RGM COLLEGE OF ENGINEERING AND TECHNOLOGY

OBJECT ORIENTED ANALYSIS AND DESIGN

Department of CSE

(A0526156) OBJECT ORIENTED ANALYSIS AND DESIGN

UNIT – I

Introduction to UML: Importance of modeling, principles of modeling, object oriented modeling, Conceptual model of the UML, Architecture, and Software Development Life Cycle.

UNIT - II

Basic Structural Modeling: Classes, Relationships, common Mechanisms, and diagrams.

Advanced Structural Modeling: Advanced classes, advanced relationships, Interfaces, Types and Roles, Packages.

UNIT - III

Class & Object Diagrams: Terms, concepts, modeling techniques for Class & Object Diagrams.

Basic Behavioral Modeling-: Interactions, Interaction diagrams. Use cases,
Use case Diagrams, Activity Diagrams.

UNIT - IV

Advanced Behavioral Modeling: Events and signals, state machines, processes and Threads, time and space, state chart diagrams.

UNIT-V

Architectural Modeling: Component, Deployment, Component diagrams and Deployment diagrams.

UNIT - VI

Case Study: The Unified Library application, ATM application.

TEXT BOOKS:

1. Grady Booch, James Rumbaugh, Ivar Jacobson: The Unified Modeling Language User Guide, Pearson Education.

OBJECT ORIENTED ANALYSIS AND DESIGN

UNIT – I

Introduction to UML: Importance of modeling, principles of modeling, object oriented modeling, Conceptual model of the UML, Architecture, and Software Development Life Cycle.

UNIT – I

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.

Importance of modeling

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.

Aims of modeling

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

Conceptual model of UML

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

- **Basic building blocks of the UML**
- **Rules**
- **Common Mechanisms in the UML**

Basic building blocks of the UML: Vocabulary of the UML can be defined

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

- It 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.

.

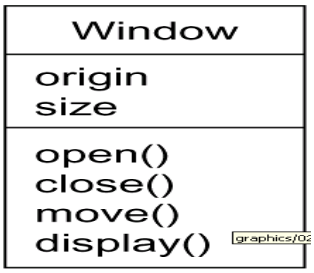


Fig: Class

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.

IUnkonwn

Fig: Interface

Collaboration

- It defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements.
- Therefore, collaborations have structural, as well as behavioral, dimensions.
- A given class might participate in several collaborations.
- Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its name.

Fig: Collaboration

Use case

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

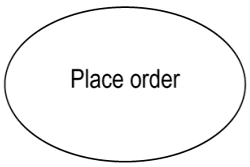


Fig: Use case

Active class

- It 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.

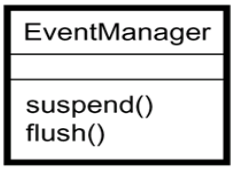


Fig: Active class

Component

- It 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.

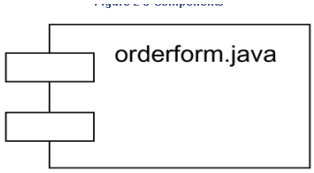


Fig: Component

Node

- It 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.

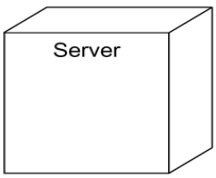


Fig: Node

Behavioral Things

- They 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

Fig: Message

State Machine

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

- State machine involves a number of other elements, including states, transitions, events and activities
- Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates

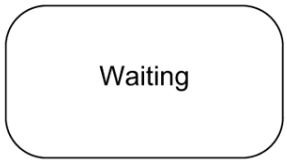


Fig: State machine

Grouping Things

1. These 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.

Fig: Package

Annotational things

- These are the explanatory parts of UML models. These are the comments you may apply to describe about any element in a model.
- There is one primary kind of Annotational thing, namely, note.

Note

- 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.

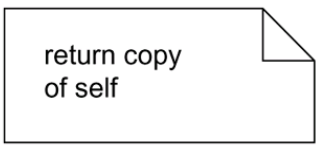


Fig: Note

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.

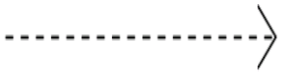


Fig: Dependency

Association

- 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.

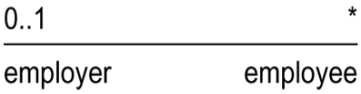


Fig: Association

Generalization

- Generalization 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.

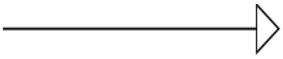


Fig: Generalization

Realization

- 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.

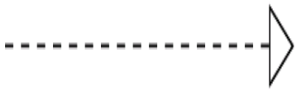


Fig: Realization

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.

Statechart 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

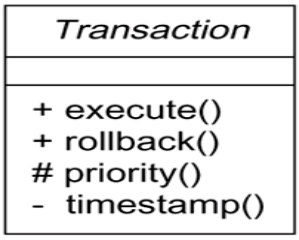
Specification

- They 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.

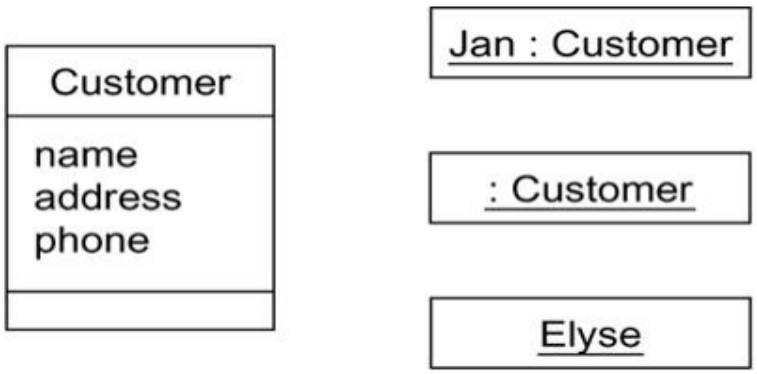


Common Divisions

In modeling object-oriented systems, the world often gets divided in at least a couple of ways.

First, there is the division of class and object. A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, you can model classes as well as objects, as shown in Figure 2-17

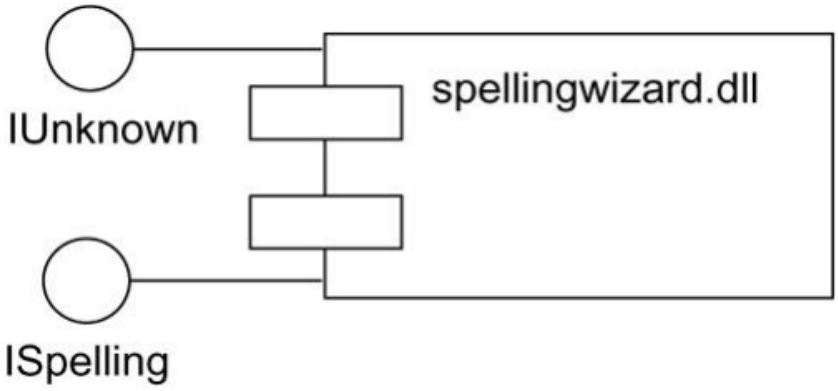
Figure 2-17 Classes And Objects



In this figure, there is one class, named Customer, together with three objects: Jan (which is marked explicitly as being a Customer object), :Customer (an anonymous Customer object), and Elyse (which in its specification is marked as being a kind of Customer object, although it's not shown explicitly here).

Second, there is the separation of interface and implementation. An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics. In the UML, you can model both interfaces and their implementations, as shown in Figure 2-18.

Figure 2-18 Interfaces And Implementations



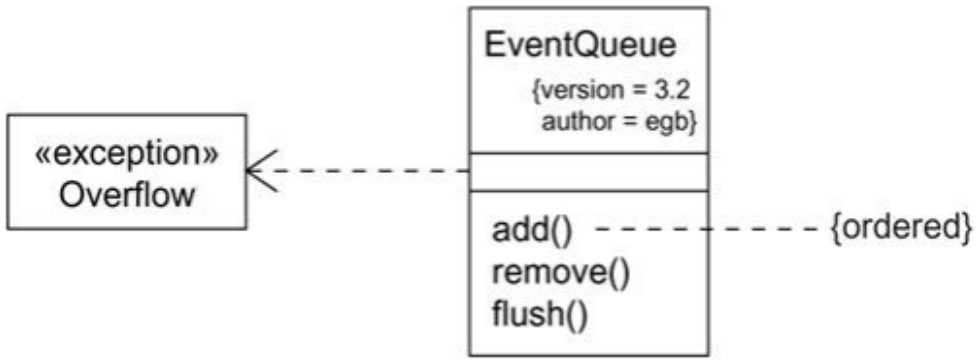
In this figure, there is one component named spellingwizard.dll that implements two interfaces, IUnknown and ISpelling.

Extensibility Mechanisms

The UML's extensibility mechanisms include

1. Stereotypes
2. Tagged values
3. Constraints

- **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. Examples for these are



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.

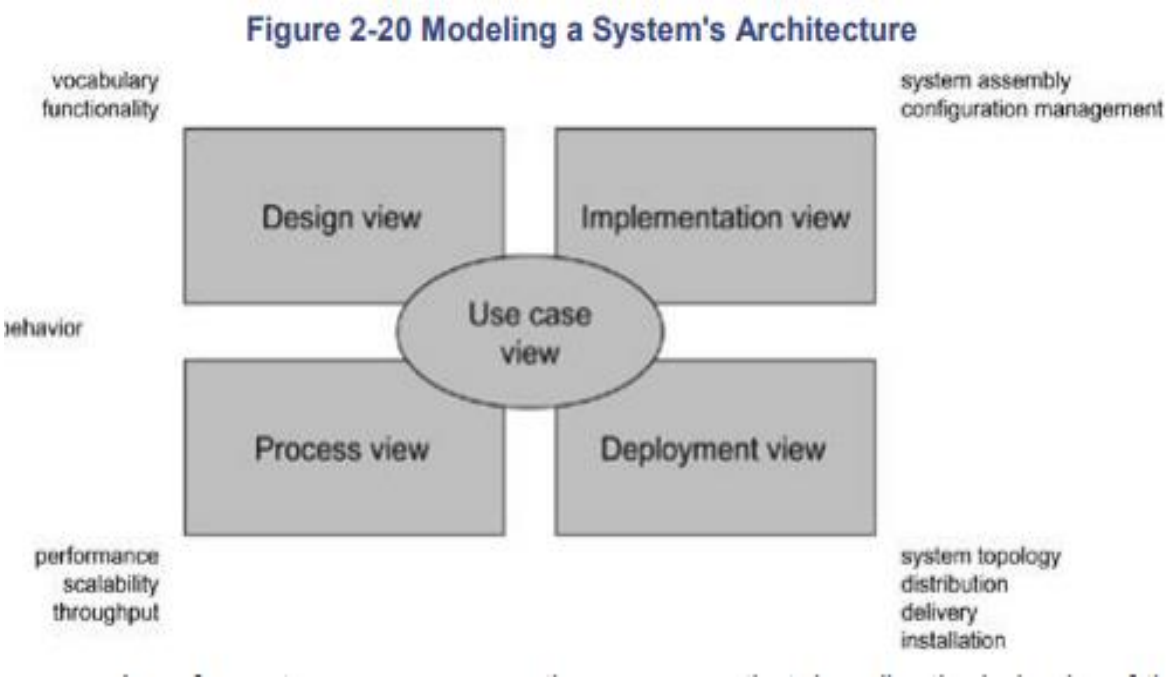


Fig: Modeling a System's Architecture

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 View

- 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.

- With the UML, the static aspects of this view are captured in deployment diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

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

- These five views also interact with one another• nodes in the deployment view hold components in the implementation view that, in turn, represent the physical realization of classes, interfaces, collaborations, and active classes from the design and process views.

- The UML permits you to express every one of these five views and their interactions.

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

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

Architecture-centric

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

An iterative process

- It is one that involves managing a stream of executable releases.

An incremental process

- It is one that involves the continuous integration of the system's architecture to produce these releases, with each new release embodying incremental improvements over the other.

- Together, an iterative and incremental process is *risk-driven,* meaning that each new release is focused on attacking and reducing the most significant risks to the success of the project. This use case driven, architecture-centric, and iterative/incremental process can be broken into phases.

- **A *phase*** is the span of time between two major milestones of the process, when a well-defined set of objectives are met, artifacts are completed, and decisions are made whether to move into the next phase.

- As below figure shows, 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.

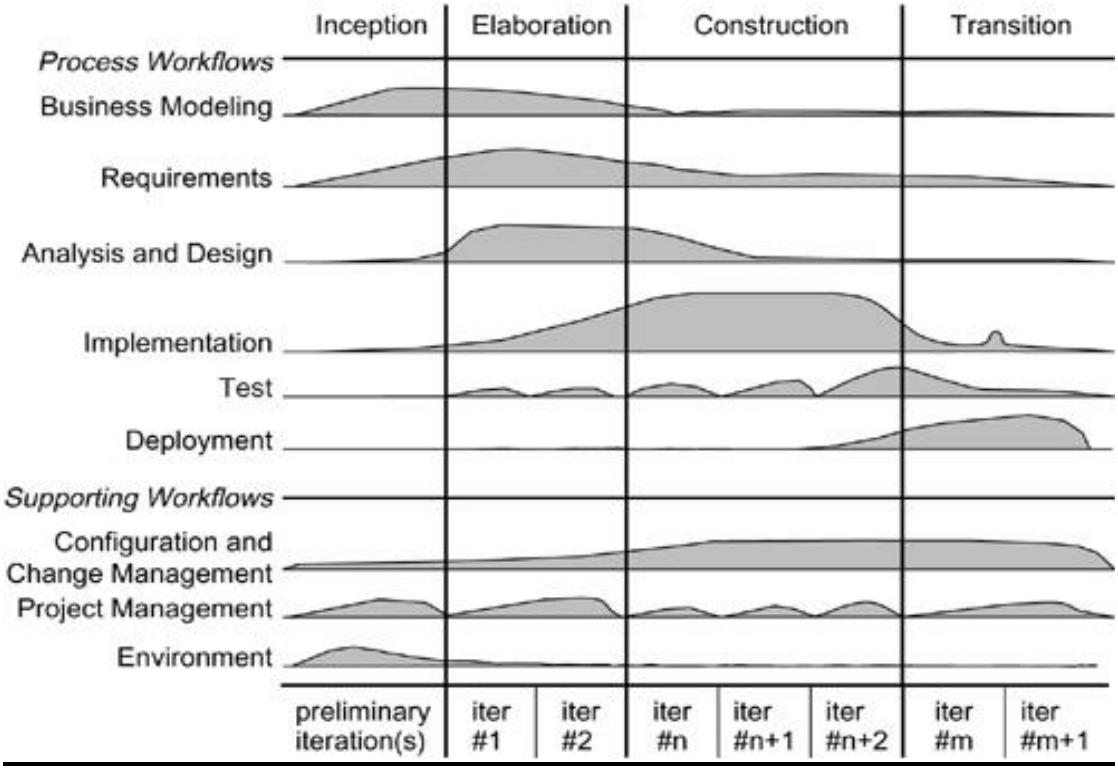


Fig: Figure Software Development Life Cycle

Inception phase:

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

Elaboration phase:

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

Construction phase:

- It **i**s the third phase of the process.
- When the software is brought from an executable architectural baseline to being ready to be transitioned to the user community.
- 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 phase:

- It is the fourth phase of the process.
- When the software is turned into the hands of the user community.

- 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.

- One element that distinguishes this process and that cuts across all four phases is an iteration.
- An iteration is a distinct set of activities, with a baselined plan and evaluation criteria that result in a release, either internal or external.
- This means that the software development life cycle can be characterized as involving a continuous stream of executable releases of the system's architecture.
- It is this emphasis on architecture as an important artifact that drives the UML to focus on modeling the different views of a system's architecture.

UNIT - II

Basic Structural Modeling: Classes, Relationships, Common Mechanisms, and diagrams.

Advanced Structural Modeling: Advanced classes, advanced relationships, Interfaces, Types and Roles, Packages.

UNIT – II

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

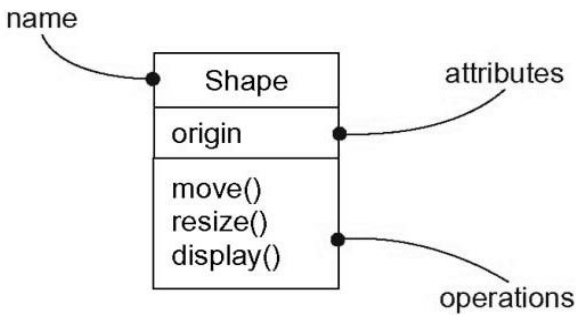


Fig: Graphical Representation of Class in UML

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.

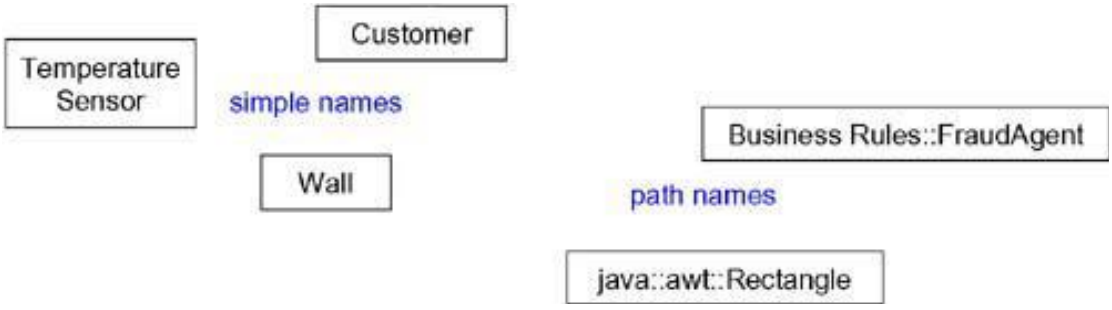


Fig: Simple and Path Names

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.

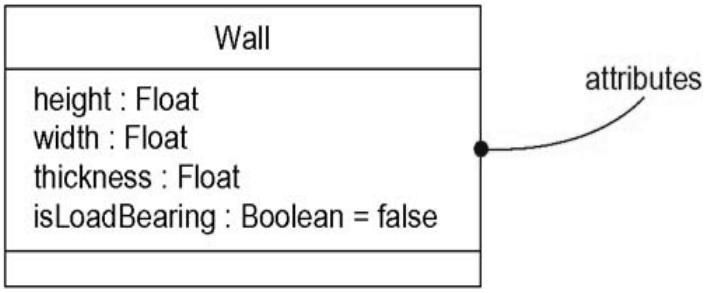


Fig: Attributes and Their Class

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.

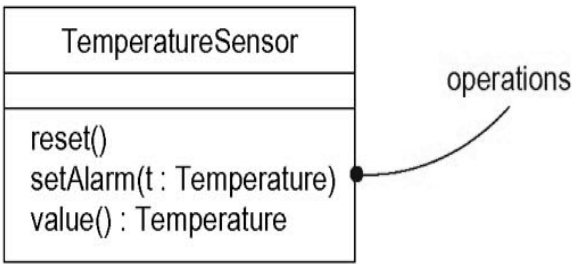
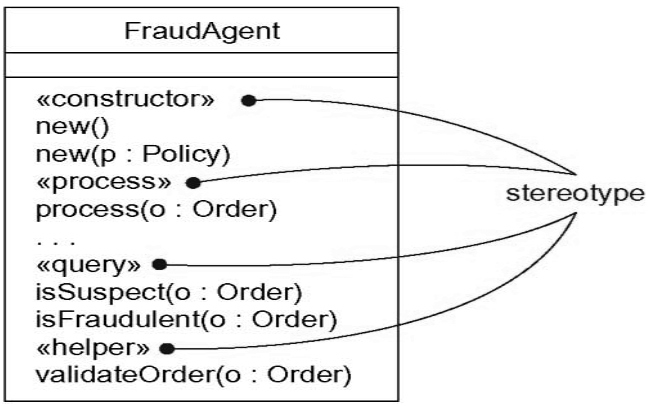


Fig: Operations

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

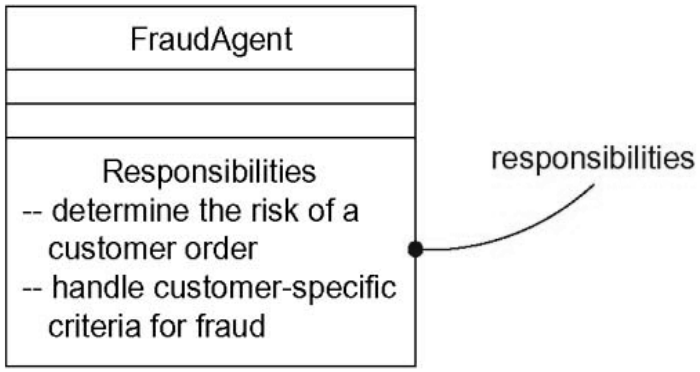


Fig: Responsibilities

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.

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

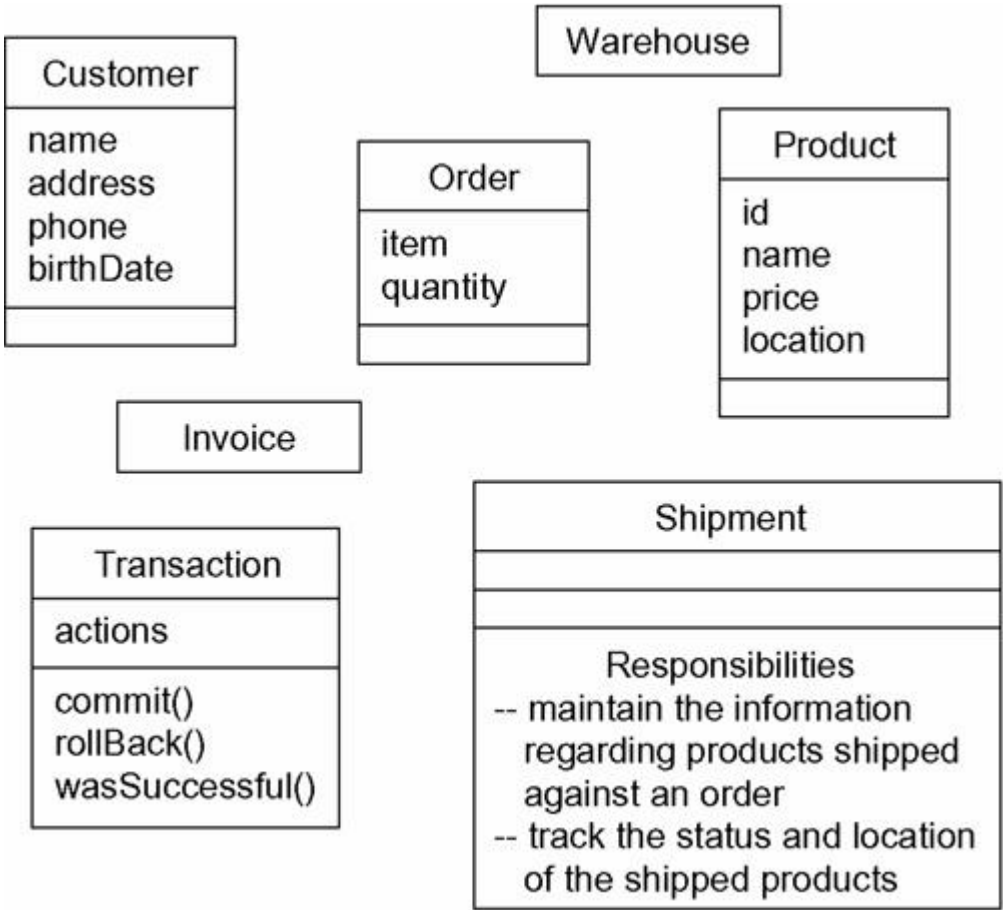


Fig: Modeling the Vocabulary of a System

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.

For example, figure shows a set of classes drawn from Smalltalk, showing the distribution of responsibilities among Model, View, and Controller classes. Notice how all these classes work together such that no one class does too much or too little.

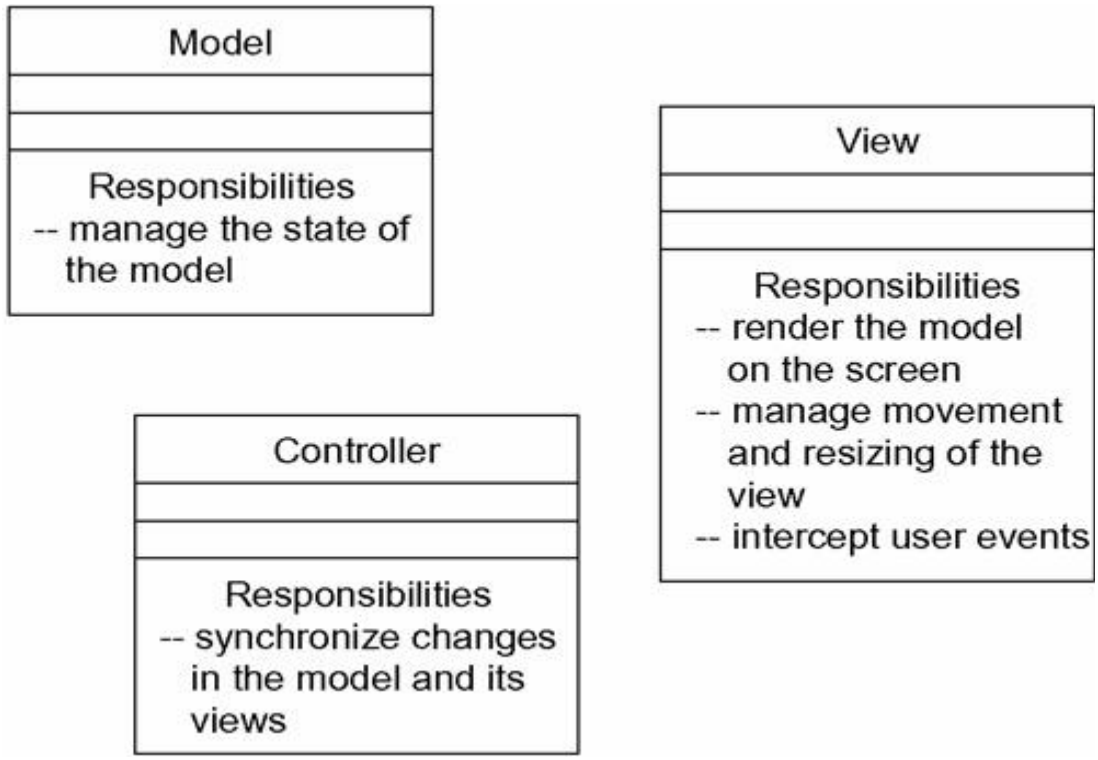


Fig: Modeling the Distribution of Responsibilities in a System

Modeling Nonsoftware 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 nonsoftware 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.

Figure below shows, it's perfectly normal to abstract humans (like AccountsReceivableAgent) and hardware (like Robot) as classes, because each represents a set of objects with a common structure and a common behavior.

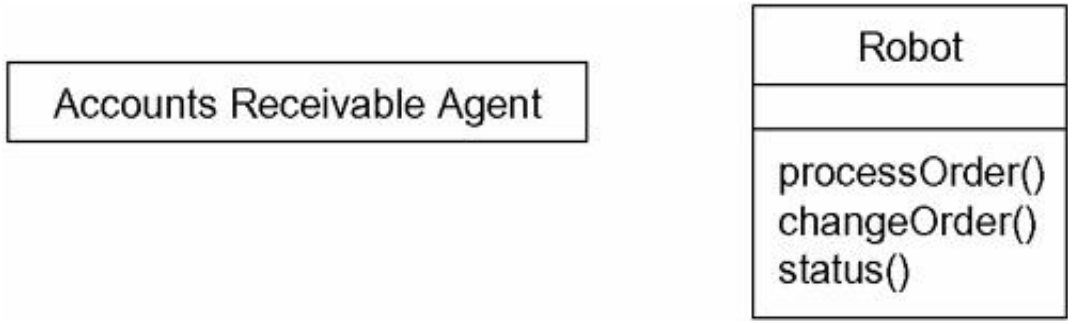


Fig: Modeling Nonsoftware Things

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.

Figure shows, these things can be modeled in the UML as types or enumerations, which are rendered just like classes but are explicitly marked via stereotypes. Things like integers (represented by the class Int) are modeled as types, and you can explicitly indicate the range of values these things can take on by using a constraint. Similarly, enumeration types, such as Boolean and Status, can be modeled as enumerations, with their individual values provided as attributes.

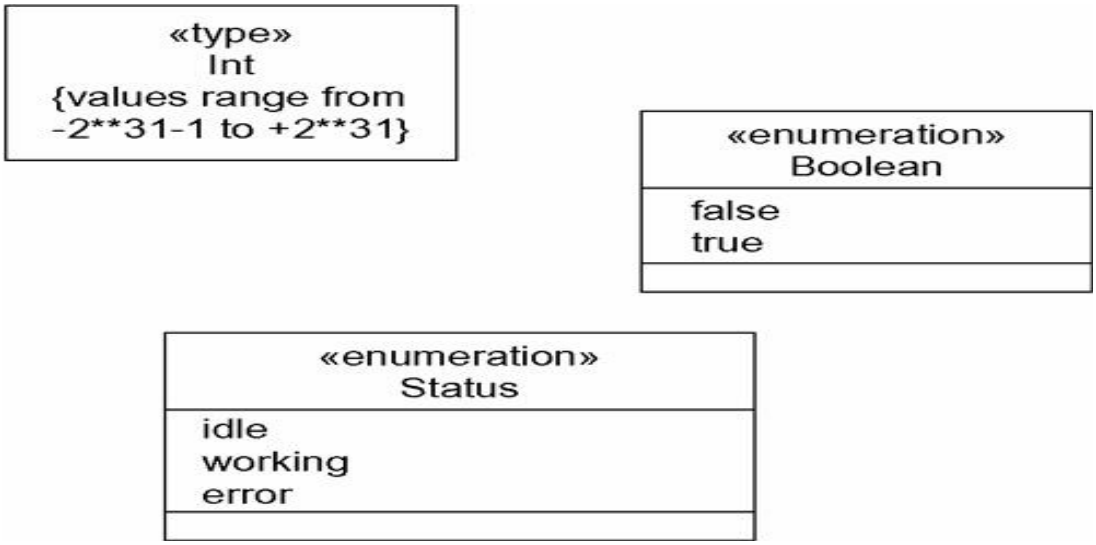


Fig: Modeling Primitive Types

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

Dependencies

- Dependencies are using relationships. For example, pipes depend on the water heater to heat the water they carry..

Generalizations

- Generalizations connect generalized classes to more-specialized ones in what is known as subclass/superclass or child/parent relationships.
- For example, a bay window is a kind of window with large, fixed panes; a patio window is a kind of window with panes that open side to side.

Associations

- Associations are structural relationships among instances.
- For example, rooms consist of walls and other things; walls themselves may have embedded doors and windows; pipes may pass through walls..

