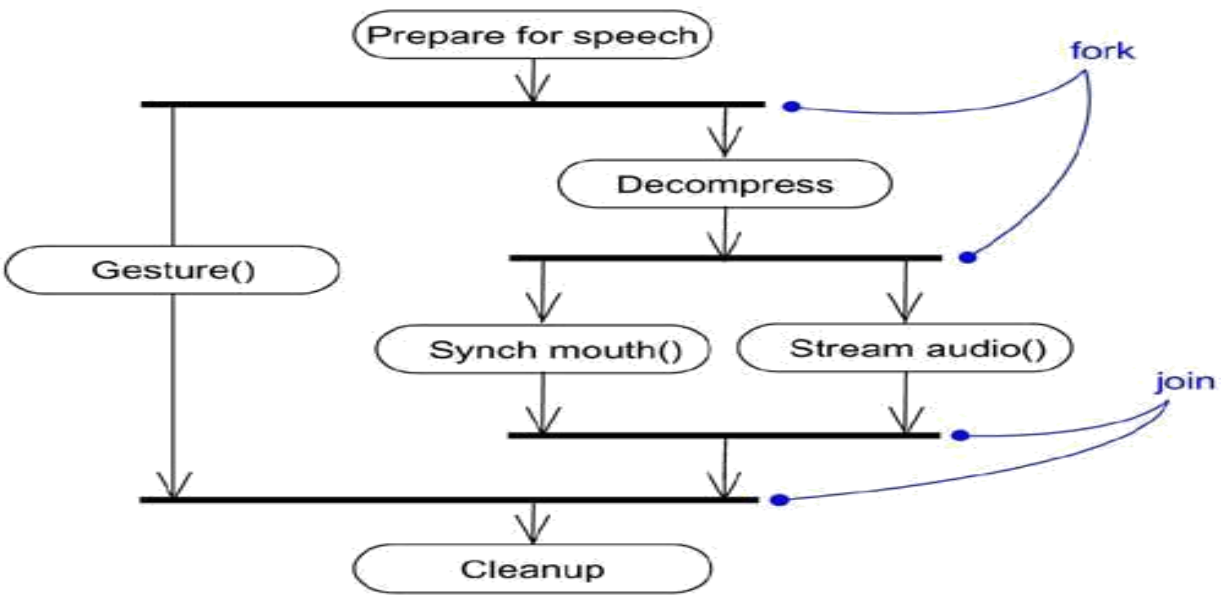
- A fork may have one incoming transitions and two or more outgoing transitions
  - each transition represents an independent flow of control
  - conceptually, the activities of each of outgoing transitions are concurrent
    - either truly concurrent (multiple nodes)
    - or sequential yet interleaved (one node)

Join

- 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
    - each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join

For example, consider the concurrent flows involved in controlling an audio-animatronic device that mimics human speech and gestures.

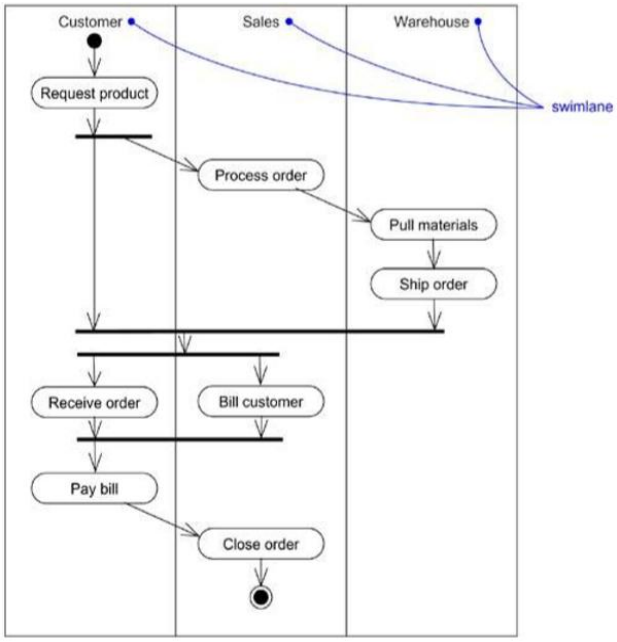
Figure - Forking and Joining



Swimlanes

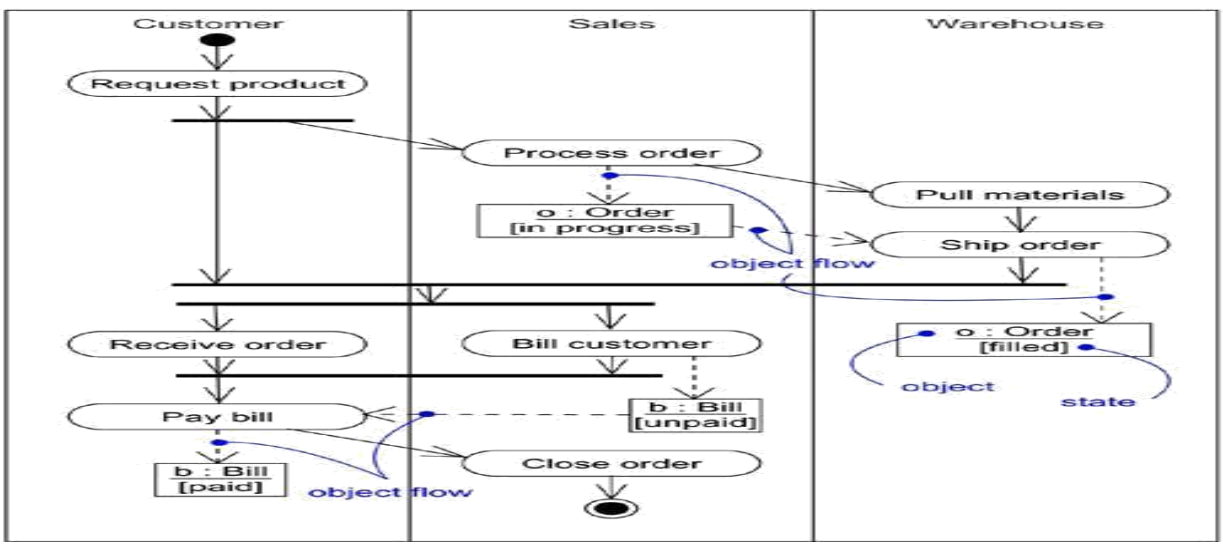
- In the UML, each group is called a swimlane because, visually, each group is divided from its neighbor by a vertical solid line.
- A swimlane specifies a locus of activities.
- Each swimlane has a name unique within its diagram. A swimlane really has no deep semantics, except that it may represent some real-world entity.
- Each swimlane represents a high-level responsibility for part of the overall activity of an activity diagram.
- swimlane may eventually be implemented by one or more classes. In an activity diagram partitioned into swimlanes, every activity belongs to exactly one swimlane, but transitions may cross lanes.

Fig: Swimlanes



Object Flow

In addition to showing the flow of an object through an activity diagram, you can also show how its role, state and attribute values change. As shown in the figure, you represent the state of an object by naming its state in brackets below the object's name. Similarly, you can represent the value of an object's attributes by rendering them in a compartment below the object's name.

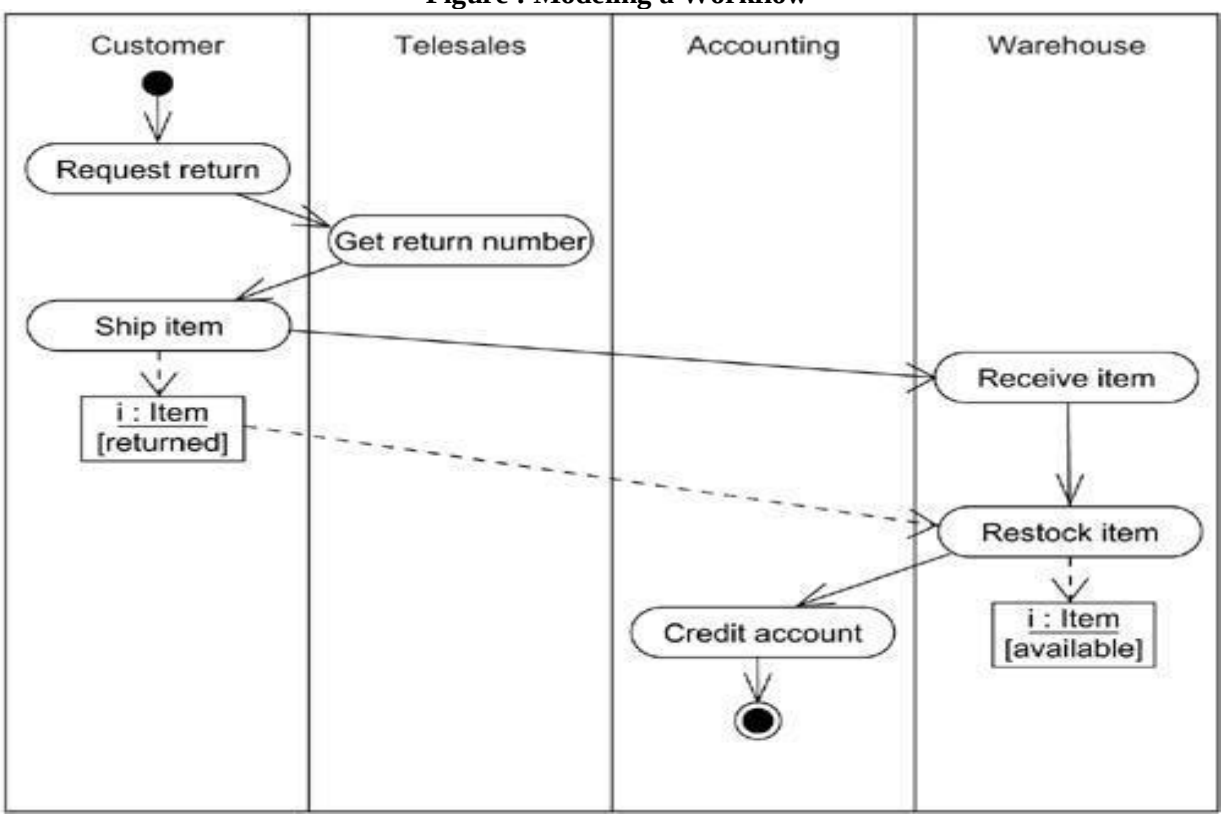


Common Modeling Techniques

1. 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.

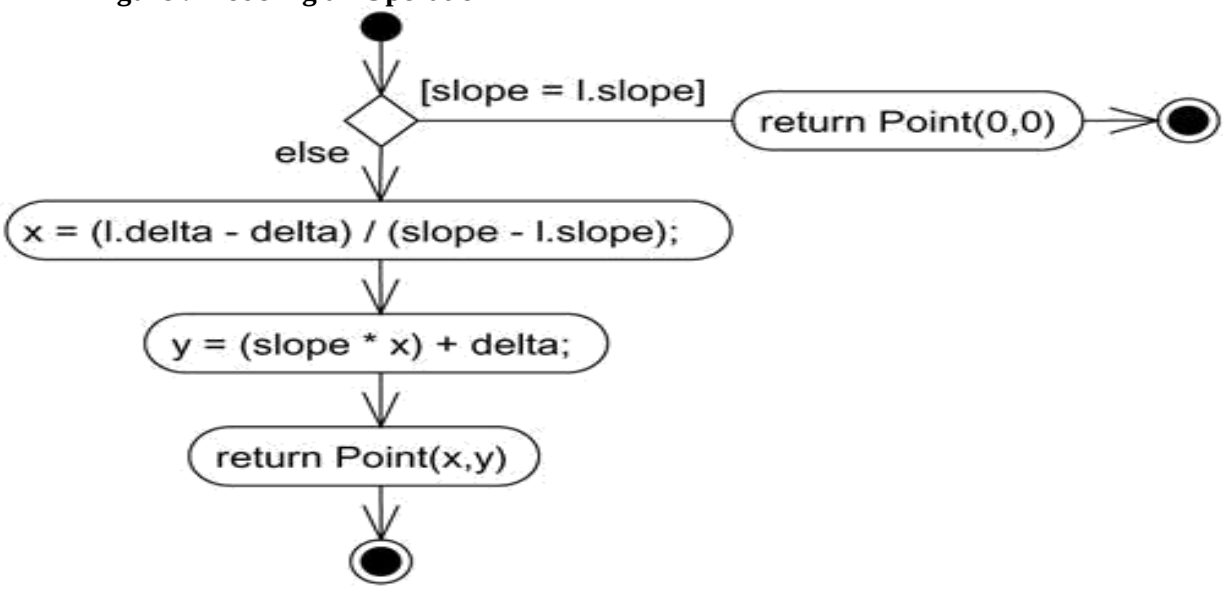
Figure : Modeling a Workflow



2. 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 postconditions 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 : Modeling an Operation



UNIT-III

Events and signals, state machines, processes and threads, time and space. State chart diagrams, component diagrams, deployment diagrams.

Events and signals

- An *event* is the specification of a significant occurrence that has a location in time and space.
- In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition.
- A *signal* is a kind of event that represents the specification of an asynchronous stimulus communicated between instances.

Kinds of Events

- Events may be external or internal. External events are those that pass between the system and its actors.
- For example, the pushing of a button and an interrupt from a collision sensor are both examples of external events.
- Internal events are those that pass among the objects that live inside the system. An overflow exception is an example of an internal event.

Fig: Event

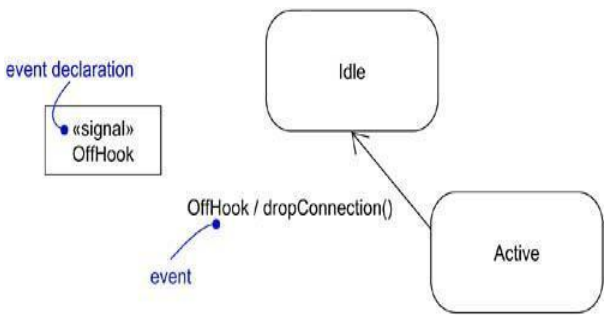
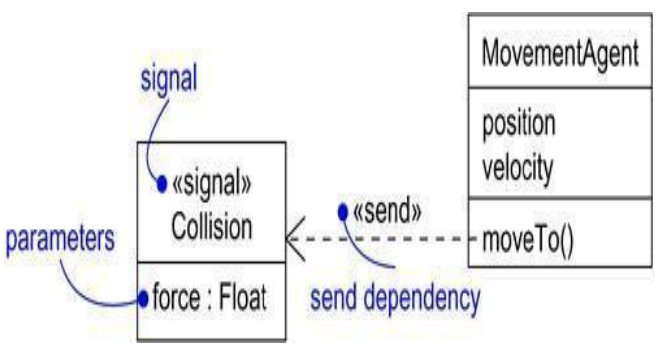


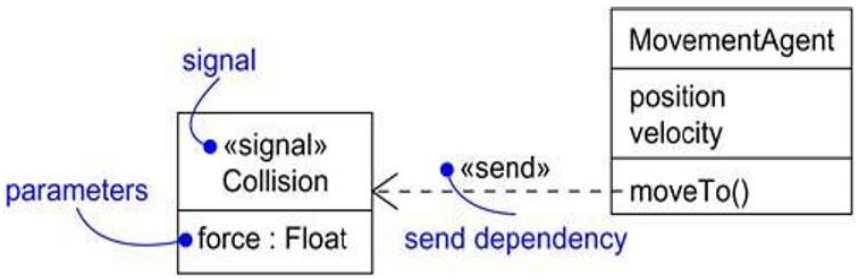
Fig: Signal



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.

Fig: Signals



2. Call Events

- Just as a signal event represents the occurrence of a signal, a call event represents the dispatch of an operation.
- Whereas a signal is an asynchronous event, a call event is, in general, synchronous.
- It means when an object invokes an operation on another object that has a state machine, control passes from the sender to the receiver, the transition is triggered by the event, the operation is completed, the receiver transitions to a new state, and control returns to the sender.

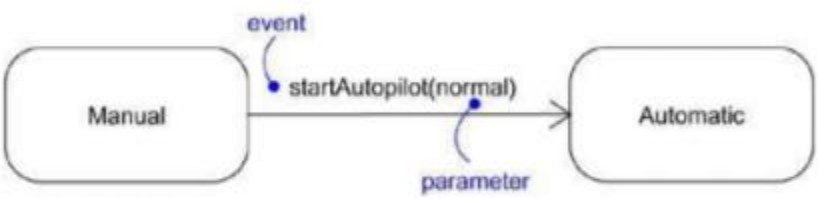
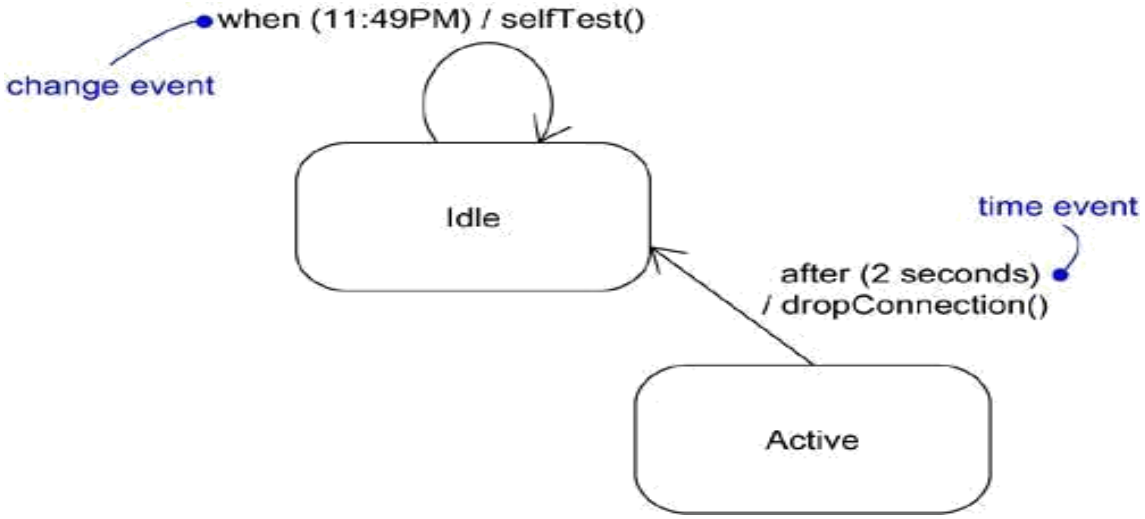


Figure: Call Events

3. Time and Change Events

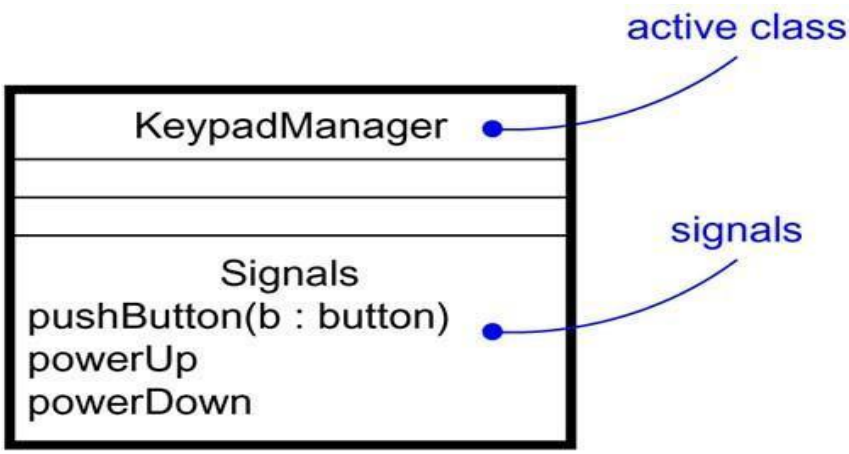
- A time event is an event that represents the passage of time. In the UML you model a time event by using the keyword **after** followed by some expression that evaluates to a period of time.
- A change event is an event that represents a change in state or the satisfaction of some condition. In the UML you model a change event by using the keyword **when** followed by some Boolean expression.



