17. Students Reference Notes

UNIT – I
INTRODUCTION TO UML

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

Model

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

Why do we model

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

Through modeling, we achieve four aims.

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

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

Principles of Modeling

There are four basic principles of model

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

Object Oriented Modeling

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

1. Algorithmic perspective
2. Object-oriented perspective

Algorithmic Perspective

The traditional view of software development takes an algorithmic perspective.
In this approach, the main building block of all software is the procedure or function.
This view leads developers to focus on issues of control and the decomposition of larger algorithms into smaller ones.
As requirements change and the system grows, systems built with an algorithmic focus turn out to be very hard to maintain.

Object-oriented perspective

The contemporary view of software development takes an object-oriented perspective.
In this approach, the main building block of all software systems is the object or class.
A class is a description of a set of common objects.
Every object has identity, state, and behavior.
Object-oriented development provides the conceptual foundation for assembling systems out of components using technology such as Java Beans or COM+.

An Overview of UML

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

  The UML is a language for
  - Visualizing
  - Specifying
  - Constructing
  - Documenting

- **Visualizing** The UML is more than just a bunch of graphical symbols. Rather, behind each symbol in the UML notation is a well-defined semantics. In this manner, one developer can write a model in the UML, and another developer, or even another tool, can interpret that model unambiguously
- **Specifying** means building models that are precise, unambiguous, and complete.
- **Constructing** the UML is not a visual programming language, but its models can be directly connected to a variety of programming languages

- **Documenting** a healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include
    - Requirements
    - Architecture
    - Design
    - Source code
    - Project plans
    - Tests
    - Prototypes
    - Releases

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

1. Things
2. Relationships
3. Diagrams

Things in the UML

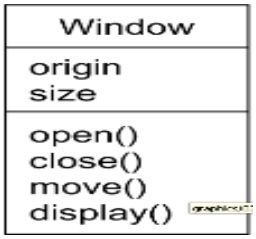
There are four kinds of things in the UML:

Structural things
Behavioral things
Grouping things
Annotational things

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

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

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



Interface

Interface is a collection of operations that specify a service of a class or component.
An interface therefore describes the externally visible behavior of that element.
An interface might represent the complete behavior of a class or component or only a part of that behavior.

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



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

Graphically, collaboration is rendered as an ellipse with dashed lines, usually including only its name

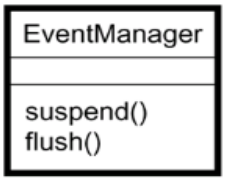


Usecase

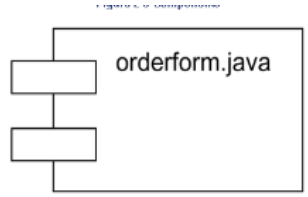
- Use case is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor
- Use case is used to structure the behavioral things in a model.
- A use case is realized by collaboration. Graphically, a use case is rendered as an ellipse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



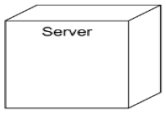
Active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations



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



Node is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. Graphically, a node is rendered as a cube, usually including only its name



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

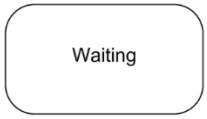
Interaction

Interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose
An interaction involves a number of other elements, including messages, action sequences and links
Graphically a message is rendered as a directed line, almost always including the name of its operation



State Machine

State machine is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events
State machine involves a number of other elements, including states, transitions, events and activities
Graphically, a state is rendered as a rounded rectangle, usually including its name and its sub states

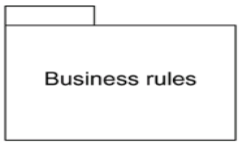


Grouping Things:-

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

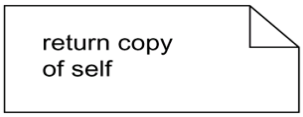
Package:-

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



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

A note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.
Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment

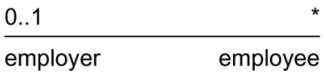


Relationships in the UML: There are four kinds of relationships in the UML:

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

Dependency:-
Dependency is a semantic relationship between two things in which a change to one thing may affect the semantics of the other thing
Graphically a dependency is rendered as a dashed line, possibly directed, and occasionally including a label

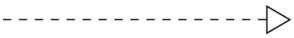
Association is a structural relationship that describes a set of links, a link being a connection among objects.
Graphically an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names



Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. 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. Graphically a realization relationship is rendered as a cross between a generalization and a dependency relationship



Diagrams in the UML

- **Diagram** is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).
- In theory, a diagram may contain any combination of things and relationships.
- For this reason, the UML includes nine such diagrams:
    - Class diagram
    - Object diagram
    - Use case diagram
    - Sequence diagram
    - Collaboration diagram
    - State chart diagram
    - Activity diagram
    - Component diagram
    - Deployment diagram

Class diagram

A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
Class diagrams that include active classes address the static process view of a system.

Object diagram

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

Use case diagram

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

Interaction Diagrams

Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams
Interaction diagrams address the dynamic view of a system
A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages
A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages
Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other

State chart diagram

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

Activity diagram

An activity diagram is a special kind of a state chart diagram that shows the flow from activity to activity within a system
Activity diagrams address the dynamic view of a system
They are especially important in modeling the function of a system and emphasize the flow of control among objects

Component diagram

- A component diagram shows the organizations and dependencies among a set of components.
- Component diagrams address the static implementation view of a system
- They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations

Deployment diagram

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

Rules of the UML

The UML has semantic rules for

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

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

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

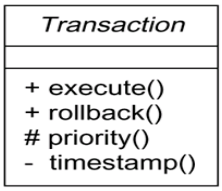
Common Mechanisms in the UML

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

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

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

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



Extensibility Mechanisms

The UML's extensibility mechanisms include

1. Stereotypes
2. Tagged values
3. Constraints

Stereotype

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

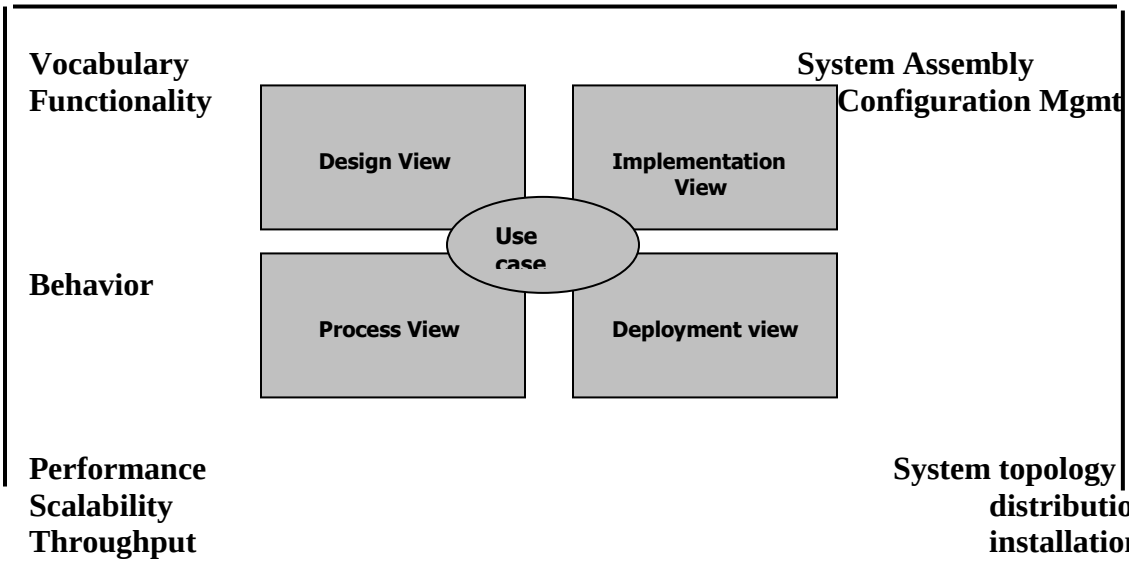
Architecture

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

Architecture is the set of significant decisions about

The organization of a software system
The selection of the structural elements and their interfaces by which the system is composed
Their behavior, as specified in the collaborations among those elements
The composition of these structural and behavioral elements into progressively larger subsystems
The architectural style that guides this organization: the static and dynamic elements and their interfaces, their collaborations, and their composition.

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



Modeling a System's Architecture

Use case view

The use case view of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers.
With the UML, the static aspects of this view are captured in use case diagrams
The dynamic aspects of this view are captured in interaction diagrams, state chart diagrams, and activity diagrams.

Design View

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

Process View

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

Implementation View

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

Deployment Diagram

The deployment view of a system encompasses the nodes that form the system's hardware topology on which the system executes.
This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system.

Software Development Life Cycle

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

Iterative and incremental

Use case driven means that use cases are used as a primary artifact for establishing the desiredbehavior of the system, for verifying and validating the system's architecture, for testing, and for communicating among the stakeholders of the project.
Architecture-centric means that a system's architecture is used as a primary artifact forconceptualizing, constructing, managing, and evolving the system under development.
iterative process is one that involves managing a stream of executable releases. An is one that involves the continuous integration of the system's architecture to produce these releases, with each new release embodying incremental improvements over the other. Together, an iterative and incremental process is *risk-driven*, meaning that each new release is focused on attacking and reducing the most significant risks to the success of the project.
This use case driven, architecture-centric, and iterative/incremental process can be broken into phases. A *phase* is the span of time between two major milestones of the process. There are four phases in the software development life cycle: inception, elaboration, construction, and transition. In the diagram, workflows are plotted against these phases, showing their varying degrees of focus over time.

Inception is the first phase of the process, when the seed idea for the development is brought upto 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 aredefined. 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 executablearchitectural 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 theuser 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.

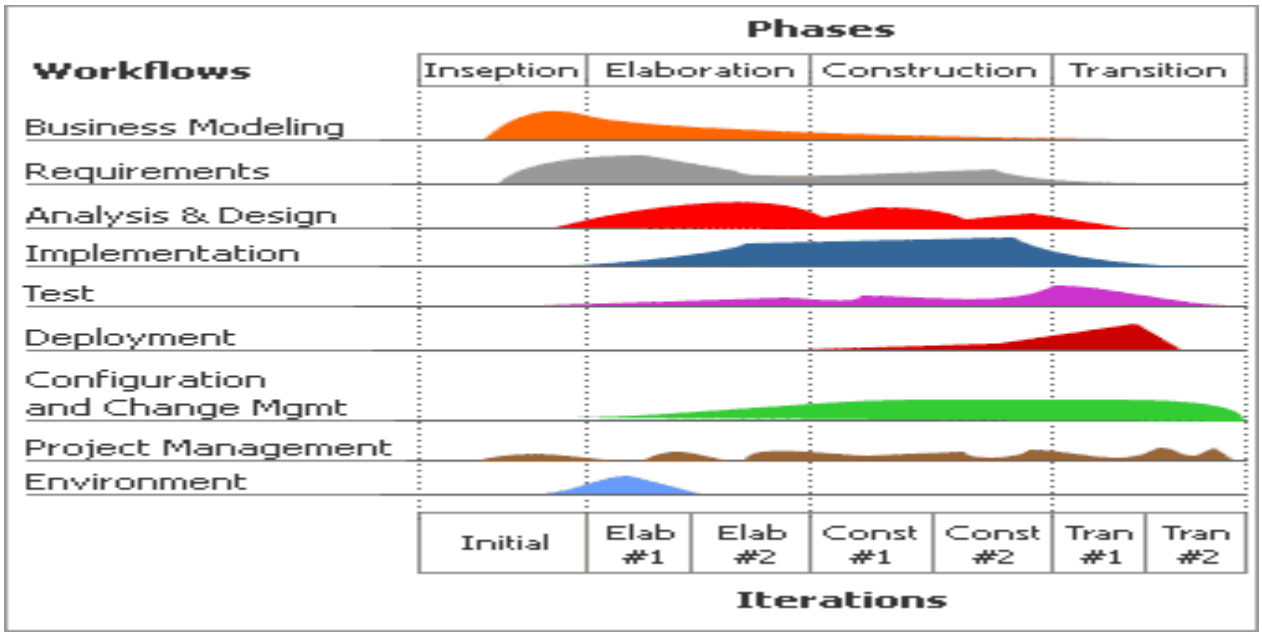
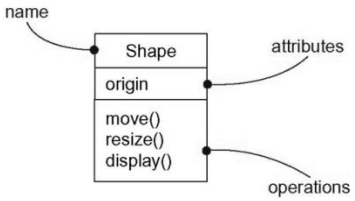


Fig: Software Development Cycle

BASIC STRUCTURAL MODELING

Class

- 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.
- The UML provides a graphical representation of class



Graphical Representation of Class in UML
Terms and Concepts

Names

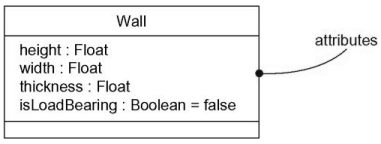
Every class must have a name that distinguishes it from other classes.
A name is a textual string that name alone is known as a simple name;
a path name is the class name prefixed by the name of the package in which that class lives.



Simple Name **Path Name**

Attributes

- An attribute is a named property of a class that describes a 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 thing you are modeling that is shared by all objects of that class
- You can further specify an attribute by stating its class and possibly a default initial value

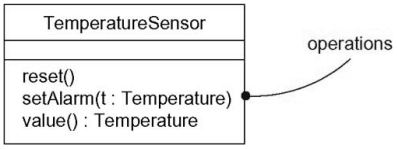


Attributes and Their Class

Operations

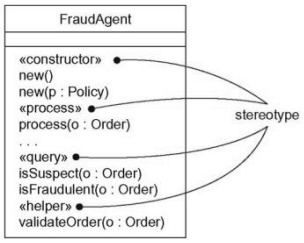
- An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior.

- A class may have any number of operations or no operations at all
- Graphically, operations are listed in a compartment just below the class attributes
- You can specify an operation by stating its signature, covering the name, type, and default value of all parameters and a return type



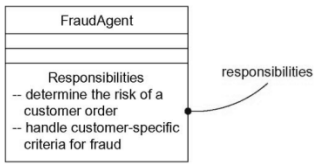
Organizing Attributes and Operations

To better organize long lists of attributes and operations, you can also prefix each group with a descriptive category by using stereotypes



Responsibilities

- A Responsibility is a contract or an obligation of a class
- When you model classes, a good starting point is to specify the responsibilities of the things in your vocabulary.
- A class may have any number of responsibilities, although, in practice, every well-structured class has at least one responsibility and at most just a handful.
- Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon



Common Modeling Techniques

Modeling the Vocabulary of a System

- You'll use classes most commonly to model abstractions that are drawn from the problem you are trying to solve or from the technology you are using to implement a solution to that problem.
- They represent the things that are important to users and to implementers
- To model the vocabulary of a system

- Identify those things that users or implementers use to describe the problem or solution.
- Use CRC cards and use case-based analysis to help find these abstractions.
- For each abstraction, identify a set of responsibilities.
- Provide the attributes and operations that are needed to carry out these responsibilities for each class.

Modeling the Distribution of Responsibilities in a System

- Once you start modeling more than just a handful of classes, you will want to be sure that your abstractions provide a balanced set of responsibilities.
- To model the distribution of responsibilities in a system

  - Identify a set of classes that work together closely to carry out 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 Non software Things

- Sometimes, the things you model may never have an analog in software
- Your application might not have any software that represents them
- To model non software things

  - 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 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
To model primitive types

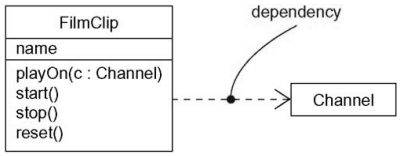
Model the thing you are abstracting as a type or an enumeration, which is rendered using class notation with the appropriate stereotype.
If you need to specify the range of values associated with this type, use constraints.

Relationships

In the UML, the ways that things can connect to one another, either logically or physically, are modeled as relationships.
Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships
In object-oriented modeling, there are three kinds of relationships that are most important:
Dependencies
Generalizations
Associations

Dependency

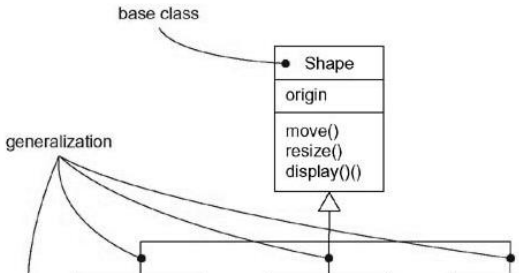
A dependency is a using relationship that states that a change in specification of one thing may affect another thing that uses it but not necessarily the reverse.
Graphically dependency is rendered as a dashed directed line, directed to the thing being depended on.
Most often, you will use dependencies in the context of classes to show that one class uses another class as an argument in the signature of an operation



Dependencies

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 means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations
- An operation of a child that has the same signature as an operation in a parent overrides the operation of the parent; this is known as polymorphism.
- Graphically generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent



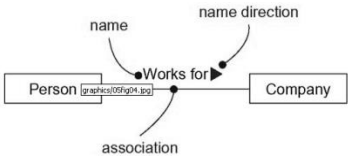
Generalization

Association

- An association is a structural relationship that specifies that objects of one thing are connected to objects of another
- An association that connects exactly two classes is called a binary association
- An associations that connect more than two classes; these are called n-ary associations.
- Graphically, an association is rendered as a solid line connecting the same or different classes.
- Beyond this basic form, there are four adornments that apply to associations

Name

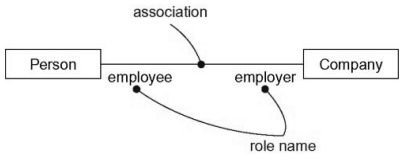
- An association can have a name, and you use that name to describe the nature of the relationship



Association Names

Role

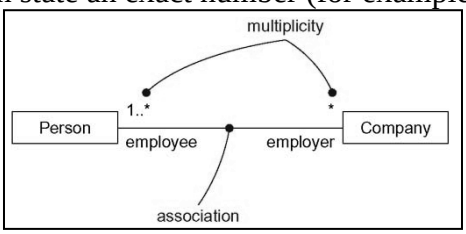
- When a class participates in an association, it has a specific role that it plays in that relationship;
- The same class can play the same or different roles in other associations.
- An instance of an association is called a link



Role Names

Multiplicity

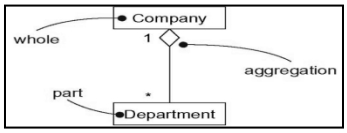
- In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association
- This "how many" is called the multiplicity of an association's role
- You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..*), or one or more (1..*). You can even state an exact number (for example, 3).



Multiplicity

Aggregation

- 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, meaning that an object of the whole has objects of the part
- Aggregation is really just a special kind of association and is specified by adorning a plain association with an open diamond at the whole end



Aggregation

Common Modeling Techniques

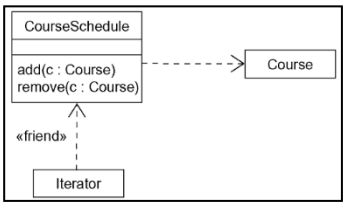
Modeling Simple Dependencies

The most common kind of dependency relationship is the connection between a class that only uses another class as a parameter to an operation.
To model this using relationship

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

The following figure shows a set of classes drawn from a system that manages the assignment of students and instructors to courses in a university.

This figure shows a dependency from Course Schedule to Course, because Course is used in both the add and remove operations of Course Schedule.
The dependency from Iterator shows that the Iterator uses the Course Schedule; the Course Schedule knows nothing about the Iterator. The dependency is marked with a stereotype, which specifies that this is not a plain dependency, but, rather, it represents a friend, as in C++.

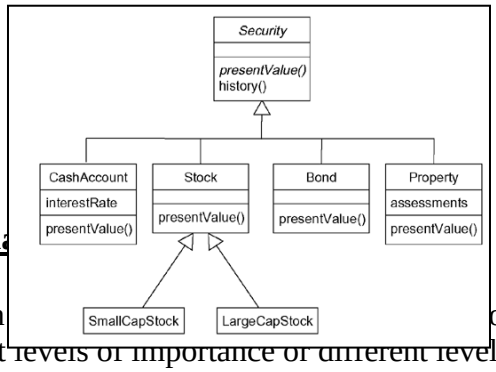


Dependency Relationships

Modeling Single Inheritance

To model inheritance relationships

- Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.
- Elevate these common responsibilities, attributes, and operations to a more general class. If necessary, create a new class to which you can assign these
- 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.



Modeling Structural Rela

- When you model with onships, you are modeling classes that represent different levels of importance or different levels of abstraction
- Given a generalization relationship between two classes, the child inherits from its parent but the parent has no specific knowledge of its children.
- Dependency and generalization relationships are one-sided.

- Associations are, by default, bidirectional; you can limit their direction
- Given an association between two classes, both rely on the other in some way, and you can navigate in either direction
- An association specifies a structural path across which objects of the classes interact.

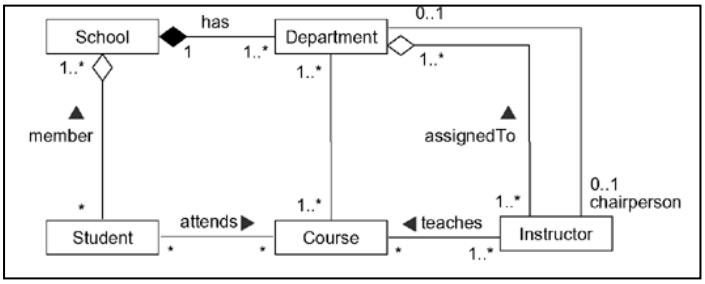
To model structural relationships

- 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.

- 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.

- 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).

- 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



Structural Relationships

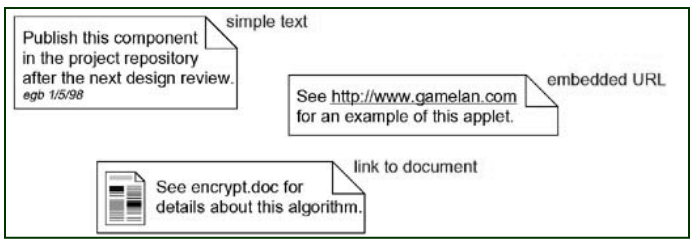
Common Mechanisms

Note

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.

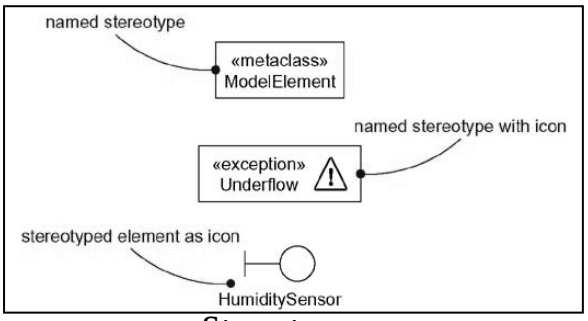
A note may contain any combination of text or graphics



Notes

Stereotypes

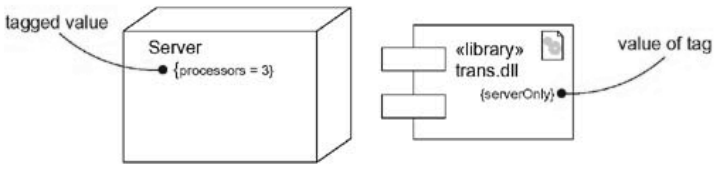
A stereotype is an extension of the vocabulary of the UML, allowing you to create new kinds of building blocks similar to existing ones but specific to your problem.
Graphically, a stereotype is rendered as a name enclosed by guillemets and placed above the name of another element



Stereotypes

Tagged Values

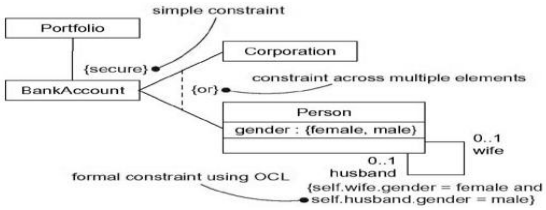
- Every thing in the UML has its own set of properties: classes have names, attributes, and operations; associations have names and two or more ends (each with its own properties); and so on.
- With stereotypes, you can add new things to the UML; with tagged values, you can add new properties.
- A tagged value is not the same as a class attribute. Rather, you can think of a tagged value as metadata because its value applies to the element itself, not its instances.
- 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.
- In its simplest form, 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).



Constraints

A constraint specifies conditions that must be held true for the model to be well-formed.
A constraint is rendered as a string enclosed by brackets and placed near the associated element

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.



Common Modeling Techniques

Modeling Comments

The most common purpose for which you'll use notes is to write down free-form observations, reviews, or explanations.

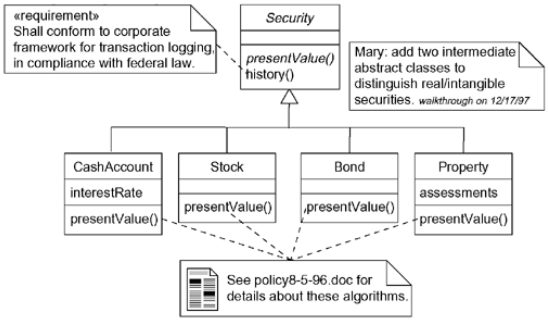
By putting these comments directly in your models, your models can become a common repository for all the disparate artifacts you'll create during development.

To model a comment,

Put your comment as text in a note and place it adjacent to the element to which it refers

Remember that you can hide or make visible the elements of your model as you see fit.

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

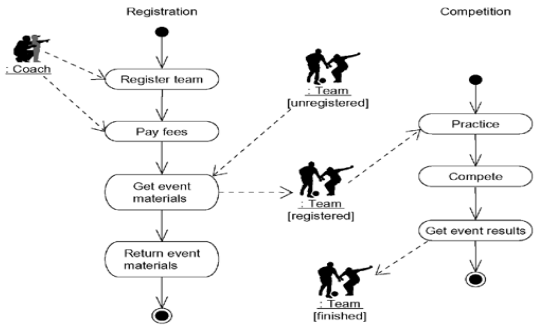


Modeling Comments

Modeling New Building Blocks

- The UML's building blocks—classes, interfaces, collaborations, components, nodes, associations, and so on—are generic enough to address most of the things you'll want to model.
- However, if you want to extend your modeling vocabulary or give distinctive visual cues to certain kinds of abstractions that often appear in your domain, you need to use stereotypes
- To model new building blocks,
    - Make sure there's not already a way to express what you want by using basic UML

- 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 and define a new stereotype for that thing
- 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.
- If you want these stereotype elements to have a distinctive visual cue, define a new icon for the stereotype



Modeling New Building Blocks

Modeling New Properties

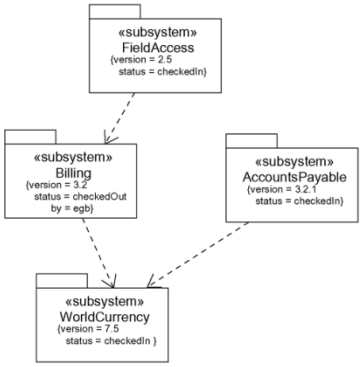
The basic properties of the UML's building blocks—attributes and operations for classes, the contents of packages, and so on—are generic enough to address most of the things you'll want to model.

However, if you want to extend the properties of these basic building blocks, you need to use tagged values.

To model new properties,

First, make sure there's not already a way to express what you want by using basic UML

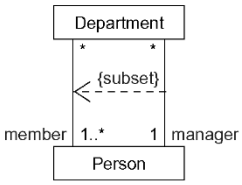
If you're convinced there's no other way to express these semantics, add this new property to an individual element or a stereotype.



Modeling New Properties

Modeling New Semantics

When you create a model using the UML, you work within the rules the UML lays down
However, if you find yourself needing to express new semantics about which the UML is silent or that you need to modify the UML's rules, then you need to write a constraint.
To model new semantics,
First, make sure there's not already a way to express what you want by using basic UML
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
If you need to specify your semantics more precisely and formally, write your new semantics using OCL.