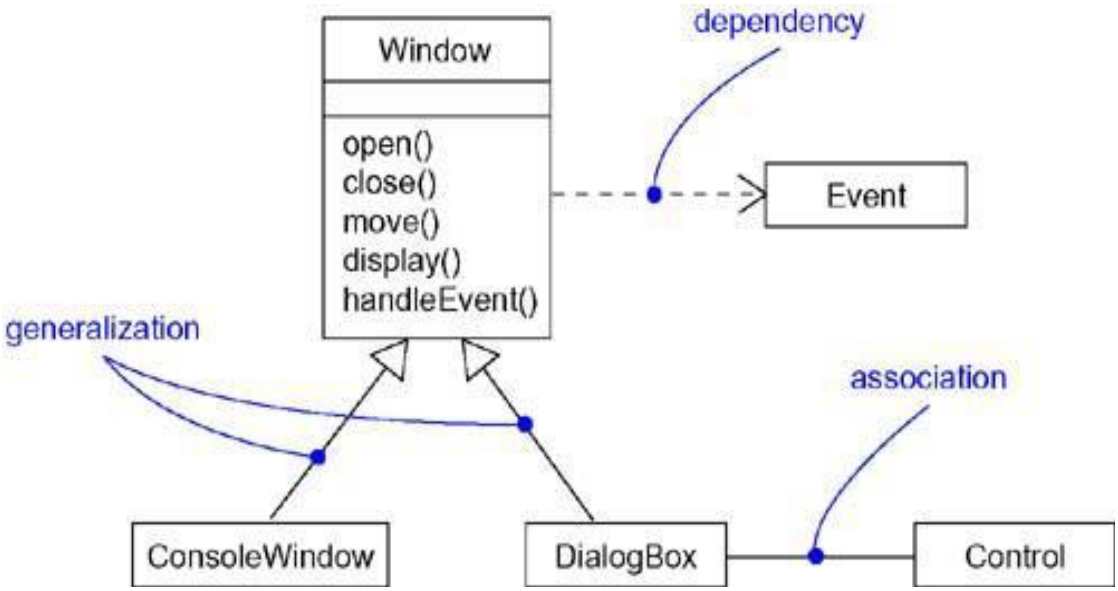


Fig: Relationships

Terms and Concepts

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

Dependency

- A dependency is a using relationship that states that a change in specification of one thing 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

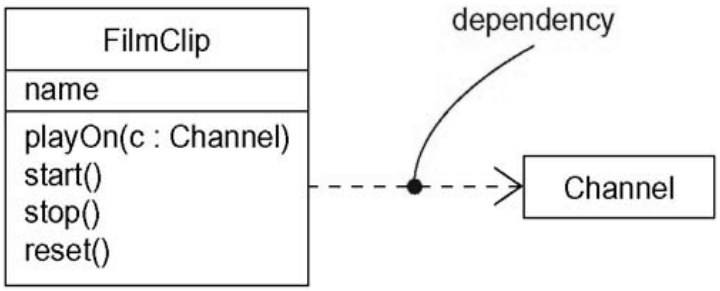


Fig: Dependencies

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

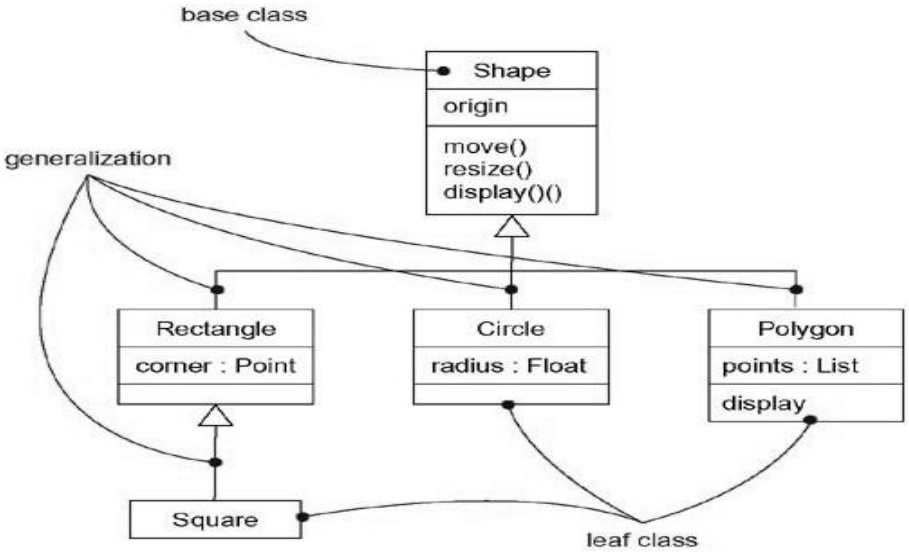


Fig: Generalization

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

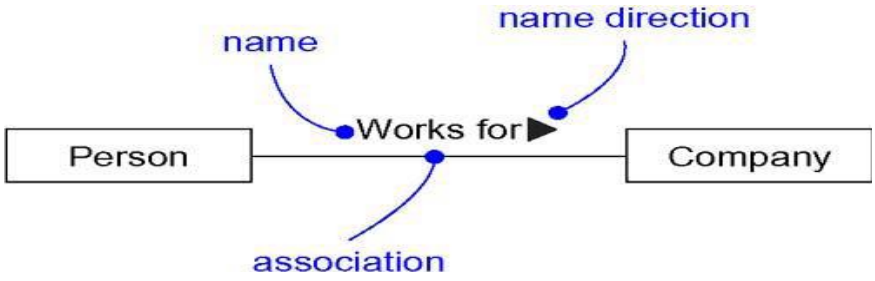


Fig: Association Names

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

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

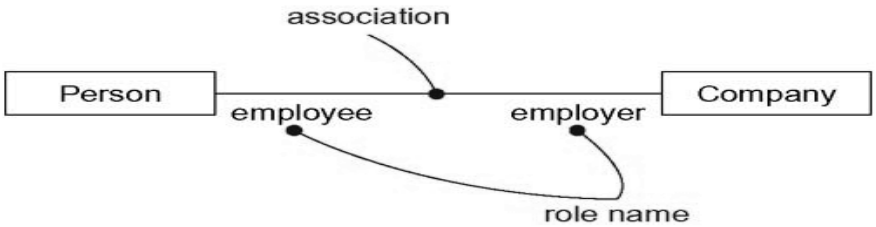


Fig: Role Names

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).

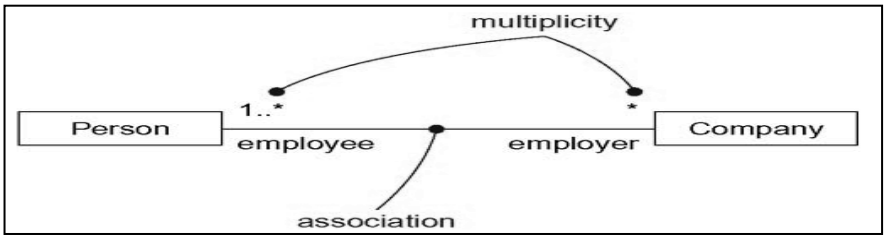


Fig: Multiplicity

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

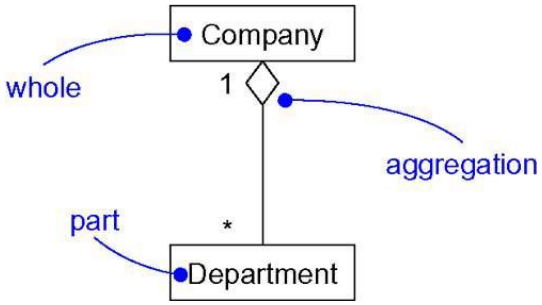


Fig: Aggregation

Common Modeling Techniques

Modeling Simple Dependencies

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

To model simple dependencies

- 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 CourseSchedule to Course, because Course is used in both the add and remove operations of CourseSchedule.
- The dependency from Iterator shows that the Iterator uses the CourseSchedule; the CourseSchedule 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++.

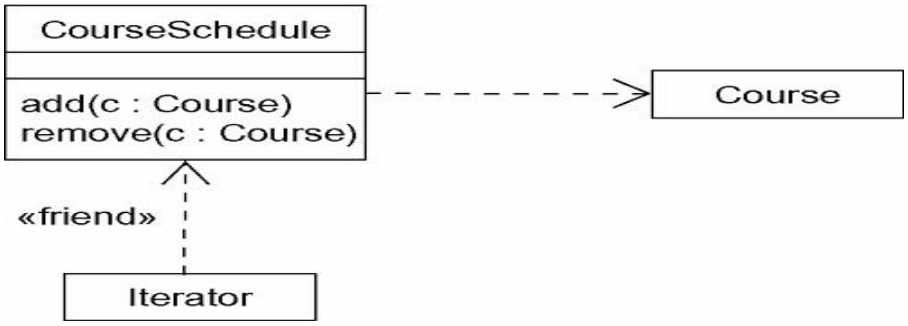


Fig: Modeling simple dependency relationships

Modeling Single Inheritance

To model single 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.

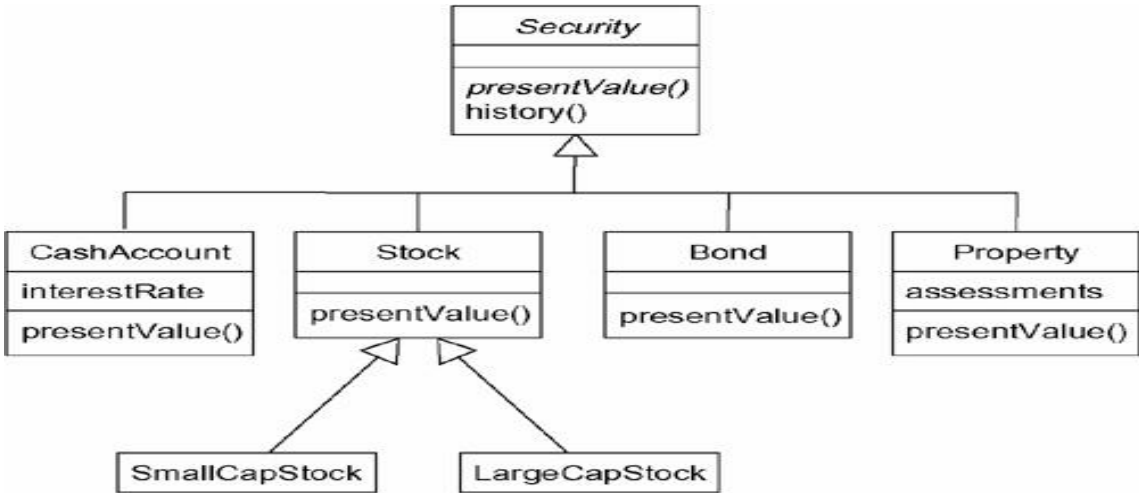


Fig: Inheritance Relationships

Modeling Structural Relationships

- When you model with dependencies or generalization relationships, 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

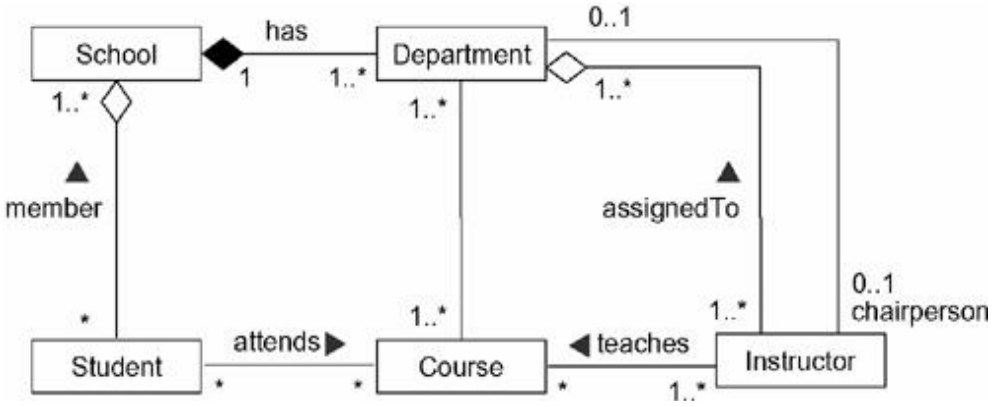


Fig: Structural Relationships

Common Mechanisms

- The UML is made simpler by the presence of four common mechanisms that apply consistently throughout the language: specifications, adornments, common divisions, and extensibility mechanisms.
- This chapter explains the use of two of these common mechanisms, adornments and extensibility mechanisms.
- Notes are the most important kind of adornment that stands alone.
- A note is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements.
- You use notes to attach information to a model, such as requirements, observations, reviews, and explanations.
- The UML's extensibility mechanisms permit you to extend the language in controlled ways.
- These mechanisms include stereotypes, tagged values, and constraints.

- A 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. You use these mechanisms to tailor the UML to the specific needs of your domain and your development culture.

Terms and Concepts

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

- A **stereotype** is an extension of the vocabulary of the UML, allowing 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. As an option, the stereotyped element may be rendered by using a new icon associated with that stereotype.

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

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

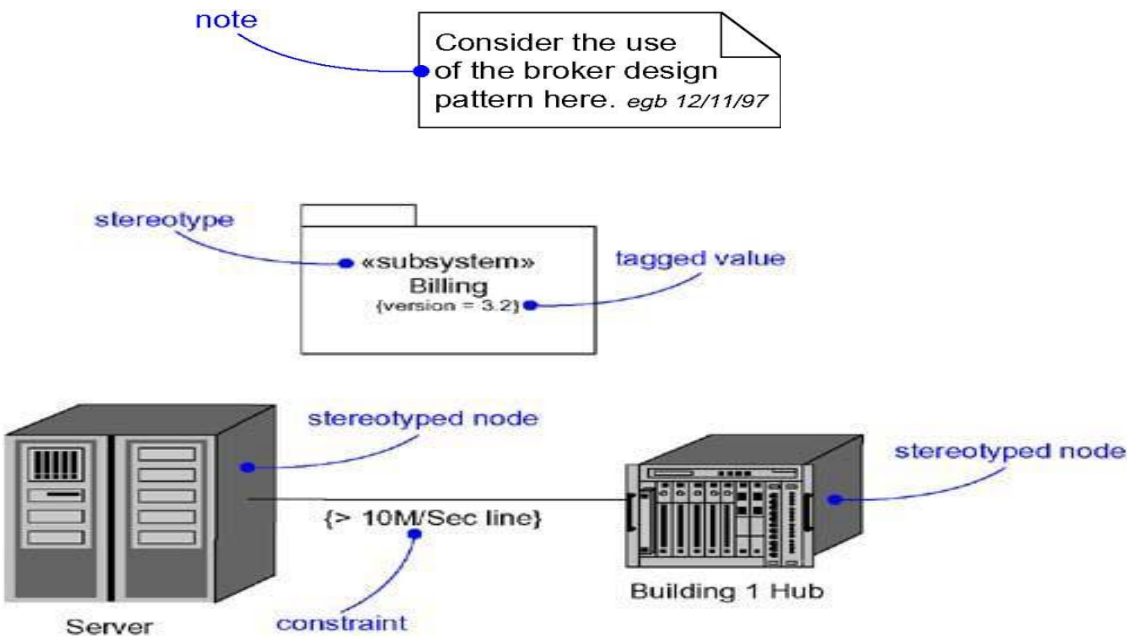


Fig: Note, Stereotypes, Tagged Values, and Constraints

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. If your implementation allows it,
- You can put a live URL inside a note, or even link to or embed another ocument. In this way, you can use the UML to organize all the artifacts you might generate or use during development.

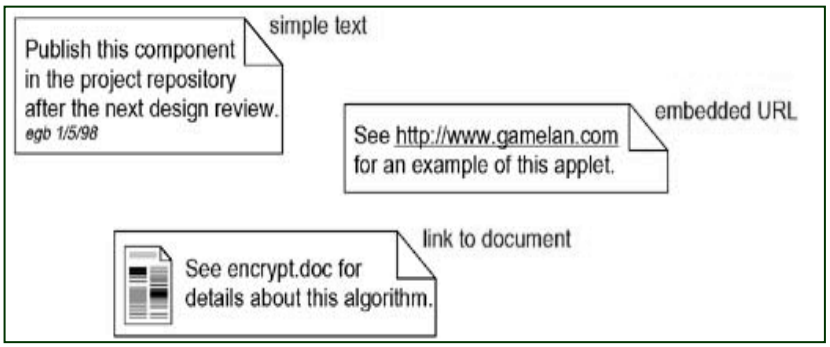


Fig: Notes

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.

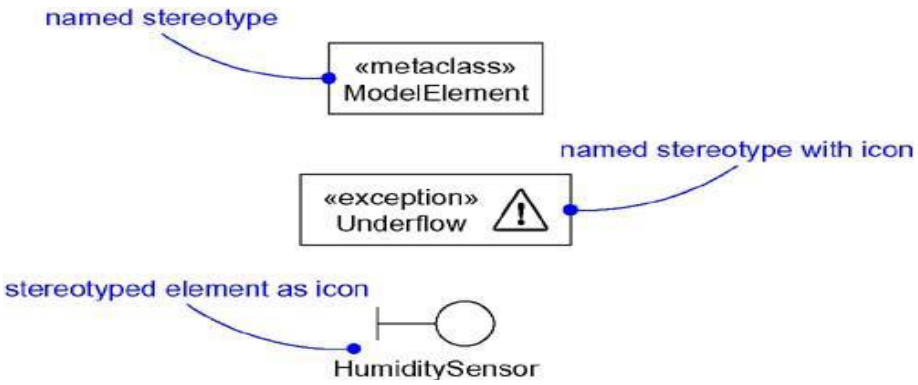


Fig: Stereotypes

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).

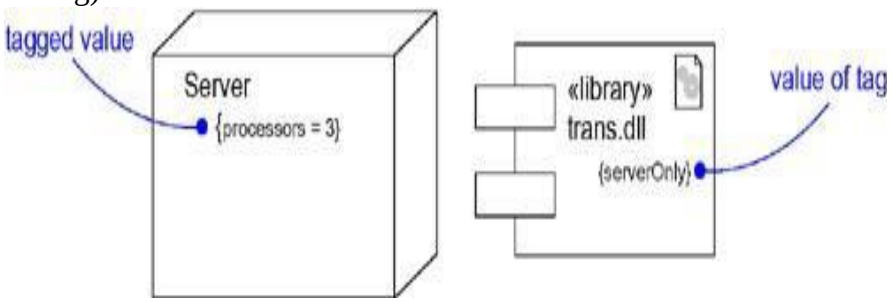


Fig: Tagged Values

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.

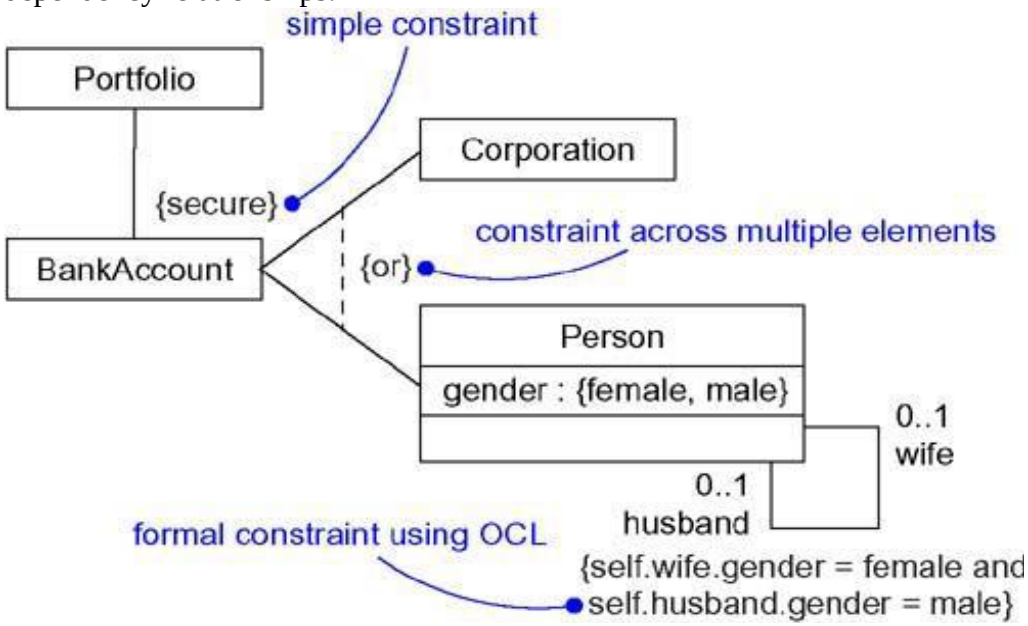


Fig: Constraints

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
- As your model evolves, keep those comments that record significant decisions that cannot be inferred from the model itself, and• unless they are of historic interest discard the others.

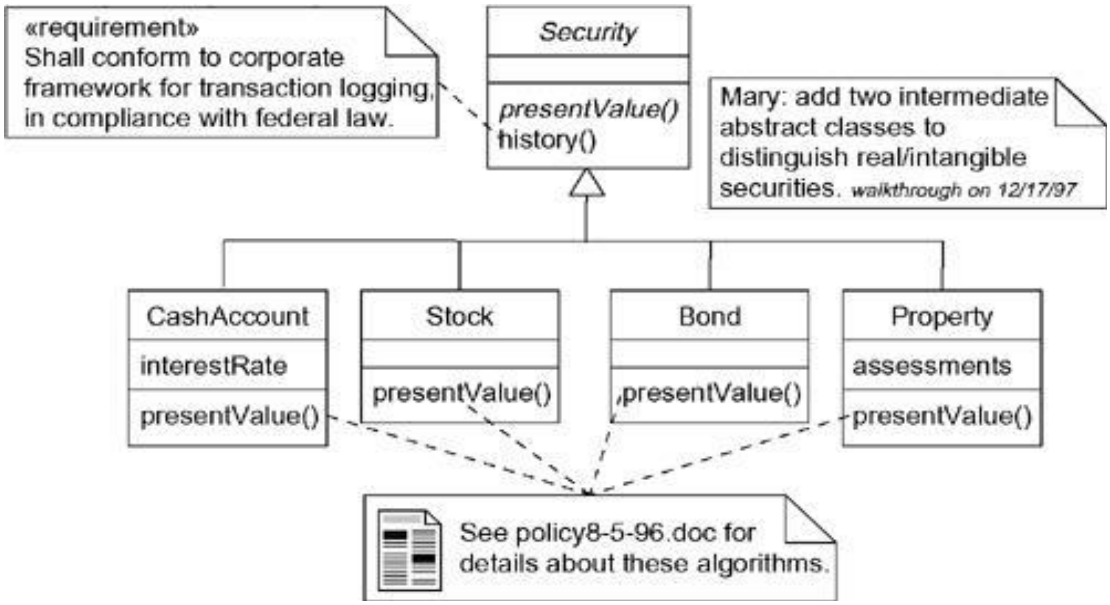


Fig: Modeling Comments

- For example, the above figure shows a model that's a work in progress of a class hierarchy, showing some requirements that shape the model, as well as some notes from a design review.
- In this example, most of the comments are simple text (such as the note to Mary), but one of them (the note at the bottom of the diagram) provides a hyperlink to another document

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

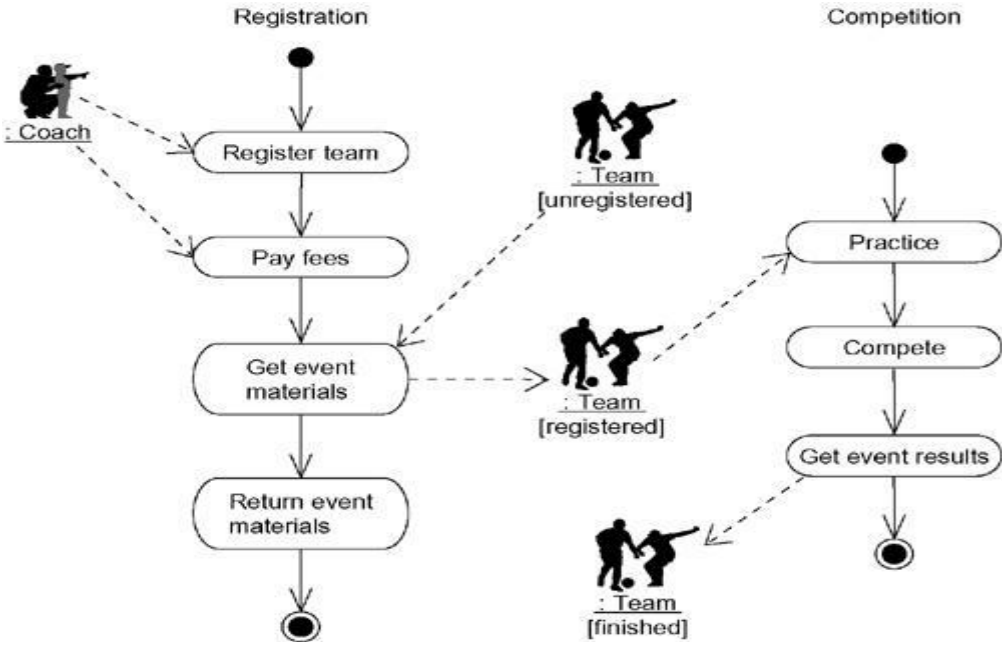


Fig: Modeling New Building Blocks.

- The above Figure shows, there are two things that stand out-Coach objects and Team objects. These are not just plain kinds of classes. Rather, they are now primitive building blocks that you can use in this context.
- You can create these new building blocks by defining a coach and team stereotype and applying them to UML's classes. In this figure, the anonymous instances called: Coach and: Team (the latter shown in various states namely, unregistered, registered, and finished) appear using the icons associated with these stereotypes.

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.

- Figure shows four subsystems, each of which has been extended to include its version number and status. In the case of the Billing subsystem, one other tagged value is shown the person who has currently checked out the subsystem.

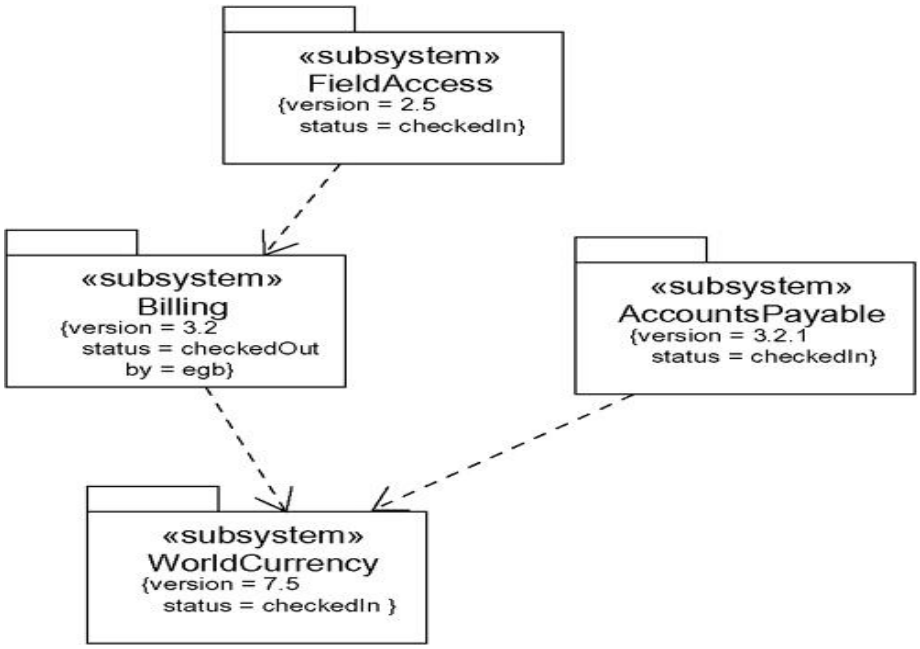


Fig: Modeling New Properties

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.

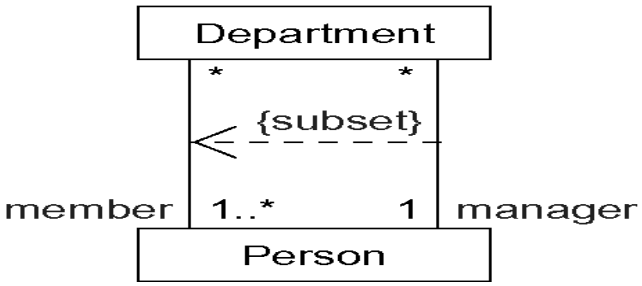


Fig: Modeling New Semantics

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

Diagrams

- 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)
    - To reconstruct models from parts of an executable system (reverse engineering).

Terms and Concepts

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
- Statechart 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
- Statechart diagram : Focused on the changing state of a system driven by Events
- Activity diagram : Focused on the flow of control from activity to Activity

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.

Statechart Diagram

- We use statechart diagrams to illustrate the dynamic view of a system.
- They are especially important in modeling the behavior of an interface, class, or collaboration.
- A statechart diagram shows a state machine, consisting of states, transitions, events, and activities.
- Statechart 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 required
• 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) * Statechart diagrams (for behavioral modeling)
• Process view	:	* Class diagrams (for structural modeling) * Interaction diagrams (for behavioral modeling)
• Implementation view	:	Component diagram
• Deployment view	:	Deployment diagrams

Modeling Different Levels of Abstraction

- Not only do you need to view a system from several angles, you'll also find people 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:
    1. By presenting diagrams with different levels of detail against the same model
    2. 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

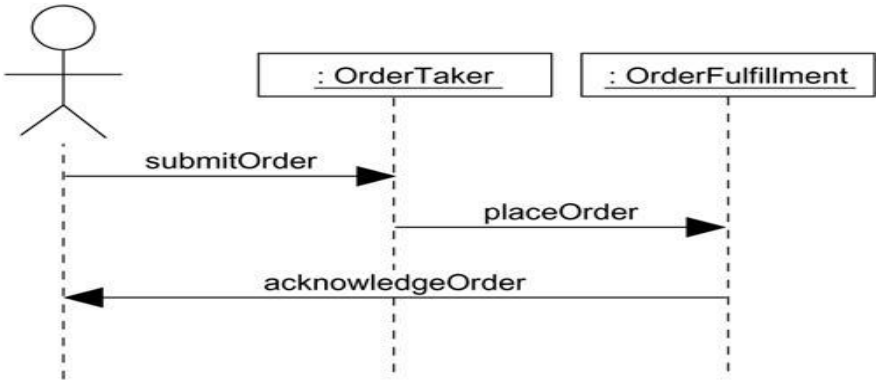


Fig: Interaction Diagram at a High Level of Abstraction

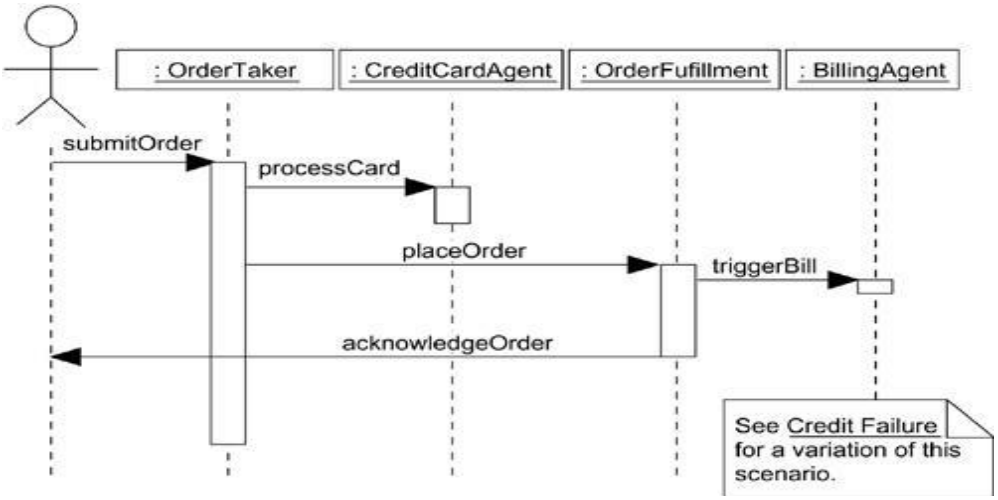


Fig: Interaction at a Low Level of Abstraction

* 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

Advanced Classes

- Classes are indeed the most important building block of any object-oriented system. However, classes are just one kind of an even more general building block in the UML• classifiers.
- A classifier is a mechanism that describes structural and behavioral features. Classifiers include classes, interfaces, datatypes, signals, components, nodes, use cases, and subsystems.

- Classifiers (and especially classes) have a number of advanced features beyond the simpler properties of attributes and operations described in the previous section: You can model multiplicity, visibility, signatures, polymorphism, and other characteristics.

- The UML provides a representation for a number of advanced properties, as Figure shows. This notation permits you to visualize, specify, construct, and document a class to any level of detail you wish, even sufficient to support forward and reverse engineering of models and code.