4. Sending and Receiving Events

- Signal events and call events involve at least two objects: the object that sends the signal or invokes the operation, and the object to which the event is directed.
- Signals are asynchronous, and asynchronous calls are themselves signals, the semantics of events interact with the semantics of active objects and passive objects.

Signals and Active Classes.



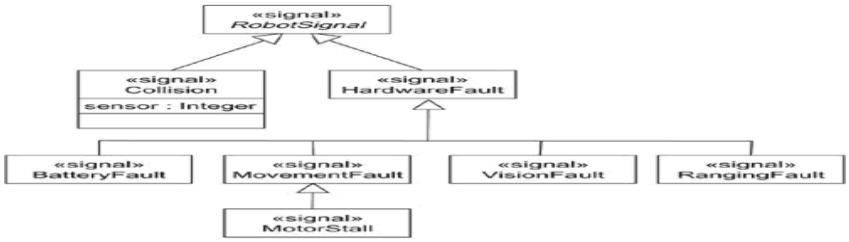
Common Modeling Techniques

1. Modeling a 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.
- Look for the opportunity for polymorphism in the state machines of these active objects.

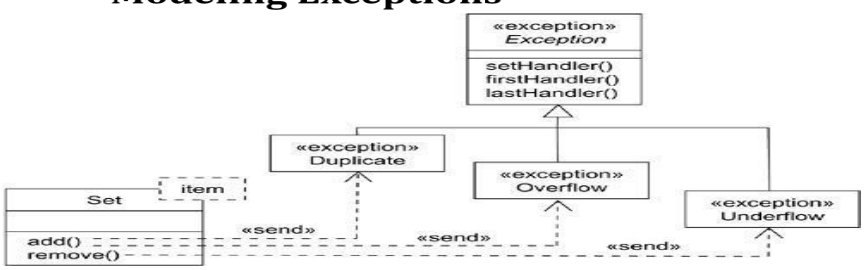
Fig: Modeling Families of Signals



2. Modeling 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.

Modeling Exceptions



State Machines

Terms and Concepts

- A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- 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.

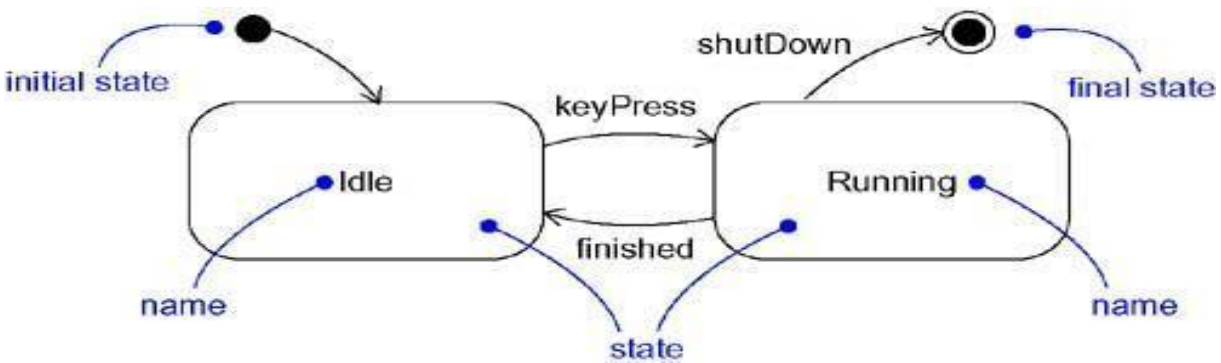
States

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.

A state has several parts.

Name	**A textual string that distinguishes the state from other states; a state may be anonymous, meaning that it has no name**
Entry/exit actions	**Actions executed on entering and exiting the state, respectively**
Internal Transitions	**Transitions that are handled without causing a change in state**
Substates	**The 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**

Fig: shows, you represent a state as a rectangle with rounded corners.



Initial and Final States

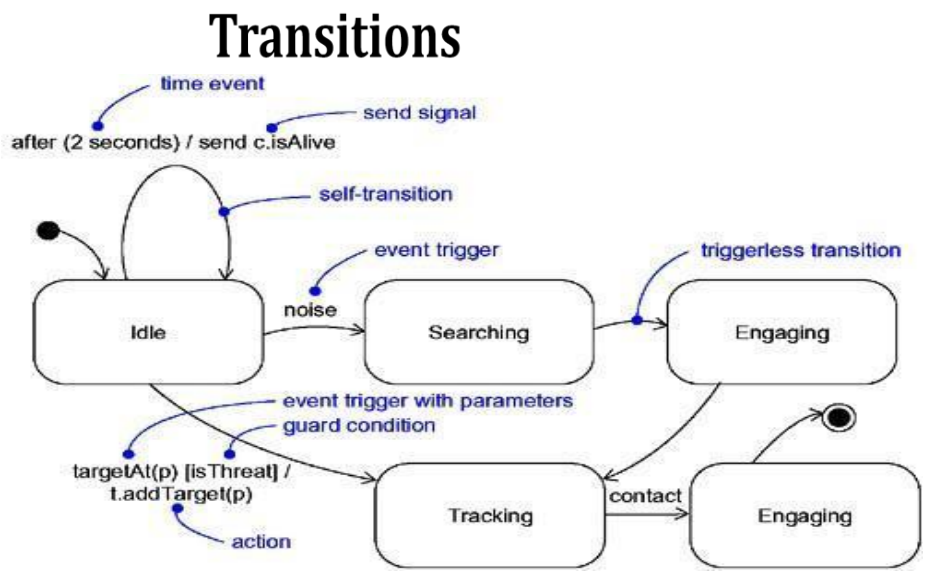
Figure shows, there are two special states that may be defined for an object's state machine.
First, there's the initial state, which indicates the default starting place for the state machine or substate. An initial state is represented as a filled black circle.
Second, there's the final state, which indicates that the execution of the state machine or the enclosing state has been completed. A final state is represented as a filled black circle surrounded by an unfilled circle.

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. A transition has five parts.

Source state	The state affected by the transition; if an object is in the source state, an outgoing transition may fire when the object receives the trigger event of the transition and if the guard condition, if any, is satisfied
Event trigger	The event whose reception by the object in the source state makes the transition eligible to fire, providing its guard condition is satisfied

Guard condition	A Boolean expression that is evaluated when the transition is triggered by the reception of the event trigger; if the expression evaluates True, the transition is eligible to fire; if the expression evaluates False, the transition does not fire and if there is no other transition that could be triggered by that same event, the event is Lost
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



Event Trigger

- An event is the specification of a significant occurrence that has a location in time and space. An event is an occurrence of a stimulus that can trigger a state transition.
- A signal or a call may have parameters whose values are available to the transition, including expressions for the guard condition and action.

Guard

- 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.

Action

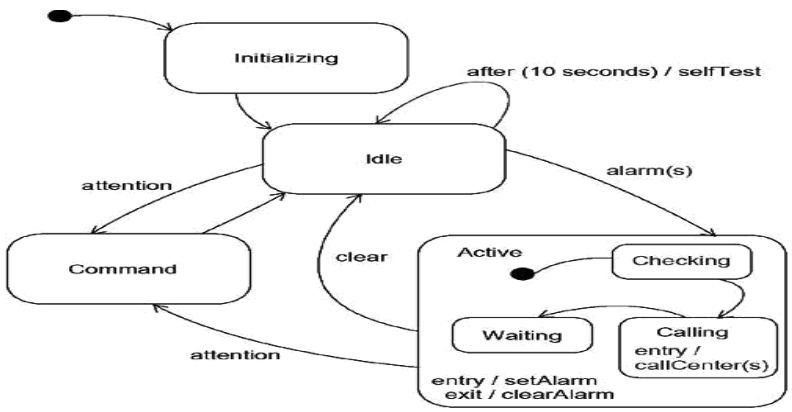
- An action is an executable atomic computation. Actions may include operation calls the creation or destruction of another object, or the sending of a signal to an object.

Common Modeling Techniques

1. Modeling 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.
- Establish the initial and final states for the object. To guide the rest of your model, possibly state the pre- and postconditions of the initial and final states, respectively.
- Decide on the events to which this object may respond.
- 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 substates.
- Check that all actions mentioned in the state machine are sustained by the relationships, methods, and operations of the enclosing object.
- 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.

Fig: Modeling the Lifetime of An Object



Processes and Threads

Terms and Concepts

- 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.
- Processes and threads are rendered as stereotyped active classes.
- An active class is a class whose instances are active objects.

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.

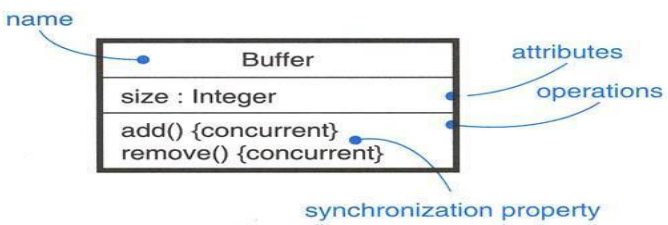
Communication

- 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.

Synchronization

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

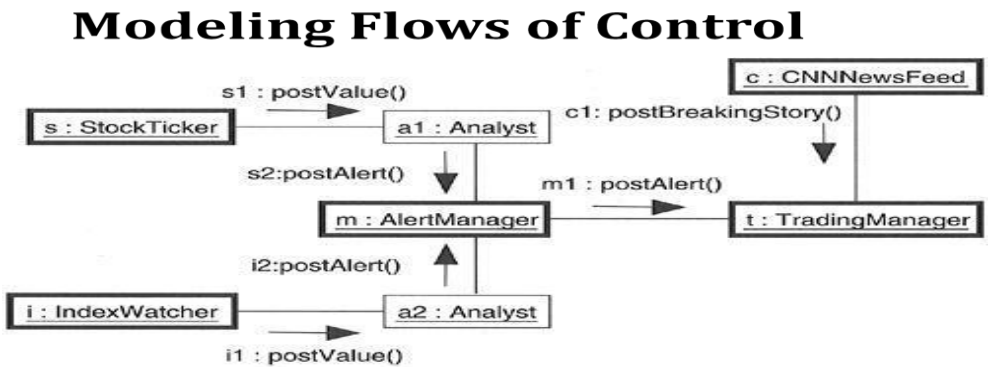
Fig: Synchronization



Common Modeling Techniques

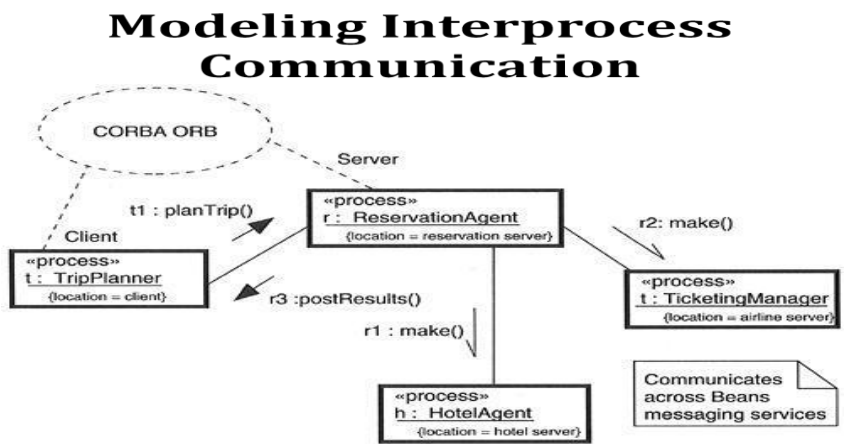
1. Modeling Multiple Flows of Control

- Identify the opportunities for concurrent action and vreify each flow as an active class. Generalize common sets of active objects into an active class.
- Capture these static decisions in class diagrams, explicitly highlighting each active class.
- 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.



2. Modeling Interprocess Communication

- Model the multiple flows of control.
- Consider which of these active objects represent processes and which represent threads.
- 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.



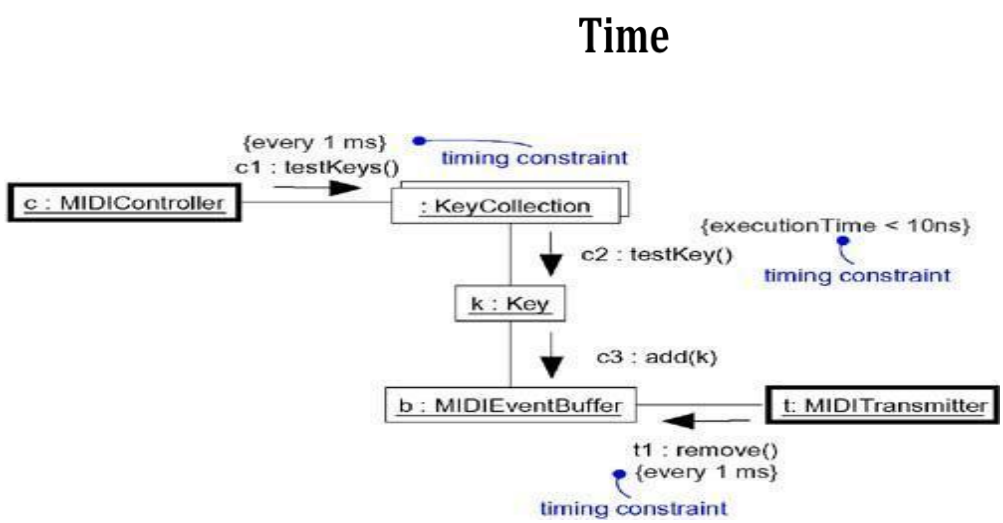
Time and Space

Terms and Concepts

- A *time expression* is an expression that evaluates to an absolute or relative value of 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.
- *Location* is the placement of a component on a node. Graphically, location is rendered as a tagged value.

Time

- Real time systems are, by their very name, 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, so when you model the time-critical nature of a system with the UML, you can give a name to each message in an interaction to be used as a timing mark

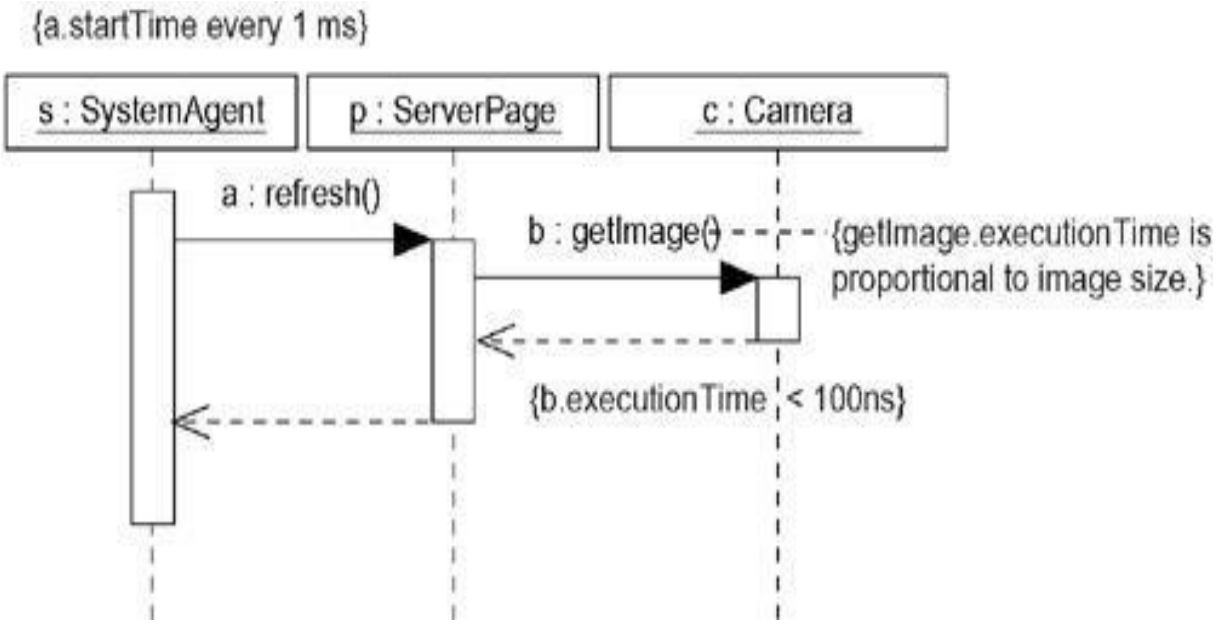


Common Modeling Techniques

1. Modeling 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.
- For each time critical operation in each class, consider its time complexity. Model those semantics as timing constraints on the operation

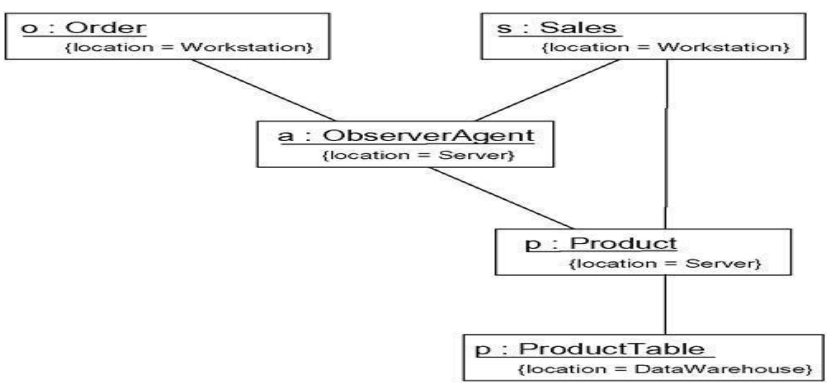
Modeling Timing Constraint



2. Modeling the Distribution of Objects

- For each interesting class of objects in your system, consider its locality of reference. In other words, consider all its neighbors and their locations. A tightly coupled locality will have neighboring objects close by
- Next consider patterns of interaction among related sets of objects. Co-locate sets of objects that have high degrees of interaction, to reduce the cost of communication. 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.