Modeling New Semantics

Diagrams

- When you view a software system from any perspective using the UML, you use diagrams to organize the elements of interest.
- The UML defines nine kinds of diagrams, which you can mix and match to assemble each view.
- Of course, you are not limited to these nine diagrams. In the UML, these nine are defined because they represent the most common packaging of viewed elements. To fit the needs of your project or organization, you can create your own kinds of diagrams to view UML elements in different ways.
- You'll use the UML's diagrams in two basic ways:
    - to specify models from which you'll construct an executable system (forward engineering)
    - and to reconstruct models from parts of an executable system (reverse engineering).

System

- A system is a collection of subsystems organized to accomplish a purpose and described by a set of models, possibly from different viewpoints

SubSystem

- A subsystem is a grouping of elements, of which some constitute a specification of the behavior offered by the other contained elements.

Model

- A model is a semantically closed abstraction of a system, meaning that it represents a complete and self-consistent simplification of reality, created in order to better understand the system. In the context of architecture

View

- view is a projection into the organization and structure of a system's model, focused on one aspect of that system

Diagram

- A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).
- A diagram is just a graphical projection into the elements that make up a system
- Each diagram provides a view into the elements that make up the system
- Typically, you'll view the static parts of a system using one of the four following diagrams.

Class diagram
Object diagram
Component diagram
Deployment diagram
You'll often use five additional diagrams to view the dynamic parts of a system.
Use case diagram
Sequence diagram
Collaboration diagram
State chart diagram
Activity diagram
The UML defines these nine kinds of diagrams.

- Every diagram you create will most likely be one of these nine or occasionally of another kind
- Every diagram must have a name that's unique in its context so that you can refer to a specific diagram and distinguish one from another
- You can project any combination of elements in the UML in the same diagram. For example, you might show both classes and objects in the same diagram

Structural Diagrams

- The UML's four structural diagrams exist to visualize, specify, construct, and document the static aspects of a system.
- 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

Class Diagram

- We use class diagrams to illustrate the static design view of a system.
- Class diagrams are the most common diagram found in modeling object-oriented systems.

- A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- Class diagrams that include active classes are used to address the static process view of a system.

Object Diagram

- Object diagrams address the static design view or static process view of a system just as do class diagrams, but from the perspective of real or prototypical cases.
- An object diagram shows a set of objects and their relationships.
- You use object diagrams to illustrate data structures, the static snapshots of instances of the things found in class diagrams.

Component Diagram

- We use component diagrams to illustrate the static implementation view of a system.
- A component diagram shows a set of components and their relationships.
- Component diagrams are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

Deployment Diagram

- We use deployment diagrams to illustrate the static deployment view of an architecture.
- A deployment diagram shows a set of nodes and their relationships.
- Deployment diagrams are related to component diagrams in that a node typically encloses one or more components.

Behavioral Diagrams

- The UML's five behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system.
- The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.
  - 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
  - State chart diagram : Focused on the changing state of a system driven by events
  - Activity diagram : Focused on the flow of control from activity to activity

Use Case Diagram

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

Sequence Diagram

- We use sequence diagrams to illustrate the dynamic view of a system.
- A sequence diagram is an interaction diagram that emphasizes the time ordering of messages.
- A sequence diagram shows a set of objects and the messages sent and received by those objects.
- The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.

Collaboration Diagram

- We use collaboration diagrams to illustrate the dynamic view of a system.
- A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.
- A collaboration diagram shows a set of objects, links among those objects, and messages sent and received by those objects.
- The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.

***** Sequence and collaboration diagrams are isomorphic, meaning that you can convert from one to the other without loss of information.

State chart Diagram

We use state chart diagrams to illustrate the dynamic view of a system.
They are especially important in modeling the behavior of an interface, class, or collaboration.
A state chart diagram shows a state machine, consisting of states, transitions, events, and activities.
State chart diagrams emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

Activity Diagram

We use activity diagrams to illustrate the dynamic view of a system.
Activity diagrams are especially important in modeling the function of a system.
Activity diagrams emphasize the flow of control among objects.
An activity diagram shows the flow from activity to activity within a system.
An activity shows a set of activities, the sequential or branching flow from activity to activity, and objects that act and are acted upon.

Common Modeling Techniques

Modeling Different Views of a System

- When you model a system from different views, you are in effect constructing your system simultaneously from multiple dimensions
- To model a system from different views,
    - Decide which views you need to best express the architecture of your system and to expose the technical risks to your project

- For each of these views, decide which artifacts you need to create to capture the essential details of that view.
- 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.
- For example, if you are modeling a simple monolithic application that runs on a single machine, you might need only the following handful of diagrams

• Use case view	:	Use case diagrams
• Design view	:	Class diagrams (for structural modeling) Interaction diagrams (for behavioral modeling)
• Process view	:	None rquired
• Implementation view	:	None required
• Deployment view	:	None required

- If yours is a reactive system or if it focuses on process flow, you'll probably want to include statechart diagrams and activity diagrams, respectively, to model your system's behavior.

- Similarly, if yours is a client/server system, you'll probably want to include component diagrams and deployment diagrams to model the physical details of your system.

- Finally, if you are modeling a complex, distributed system, you'll need to employ the full range of the UML's diagrams in order to express the architecture of your system and the technical risks to your project, as in the following.

• Use case view	:	Use case diagrams Activity diagrams (for behavioral modeling)
• Design view	:	* Class diagrams (for structural modeling) * Interaction diagrams (for behavioral modeling) * State chart diagrams (for behavioral modeling)
• Process view	:	* Class diagrams (for structural modeling) * Interaction diagrams (for behavioral modeling)
• Implementation view	:	Component diagram
• Deployment view	:	Deployment diagrams

Mo involved in development who need the same view of the system but at different levels of abstraction

- Basically, there are two ways to model a system at different levels of abstraction:
  - By presenting diagrams with different levels of detail against the same model
  - By creating models at different levels of abstraction with diagrams that trace from one model to another.
- To model a system at different levels of abstraction by presenting diagrams with different levels of detail,
  - 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

- If she is using the model to present a conceptual model to an end user, she'll need 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 your model:

Building blocks and relationships:

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

Adornments:

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

Flow:

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

- The main advantage of this approach is that you are always modeling from a common semantic repository.

- The main disadvantage of this approach is that changes from diagrams at one level of abstraction may make obsolete diagrams at a different level of abstraction.

To model a system at different levels of abstraction by creating models at different levels of abstraction,

- Consider the needs of your readers and decide on the level of abstraction that each should view, forming a separate model for each level.
- In general, populate your models that are at a high level of abstraction with simple abstractions and your models that are at a low level of abstraction with detailed abstractions. Establish trace dependencies among the related elements of different models.
- In practice, if you follow the five views of an architecture, there are four common situations you'll encounter when modeling a system at different levels of abstraction:

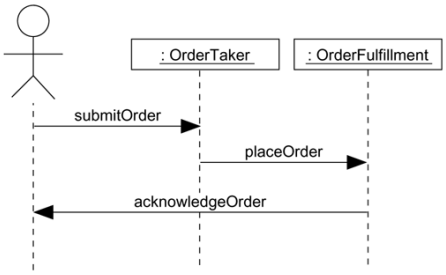
Use cases and their realization:
Use cases in a use case model will trace to collaborations in a design model.
Collaborations and their realization:
Collaborations will trace to a society of classes that work together to carry out the collaboration.
Components and their design:
Components in an implementation model will trace to the elements in a design model.

Nodes and their components:

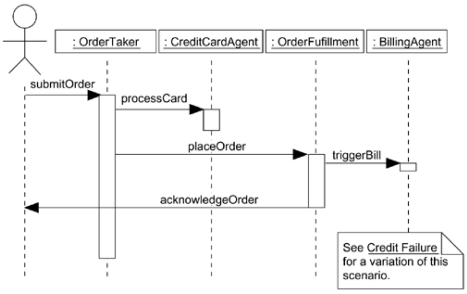
Nodes in a deployment model will trace to components in an implementation model.

The main advantage of the approach is that diagrams at different levels of abstraction remain more loosely coupled. This means that changes in one model will have less direct effect on other models.

The main disadvantage of this approach is that you must spend resources to keep these models and their diagrams synchronized



Interaction Diagram at a High Level of Abstraction



Interaction at a Low Level of Abstraction

* Both of these diagrams work against the same model, but at different levels of detail.

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.

Advanced Structural Modeling

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

Dependency

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

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

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

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

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

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

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

instantiate
when you want to specify which element creates objects of another.
powertype
when you want to model classes that cover other classes, such as you'll find when modeling databases
refine
when you want to model classes that are essentially the same but at different levels of abstraction.
use
when you want to explicitly mark a dependency as a using relationship

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

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

* Two stereotypes apply to dependency relationships among use cases:

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

* There are three stereotypes when modeling interactions among objects.

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

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

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

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

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

* We'll use send when you want to model an operation dispatching a given event to a target object.
* The send dependency in effect lets you tie independent state machines together.

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

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

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

Generalization

A generalization is a relationship between a general thing (called the superclass or parent) and a more specific kind of that thing(called the subclass or child).
In a generalization relationship, instances of the child may be used anywhere instances of the parent apply—meaning that the child is substitutable for the parent.
A plain, unadorned generalization relationship is sufficient for most of the inheritance relationships you'll encounter. However, if you want to specify a shade of meaning,
The UML defines one stereotype and four constraints that may be applied to generalization relationships.

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

?implementation

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

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

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

complete

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

incomplete

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

Disjoint & overlapping

- These two constraints apply only in the context of multiple inheritance.
- We'll use disjoint and overlapping when you want to distinguish between static classification (disjoint) and dynamic classification (overlapping).

Association

An association is a structural relationship, specifying that objects of one thing are connected to objects of another.

We use associations when you want to show structural relationships.

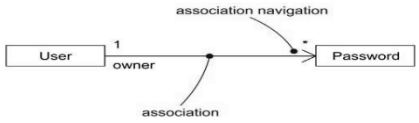
There are four basic adornments that apply to an association: a name, the role at each end of the association, the multiplicity at each end of the association, and aggregation.

For advanced uses, there are a number of other properties you can use to model subtle details, such as

Navigation
Vision
Qualification
various flavors of aggregation.

Navigation

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

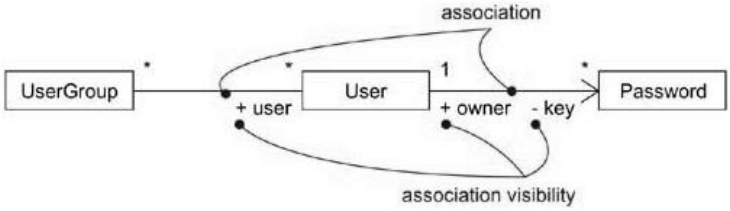


Navigation

Visibility

- Given an association between two classes, objects of one class can see and navigate to objects of the other, unless otherwise restricted by an explicit statement of navigation.
- However, there are circumstances in which you'll want to limit the visibility across that association relative to objects outside the association.

- In the UML, you can specify three levels of visibility for an association end, just as you can for a class's features by appending a visibility symbol to a role name the visibility of a role is public.
- Private visibility indicates that objects at that end are not accessible to any objects outside the association.
- Protected visibility indicates that objects at that end are not accessible to any objects outside the associ



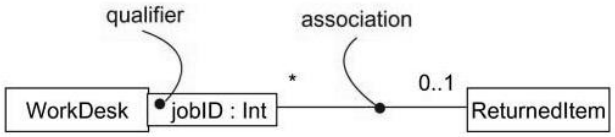
Visibility

Qualification

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

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

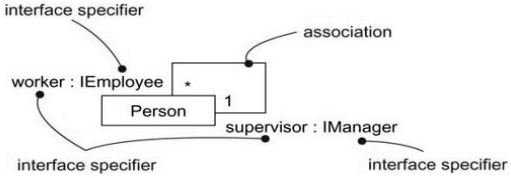
You render a qualifier as a small rectangle attached to the end of an association, placing the attributes in the rectar



Interface Specifier

An interface is a collection of operations that are used to specify a service of a class or a component Collectively, the interfaces realized by a class represent a complete specification of the behavior of that class.

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

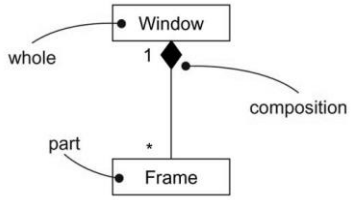


a Person class may realize many interfaces: IManager, IEmployee, IOfficer, and so on you can model the relationship between a supervisor and her workers with a one-to-many association, explicitly labeling the roles of this association as supervisor and worker

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

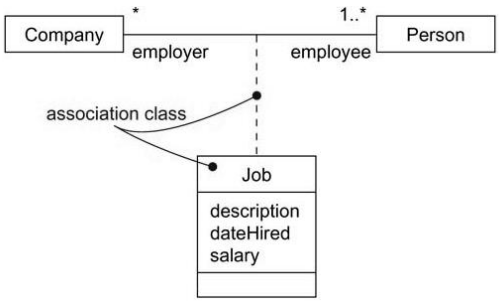
Composition

* Simple aggregation is entirely conceptual and does nothing more than distinguish a "whole" from a "part."
* Composition is a form of aggregation, with strong ownership and coincident lifetime as part of the whole.
* Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it. Such parts can also be explicitly removed before the death of the composite.
* This means that, in a composite aggregation, an object may be a part of only one composite at a time
* In addition, in a composite aggregation, the whole is responsible for the disposition of its parts, which means that the composite must manage the creation and destruction of its parts



Association Classes

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



Association Classes

Constraints

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

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

implicit

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

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

ordered

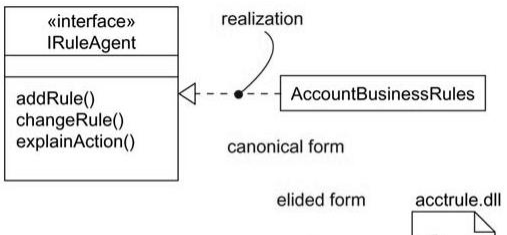
* For example, in a User/Password association, the Passwords associated with the User might be kept in a least-recently used order, and would be marked as ordered.

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

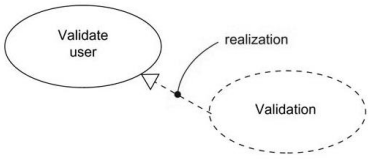
1	**xor**	Specifies that, over a set of associations, exactly one is manfest for each associated object

Realization

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



Realization of an Interface



Realization of a Use Case

Common Modeling Techniques

Modeling Webs of Relationships

1. When you model the vocabulary of a complex system, you may encounter dozens, if not hundreds or thousands, of classes, interfaces, components, nodes, and use cases.
2. Establishing a crisp boundary around each of these abstractions is hard
3. This requires you to form a balanced distribution of responsibilities in the system as a whole, with individual abstractions that are tightly cohesive and with relationships that are expressive, yet loosely coupled
4. When you model these webs of relationships,
    - Don't begin in isolation. Apply use cases and scenarios to drive your discovery of the relationships among a set of abstractions.
    - In general, start by modeling the structural relationships that are present. These reflect the static view of the system and are therefore fairly tangible.
    - Next, identify opportunities for generalization/specialization relationships; use multiple inheritance sparingly.
    - Only after completing the preceding steps should you look for dependencies; they generally represent more-subtle forms of semantic connection.
    - For each kind of relationship, start with its basic form and apply advanced features only as absolutely necessary to express your intent.

- Remember that it is both undesirable and unnecessary to model all relationships among a set of abstractions in a single diagram or view. Rather, build up your system's relationships by considering different views on the system. Highlight interesting sets of relationships in individual diagrams.

Interfaces, type and roles

Interface

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

type

- A type is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object.

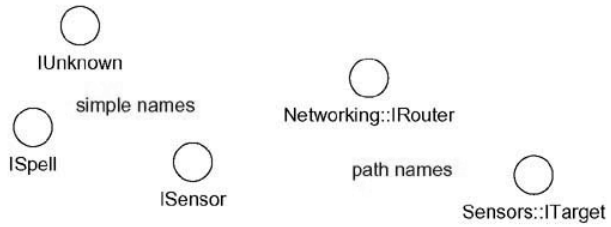
role

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

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

Names

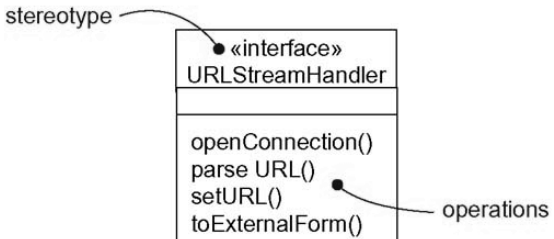
- Every interface must have a name that distinguishes it from other interfaces.
- A name is a textual string. That name alone is known as a simple name;
- A path name is the interface name prefixed by the name of the package



Simple and Path Names

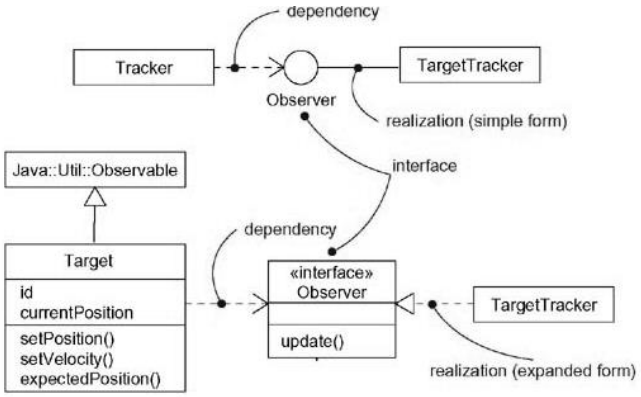
Operations

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



Relationships

- Like a class, an interface may participate in generalization, association, and dependency relationships. In addition, an interface may participate in realization relationships.
- An interface specifies a contract for a class or a component without dictating its implementation. A class or component may realize many interfaces
- We can show that an element realizes an interface in two ways.
  - First, you can use the simple form in which the interface and its realization relationship are rendered as a lollipop sticking off to one side of a class or component.
  - Second, you can use the expanded form in which you render an interface as a stereotyped class which allows you to visualize its operations and other properties, and then draw a realization relationship from the classifier or component to the interface.



Understandin

- In the UM interface in order to make it understand

- First, you may attach pre- and post conditions to each operation and invariants to the class or component as a whole. By doing this, a client who needs to use an interface will be able to understand what the interface does and how to use it, without having to dive into an implementation.

- We can attach a state machine to the interface. You can use this state machine to specify the legal partial ordering of an interface's operations.

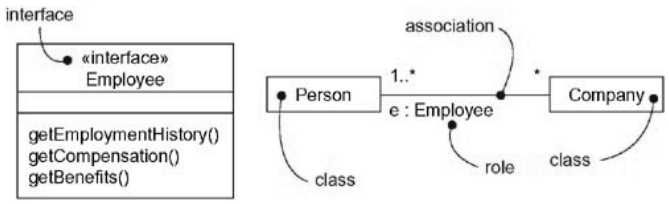
- We can attach collaborations to the interface. You can use collaborations to specify the expected behavior of the interface through a series of interaction diagrams.

Types and Roles

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

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

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



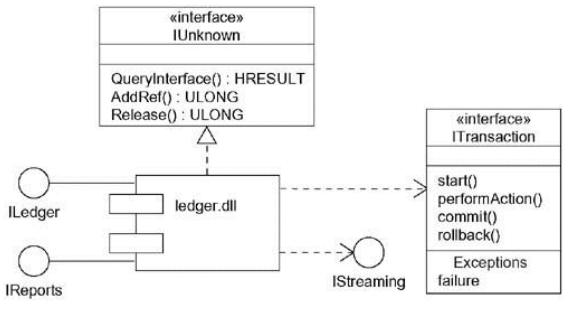
Roles

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

Common Modeling Techniques

➢ Modeling the Seams in a System.

- The most common purpose for which you'll use interfaces is to model the seams in a system composed of software components, such as COM+ or Java Beans.
- Identifying the seams in a system involves identifying clear lines of demarcation in your architecture. On either side of those lines, you'll find components that may change independently, without affecting the components on the other side,



Modeling the Seams in a System

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

➢ Modeling Static and Dynamic Typeseling Static and Dynamic Types

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

  - Specify the different possible types of that object by rendering each type as a class stereotyped as
    type (if the abstraction requires structure and behavior) or as interface (if the abstraction requires
    only behavior).

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

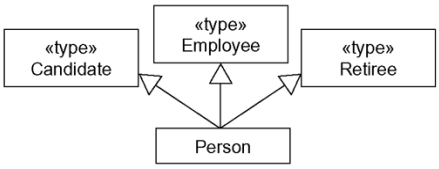
    1.) First, in a class diagram, explicitly type each role that the class plays in its association with
    Other classes. Doing this specifies the face instances of that class put on in the context of the

associated object.

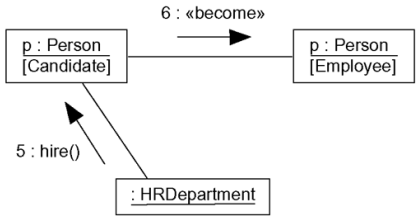
2.) Second, also in a class diagram, specify the class-to-type relationships using generalization.

- In an interaction diagram, properly render each instance of the dynamically typed class. Display the role of the instance in brackets below the object's name.

- To show the change in role of an object, render the object once for each role it plays in the interaction, and connect these objects with a message stereotyped as become.



Modeling Static Types



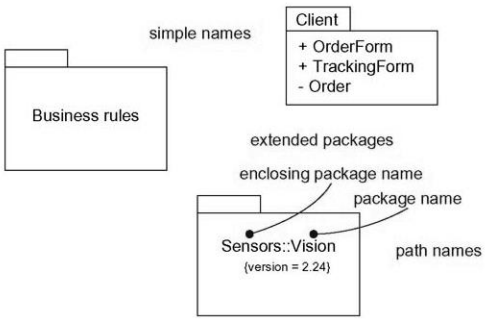
Modeling Dynamic Types

Package

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

Names

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



Simple and Extended Package

Owned Elements

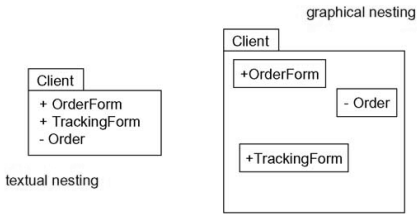
A package may own other elements, including classes, interfaces, components, nodes, collaborations, use cases, diagrams, and even other packages.

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

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

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

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



Owned Elements

Visibility

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

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

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

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

Importing and Exporting

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

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

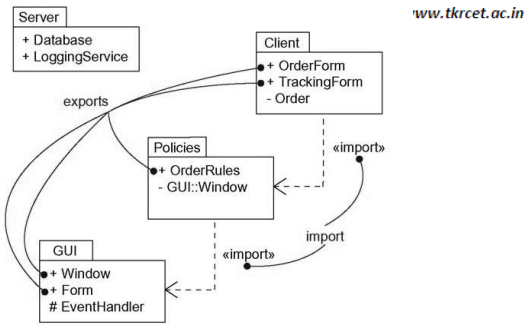
Import adds the contents of the target to the source's namespace

Access does not add the contents of the target

The public parts of a package are called its exports.

The parts that one package exports are visible only to the contents of those packages that explicitly import the package.

Import and access dependencies are not transitive



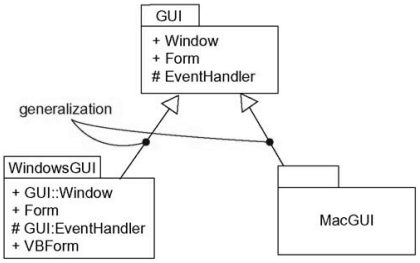
Importing and Exporting

Generalization

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

Generalization among packages is very much like generalization among classes

Packages involved in generalization relationships follow the same principle of substitutability as do classes. A specialized package (such as WindowsGUI) can be used anywhere a more general package (such as GUI) is



Standard Elements

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

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

The UML does not specify icons for any of these stereotypes

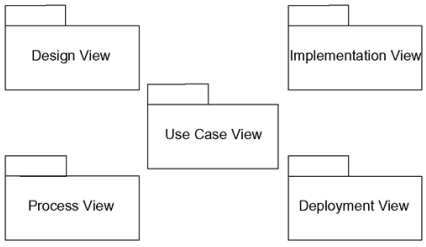
Common Modeling Techniques

Modeling Groups of Elements

The most common purpose for which you'll use packages is to organize modeling elements into groups that you can name and manipulate as a set.
There is one important distinction between classes and packages:
Packages have no identity (meaning that you can't have instances of packages, so they are invisible in the running system);
classes do have identity (classes have instances, which are elements of a running system).
To model groups of elements,
Scan the modeling elements in a particular architectural view and look for clumps defined by elements that are conceptually or semantically close to one another.
Surround each of these clumps in a package.
For each package, distinguish which elements should be accessible outside the package. Mark them public, and all others protected or private. When in doubt, hide the element.
Explicitly connect packages that build on others via import dependencies
In the case of families of packages, connect specialized packages to their more general part via generalizations

Modeling Architectural Views

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



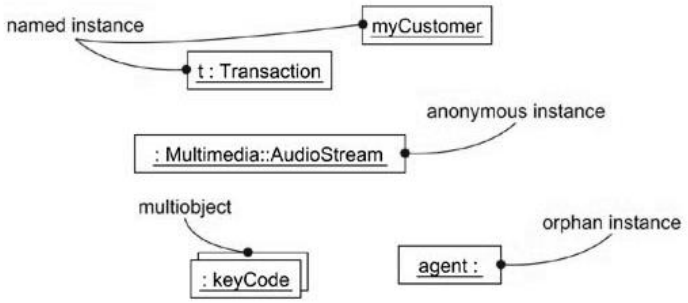
Modeling Architectural Views

Instances

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

Abstractions and Instances

- Most instances you'll model with the UML will be instances of classes although you can have instances of other things, such as components, nodes, use cases, and associations
- In the UML, an instance is easily distinguishable from an abstraction. To indicate an instance, you underline its name.
- We can use the UML to model these physical instances, but you can also model things that are not so concrete.



Named, Anonymous, Multiple, and Orphan Instances

Names

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

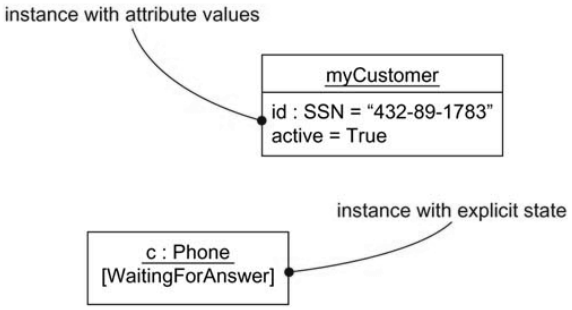
Operations

- The operations you can perform on an object are declared in the object's abstraction

- For example, if the class Transaction defines the operation commit, then given the instance t : Transaction, you can write expressions such as t.commit()

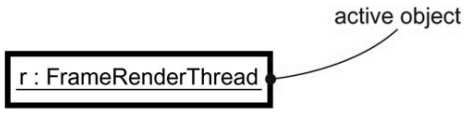
State

An object also has state. An object's state is therefore dynamic. So when you visualize its state, you are really specifying the value of its state at a given moment in time and space.
It's possible to show the changing state of an object by showing it multiple times in the same interaction diagram, but with each occurrence representing a different state.
When you operate on an object, you typically change its state;
when you query an object, you don't change its state



Other Features

Processes and threads are an important element of a system's process view, so the UML provides a visual cue to distinguish elements that are active from those that are passive.
You can declare active classes that reify a process or thread, and in turn you can distinguish an instance of an active class

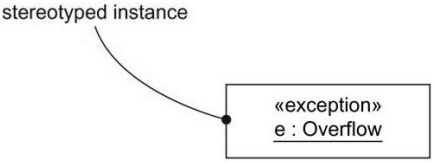


Active Objects

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

Standard Elements

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



Stereotyped Objects

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

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

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

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

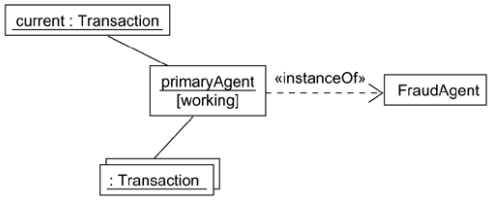
The UML defines a standard constraint that applies to objects:

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

Common Modeling Techniques

Modeling Concrete Instances

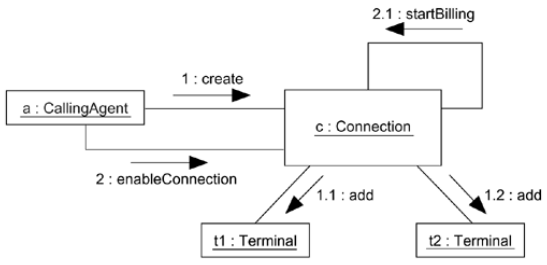
- When you model concrete instances, you are in effect visualizing things that live in the real world
- One of the things for which you'll use objects is to model concrete instances that exist in the real world
- To model concrete instances,
    - Identify those instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.
    - Render these objects in the UML as instances. Where possible, give each object a name. If there is no meaningful name for the object, render it as an anonymous object.
    - Expose the stereotype, tagged values, and attributes (with their values) of each instance necessary and sufficient to model your problem.
    - Render these instances and their relationships in an object diagram or other diagram appropriate to the kind of the instance.