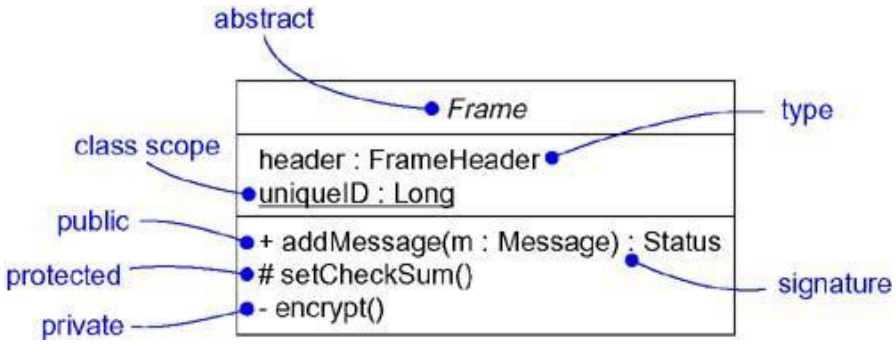


Fig: Advanced Classes

Terms and Concepts

- A classifier is a mechanism that describes structural and behavioral features. Classifiers include classes, interfaces, datatypes, signals, components, nodes, use cases, and subsystems.
- The most important kind of classifier in the UML is the class. A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Classes are not the only kind of classifier, however.

- The UML provides a number of other kinds of classifiers to help you model.

• Interface	A collection of operations that are used to specify a service of a class or a component
• Datatype	A type whose values have no identity, including primitive built-in types (such as numbers and strings), as well as enumeration types (such as Boolean)
• Signal	The specification of an asynchronous stimulus communicated between instances
• Component	A physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces
• Node	A physical element that exists at run time and that represents a computational resource, generally having at least some memory and often processing capability
• Use case	A description of a set of a sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor
• Subsystem	A grouping of elements of which some constitute a specification of the behavior offered by the other contained elements

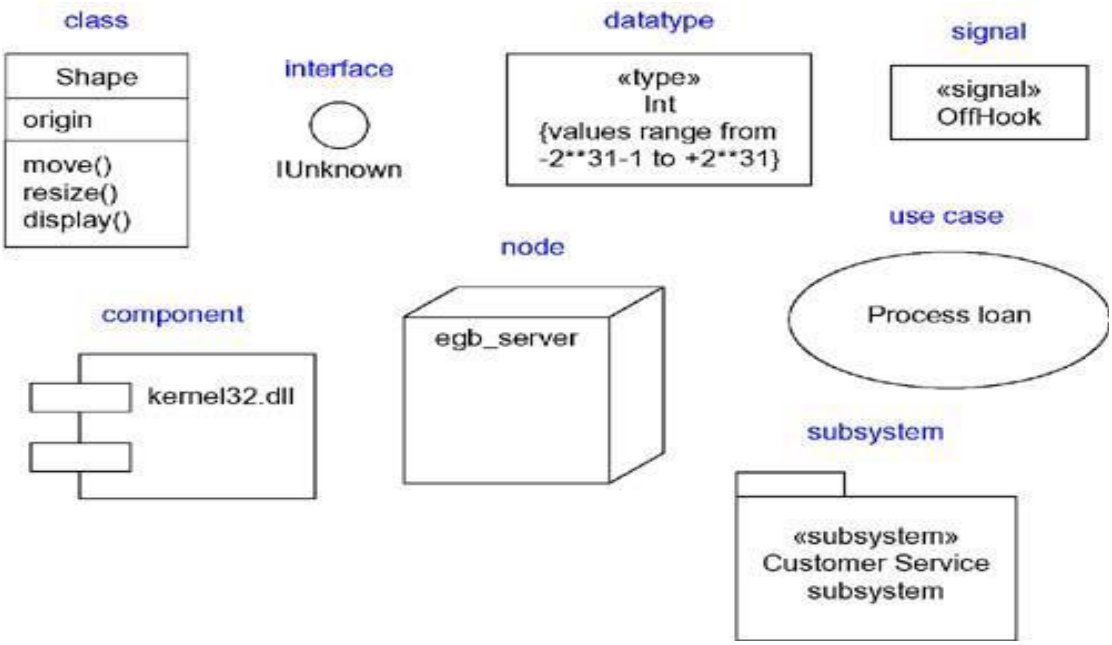


Fig: Classifiers

Visibility

- One of the most important details you can specify for a classifier's attributes and operations is its visibility.
- The visibility of a feature specifies whether it can be used by other classifiers. In the
- UML, you can specify any of three levels of visibility.

1. `public`	Any outside classifier with visibility to the given classifier can use the feature; specified by prepending the symbol +
2. `protected`	Any descendant of the classifier can use the feature; specified by prepending the symbol #
3. `private`	Only the classifier itself can use the feature; specified by prepending the symbol

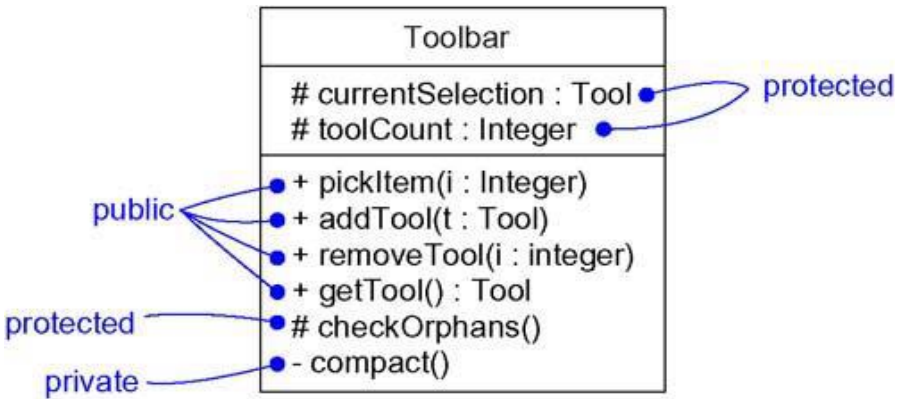


Fig: Visibility

The above Figure shows a mix of public, protected, and private figures for the class Toolbar.

Scope

- Another important detail you can specify for a classifier's attributes and operations is its owner scope.
- The owner scope of a feature specifies whether the feature appears in each instance of the classifier or whether there is just a single instance of the feature for all instances of the classifier.
- In the UML, you can specify two kinds of owner scope.

1. instance	Each instance of the classifier holds its own value for the feature.
2. classifier	There is just one value of the feature for all instances of the classifier.

As Figure (a simplification of the first figure) shows, a feature that is classifier scoped is rendered by underlining the feature's name. No adornment means that the feature is instance scoped.

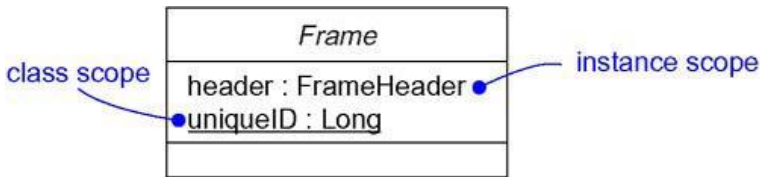


Fig: Owner Scope

Abstract, Root, Leaf, and Polymorphic Elements

- You use generalization relationships to model a lattice of classes, with more-generalized
- Abstractions at the top of the hierarchy and more-specific ones at the bottom.
- Within these hierarchies, it's common to specify that certain classes are abstract• meaning that they may not have any direct instances.
- In the UML, you specify that a class is abstract by writing its name in italics.

- For example, as figure below shows, Icon, RectangularIcon, and ArbitraryIcon are all abstract classes. By contrast, a concrete class (such as Button and OKButton) is one that may have direct instances.

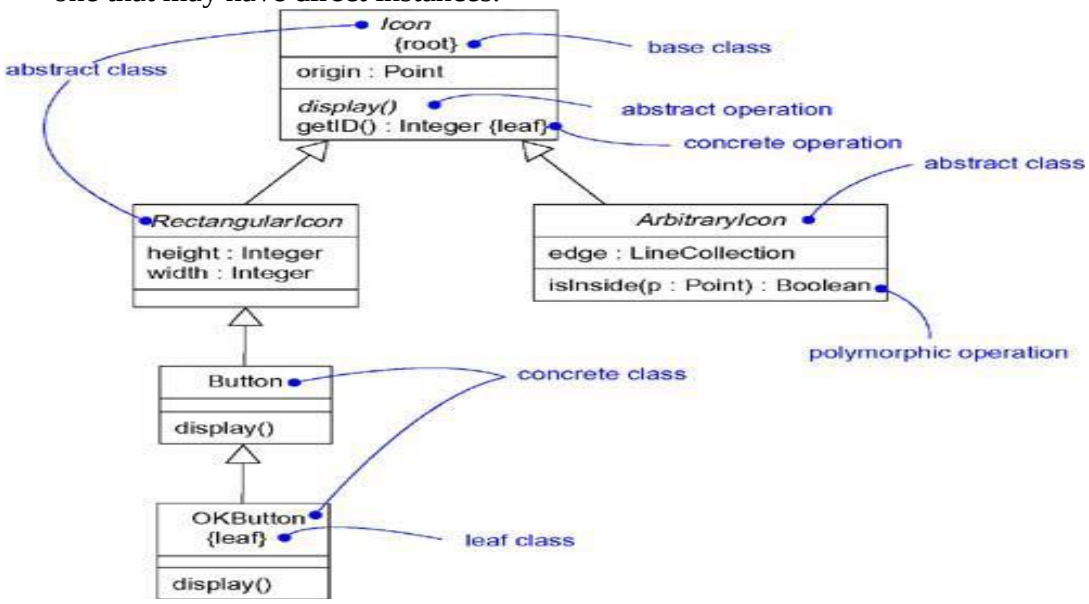


Fig: Abstract and Concrete Classes and Operations

Multiplicity

- The number of instances a class may have is called its multiplicity. Multiplicity is a specification of the range of allowable cardinalities an entity may assume.
- In the UML, you can specify the multiplicity of a class by writing a multiplicity expression in the upper-right corner of the class icon.

- For example, in Figure NetworkController is a singleton class. Similarly, there are exactly three instances of the class ControlRod in the system.

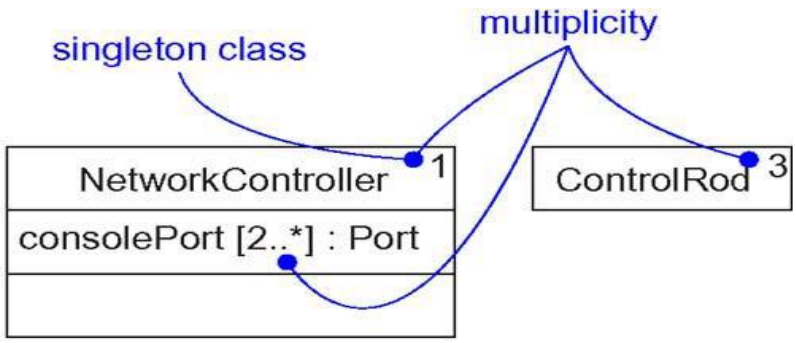


Fig: Multiplicity

Attributes

- At the most abstract level, when you model a class's structural features (that is, its attributes), you simply write each attribute's name.
- That's usually enough information for the average reader to understand the intent of your model.
- You can also specify the visibility, scope, and multiplicity of each attribute. There's still more. You can also specify the type, initial value, and changeability of each attribute.

In its full form, the syntax of an attribute in the UML is

[Visibility] name [multiplicity] [: type] [= initial-value] [{property-string}]

For example, the following are all legal attribute declarations:

• origin	Name only
• + origin	Visibility and name
• origin : Point	Name and type
• head : *Item	Name and complex type
• name [0..1] : String	Name, multiplicity, and type
• origin : Point = (0,0)	Name, type, and initial value
• id : Integer {frozen}	Name and property

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

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

Operations

- At the most abstract level, when you model a class's behavioral features (that is, its operations and its signals), you will simply write each operation's name
- You can also specify the visibility and scope of each operation.
- You can also specify the parameters, return type, concurrency semantics, and other properties of each operation. Collectively, the name of an operation plus its parameters (including its return type, if any) is called the operation's signature.

- The UML distinguishes between operation and method.
- An operation specifies a Service that can be requested from any object of the class to affect behavior;
- a method is an implementation of an operation. Every non abstract operation of a class must have a method, which supplies an executable algorithm as a body.

In its full form, the syntax of an operation in the UML is

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

For example, the following are all legal operation declarations:

• display	Name only
• + display	Visibility and name
• set(n : Name, s : String)	Name and parameters
• getID() : Integer	Name and return type
• restart() {guarded}	Name and property

In an operation's signature, you may provide zero or more parameters, each of which follows the
Syntax [direction] name : type [= default-value]

Direction may be any of the following values:

• in	An input parameter; may not be modified
• out	An output parameter; may be modified to communicate information to the caller
• inout	An input parameter; may be modified

In addition to the leaf property described earlier, there are four defined properties that you can use with operations

1. isQuery	Execution of the operation leaves the state of the system unchanged. In other words, the operation is a pure function that has no side effects.
2. 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.
3. 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.
4. concurrent	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by treating the operation as atomic. Multiple calls from concurrent flows of control may occur simultaneously to one object on any concurrent operation, and all may proceed concurrently with correct semantics;

Template Classes

- A template is a parameterized element. In such languages as C++ and Ada, you can write template classes, each of which defines a family of classes (you can also write template functions, each of which defines a family of functions).
- A template includes slots for classes, objects, and values, and these slots serve as the template's parameters.
- You can't use a template directly; you have to instantiate it first. Instantiation involves binding these formal template parameters to actual ones.
- For a template class, the result is a concrete class that can be used just like any ordinary class.
- The most common use of template classes is to specify containers that can be instantiated for specific elements, making them type-safe.

For example, the following C++ code fragment declares a parameterized Map class.

```
template<class Item, class Value, int Buckets>
class Map {
public:
```

```
virtual Boolean bind(const Item&, const Value&);
virtual Boolean isBound(const Item&) const;
...
};
```

You might then instantiate this template to map Customer objects to Order objects.

```
m : Map<Customer, Order, 3>;
```

You can model template classes in the UML as well. As Figure shows, you render a Template class just as you do an ordinary class, but with an additional dashed box in the upper right corner of the class icon, which lists the template parameters.

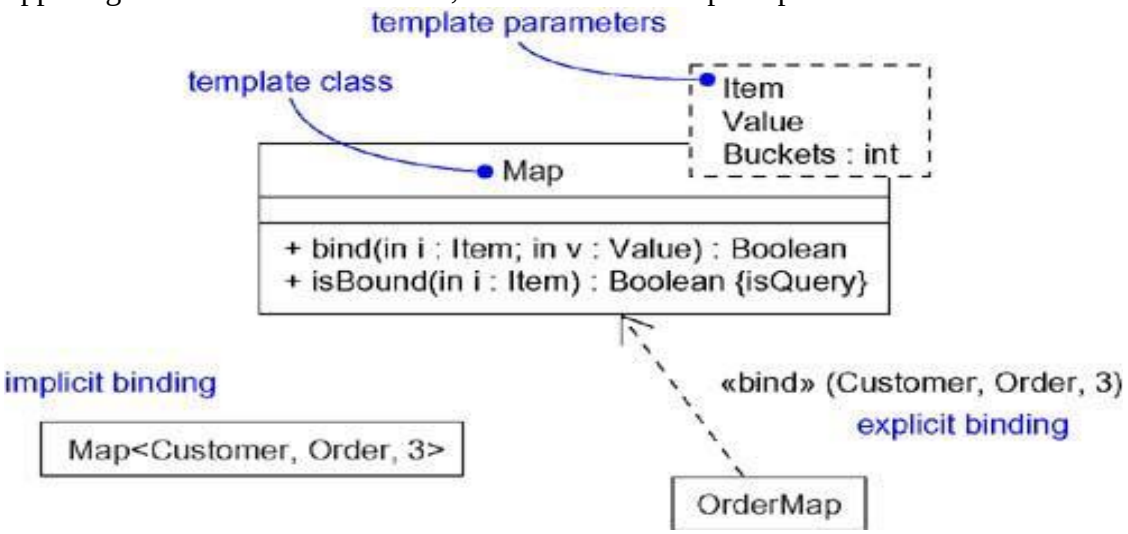


Fig: Template Classes

Standard Elements

- All of the UML's extensibility mechanisms apply to classes. Most often, you'll use tagged values to extend class properties (such as specifying the version of a class) and stereotypes to specify new kinds of components (such as model- specific components).
- The UML defines four standard stereotypes that apply to classes.

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

Common Modeling Techniques

Modeling the Semantics of a Class

To model the semantics of a class, choose among the following possibilities, arranged from informal to formal.

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

Advanced Relationships

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

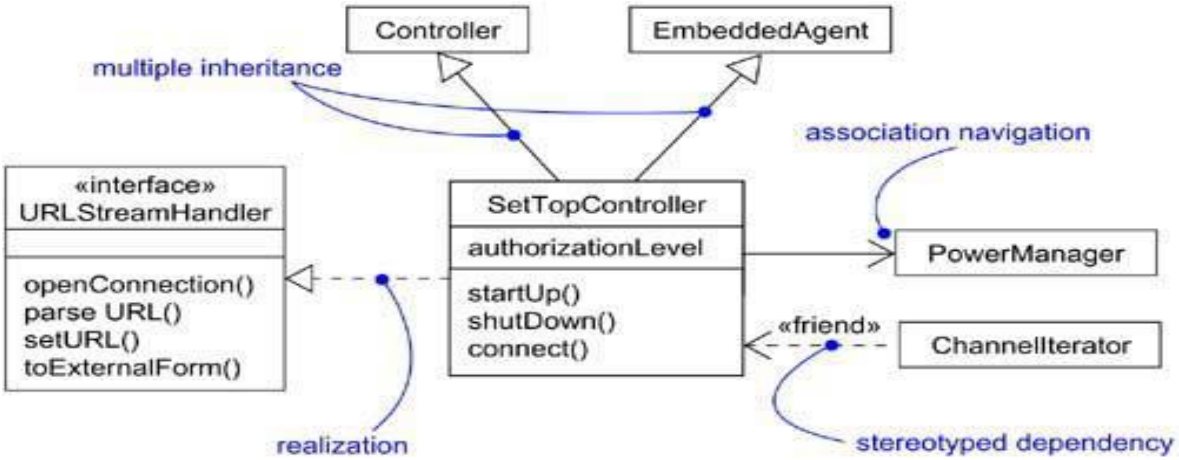


Fig: Advanced Relationships

Terms and Concepts

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

Dependency

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

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

bind:

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

derive

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

friend

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

instanceOf

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

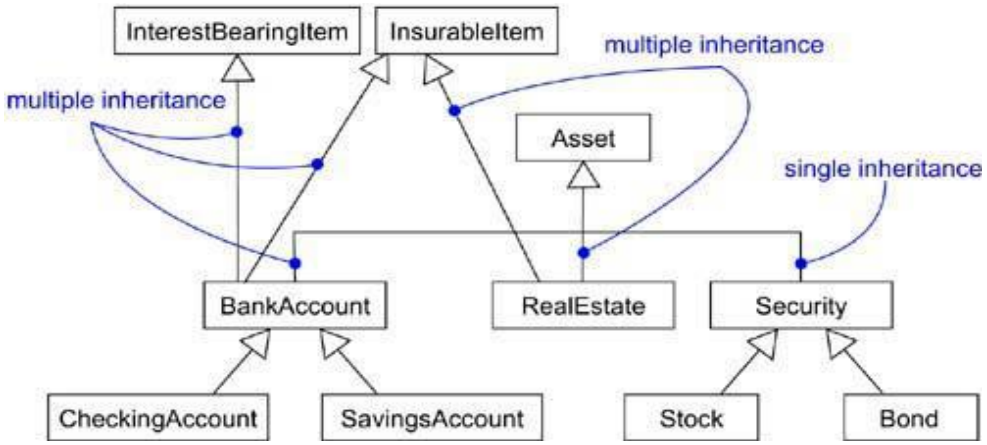


Fig: Multiple Inheritance

- For example, the above figure shows a set of classes drawn from a financial services application. You see the class Asset with three children: BankAccount, RealEstate, and Security. Two of these children (BankAccount and Security) have their own children.

- For example, Stock and Bond are both children of Security. Two of these children (BankAccount and RealEstate) inherit from multiple parents. RealEstate, for example, is a kind of Asset, as well as a kind of InsurableItem, and BankAccount is a kind of Asset, as well as a kind of InterestBearingItem and an InsurableItem.

- 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.

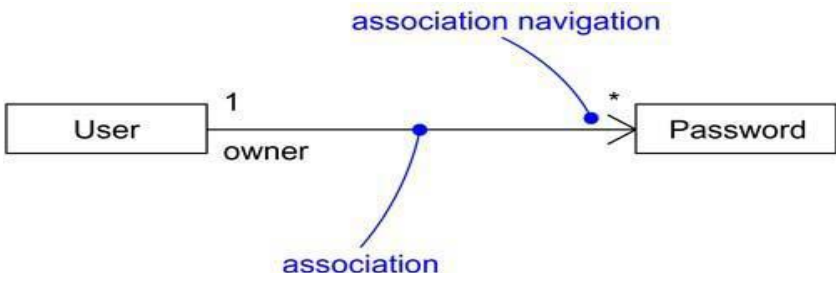


Fig: Navigation

- The above figure shows, when modeling the services of an operating system, you'll find an association between User and Password objects.
- Given a User, you'll want to be able to find the corresponding Password objects; but given a Password, you don't want to be able to identify the corresponding User.
- You can explicitly represent the direction of navigation by adorning an association with an arrowhead pointing to the direction of traversal.

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 association, except for children of the other end.

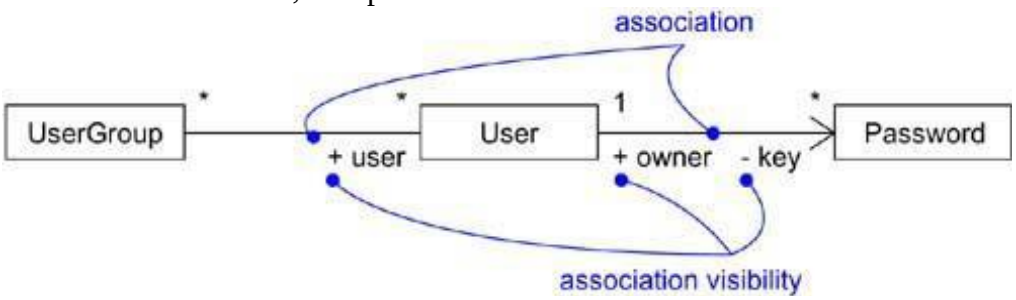


Fig: Visibility

Qualification

- In the context of an association, one of the most common modeling idioms you'll encounter is the problem of lookup. Given an object at one end of an association, how do you identify an object or set of objects at the other end.
- In the UML, you'd model this idiom using a qualifier, which is an association attribute whose values partition the set of objects related to an object across an association.
- You render a qualifier as a small rectangle attached to the end of an association, placing the attributes in the rectangle.

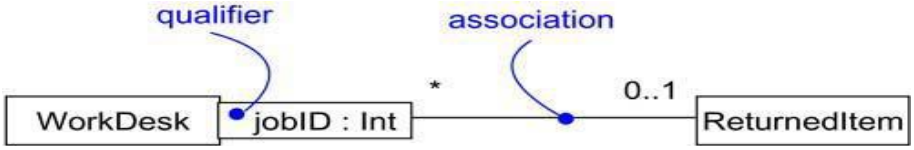


Fig: Qualification

- The above figure shows, you'd model an association between two classes, WorkDesk and ReturnedItem.
- In the context of the WorkDesk, you'd have a jobId that would identify a particular ReturnedItem. In that sense, jobId is an attribute of the association.
- It's not a feature of ReturnedItem because items really have no knowledge of things like repairs or jobs.
- An object of WorkDesk and given a particular value for jobId, you can navigate to zero or one objects of ReturnedItem

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.

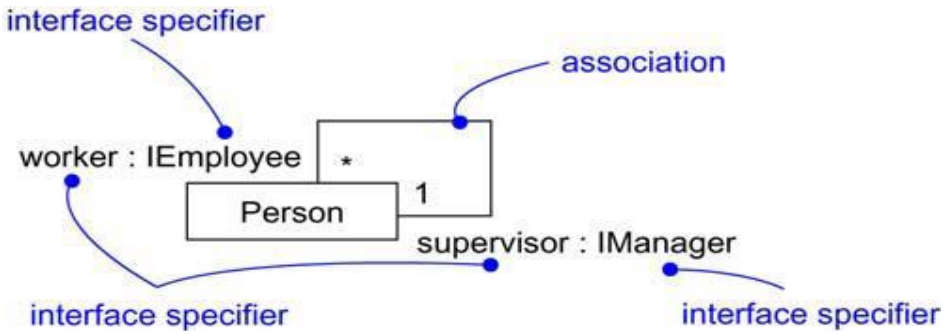


Fig: Interface Specifiers

- 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

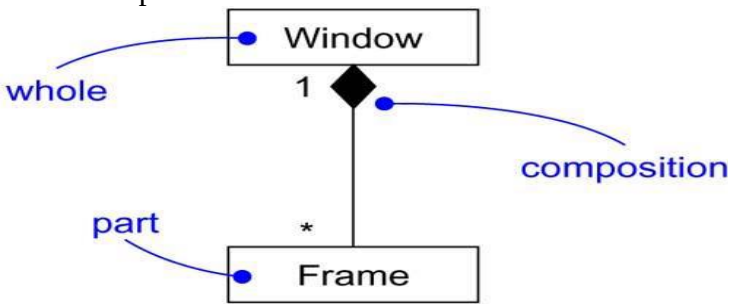


Fig: Composition

- The above figure shows, composition is really just a special kind of association and is specified by adorning a plain association with a filled diamond at the whole end.

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.

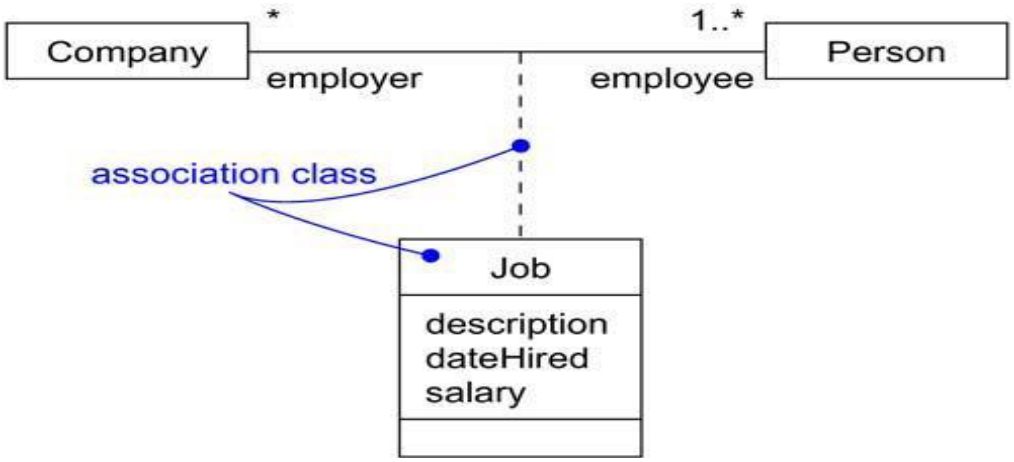


Fig: Association Classes

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

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

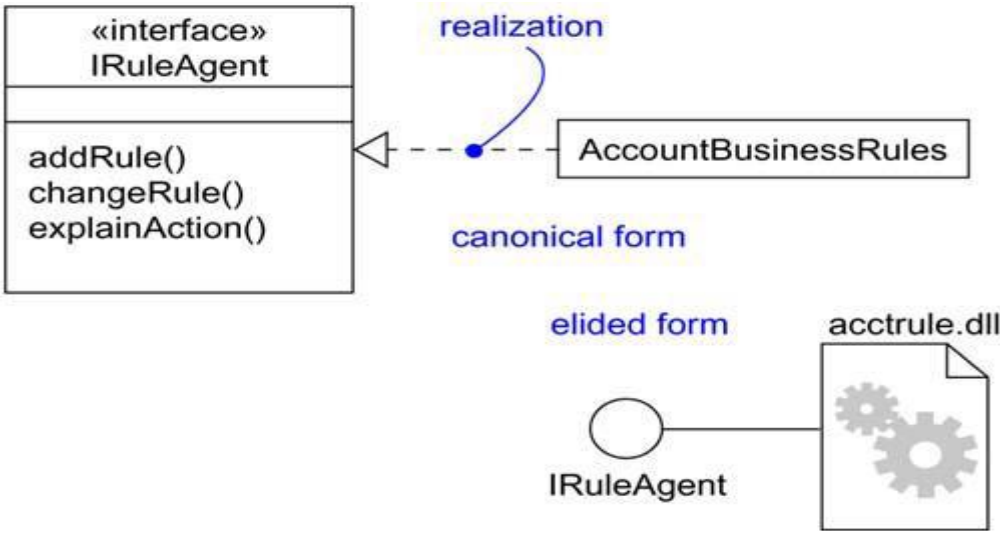


Fig: Realization of an Interface

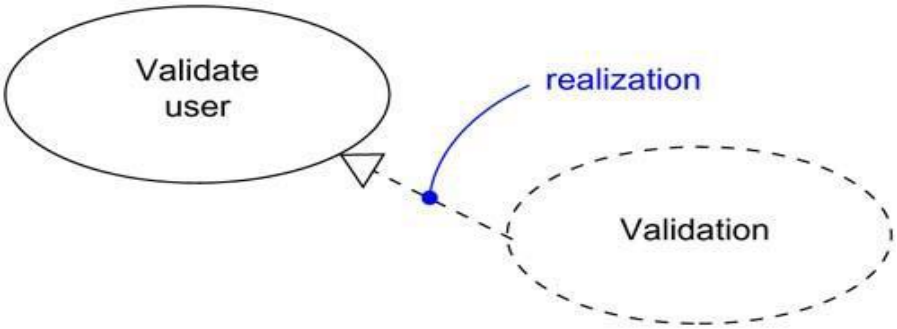


Fig: Realization of a Use Case

Common Modeling Techniques

Modeling Webs of Relationships

- 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.
- Establishing a crisp boundary around each of these abstractions is hard
- 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

To model 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

- In the UML, you use interfaces to model the seams in a system.
- An interface is a collection of operations used to specify a service of a class or a component.
- By declaring an interface, you can state the desired behavior of an abstraction independent of an implementation of that abstraction.

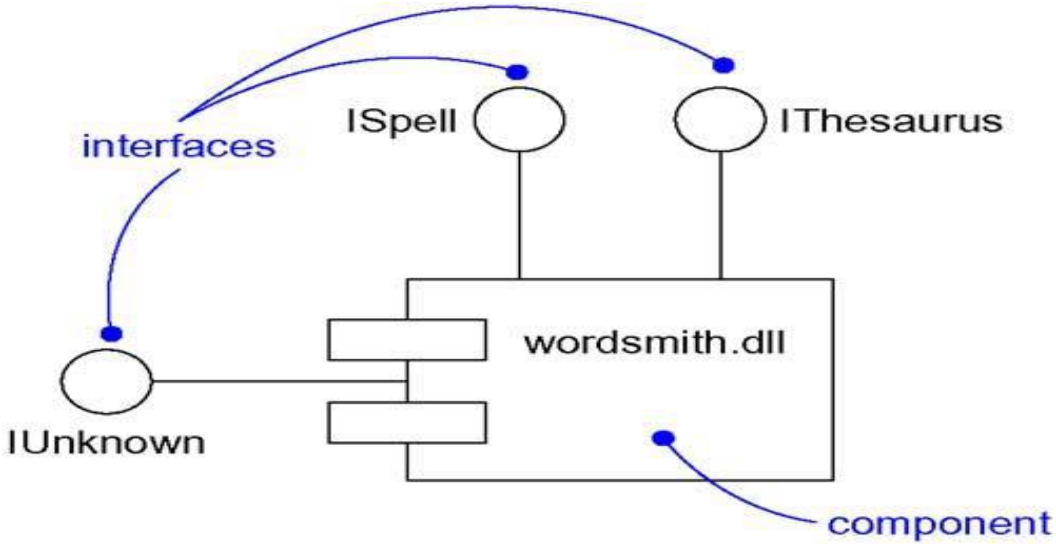


Fig: Interfaces

Terms and Concepts

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

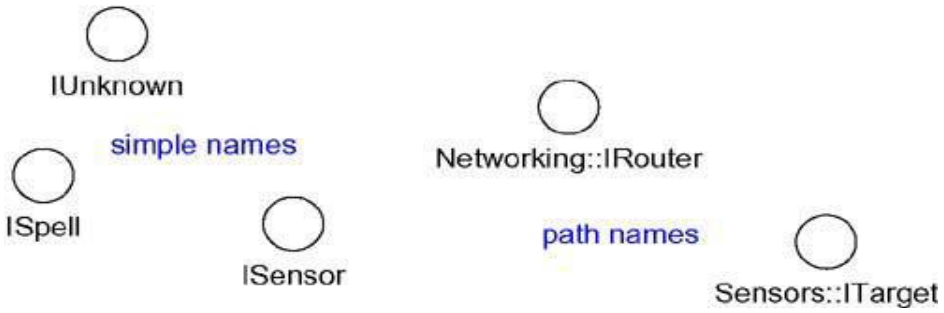


Fig: Simple and Path Names

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.