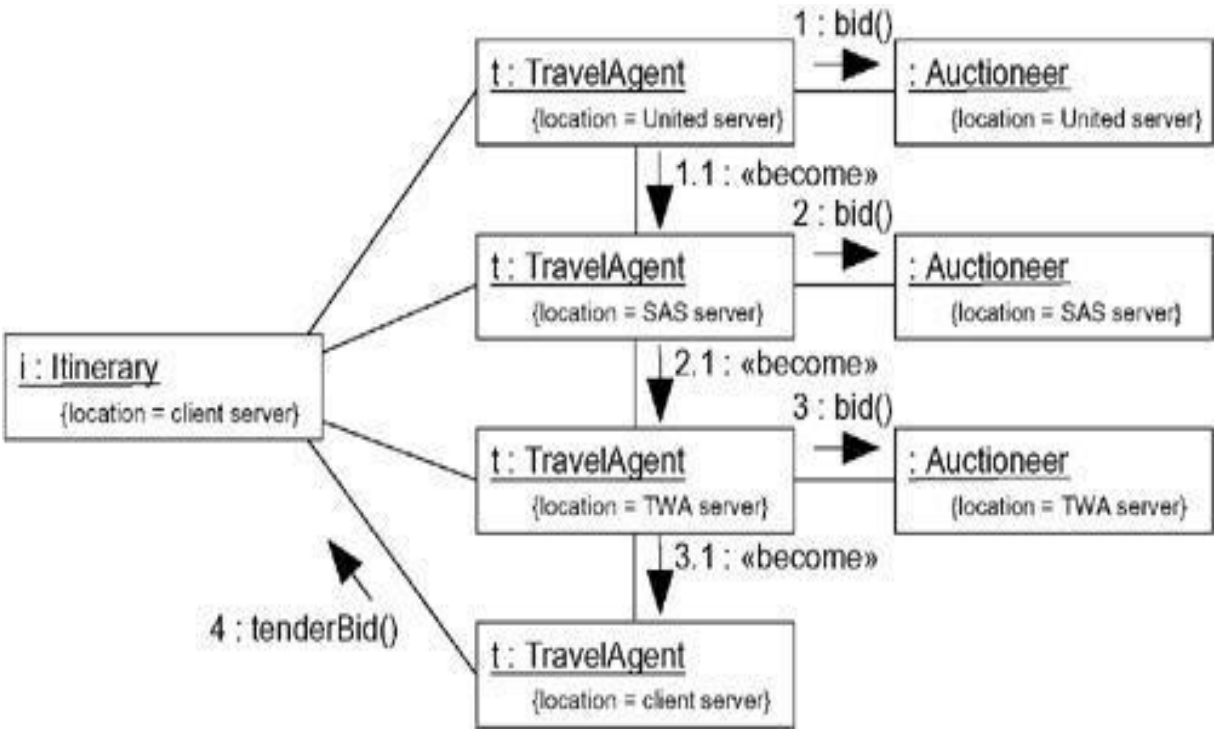
Fig: Modeling the Distribution of Objects



3. Modeling Objects that Migrate

- Select an underlying mechanism for physically transporting objects across nodes.
- Render the allocation of an object to a node by explicitly indicating its location as a tagged value.
- Using the **become** and **copy** stereotyped messages, render the allocation of an object to a new node.
- Consider the issues of synchronization (keeping the state of cloned objects consistent) and identity (preserving the name of the object as it moves).

Fig: Modeling Objects that Migrate



Statechart Diagrams

Terms and Concepts

- A *statechart diagram* shows a state machine, emphasizing the flow of control from state to state.
- A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- 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.

Contents - Statechart diagrams commonly contain

- ✓ Simple states and composite states
- ✓ Transitions, including events and actions
- A statechart diagram is basically a projection of the elements found in a state machine.
- statechart diagrams contain branches, forks, joins, action states, activity states, objects, initial states, final states, history states.

Common uses

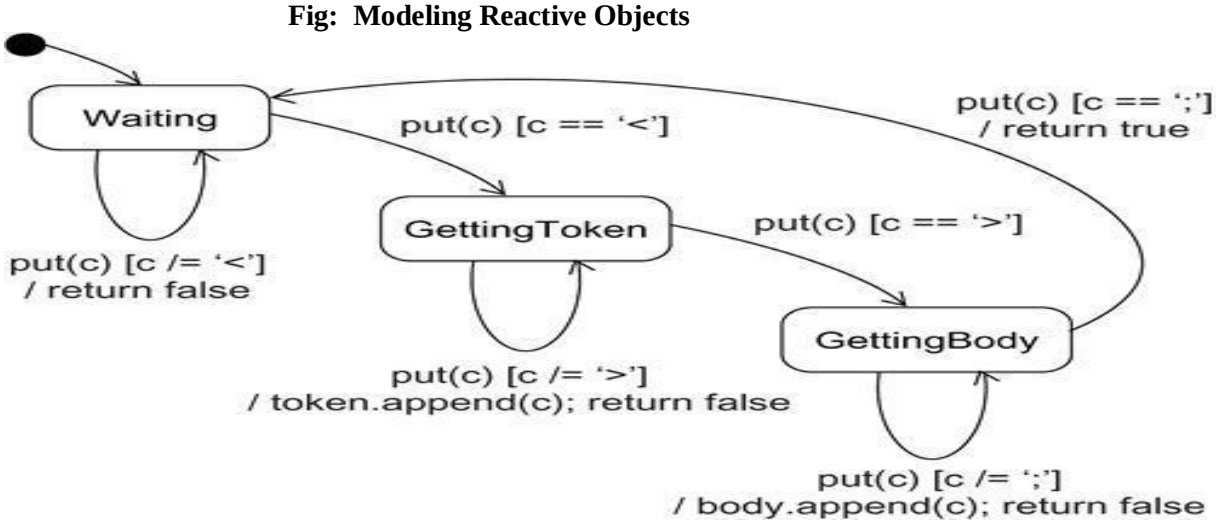
- A statechart diagram will use to model some dynamic aspect of a system.
- It is also used in the context of virtually any modeling element.
- Typically, use statechart diagrams in the context of the system as a whole, a subsystem, or a class.
- It can also be used to attach a statechart diagrams to use cases.

Common Modeling Technique

1. 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 postconditions 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 substates.
- 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 substates, branches, forks, joins, and history states.

Fig: Modeling Reactive Objects



2. Forward and Reverse Engineering

- *Forward engineering*(the creation of code from a model) is possible for statechart diagrams, especially if the context of the diagram is a class.
- The forward engineering tool must generate the necessary private attributes and final static constants.
- *Reverse engineering* (the creation of a model from code) is theoretically possible, but practically not very useful. The choice of what constitutes a meaningful state is in the eye of the designer.
- Reverse engineering tools have no capacity for abstraction and therefore cannot automatically produce meaningful statechart diagrams.

A well-structured statechart diagram

• Is focused on communicating one aspect of a system's dynamics.
• Contains only those elements that are essential to understanding that aspect.
• Provides detail consistent with its level of abstraction (expose only those features that are essential to understanding).
• Uses a balance between the styles of Mealy and Moore machines.

When you draw a statechart diagram

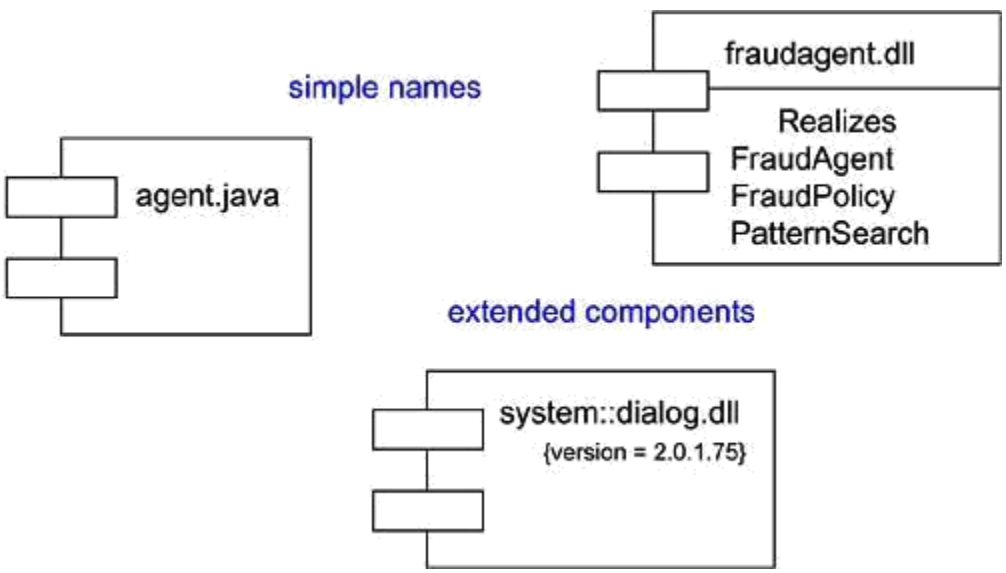
- Give it a name that communicates its purpose.
- Start with modeling the stable states of the object, then follow with modeling the legal transitions from state to state. Address branching, concurrency, and object flow as secondary considerations, possibly in separate diagrams.
- Lay out its elements to minimize lines that cross.

Component

- A *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.

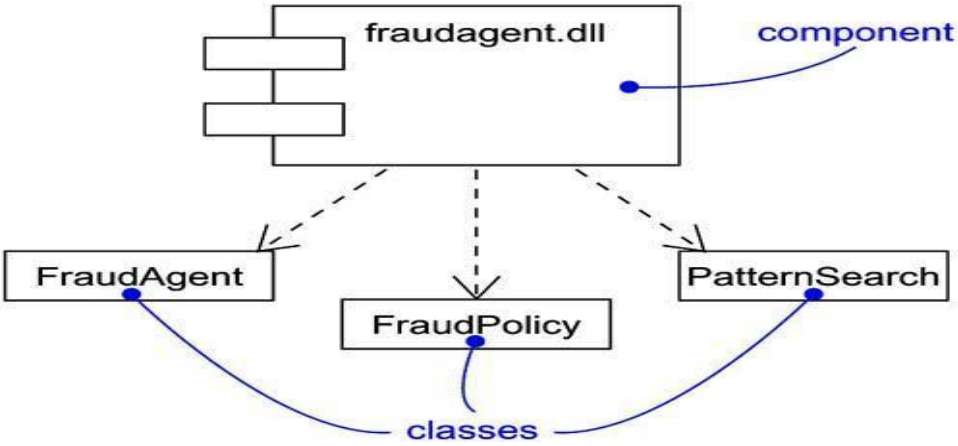
Names

- *A component name must be unique within its enclosing package*



Components and Classes

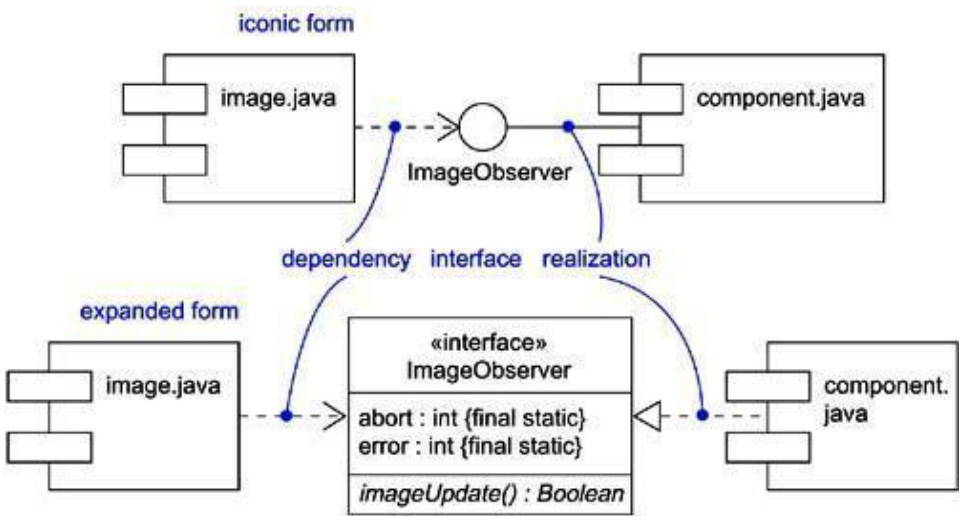
- There are some significant differences between components and classes.
- Classes represent logical abstractions; components represent physical things that live in the world of bits. In short, components may live on nodes, classes may not.
- Components represent the physical packaging of otherwise logical components and are at a different level of abstraction.
- Classes may have attributes and operations directly. In general, components only have operations that are reachable only through their interfaces.



Components and Interfaces

- An interface is a collection of operations that are used to specify a service of a class or a component. The relationship between component and interface is important. All the most common component-based operating system facilities (such as COM+, CORBA, and Enterprise Java Beans) use interfaces as the glue that binds components together.
- An interface that a component realizes is called an *export interface,* meaning an interface that the component provides as a service to other components. A component may provide many export interfaces.
- The interface that a component uses is called an *import interface,* meaning an interface that the component conforms to and so builds on.
- A component may conform to many import interfaces.
- A component may both import and export interfaces.

Fig: Components and Interfaces



Binary Replaceability

- The basic intent of every component-based operating system facility is to permit the assembly of systems from binary replaceable parts.
- This means that you can create a system out of components and then evolve that system by adding new components and replacing old ones, without rebuilding the system.

- Interfaces are the key to making this happen. When you specify an interface, you can drop into the executable system any component that conforms to or provides that interface.
- You can extend the system by making the components provide new services through other interfaces, which, in turn, other components can discover and use. These semantics explain the intent behind the definition of components in the UML.

Kinds of Components

Three kinds of components may be distinguished

- First, there are *deployment components.*
- Second, there are *work product components.*
- Third are *execution components.*

Organizing Components

- You can organize components by grouping them in packages in the same manner in which you organize classes.
- The UML defines five standard stereotypes that apply to components

1. executable
2. library
3. table
4. file
5. document

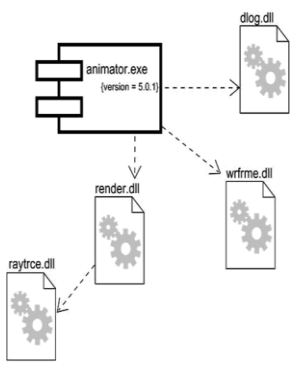
Common Modeling Techniques

1. Modeling Executables and Libraries

To model executables and libraries:

- Identify the partitioning of your physical system. Consider the impact of technical, configuration management, and reuse issues.
- Model any executables and libraries as components, using the appropriate standard elements. If your implementation introduces new kinds of components, introduce a new appropriate stereotype.
- If it's important for you to manage the seams in your system, model the significant interfaces that some components use and others realize.
- As necessary to communicate your intent, model the relationships among these executables, libraries, and interfaces. Most often, you'll want to model the dependencies among these parts in order to visualize the impact of change.

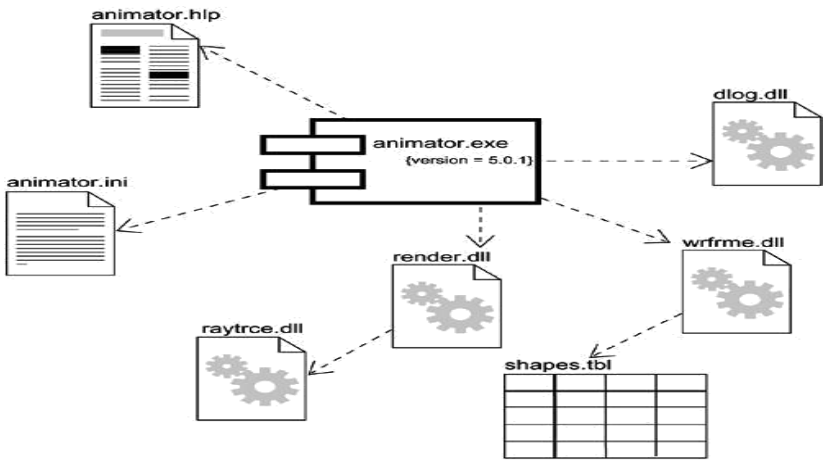
Fig: Modeling Executables and Libraries



2. Modeling Tables, Files, and Documents

To model tables, files, and documents,

- Identify the ancillary components that are part of the physical implementation of your system.
- Model these things as components. If your implementation introduces new kinds of artifacts, introduce a new appropriate stereotype.
- As necessary to communicate your intent, model the relationships among these ancillary components and the other executables, libraries, and interfaces in your system.
- Most often, you'll want to model the dependencies among these parts in order to visualize the impact of change.

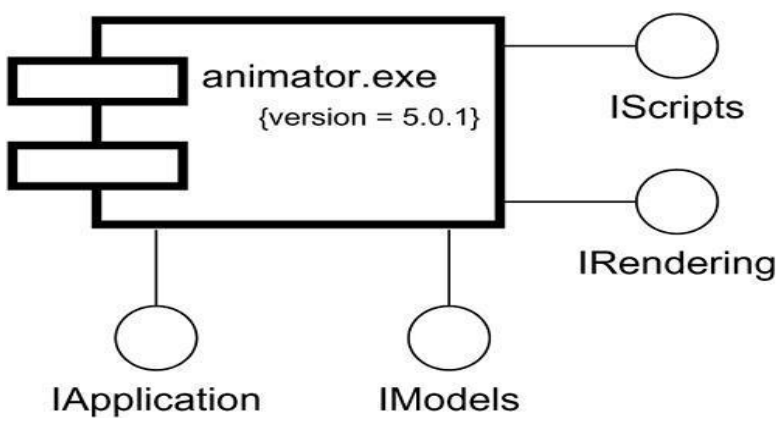


3. Modeling an API

To model an API,

- Identify the programmatic seams in your system and model each seam as an interface, collecting the attributes and operations that form this edge.
- Expose only those properties of the interface that are important to visualize in the given context; otherwise, hide these properties, keeping them in the interface's specification for reference, as necessary.
- Model the realization of each API only insofar as it is important to show the configuration of a specific implementation.

Fig: Modeling an API

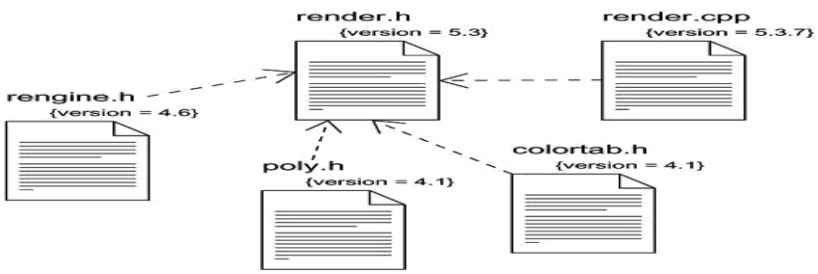


4. Modeling Source Code

To model source code,

- Depending on the constraints imposed by your development tools, model the files used to store the details of all your logical elements, along with their compilation dependencies.
- If it's important for you to bolt these models to your configuration management and version control tools, you'll want to include tagged values, such as version, author, and check in/check out information, for each file that's under configuration management.
- As far as possible, let your development tools manage the relationships among these files, and use the UML only to visualize and document these relationships.

Fig: Modeling Source Code



Component Diagrams

- A *component diagram* shows a set of components and their relationships. Graphically, a component diagram is a collection of vertices and arcs.

Contents

Component diagrams commonly contain

- Components
- Interfaces
- Dependency, generalization, association, and realization relationships Like all other diagrams, component diagrams may contain notes and constraints.

Common uses

- Component diagrams used to model the static implementation view of a system.
- Typically use component diagrams in one of four ways.

1. To model source code
2. To model executable releases
3. To model physical databases
4. To model adaptable systems

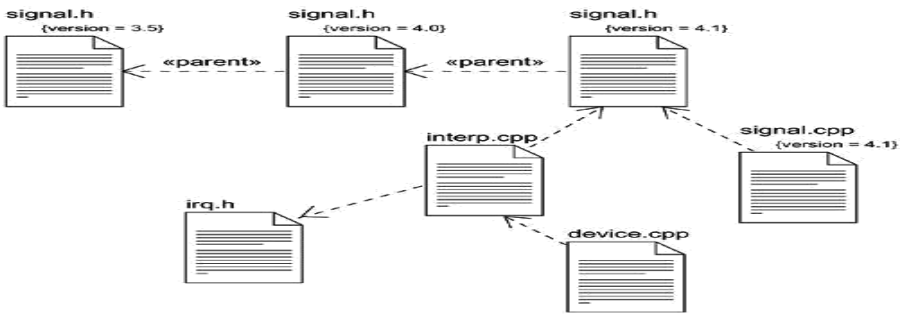
Common Modeling Techniques

1. Modeling Source Code

To model a system's source code:

- Either by forward or reverse engineering, identify the set of source code files of interest and model them as components stereotyped as files.
- For larger systems, use packages to show groups of source code files.