Modeling Concrete Instances

Modeling Prototypical Instances

- Perhaps the most important thing for which you'll use instances is to model the dynamic interactions among objects. When you model such interactions, you are generally not modeling concrete instances that exist in the real world.
- These are prototypical objects and, therefore, are roles to which concrete instances conform.
- Concrete objects appear in static places, such as object diagrams, component diagrams, and deployment diagrams.
- Prototypical objects appear in such places as interaction diagrams and activity diagrams.
- To model prototypical instances,
    - Identify those prototypical instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.
    - Render these objects in the UML as instances. Where possible, give each object a name. If there is no meaningful name for the object, render it as an anonymous object.
    - Expose the properties of each instance necessary and sufficient to model your problem.
    - Render these instances and their relationships in an interaction diagram or an activity diagram.



Class Diagrams

- A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- Graphically, a class diagram is a collection of vertices and arcs.

Common Properties

Contents

- Class diagrams commonly contain the following things:

- Classes
- Interfaces
- Collaborations
- Dependency, generalization, and association relationships

- Like all other diagrams, class diagrams may contain notes and constraints
- Class diagrams may also contain packages or subsystems

Note: Component diagrams and deployment diagrams are similar to class diagrams, except that instead of containing classes, they contain components and nodes

Common Uses

- You use class diagrams to model the static design view of a system. This view primarily supports the functional requirements of a system
- We'll typically use class diagrams in one of three ways:

1. To model the vocabulary of a system
2. To model simple collaborations
3. To model a logical database schema

Modeling the vocabulary of a system

- Modeling the vocabulary of a system involves making a decision about which abstractions are a part of the system under consideration and which fall outside its boundaries

Modeling simple collaborations

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

Modeling logical database schema

- We can model schemas for these databases using class diagrams.

Common Modeling Techniques

Modeling Simple Collaborations

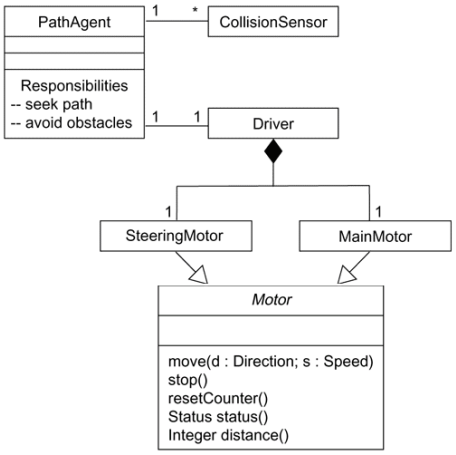
- When you create a class diagram, you just model a part of the things and relationships that make up your system's design view. For this reason, each class diagram should focus on one collaboration at a time.
- To model a collaboration

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

- For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration. Identify the relationships among these things, as well.

- Use scenarios to walk through these things. Along the way, you'll discover parts of your model that were missing and parts that were just plain semantically wrong.

- Be sure to populate these elements with their contents. For classes, start with getting a good balance of responsibilities. Then, over time, turn these into concrete attributes and operations.



Modeling a Logical Database Schema

- The UML is well-suited to modeling logical database schemas, as well as physical databases themselves.
- The UML's class diagrams are a superset of entity-relationship (E-R) diagrams, Whereas classical E-R diagrams focus only on data, class diagrams go a step further by permitting the modeling of behavior, as well. In the physical database these logical operations are generally turned into triggers or stored procedures.
- To model a schema,

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

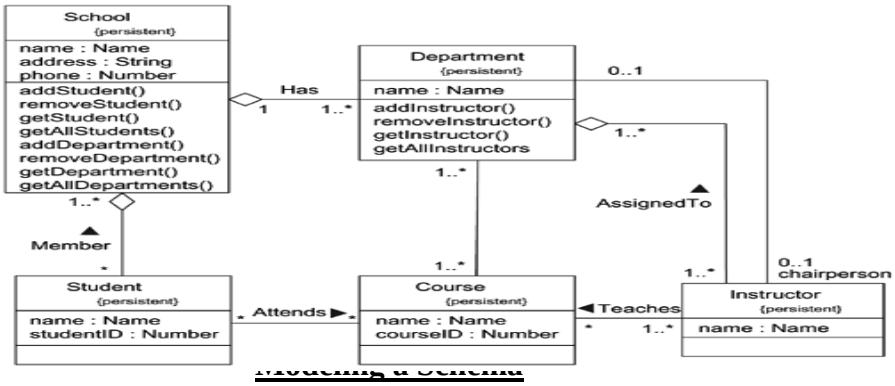
  - Create a class diagram that contains these classes and mark them as persistent (a standard tagged value). You can define your own set of tagged values to address database-specific details.

  - Expand the structural details of these classes. In general, this means specifying the details of their attributes and focusing on the associations and their cardinalities that structure these classes.

- Watch for common patterns that complicate physical database design, such as cyclic associations, one-to-one associations, and n-ary associations. Where necessary, create intermediate abstractions to simplify your logical structure.

- Consider also the behavior of these classes by expanding operations that are important for data access and data integrity. In general, to provide a better separation of concerns, business rules concerned with the manipulation of sets of these objects should be encapsulated in a layer above these persistent classes.

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



Forward and Reverse Engineering

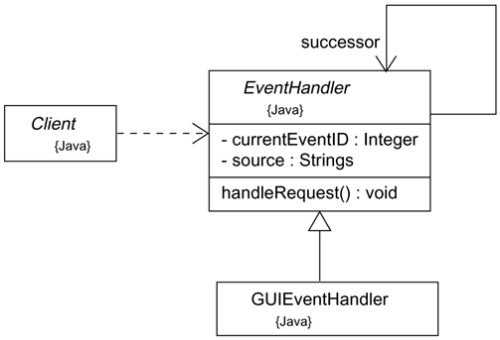
- **Forward engineering** is the process of transforming a model into code through a mapping to an implementation language
- Forward engineering results in a loss of information, because models written in the UML are semantically richer than any current object-oriented programming languageTo forward engineer a class diagram,

  - Identify the rules for mapping to your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.

  - Depending on the semantics of the languages you choose, you may have to constrain your use of certain UML features. For example, the UML permits you to model multiple inheritance, but Smalltalk permits only single inheritance. You can either choose to prohibit developers from modeling with multiple inheritance (which makes your models language-dependent) or develop idioms that transform these richer features into the implementation language (which makes the mapping more complex).

  - Use tagged values to specify your target language. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages.

  - Use tools to forward engineer your models.



Forward Engineering

- **Reverse engineering** is the process of transforming code into a model through a mapping from a specific implementation language.
- Reverse engineering results in a flood of information, some of which is at a lower level of detail than you'll need to build useful models.
- Reverse engineering is incomplete. There is a loss of information when forward engineering models into code, and so you can't completely recreate a model from code unless your tools encode information in the source comments that goes beyond the semantics of the implementation language.
- To reverse engineer a class diagram,

    - Identify the rules for mapping from your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.

    - Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered.

    - Using your tool, create a class diagram by querying the model. For example, you might start with one or more classes, then expand the diagram by following specific relationships or other neighboring classes. Expose or hide details of the contents of this class diagram as necessary to communicate your intent.

Object Diagram

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

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

Contents

- Object diagrams commonly contain
    - Objects
    - Links
- Like all other diagrams, object diagrams may contain notes and constraints.
- Object diagrams may also contain packages or subsystems

Common Uses

- You use object diagrams to model the static design view or static process view of a system just as you do with class diagrams
- When you model the static design view or static process view of a system, you typically use object diagrams in one way:
    - To model object structures

Modeling Object Structures

- Modeling object structures involves taking a snapshot of the objects in a system at a given moment in time.
- An object diagram represents one static frame in the dynamic storyboard represented by an interaction diagram

Common Modeling Techniques

Modeling Object Structures

- An object diagram shows one set of objects in relation to one another at one moment in time.
- To model an object structure,

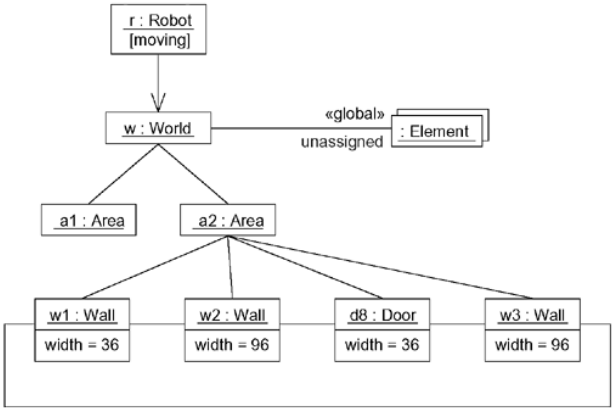
    - Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.

    - For each mechanism, identify the classes, interfaces, and other elements that participate in this collaboration; identify the relationships among these things, as well.

    - Consider one scenario that walks through this mechanism. Freeze that scenario at a moment in time, and render each object that participates in the mechanism.

- Expose the state and attribute values of each such object, as necessary, to understand the scenario.

- Similarly, expose the links among these objects, representing instances of associations among them.



Modeling Object Structures

Forward and Reverse Engineering

- Forward engineering an object diagram is theoretically possible but pragmatically of limited value
- In an object-oriented system, instances are things that are created and destroyed by the application during run time. Therefore, you can't exactly instantiate these objects from the outside.
- Component instances and node instances are things that live outside the running system and are amenable to some degree of forward engineering.

- Reverse engineering an object diagram is a very different thing
- To reverse engineer an object diagram,

  - Chose the target you want to reverse engineer. Typically, you'll set your context inside an operation or relative to an instance of one particular class.

  - Using a tool or simply walking through a scenario, stop execution at a certain moment in time.

  - Identify the set of interesting objects that collaborate in that context and render them in an object diagram.

- As necessary to understand their semantics, expose these object's states.
- As necessary to understand their semantics, identify the links that exist among these objects.
- If your diagram ends up overly complicated, prune it by eliminating objects that are not germane to the questions about the scenario you need answered. If your diagram is too simplistic, expand the neighbors of certain interesting objects and expose each object's state more deeply.

UNIT – III
BASIC BEHAVIORAL MODELING

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.

Context

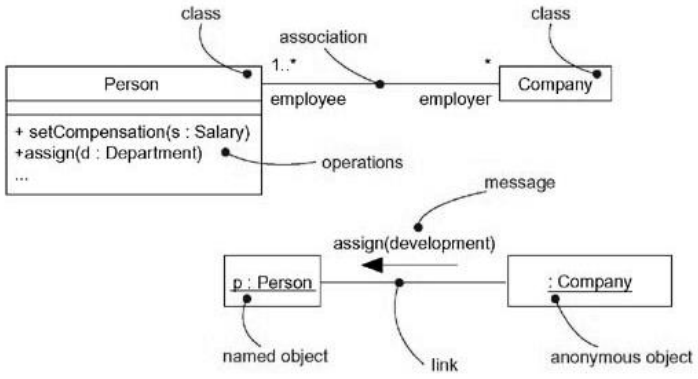
- We can use interactions to visualize, specify, construct, and document the semantics of a class
- We may find an interaction wherever objects are linked to one another.
- We'll find interactions in the collaboration of objects that exist in the context of your system or subsystem.
- We will also find interactions in the context of an operation.
- We might create interactions that show how the attributes of that class collaborate with one another
- Finally, you'll find interactions in the context of a class.

Objects and Roles

- The objects that participate in an interaction are either concrete things or prototypical things.
- As 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
- As a prototypical thing, p might represent any instance of Person.
- Although abstract classes and interfaces, by definition, may not have any direct instances, you may find instances of these things in an interaction
- Such instances do not represent direct instances of the abstract class or of the interface, but may represent, respectively, indirect (or prototypical) instances of any concrete children of the abstract class of some concrete class that realizes that interface.

Links

- A link is a semantic connection among objects. In general, a link is an instance of an association
- 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.
- A link specifies a path along which one object can dispatch a message to another (or the same) object.



Links and Associations

- We can adorn the appropriate end of the link with any of the following standard stereotypes

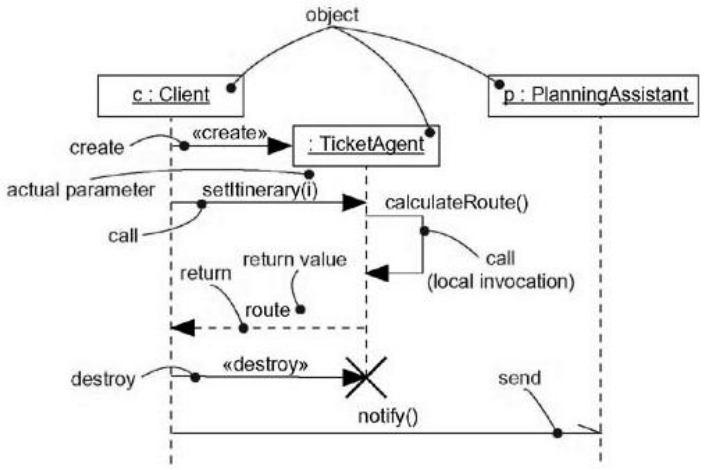
association	Specifies that the corresponding object is visible by association
self	Specifies that the corresponding object is visible because it is the dispatcher of the operation
global	Specifies that the corresponding object is visible because it is in an enclosing scope
local	Specifies that the corresponding object is visible because it is in a local scope
parameter	Specifies that the corresponding object is visible because it is a parameter

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.
- In the UML, you can model several kinds of actions

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

- The UML provides a visual distinction among these kinds of messages, as follows



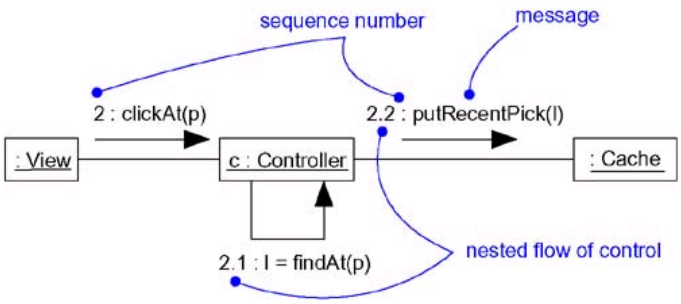
Messages

- When an object calls an operation or sends a signal to another object, you can provide actual parameters to the message.
- Similarly, when an object returns control to another object, you can model the return value.

Sequencing

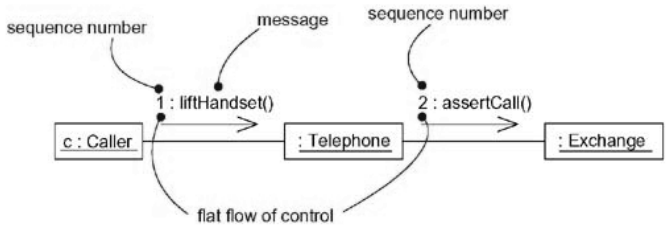
- When an object passes a message to another object the receiving object might in turn send a message to another object, which might send a message to yet a different object, and so on. This stream of messages forms a sequence

- Any sequence must have a beginning; the start of every sequence is rooted in some process or thread.
- Any sequence will continue as long as the process or thread that owns it lives.
- Messages are ordered in sequence by time. To better visualize the sequence of a message, you can explicitly model the order of the message relative to the start of the sequence by prefixing the message with a sequence number set apart by a colon separator
- Most commonly, you can specify a **procedural or nested flow of control**, rendered using a filled solid arrowhead



Procedural Sequence

- We can specify a flat flow of control, rendered using a stick arrowhead, to model the nonprocedural progression of control from step to step.



Flat Sequence

- Typically, you'll use flat sequences only when modeling interactions in the context of use cases that involve the system as a whole, together with actors outside the system.
- Such sequences are often flat because control simply progresses from step to step, without any consideration for nested flows of control.

- We'll want to use procedural sequences, because they represent ordinary, nested operation calls of the type you find in most programming languages.

- **Creation, Modification, and Destruction**

- Most of the time, the objects you show participating in an interaction exist for the entire duration of the interaction. However, in some interactions, objects may be created (specified by a create message) and destroyed (specified by a destroy message).
- The same is true of links: the relationships among objects may come and go. 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

- Specifies that the instance or link is created during execution of the enclosing interaction but is destroyed before completion of execution
- Specifies that the instance or link is created during execution of the enclosing interaction but is destroyed before completion of execution

Representation

- When you model an interaction, you typically include both objects and messages
- We can visualize those objects and messages involved in an interaction in two ways
    - By emphasizing the time ordering of its messages
    - 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
- Sequence diagrams and collaboration diagrams are largely isomorphic
- Sequence diagrams permit you to model the lifeline of an object.
- Collaboration diagrams permit you to model the structural links that may exist among the objects in an interaction.

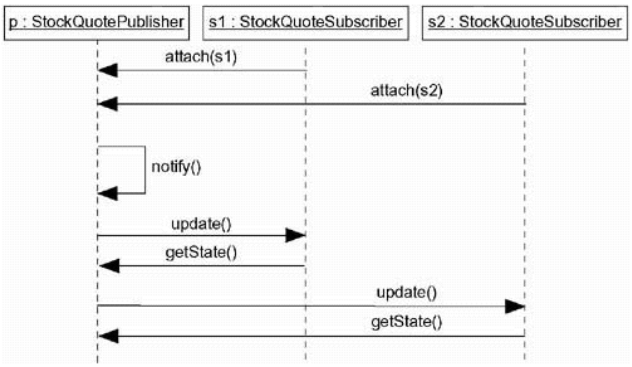
Common Modeling Techniques

Modeling a Flow of Control

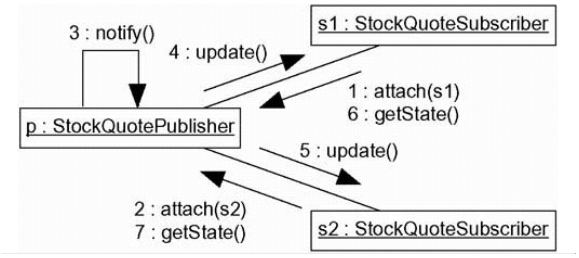
- The most common purpose for which you'll use interactions is to model the flow of control that characterizes the behavior of a system as a whole, including use cases, patterns, mechanisms, and frameworks, or the behavior of a class or an individual operation.
- classes, interfaces, components, nodes, and their relationships model the static aspects of your system
- Interactions model its dynamic aspects of your system.
- 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.
- 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.



- This figure is semantically equivalent to the previous one, but it is drawn as a collaboration diagram, which emphasizes the structural organization of the objects.

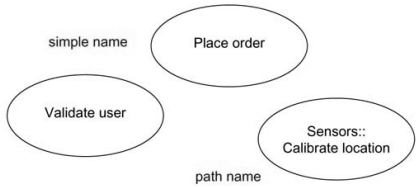


Use Cases

- 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

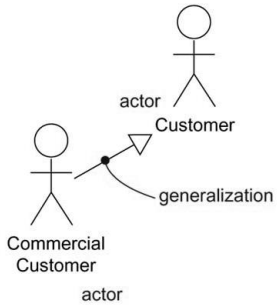
- Every use case must have a name that distinguishes it from other use cases. A name is a textual string.
- That name alone is known as **a simple name**; **a path name** is the use case name prefixed by the name of the package in which that use case lives.
- A use case is typically drawn showing only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



Simple and Path Names

Use Cases and Actors

- An actor represents a coherent set of roles that users of use cases play when interacting with these
  use cases.
- Typically, an actor represents a role that a human, a hardware device, or another system plays with a system.
- An instance of an actor, therefore, represents an individual interacting with the system in a specific way
- Actors may be connected to use cases only by association
- An association between an actor and a use case indicates that the actor and the use case communicate with one another, each one possibly sending and receiving messages.



Actors

Use Cases and Flow of Events

- A use case describes what a system does but it does not specify how it does it.
- You can specify the behavior of a use case by describing a flow of events in text clearly enough for an outsider to understand it easily
- When you write this flow of events, you should include how and when the use case starts and ends
- When the use case interacts with the actors and what objects are exchanged, and the basic flow and alternative flows of the behavior.

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:**

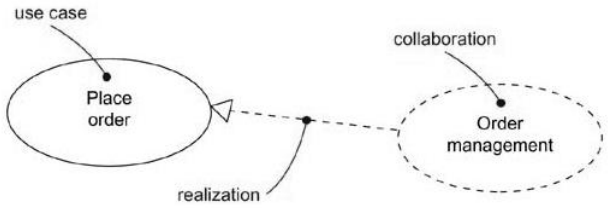
If the Customer enters an invalid PIN number, the use case restarts. If this happens three times in a row, the system cancels the entire transaction, preventing the Customer from interacting with the ATM for 60 seconds

Use Cases and Scenarios

- Typically, we'll first describe the flow of events for a use case in text.
- Typically, we'll use one sequence diagram to specify a use case's main flow, and variations of that diagram to specify a use case's exceptional flows.
- Use case describes a set of sequences, not just a single sequence, and it would be impossible to express all the details of an interesting use case in just one sequence.
- Each sequence is called a **scenario**. 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 you are developing, without having to specify how that behavior is implemented.
- however, you have to implement your use cases, and you do so by creating a society of classes and other elements that work together to implement the behavior of this use case
- This society of elements, including both its static and dynamic structure, is modeled in the UML as a **collaboration.**
- you can explicitly specify the realization of a use case by a collaboration



Use Cases and Collaborations

Organizing Use Cases

- We can organize use cases by grouping them in packages in the same manner in which you can organize classes.
- You can also organize use cases by specifying generalization, include, and extend relationships among them.
- generalization among use cases is rendered as a solid directed line with a large open arrowhead, just like generalization among classes.

- 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.
- You use an include relationship to avoid describing the same flow of events several times, by putting the common behavior in a use case of its own
- The include relationship is essentially an example of delegation—you take a set of responsibilities of the system and capture it in one place (the included use case), then let all other parts of the system (other use cases) include the new aggregation of responsibilities whenever they need to use that functionality.
- include followed by the name of the use case you want to include
- You render an include relationship as a dependency, stereotyped as include.

- An **extend relationship** between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case.
- This base use case may be extended only at certain points called, not surprisingly, its extension points
- We use an extend relationship to model the part of a use case the user may see as optional system behavior.
- We may also use an extend relationship to model a separate subflow that is executed only under given conditions.

- Finally, we may use an extend relationship to model several flows that may be inserted at a certain point, governed by explicit interaction with an actor.
- We render an extend relationship as a dependency, stereotyped as extend.

Common Modeling Techniques

Modeling the Behavior of an Element

- The most common thing for which you'll apply use cases is to model the behavior of an element, whether it is the system as a whole, a subsystem, or a class.
- To model the behavior of an element

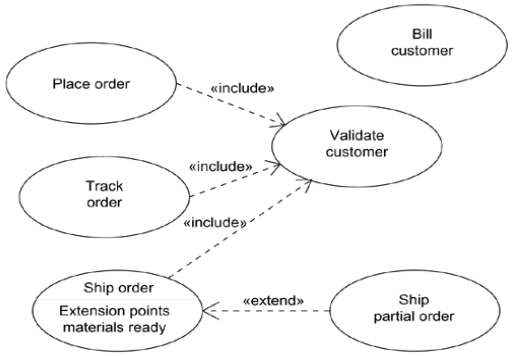
  - Identify the actors that interact with the element. Candidate actors include groups that require certain behavior to perform their tasks or that are needed directly or indirectly to perform the element's functions.

  - 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.



Modeling the Behavior of an Element

Use Case Diagram

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

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.
- Use case diagrams may also contain packages
- Occasionally, you'll want to place instances of use cases in your diagrams, as well, especially when you want to visualize a specific executing system.

Common Uses

- We apply use case diagrams to model the static use case view of a system. This view primarily supports the behavior of a system
- When you model the static use case view of a system, you'll typically apply use case diagrams in one of two ways.
    - To model the context of a system
    - To model the requirements of a system

Modeling the context of a system involves drawing a line around the whole system and asserting which actors lie outside the system and interact with it.Here, you'll apply use case diagrams to specify the actors and the meaning of their roles.

Modeling the requirements of a system involves specifying what that system should do (from a point of view of outside the system), independent of how that system should do it. Here, you'll apply use case diagrams to specify the desired behavior of the system.

Common Modeling Techniques

Modeling the Context of a System

- In the UML, you can model the context of a system with a use case diagram, emphasizing the actors that surround the system.
- To model the context of a system

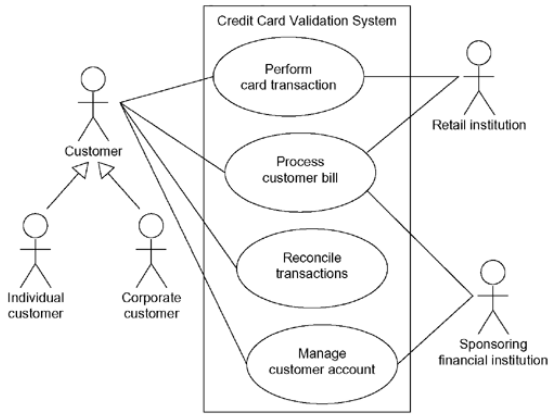
    - Identify the actors that surround the system by considering which groups require help from the system to perform their tasks; which groups are needed to execute the system's functions; which groups interact with external hardware or other software systems; and which groups perform secondary functions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

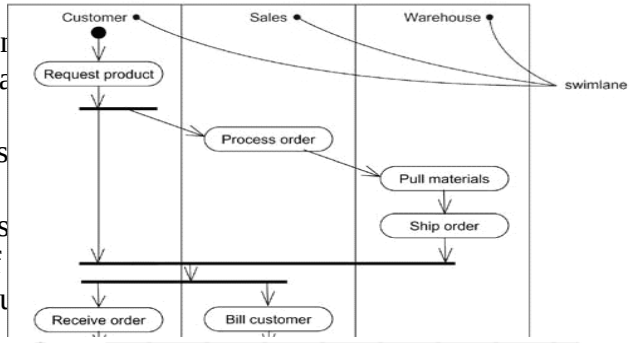
- This same technique applies to modeling the context of a subsystem. A system at one level of abstraction is often a subsystem of a larger system at a higher level of abstraction. Modeling the context of a subsystem is therefore useful when you are building systems of interconnected systems.

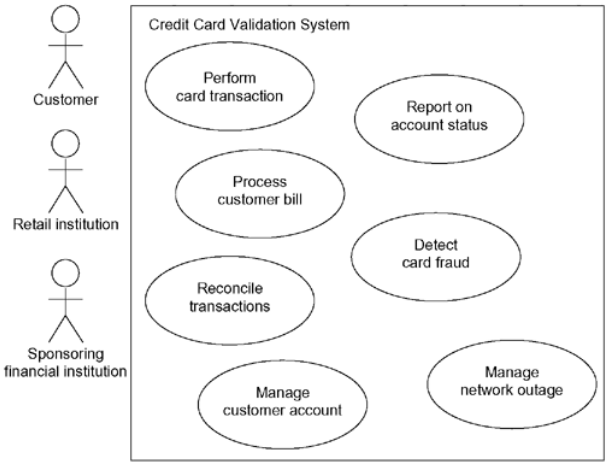


Modeling the Requiren

To model the requireme

- 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 co
- Factor commo ers; factor variant behavior into new use ca
- Model these us se diagram.
- Adorn these us irements; you may have to attach some of
- This same techniqu ystem





Forward and Reverse Engineering

- **Forward engineering** is the process of transforming a model into code through a mapping to an implementation language.
- A use case diagram can be forward engineered to form tests for the element to which it applies.
- Each use case in a use case diagram specifies a flow of events and these flows specify how the element is expected to behave
- 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.