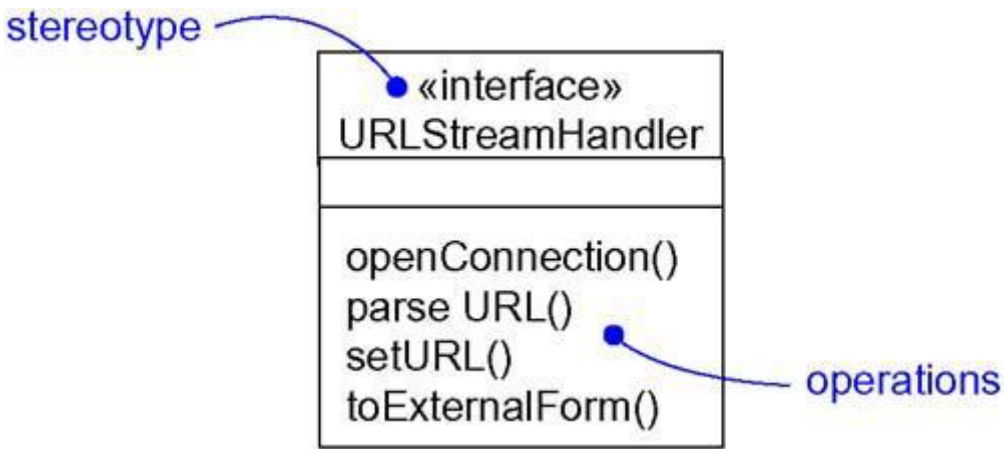


Fig: Operations

Relationships

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

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

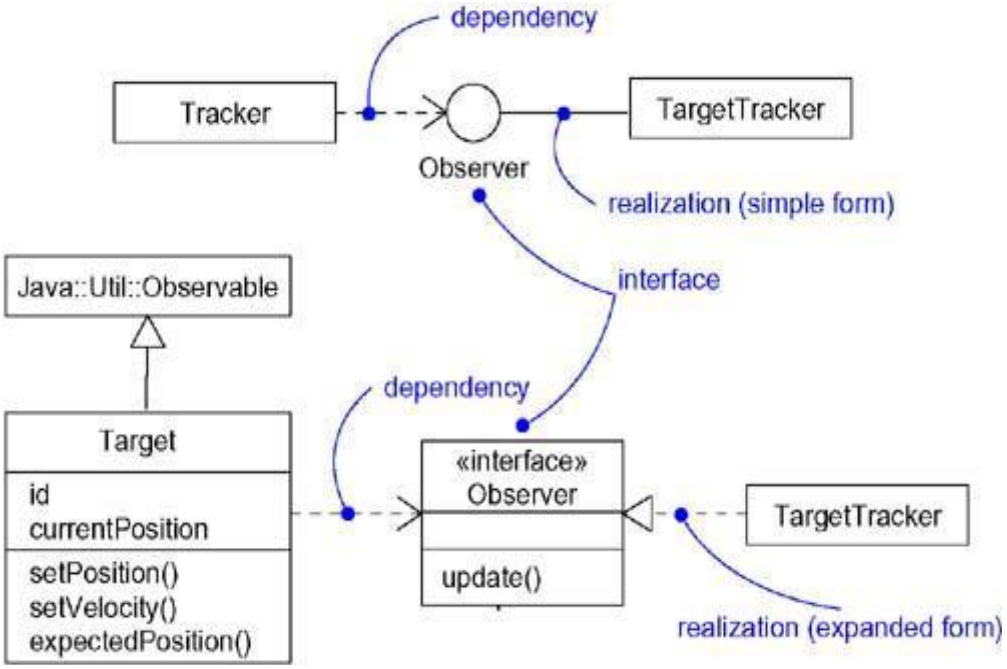


Fig: Realizations

Understanding an Interface

- In the UML, you can supply much more information to an interface in order to make it understandable and approachable.

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

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

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

Types and Roles

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

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

- An instance of Person in the role of Manager would present a different set of properties than if the instance were playing the role of Mother.
- In the UML, you can specify a role an abstraction presents to another abstraction by adorning the name of an association end with a specific interface.

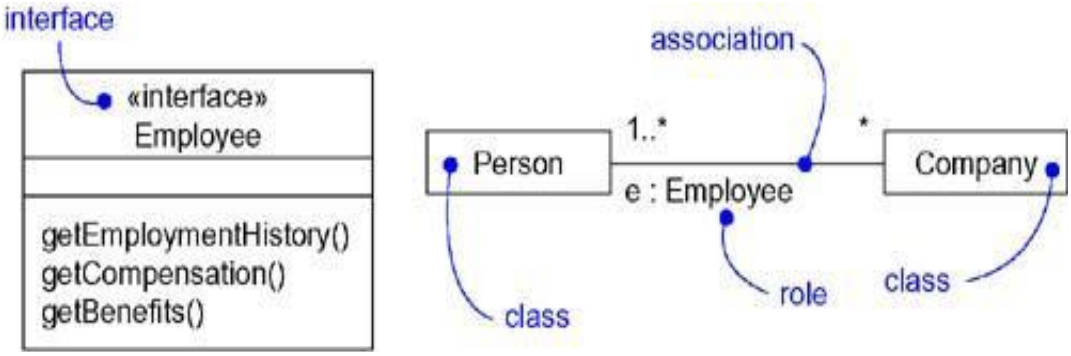


Fig: Roles

- 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,

To Modeling the Seams in a System

- Within the collection of classes and components in your system, draw a line around those that tend to be tightly coupled relative to other sets of classes and components.
- Refine your grouping by considering the impact of change. Classes or components that tend to change together should be grouped together as collaborations.
- Consider the operations and the signals that cross these boundaries, from instances of one set of classes or components to instances of other sets of classes and components.

- Package logically related sets of these operations and signals as interfaces.
- For each such collaboration in your system, identify the interfaces it relies on (imports) and those it provides to others (exports). You model the importing of interfaces by dependency relationships, and you model the exporting of interfaces by realization relationships.
- For each such interface in your system, document its dynamics by using pre- and postconditions for each operation, and use cases and state machines for the interface as a whole.

- The below 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.

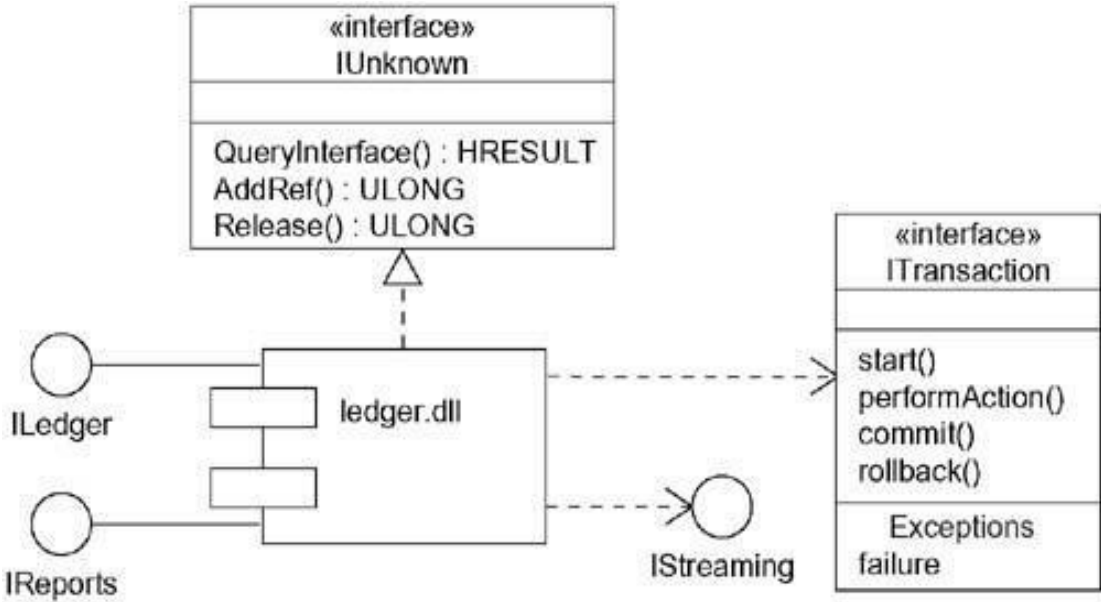


Fig: Modeling the Seams in a System

Modeling 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:

    a. 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.
    b. 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.

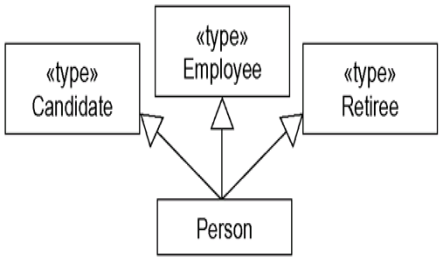


Fig: Modeling Static Types

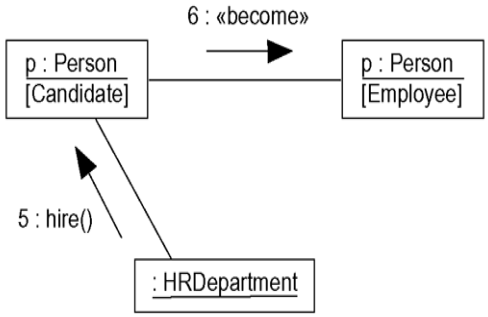


Fig: Modeling Dynamic Types

Package

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

- The UML provides a graphical representation of package, This notation permits you to visualize groups of elements that can be manipulated as a whole and in a way that lets you control the visibility of and access to individual elements.

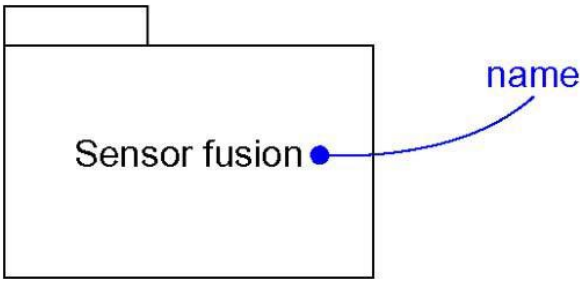


Fig: Packages

Terms and Concepts

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

Names

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

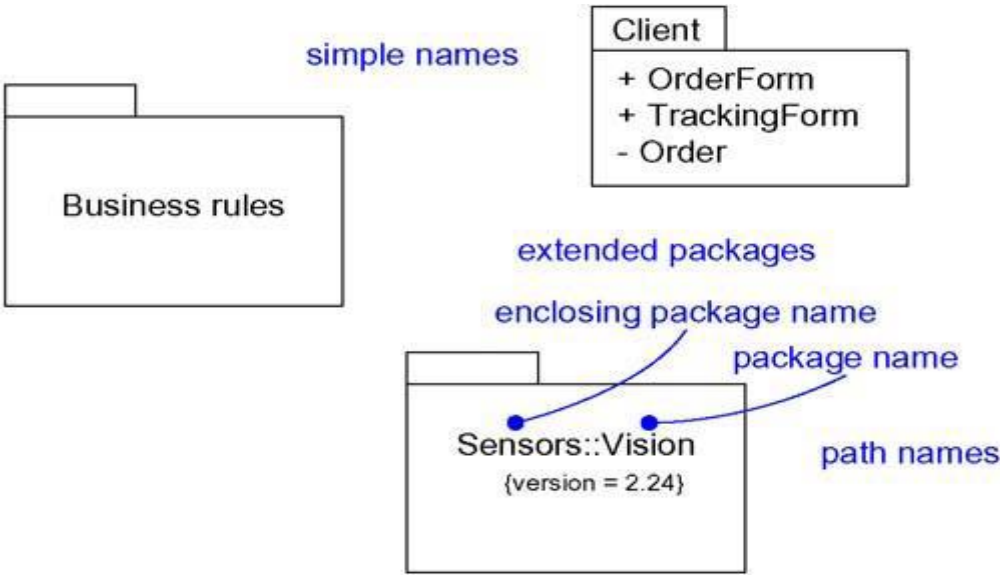


Fig: Simple and Extended Package

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.

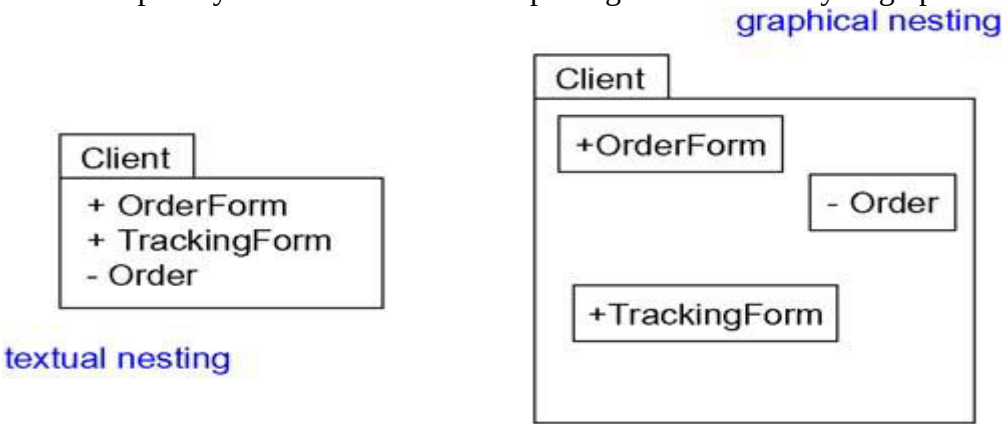


Fig: Owned Elements

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

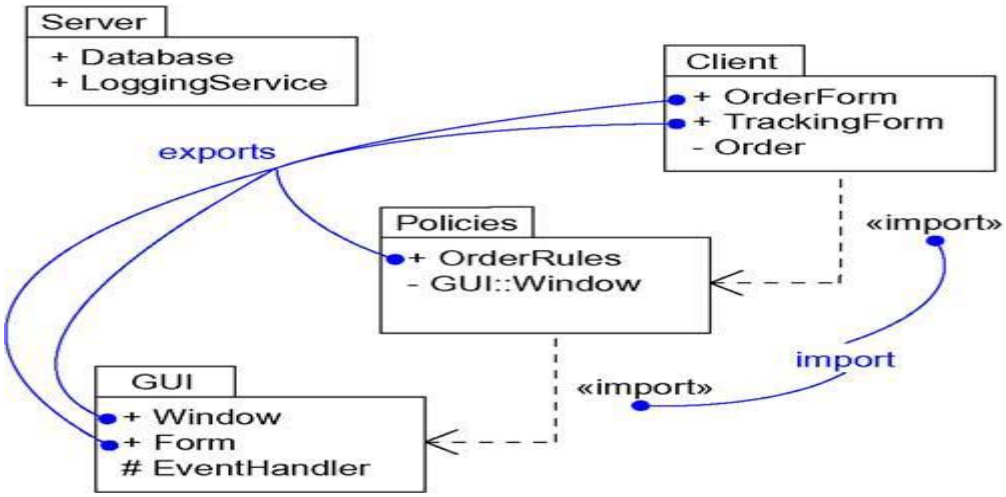


Fig: Importing and Exporting

Generalization

- There are two kinds of relationships you can have between packages: import and access dependencies used to import into one package elements exported from another and generalizations, used to specify families of packages
- Generalization among packages is very much like generalization among classes
- Packages involved in generalization relationships follow the same principle of substitutability as do classes. A specialized package (such as WindowsGUI) can be used anywhere a more general package (such as GUI) is used

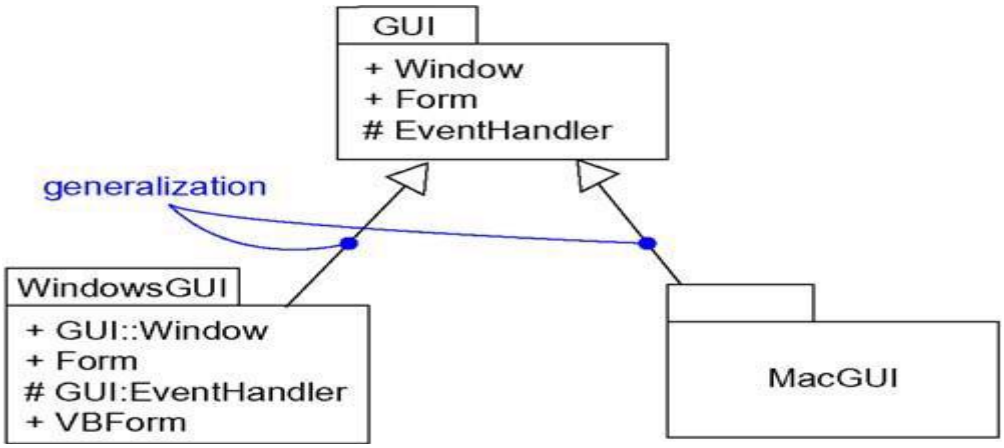


Fig: Generalization among Packages

Standard Elements

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

1. facade	Specifies a package that is only a view on some other package
2. framework	Specifies a package consisting mainly of patterns
3. stub	Specifies a package that serves as a proxy for the public contents

	of another package
4. subsystem	Specifies a package representing an independent part of the entire system being modeled
5. system	Specifies a package representing the entire system being modeled

The UML does not specify icons for any of these stereotypes

Common Modeling Techniques

Modeling Groups of Elements

- The most common purpose for which you'll use packages is to organize modeling elements into groups that you can name and manipulate as a set.
- There is one important distinction between classes and packages:
- Packages have no identity (meaning that you can't have instances of packages, so they are invisible in the running system); classes do have identity (classes have instances, which are elements of a running system).

To model groups of elements

- Scan the modeling elements in a particular architectural view and look for clumps defined by elements that are conceptually or semantically close to one another.
- Surround each of these clumps in a package.
- For each package, distinguish which elements should be accessible outside the package. Mark them public, and all others protected or private. When in doubt, hide the element.
- Explicitly connect packages that build on others via import dependencies
- In the case of families of packages, connect specialized packages to their more general part via generalizations.

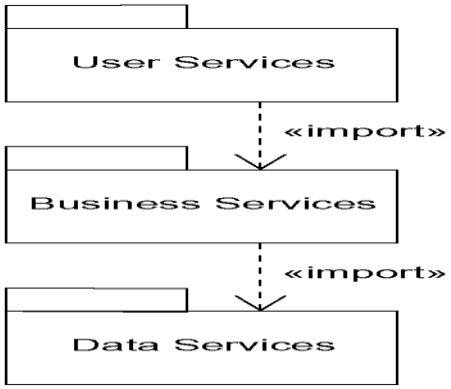


Fig: Modeling Groups of Elements

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.

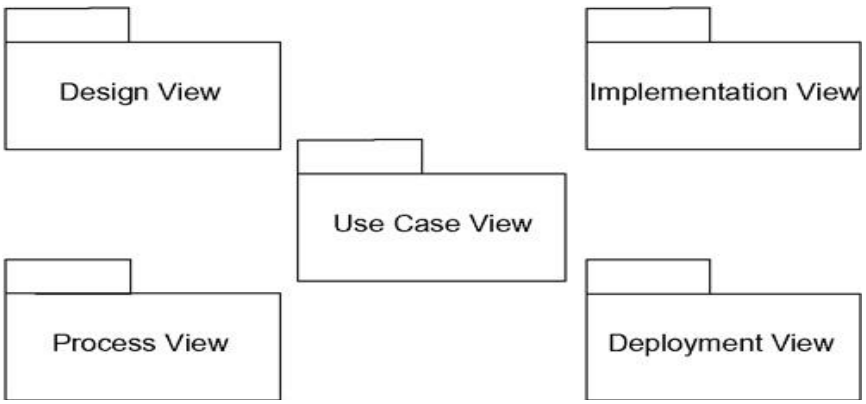


Fig: Modeling Architectural Views

UNIT - III
Class & Object Diagrams: Terms, concepts, modeling techniques for Class & Object Diagrams.
Basic Behavioral Modeling-: Interactions, Interaction diagrams. Use cases,
Use case Diagrams, Activity Diagrams.

UNIT-III
Class Diagram

- Class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- Class diagrams are also the foundation for a couple of related diagrams: component diagrams and deployment diagrams.
- Class diagrams are important not only for visualizing, specifying, and documenting structural models, but also for constructing executable systems through forward and reverse engineering.
- Graphically, a class diagram is a collection of vertices and arcs.
- Class diagrams are the most common diagram found in modeling object- oriented systems.

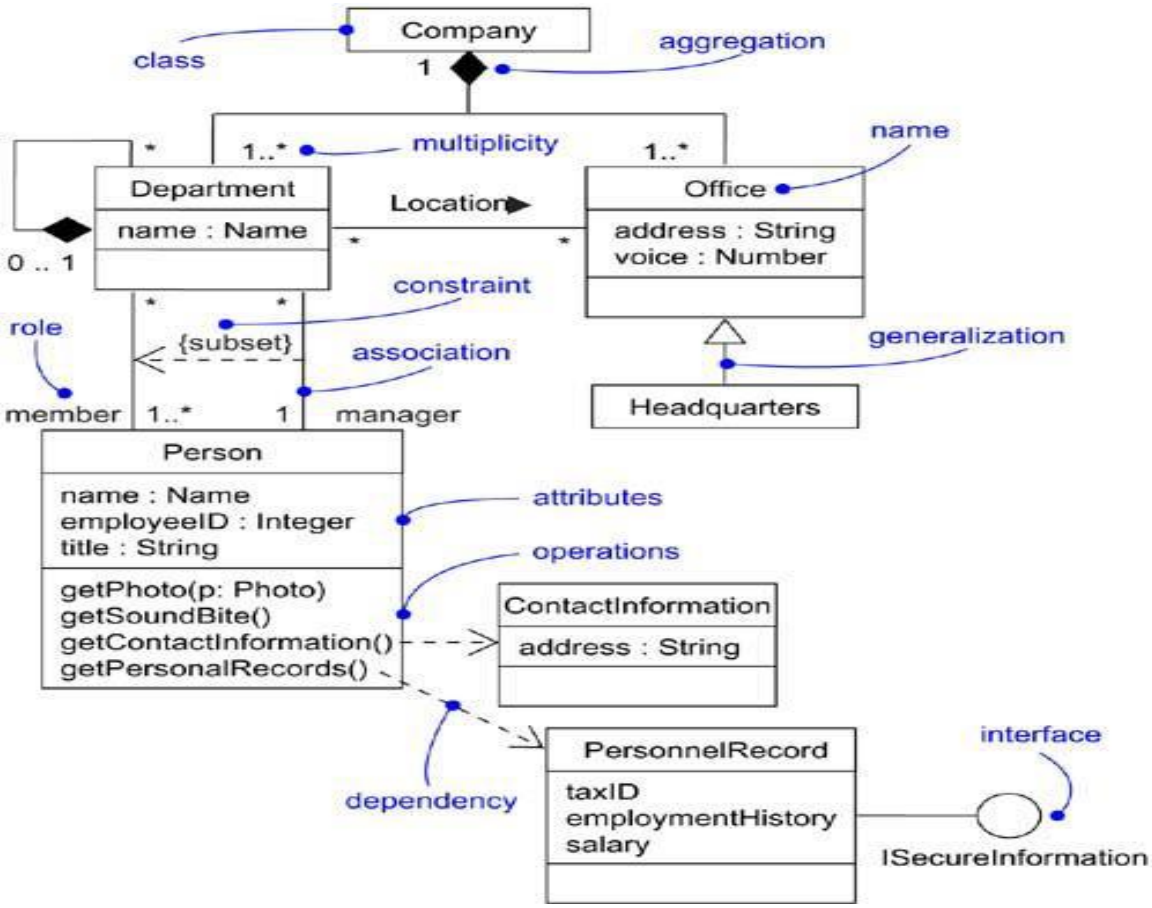


Fig: A Class Diagram

Terms and Concepts

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

Common Properties

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

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 will 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

- Schema as the blueprint for the conceptual design of a database.
- In many domains, you'll want to store persistent information in a relational database or in an object-oriented database.
- You 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 simple 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.

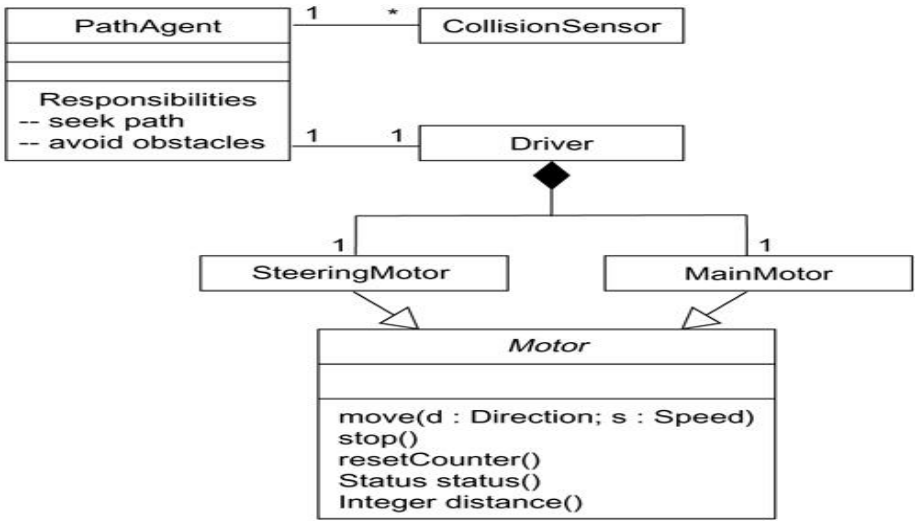


Fig: Modeling Simple Collaborations

- The above Figure shows a set of classes drawn from the implementation of an autonomous robot.
- The figure focuses on the classes involved in the mechanism for moving the robot along a path. You'll find one abstract class (Motor) with two concrete children, SteeringMotor and MainMotor.
- Both of these classes inherit the five operations of their parent, Motor. The two classes are, in turn, shown as parts of another class, Driver. The class PathAgent has a one-to-one association to Driver and a one-to-many association to CollisionSensor. No attributes or operations are shown for PathAgent, although its responsibilities are given.

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 Logical Database 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.

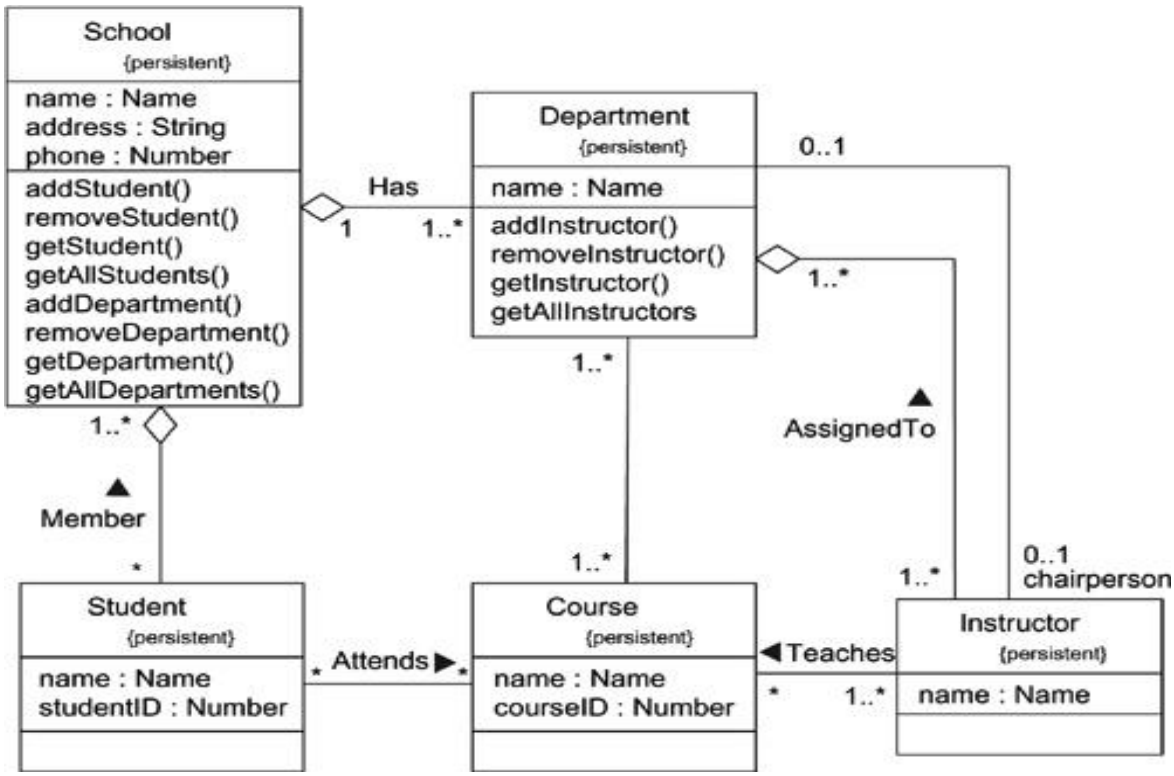


Fig: Modeling a Schema

- The above Figure shows a set of classes drawn from an information system for a school. This figure expands upon an earlier class diagram, and you'll see the details of these classes revealed to a level sufficient to construct a physical database. Starting at the bottom-left of this diagram, you will find the classes named Student, Course, and Instructor. There's an association between Student and Course, specifying that students attend courses. Furthermore, every student may attend any number of courses and every course may have any number of students.

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 language.

To forward engineer a class diagram

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

- Depending on the semantics of the languages you choose, you may 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.

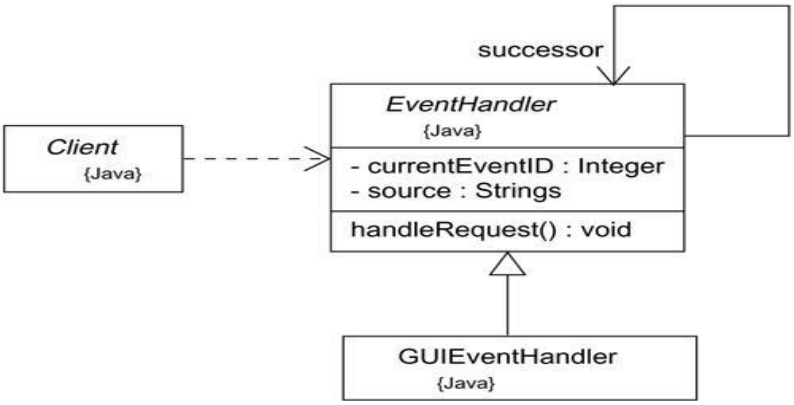


Fig: Forward Engineering

Reverse 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 Diagrams

- An object diagram is a diagram that shows a set of objects and their relationships at a point in time.
- Object diagrams model the instances of things contained in class diagrams.
- 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.
- You use object diagrams to model the static design view or static process view of a system.
- This involves modeling a snapshot of the system at a moment in time and rendering a set of objects, their state, and their relationships.