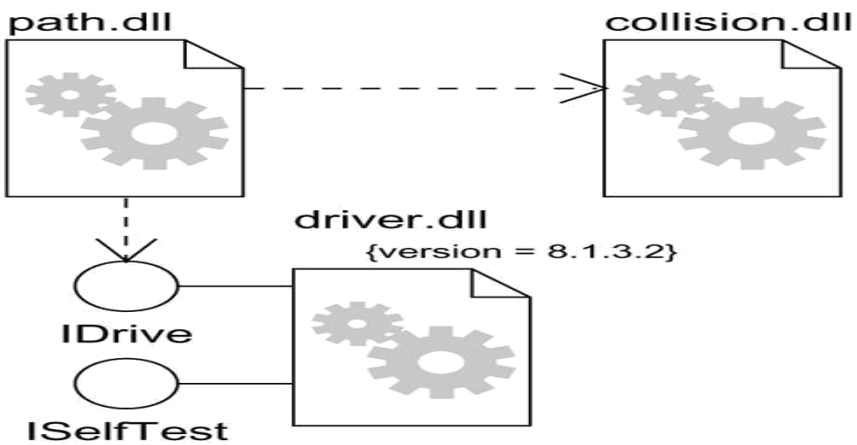
- Consider exposing a tagged value indicating such information as the version number of the source code file, its author, and the date it was last changed. Use tools to manage the value of this tag.
- Model the compilation dependencies among these files using dependencies. Again, use tools to help generate and manage these dependencies.



2. Modeling an Executable Release

To model an executable release

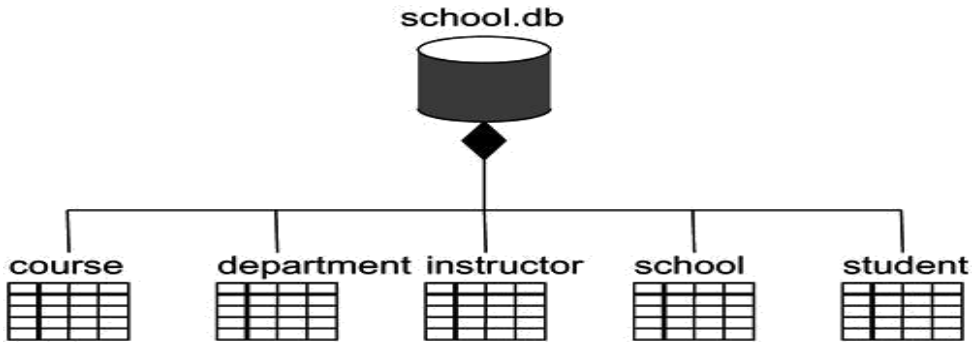
- Identify the set of components you'd like to model. Typically, this will involve some or all the components that live on one node, or the distribution of these sets of components across all the nodes in the system.
- Consider the stereotype of each component in this set. For most systems, you'll find 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 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.



3. Modeling a Physical Database

To model a physical database:

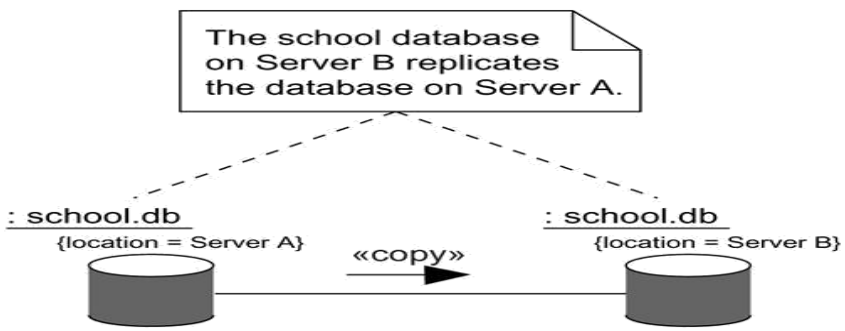
- Identify the classes in your model that represent your logical database schema.
- Select a strategy for mapping these classes to tables. You will also want to consider the physical distribution of your databases. Your mapping strategy will be affected by the location in which you want your data to 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.



4. Modeling Adaptable Systems

- 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 a Physical Database



5. Forward and Reverse Engineering

To forward engineer a component diagram:

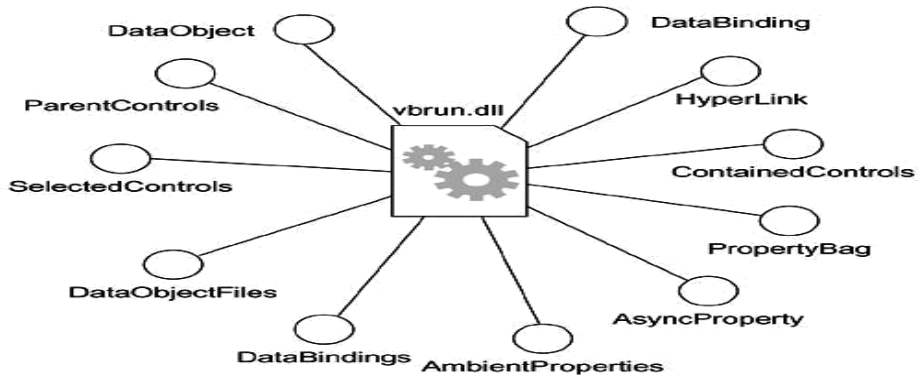
- For each component, identify the classes or collaborations that the component implements.
- Choose the target for each component. Your choice is basically between source code (a form that can be manipulated by development tools) or a binary library or executable (a form that can be dropped into a running system).
- Use tools to forward engineer your models.

To reverse engineer a component diagram,

- Choose the target you want to reverse engineer. Source code can be reverse engineered to components and then classes. Binary libraries can be reverse engineered to uncover their interfaces. Executables can be reverse engineered the least.
- Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or to modify an existing one that was previously forward engineered.
- Using your tool, create a component diagram by querying the model. For example, you might start with one or more components, then expand the diagram by following relationships or

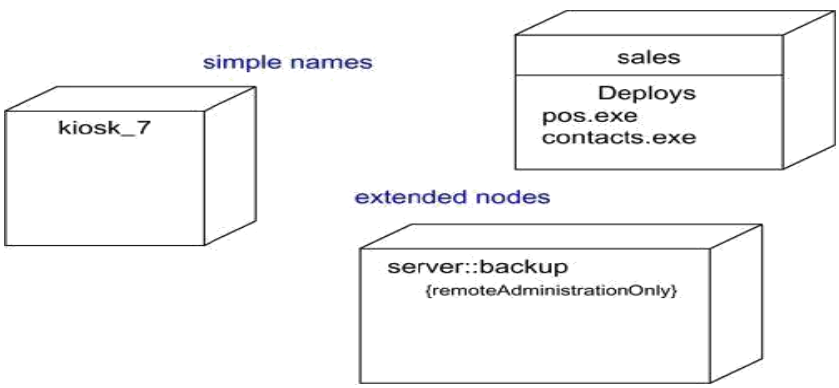
neighboring components. Expose or hide the details of the contents of this component diagram as necessary to communicate your intent.

Figure : Reverse Engineering



Deployment

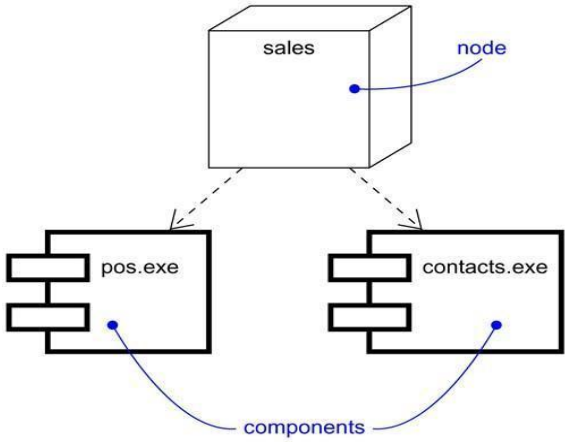
- A *node* is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. Graphically, a node is rendered as a cube.
- Every node must have a name that distinguishes it from other nodes.
- A *name* is a textual string. That name alone is known as a *simple name;* a *path name* is the node name prefixed by the name of the package in which that node lives.
- A node is typically drawn showing only its name, as in. Just as with classes, you may draw nodes adorned with tagged values or with additional compartments to expose their details.



Nodes and Components

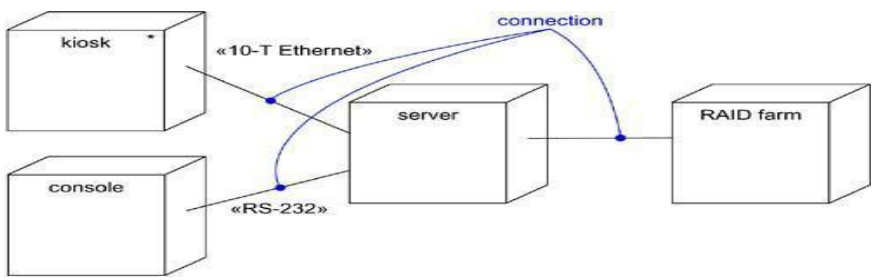
There are some significant differences between nodes and components.

- 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.



Connections

- The most common kind of relationship you'll 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, as Figure .

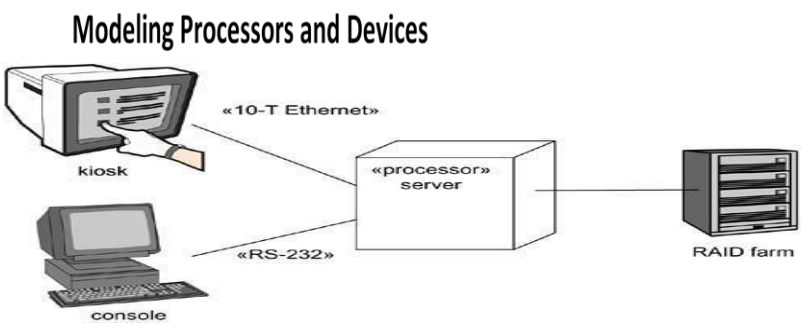


Common Modeling Techniques

1. Modeling Processors and Devices

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.



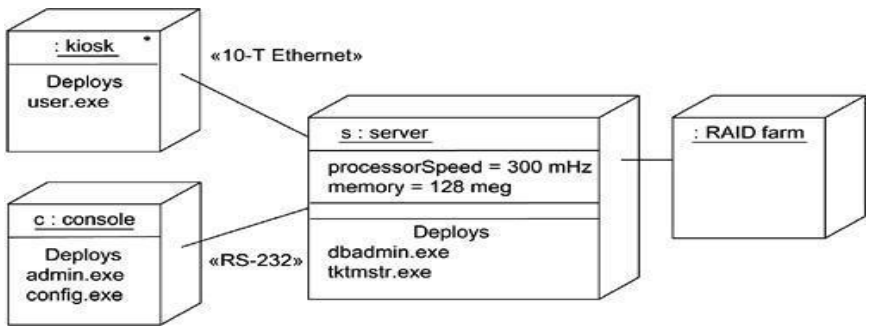
2. Modeling the Distribution of Components

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.
- Don't make the allocation visible, but leave it as part of the backplane of your model• that is, in each node's specification.
- Using dependency relationships, connect each node with the components it deploys.
- List the components deployed on a node in an additional compartment.

Modeling the Distribution of Components



Deployment Diagrams

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

Contents

Deployment diagrams commonly contain

- Nodes
- Dependency and association relationships

Like all other diagrams, deployment diagrams may contain notes and constraints

Common Uses

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

1. To model embedded systems 2. To model client/server systems 3. To model fully distributed systems

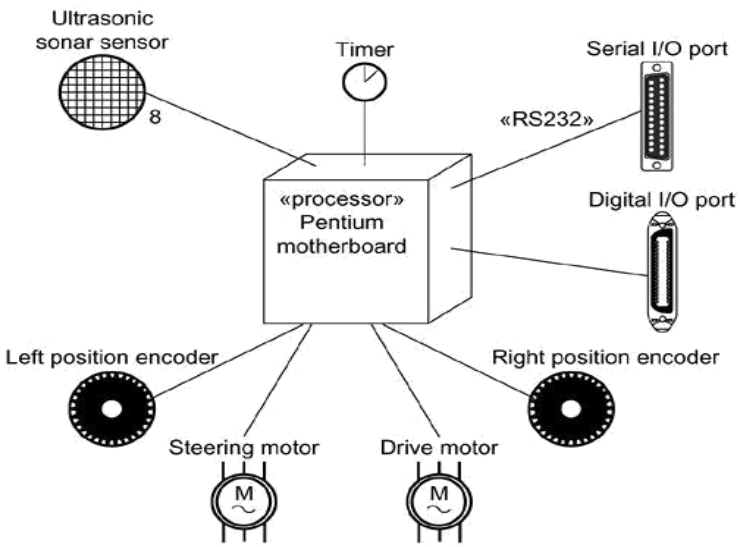
Common Modeling Techniques

1. 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

Modeling an Embedded System

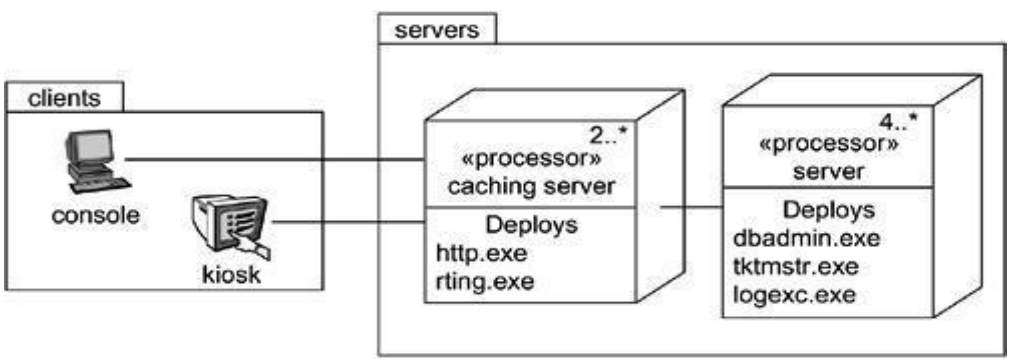


2. Modeling a Client/Server System

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.

Modeling a Client/Server System



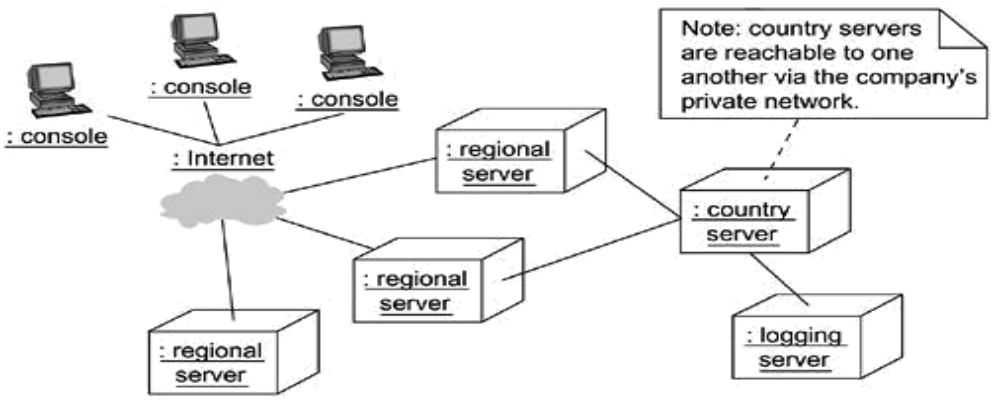
3. Modeling a Fully Distributed System

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.

Modeling a Fully Distributed System



4. Forward and Reverse Engineering

To reverse engineer a deployment diagram:

- Choose the target that you want to reverse engineer. In some cases, you'll want to sweep across your entire network; in others, you can limit your search.
- Choose also the fidelity of your reverse engineering. In some cases, it's sufficient to reverse engineer just to the level of all the system's processors; in others, you'll want to reverse engineer the system's networking peripherals, as well.
- Use a tool that walks across your system, discovering its hardware topology. Record that topology in a deployment model.
- Along the way, you can use similar tools to discover the components that live on each node, which you can also record in a deployment model. You'll want to use an intelligent search, for even a basic personal computer can contain gigabytes of components, many of which may not be relevant to your system.
- Using your modeling tools, create a deployment diagram by querying the model. For example, you might start with visualizing the basic client/server topology, then expand on the diagram by populating certain nodes with components of interest that live on them. Expose or hide the details of the contents of this deployment diagram as necessary to communicate your intent.

UNIT- IV: GRASP: Designing objects with responsibilities, creator, information expert, low coupling, high cohesion, Design patterns, creational, factory method, structural, bridge, adaptor, behavioral, strategy.

GRASP INTRODUCTION

What is Pattern?

It is common that when experts works on particular problems, they need not reinvent the wheel i.e. they need not find a complete new solution that is completely different from existing solutions. Rather they remember the similar problem and it solution and reuse the core of it for solving new problem. They think in terms of **"**Problem Solution**"** pairs. Abstracting from such pairs and churning out the commonalities leads to patterns.

Each pattern is three-part rule which expresses a relation between a certain context, a problem and a solution.

Patterns have names and they facilitate communication- common vocabulary and understanding-easy to remember whole concept in a nutshell.

GRASP:

Craig Larman coined these terms for the first time.

stands for **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns.

The mnemonic meaning could be **"**successful OOAD (Object Oriented Analysis & Design) requires grasping (understanding)**"**.

General: = Abstract i.e. widely applicable for variety of situations

Responsibility: = Obligations, duties –information object performing duty of holding information, computational object performing duty of computation

Assignment: = Giving a responsibility to a module- capability to carry out responsibility structure and methods

Software: = Computer code

Patterns: = Regularities, templates, abstraction

Assigning responsibilities to objects is always is major task in designing the software**'**s. Specifically it**'**s always a challenge to decide which object carry out what job (responsibilities) and quality of software design largely depended upon such responsibilities assignment.

Poor choices about such responsibities leads to fragile systems and components which are hard to maintain, understand, reuse or extend.

Let's define responsibility:

Responsibility is definedas a contract or obligation of a type or class and is related to behavior.

There are 2 types of responsibilities

- **Knowing -** responsibilities of an object include

  Knowing about private encapsulated data-member data

  Knowing about related objects

  Knowing about things it can derive or calculate

- **Doing -** responsibility of an object include

  Doing something itself-assign, calculate, create

  Initiating action in other objects

  Controlling and coordinating activities in other objects

The responsibilities would be translated into classes and their methods and it is largely influenced by the granularity of the responsibility. Don't confuse the responsibility as a method in a class even though they are implemented using methods which either act alone or collaborate with other methods and objects. Object-oriented principles are applied by experienced developers/experts in deciding general assignment of responsibilities.

GRASP patterns describe fundamental principles of assigning responsibilities to objects and it has following principles and patterns in the family

- Information Expert
- Creator
- Controller
- Low Coupling
- High Cohesion
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations.

Creator

The main Objective is:

To understand the GRASP pattern **"**Creator**"**.

Problem: Who creates the new instance of some class?

Solution:

Assign class A the responsibility to create an instance of class B if….

- A aggregates (whole-part relationship) B objects
- A contains B objects
- A records instances of B objects
- A closely uses B objects
- A has initializing data that is needed while creating B objects (thus A is an expert with respect to creating B)

Approach:

Step I: Closely look at domain/ design model and ask: who should be creating these classes?