- **Reverse engineering** is the process of transforming code into a model through a mapping from a specific implementation language.
- The UML's use case diagrams simply give you a standard and expressive language in which to state what you discover.
- 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

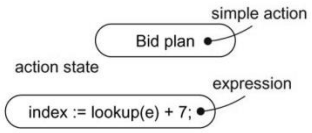
- An activity diagram shows the flow from activity to activity. An is an ongoing nonatomic execution within a state machine.
- Activities ultimately result in some action, which is made up of executable atomic computations that result in a change in state of the system or the return of a value.
- Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation, such as evaluating an expression.
- Graphically, an activity diagram is a collection of vertices and arcs.

Contents

- Activity diagrams commonly contain
  - Activity states and action states
  - Transitions
  - Objects
- Like all other diagrams, activity diagrams may contain notes and constraints.

Action States and Activity States

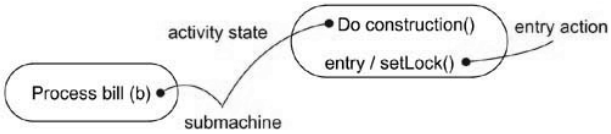
- Executable, atomic computations are called **action states** because they are states of the system, each representing the execution of an action.
- We represent an action state using a lozenge shape (a symbol with horizontal top and bottom and convex sides). Inside that shape, you may write any expression.
- Action states can't be decomposed. Furthermore, action states are atomic, 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.



Action States

- **activity states** can be further decomposed, their activity being represented by other activity diagrams
- Furthermore, activity states are not atomic, meaning that they may be interrupted and, in general, are considered to take some duration to complete.
- An action state is an activity state that cannot be further decomposed.

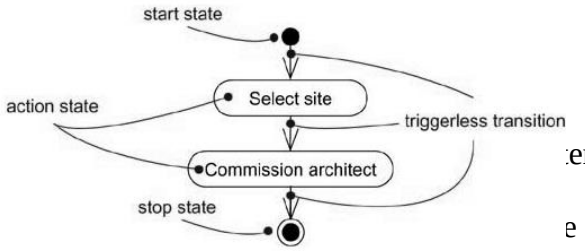
- We can think of an activity state as a composite, whose flow of control is made up of other activity states and action states.



Activity States

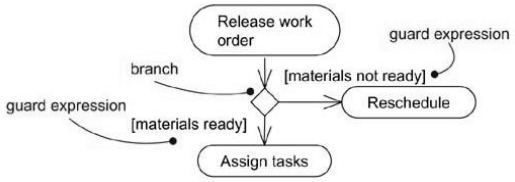
Transitions

- When the action or activity of a state completes, flow of control passes immediately to the next action or activity state.
- We specify this flow by using transitions to show the path from one action or activity state to the next action or activity state.
- In the UML, you represent a transition as a simple directed line



Branching

- As in a flowchart ernate paths taken based on some Boolean expressi
- We represent a b e incoming transition and two or more outgoing ones.
- On each outgoing transition, you place a Boolean expression, which is evaluated only once on entering the branch.
- On each outgoing transition, you place a Boolean expression, which is evaluated only once on entering the branch. Across all these outgoing transitions, guards should not overlap (otherwise, the flow of control would be ambiguous), but they should cover all possibilities (otherwise, the flow of control would freeze).
- As a convenience, you can use the keyword else to mark one outgoing transition, representing the path taken if no other guard expression evaluates to true.

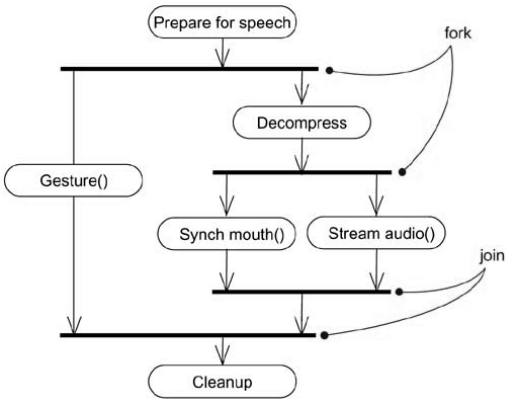


Branching

Forking and Joining

- When we are modeling workflows of business processes—we might encounter flows that are concurrent.
- In the UML, you use a synchronization bar to specify the forking and joining of these parallel flows of control. A synchronization bar is rendered as a thick horizontal or vertical line.
- **Fork** represents the splitting of a single flow of control into two or more concurrent flows of control
- A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control.

- Below the fork, the activities associated with each of these paths continues in parallel.
- Conceptually, the activities of each of these flows are truly concurrent, although, in a running system, these flows may be either truly concurrent or sequential yet interleaved, thus giving only the illusion of true concurrency.



Forking and Joining

- **A Join** represents the synchronization of two or more concurrent flows of control.
- A join may have two or more incoming transitions and one outgoing transition.
- Above the join, the activities associated with each of these paths continues in parallel.
- At the join, the concurrent flows synchronize, meaning that each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join.

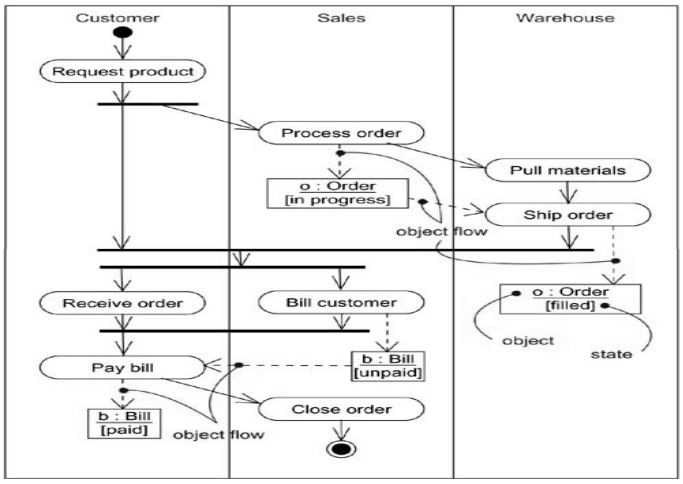
Swimlanes

- We'll find it useful, especially when you are modeling workflows of business processes, to partition the activity states on an activity diagram into groups, each group representing the business organization responsible for those activities.
- 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.
- Each swimlane represents a high-level responsibility for part of the overall activity of an activity diagram, and each 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.

Object Flow

- Objects may be involved in the flow of control associated with an activity diagram.
- We can specify the things that are involved in an activity diagram by placing these objects in the diagram, connected using a dependency to the activity or transition that creates, destroys, or modifies them.
- This use of dependency relationships and objects is called an object flow because it represents the participation of an object in a flow of control.
- We can also show how its role, state and attribute values change.
- We represent the state of an object by naming its state in brackets below the object's name.

- Similarly, We can represent the value of an object's attributes by rendering them in a compartment below the object's name.



Common Uses

- We use activity diagrams to model the dynamic aspects of a system
- These dynamic aspects may involve the activity of any kind of abstraction in any view of a system's architecture, including classes, interfaces, components, and nodes.
- When you model the dynamic aspects of a system, we'll typically use activity diagrams in two ways.
    - To model a workflow
    - To model an operation

Common Modeling Techniques

Modeling a Workflow

- No software-intensive system exists in isolation; there's always some context in which a system lives, and that context always encompasses actors that interact with the system.
- Especially for mission critical, enterprise software, you'll find automated systems working in the context of higher-level business processes.
- These business processes are kinds of workflows because they represent the flow of work and objects through the business.
- 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.

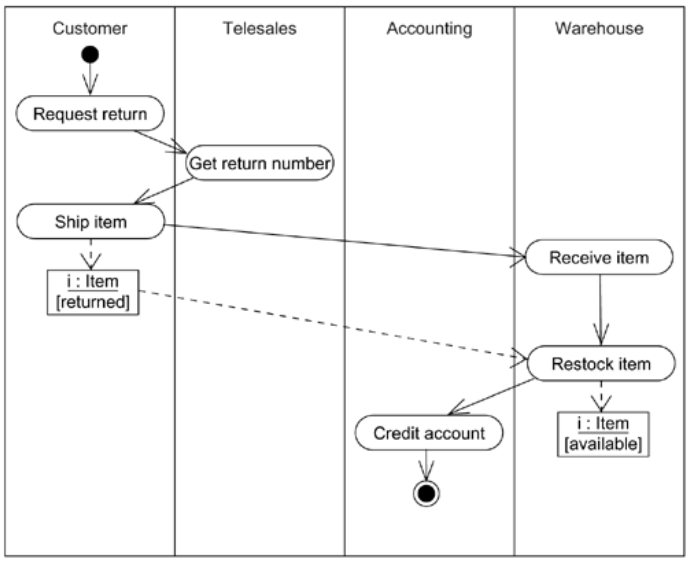
- Identify the preconditions of the workflow's initial state and the postconditions of the workflow's final state. This is important in helping you model the boundaries of the workflow.

- 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.

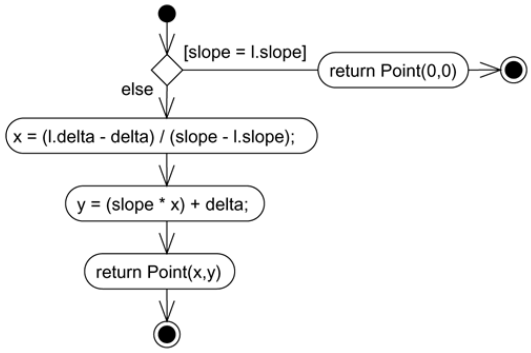
- If there are important objects that are involved in the workflow, render them in the activity diagram, as well. Show their changing values and state as necessary to communicate the intent of the object flow.



Modeling an Operation

- An activity diagram can be attached to any modeling element for the purpose of visualizing, specifying, constructing, and documenting that element's behavior.
- You can attach activity diagrams to classes, interfaces, components, nodes, use cases, and collaborations.
- The most common element to which you'll attach an activity diagram is an operation.
- An activity diagram is simply a flowchart of an operation's actions.

- An activity diagram's primary advantage is that all the elements in the diagram are semantically tied to a rich underlying model.
- 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.



Modeling an Operation

Forward and Reverse Engineering

- **Forward engineering** (the creation of code from a model) is possible for activity diagrams, especially if the context of the diagram is an operation.
- For example, using the previous activity diagram, a forward engineering tool could generate the following C++ code for the operation intersection.

```
Point Line::intersection (l : Line) {
  if (slope == l.slope) return Point(0,0);
  int x = (l.delta - delta) / (slope - l.slope);
  int y = (slope * x) + delta;
  return Point(x, y);
```

}

- **Reverse engineering** (the creation of a model from code) is also possible for activity diagrams, especially if the context of the code is the body of an operation.
- In particular, the previous diagram could have been generated from the implementation of the class Line.

UNIT-IV
ADVANCED BEHAVIORAL MODELING

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.

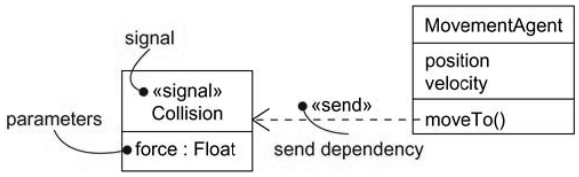
Kinds of Events

- Events may be external or internal.
- External events are those that pass between the system and its actors.
- Internal events are those that pass among the objects that live inside the system.
- An overflow exception is an example of an internal event.
- In the UML, you can model four kinds of events: signals, calls, the passing of time, and a change in state.

Signals

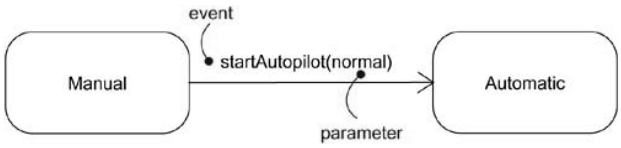
- A signal represents a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another.
- Exceptions are supported by most contemporary programming languages and are the most common kind of internal signal that you will need to model.
- Signals may also be involved in generalization relationships, permitting you to model hierarchies of events, some of which are general and some of which are specific
- Also as for classes, signals may have attributes and operations.
- A signal may be sent as the action of a state transition in a state machine or the sending of a message in an interaction. The execution of an operation can also send signals
- In fact, when you model a class or an interface, an important part of specifying the behavior of that element is specifying the signals that its operations can send.
- We model signals (and exceptions) as stereotyped classes. We can use a dependency, stereotyped as **send**, to indicate that an operation sends a particular signal.

Signals



Call Events

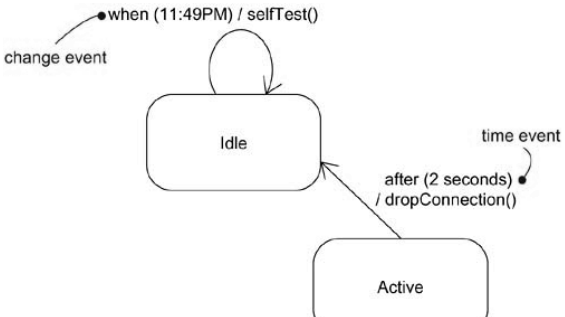
- Just as a signal event represents the occurrence of a signal, a **call** event represents the dispatch of an operation. In both cases, the event may trigger a state transition in a state machine
- Whereas a signal is an asynchronous event, a call event is synchronous
- This means that when an object 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.
- Modeling a call event is indistinguishable from modeling a signal event. In both cases, you show the event, along with its parameters, as the trigger for a state transition.



Call Events

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.
- Unless you specify it explicitly, the starting time of such an expression is the time since entering the current state.
- **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
- You can use such expressions to mark an absolute time (such as when time = 11:59) or for the continuous test of an expression

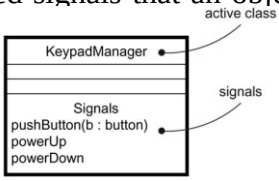


Time and Change Events

Note: Although a change event models a condition that is tested continuously, you can typically analyze the situation to see when to test the condition at discrete points in time.

Sending and Receiving Events

- Signal events and call events involve at least two objects:
    - The object that sends the signal or invokes the operation
    - The object to which the event is directed.
- Any instance of any class can send a signal to or invoke an operation of a receiving object.
- When an object sends a signal, the sender dispatches the signal and then continues along its flow of control, not waiting for any return from the receiver.
- Any instance of any class can receive a call event or a signal. If this is a synchronous call event, then the sender and the receiver are in a rendezvous(assignation) for the duration of the operation.
- This means that the flow of control of the sender is put in lock step with the flow of control of the receiver until the activity of the operation is carried out.
- If this is a signal, then the sender and receiver do not rendezvous: the sender dispatches the signal but does not wait for a response from the receiver. In either case, this event may be lost
- In the UML, you model the named signals that an object may receive by naming them in an extra compartment of the class



Signals and Active Classes

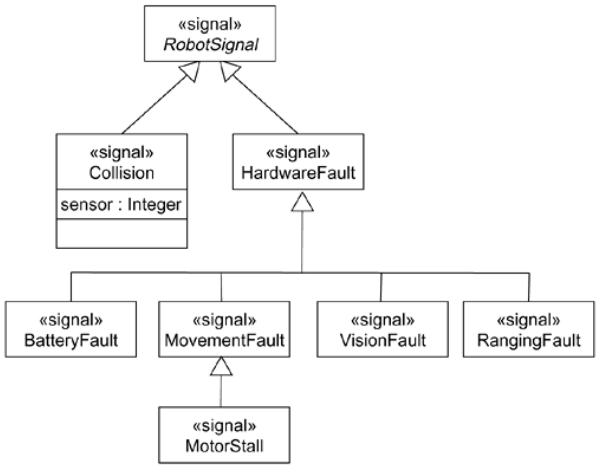
Common Modeling Techniques

Modeling a Family of Signals

- In most event-driven systems, signal events are hierarchical.
- External and internal signals need not be disjoint, however. Even within these two broad classifications, you might find specializations
- To model a family of signals
    - Consider all the different kinds of signals to which a given set of active objects may respond.

- Look for the common kinds of signals and place them in a generalization/specialization hierarchy using inheritance. Elevate more general ones and lower more specialized ones.

- Look for the opportunity for polymorphism in the state machines of these active objects. Where you find polymorphism, adjust the hierarchy as necessary by introducing intermediate abstract signals.

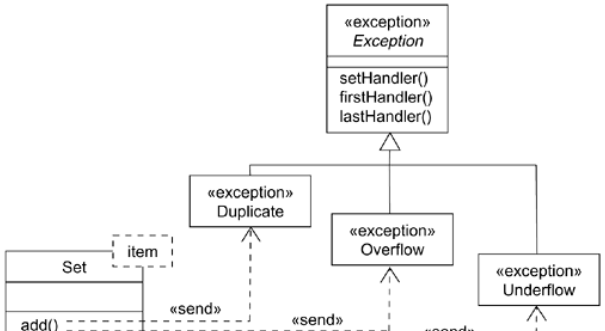


Modeling Exceptions

- An important part of visualizing, specifying, and documenting the behavior of a class or an interface is specifying the exceptions that its operations can raise.
- In the UML, exceptions are kinds of signals, which you model as stereotyped classes. Exceptions may be attached to specification operations.
- Modeling exceptions is somewhat the inverse of modeling a general family of signals.
- We model a family of signals primarily to specify the kinds of signals an active object may receive
- We model exceptions primarily to specify the kinds of exceptions that an object may throw through its operations
- To model exceptions
    - For each class and interface, and for each operation of such elements, consider the exceptional conditions that may be raised.

    - Arrange these exceptions in a hierarchy. Elevate general ones, lower specialized ones, and introduce intermediate exceptions, as necessary.

    - For each operation, specify the exceptions that it may raise. You can do so explicitly (by showing send dependencies from an operation to its exceptions) or you can put this in the operation's specification.



State Machines

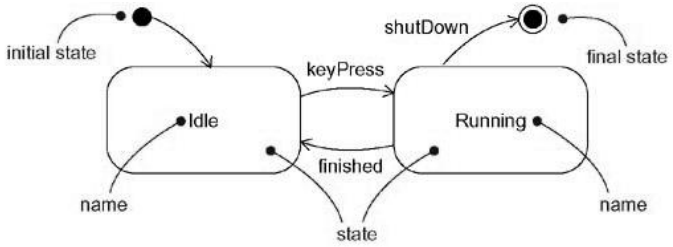
- **A state machine** is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events
- **A state** is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- **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 transition** is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- **An activity** is ongoing nonatomic execution within a state machine
- **An action** is an executable atomic computation that results in a change in state of the model or the return of a value

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.
- When an object's state machine is in a given state, the object is said to be in that state. A state has several parts.

1. Name	A textual string that distinguishes the state from other states; a state may be anonymous, meaning that it has no name
2. Entry/exit actions	Actions executed on entering and exiting the state, respectively
3. Internal transitions	Transitions that are handled without causing a change in state
4. Substates	The nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) substates
5. 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

We represent a state as a rectangle with rounded corners.



Initial and Final States

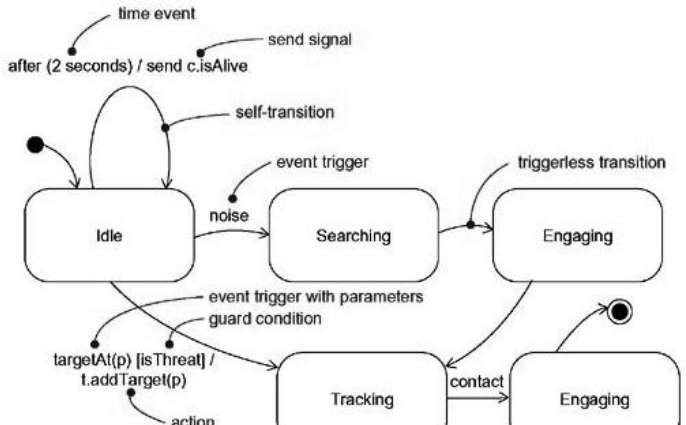
- There are two special states that may be defined for an object's state machine.
- **initial state**, which indicates the default starting place for the state machine or substate.
- An initial state is represented as a filled black circle
- **final state**, which indicates that the execution of the 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.
- On such a change of state, the transition is said to fire. Until the transition fires, the object is said to be in the source state; after it fires, it is said to be in the target state.
- A transition has five parts

1. 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
2. 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
3. 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
4. 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
5. Target state	The state that is active after the completion of the transition

- A transition is rendered as a solid directed line from the source to the target state.
- A self-transition is a transition whose source and target states are the same.



Event Trigger

- 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 or a call may have parameters whose values are available to the transition, including expressions for the guard condition and action.
- It is also possible to have a triggerless transition, represented by a transition with no event trigger. A triggerless transition—also called a completion transition—is triggered implicitly when its source state has completed its activity.

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. Therefore, it's possible to have multiple transitions from the same source state and with the same event trigger, as long as those conditions don't overlap.
- A guard condition is evaluated just once for each transition at the time the event occurs, but it may be evaluated again if the transition is retriggered. Within the Boolean expression, you can include conditions about the state of an object

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.
- An action is atomic, meaning that it cannot be interrupted by an event and therefore runs to completion.

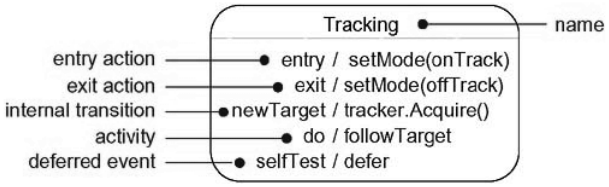
Advanced States and Transitions

- However, the UML's state machines have a number of advanced features that help you to manage complex behavioral models.
- These features often reduce the number of states and transitions you'll need, and they codify a number of common and somewhat complex idioms you'd otherwise encounter using flat state machines.
- Some of these advanced features include entry and exit actions, internal transitions, activities, and deferred events

Entry and Exit Actions

- In a number of modeling situations, you'll want to dispatch the same action whenever you enter a state, no matter which transition led you there. Similarly, when you leave a state, you'll want to dispatch the same action no matter which transition led you away.

- UML provides a shorthand for this idiom. In the symbol for the state, you can include an entry action (marked by the keyword event entry) and an exit action (marked by the keyword event exit), together with an appropriate action.
- Whenever you enter the state, its entry action is dispatched; whenever you leave the state, its exit action is dispatched.



Advanced States and Transitions

Internal Transitions

- Once inside a state, you'll encounter events you'll want to handle without leaving the state. These are called **internal transitions**, and they are subtly different from self-transitions.
- In a self-transition an event triggers the transition, you leave the state, an action (if any) is dispatched, and then you reenter the same state.
- UML provides a shorthand for this idiom, as well (for example, for the event newTarget). In the symbol for the state, you can include an internal transition (marked by an event).
- Whenever you are in the state and that event is triggered, the corresponding action is dispatched without leaving and then reentering the state.
- Therefore, the event is handled without dispatching the state's exit and then entry actions.

Activities

- When an object is in a state, it generally sits idle, waiting for an event to occur.
- Sometimes, however, you may wish to model an ongoing activity. While in a state, the object does some work that will continue until it is interrupted by an event.
- in the UML, you use the special do transition to specify the work that's to be done inside a state after the entry action is dispatched. The activity of a **do** transition might name another state machine
- You can also specify a sequence of actions—for example, do / op1(a); op2(b); op3(c).
- Actions are never interruptible, but sequences of actions are. In between each action (separated by the semicolon), events may be handled by the enclosing state, which results in transitioning out of the state.

Deferred Events

- In every modeling situation, you'll want to recognize some events and ignore others. You include those you want to recognize as the event triggers of transitions; those you want to ignore you just leave out.
- However, in some modeling situations, you'll want to recognize some events but postpone a response to them until later.
- In the UML, you can specify this behavior by using deferred events.

- **A deferred event** is a list of events whose occurrence in the state is postponed until a state in which the listed events are not deferred becomes active, at which time they occur and may trigger transitions as if they had just occurred.

Note: The implementation of deferred events requires the presence of an internal queue of events. If an event happens but is listed as deferred, it is queued. Events are taken off this queue as soon as the object enters a state that does not defer these events.

Substates

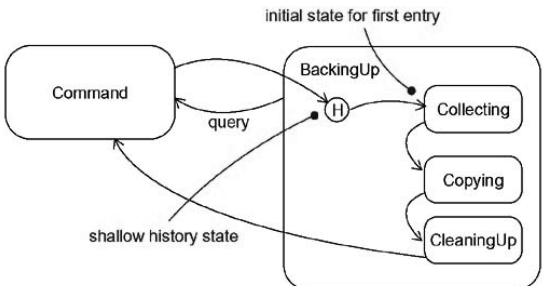
- There's one more feature of the UML's state machines—substates—that does even more to help you simplify the modeling of complex behaviors
- A simple state is a state that has no substructure.
- A substate is a state that's nested inside another one.
- A state that has substates— that is, nested states—is called a composite state.
- A composite state may contain either concurrent (orthogonal) or sequential (disjoint) substates.
- In the UML, you render a composite state just as you do a simple state, but with an optional graphic compartment that shows a nested state machine. Substates may be nested to any level

Sequential Substates

- Friends better review text book for this concept.

History States

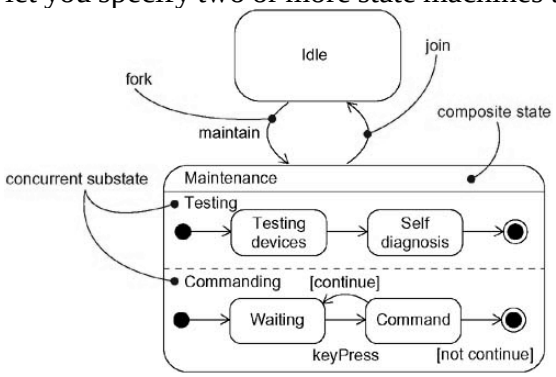
- A state machine describes the dynamic aspects of an object whose current behavior depends on its past.
- A state machine in effect specifies the legal ordering of states an object may go through during its lifetime
- When a transition enters a composite state, the action of the nested state machine starts over again at its initial state.
- However, there are times you'd like to model an object so that it remembers the last substate that was active prior to leaving the composite state. Using flat state machines, you can model this, but it's messy. For each sequential substate, you'd need to have its exit action post a value to some variable local to the composite state. Then the initial state to this composite state would need a transition to every substate with a guard condition, querying the variable. That's messy because it requires you to remember to touch every substate and to set an appropriate exit action. It leaves you with a multitude of transitions fanning out from the same initial state to different target substates with very similar (but different) guard conditions.
- In the UML, a simpler way to model this idiom is by using history states.
- **A history state** allows a composite state that contains sequential substates to remember the last substate that was active in it prior to the transition from the composite state
- As Figure shows, you represent a shallow history state as a small circle containing the symbol H.



- If you want a transition to activate the last substate, you show a transition from outside the composite state directly to the history state. The first time you enter a composite state, it has no history.
- In either case, if a nested state machine reaches a final state, it loses its stored history and behaves as if it had not yet been entered for the first time.

Concurrent Substates

- Sequential substates are the most common kind of nested state machine you'll encounter. In certain modeling situations, however, you'll want to specify concurrent substates.
- These substates let you specify two or more state machines that execute in parallel in the context of the enclosin



Concurrent Substates

- Execution of these two concurrent substates continues in parallel.
- Eventually, each nested state machine reaches its final state. If one concurrent substate reaches its final state before the other, control in that substate waits at its final state.
- When both nested state machines reach their final state, control from the two concurrent substates joins back into one flow.
- Whenever there's a transition to a composite state decomposed into concurrent substates, control forks into as many concurrent flows as there are concurrent substates. Similarly, whenever there's a transition from a composite substate decomposed into concurrent substates, control joins back into one flow.
- This holds true in all cases. If all concurrent substates reach their final state, or if there is an explicit transition out of the enclosing composite state, control joins back into one flow

Common Modeling Techniques

Modeling the Lifetime of an Object

- When you model the lifetime of an object, you essentially specify three things: the events to which the object can respond, the response to those events, and the impact of the past on current behavior.
- Modeling the lifetime of an object also involves deciding on the order in which the object can meaningfully respond to events, starting at the time of the object's creation and continuing until its destruction
- To model the lifetime of an object,
    - Set the context for the state machine, whether it is a class, a use case, or the system as a whole.