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

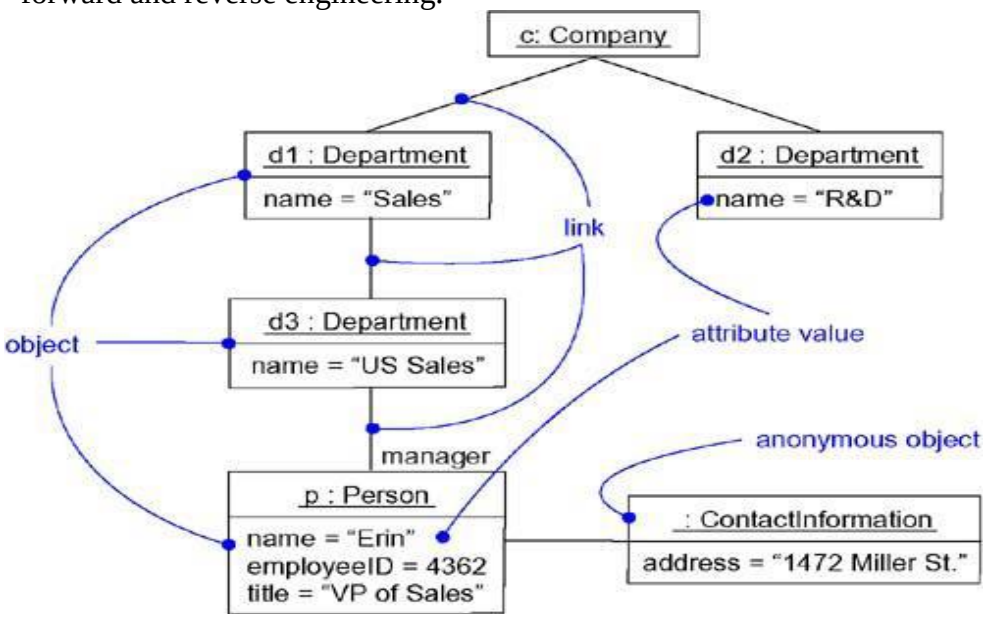


Fig: An Object Diagram

Terms and Concepts

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

Common Properties

- An object diagram is a special kind of diagram and shares the same common properties as all other diagrams that is, a name and graphical contents that are a projection into a model. What distinguishes an object diagram from all other kinds of diagrams is its particular content.

Contents

- Object diagrams commonly contain
    - Objects
    - Links

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

- 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.

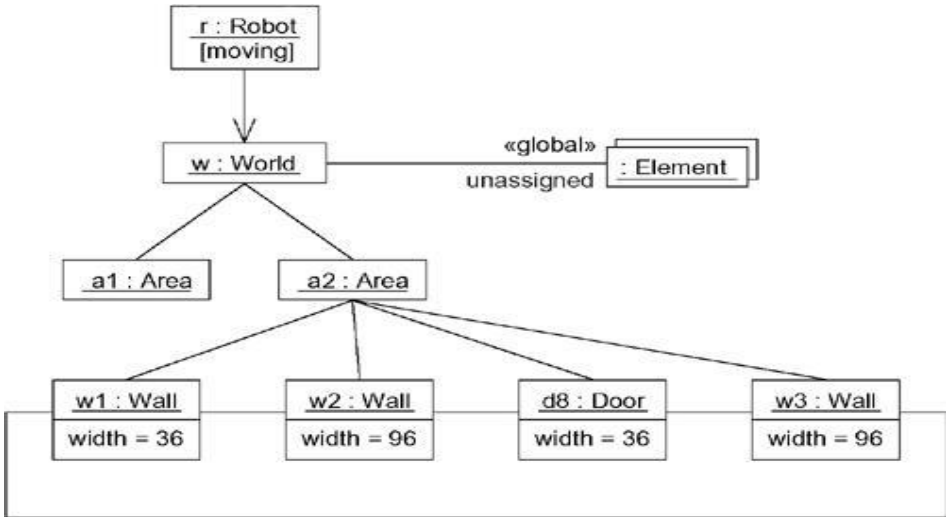


Fig: Modeling Object Structures

- For example, the above figure shows a set of objects drawn from the implementation of an autonomous robot. This figure focuses on some of the objects involved in the mechanism used by the robot to calculate a model of the world in which it moves. There are many more objects involved in a running system, but this diagram focuses on only those abstractions that are directly involved in creating this world view.

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.

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.

- The UML provides a graphical representation of messages, as Figure 15-1 shows. This notation permits you to visualize a message in a way that lets you emphasize its most important parts: its name, parameters (if any), and sequence.

- Graphically, a message is rendered as a directed line and almost always includes the name of its operation.

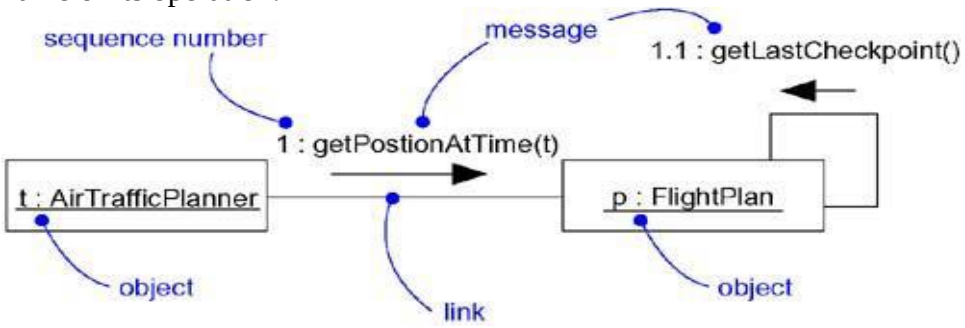


Fig: Messages, Links, and Sequencing

Terms and Concepts

- 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.

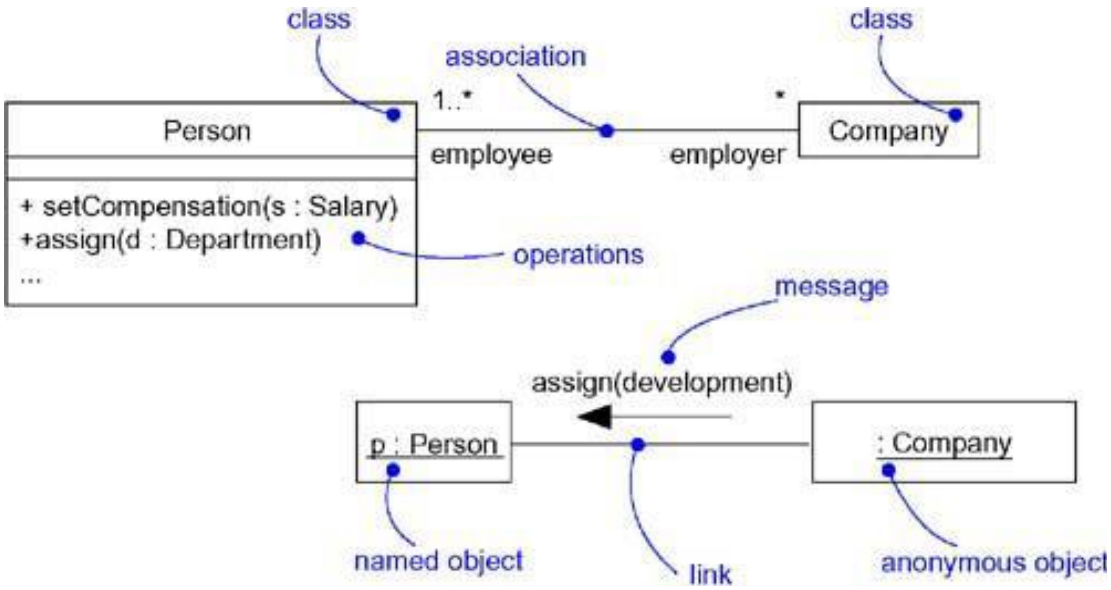


Fig: Links and Associations

- 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

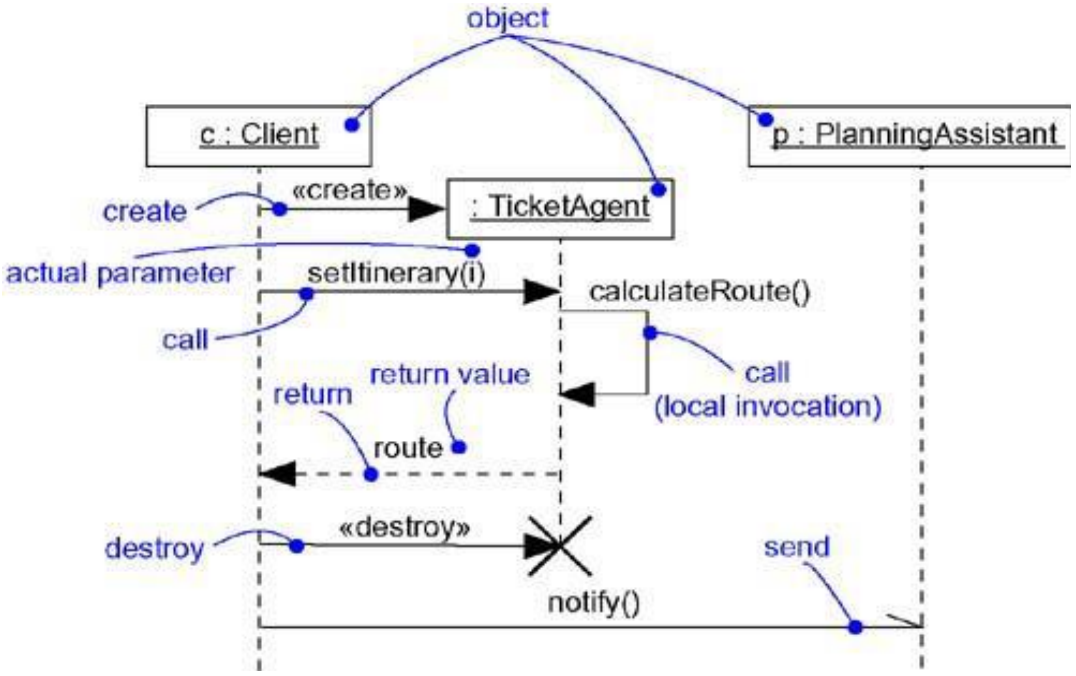


Fig: Messages

- 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

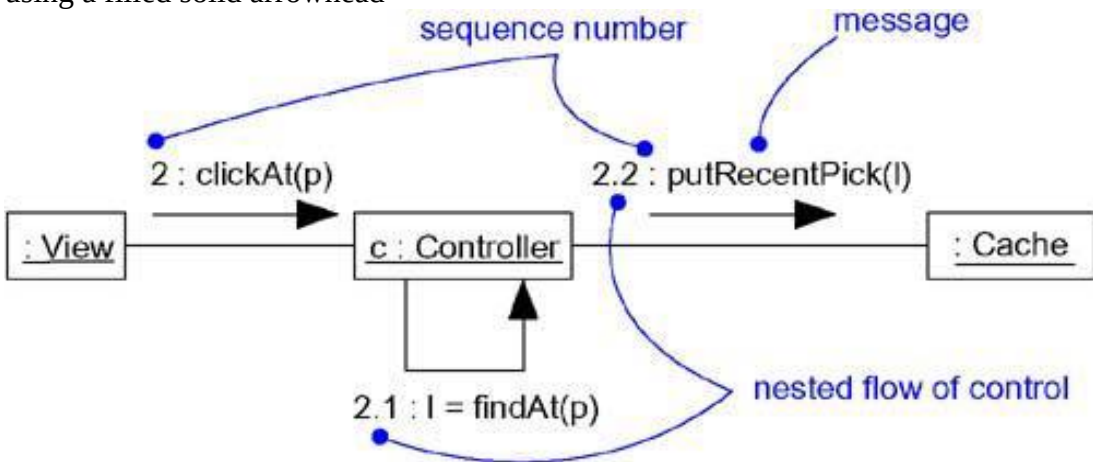


Fig: Procedural Sequence

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

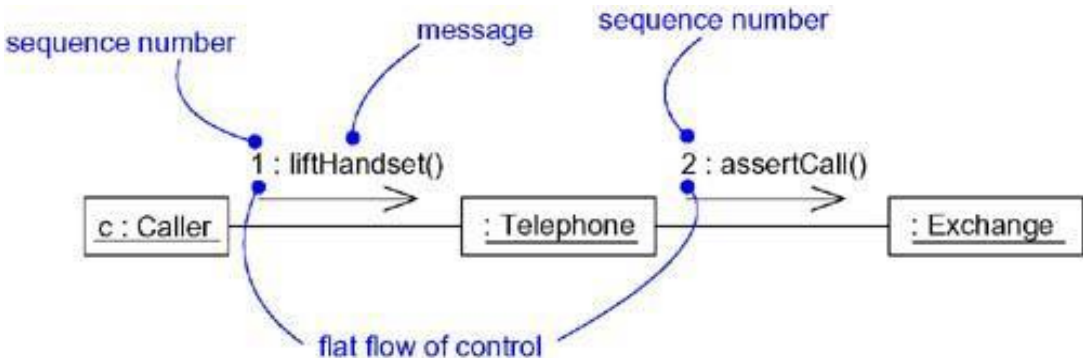


Fig: Flat Sequence

- 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.

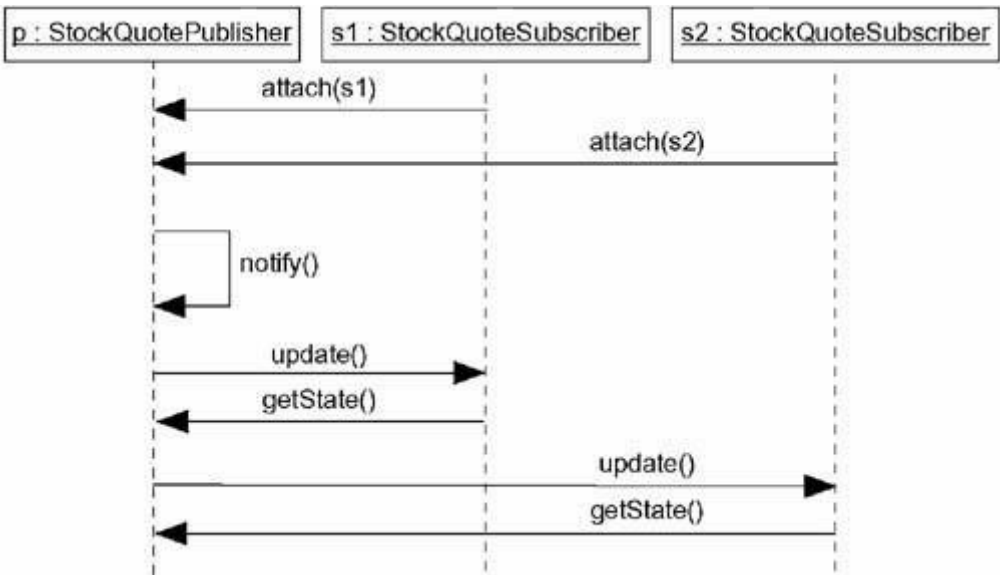


Fig: Flow of Control by Time

- 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.

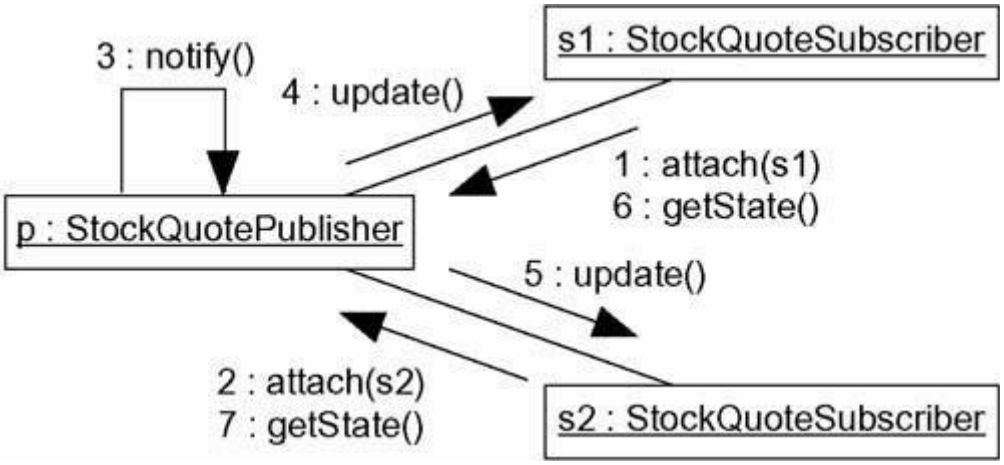


Fig: Flow of Control by Organization

Interaction Diagrams

- Sequence diagrams and collaboration diagrams• both of which are called interaction diagrams are two of the five diagrams used in the UML for modeling the dynamic aspects of systems.
- An interaction diagram shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them.

- A sequence diagram is an interaction diagram that emphasizes the time ordering of messages.
- A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.

- The below figure shows, you can build up these storyboards in two ways: by emphasizing the time ordering of messages and by emphasizing the structural relationships among the objects that interact. Either way, the diagrams are semantically equivalent; you can convert one to the other without loss of information.

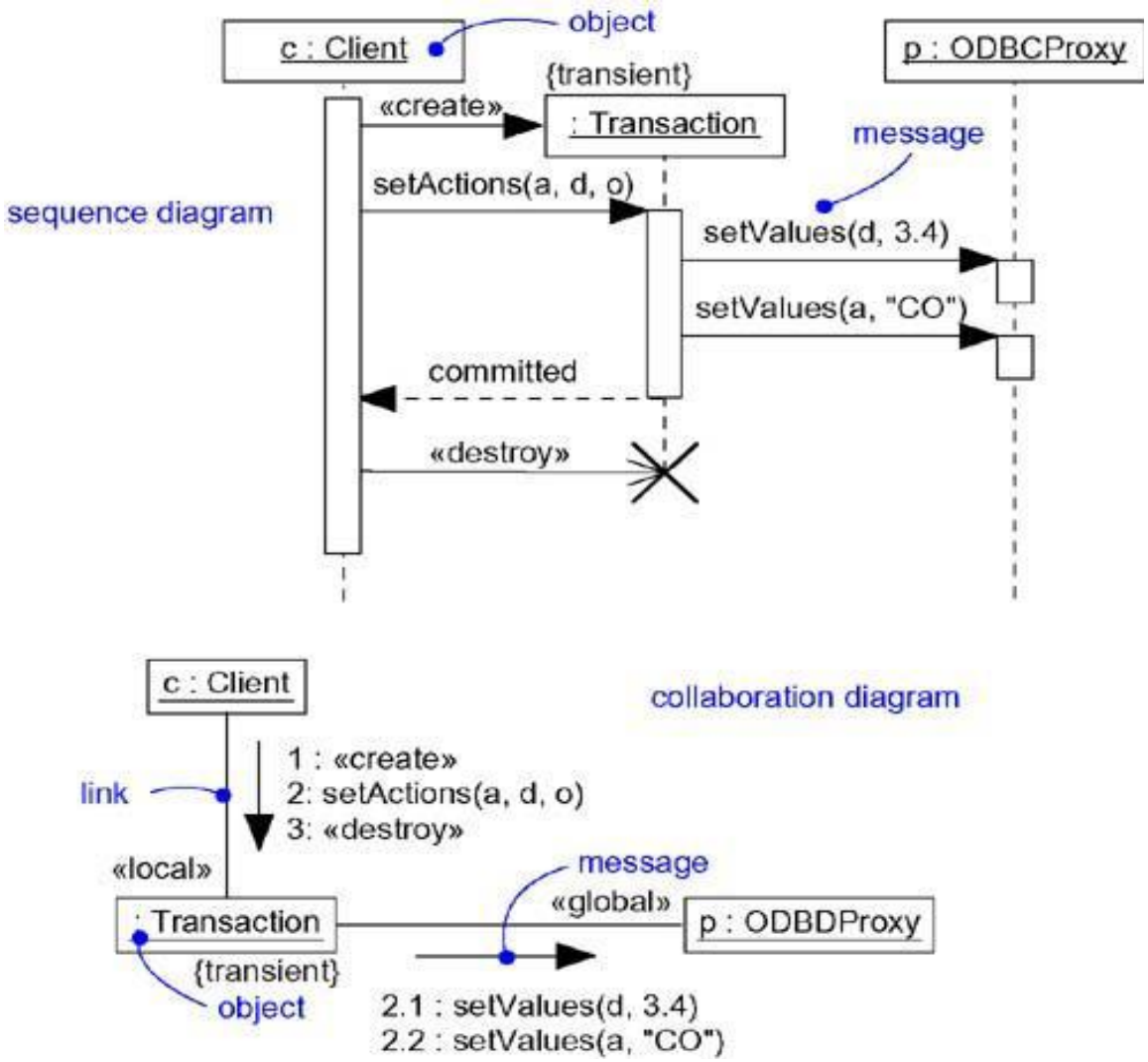


Fig: Interaction Diagrams

Terms and Concepts

- An interaction diagram shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them.

- A sequence diagram is an interaction diagram that emphasizes the time ordering of messages. Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis.

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

- Graphically, a collaboration diagram is a collection of vertices and arcs.

Common Properties

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

Contents

Interaction diagrams commonly contain

- Objects
- Links
- Messages

Note

- An interaction diagram is basically a projection of the elements found in an interaction.
- The semantics of an interaction's context, objects and roles, links, messages, and sequencing apply to interaction diagrams.

Sequence Diagrams

- A sequence diagram emphasizes the time ordering of messages.
- As the below figure shows, you form a sequence diagram by first placing the objects that participate in the interaction at the top of your diagram, across the X axis.
- Typically, you place the object that initiates the interaction at the left, and increasingly more subordinate objects to the right. Next, you place the messages that these objects send and receive along the Y axis, in order of increasing time from top to bottom.
- This gives the reader a clear visual cue to the flow of control over time.

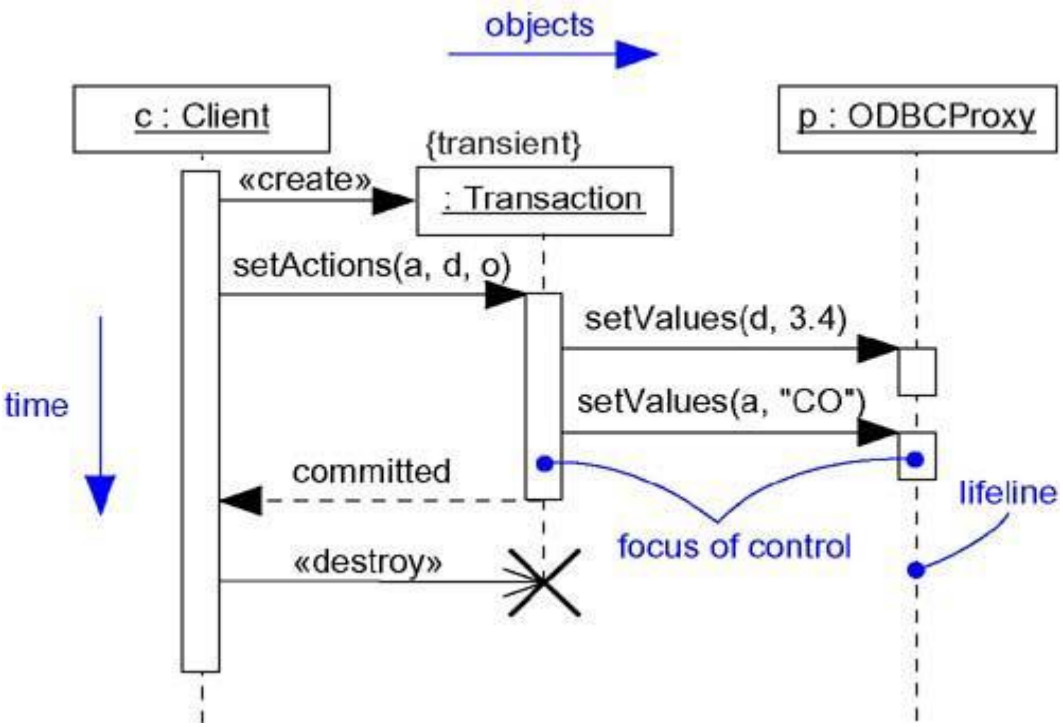


Fig: Sequence Diagram

Sequence diagrams have **two** features that distinguish them from collaboration diagrams

First, there is the object lifeline.

- An object lifeline is the vertical dashed line that represents the existence of an object over a period of time.

- Most objects that appear in an interaction diagram will be in existence for the duration of the interaction, so these objects are all aligned at the top of the diagram, with their lifelines drawn from the top of the diagram to the bottom.
- Objects may be created during the interaction. Their lifelines start with the receipt of the message stereotyped as create.
- Objects may be destroyed during the interaction. Their lifelines end with the receipt of the message stereotyped as destroy (and are given the visual cue of a large X, marking the end of their lives).

Second, there is the **focus of control**.

- The focus of control is a tall, thin rectangle that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure.
- The top of the rectangle is aligned with the start of the action; the bottom is aligned with its completion (and can be marked by a return message).

Collaboration Diagrams

- A collaboration diagram emphasizes the organization of the objects that participate in an Interaction.
- As Figure shows, you form a collaboration diagram by first placing the objects that participate in the interaction as the vertices in a graph.
- Next, you render the links that connect these objects as the arcs of this graph.
- Finally, you adorn these links with the messages that objects send and receive.
- This gives the reader a clear visual cue to the flow of control in the context of the structural organization of objects that collaborate.

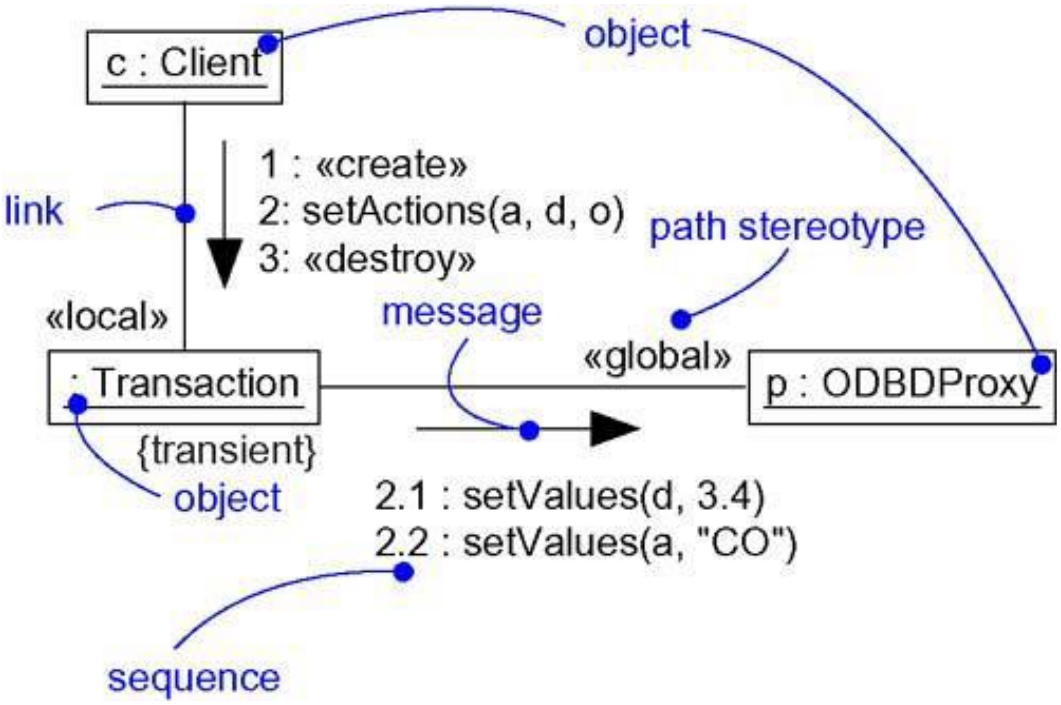


Fig: Collaboration Diagram

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

First, there is the path.

- To indicate how one object is linked to another, you can attach a path stereotype to the far end of a link (such as »local«, indicating that the designated object is local to the sender).
- Typically, you will only need to render the path of the link explicitly for local, parameter, global, and self (but not association) paths.

Second, there is the sequence number.

- To indicate the time order of a message, you prefix the message with a number (starting with the message numbered 1), increasing monotonically for each new message in the flow of control (2, 3, and so on).
- To show nesting, you use Dewey decimal numbering (1 is the first message; 1.1 is the first message nested in message 1; 1.2 is the second message nested in message 1; and so on).
- You can show nesting to an arbitrary depth. Note also that, along the same link, you can show many messages (possibly being sent from different directions), and each will have a unique sequence number

Semantic Equivalence

- Sequence diagrams and collaboration diagrams are semantically equivalent. As a result, you can take a diagram in one form and convert it to the other without any loss of information.

Common Uses

When you model the dynamic aspects of a system, you typically use interaction diagrams in two ways.

1. To model flows of control by time ordering

- Here you'll use sequence diagrams. Modeling a flow of control by time ordering emphasizes the passing of messages as they unfold over time, which is a particularly useful way to visualize dynamic behavior in the context of a use case scenario.
- Sequence diagrams do a better job of visualizing simple iteration and branching than do collaboration diagrams.

2. To model flows of control by organization

- Here you'll use collaboration diagrams. Modeling a flow of control by organization emphasizes the structural relationships among the instances in the interaction, along which messages may be passed.
- Collaboration diagrams do a better job of visualizing complex iteration and branching and of visualizing multiple concurrent flows of control than do sequence diagrams.

Common Modeling Techniques

Modeling Flows of Control by Time Ordering

To model a flow of control by time ordering

- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the sequence diagram from left to right, placing the more important objects to the left and their neighboring objects to the right.
- Set the lifeline for each object. In most cases, objects will persist through the entire interaction. For those objects that are created and destroyed during the interaction, set their lifelines, as appropriate, and explicitly indicate their birth and death with appropriately stereotyped messages.
- Starting with the message that initiates this interaction, lay out each subsequent message from top to bottom between the lifelines, showing each message's properties (such as its parameters), as necessary to explain the semantics of the interaction.
- If you need to visualize the nesting of messages or the points in time when actual computation is taking place, adorn each object's lifeline with its focus of control
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.
- If you need to specify this flow of control more formally, attach pre- and post-conditions to each message.