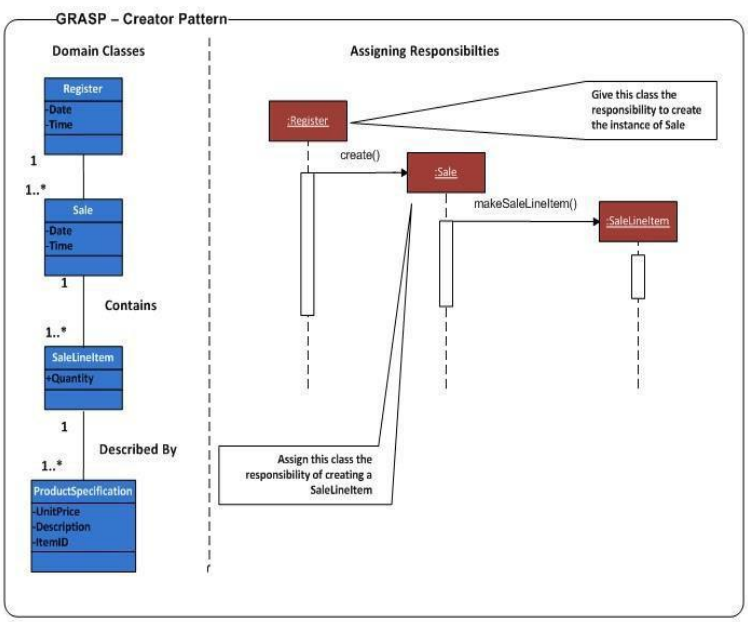
Step II: Search for the classes who create, aggregate etc.

Step III: Assign the responsibility of creation

Description

Let's extend an example of POS (Point Of Sale) systems explained in previous part. This is what we come across malls where there are many POS counters. In the domain model, following classes are identified. These classes are shown in the left part of diagram (Fig. No.1). A class **"**Register**"** represents the sales register. This would have details of all sales done. As the **"**Register**"** aggregates the **"**Sale**"**, it fulfils the first criteria of the solution. Similarly the **"**Sale**"** contains the **"**SaleLineItem**"**. Hence **"**Register**"** should create the instance of **"**Sale**"** and **"**Sale**"** should create instance of **"**SaleLineItem**"**. Fig. No. 1

Fig. No. 1



Class	Relationship with other classes	Responsibility and method
Register	Contains Class "Sale"	Creating an instance createSale()
Sale	Composite aggregation with SaleLineItem	Creating an instance makeSaleLineItem()

Following diagram depicts the assignment of responsibilities and corresponding methods. Fig No. 2

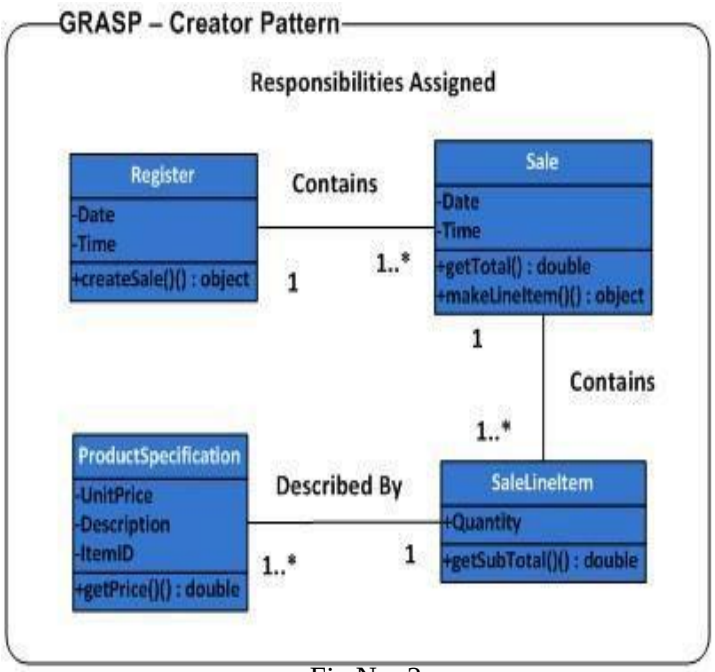


Fig No. 2

Benefits:

- Support low coupling
- The responsibility is assigned to a class which fulfils the criteria thus separating the logic for creation and actual execution.

Liabilities /Contradictions:

- Sometimes creation may require significant complexity. Many a times there would be tight coupling among the creator and create class. Also the task of destroying the objects is to be thought of. Recycling of such instances (pooling) becomes important for the performance reasons.
- In many scenarios the creations is conditional and it may be from family of classes making it very complex.
- For creation of family of classes or to tackle the creation and recycling, other patterns are available which are factory, abstract factory etc.

Information Expert

Objective

To understand about the GRASP pattern "Information Expert'.

Problem: Any real world application has hundreds of classes and thousand of actions. How do I start assigning the responsibility?

Solution: Assign a responsibility to Information Expert, the class that has information necessary to fulfil the responsibility.**"**

Generally, if domain model is ready then it is pretty straight forward to assign the responsibilities and arrive at design model.

Approach

Step I: State the responsibility clearly.

Step II: Search for the classes who have the information needed to fulfil the responsibility.

Step III: Assign the responsibility

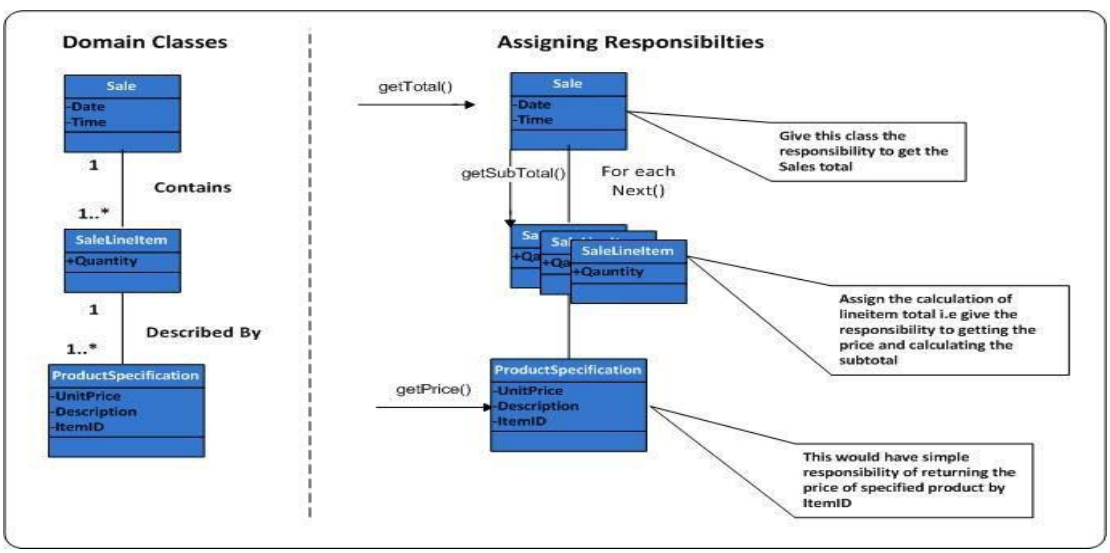
Description

Let's take an example of POS (Point Of Sale) systems. This is what we come across malls where there are many POS counters. In the domain model, following classes are identified. These classes are shown in the left part of diagram (Fig. No.1). Let's think about some responsibility.

Who should have the responsibility of calculating the Sales Total?

Sales Total =Sum of (Unit price X quantity of sale line item)

E.g. a person is buying 2 line items, toothpaste (2 no's) and soap (2 no's) are being bought Sales total = (Price of toothpaste)* (quantity of toothpaste) + (Price of soap)* (quantity of soap)

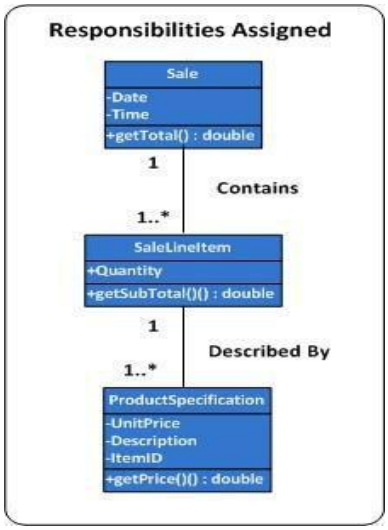


In the above scenario, the classes Sale, "SaleLineItem" and "ProductSpecification" has the information such as unit price for item/ product ("ProductSpecification"), sales quantity ("SaleLineItem") and no. of saleLineItems ("Sale"). So the responsibility of arriving at Sales Total needs to be assigned to all these classes. Now in this case, If the class is encapsulating data members then maintaining these data values is the responsibility of the class and it should provide the get and set methods for accessing such data.

Class	Data Values Encapsulated	Responsibility and method
ProductDescription	UnitPrice	Getting the unit price getPrice()
SaleLineItem	Quantity of items and hence the subtotal for particular item Quantity X Unit Price	Getting the Subtotal getSubTotal()
Sale	No. of Saleline Items, data and time Sumof(getSubTotal()x SaleLine item)	Get the Total of Sales getTotal()

It is demonstrated in right portion of above diagram (Fig. No.1).

Following diagram depicts the assignment of responsibilities and corresponding methods.



Benefits:

- Encapsulation is maintained as objects use their own data to fulfil the task.
- Support low coupling
- Behaviour (responsibility) is distributed across the classes that have the required information thus encouraging the highly cohesive lightweight classes

Liabilities /Contradictions:

- Sometimes while assigning responsibilities, lower level responsibilities are uncovered e.g what the profit or loss for a particular sale?
- Not all the times the application of this pattern is desirable. Consider the responsibility of saving such Sale to database. By the principle for this pattern, it should be the class **"**Sale**"** to carry out this responsibility but as experienced programmer we know it is hardly the case. The DAL (Data Access Layer) carries out this responsibilities may be by providing methods such as SaveSale() into some controller class.

Low Coupling

Objective

To understand the GRASP pattern **"**Low Coupling**"**

The principle behind this pattern is ***"How to support low dependency and increased reuse?" and "How to assign the responsibilities so that the coupling is low?"***

Coupling refers to connectedness or linking. It is a measure of how strongly one class is connected to or has knowledge (information or knowhow) of or relies/depends upon the other classes. With experience, programmers would agree on something i.e. making changes is always been the hardest task and given a chance all would like to work upon systems from ground up. With increased complexity in the programming world and programming languages, designing is becoming driver for system's overall quality and usability. Designing for low connectedness is an important activity which can help to have a system in such a way that the changes in one element (sub-system, system, class, etc) will limit changes to other elements. Being at extreme, is it possible to have no coupling at all? The answer to this question is **"**No**"** as without having any connection; a system can't be called a system.

High coupling (high dependency, linking and connectedness) leads to

- Cascade of changes – this forces changes in related classes
- Harder to reuse its use of highly coupled elements requires the additional presence of the classes it is dependent on
- Harder to understand in isolation

Problem: How to reduce impact of changes and encourage reuse?

Solution: Assign a responsibility so that the coupling (linking classes) remains low

Approach:

Step I: Closely look for classes with many associations to other classes.

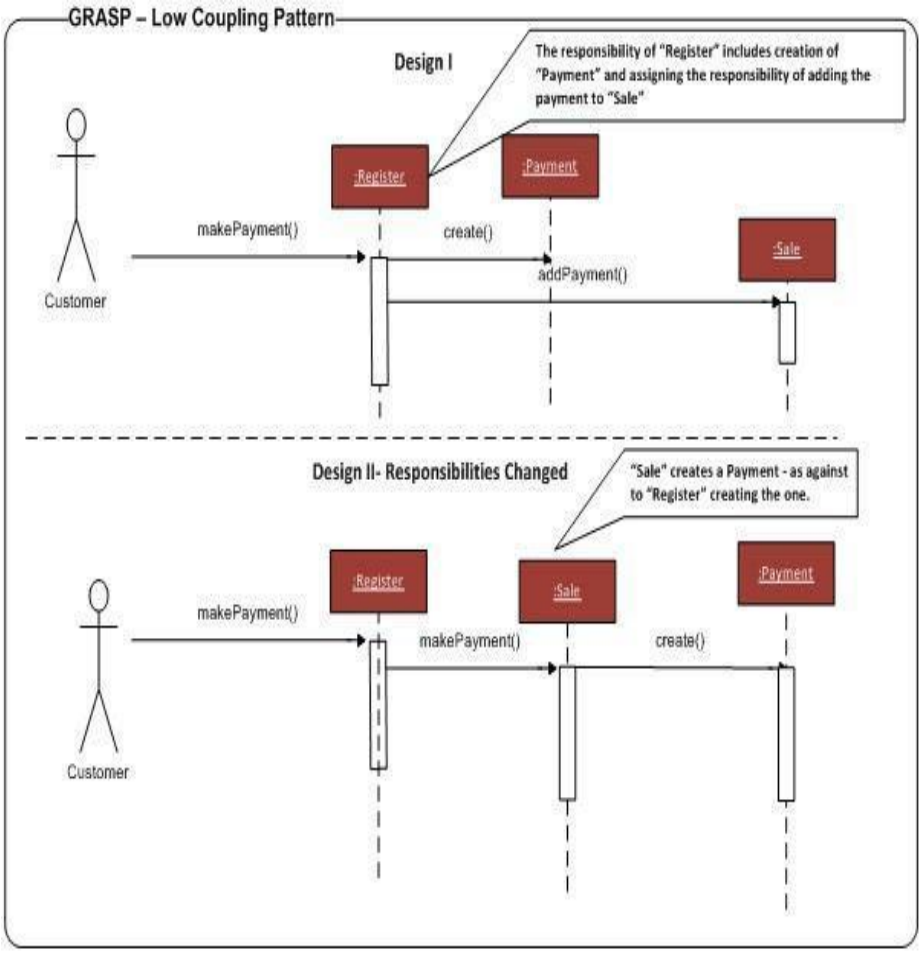
Step II: Look for the methods that rely on a lot of other methods (or methods in other classes i.e. dependencies

Step III: Rework the design so as assign responsibilities to different classes in such a way that they have lesser association and dependency.

Description

Let's extend an example of POS (Point Of Sale) systems carried in series of articles. In the domain model, following classes like Register, Payment and Sale are identified. These classes are shown in the left part of diagram (Fig. No.1). There are many system operations such as closing a sale, make payment etc which are related to business function "Sales".

Fig No. 1



Class	Relationship with other classes	Responsibility and method	Remarks
"Register" of Design I	Creator of the "Payment" Controls the "Sale"	Controls and coordinates the sale through makePayment()	So if Sale or payment changes its going to affect the Register. This results in tight Coupling
Register" and "Sale" of Design II	Controller of "Sale" and "Sale" being the creator of "Payment"	Creating an instance "create" Controlling the Sale through makePayment()	This results in division of Responsibilities and Hence low coupling as "Register" need not know about the payment.

The common forms of coupling for A to B include

- A has a data member which refers to B instance
- A has a method which B is an argument
- A is direct or indirect subclass of B
- B is an interface and A implements B

Related Patterns: The pattern related to is "High Cohesion".

Benefits

- Maintainability -Little or not affected by changes in other components
- Understandability- simple to understand in the isolation
- Reusability- convenient to reuse and easier to grab hold of classes

Liabilities / Contradictions:

- Coupling to stable elements (classes in library or well tested classes) doesn't account to be a problem.

High Cohesion

Objective

To understand the GRASP pattern **"**High Cohesion**"**

The dictionary meaning of the word **"**cohesion**"** is **"**the state of cohering or sticking together**"**. In botany world **"**the process of some parts (which are generally separate) of plant growing together**"** and in physics world **"**it's the intermolecular force that holds together the molecules in solid or liquid**"**. So this meaning would give us good forethought as what would **"**High Cohesion**"** would mean in the context of responsibilities? As we have proceeded through this series, we have seen principles behind assigning the responsibilities. Once they are assigned or being assigned, an important aspect is how they are related or focused? If they are related, sticking together or focussed, the element carrying to those responsibilities would become bloated and result into system quite complex and harder to maintain and reuse. The principle behind this pattern is ***"How to keep complexity manageable?***

Cohesion can be said to a measure of **"**relatedness**"** or **"**how related all the attributes and behaviour in a class are**"**. High cohesion refers to how strongly relate the responsibilities are.

A class with low cohesion i.e. carrying out the unrelated responsibilities in turn does so many unrelated things or end up doing too much of a work. Such classes culminates into

- Hard to comprehend or understand
- Harder to reuse as it is tough job to segregate and mark the reusable elements
- Harder to maintain and extend

Problem: How to keep classes focused and manageable?

Solution: Assign responsibility so that cohesion remains high

Approach:

Step I: Closely look for classes with probability of many responsibilities i.e. classes with too-few or disconnected methods.

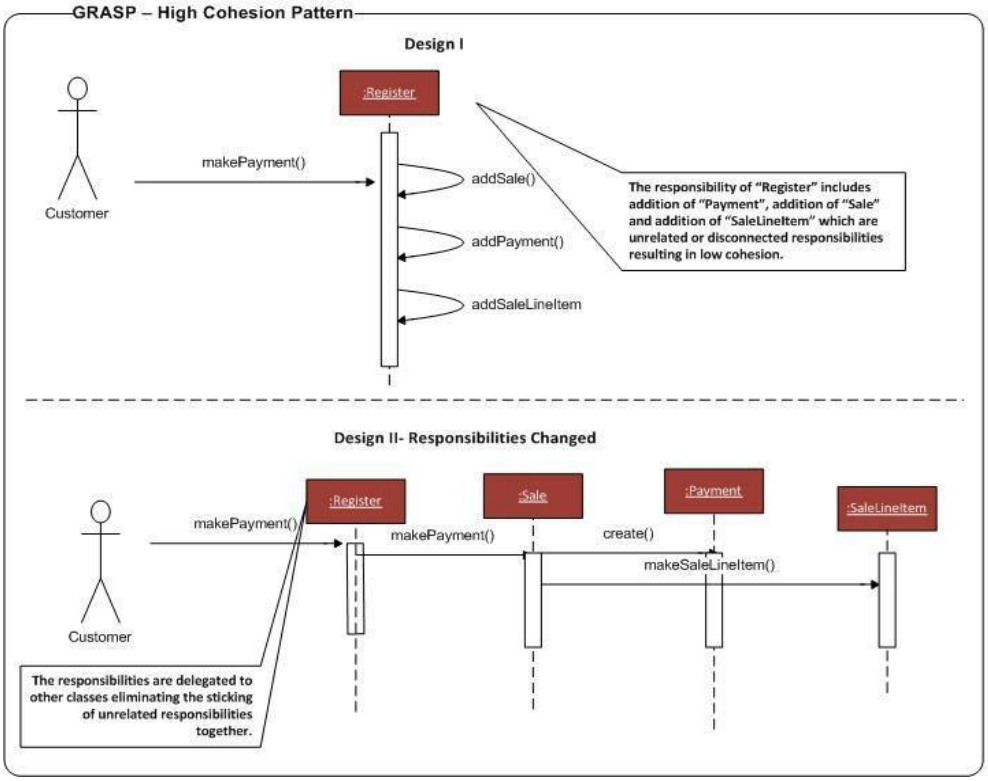
Step II: Look for the methods which do a lot (look for method names having words like **"**AND**"**, **"**OR**"**)

Step III: Assign a responsibility in such a way that all the responsibilities in a class are related

Description

Let's extend an example of POS (Point Of Sale) systems carried in series of articles. In the domain model, following classes like Register, Payment and Sale are identified. In the upper part of diagram (Fig. No.1) a class **"**Register**"** is displayed carrying out many unrelated responsibilities. There are many system operations such as closing a sale, make payment etc which are related to business function **"**Sales**"**.