        1. If the context is a class or a use case, collect the neighboring classes, including any parents of the class and any classes reachable by associations or dependences. These neighbors are candidate targets for actions and are candidates for including in guard conditions.

        2. If the context is the system as a whole, narrow your focus to one behavior of the system. Theoretically, every object in the system may be a participant in a model of the system's lifetime, and except for the most trivial systems, a complete model would be intractable.

    - Establish the initial and final states for the object. To guide the rest of your model, possibly state the pre- and postconditions of the initial and final states, respectively.

    - Decide on the events to which this object may respond. If already specified, you'll find these in the object's interfaces; if not already specified, you'll have to consider which objects may interact with the object in your context, and then which events they may possibly dispatch.

    - Starting from the initial state to the final state, lay out the top-level states the object may be in. Connect these states with transitions triggered by the appropriate events. Continue by adding actions to these transitions.

    - Identify any entry or exit actions (especially if you find that the idiom they cover is used in the state machine).

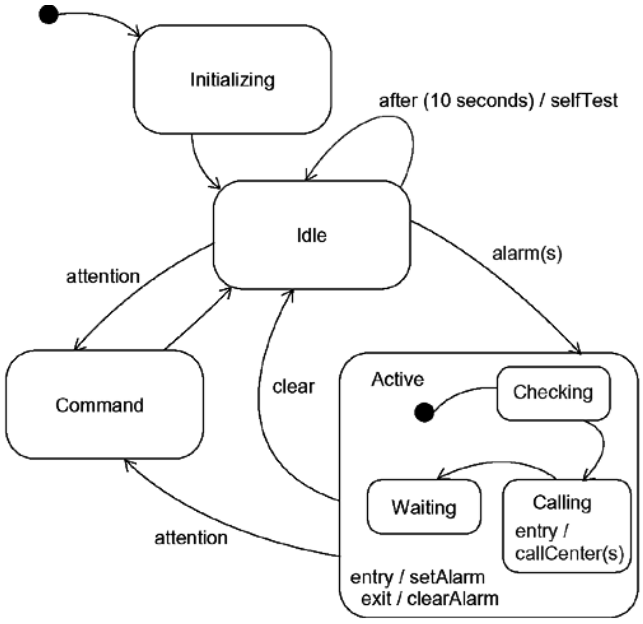
    - Expand these states as necessary by using substates.

    - Check that all events mentioned in the state machine match events expected by the interface of the object. Similarly, check that all events expected by the interface of the object are handled by the state machine. Finally, look to places where you explicitly want to ignore events.

    - Check that all actions mentioned in the state machine are sustained by the relationships, methods, and operations of the enclosing object.

- Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses. Be especially diligent in looking for unreachable states and states in which the machine may get stuck.

- After rearranging your state machine, check it against expected sequences again to ensure that you have not changed the object's semantics.



Processes and Threads

- **An active object** is an object that owns a process or thread and can initiate control activity.
- **An active class** is a class whose instances are active objects
- **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.
- Graphically, an active class is rendered as a rectangle with thick lines.
- Processes and threads are rendered as stereotyped active classes

Flow of Control

- In a purely sequential system, there is one flow of control. This means that one thing, and one thing only, can take place at a time. When a sequential program starts, control is rooted at the beginning of the program and operations are dispatched one after another.
- A sequential program will process only one event at a time, queuing or discarding any concurrent external events.

- In a concurrent system, there is more than one flow of control—that is, more than one thing can take place at a time.

- In a concurrent system, there are multiple simultaneous flows of control, each rooted at the head of an independent process or a thread.
- If you take a snapshot of a concurrent system while it's running, you'll logically see multiple loci of execution.

Note: You can achieve true concurrency in one of three ways:

- By distributing active objects across multiple nodes
- By placing active objects on nodes with multiple processors
- By a combination of both methods.

Classes and Events

- Active classes are just classes, albeit ones with a very special property. An active class represents an independent flow of control, whereas a plain class embodies no such flow.
- You use active classes to model common families of processes or threads.
- By modeling concurrent systems with active objects, you give a name to each independent flow of control.
    - 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.
- Active classes share the same properties as all other classes.
- Active classes may have instances.
- Active classes may have attributes and operations.
- Active classes may participate in dependency, generalization, and association relationships.
- Active classes may use any of the UML's extensibility mechanisms, including stereotypes, tagged values, and constraints.
- Active classes may be the realization of interfaces.
- Active classes may be realized by collaborations, and the behavior of an active class may be specified by using state machines

Standard Elements

- All of the UML's extensibility mechanisms apply to active classes. Most often, you'll use tagged values to extend active class properties.
- The UML defines two standard stereotypes that apply to active classes.

1. process	Specifies a heavyweight flow that can execute concurrently with other processes
2. thread	Specifies a lightweight flow that can execute concurrently with other threads within the same process

- A process is heavyweight, Each program runs as a process in its own address space
- Processes are never nested inside one another. If the node has multiple processors, then true concurrency on that node is possible.
- If the node has only one processor, there is only the illusion of true concurrency, carried out by the underlying operating system.

- A thread is lightweight. the threads that live in the context of a process are peers of one another, contending for the same resources accessible inside the process.
- Threads are never nested inside one another.

Communication

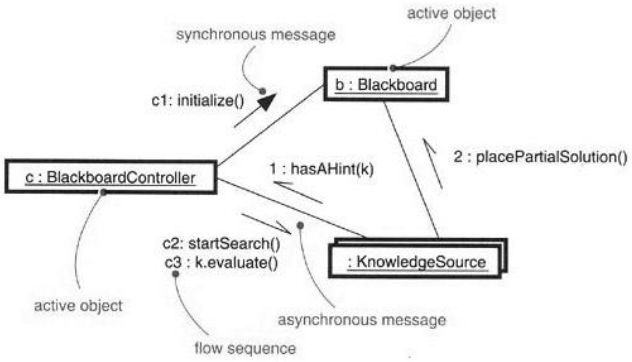
- When objects collaborate with one another, they interact by passing messages from one to the other. In a system with both active and passive objects, there are four possible combinations of interaction that you must consider
    1. A message may be passed from one passive object to another. Assuming there is only one flow of control passing through these objects at a time, such an interaction is nothing more than the simple invocation of an operation.
    2. A message may be passed from one active object to another. When that happens, you have interprocess communication, and there are two possible styles of communication

1.) One active object might synchronously call an operation of another.

2.) One active object might asynchronously send a signal or call an operation of another object

That kind of communication has mailbox semantics

- In the UML, we render a synchronous message as a full arrow and an asynchronous message as a half arrow



Communication

3. A message may be passed from an active object to a passive object. A difficulty arises if more than

one active object at a time passes their flow of control through one passive object. In that situation,

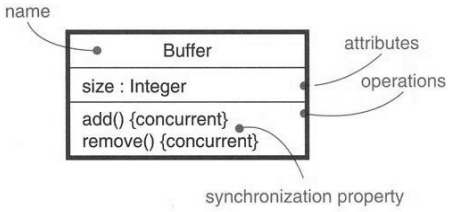
you have to model the synchronization of these two flows very carefully.

4. A message may be passed from a passive object to an active one. At first glance, this may seem illegal, but if you remember that every flow of control is rooted in some active object, you'll understand that a passive object passing a message to an active object has the same semantics as an active object passing a message to an active object.

Synchronization

- The problem arises when more than one flow of control is in one object at the same time. If you are not careful, anything more than one flow will interfere with another, corrupting the state of the object
- This is the classical problem of mutual exclusion. The key to solving this problem in object-oriented systems is by treating an object as a critical region.
- There are three alternatives to this approach, each of which involves attaching certain synchronization properties to the operations defined in a class. In the UML, you can model all three approaches.

1. Sequential	Callers must coordinate outside the object so that only one flow is in the object at a time. In the presence of multiple flows of control, the semantics and integrity of the object cannot be guaranteed.
2. Guarded	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by sequentializing all calls to all of the object's guarded operations. In effect, exactly one operation at a time can be invoked on the object, reducing this to sequential semantics.
3. Concurrent	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by treating the operation as atomic



Synchronization

Process Views

- Active objects play an important role in visualizing, specifying, constructing, and documenting a system's process view.
- The process view of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms.
- This view primarily addresses the performance, scalability, and throughput of the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view—that is, class diagrams, interaction diagrams, activity diagrams, and statechart diagrams,
- But with a focus on the active classes that represent these threads and processes.

UNIT – V
ARCHITECTURAL MODELING

Components

Terms and Concepts

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, as discussed in Chapter 12*; tagged values and compartments are discussed in* Chapter 6.

Every component must have a name that distinguishes it from other components. A *name* is a textual string. That name alone is known as a *simple name;* a *path name* is the component name prefixed by the name of the package in which that component lives. A component is typically drawn showing only its name, Just as with classes, you may draw components adorned with tagged values or with additional compartments to expose their details

Components and Classes

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

The first difference is the most important. When modeling a system, deciding if you should use a class or a component involves a simple decision—if the thing you are modeling lives directly on a node, use a component; otherwise, use a class.

The second difference suggests a relationship between classes and components. In particular, a component is the physical implementation of a set of other logical elements, such as classes and collaborations. As Figure 25-3 shows, the relationship between a component and the classes it implements can be shown explicitly by using a dependency relationship. Most of the time, you'll never need to visualize these relationships graphically. Rather, you will keep them as a part of the component's specification.

The third difference points out how interfaces bridge components and classes. As described in more detail in the next section, components and classes may both realize an interface, but a component's services are usually available only through its 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.

Using one of these facilities, you decompose your physical implementation by specifying interfaces that represent the major seams in the system. You then provide components that realize the interfaces, along with other components that access the services through their interfaces. This mechanism permits you to deploy a system whose services are somewhat location-independent and, as discussed in the next section, replaceable.

WE can show the relationship between a component and its interfaces in one of two ways. The first (and most common) style renders the interface in its elided, iconic form. The component that realizes the interface is connected to the interface using an elided realization relationship. The second style renders the interface in its expanded form, perhaps revealing its operations. The component that realizes the interface is connected to the interface using a full realization relationship. In both cases, the component that accesses the services of the other component through the interface is connected to the interface using a dependency relationship.

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. Also, a component may both import and export interfaces.

A given interface may be exported by one component and imported by another. The fact that this interface lies between the two components breaks the direct dependency between the components. A component that uses a given interface will function properly no matter what component realizes that interface. Of course, a component can be used in a context if and only if all its import interfaces are provided by the export interfaces of other components.

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

Second, a component is *replaceable.* A component is substitutable—it is possible to replace a component with another that conforms to the same interfaces. Typically, the mechanism of inserting or replacing a component to form a run time system is transparent to the component user and is enabled by object models (such as COM+ and Enterprise Java Beans) that require little or no intervening transformation or by tools that automate the mechanism.

Third, a component is *part of a system.* A component rarely stands alone. Rather, a given component collaborates with other components and in so doing exists in the architectural or technology context in which it is intended to be used. A component is logically and physically cohesive and thus denotes a meaningful structural and/or behavioral chunk of a larger system. A component may be reused across many systems. Therefore, a component represents a fundamental building block on which systems can be designed and composed. This definition is recursive—a system at one level of abstraction may simply be a component at a higher level of abstraction.

Fourth, as discussed in the previous section, a component *conforms to and provides the realization of a set of interfaces.*

Kinds of Components

Three kinds of components may be distinguished.

First, there are *deployment components.* These are the components necessary and sufficient to form an executable system, such as dynamic libraries (DLLs) and executables (EXEs). The UML's definition of component is broad enough to address classic object models, such as COM+, CORBA, and Enterprise Java Beans, as well as alternative object models, perhaps involving dynamic Web pages, database tables, and executables using proprietary communication mechanisms.

Second, there are *work product components.* These components are essentially the residue of the development process, consisting of things such as source code files and data files from which deployment components are created. These components do not directly participate in an executable system but are the work products of development that are used to create the executable system.

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

Organizing Components

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

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

Standard Elements

The UML's extensibility mechanisms

All the UML's extensibility mechanisms apply to components. Most often, you'll use tagged values to extend component properties (such as specifying the version of a development component) and stereotypes to specify new kinds of components (such as operating system-specific components).

The UML's standard elements are summarized in Appendix B.

The UML defines five standard stereotypes that apply to components:

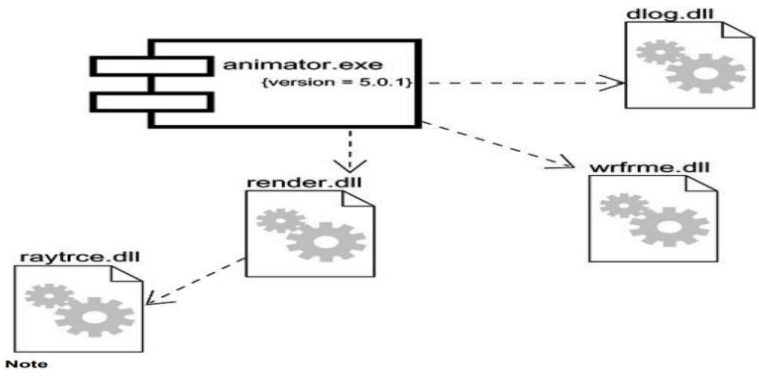
1. `executable`	Specifies a component that may be executed on a node
2. `library`	Specifies a static or dynamic object library
3. `table`	Specifies a component that represents a database table
4. `file`	Specifies a component that represents a document containing source code or data
5. `document`	Specifies a component that represents a document

Common Modeling Techniques

Modeling Executables and Libraries

To model executables and libraries,

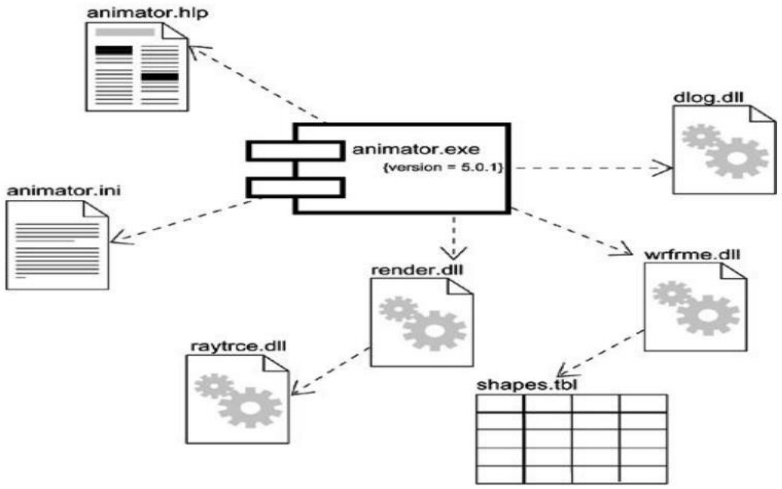
- Identify the partitioning of your physical system. Consider the impact of technical, configuration management, and reuse issues.
- Model any executables and libraries as components, using the appropriate standard elements. If your implementation introduces new kinds of components, introduce a new appropriate 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.



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.

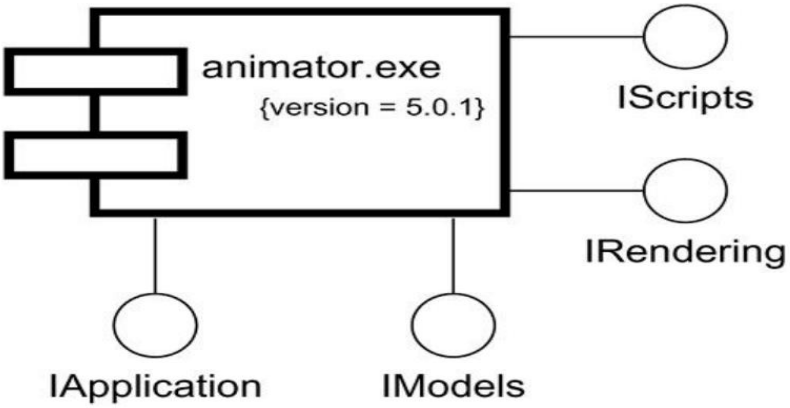


Modeling an API

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.

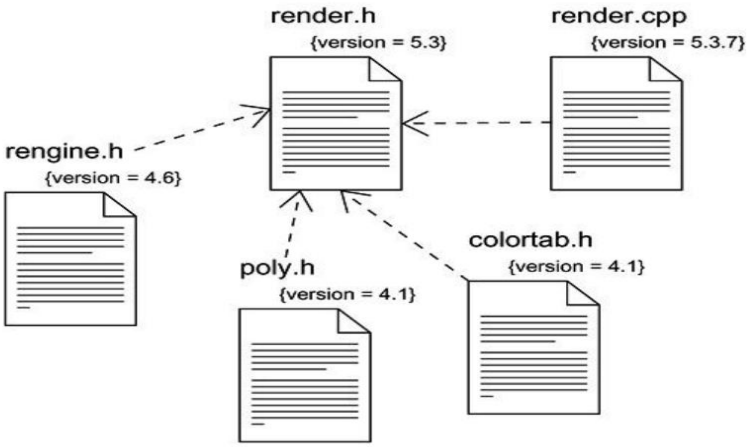
Figure 25-7 Modeling an API



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.



Deployment

Terms and Concepts

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

Names

A node name must be unique within its enclosing package,

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 Figure 26-2. Just as with classes, you may draw nodes adorned with tagged values or with additional compartments to expose their details.

Nodes and Components

In many ways, nodes are a lot like components: Both have names; both may participate in dependency, generalization, and association relationships; both may be nested; both may have instances; both may be participants in interactions. However, 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.

This first difference is the most important. Simply put, nodes execute components; components are things that are executed by nodes.

The second difference suggests a relationship among classes, components, and nodes. In particular, a component is the materialization of a set of other logical elements, such as classes and collaborations, and a node is the location upon which components are deployed. A class may be implemented by one or more components, and, in turn, a component may be deployed on one or more nodes. the relationship between a node and the components it deploys can be shown explicitly by using a dependency relationship. Most of the time, you won't need to visualize these relationships graphically but will keep them as a part of the node's specification.

Organizing Nodes

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

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

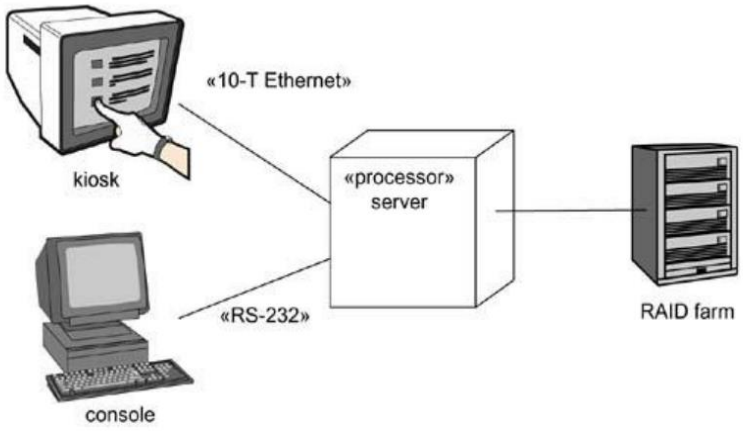
Common Modeling Techniques

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.

Figure 26-5 Processors and Devices



Modeling the Distribution of Components

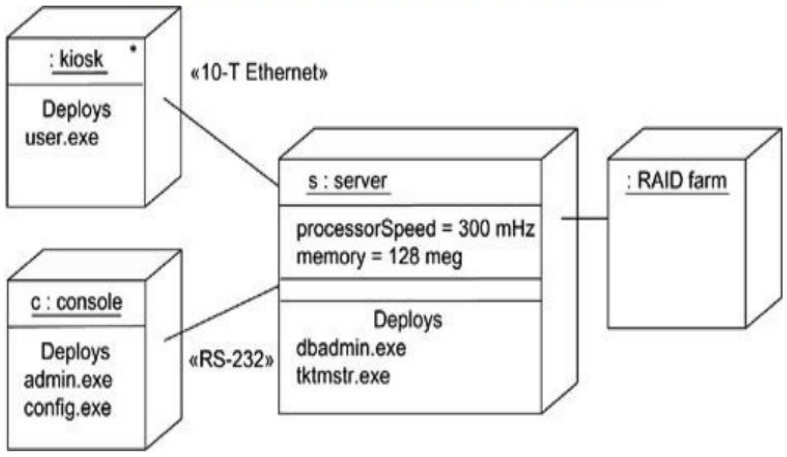
When you model the topology of a system, it's often useful to visualize or specify the physical distribution of its components across the processors and devices that make up the system.

To model the distribution of components,

- For each significant component in your system, allocate it to a given node.
- Consider duplicate locations for components. It's not uncommon for the same kind of component (such as specific executables and libraries) to reside on multiple nodes simultaneously.

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

Figure 26-6 Modeling the Distribution of Components.



Collaborations

A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all its parts. A collaboration is also the specification of how an element, such as a classifier (including a class, interface, component, node, or use case) or an operation, is realized by a set of classifiers and associations playing specific roles used in a specific way. Graphically, a collaboration is rendered as an ellipse with dashed lines.

Names

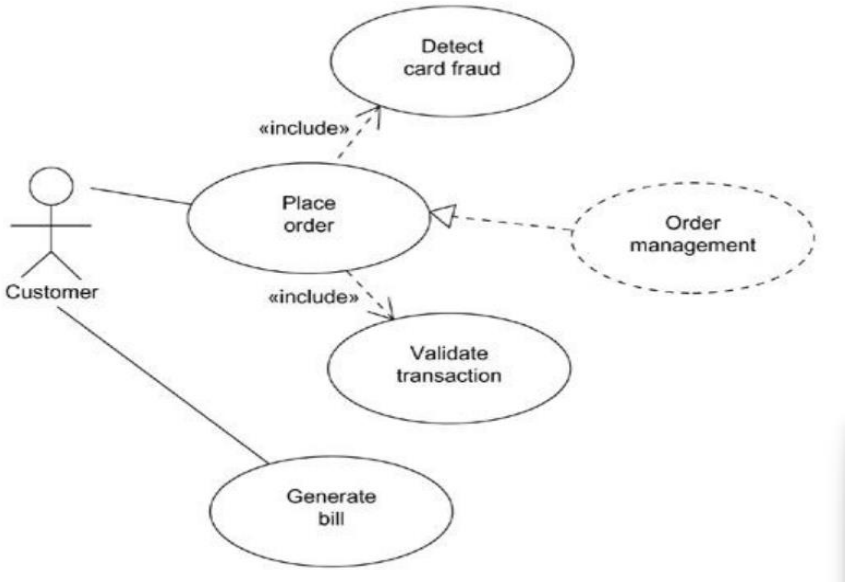
A collaboration name must be unique within its enclosing package.Every collaboration must have a name that distinguishes it from other collaborations. A name is a textual string. That name alone is known as a simple name; a path name is the collaboration name prefixed by the name of the package in which that collaboration lives. Typically, a collaboration is drawn showing only its name.

Common Modeling Techniques

Modeling the Realization of a Use Case

To model the realization of a use case,

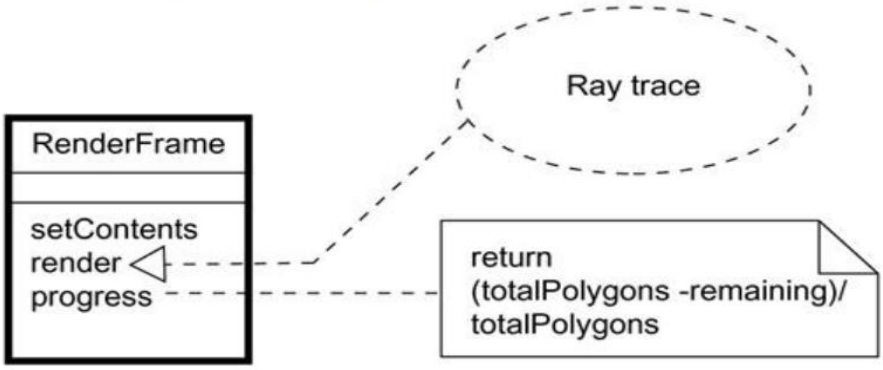
- Identify those structural elements necessary and sufficient to carry out the semantics of the use case.
- Capture the organization of these structural elements in class diagrams.
- Consider the individual scenarios that represent this use case. Each scenario represents a specific path through the use case.
- Capture the dynamics of these scenarios in interaction diagrams. Use sequence diagrams if you want to emphasize the time ordering of messages. Use collaboration diagrams if you want to emphasize the structural relationships among these objects as they collaborate.
- Organize these structural and behavioral elements as a collaboration that you can connect to the use case via realization.



Modeling the Realization of an Operation:

- Identify the parameters, return value, and other objects visible to the operation.
- If the operation is trivial, represent its implementation directly in code, which you can keep in the backplane of your model, or explicitly visualize it in a note.
- If the operation is algorithmically intensive, model its realization using an activity diagram.
- If the operation is complex or otherwise requires some detailed design work, represent its implementation as a collaboration. You can further expand the structural and behavioral parts of this collaboration using class and interaction diagrams, respectively.

Figure 27-6 Modeling the Realization of an Operation



Modeling a Mechanism

To model a mechanism,

- Identify the major mechanisms that shape your system's architecture. These mechanisms are driven by the overall architectural style you choose to impose on your implementation, along with the style appropriate to your problem domain.
- Represent each of these mechanisms as a collaboration.
- Expand on the structural and behavioral part of each collaboration. Look for sharing, where possible.
- Validate these mechanisms early in the development lifecycle (they are of strategic importance), but evolve them with each new release, as you learn more about the details of your implementation.

Patterns and Frameworks

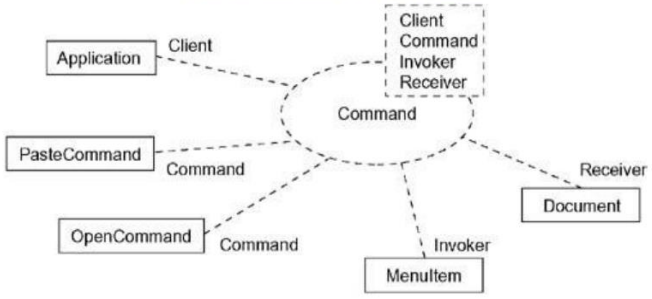
A pattern is a common solution to a common problem in a given context. A mechanism is a design pattern that applies to a society of classes. A framework is an architectural pattern that provides an extensible template for applications within a domain.

A framework is an architectural pattern that provides an extensible template for applications within a domain. For example, one common architectural pattern you'll encounter in real time systems is a cyclic executive, which divides time into frames and subframes, during which processing takes place under strict deadlines. Choosing this pattern versus its alternative (an even-driven architecture) colors your entire system. Because this pattern (and its alternative) is so common, it makes sense to name it as a framework.