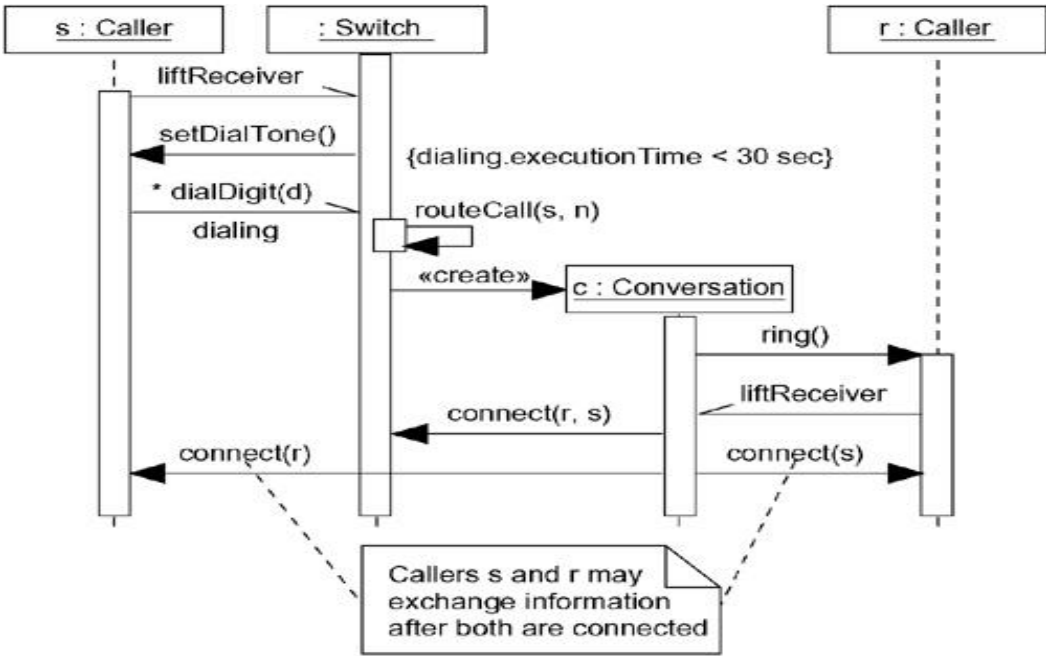


Fig: Modeling Flows of Control by Time Ordering

Modeling Flows of Control by Organization

To model a flow of control by organization

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

- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the collaboration diagram as vertices in a graph, placing the more important objects in the center of the diagram and their neighboring objects to the outside.
- Set the initial properties of each of these objects. If the attribute values, tagged values, state, or role of any object changes in significant ways over the duration of the interaction, place a duplicate object on the diagram, update it with these new values, and connect them by a message stereotyped as become or copy (with a suitable sequence number).
- Specify the links among these objects, along which messages may pass.
    i. Lay out the association links first; these are the most important ones, because they represent structural connections.
    ii. Lay out other links next, and adorn them with suitable path stereotypes (such as global and local) to explicitly specify how these objects are related to one another.
- Starting with the message that initiates this interaction, attach each subsequent message to the appropriate link, setting its sequence number, as appropriate. Show nesting by using Dewey decimal numbering.
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.
- If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

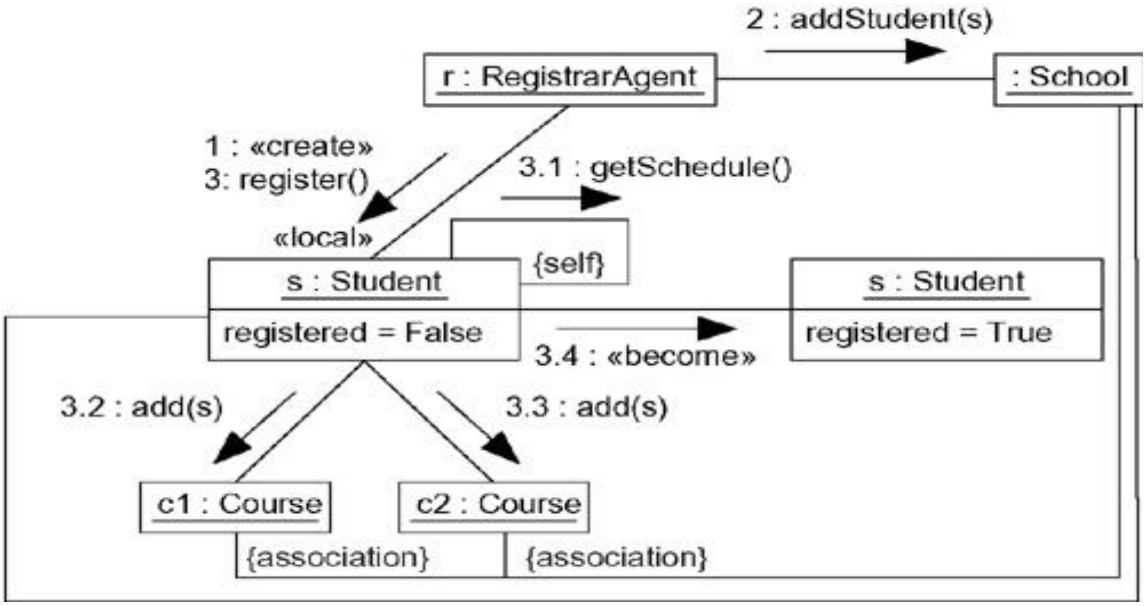


Fig: Modeling Flows of Control by Organization

Forward and Reverse Engineering

- Forward engineering (the creation of code from a model) is possible for both sequence and collaboration diagrams, especially if the context of the diagram is an operation.

- For example, using the previous collaboration diagram, a reasonably clever forward engineering tool could generate the following Java code for the operation register, attached to the Student class.

```
public void register() {
CourseCollection c = getSchedule();
for (int i = 0; i < c.size(); i++)
c.item(i).add(this);
this.registered = true;
}
```

- "Reasonably clever" means the tool would have to realize that getSchedule returns a CourseCollection object, which it could determine by looking at the operation's signature. By walking across the contents of this object using a standard iteration idiom (which the tool could know about implicitly), the code could then generalize to any number of course offerings.

- Reverse engineering (the creation of a model from code) is also possible for both sequence and collaboration diagrams, especially if the context of the code is the body of an operation. Segments of the previous diagram could have been produced by a tool from a prototypical execution of the register operation.

Usecases

- 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.
- The UML provides a graphical representation of a use case and an actor, shown in figure.

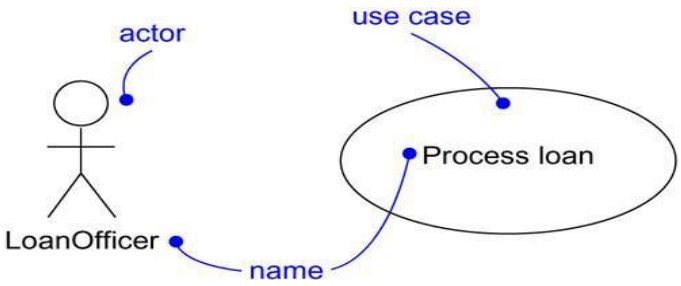


Fig: Actors and Use Cases

Terms and Concepts

- 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

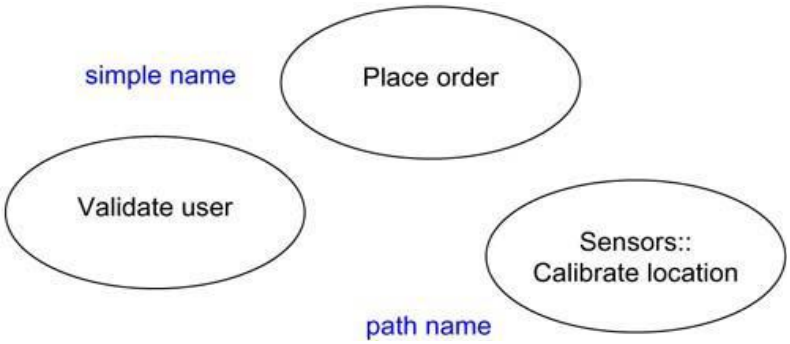


Fig: Simple and Path Names

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.

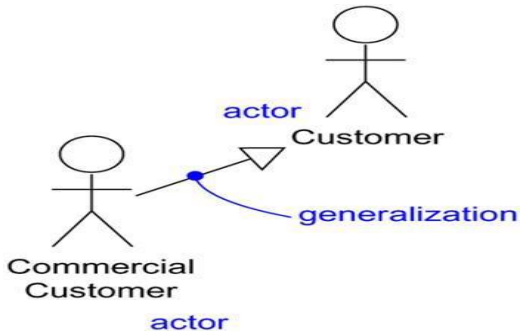


Fig: Actors

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.

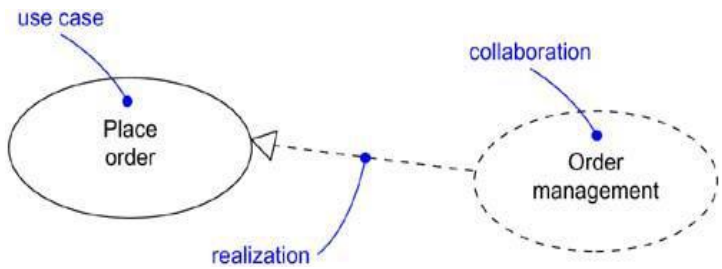


Fig: Use Cases and Collaborations

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.

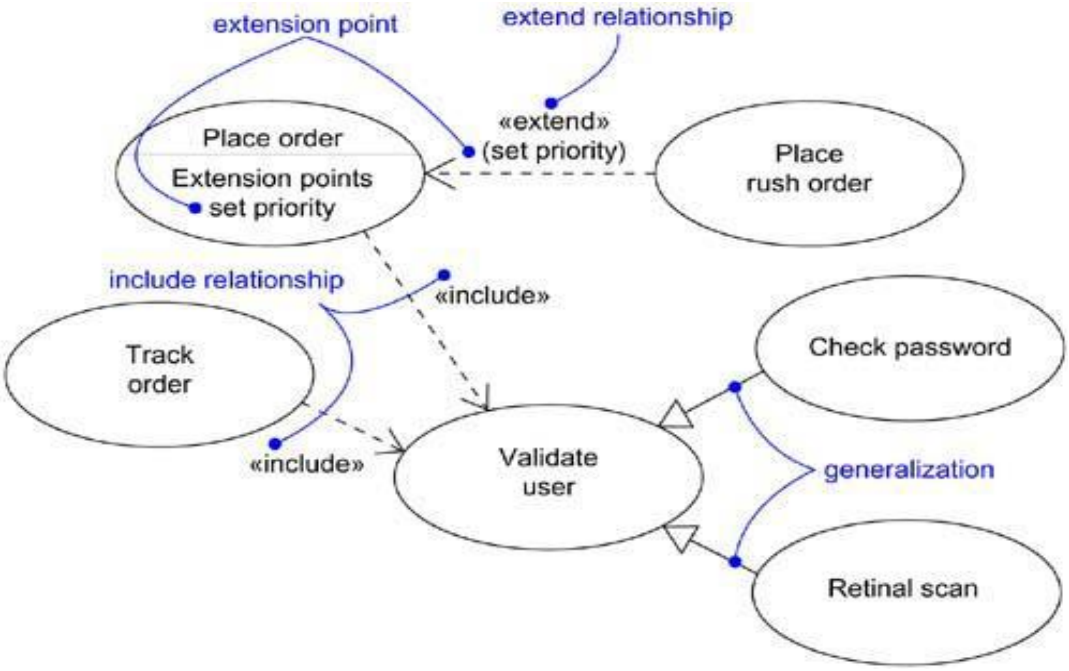


Fig: Generalization, Include, and Extend

Other Features

- Use cases are classifiers, so they may have attributes and operations that you may render just as for classes.
- You can think of these attributes as the objects inside the use case that you need to describe its outside behavior. Similarly, you can think of these operations as the actions of the system you need to describe a flow of events.
- These objects and operations may be used in your interaction diagrams to specify the behavior of the use case
- As classifiers, you can also attach state machines to use cases
- We can use state machines as yet another way to describe the behavior represented by a use case.

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.

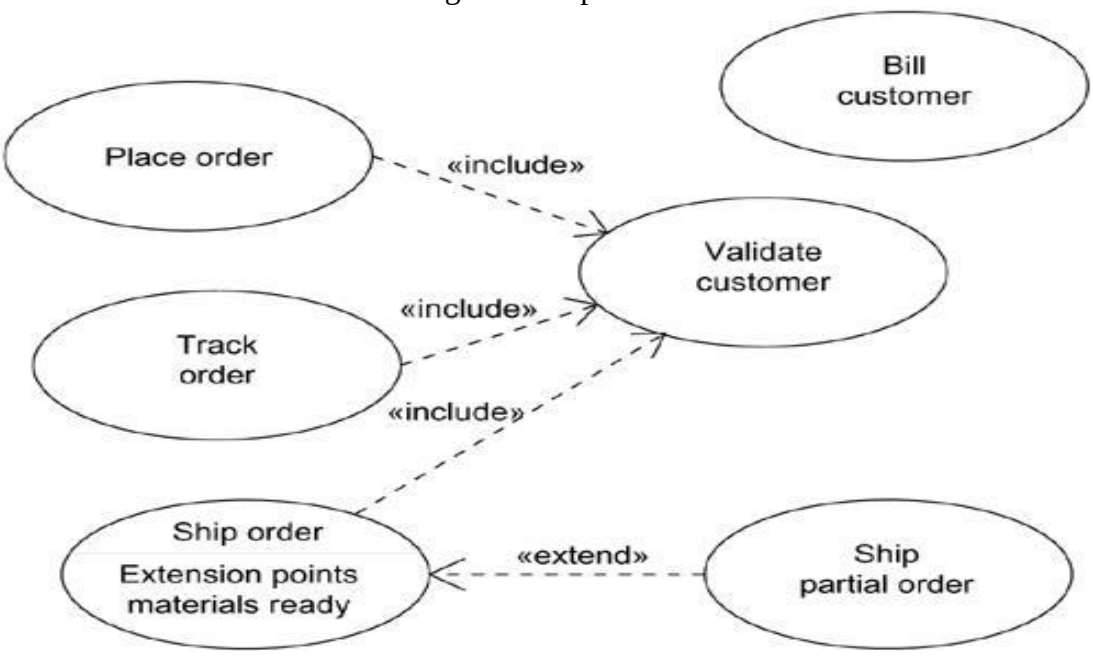


Fig: Modeling the Behavior of an Element

Use Case Diagrams

- Use case diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems (activity diagrams, statechart diagrams, sequence diagrams, and collaboration diagrams are four other kinds of diagrams in the UML for modeling the dynamic aspects of systems).

- Use case diagrams are central to modeling the behavior of a system, a subsystem, or a class.

- Each one shows a set of use cases and actors and their relationships.

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

- You apply use case diagrams to model the use case view of a system.

- Use case diagrams are important for visualizing, specifying, and documenting the behavior of an element. They make systems, subsystems, and classes approachable and understandable by presenting an outside view of how those elements may be used in context.

- Use case diagrams are also important for testing executable systems through forward engineering and for comprehending executable systems through reverse engineering.

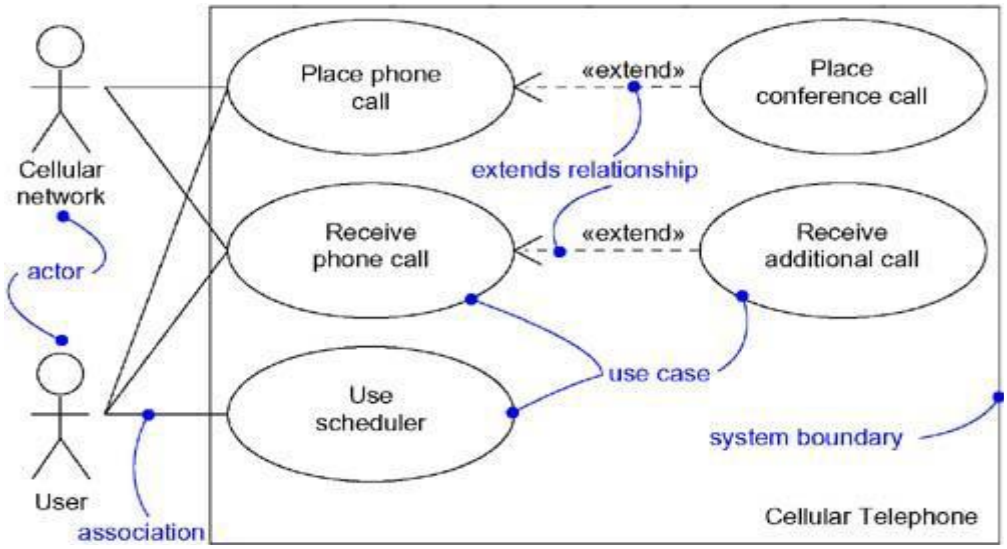


Fig: A Use Case Diagram

Terms and Concepts

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

Common Properties

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

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

- It 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

- It 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

- Given a system—any system—some things will live inside the system, some things will live outside it. For example, in a credit card validation system, you'll find such things as accounts, transactions, and fraud detection agents inside the system. Similarly, you'll find such things as credit card customers and retail institutions outside the system. The things that live inside the system are responsible for carrying out the behavior that those on the outside expect the system to provide. All those things on the outside that interact with the system constitute the system's context. This context defines the environment in which that system lives.

- 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.

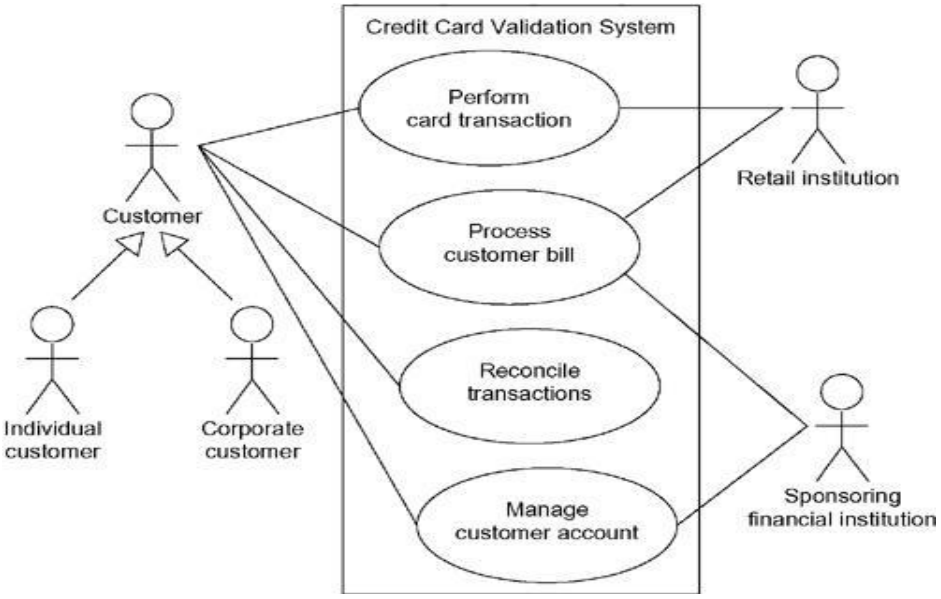


Fig: Modeling the Context of a System

Modeling the Requirements of a System

- A requirement is a design feature, property, or behavior of a system. When you state a system's requirements, you are asserting a contract, established between those things that lie outside the system and the system itself, which declares what you expect that system to do.
- Requirements can be expressed in various forms, from unstructured text to expressions in a formal language, and everything in between.

- Most, if not all, of a system's functional requirements can be expressed as use cases, and the UML's use case diagrams are essential for managing these requirements.

To model the requirements of a system

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

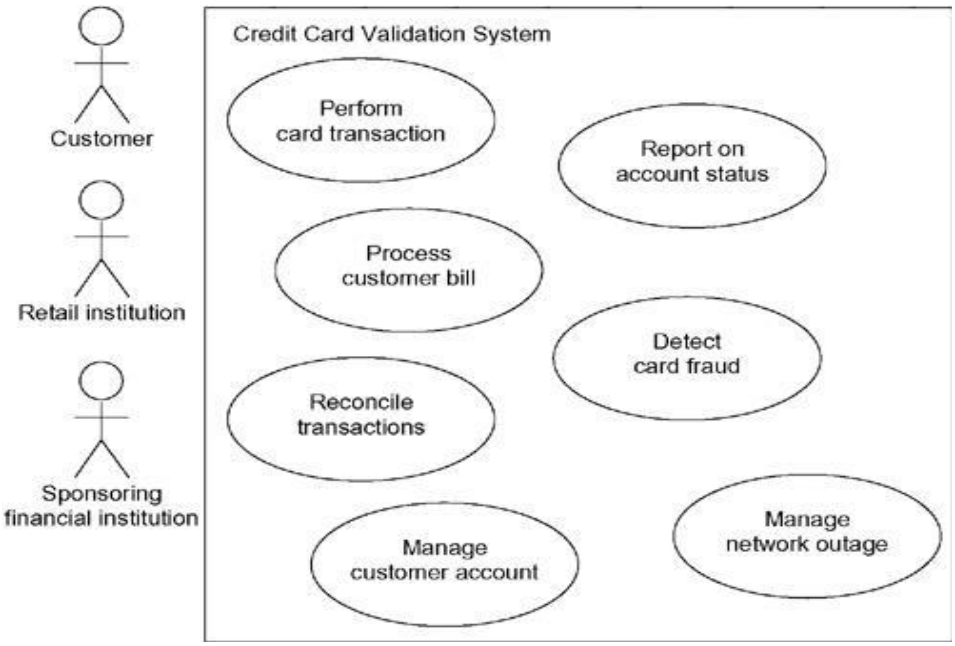


Fig: Modeling the Requirements of a System

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

- Activity diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. An activity diagram is essentially a flowchart, showing flow of control from activity to activity.
- You use activity diagrams to model the dynamic aspects of a system.
- An activity diagram shows the flow from activity to activity. An activity 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.
- Activity diagrams may stand alone to visualize, specify, construct, and document the dynamics of a society of objects, or they may be used to model the flow of control of an operation.
- Whereas interaction diagrams emphasize the flow of control from object to object, activity diagrams emphasize the flow of control from activity to activity.

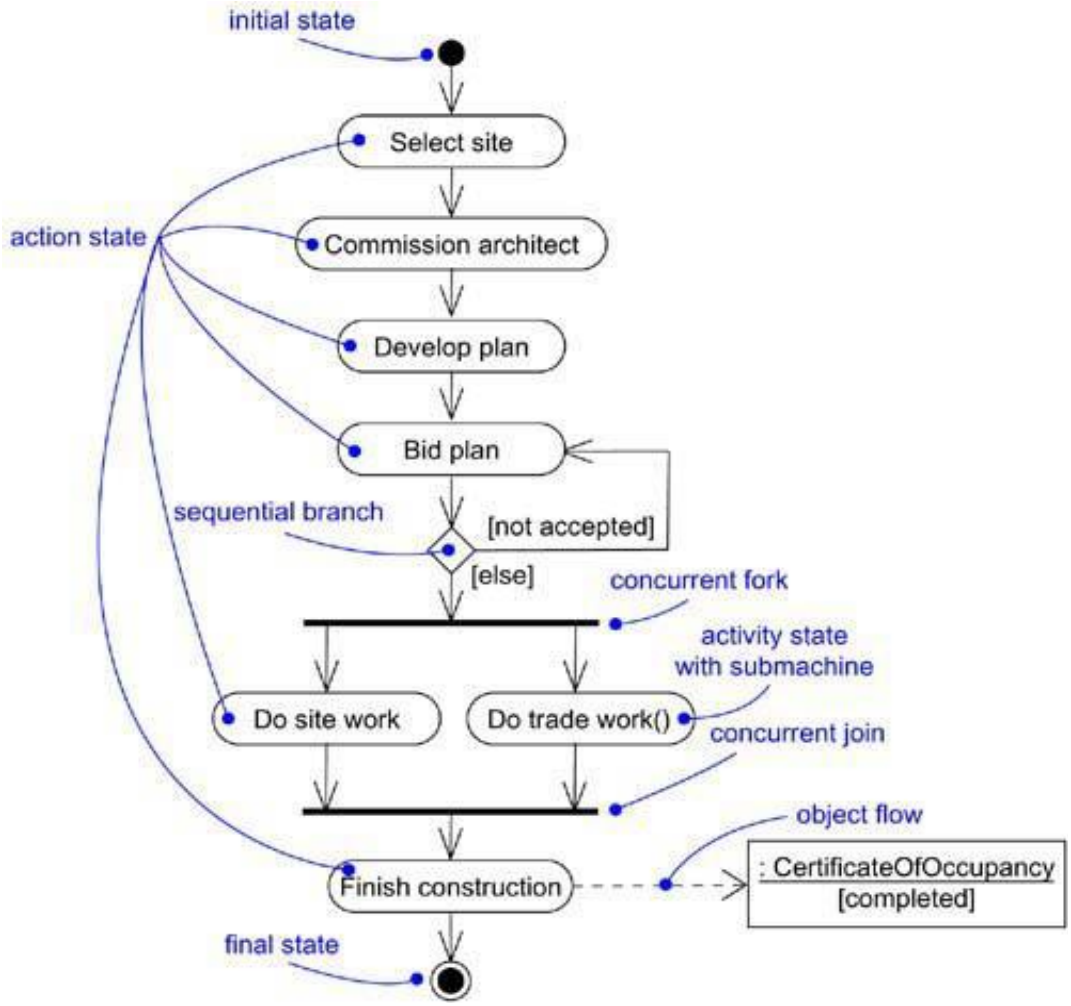


Fig: Activity Diagrams

Terms and Concepts

- An activity diagram shows the flow from activity to activity.

- An activity 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.

Common Properties

- An activity diagram is just a special kind of diagram and shares the same common properties as do all other diagrams a name and graphical contents that are a projection into a model. What distinguishes an interaction diagram from all other kinds of diagrams is its content.

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.

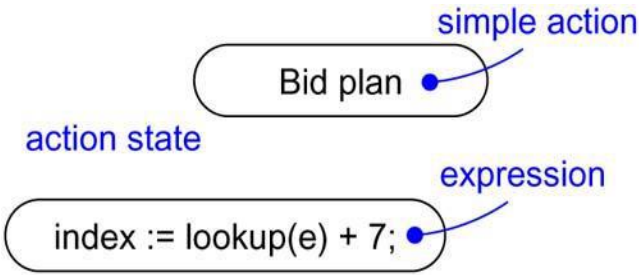


Fig: Action States

- 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.

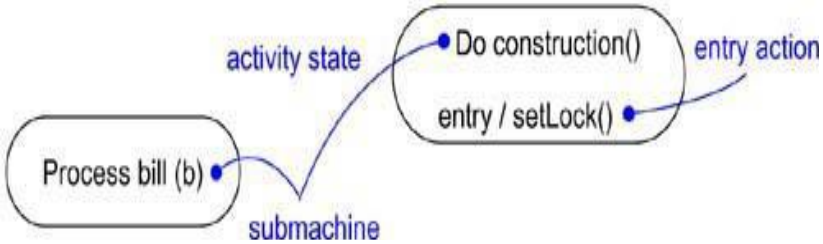


Fig: Activity States

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

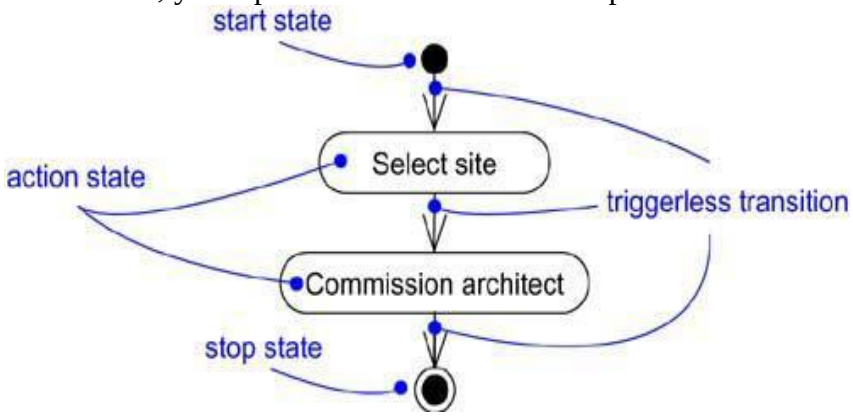


Fig: Triggerless Transitions

Branching

- As in a flowchart, you can include a branch, which specifies alternate paths taken based on some Boolean expression.
- We represent a branch as a diamond. A branch may have one 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.

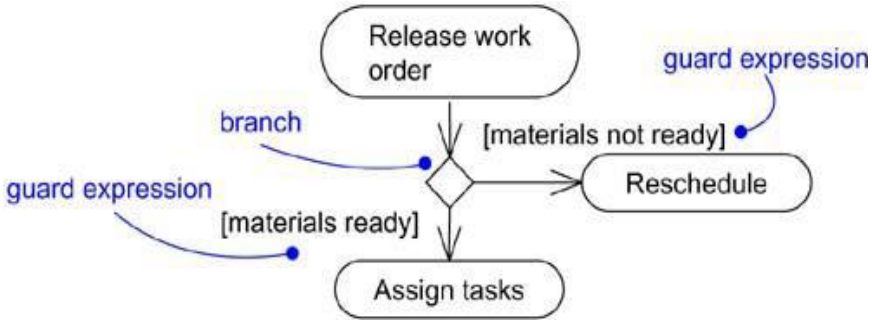


Fig: Branching

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.

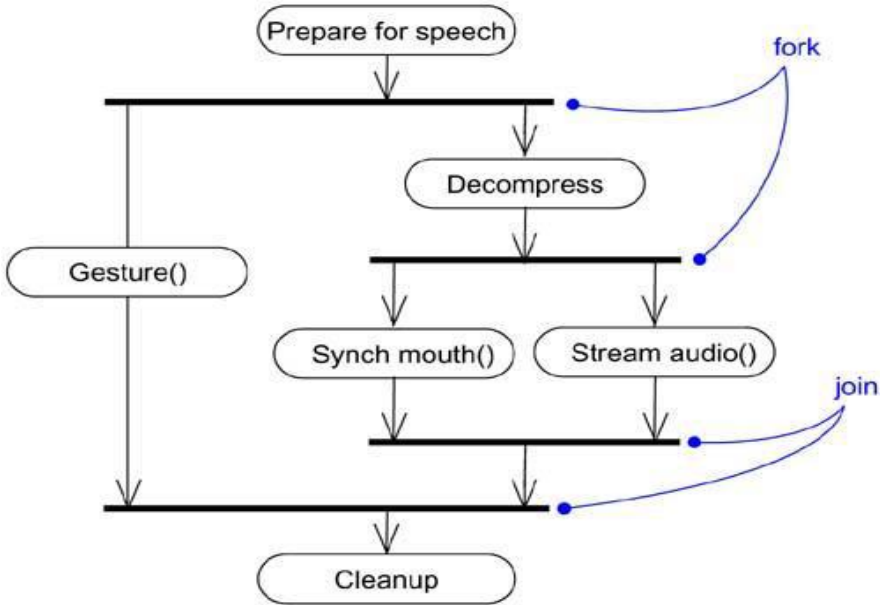


Fig: Forking and Joining

- 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.

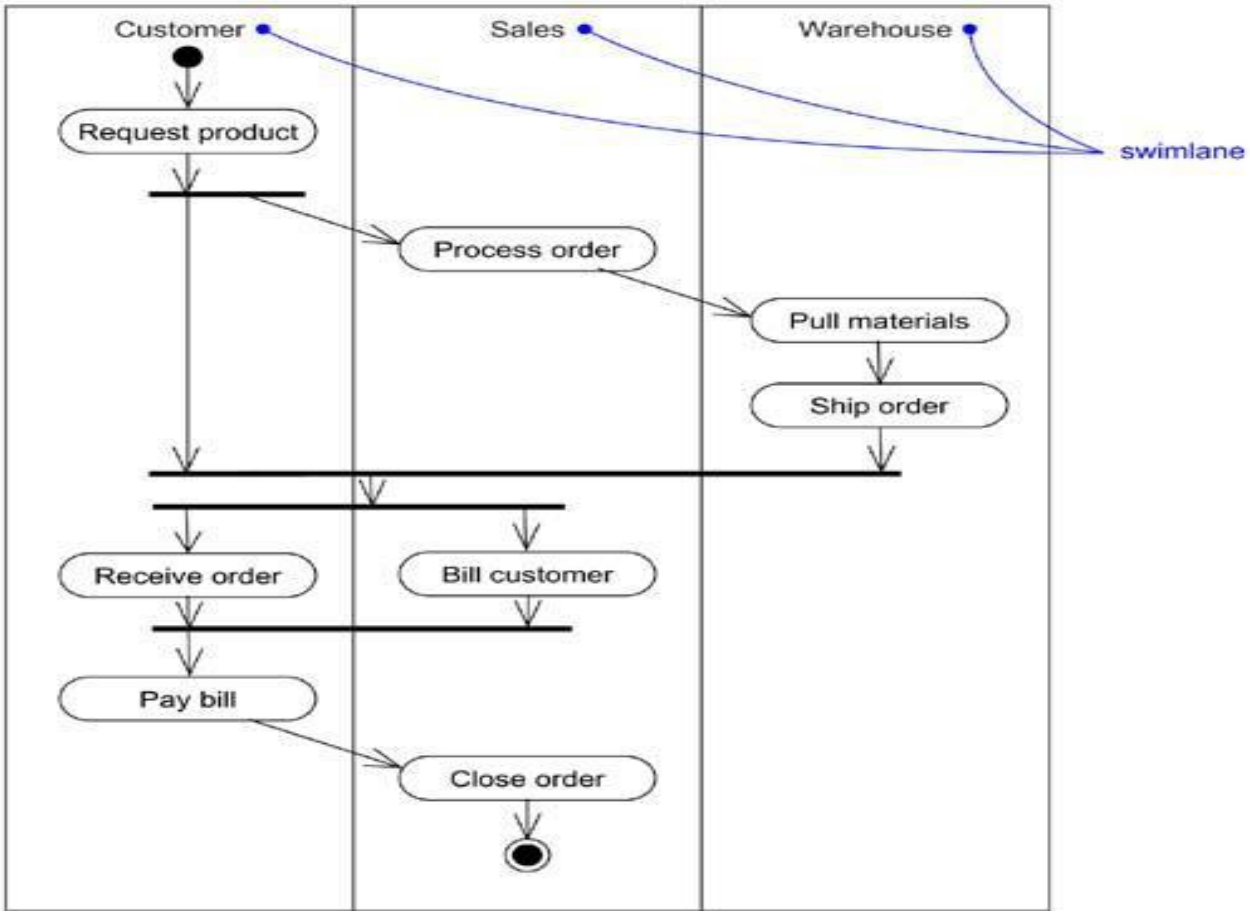


Fig: Swimlanes

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.