Fig No. 1



Class	Relationship with other classes	Responsibility and method	Remarks
"Register"of Design I	Creator of the "Payment", "Sale" and "SaleLineItem"	Carries out responsibilities for adding sale, payment and also the sale line item.	These are absolutely unrelated responsibilities resulting in low cohesion
"Register" And "Sale" of Design II	Controller of "Sale" and "Sale" being the creator of "Payment"	Creating an instance "create"Controlling the Sale through makePayment()	This results in Division of responsibilities according to relations among them and reduces the complexity hence high cohesion

The common examples of Low cohesion are

- System level: ATM having functionality for getting "Teller Reports" i.e. for user this is not required and related
- Class level: An "Employee" class with a method called "getDrivingRecords()"
- Method level: The methods with having "AND","OR" in their names which implies that the method is going to execute multiple functions making it difficult to reuse.
- This also applies to subsystem (package), component level. E.g. a component for exception handling also doing job of caching.

One of the fundamental goals of an effective design is to achieve high cohesion with low coupling.

Larman has described degree of cohesion from low to high

- Very low cohesion:- Class responsible for many things in unrelated areas
- Low cohesion:-Class responsible for many things in related areas
- Moderate cohesion :- Class has few responsibilities in different areas which are related to the class but not to each other
- High Cohesion- Class has few responsibilities in one area and collaborates with other classes to fulfill tasks

Related Patterns: This pattern is related rather complementary to "Low Coupling" and little contradictory to information expert.

Benefits

- Understandability :- Clarity and ease of understanding design is increased
- Maintainability :- Maintenance and enhancements are simplified
- Low coupling is often supported
- Reusability :Small size classes with highly related functionality increases reuse

Design Patterns

A design patterns are **well-proved solution** for solving the specific problem/task.

Now, a question will be arising in your mind what kind of specific problem? Let me explain by taking an example.

Problem Given:

Suppose you want to create a class for which only a single instance (or object) should be created and that single object can be used by all other classes.

Solution:

Singleton design pattern is the best solution of above specific problem. So, every design pattern has **some specification or set of rules** for solving the problems. What are those specifications, you will see later in the types of design patterns.

Advantage of design pattern:

- They are reusable in multiple projects.
- They provide the solutions that help to define the system architecture.
- They capture the software engineering experiences.
- They provide transparency to the design of an application.
- They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
- Design patterns don?t guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

When should we use the design patterns?

We must use the design patterns **during the analysis and requirement phase of** S**DLC**(Software Development Life Cycle).

Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

Categorization of design patterns:

Basically, design patterns are categorized into two parts:

1. Core java (or JSE) Design Patterns.
2. JEE Design Patterns.

Core Java Design Patterns

In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:

- Creational Design Pattern

  Factory Pattern

  Factory Method Pattern

  Abstract Factory Pattern

  Singleton Pattern

  Prototype Pattern

  Builder Pattern.

- Structural Design Pattern

  Adapter Pattern

  Bridge Pattern

  Composite Pattern

  Decorator Pattern

  Facade Pattern

  Flyweight Pattern

  Proxy Pattern

- Behavioral Design Pattern
  - Chain Of Responsibility Pattern
  - Command Pattern
  - Interpreter Pattern
  - Iterator Pattern
- Mediator Pattern
- Memento Pattern
- Observer Pattern
- State Pattern

- Strategy Pattern
- Template Pattern
- Visitor Pattern

Creational Design Pattern

What are creational patterns?

- Design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation
- Make a system independent of the way in which objects are created, composed and represented
- Recurring themes:
  - Encapsulate knowledge about which concrete classes the system uses (so we can change them easily later)
  - Hide how instances of these classes are created and put together (so we can change it easily later)

Benefits of creational patterns

- Creational patterns let you program to an interface defined by an abstract class
- That lets you configure a system with "product" objects that vary widely in structure and functionality
- Example: GUI systems
  - InterViews GUI class library
  - Multiple look-and-feels
  - Abstract Factories for different screen components

- Generic instantiation – Objects are instantiated without having to identify a specific class type in client code (Abstract Factory, Factory)
- Simplicity – Make instantiation easier: callers do not have to write long complex code to instantiate and set up an object (Builder, Prototype pattern)
- Creation constraints – Creational patterns can put bounds on *who* can create objects, *how* they are created, and *when* they are created

Builder Design Pattern

Builder Pattern says that **"construct a complex object from simple objects using step-by-step approach"**

It is mostly used when object can't be created in single step like in the de-serialization of a complex object.

Advantage of Builder Design Pattern

The main advantages of Builder Pattern are as follows:

- It provides clear separation between the construction and representation of an object.
- It provides better control over construction process.
- It supports to change the internal representation of objects.

When to Use Builder Pattern

1. When the process involved in creating an object is extremely complex, with lots of mandatory and optional parameters
2. When an increase in the number of constructor parameters leads to a large list of constructors
3. When client expects different representations for the object that's constructed

Factory Method

Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Defining a "virtual" constructor.
- The new operator considered harmful.

Problem

A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

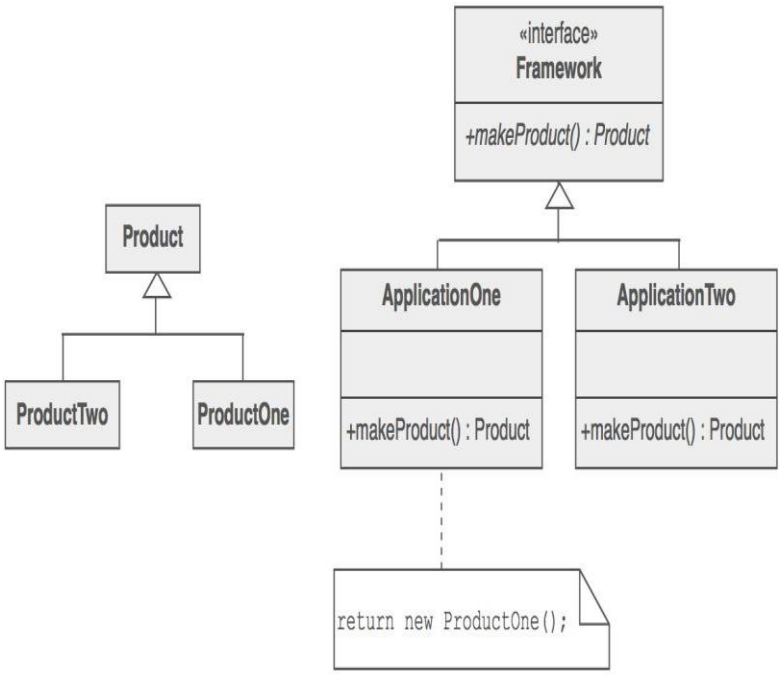
Discussion

Factory Method is to creating objects as Template Method is to implementing an algorithm. A superclass specifies all standard and generic behavior (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client.

- Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation.
- People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation).
- Factory Method is similar to Abstract Factory but without the emphasis on families.
- Factory Methods are routinely specified by an architectural framework, and then implemented by the user of the framework.

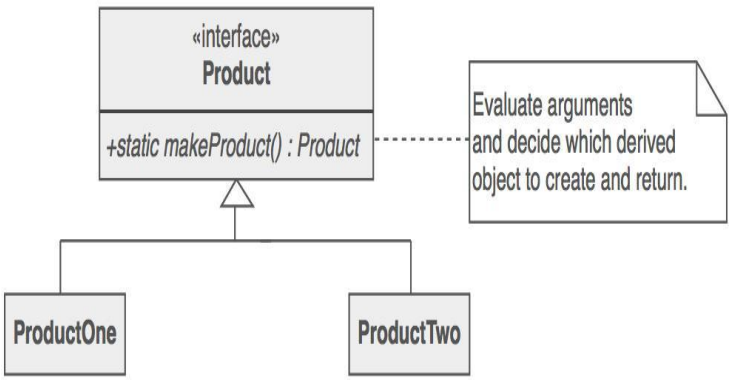
Structure

The implementation of Factory Method discussed in the Gang of Four (below) largely overlaps with that of Abstract Factory. For that reason, the presentation in this chapter focuses on the approach that has become popular since.

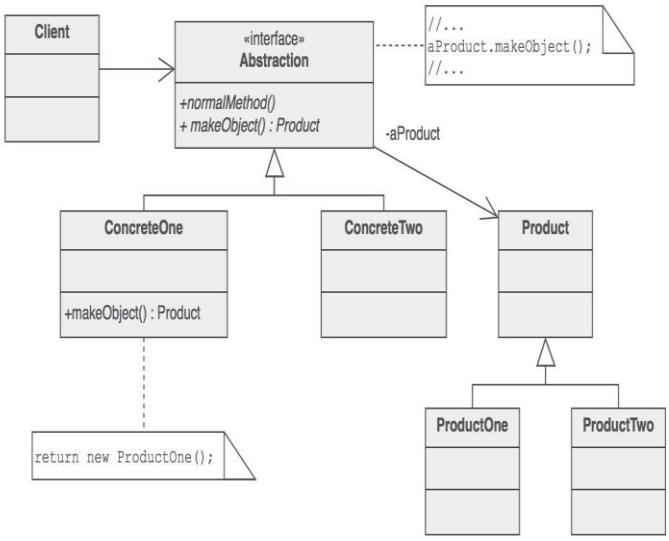


An increasingly popular definition of factory method is: a static method of a class that returns an object of that class' type. But unlike a constructor, the actual object it returns might be an instance of a subclass. Unlike a constructor, an existing object might be reused, instead of a new object created. Unlike a constructor, factory methods can have different and more descriptive

names (e.g.Color.make_RGB_color(float red, float green, float blue) andColor.make_HSB_color(float hue, float saturation, float brightness)

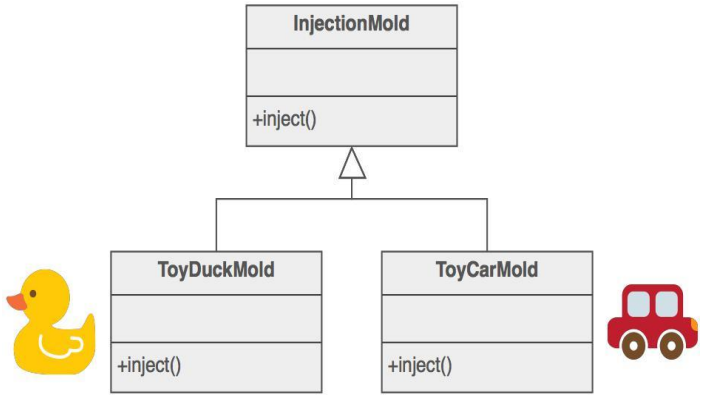


The client is totally decoupled from the implementation details of derived classes. Polymorphic creation is now possible.



Example

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.



Usage of Factory Design Pattern

- When a class doesn't know what sub-classes will be required to create
- When a class wants that its sub-classes specify the objects to be created.
- When the parent classes choose the creation of objects to its sub-classes.

Structural Design pattern

Structural design patterns are concerned with how classes and objects can be composed, to form larger structures.

The structural design patterns **simplifies the structure by identifying the relationships**.

These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

Types of structural design patterns

There are following 7 types of structural design patterns.

1. Adapter Pattern

   Adapting an interface into another according to client expectation.
2. Bridge Pattern

   Separating abstraction (interface) from implementation.
3. Composite Pattern

   Allowing clients to operate on hierarchy of objects.
4. Decorator Pattern

   Adding functionality to an object dynamically.
5. Facade Pattern

   Providing an interface to a set of interfaces.
6. Flyweight Pattern

   Reusing an object by sharing it.
7. proxy Pattern

   Representing another object.

Bridge Design Pattern

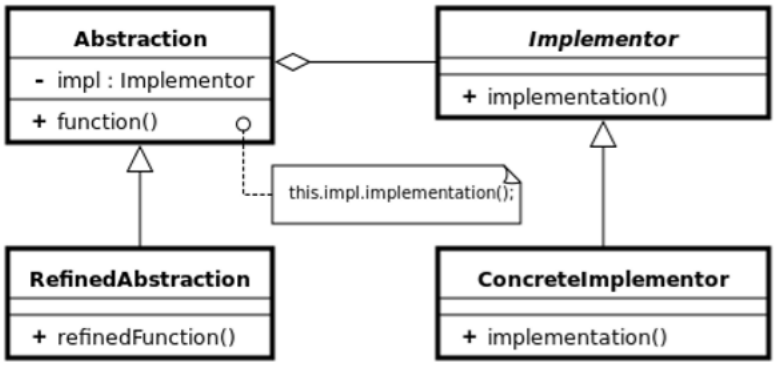
The bridge pattern is a design pattern used in software engineering which is meant to "decouple an abstraction from its implementation so that the two can vary independently". The bridge uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.

When a class varies often, the features of object-oriented programming become very useful because changes to a program's code can be made easily with minimal prior knowledge about the program. The bridge pattern is useful when both the class and what it does vary often. The class itself can be thought of as the implementation and what the class can do as the abstraction. The bridge pattern can also be thought of as two layers of abstraction.

When there is only one fixed implementation, this pattern is known as the Pimpl idiom in the C++ world.

The bridge pattern is often confused with the adapter pattern. In fact, the bridge pattern is often implemented using the class adapter pattern, e.g. in the Java code below.

Variant: The implementation can be decoupled even more by deferring the presence of the implementation to the point where the abstraction is utilized.



- Abstraction (abstract class) defines the abstract interface maintains the Implementor reference.
- RefinedAbstraction (normal class) extends the interface defined by Abstraction
- Implementor (interface)defines the interface for implementation classes
- ConcreteImplementor (normal class)implements the Implementor interface

Advantage of Bridge Pattern

- It enables the separation of implementation from the interface.
- It improves the extensibility.
- It allows the hiding of implementation details from the client.

Usage of Bridge Pattern

- When you don't want a permanent binding between the functional abstraction and its implementation.
- When both the functional abstraction and its implementation need to extended using sub-classes.
- It is mostly used in those places where changes are made in the implementation does not affect the clients.

ADAPTER DESIGN PATTERN:

Intent

An Adapter Pattern says that just **"converts the interface of a class into another interface that a client wants".** In other words, to provide the interface according to client requirement while using the services of a class with a different interface**.**Wrap an existing class with a new interface. The Adapter Pattern is also known as **Wrapper.** Impedance match an old component to a new system.

Adapter is about creating an intermediary abstraction that translates, or maps, the old component to the new system. Clients call methods on the Adapter object which redirects them into calls to the legacy component. This strategy can be implemented either with inheritance or with aggregation. Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class.

Advantage of Adapter Pattern

- It allows two or more previously incompatible objects to interact.
- It allows reusability of existing functionality.

Usage of Adapter pattern:

It is used:

- When an object needs to utilize an existing class with an incompatible interface.
- When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.
- When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.

UML for Adapter Pattern:

There are the following specifications for the adapter pattern:

- **Target Interface:** This is the desired interface class which will be used by the clients.
- **Adapter class:** This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class.
- **Adaptee class:** This is the class which is used by the Adapter class to reuse the existing functionality and modify them for desired use.
- **Client:** This class will interact with the Adapter class.

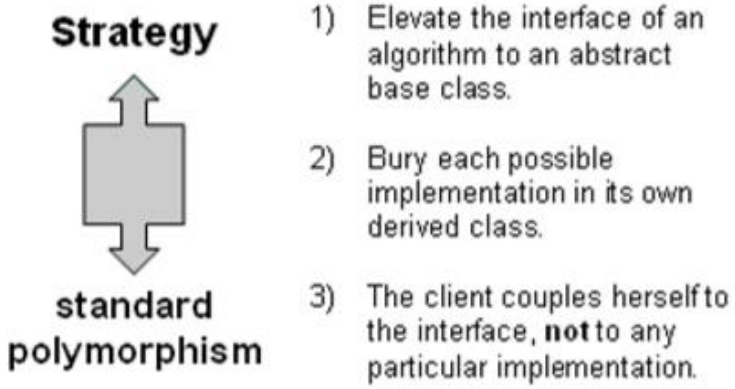
BEHAVIORAL DESIGN PATTERN

A generic value of the software community for years has been, "maximize cohesion and minimize coupling". The object-oriented design approach shown in Figure 21-1 is all about minimizing coupling. Since the client is coupled only to an abstraction (i.e. a useful fiction), and not a particular realization of that abstraction, the client could be said to be practicing "abstract coupling" . an object-oriented variant of the more generic exhortation "minimize coupling".

A more popular characterization of this "abstract coupling" principle is ...

Program to an interface, not an implementation.

Clients should prefer the "additional level of indirection" that an interface (or an abstract base class) affords. The interface captures the abstraction (i.e. the "useful fiction") the client wants to exercise, and the implementations of that interface are effectively hidden.



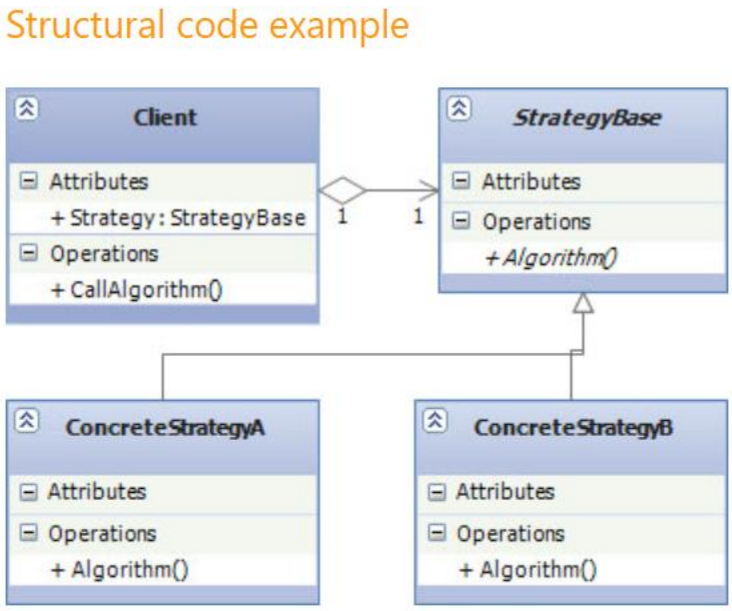
Example:

"A Strategy defines a set of algorithms that can be used interchangeably. Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on tradeoffs between cost, convenience, and time."

STRATEGY DESIGN PATTERN:

The strategy pattern is a design pattern that allows a set of similar algorithms to be defined and encapsulated in their own classes. The algorithm to be used for a particular purpose may then be selected at run-time according to your requirements.

This design pattern can be used in common situations where classes differ only in behavior. In this case it is a good idea to separate these behavior algorithms in separate classes which can be selected at run-time. This pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. This separation allows behaviors may vary independently of clients that use it. It also increases the flexibility of the application allowing to add new algorithms in future.



The UML diagram below describes an implementation of the strategy design pattern. This diagram consists of three parts:

- **Client:** this class uses interchangeable algorithms. It maintains a reference to the StrategyBase object. Sometimes it defines an interface that lets StrategyBase access its data.
- **StrategyBase:** declares an interface common to all supported algorithms. The client uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy:** this is the concrete strategy class inherited from the StrategyBase class. Each instance provides a different algorithm that may be used by the client.

UNIT-V APPLYING DESIGN PATTENS

System sequence diagrams, relation between sequence diagrams and use cases logical architecture and UML package diagram, logical architecture refinement; Case study: The next gen POS system, inception, use case modeling, relating use cases, include, extend and generalization, domain models, domain model refinement.