Common Modeling Techniques
Modeling Design Pattern

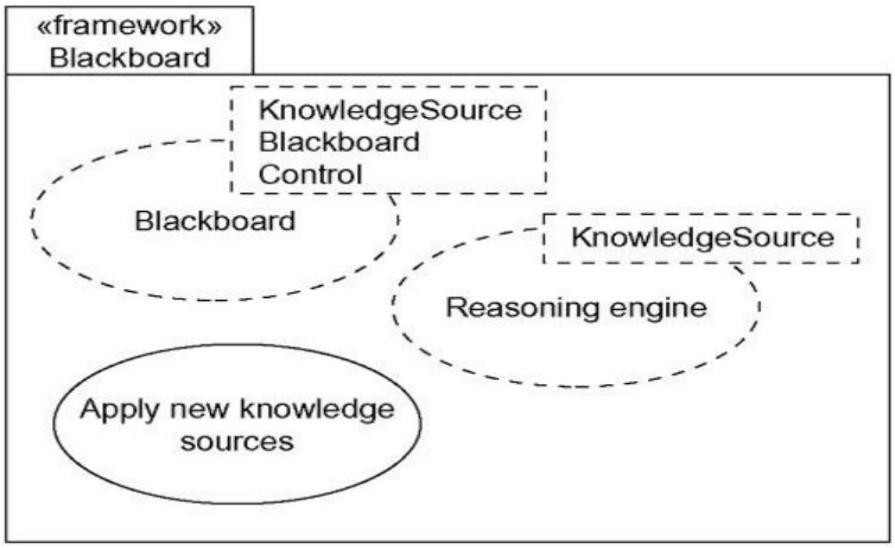
· Identify the common solution to the common problem and reify it as a mechanism.
· Model the mechanism as a collaboration, providing its structural, as well as its behavioral, aspects.
· Identify the elements of the design pattern that must be bound to elements in a specific context and render them as parameters to the collaboration.

Figure 28-4 Modeling a Design Pattern



Modeling Architectural Patterns

· Harvest the framework from an existing, proven architecture.
· Model the framework as a stereotyped package, containing all the elements (and especially the design patterns) that are necessary and sufficient to describe the various views of that framework.
· Expose the slots, tabs, knobs, and dials necessary to adapt the framework in the form of design patterns and collaborations. For the most part, this means making it clear to the user of the pattern which classes must be extended, which operations must be implemented, and which signals must be handled.



Component Diagrams

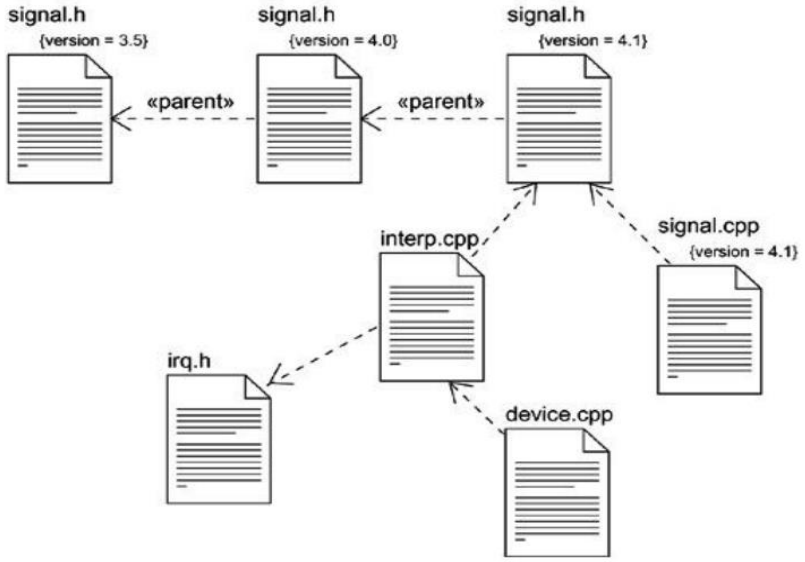
Component diagrams commonly contain
· Components
· Interfaces
· Dependency, generalization, association, and realization relationships
Like all other diagrams, component diagrams may contain notes and constraints.
Component diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place instances in your component diagrams, as well, especially when you want to visualize one instance of a family of component-based systems.

Common Modeling Techniques

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.

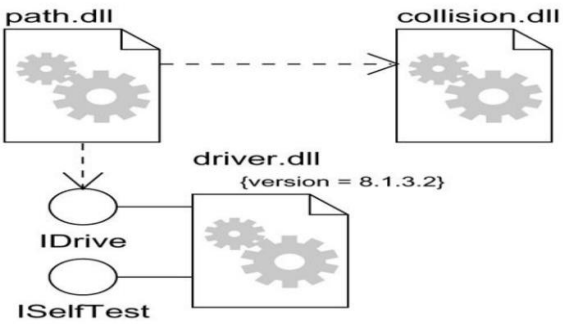


Modeling an Executable Release

· 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.

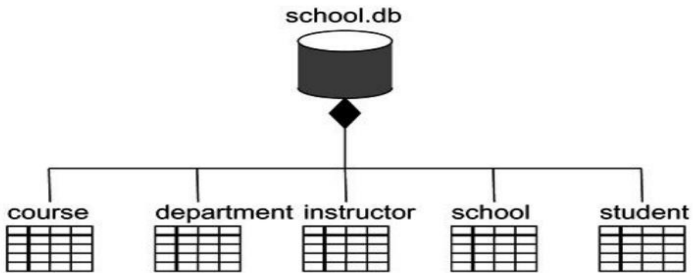
· Consider the stereotype of each component in this set. For most systems, you'll find a small number of different kinds of components (such as executables, libraries, tables, files, and documents). You can use the UML's extensibility mechanisms to provide visual cues for these stereotypes.

· 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.



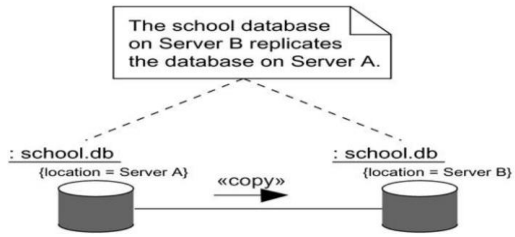
Modeling a Physical Database

· 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.



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.



Deployment Diagrams

A deployment diagram 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.

Deployment diagrams commonly contain
· Nodes
· Dependency and association relationships
Like all other diagrams, deployment diagrams may contain notes and constraints.

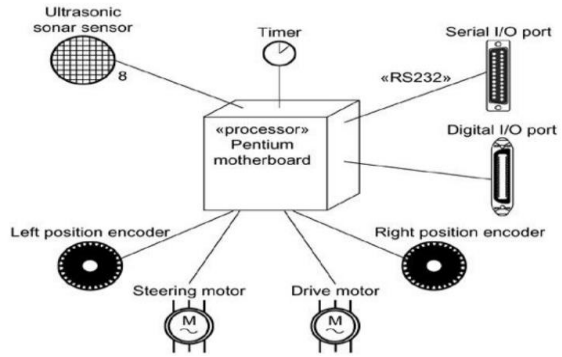
Deployment diagrams may also contain components, each of which must live on some node. Deployment diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place instances in your deployment diagrams, as well, especially when you want to visualize one instance of a family of hardware topologies.

Common Modeling Techniques

Modeling an Embedded System

To model an embedded system,

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

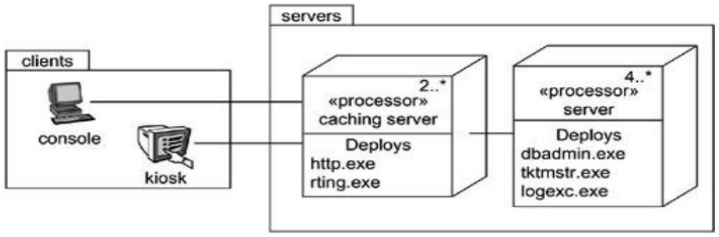


Modeling a Client/Server System

To model a client/server system,

- Identify the nodes that represent your system's client and server processors.
- Highlight those devices that are germane to the behavior of your system. For example, you'll want to model special devices, such as credit card readers, badge readers, and display devices other than monitors, because their placement in the system's hardware topology are likely to be architecturally significant.
- Provide visual cues for these processors and devices via stereotyping.
- Model the topology of these nodes in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.

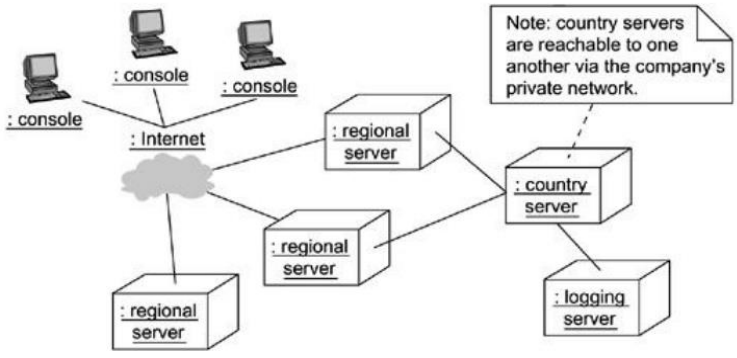
Figure 30-3 Modeling a Client/Server System



Modeling a Fully Distributed System

To model a fully distributed system,

- Identify the system's devices and processors as for simpler client/server systems.
- If you need to reason about the performance of the system's network or the impact of changes to the network, be sure to model these communication devices to the level of detail sufficient to make these assessments.
- Pay close attention to logical groupings of nodes, which you can specify by using packages.
- Model these devices and processors using deployment diagrams. Where possible, use tools that discover the topology of your system by walking your system's network.
- If you need to focus on the dynamics of your system, introduce use case diagrams to specify the kinds of behavior you are interested in, and expand on these use cases with interaction diagrams.



Case study:

LIBRARY MANAGEMENT SYSTEM

SYSTEM REQUIREMENT SPECIFICATION

To model the library management system using 'Object Oriented Analysis and Designing' concept and the unified modeling language (UML) objective behind the development of this software is to model an object oriented system which helps the librarians to easily performs their activities. It must as a best interface between the librarian and the computer so as to perform the required activities effectively. It must cover the activities of maintaining the records of issued and received books and

Other things available in the library separately for both faculty and student when by taking the scenario as a college. Uml diagrams drew below helps us to easily understand how the software is developed. The diagrams are class, use-case, sequence, collaborations, and component and deployment diagrams.

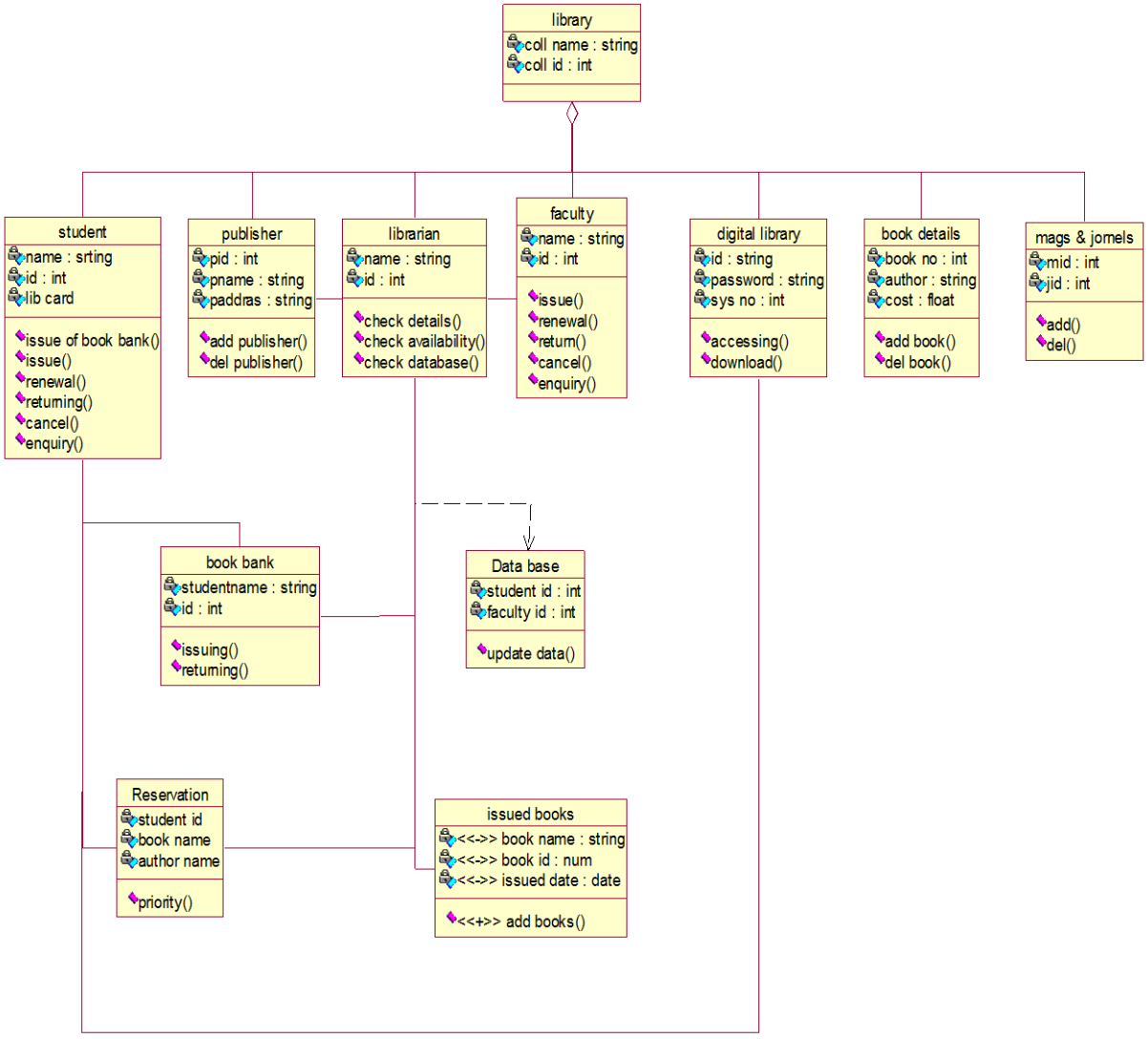
CLASS DIAGRAM:

Class diagram shows structure of the software system. The class diagram shows a set of class, interfaces and their relationships. The components are:

a) Class
b) Relationship

The forms of relationship are:

1. Associations
2. Aggregations
3. Generalization
4. Compositions
5. Dependency



USE CSE DIAGRAM

Use case diagram is created to visualize the interaction of our system with the outside world. The components of use case diagrams are:

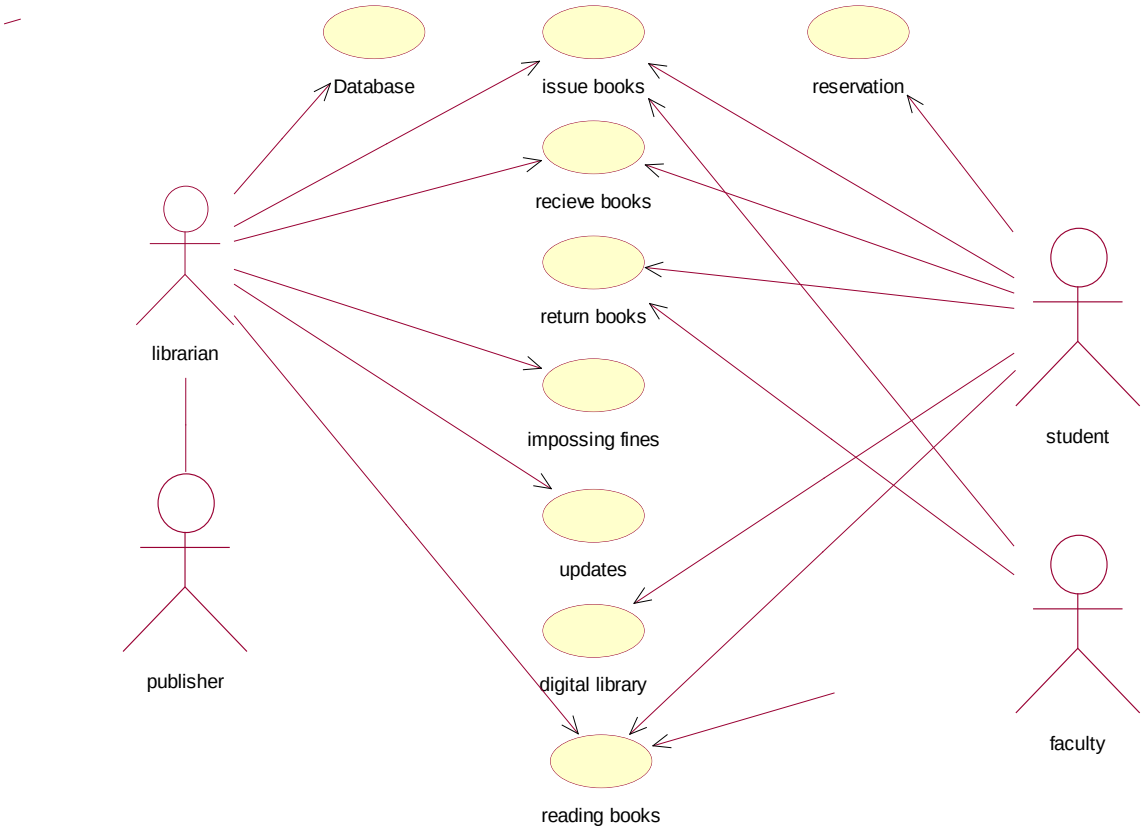
Use Case: Scenarios of the system

Actors: some or something who is interacting with the system

Relationship: semantic link between use case and actor. The forms of relationship are:

a) Association
b) Dependency
c) Generalization

Use case diagram of Library Management:



INTERACTION DIAGRAM

An interaction diagram models the dynamic aspects of the system by showing the relationship among the objects and message they may dispatch. There are two types of interaction diagrams:

SEQUENCE DIAGRAM

Sequence diagram shows the steps to steps what much happen to accomplish a piece of functionality provided by the system. The components are:

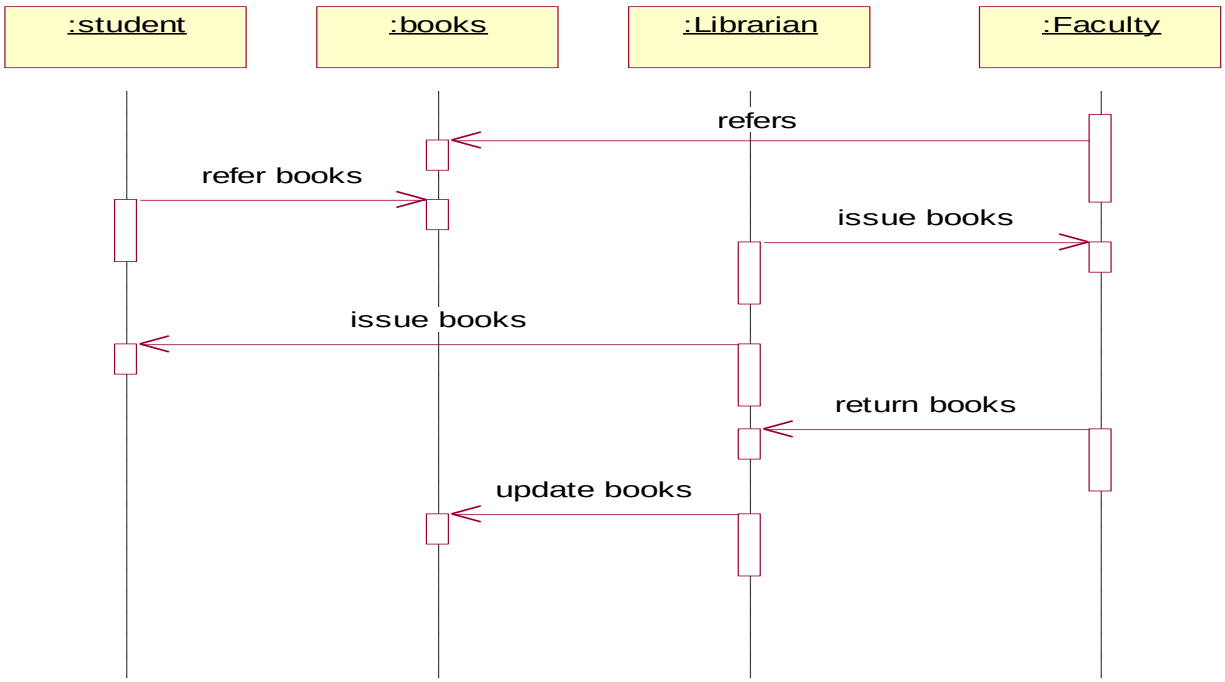
a) Actors
b) Objects
c) Message
d) Lifeline
e) Focus of Control

COLLABORATION DIAGRAM

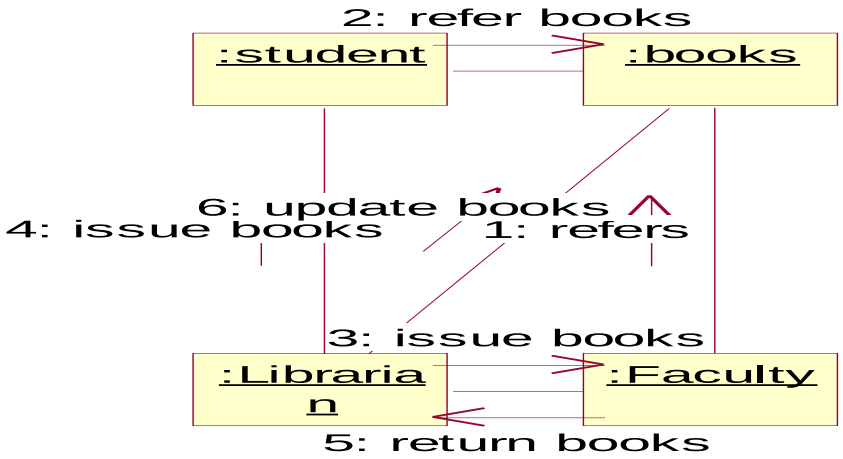
Collaboration diagrams displays object interactions organized around object and their links to one another. The components are:

a) Actor
b) Object
c) Link

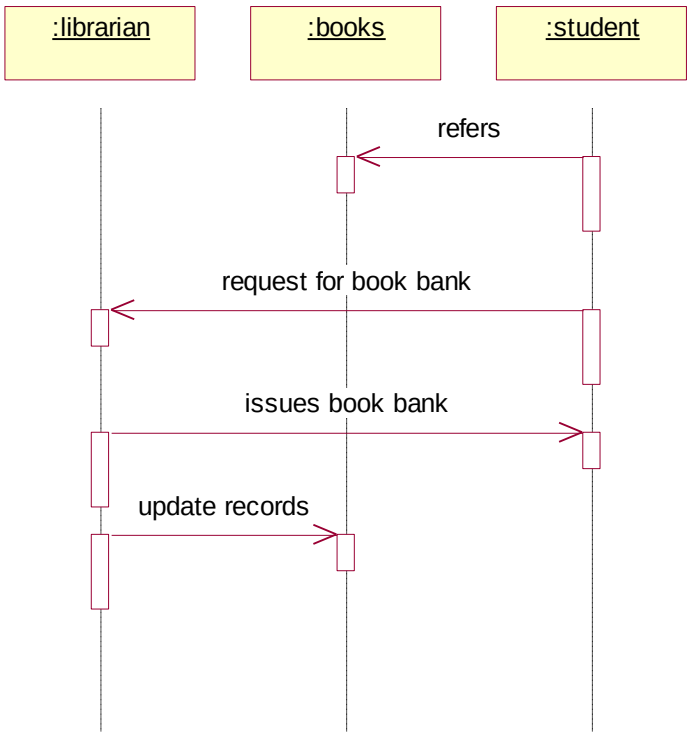
Sequence diagram for Issue and Return of Books:



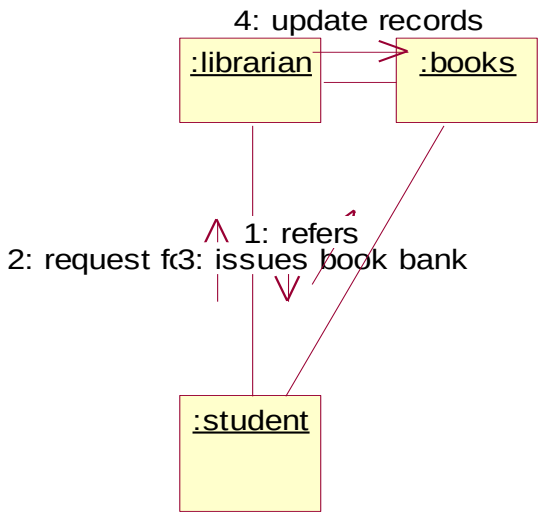
Collaboration diagram for Issue and Return of Books:



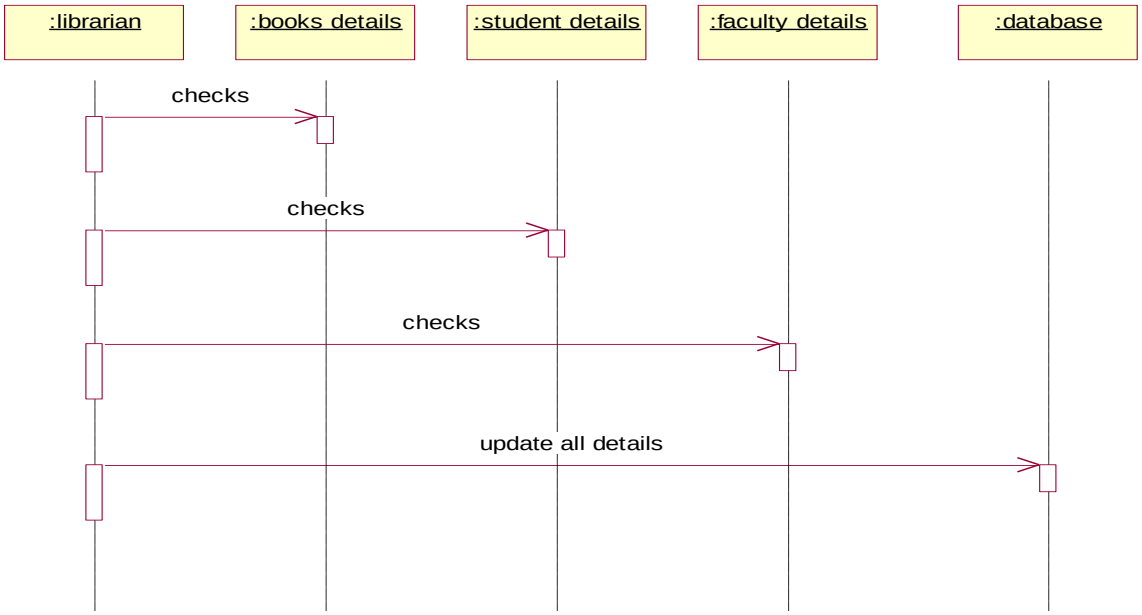
Sequence diagram for Book Bank:



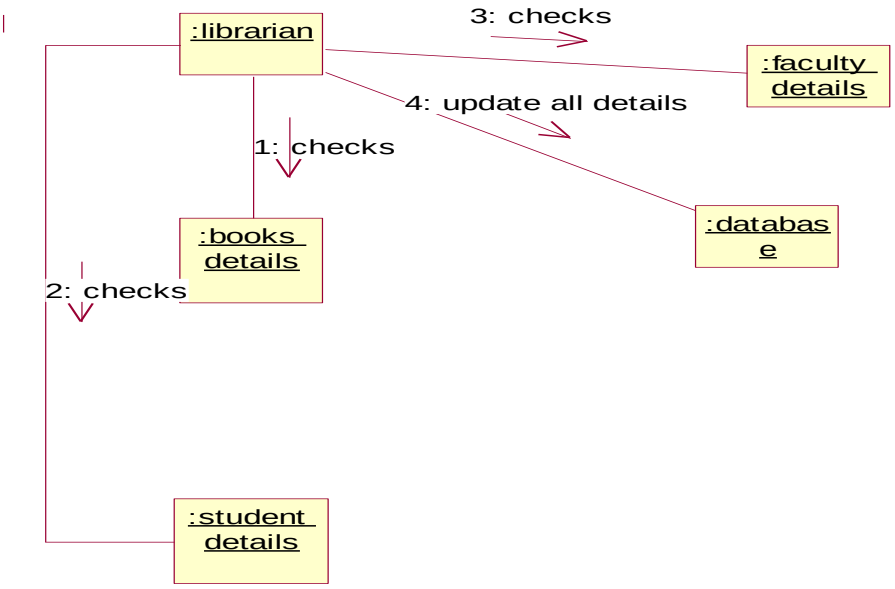
Collaboration diagram for Book Bank:



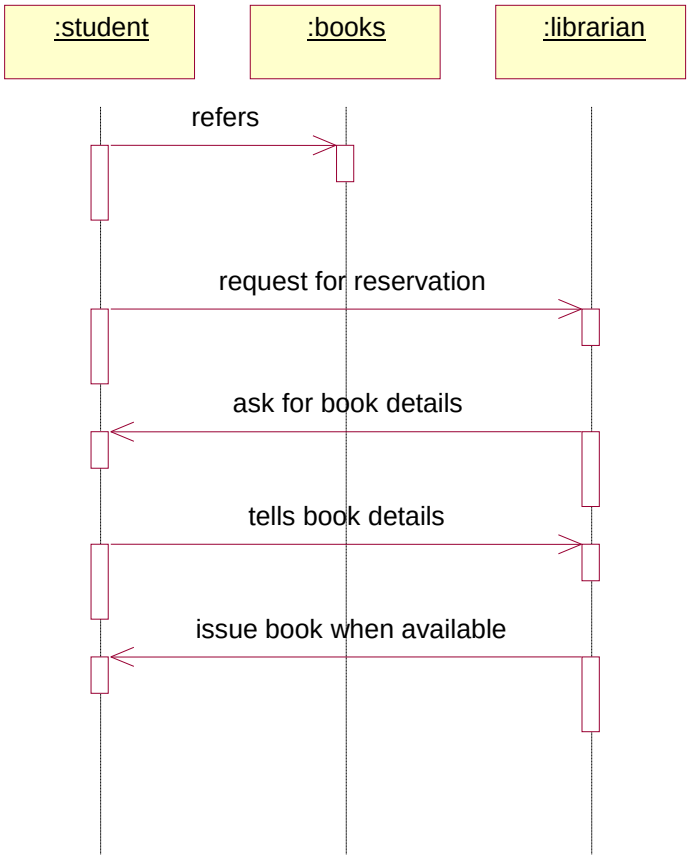
Sequence diagram for Database:



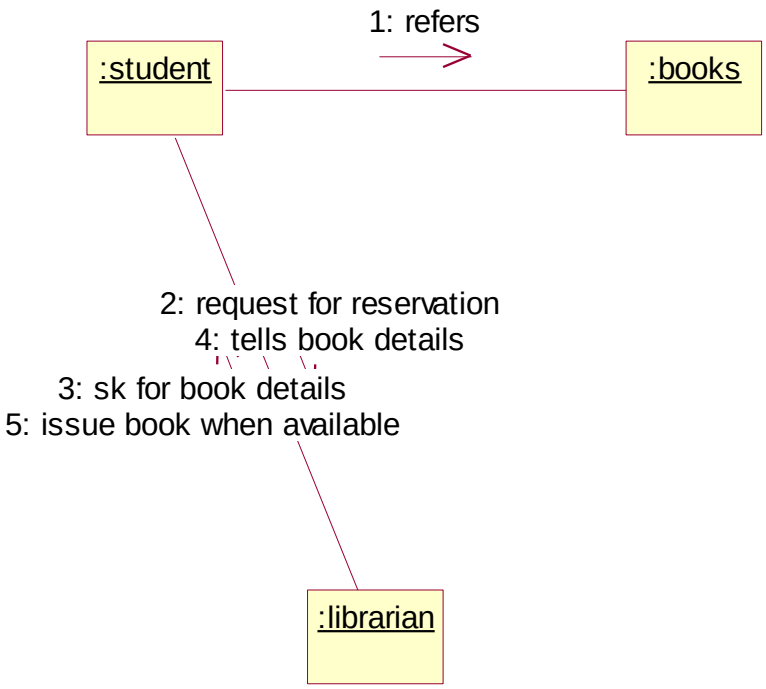
Collaboration diagram for Database:



Sequence diagram for Reservation:



Collaboration diagram for Reservation:

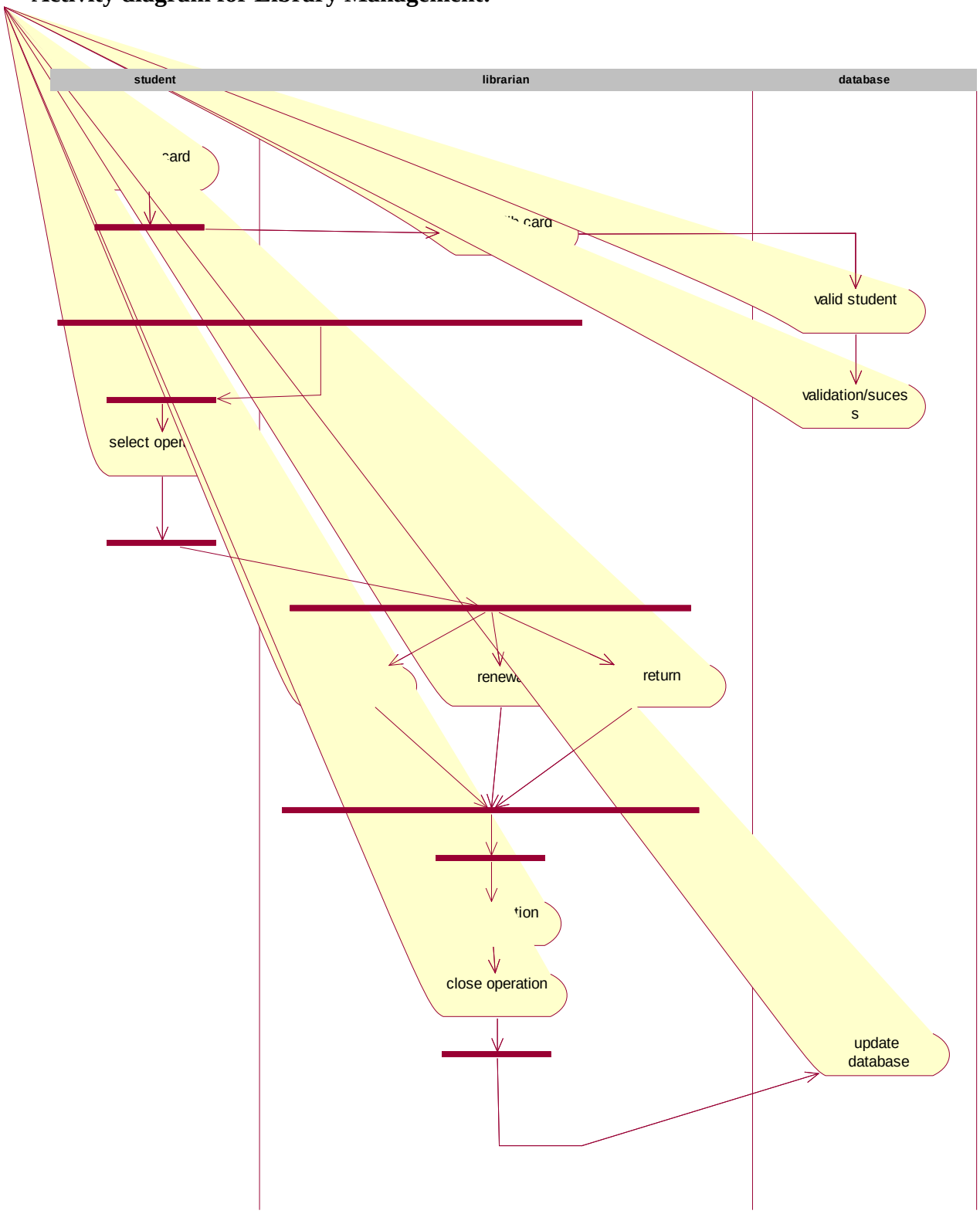


ACTIVITY DIAGRAM

Activity diagram shows the flow of events within our systems.
The components are:

a) Start state
b) End state
c) Transaction
d) Decision Box
e) Synchronization Bar
f) Swim lane

Activity diagram for Library Management:

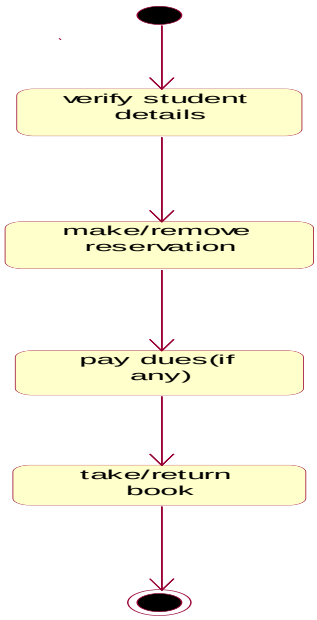


STATE CHART

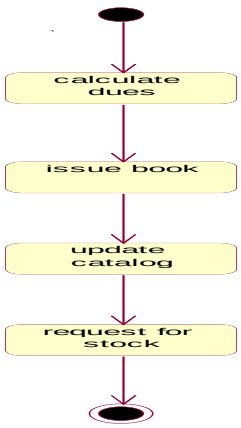
State chart diagrams shows a life cycle of a single class. The state is a condition where the object may be in. The components are:

a) Start state
b) End state
c) State
d) Transition

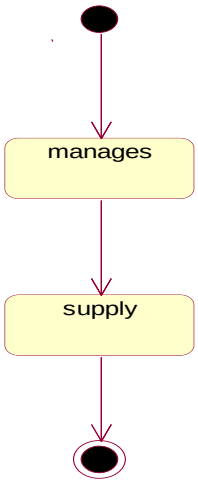
State Chart diagram of Student:



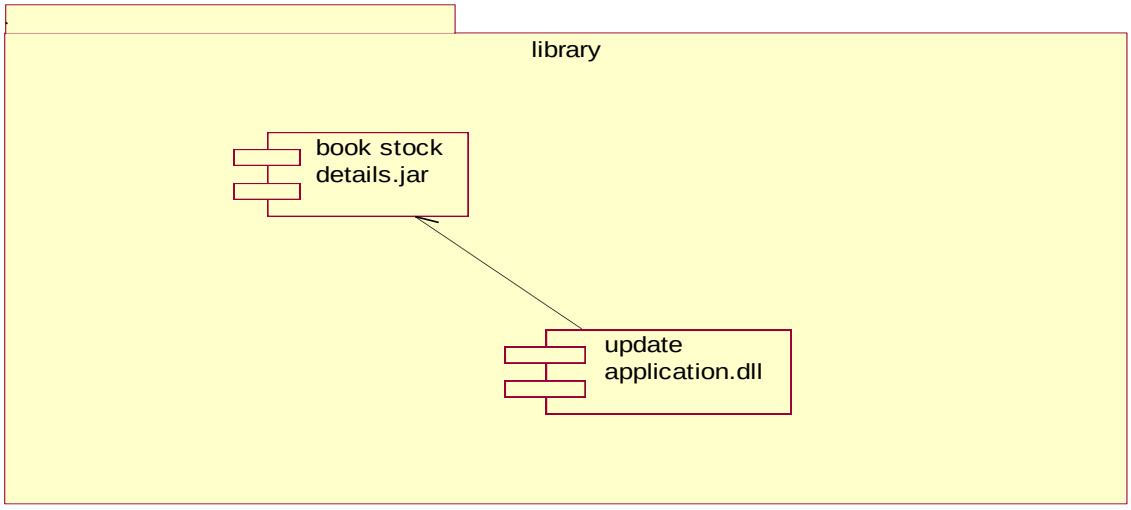
State Chart diagram of Librarian:



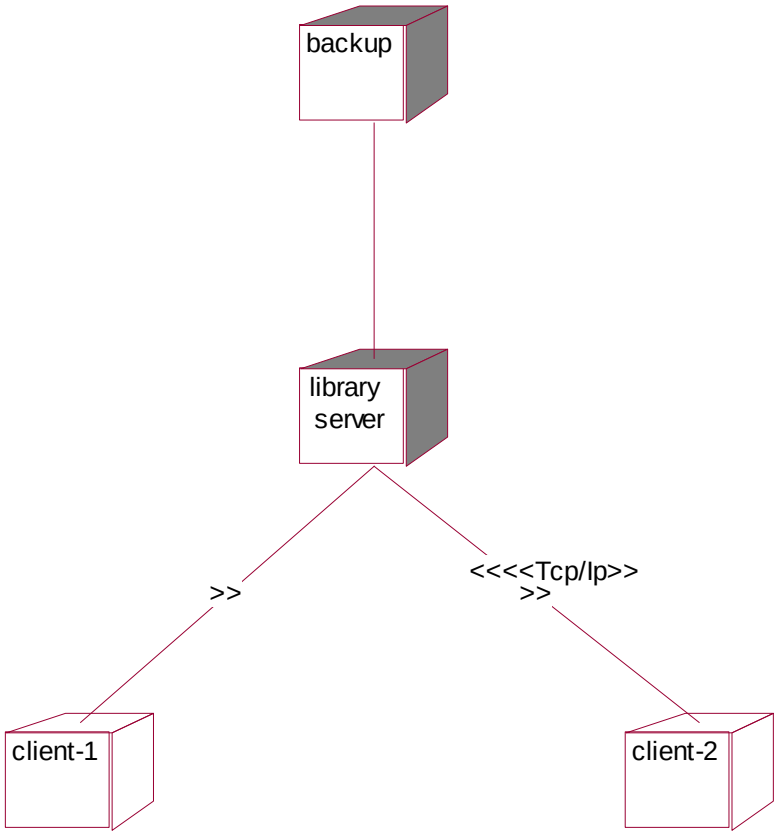
State Chart diagram of Database:



Component diagram of Library Management:



Deployment diagram of Library Management:



UNIFIED LIBRARY APPLICATION

INTRODUCTION

Unified Library Application System emphasizes on the online reservation, issue and return of books. This system globalizes the present library system. Using this application the member can reserve any book from anywhere in the world. Still in nascent stages, this application soon revolutionizes present library system.

Let us just have an overview of the unified library application system:

- Librarian lends books and magazines
- Librarian maintains the list of all the members of library

- Borrower makes reservation online
- Borrower can remove reservation online
- Librarian issues books to the borrower
- Librarian calculates dues to be paid by the borrower
- Borrower issues/returns books and/or magazines
- Librarian places order about the requirements to the master librarian
- Librarian updates system
- Master librarian maintains librarians

TEXTUAL ANALYSIS

(a) ACTORS

i. Librarian
ii. Borrower
iii. Catalog
iv. Master Librarian

(b) VERBS

i. Borrower:
    1. Logs into the system
    2. Browses/searches for books or magazines
    3. Makes/removes reservation
    4. Views results and reports from the unified library application system
ii. Librarian:
    1. Manages and validates members
    2. View reports from the system

3. Issues books
4. Calculates dues
5. Takes books
6. Places orders to the master librarian
7. Maintains list of books and magazine

iii. Master Librarian

1. Maintains other librarians

USECASE DIAGRAM:

Use case diagram is created to visualize the interaction of our system with the outside world. The components of use case diagram are:

Use Case: Scenarios of the system

Actor: Someone or something who is interacting with the system Relationship: Semantic link between use case and actor.

TKR COLLEGE OF ENGINEERING AND TECHNOLOGY

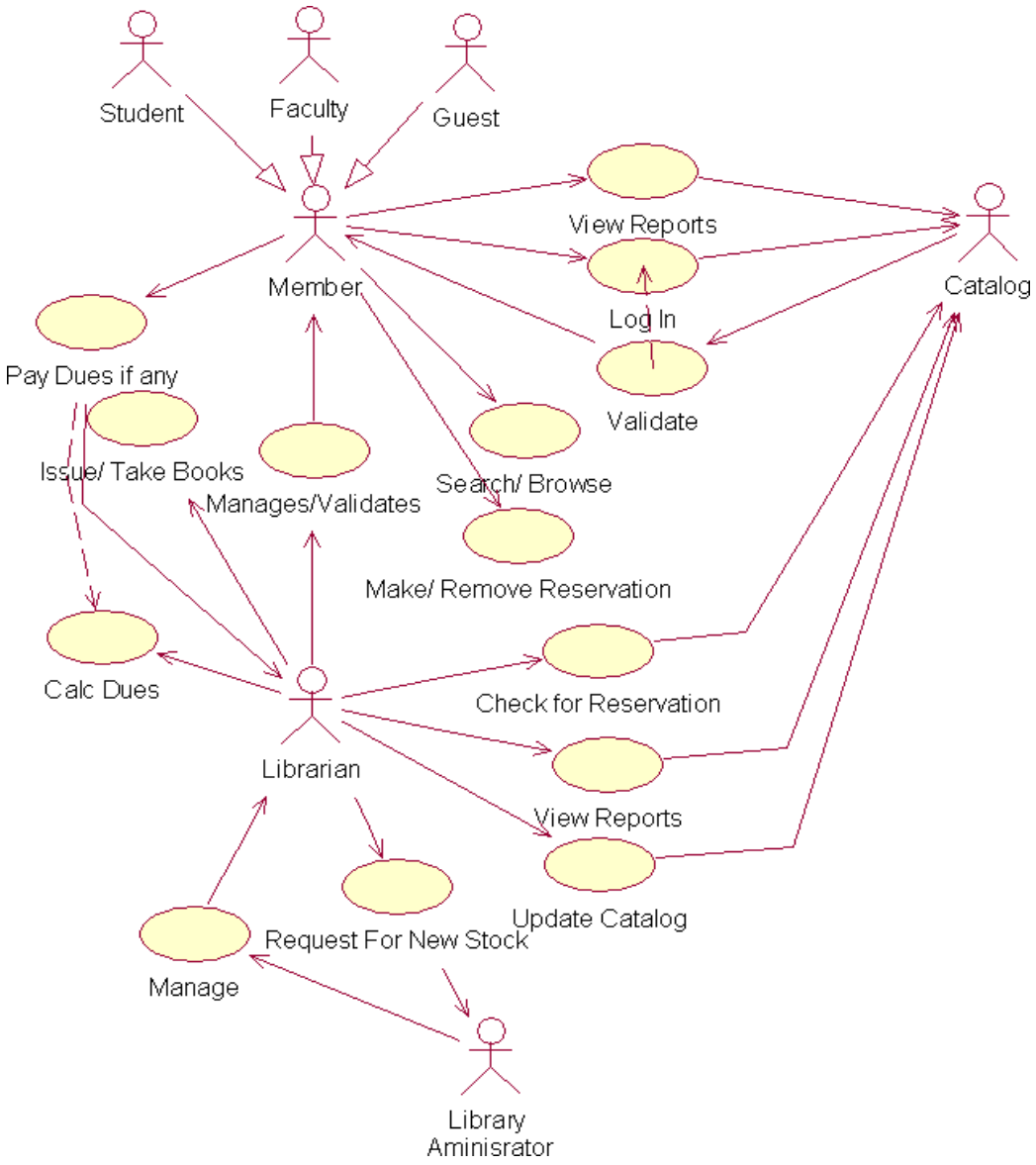
AN AUTONOMOUS INSTITUTION

Accredited by NAAC with 'A+' Grade.

(Sponsored by TKR Educational Society, Approved by AICTE, Affiliated to JNTU H)

Medbowli, Meerpet, Balapur, Hyderabad, Telangana – 500 097

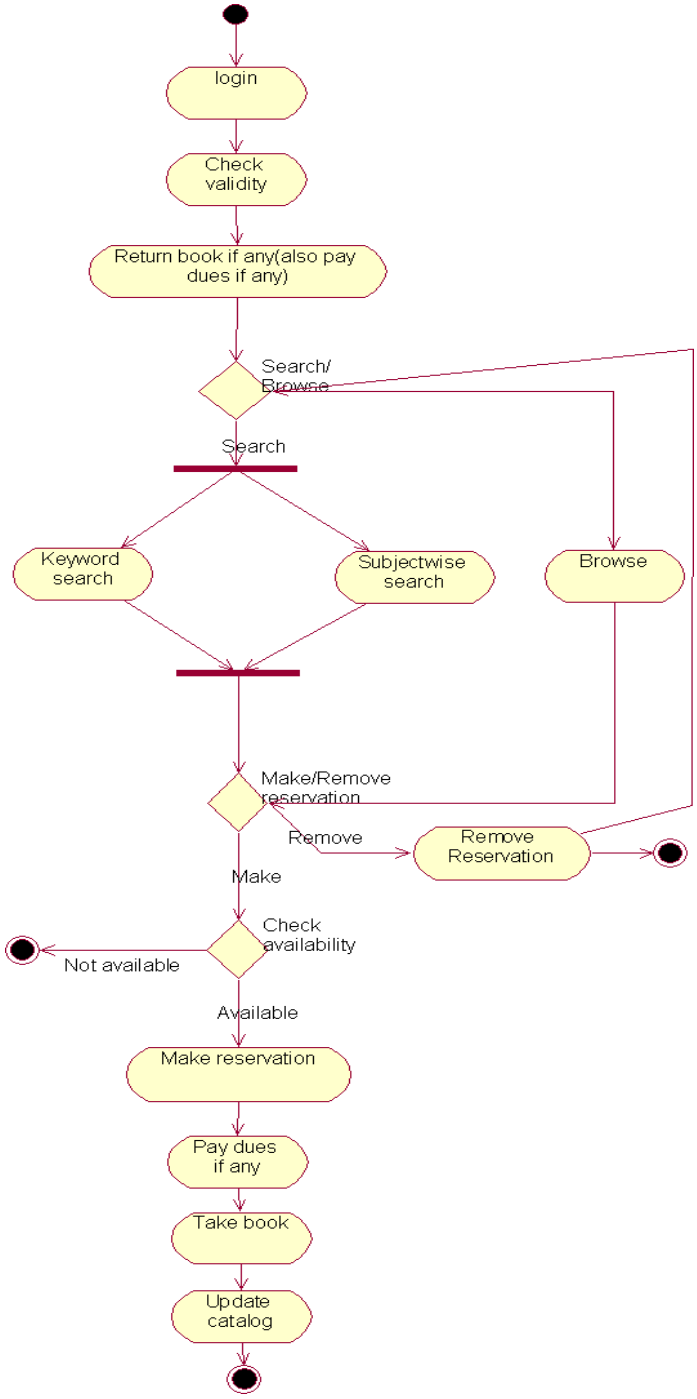
Phone: 9100377790, email: info@tkrcet.ac.in, web site: www.tkrcet.ac.in



ACTIVITY DIAGRAM:

Activity diagram shows the flow of events within our system. The components are:

a) Start State
b) End State
c) Transition
d) Decision Box
e) Synchronization Bar
f) Swim Lane


login
Check validity
Return book if any(also pay dues if any)
Search/ Browse
Search
Keyword search
Subjectwise search
Browse
Make/Remove reservation
Remove
Remove Reservation
Make
Check availability
Not available
Available
Make reservation
Pay dues if any
Take book
Update catalog

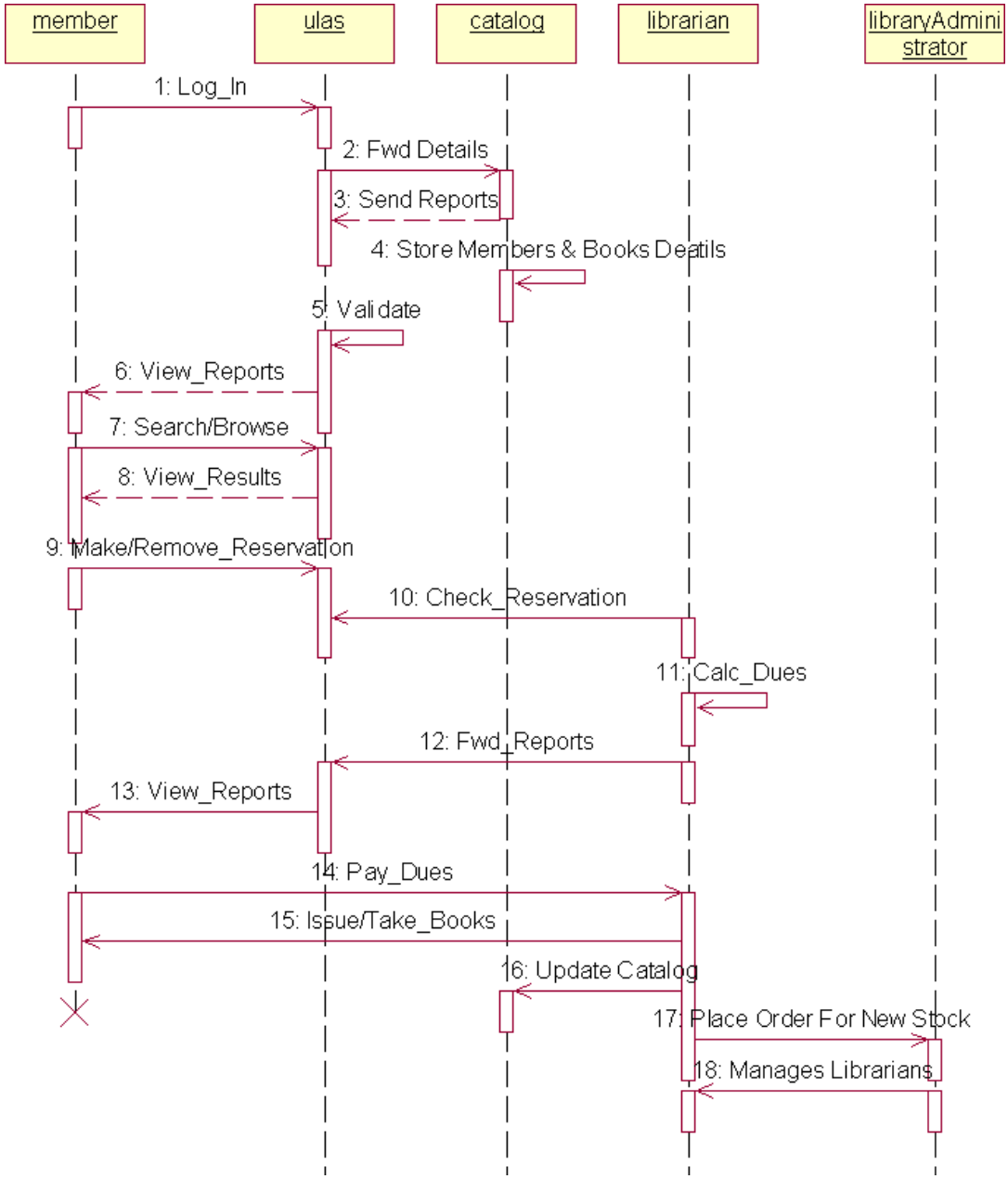
INTERACTION DIAGRAM:

An interaction diagram models the dynamic aspects of the system by showing the relationship among the objects and messages they may dispatch. There are two types of interaction diagrams:

SEQUENCE DIAGRAM:

Sequence diagram shows the step to step what mush happen to accomplish a piece of functionality provided by the system. The components are:

a) Actor
b) Object
c) Messages
d) Lifeline
e) Focus of Control


member
ulas
catalog
librarian
libraryAdministrator
1: Log_In
2: Fwd Details
3: Send Reports
4: Store Members & Books Deatils
5: Validate
6: View_Reports
7: Search/Browse
8: View_Results
9: Make/Remove_Reservation
10: Check_Reservation
11: Calc_Dues
12: Fwd_Reports
13: View_Reports
14: Pay_Dues
15: Issue/Take_Books
16: Update Catalog
17: Place Order For New Stock
18: Manages Librarians