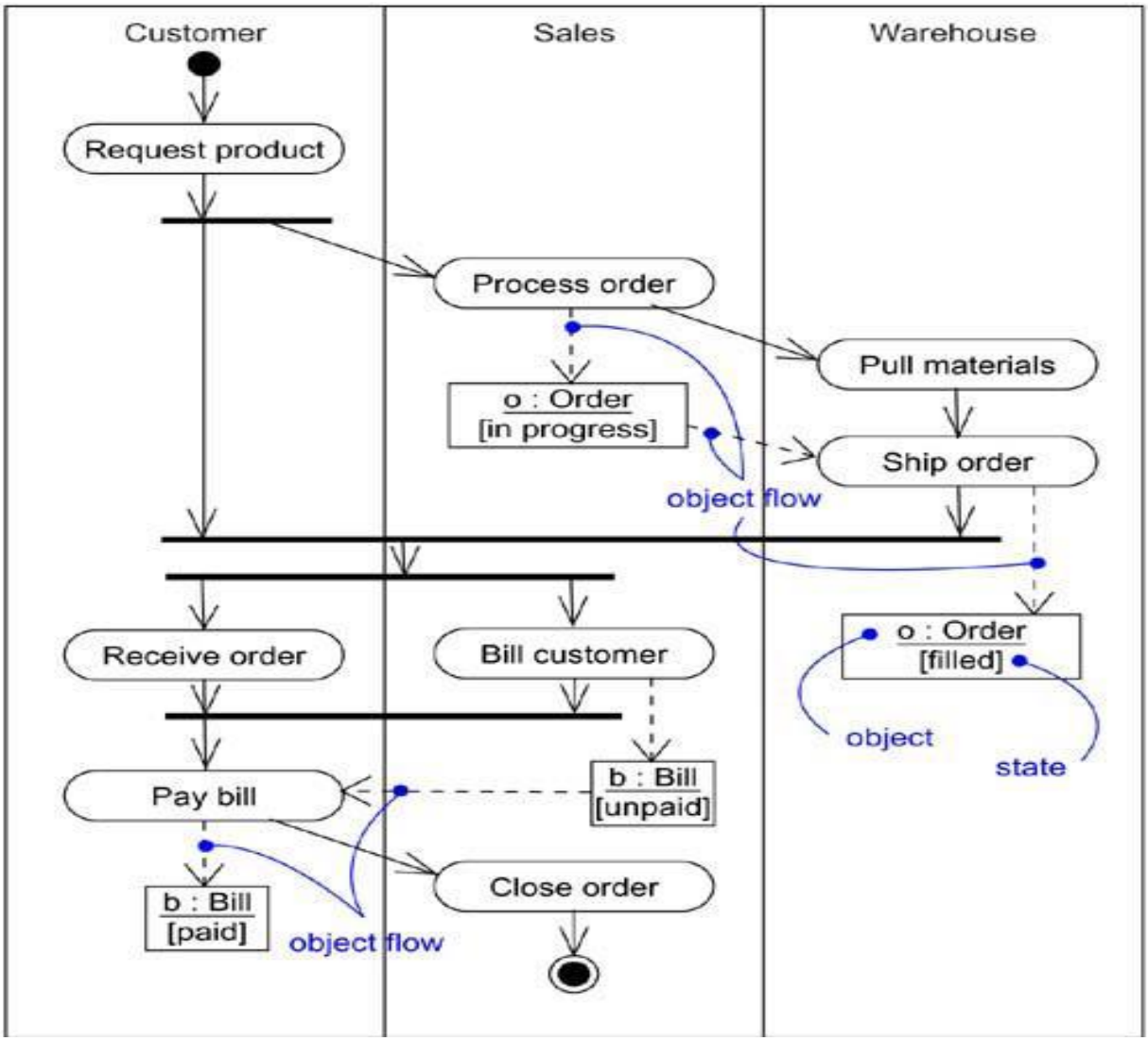


Fig: Object Flow

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

1. To model a workflow

- Here you'll focus on activities as viewed by the actors that collaborate with the system. Workflows often lie on the fringe of software-intensive systems and are used to visualize, specify, construct, and document business processes that involve the system you are developing. In this use of activity diagrams, modeling object flow is particularly important.

2. To model an operation

- Here you'll use activity diagrams as flowcharts, to model the details of a computation. In this use of activity diagrams, the modeling of branch, fork, and join states is particularly important. The context of an activity diagram used in this way involves the parameters of the operation and its local objects.

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.

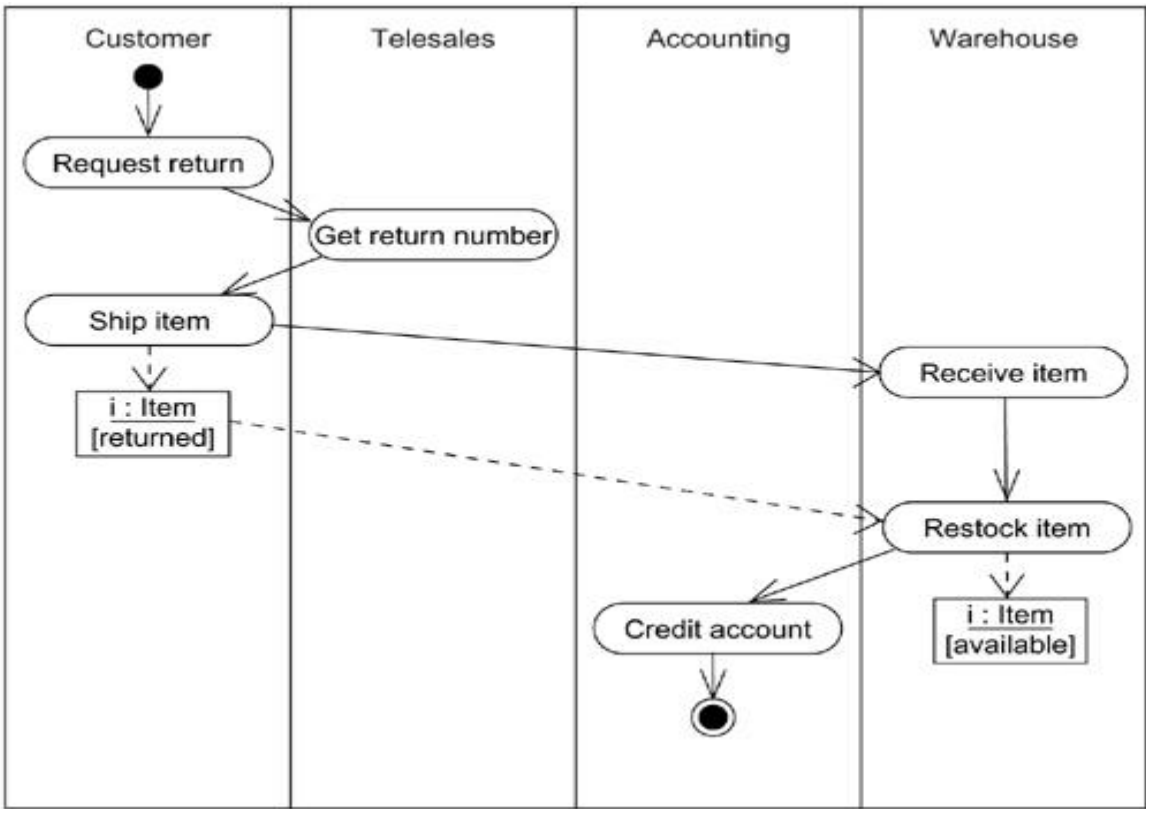


Fig: Modeling a Workflow

- The above figure shows an activity diagram for a retail business, which specifies the Workflow involved when a customer returns an item from a mail order. Work starts with the Customer action Request return and then flows through Telesales (Get return number), back to the Customer (Ship item), then to the Warehouse (Receive item then Restock item), finally ending in Accounting (Credit account). As the diagram indicates, one significant object (i, an instance of Item) also flows the process, changing from the returned to the available state.

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.

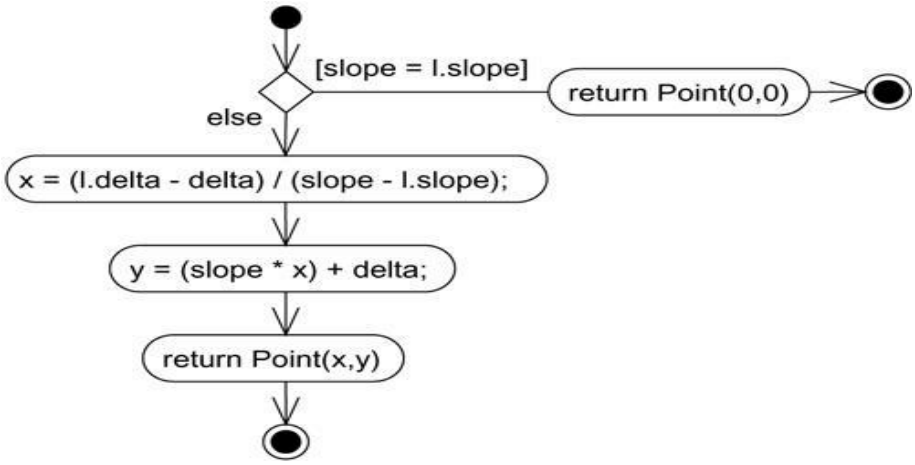


Fig: Modeling an Operation

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, state machines, processes and threads, time and space, state chart diagrams.

UNIT - IV

Events and Signals

- In the real world, things happen. Not only do things happen, but lots of things may happen at the same time, and at the most unexpected times. "Things that happen" are called events, and each one represents the specification of a significant occurrence that has a location in time and space.

- In the context of state machines, you use events to model the occurrence of a stimulus that can trigger a state transition. Events may include signals, calls, the passing of time, or a change in state.

- Events may be synchronous (occurring at the same time.) or asynchronous, so modeling events is wrapped up in the modeling of processes and threads.

- In the UML, each thing that happens is modeled as an event. An event is the specification of a significant occurrence that has a location in time and space. A signal, the passing of time, and a change of state are asynchronous events, representing events that can happen at arbitrary times.

- Calls are generally synchronous events, representing the invocation of an operation.

- The UML provides a graphical representation of an event, as Figure shows. This notation permits you to visualize the declaration of events (such as the signal OffHook), as well as the use of events to trigger a state transition

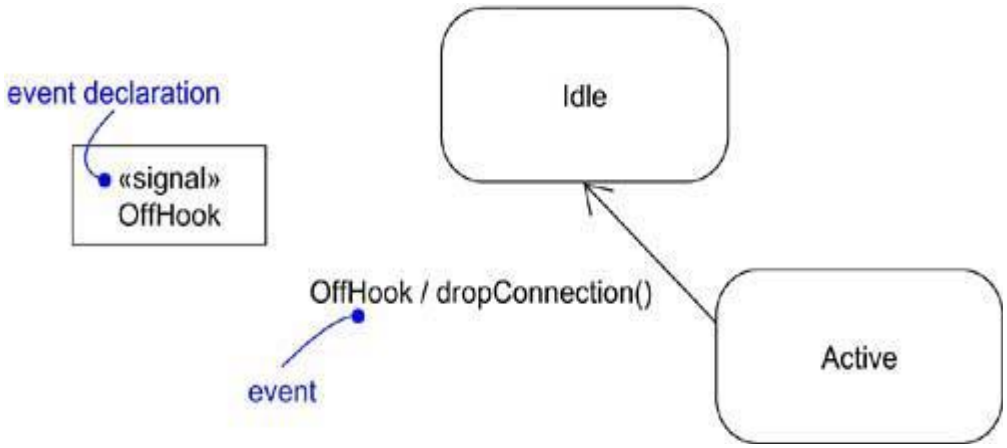


Fig: Events

Terms and Concepts

- An event is the specification of a significant occurrence that has a location in time and space.

- In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition.

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

Kinds of Events

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

- In the UML, you can model four kinds of events:
  - Signals
  - Calls
  - The passing of time,
  - 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.

- 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 the UML, as the below figure shows, you model signals (and exceptions) as stereotyped classes. You can use a dependency, stereotyped as send, to indicate that an operation sends a particular signal.

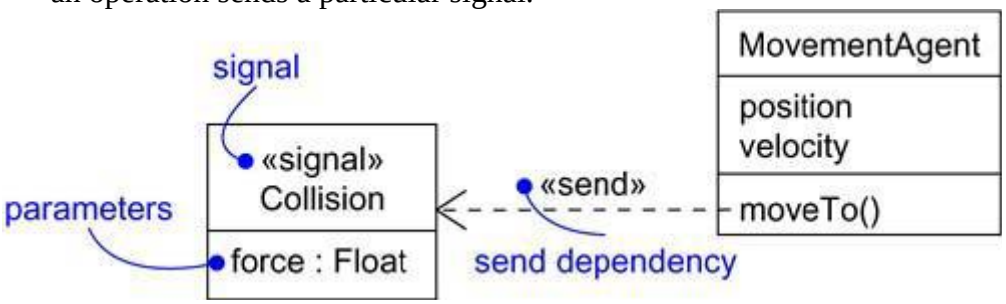


Fig: Signals

Call Events

- 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, in general, 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.

- As figure shows, 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.

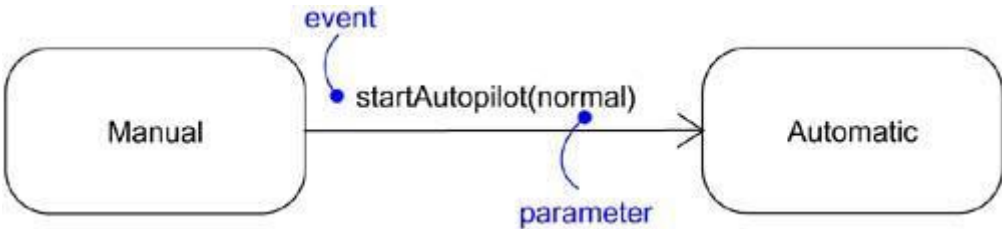


Fig: Call Events

Time and Change Events

- A time event is an event that represents the passage of time.
- As the below figure shows, in the UML you model a time event by using the keyword after followed by some expression that evaluates to a period of time.
- Such expressions can be simple (for example, after 2 seconds) or complex (for example, after 1 ms since exiting Idle).
- Unless you specify it explicitly, the starting time of such an expression is the time since entering the current state.

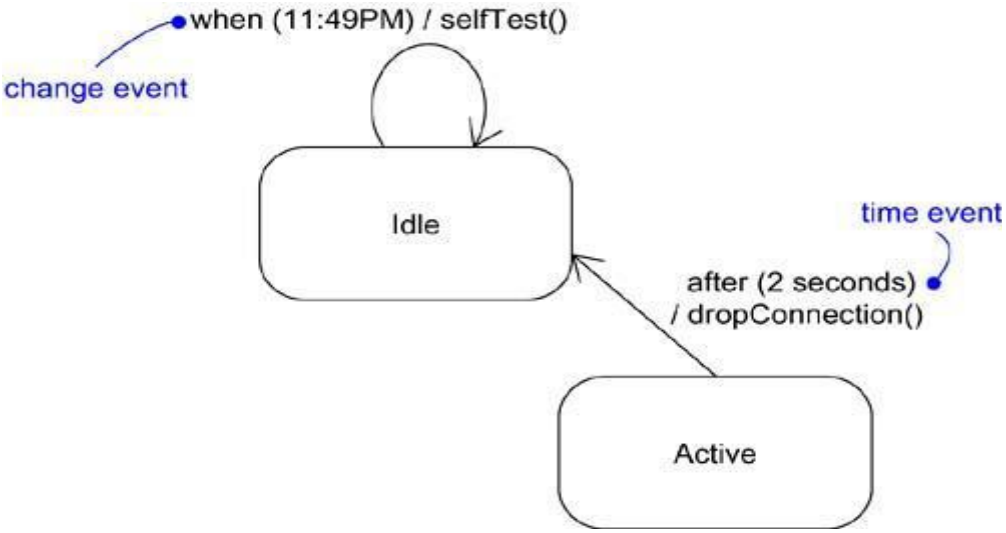


Fig: Time and Change Events

- A change event is an event that represents a change in state or the satisfaction of some condition.
- As the above figure shows, 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 (for example, when altitude < 1000).

Sending and Receiving Events

- Signal events and call events involve at least two objects:

- The object that sends the signal or invokes the operation, and the object to which the event is directed.
- Because signals are asynchronous, and because asynchronous calls are themselves signals, the semantics of events interact with the semantics of active objects and passive objects.
- In the UML, you model the call events that an object may receive as operations on the class of the object.
- In the UML, you model the named signals that an object may receive by naming them in an extra compartment of the class, as shown in below figure.

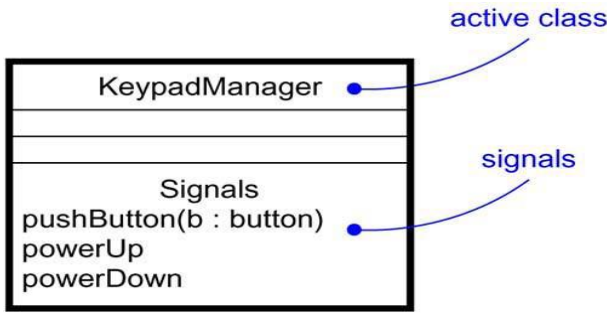


Fig: Signals and Active Classes

Common Modeling Techniques

Modeling a Family of Signals

To model a family of signals

- Consider all the different kinds of signals to which a given set of active objects may respond.
- Look for the common kinds of signals and place them in a generalization/specialization hierarchy using inheritance. 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.

- The below figure models a family of signals that may be handled by an autonomous robot. Note that the root signal (RobotSignal) is abstract, which means that there may be no direct instances. This signal has two immediate concrete specializations (Collision and HardwareFault), one of which (HardwareFault) is further specialized. Note that the Collision signal has one parameter

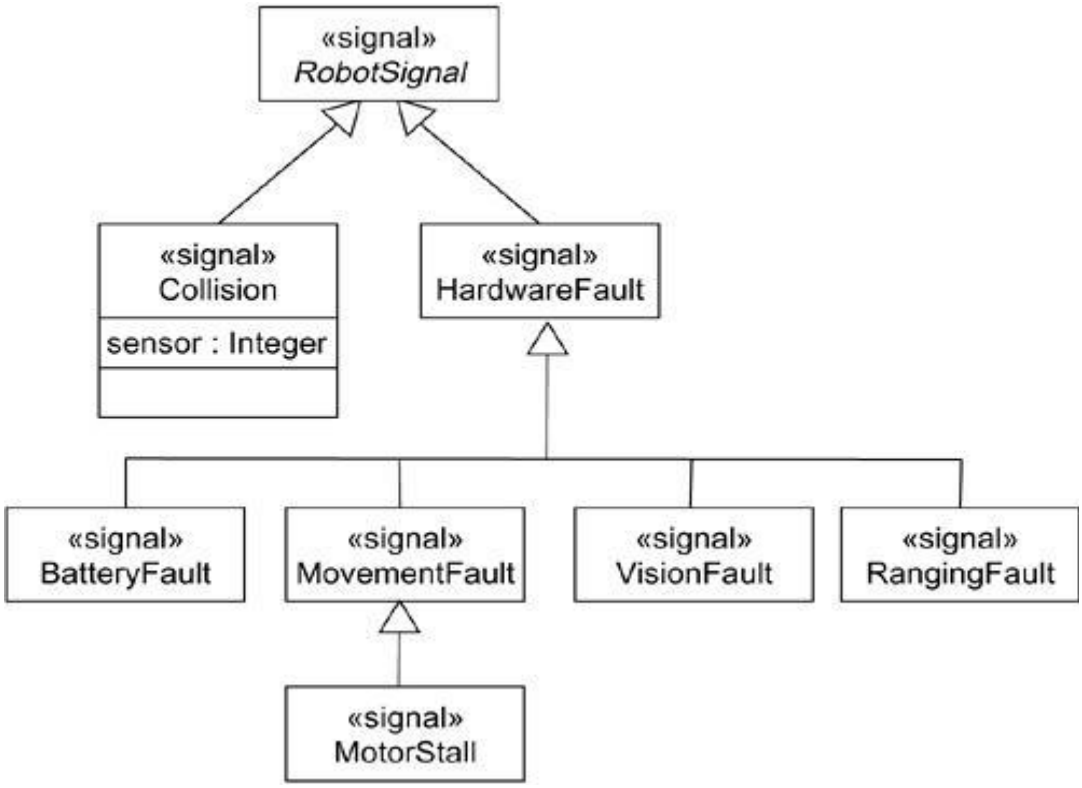


Fig: Modeling Families of Signals

Modeling Exceptions

- In the UML, exceptions are kinds of signals, which you model as stereotyped classes. Exceptions May be attached to specification operations.
- You 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.

The below figure models a hierarchy of exceptions that may be raised by a standard library of container classes, such as the template class Set. This hierarchy is headed by the abstract signal Exception and includes three specialized exceptions: Duplicate, Overflow, and Underflow. As shown, the add operation raises Duplicate and Overflow exceptions, and the remove operation raises only the Underflow exception. Alternatively, you could have put these dependencies in the background by naming them in each operation's specification. Either way, by knowing which exceptions each operation may send, you can create clients that use the Set class correctly.

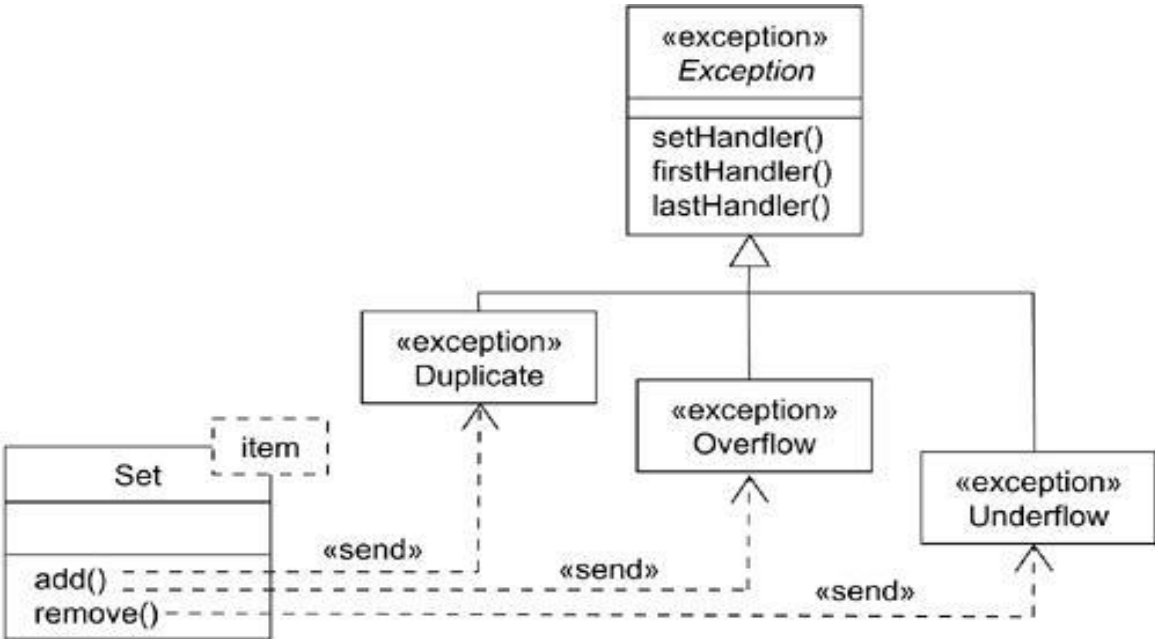


Fig: Modeling Exceptions

State Machines

- Using an interaction, you can model the behavior of a society of objects that work together.
- Using a state machine, you can model the behavior of an individual object.
- 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.

- You use state machines to model the dynamic aspects of a system.

- You can visualize a state machine in two ways: by emphasizing the flow of control from activity to activity (using activity diagrams), or by emphasizing the potential states of the objects and the transitions among those states (using statechart diagrams).

- Well-structured state machines are like well-structured algorithms: They are efficient, simple, adaptable, and understandable.

- The UML provides a graphical representation of states, transitions, events, and actions, as Below figure shows. This notation permits you to visualize the behavior of an object in a way that lets you emphasize the important elements in the life of that object.

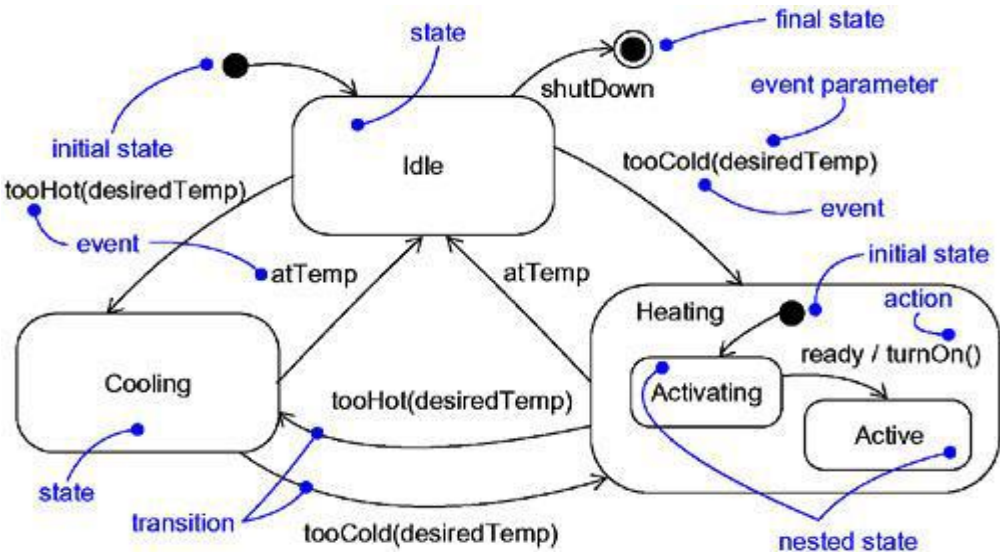


Fig: State Machines

Terms and Concepts

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

- A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.

- 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.

- Graphically, a state is rendered as a rectangle with rounded corners. A transition is rendered as a solid directed line.

States

- A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.

- An object remains in a state for a finite amount of time. For example, a Heater in a home might be in any of four states: Idle (waiting for a command to start heating the house), Activating (its gas is on, but it's waiting to come up to temperature), Active

(its gas and blower are both on), and ShuttingDown (its gas is off but its blower is on, flushing residual heat from the system).

Astate 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

As below figure shows, you represent a state as a rectangle with rounded corners.

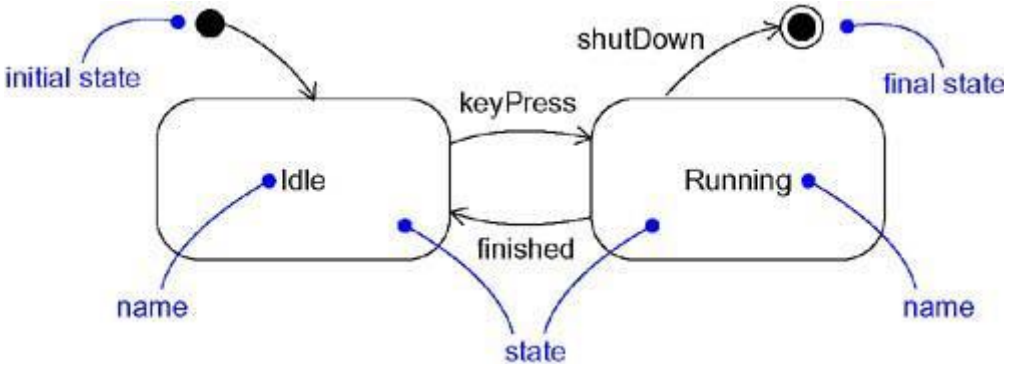


Fig: States

Initial and Final States

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

Transitions

- A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- 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.
- For example, a Heater might transition from the Idle to the Activating state when an event such as tooCold (with the parameter desiredTemp) occurs.

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

- As below figure shows, 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.

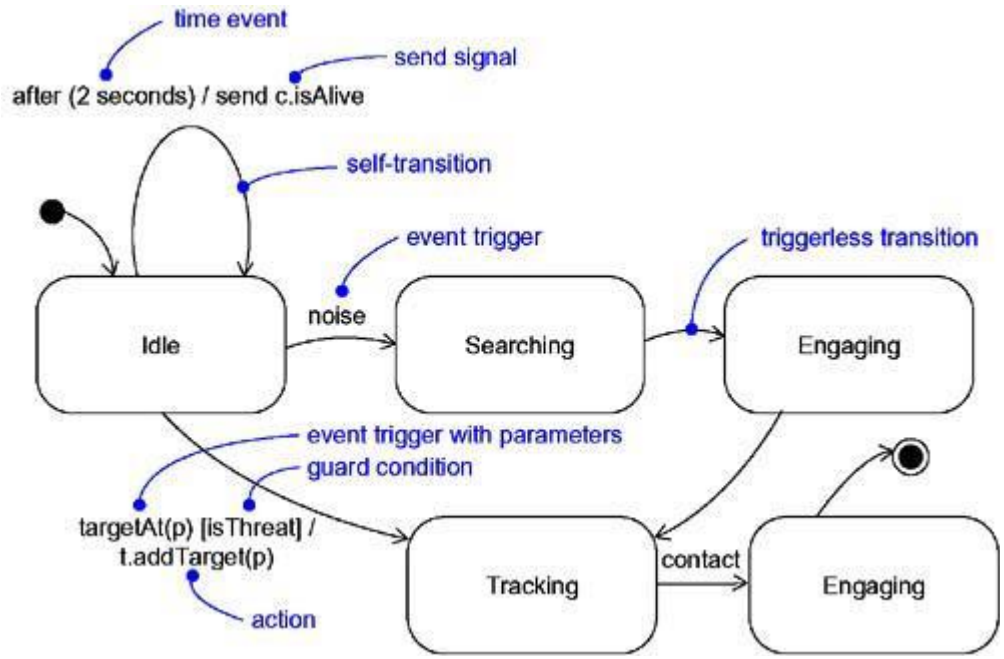


Fig: Transitions

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.

- As shown in the previous figure, events may include signals, calls, the passing of time, or a change in state.

- 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

- As the previous figure shows, a guard condition is rendered as a Boolean expression enclosed in square brackets and placed after the trigger event.

- A guard condition is evaluated only after the trigger event for its transition occurs.

Action

- An action is an executable atomic computation. Actions may include operation calls (to the object that owns the state machine, as well as to other visible objects), the creation or destruction of another object, or the sending of a signal to an object.

- As the previous figure shows, there's a special notation for sending a signal• the signal name is prefixed with the keyword send as a visual cue.

- An action is atomic, meaning that it cannot be interrupted by an event and therefore runs to completion. This is in contrast to an activity, which may be interrupted by other events.

Advanced States and Transitions

- The UML's state machines have a number of advanced features that help you to manage complex behavioral models

- Some of these advanced features include entry and exit actions, internal transitions, activities, and deferred events.

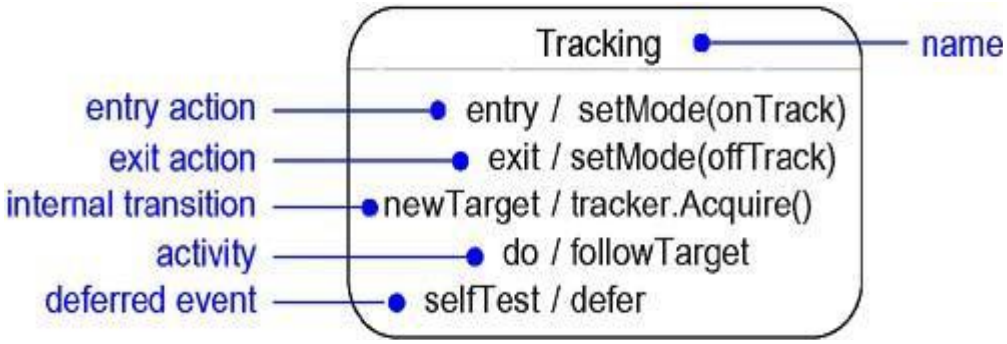


Fig A: Advanced States and Transitions

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.