System sequence diagrams

- A System Sequence Diagram is an artifact that illustrates input and output events related to the system under discussion.
- System Sequence Diagrams are typically associated with use-case realization in the logical view of system development.
- Sequence Diagrams (Not *System* Sequence Diagrams) display object interactions arranged in time sequence.

Sequence Diagram

Sequence Diagrams depict the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the system.
Sequence diagrams can be used to drive out testable user interface requirements.

SSD - System Behavior

- System behaves as **"**Black Box**"**.
- Interior objects are not shown, as they would be on a Sequence Diagram.

System

System Sequence Diagrams are

- Use cases describe-
- How actors interact with system.
- **For a particular scenario of use-case an SSD shows-**

The external actors that interact directly with the system.

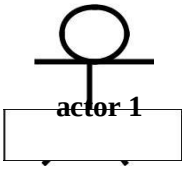
- The System (as a black box).
- The system events that the actors generate.
- The operations of the system in response to the events generated.
- System Sequence Diagrams depict the temporal order of the events.
- System Sequence Diagrams should be done for the main success scenario of the use-case, and frequent and alternative scenarios.

Notations

Object: Objects are instances of classes. Object is represented as a rectangle which contains the name of the object underlined.

Because the system is instantiated, it is shown as an object.

Actor: An Actor is modeled using the ubiquitous symbol, the stick figure.



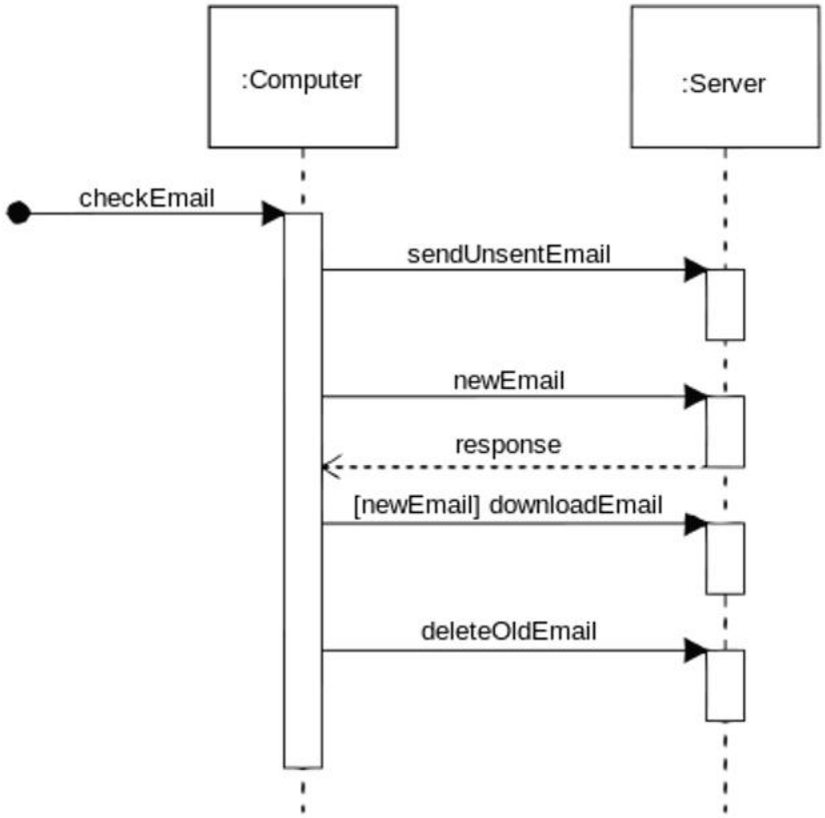
Lifeline: The LifeLine identifies the existence of the object over time. The notation for a Lifeline is a vertical dotted line extending from an object.

Message: Messages, modeled as horizontal arrows between Activations, indicate the communications between objects.

messageName(argument)

Examples of SSD

Following example shows the success scenario of the Computer and Server.



RELATIONSHIP BETWEEN SEQUENCE DIAGRAMS AND USE CASES:

Realizing use cases by means of sequence diagrams is an important part of our analysis. It ensures that we have an accurate and complete class diagram.
The sequence diagrams increase the completeness and understandability of our analysis model. Often, analysts use the sequence diagram to assign responsibilities to classes. The behavior is associated with the class the first time it is required, and then the behavior is reused for every other use case that requires the behavior.
When assigning behaviors or responsibilities to a class while mapping a use case to the analysis model, you must take special care to assign the responsibility to the correct class. The responsibility or behavior belongs to the class if it is something you would do to the thing the class represents.
For example, if our class represented a table and our application must keep track of the physical location of the table, we would expect to find the method MOVE in the class. We also expect the object to maintain all the information associated with an object of a given type.
Therefore, the TABLE class should include the method WHERE LOCATED.
This simple rule of thumb states that a class will include any methods needed to provide information about an object of the given class, will provide necessary methods to manipulate the objects of the given class, and will be responsible for creating and deleting instances of the class.

- System Sequence Diagram is generated from inspection of a use case.
- **Constructing a systems sequence diagram from a use case-**

Draw a line representing the system as a black box.

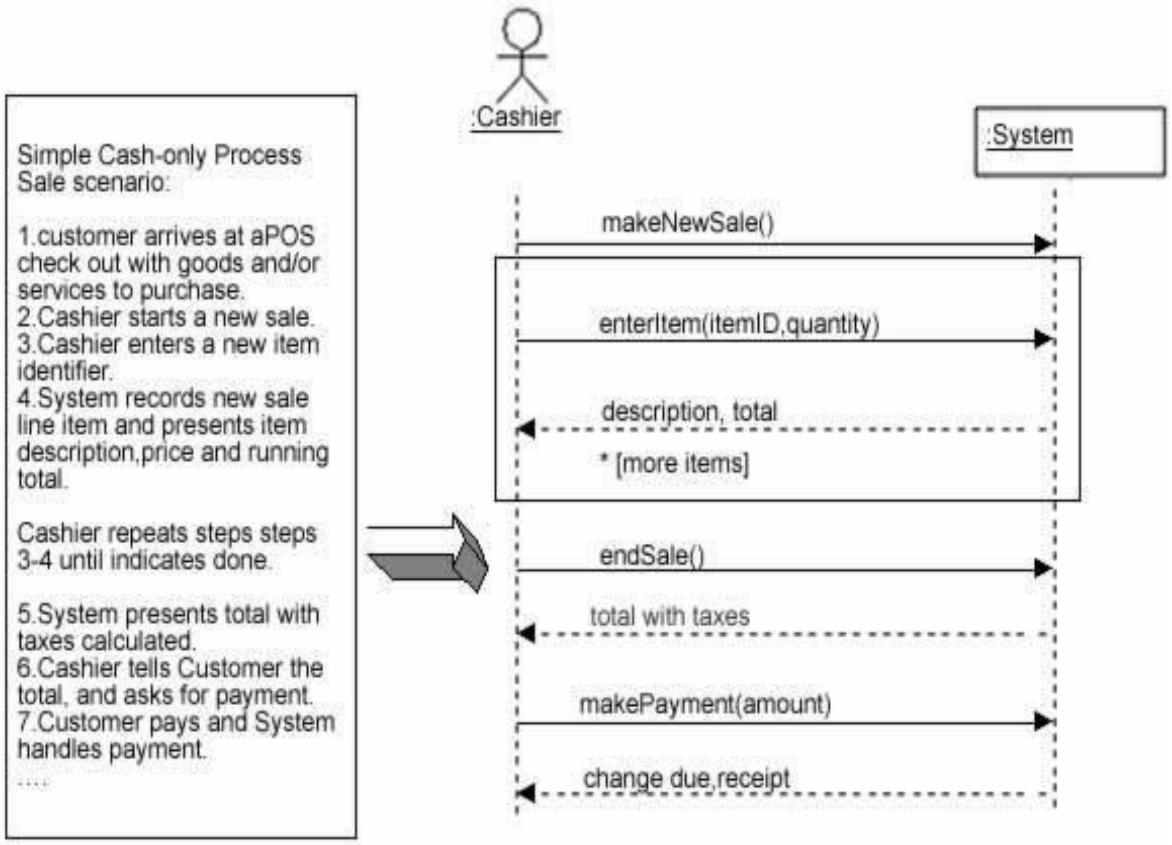
Identify each actor that directly operates on the system. Draw a line for each such actor.

- From the use case, typical course of events text, identify the system (external) events that each actor generates. They will correspond to an entry in the right hand side of the typical use case. Illustrate them on the diagram.

- Optionally, include the use case text to the left of the diagram.

SSDs are derived from Use Cases

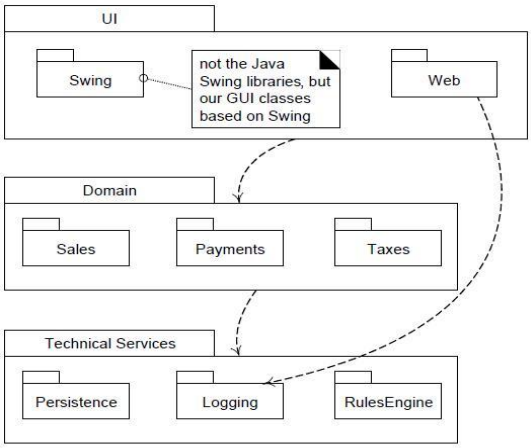
-



LOGICAL ARCHITECTURE AND UML PACKAGE DIAGRAM:

- High level, large scale Architecture Model.
- At this level, the design of a typical OO system is based on several architectural layers, such as a UI layer, an application logic (or "domain") layer, and so forth.
- Goal is to design a logical architecture with layers and partitions using UML package diagrams
- Logical Architecture And Layers
- Logical architecture is the large-scale organization of the software classes into packages (or namespaces), subsystems, and layers.
- It's called the logical architecture because there's no decision about how these elements are deployed across different operating system processes or across physical computers in a network (these latter decisions are part of the deployment architecture).

Logical Architecture and UML Package Diagram



At this level, the design of a typical OO system is based on several architectural layers, such as

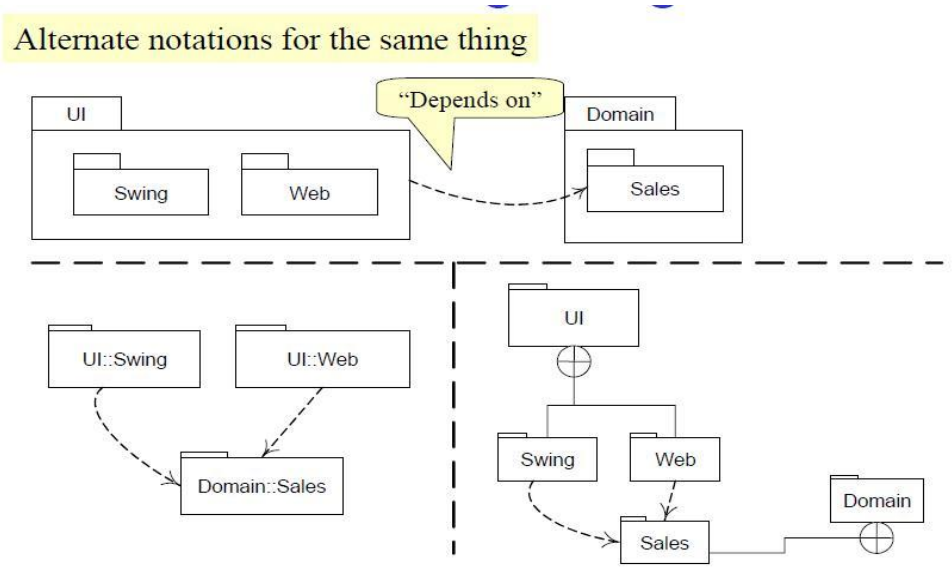
- UI layer
- Application logic (or "domain") layer
- Technical Service
- **Layer** is a coarse-grained grouping of classes, packages, or subsystems that has cohesive (strongly related) responsibilities for a major aspect of the system.
- –E.g. Application Logic and Domain Objects—software objects representing domain concepts (for example, a software class Sale) that fulfill application requirements, such as calculating a sale total.

- Application layer is the focus of Use Cases.
- Higher layers (such as UI layer) call upon services of lower layers, but not normally vice versa.
- In a strict layered architecture, a layer only calls upon the services of the layer directly below it.
- But not in information systems, which have a relaxed layered architecture, in which a higher layer calls upon several lower layers.
- –For example, UI layer may call upon its directly subordinate application logic layer, and also upon elements of a lower technical service layer, for logging and so forth.

Software 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,
- together with their behaviour as specified in the collaborations among those elements
- the composition of these structural and behavioral elements into progressively larger subsystems.

UML Package Diagram



Case study: The next gen POS system,

- A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store.
- It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications, such as a thirdparty tax calculator and inventory control.

- These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).
- A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.
- Furthermore, we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing.
- Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added. Therefore, we will need a mechanism to provide this flexibility and customization.

User-Level Goals

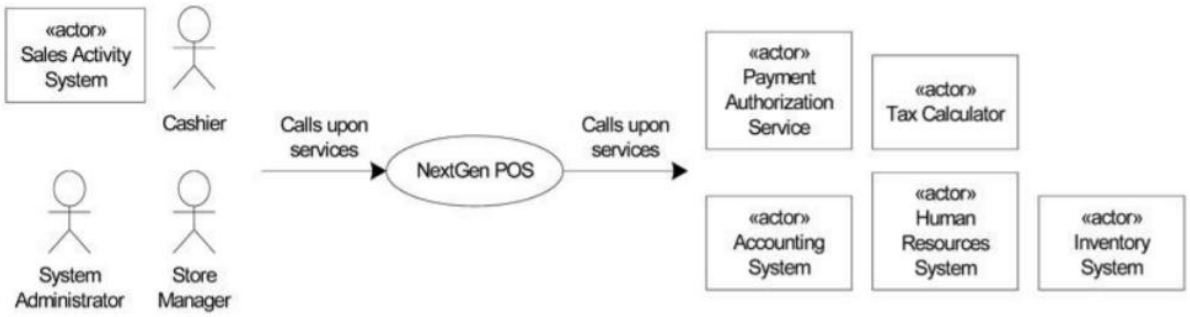
The users (and external systems) need a system to fulfill these goals:

• Cashier: process sales, handle returns, cash in, cash out

• System administrator: manage users, manage security, manage system tables

• Manager: start up, shut down

• Sales activity system: analyze sales data …….



Inceptions:

1. The inception means specifying the beginning of the software project. Most of the software projects get started due to business requirements
2. There may be potential demand from the market for a particular product and then the specific software project needs to be developed.
3. there exist several stakeholders who define the business ideas/ Stakeholders means an entity that takes active participation in project development. In software project development, the stakeholders that are responsible for defining the ideas are business managers, marketing people, and product managers and so on. Their role is to do rough feasibility study and to identify the scope of the project.
4. During the inception a set of context free questions is discussed. the purpose of inception is to decide following things---
    - Find out what is the vision and business case for this project?
    - Is the project feasible to implement?
    - whether to buy or build the software system?

- What is the rough estimate of the project?
- The current development process should be stopped or continued?

5. The purpose of inception phase is not to define all the requirements or to generate a project plan.
6. Thus the purpose of inception is to establish common goals and objectives for the project. Only 10% of requirement gathering can be done during inception.

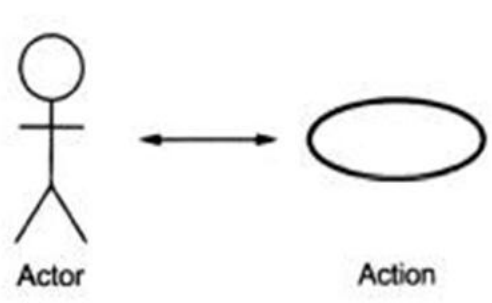
Use case modeling

- During requirement gathering the primary requirements of the system are understood. These basic requirements of the system are called the functional requirements.
- In order to understand the working of these functional requirements, the developers and user create a set of scenarios. These scenarios are nothing but the text stories. These text stories are called as **use cases.**
- The use cases are the textual representation of particular scenario, whereas the **use cse model** is the graphical representation of the scenario. Following table shows then template of the use case -

Use Case template	
Use case	Write the name of the scenario.
Primary Actor	Specify the role of the entity who interacts with the system
Goal in Context	Specify the purpose of this use case scenario
Preconditions	The condition that is to be satisfied before the use case start
Scenario	Sequence of steps that describe the main scenarios
Triggers Extensions points	The triggering event due to which the function stars, is described The extension points describe the extend relationship.
Exceptions	Describe the exceptional situations that may occur during the execution of the use case
Priority	Describe the necessity of the system to be implemented
Open Issues	Certain additional issues that are required for execution of the scenario and that are not specified in the main scenario

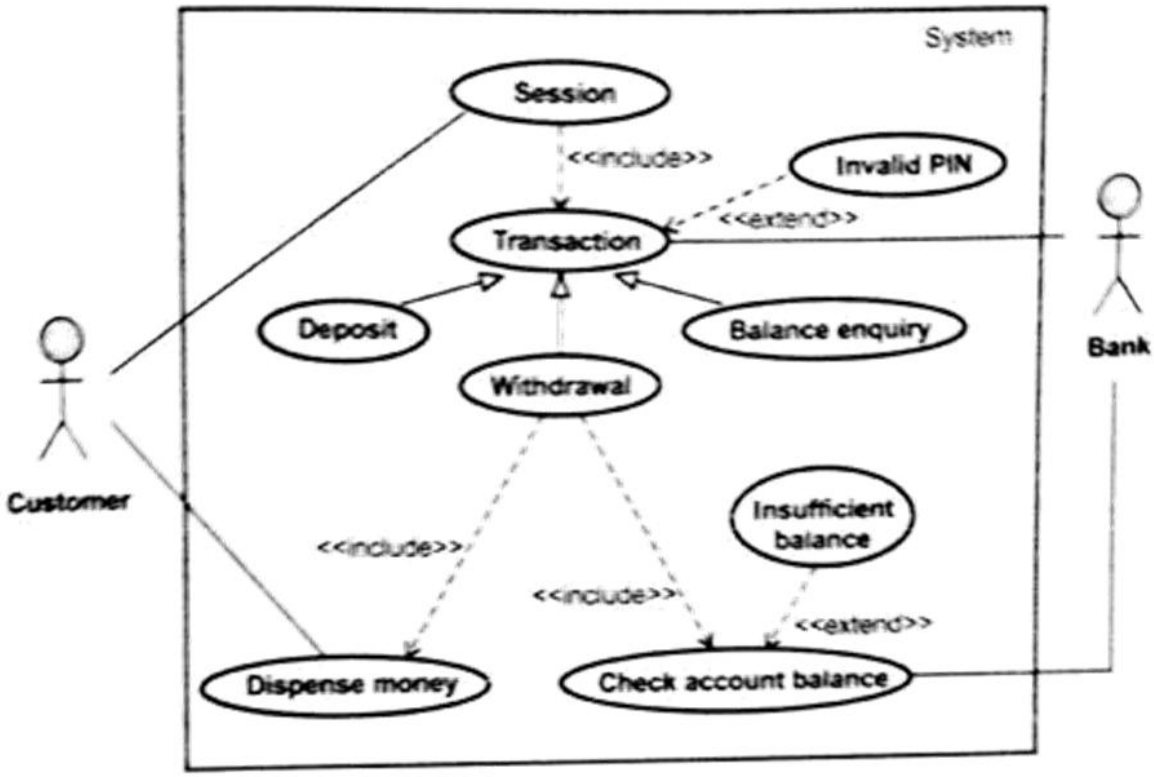
Actors, Scenarios and Use Cases

- Use cases are the fundamentals units of modeling language in which functionalities are distinctly represented.
- The first step in writing use case is to identify the actors. The actors are the entities that use the sytem or product within the context of behavior of the system. Actors represent the role of the people as system gets operated. Actors can be a person, computer system or organization.
- Actor is anything that communicates with the system or product. It is external to the system. Every actor has one or more goals when using the system.