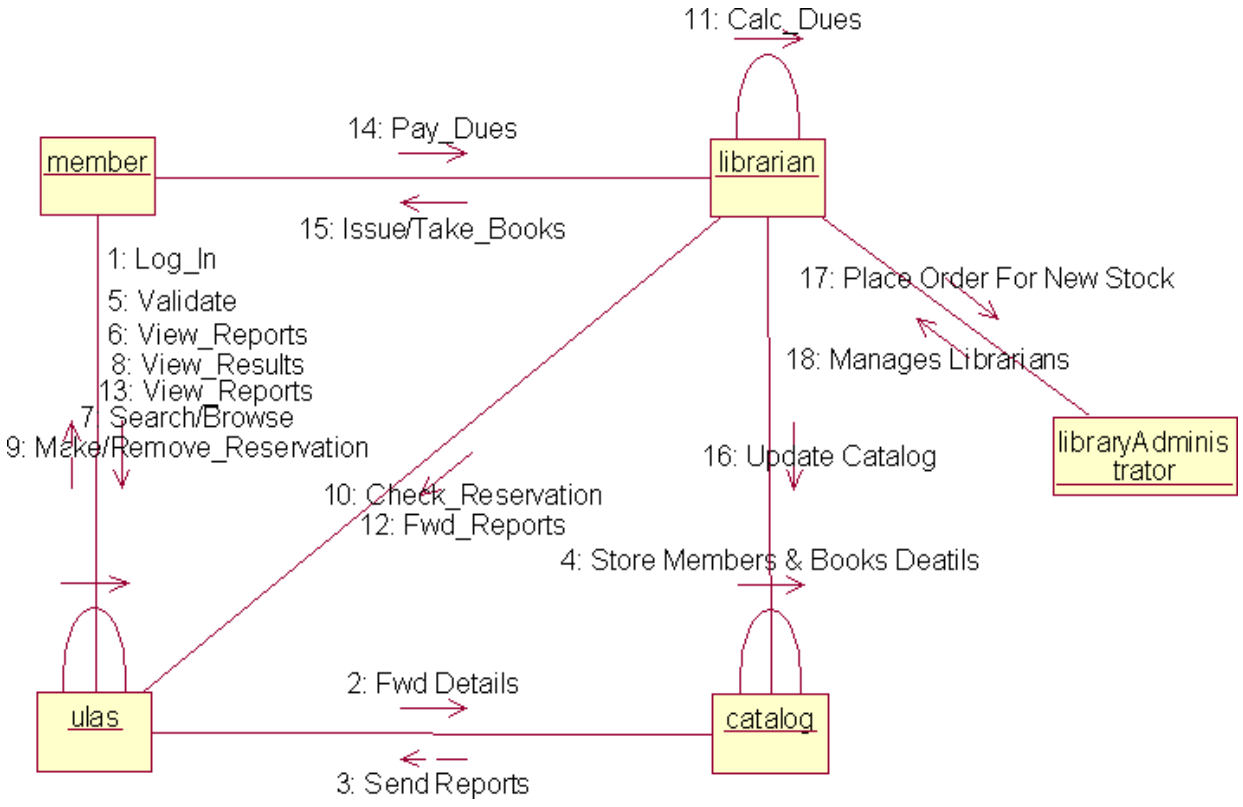
COLLABORATION DIAGRAM:

Collaboration diagram displays object interactions organized around objects and their links to one another. The components are:

a) Actor

b) Object

c) Link

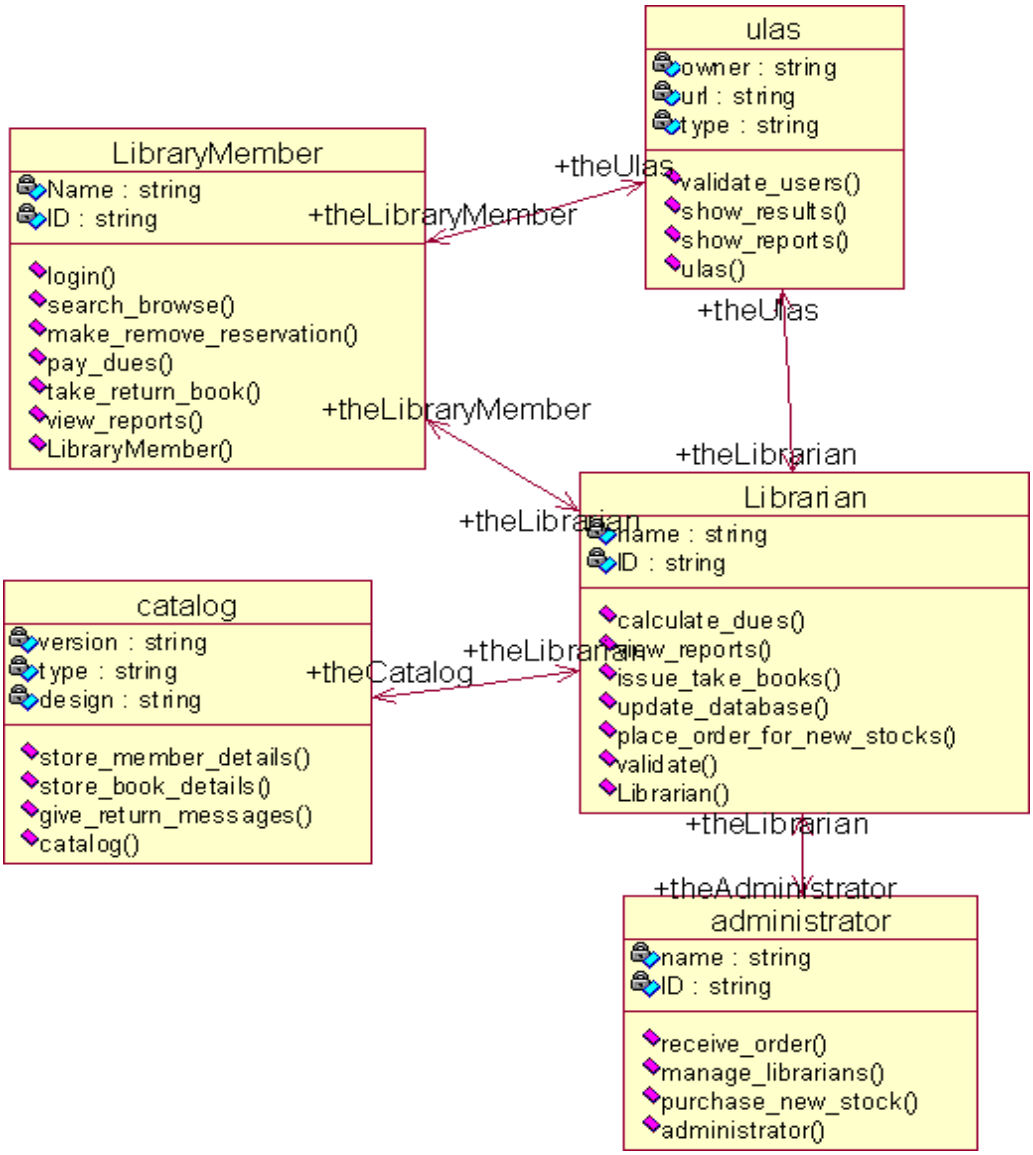


CLASS DIAGRAM:

Class diagram shows structure of the software system. The class diagram shows a set of classes, interfaces and their relationships. The components are:

a) Class
b) Relationship:
   The forms of relationship are:

   1. Association
   2. Aggregation
   3. Generalization
   4. Composition
   5. Dependency


ulas
owner : string
url : string
type : string
validate_users()
show_results()
show_reports()
ulas()
LibraryMember
Name : string
ID : string
login()
search_browse()
make_remove_reservation()
pay_dues()
take_return_book()
view_reports()
LibraryMember()
+theUlas
+theLibraryMember
+theLibraryMember
+theUlas
+theLibrarian
+theLibrarian
Librarian
name : string
ID : string
calculate_dues()
view_reports()
issue_take_books()
update_database()
place_order_for_new_stocks()
validate()
Librarian()
catalog
version : string
type : string
design : string
store_member_details()
store_book_details()
give_return_messages()
catalog()
+theCatalog
+theLibrarian
+theLibrarian
+theAdministrator
administrator
name : string
ID : string
receive_order()
manage_librarians()
purchase_new_stock()
administrator()

STATE CHART DIAGRAM:

State chart diagram show a life cycle of a single class. The state is a condition where the object may be in. The components are:

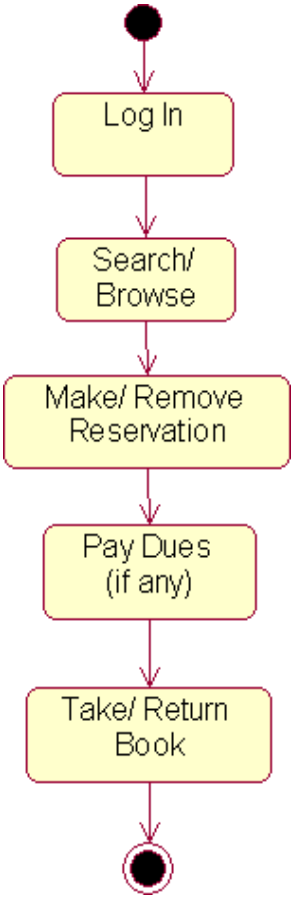
a) Start state

b) End state

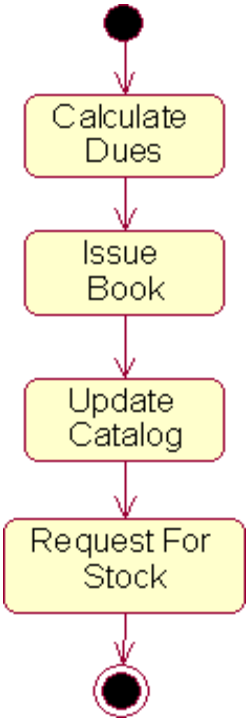
c) State

d) Transition

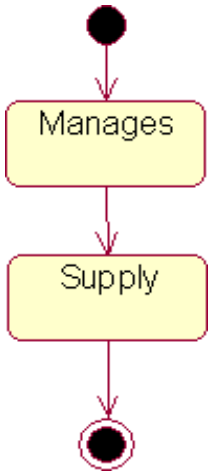
Member:



Librarian:



Library Administrator:



ATM SYSTEM

Introduction:

ATM system needs enhancement to record card information electronically is automatically displays the details in the card. The ATM will communicate with the bank computer over an appropriate communication link. The ATM will serve one customer at a time. A customer will be required to insert an ATM card and enter a PIN number both of which will be sent to bank for valuation as a part of each transaction. The customer will then be able to perform one or more transactions. The card will be retained in the machine until the customer indicates that he/she desires no further transactions, at which point it will be returned.

Objective :

The main objective of the ATM system is to facilitate the user with easy transaction of money at a faster rate. The ATM application will run automatically and there will be no need of any manual interventions. Some built in functions are provided which performs the depository with-drawl functions.

Scope :

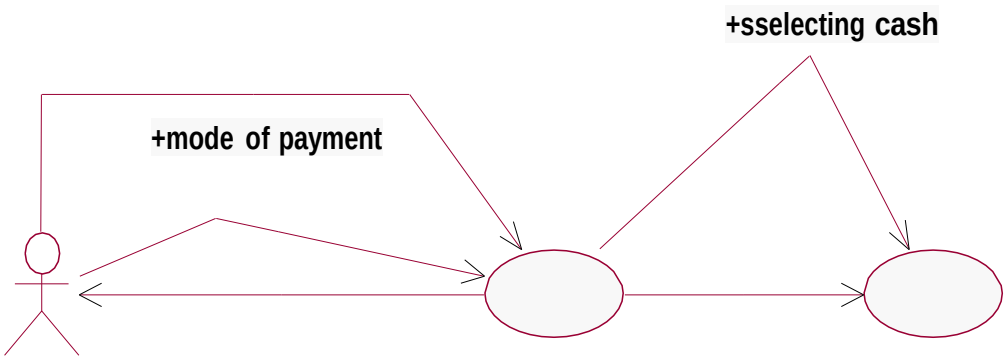
The scope of the project of the ATM system is to understand the working of the system and also to provide more security to the user as it facilitates with easy transaction.

Problem Statement :

The operations on the function of the ATM are as follows:

1. The card has to be inserted in the place of the provided.
2. As the system accepts the card it displays the name and other details of the user.
3. It asks for the password, which is exclusively assigned for the particular card.
4. If the code is correct then the system gets activated with details.

Withdraw usecase diagram :-



TKR COLLEGE OF ENGINEERING AND TECHNOLOGY

AN AUTONOMOUS INSTITUTION

Accredited by NAAC with 'A+' Grade.

(Sponsored by TKR Educational Society, Approved by AICTE, Affiliated to JNTU H)

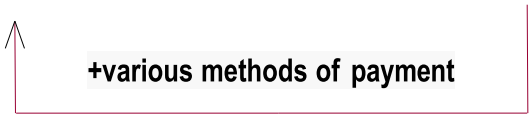
Medbowli, Meerpet, Balapur, Hyderabad, Telangana – 500 097

Phone: 9100377790, email: info@tkrcet.ac.in, web site: www.tkrcet.ac.in

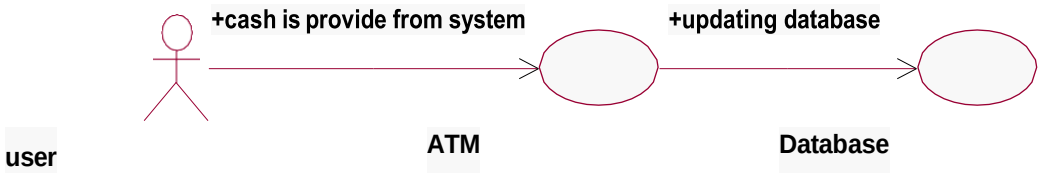
+Enter Amount

+selecting

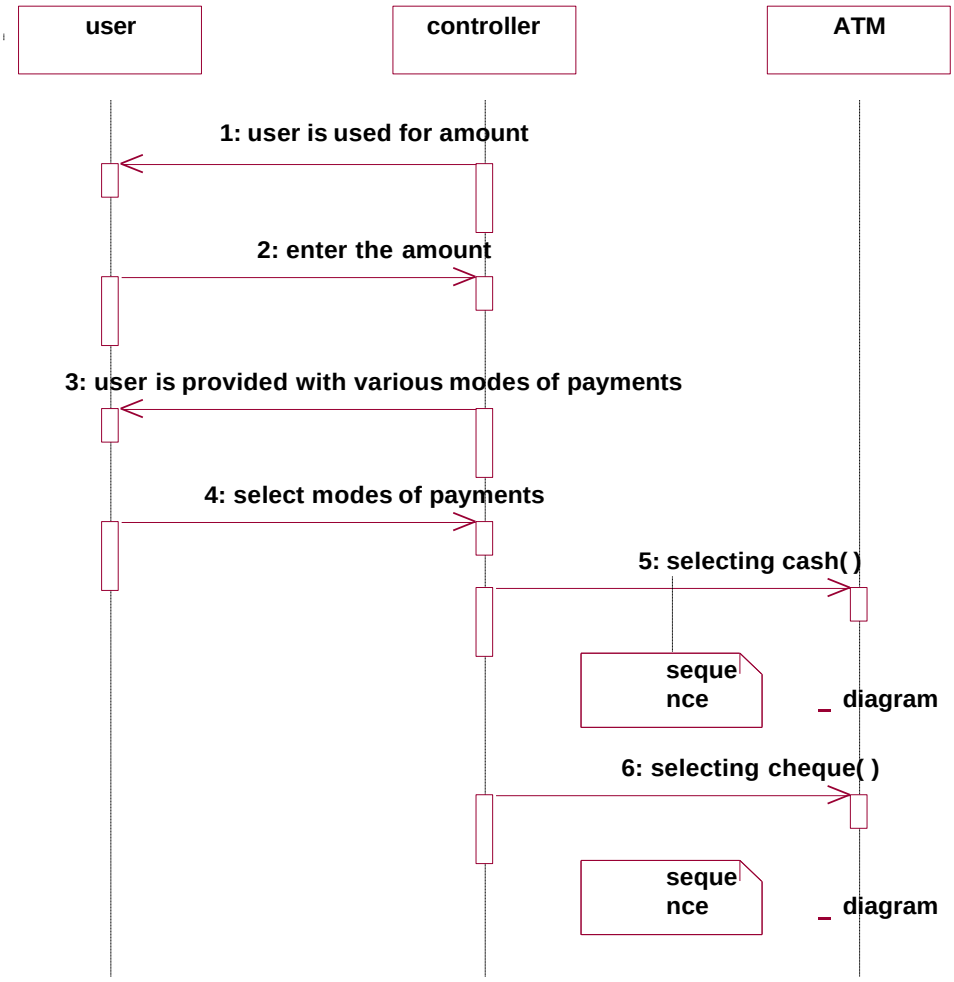
+User asked for amount


User
Controller
ATM
+various methods of payment

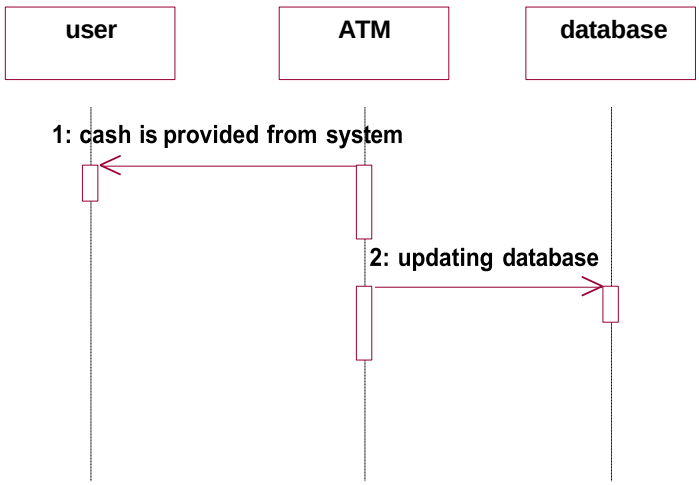
Withdraw cash usecase diagram :-



Withdraw sequence diagram :-



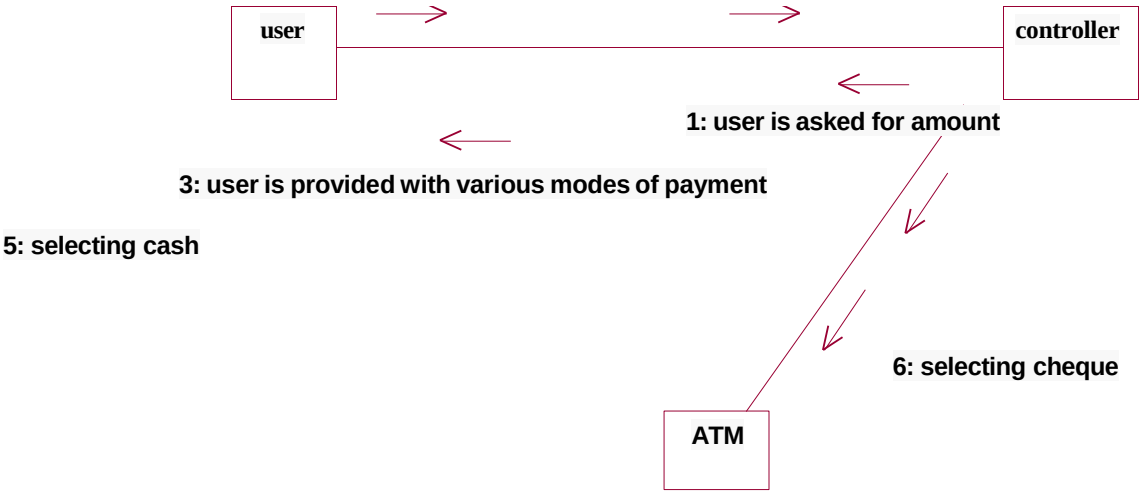
Withdraw cash sequence diagram :-



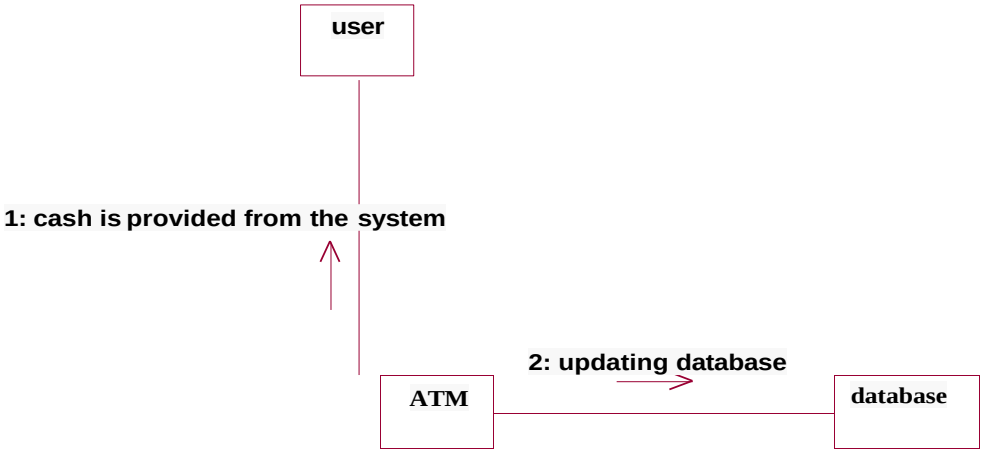
Withdraw collaboration:-

2: enter the amount

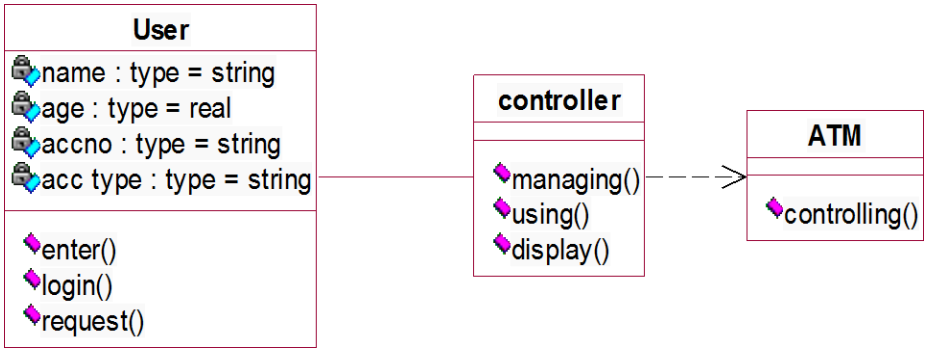
4: select mode of payment



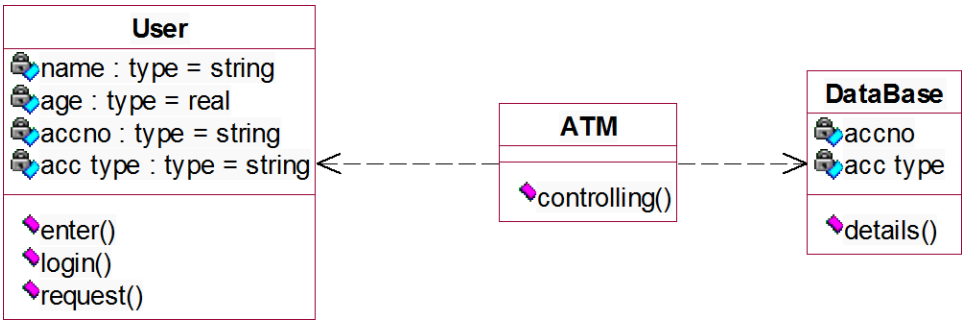
Withdraw cash collaboration :-



Withdraw class diagram :-



Withdraw cash class diagram:-



State Transition diagram :

TKR COLLEGE OF ENGINEERING AND TECHNOLOGY

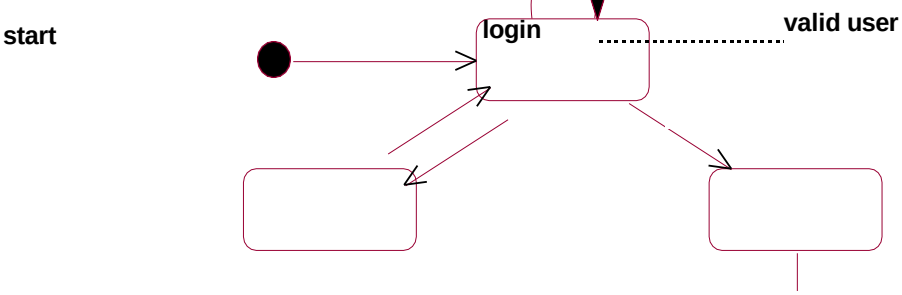
AN AUTONOMOUS INSTITUTION

Accredited by NAAC with 'A+' Grade.

(Sponsored by TKR Educational Society, Approved by AICTE, Affiliated to JNTU H)

Medbowli, Meerpet, Balapur, Hyderabad, Telangana – 500 097

Phone: 9100377790, email: info@tkrcet.ac.in, web site: www.tkrcet.ac.in



TKR COLLEGE OF ENGINEERING AND TECHNOLOGY

AN AUTONOMOUS INSTITUTION

Accredited by NAAC with 'A^+' Grade.

(Sponsored by TKR Educational Society, Approved by AICTE, Affiliated to JNTU H)

Medbowli, Meerpet, Balapur, Hyderabad, Telangana – 500 097

Phone: 9100377790, email: info@tkrcet.ac.in, web site: www.tkrcet.ac.in

TKR COLLEGE OF ENGINEERING AND TECHNOLOGY
AN AUTONOMOUS INSTITUTION
Accredited by NAAC with 'A+' Grade.
(Sponsored by TKR Educational Society, Approved by AICTE, Affiliated to JNTU H)
Medbowli, Meerpet, Balapur, Hyderabad, Telangana – 500 097
Phone: 9100377790, email: info@tkrcet.ac.in, web site: www.tkrcet.ac.in

logged on

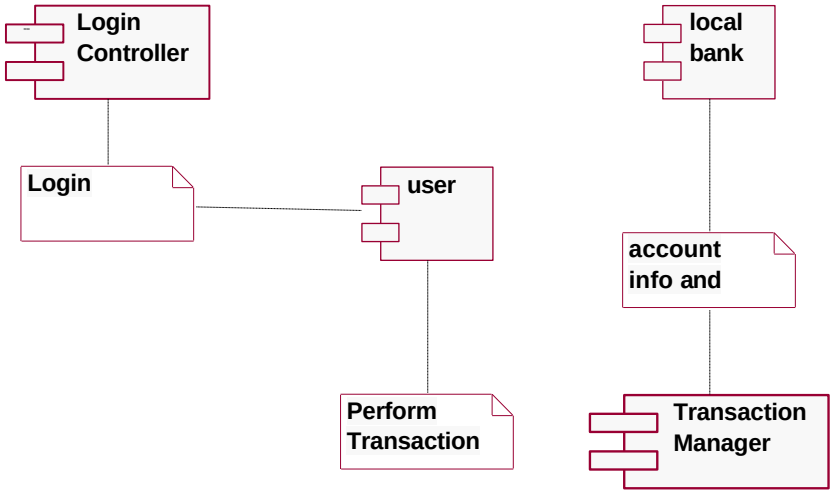
invalid login

transition

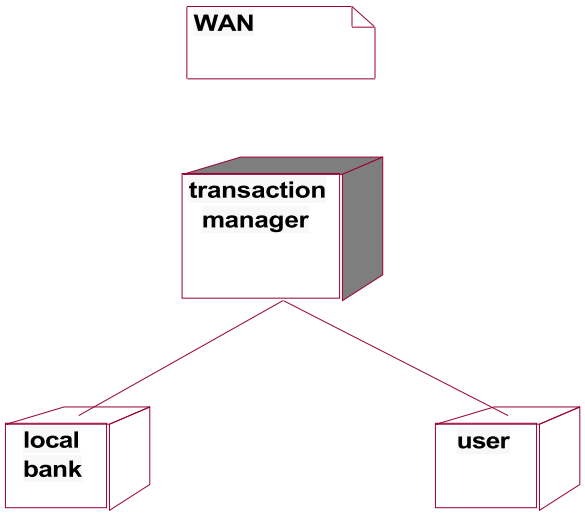
relogin



Component Diagram :-



Deployment Diagram :-



18. References/ E-Links:

Reference Books :

1. Meilir Page-Jones: Fundamentals of Object Oriented Design in UML, Pearson Education.
2. Pascal Roques: Modeling Software Systems Using UML2, WILEY-Dreamtech India Pvt. Ltd.
3. Atul Kahate: Object Oriented Analysis & Design, The McGraw-Hill Companies.
4. Mark Priestley: Practical Object-Oriented Design with UML,TATA McGrawHill
5.Appling UML and Patterns: An introduction to Object - Oriented Analysis and Design and Unified Process, Craig Larman, Pearson Education.

E-Links

1. http://www.visualbuilder.com/uml/tutorial/
2. http://www.exforsys.com/tutorials/ooad.html
3. www.uml-diagrams.org/use-case-diagrams-examples.html
4. www.uml.org/
5. http://www.tutorialspoint.com/uml/
6. www.awl.com/cseng/otseries
7. http://meusite.mackenzie.com.br/rogerio/the-unified-modeling-language-user-guide.9780201571684.997.pdf