- **Example:** Customer, Inventory database, Accountant
- **Scenario** is a specific sequence of actions or interactions between actor and the system. It is also called as use case instance. The scenarios describe the text stories. For example: scenario of issuing of a book from library, withdrawal of money from ATM.
- Use case a success or failure scenarios in which the actor communicates with the system in order to achieve certain functionality.
- For example : Online purchase system.

Fig: Use case diagram for withdrawal of money from ATM system



Types of Actors

Actor is something that interacts with the system. There are three kinds of actors

I. **Primary actor:** The primary actors are the actors that interact with the system in order to achieve the user goals. For example librarian is the primary actor for the use case issuing of books.
II. **Supporting actor:** The supporting actors are used in conjunction with the primary actors in order to support for the main services. The secondary actors support the system in such a way that the primary actors can perform their task. For example payment validation system is a supporting actor for the on-line purchase system
III. **OFFstage actor:** These actors help in behavior of the system. For example: tax calculation agency is the offstage actor.

Relating use cases include, extend and generalization

Several use cases can be related with each other. Similarity the use cases are also related to the actors who are intended to achieve the goal of the system. The use cases are related to any other entity by using include, extend and generalization relationship.

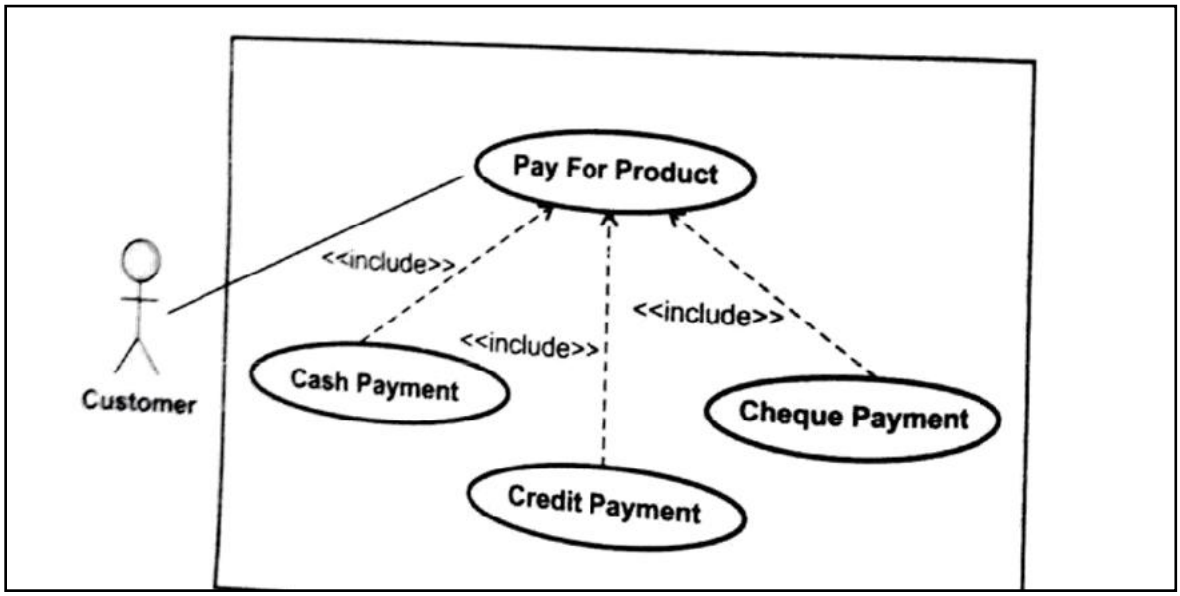
Include

- This is the most commonly used relationship which denotes that a given use case may include another. It is denoted by

----------->
<< Include>>

- The use case at the arrow head position is the sue case which is included by case on the other side of the arrow. When there are multiple steps to carry on single task then this task is divided into the sub-functions and these sub-functions are denoted by the use cases. For example –

Fig: Use case for payment for the product (include – relationship)

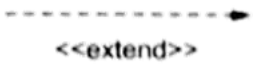


Extend

The extend relationship is used when an extended use case is connected to the base use case. While using the extend use the extension points can be explicitly show in the base use cases. Following are the reasons why extension occur –

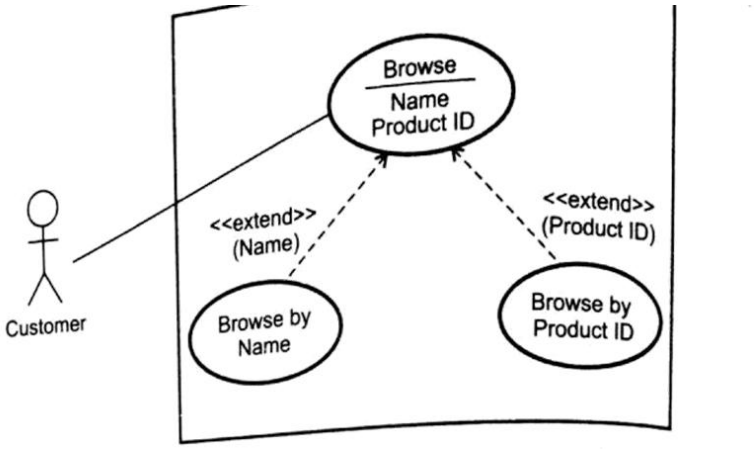
1. When some part of the use case is optional and we need to show separate behavior of the system.
2. When we want to show a subflow of the system when some specific conditions occur.

- This relationship is denoted by



- The extension is conditional. That means its execution is dependent on what has happened while executing the base use case. These conditions ar described by the extend relationship.
- The extension use case may access and modify attributes of the base use case.
- Example : Online purchase system, when user browses for the product, it may browse either by the product name or by product ID.
- Note that extension points are used when we use the extend relationship. The extension points are the labels in the base use case which are referred by the extending use case.

Fig: Use case for browsing product (Extend – relationship)

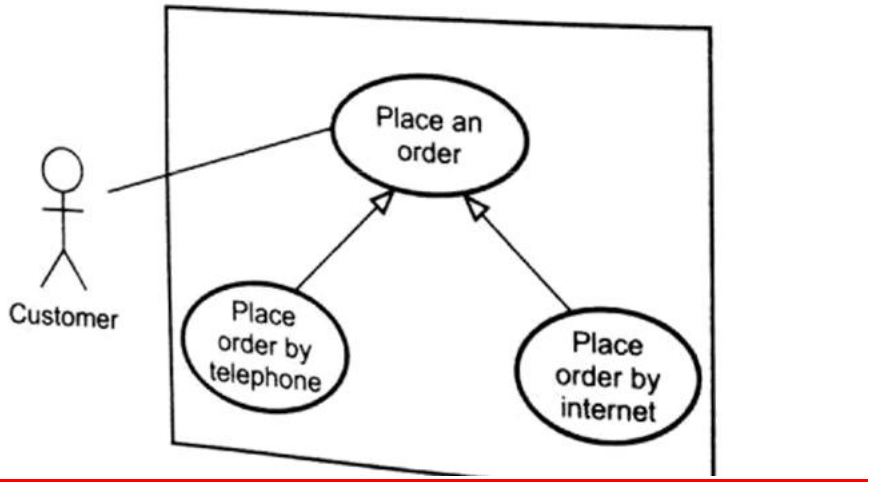


Generalization

- In this type of relationship there are two types of use cases – the parent use case and the child use cases.
- A parent use case may be specialized into one or more child use cases for represent more specific forms of the parent. The child use cases inherit all the structure, behavior and relationships of parents.
- Generalization is used when we find two or more use cases having common structure and behavior and purpose.
- The test word used for generalization relationship is **"kind of".** It is denoted by

- For example : For the Purchase system the customer can place an order telephone or using internet.

Fig: Generalization Use cases



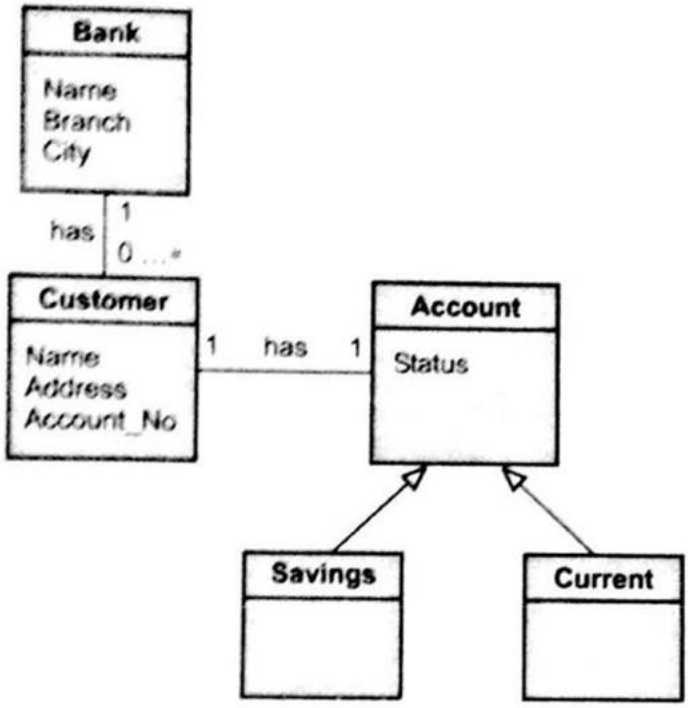
Domain models

Domain model is a visual representation of conceptual classes or real-situation in a domain. The domain models are also called as **conceptual models, domain object models, or analysis object models**

According to Unified Process the domain model is a part of business models. Hence domain model is also called as **business object model.**
The domain model focuses on explaining the core things and products of the business domain.
Using the UML modeling the domain model can be represented using a class diagram. But in these class diagrams there is no operation that is defined. The domain model represents –

- **Domain objects** or **conceptual classes**
- Association or **relationship between the conceptual classes**
- **Attributes** of conceptual classes

Fig: Consider a banking system in which a bank has one or more branches. Each branch has customer which account is present in that branch. The account can be of two types savings account and current account.

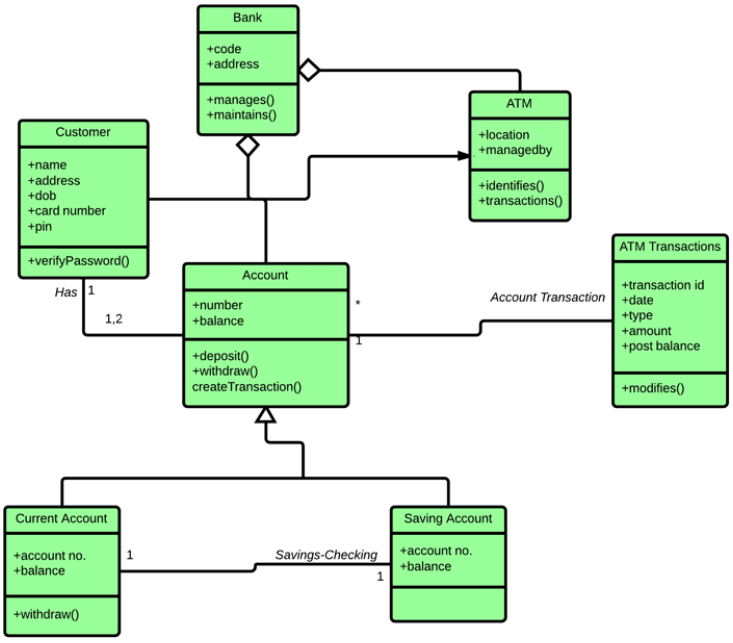


Domain model as a Visual Dictionary
Domain model shows the abstraction of the conceptual classes. The information in domain model can be expressed in plain text. But the conceptual classes and their relationship with other classes can be expressed using visual language using a domain model. Hence domain model is referred as visual dictionary.
Domain model and Software Business Objects
Domain model is a visualization of real world entities and not the representation of software business objects. Following elements are not suitable in domain model –

- Software artifacts such as database or file
- Responsibilities or methods or functions

Domain model refinement

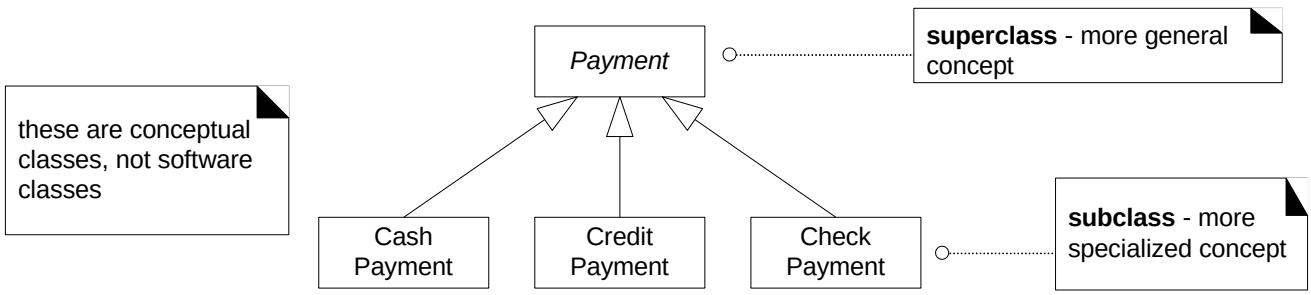
The domain models are created to understand the requirements of the system. A useful domain model captures the essential abstraction and information required for understanding the domain of the targeted system. The domain model can be refined by adding the attributes to the class and showing the associations among these classes. Figure shows the simple class diagram for the ATM system which shows the domain model refinement.



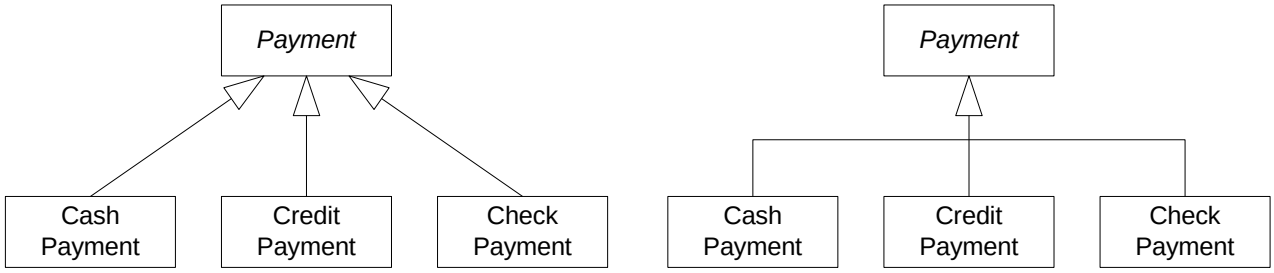
Things not seen before *in the Domain Model*

- Similar to the concepts in the Object Models
- Generalization and specialization
- Conceptual class hierarchies
- Association classes that capture information about the association
- Time intervals
- Packages as a means of organization

Domain classes



They each show the same thing

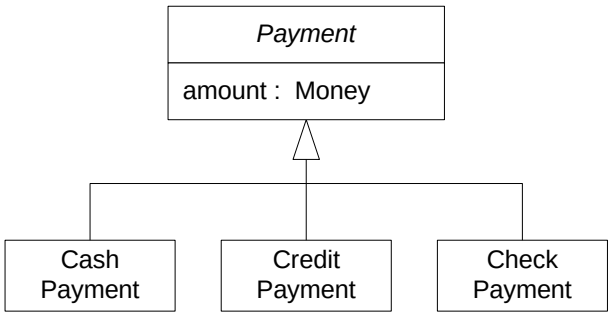


Conceptual Superclasses and Subclasses

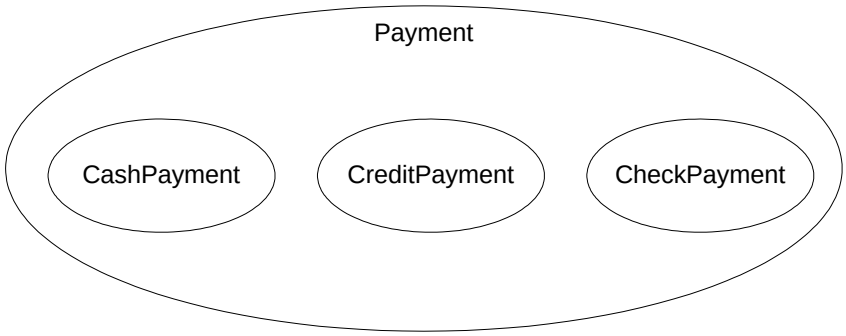
- *Conceptual* superclass is more general than subclass
  - *All members* of subclass must be members of the superclass
  - *100% of superclass definition shall apply* to the subclass

- Subclass "is-a" superclass
  - Woman "is-a" human
  - Man "is-a" human
  - Anything else that is a human?
  - All humans have a heart and a brain.

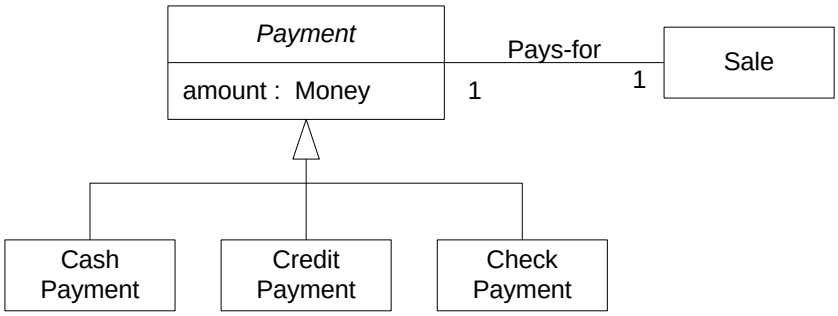
Cash Payment is-a payment.
No other type of payment is possible in this domain, e.g. no gift certifications



- Venn Diagram view:
- All instances of “cash payment” are also members of the “payment class.



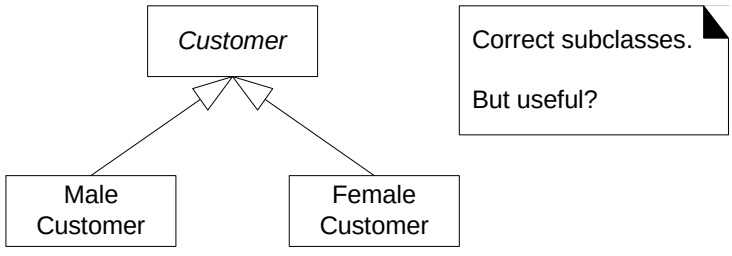
Pays-for applies to all payments!



When to use a subclass

- Start with the super-class and look at differences to find the sub-classes that make sense.
  - Subclass has additional attributes
  - Subclass has additional associations
  - Subclass usage differs from superclass
  - Subclass represents an animate entity that behaves differently

Only useful to POS if men and women pay different amounts, e.g. have specific gender discounts. Useful for age.



When to define a superclass

- Start with a set of sub-classes and look for commonalities that help the model.
  - Potential subclasses represent variations of concept
  - Subclasses meet “is-a” rule and 100% rule
  - All subclasses have common attributes that can be *factored* out
  - All subclasses have the same association that can be *factored* out