- As Figure A: shows, the 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.

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

- As Figure A shows, the 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 handledwithout 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.

- As Figure A shows, 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 (such as followTarget). You can also specify a sequence of actions• for example, do / op1(a); op2(b); op3(c).

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.

- As you can see in the previous figure, you can specify a deferred event by listing the event with the special action defer. In this example, selfTest events may happen while in the Tracking state, but they are held until the object is in the Engaging state, at which time it appears as if they just occurred.

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 substate is a state that's nested inside another one.
- For example, a Heater might be in the Heating state, but also while in the Heating state, there might be a nested state called Activating. In this case, it's proper to say that the object is both Heating and Activating.
- A simple state is a state that has no substructure.
- 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

- Using sequential substates, there's a simpler way to model this problem,
- The below figure B shows. Here, the Active state has a substructure, containing the substates Validating, Selecting, Processing, and Printing.
- The state of the ATM changes from Idle to Active when the customer enters a credit card in the machine.
- On entering the Active state, the entry action readCard is performed. Starting with the initial state of the substructure, control passes to the Validating state, then to the Selecting state, and then to the Processing state.
- After Processing, control may return to Selecting (if the customer has selected another transaction) or it may move on to Printing. After Printing, there's a triggerless transition back to the Idle state.
- Notice that the Active state has an exit action, which ejects the customer's credit card.

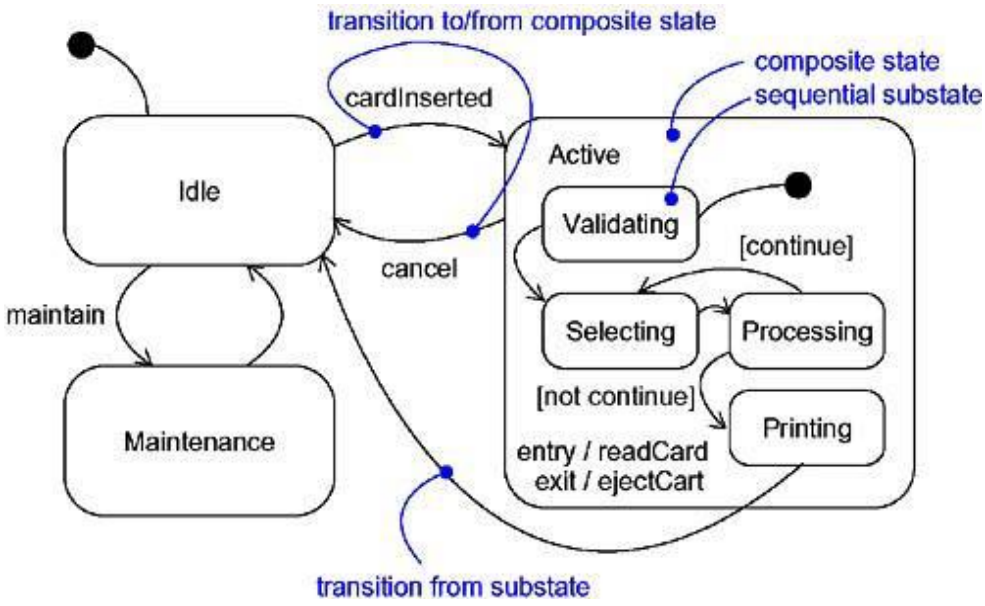


Fig B: Sequential Substates

History States

- 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.
- The below figure shows, you represent a shallow history state as a small circle containing the symbol H.

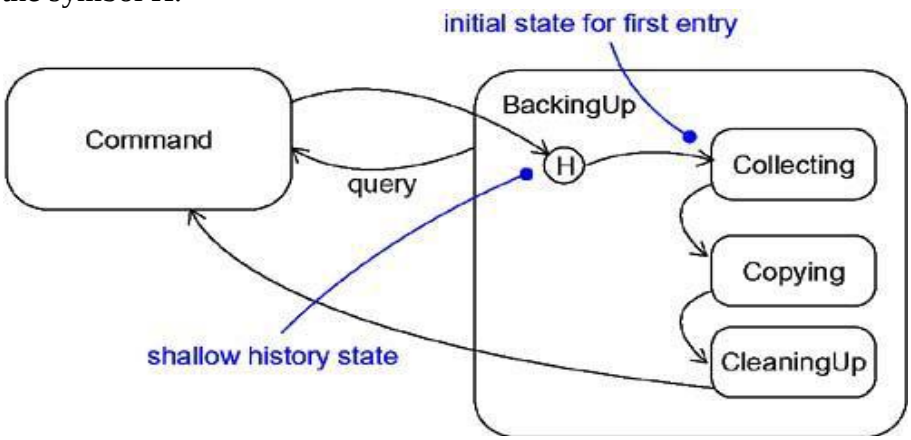


Fig: History State

Concurrent Substates

- Sequential substates are the most common kind of nested state machine.
- 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 enclosing object.
- For example Maintenance state from below figure. Maintenance is decomposed into two concurrent substates,
- Testing and Commanding, shown by nesting them in the Maintenance state but separating them from one another with a dashed line.
- Each of these concurrent substates is further decomposed into sequential substates.
- When control passes from the Idle to the Maintenance state, control then forks to two concurrent flows the enclosing object will be in the Testing state and the Commanding state.
- Furthermore, while in the Commanding state, the enclosing object will be in the Waiting or the Command state.

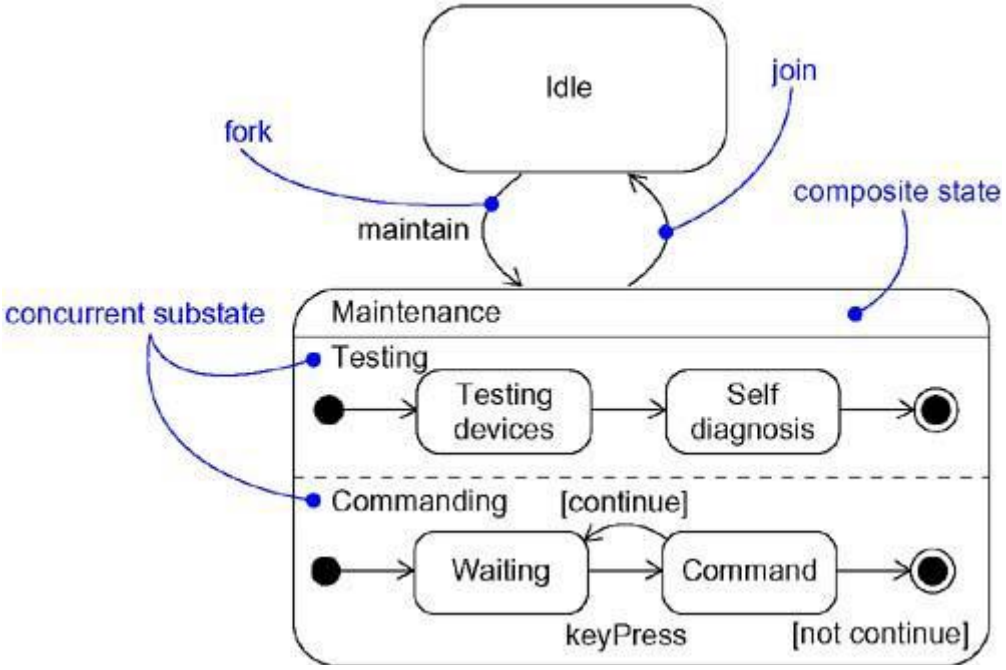


Fig: Concurrent Substates

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.

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

- If the context is the system as a whole, narrow your focus to one behavior of the system. Theoretically, every object in the system may be a participant in a model of the system's lifetime, and except for the most trivial systems, a complete model would be intractable.
- Establish the initial and final states for the object. To guide the rest of your model, possibly state the pre- and post conditions of the initial and final states, respectively.

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

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

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

- Expand these states as necessary by using 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.

- For example, the below figure shows the state machine for the controller in a home security system, which is responsible for monitoring various sensors around the perimeter of the house.

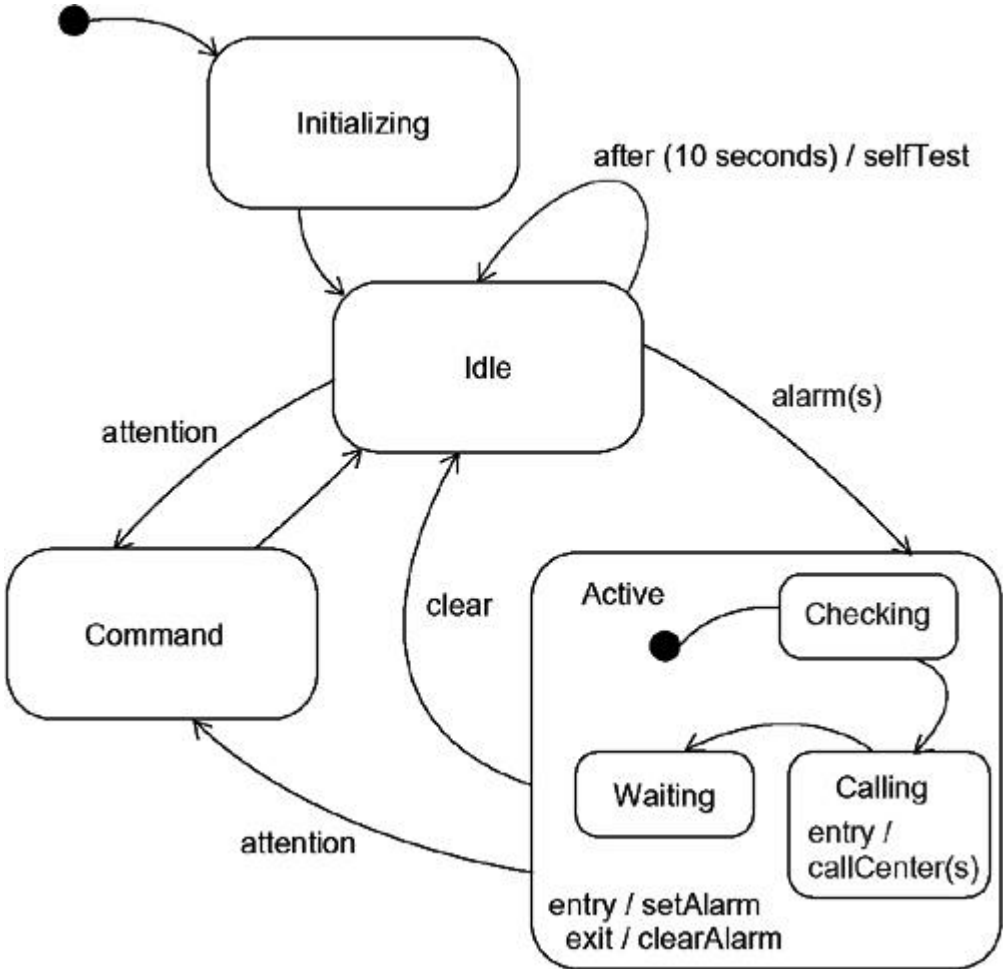


Fig: Modeling the Lifetime of an Object

Processes and Threads

- In the UML, you model each independent flow of control as an active object that represents a process or thread that can initiate control activity.

- 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.

- In the UML, each independent flow of control is modeled as an active object. An active object is a process or thread that can initiate control activity. As for every kind of object, an active object is an instance of a class. In this case, an active object is an instance of an active class.

- The UML provides a graphical representation of an active class, the below figure shows.

- Active classes are kinds of classes, so have all the usual compartments for class name, attributes, and operations. Active classes often receive signals, which you typically enumerate in an extra compartment.

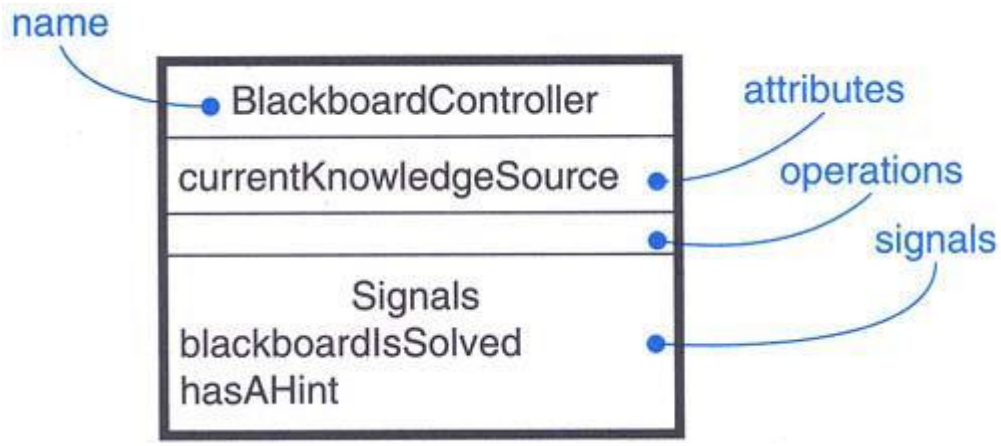


Fig: Active Class

Terms and Concepts

- 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 (and also appear as sequences in interaction diagrams).

Classes and Events

- Active classes are just classes

- 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 (including aggregation) 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

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, which means that it is a thing known to the operating system itself and runs in an independent address space. Under most operating systems, such as Windows and Unix, each program runs as a process in its own address space

- A thread is lightweight. It may be known to the operating system itself. More often, it is hidden inside a heavier-weight process and runs inside the address space of the enclosing process. In Java, for example, a thread is a child of the class Thread.

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.

- First, 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.

- Second, 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.
    - **First,** one active object might synchronously call an operation of another. That kind of communication has **rendezvous semantics**, which means that the caller calls the operation; the caller waits for the receiver to accept the call; the operation is invoked; a return object (if any) is passed back to the caller.

    - **Second,** one active object might asynchronously send a signal or call an operation of another object. That kind of communication has **mailbox semantics**, which means that the caller sends the signal or calls the operation and then continues on its independent way.
    - The two objects are not synchronized; rather, one object drops off a message for the other.

    - In the UML, you render a synchronous message as a full arrow and an asynchronous message as a half arrow, as shown below figure.

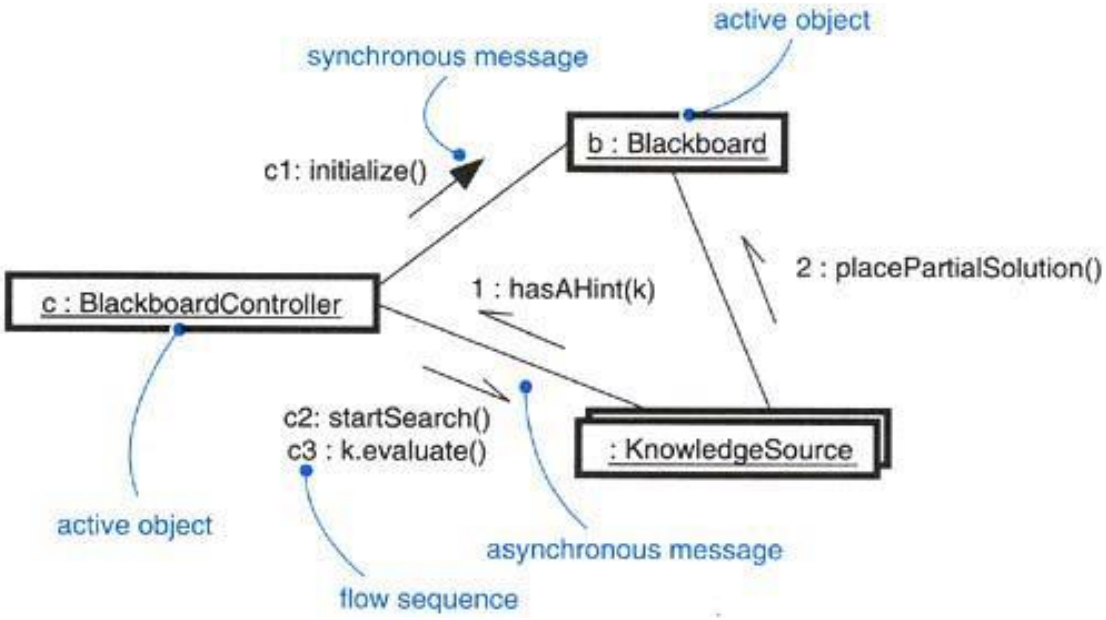


Fig: Communication

- Third, 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,

- Fourth, 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.

Common Modeling Techniques

Modeling Multiple Flows of Control

To model multiple flows of control

- Identify the opportunities for concurrent action and reify each flow as an active class. Generalize common sets of active objects into an active class. Be careful not to over engineer the process view of your system by introducing too much concurrency.

- Consider a balanced distribution of responsibilities among these active classes, then examine the other active and passive classes with which each collaborates statically.

- Ensure that each active class is both tightly cohesive and loosely coupled relative to these neighboring classes and that each has the right set of attributes, operations, and signals.

- Capture these static decisions in class diagrams, explicitly highlighting each active class.

- Consider how each group of classes collaborates with one another dynamically. Capture those decisions in interaction diagrams. Explicitly show active objects as the root of such flows. Identify each related sequence by identifying it with the name of the active object.

- Pay close attention to communication among active objects. Apply synchronous and asynchronous messaging, as appropriate.

- Pay close attention to synchronization among these active objects and the passive objects with which they collaborate. Apply sequential, guarded, or concurrent operation semantics, as appropriate.

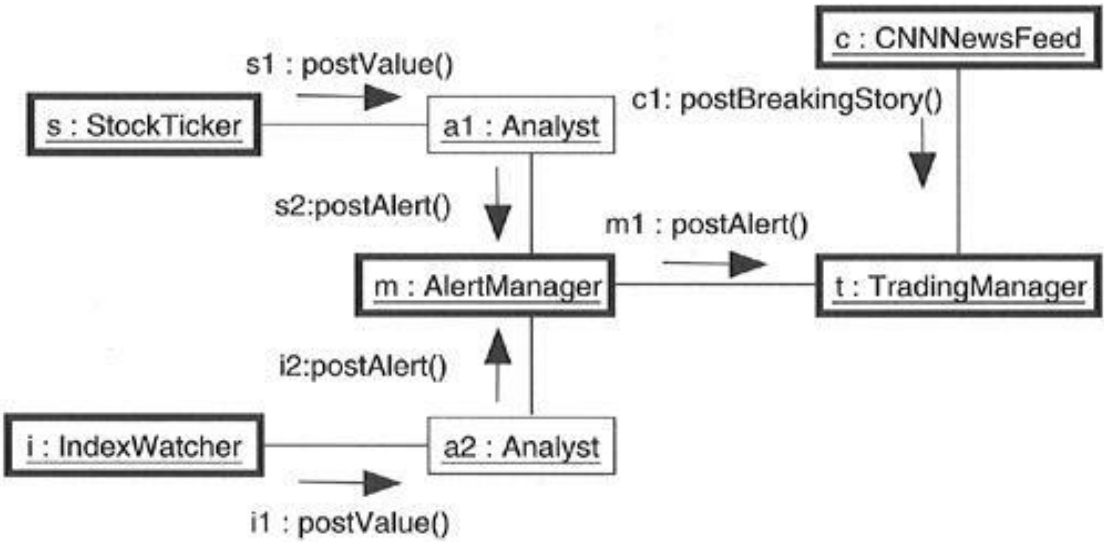


Fig: Modeling Flows of Control

Modeling Interprocess Communication

To model interprocess communication

- Consider which of these active objects represent processes and which represent threads. Distinguish them using the appropriate stereotype.

- Model messaging using asynchronous communication; model remote procedure calls using synchronous communication.

- Informally specify the underlying mechanism for communication by using notes, or more formally by using collaborations.

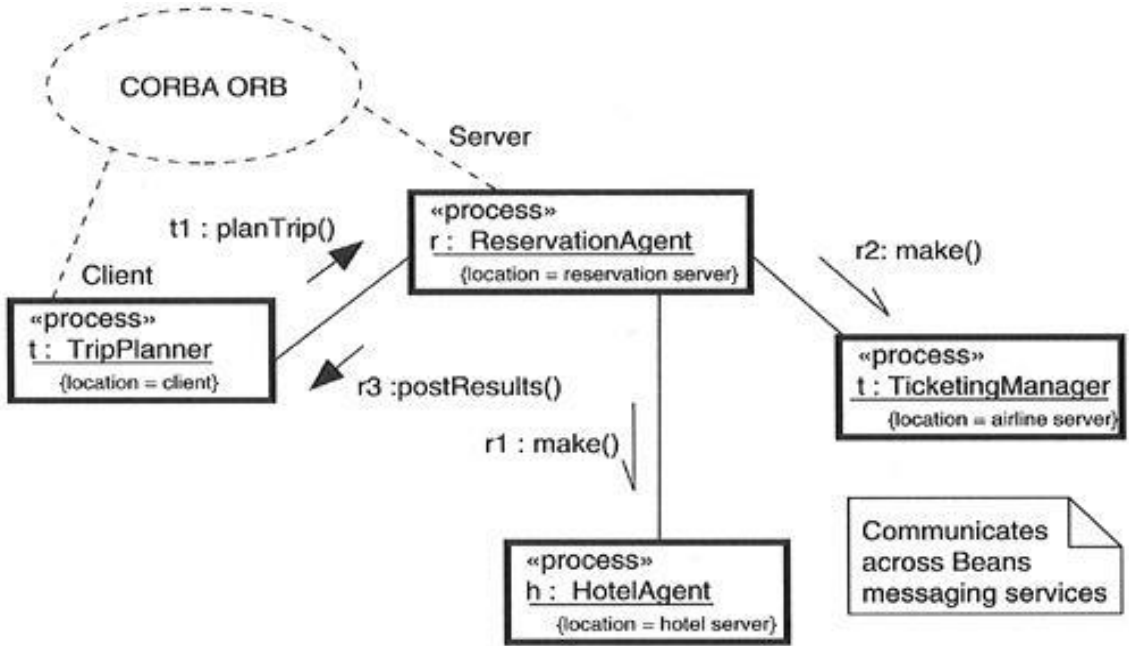


Fig: Modeling Interprocess Communication

Time and Space

- Modeling time and space is an essential element of any real time and/or distributed system.

- You use a number of the UML's features, including timing marks, time expressions, constraints, and tagged values, to visualize, specify, construct, and document these systems.

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

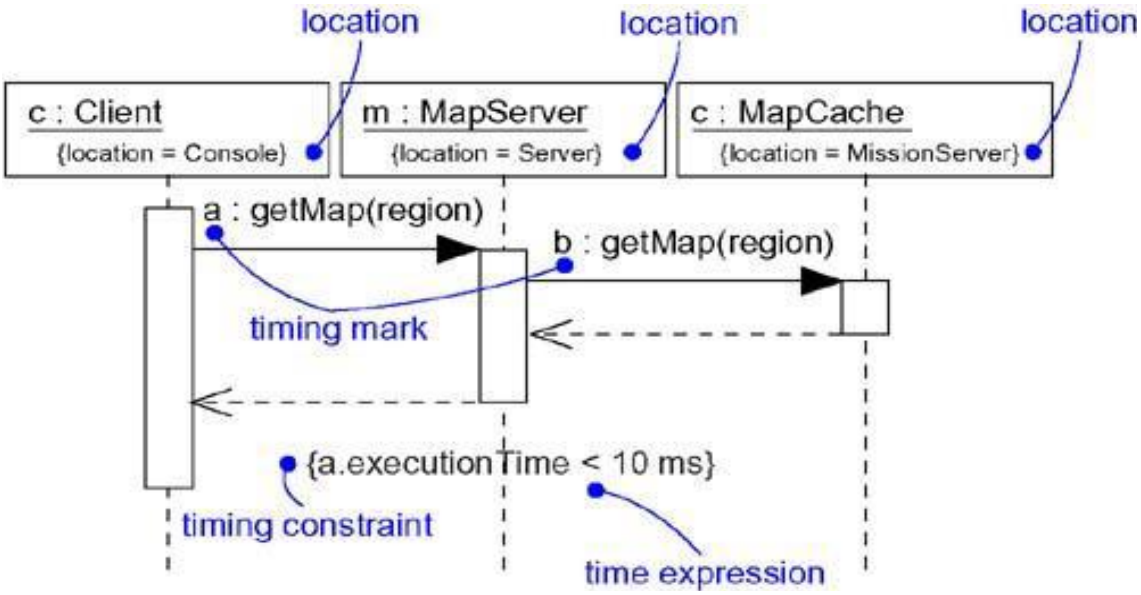


Fig: Timing Constraints and Location

Terms and Concepts

- A timing mark is a denotation for the time at which an event occurs.
- Graphically, a timing mark is formed as an expression from the name given to the message (which is typically different from the name of the action dispatched by the message).
- A time expression is an expression that evaluates to an absolute or relative value of time.
- A timing constraint is a semantic statement about the relative or absolute value of time.
- Graphically, a timing constraint is rendered as for any constraint• that is, a string enclosed by brackets and generally connected to an element by a dependency relationship.
- Location is the placement of a component on a node.
- Graphically, location is rendered as a tagged value• that is, a string enclosed by brackets and placed below an element's name, or as the nesting of components inside nodes.

Time

- Real time systems are, by their very name, time-critical systems. Events may happen at regular or irregular times; the response to an event must happen at predictable absolute times or at predictable times relative to the event itself.
- The passing of messages represents the dynamic aspect of any system, so when you model the time-critical nature of a system with the UML.

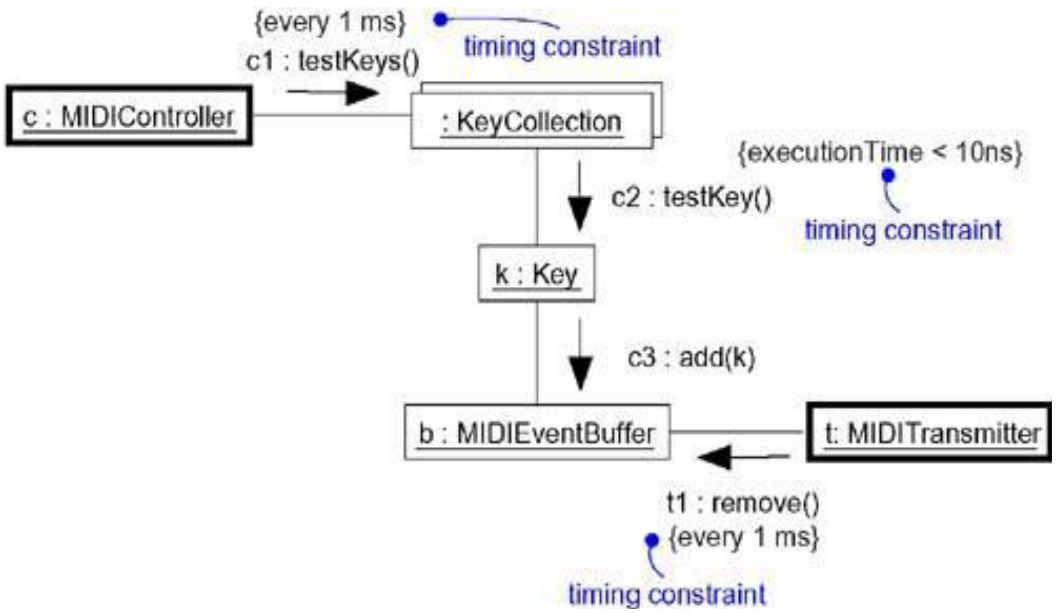


Fig: Time

Location

- In the UML, you model the deployment view of a system by using deployment diagrams that represent the topology of the processors and devices on which your system executes.

- Components such as executables, libraries, and tables reside on these nodes.

- Each instance of a node will own instances of certain components, and each instance of a component will be owned by exactly one instance of a node (although instances of the same kind of component may be spread across different nodes). For example, as Figure 23-3 shows, the executable component vision.exe may reside on the node named KioskServer.

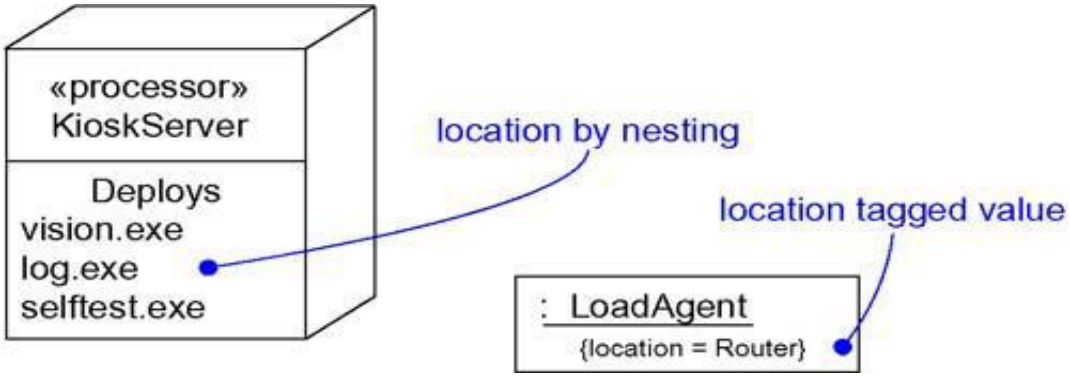


Fig: Location

Common Modeling Techniques

Modeling Timing Constraints

To model timing constraints

- For each event in an interaction, consider whether it must start at some absolute time. Model that real time property as a timing constraint on the message.

- For each interesting sequence of messages in an interaction, consider whether there is an associated maximum relative time for that sequence. Model that real time property as a timing constraint on the sequence.

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

- For example, as shown in figure, the left-most constraint specifies the repeating start time the call event refresh. Similarly, the center timing constraint specifies the maximum duration for calls to getImage. Finally, the right-most constraint specifies the time complexity of the call event getImage.

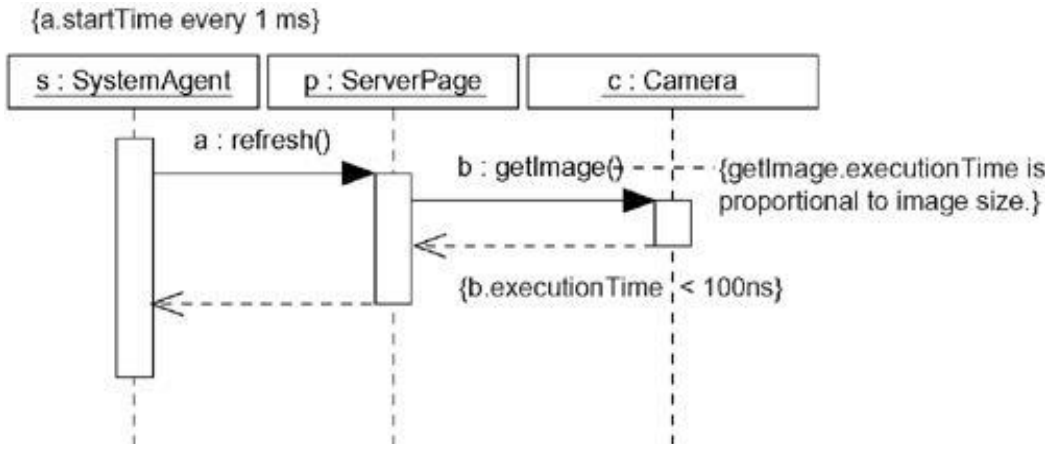


Fig: Modeling Timing Constraint

Modeling the Distribution of Objects

To model the distribution of objects

- For each interesting class of objects in your system, consider its locality of reference. In other words, consider all its neighbors and their locations.
- A tightly coupled locality will have neighboring objects close by; a loosely coupled one will have distant objects (and thus, there will be latency in communicating with them).
- Tentatively allocate objects closest to the actors that manipulate them.
- Next consider patterns of interaction among related sets of objects. Co-locate sets of objects that have high degrees of interaction, to reduce the cost of communication.
- Partition sets of objects that have low degrees of interaction.
- Next consider the distribution of responsibilities across the system. Redistribute your objects to balance the load of each node.
- Consider also issues of security, volatility, and quality of service, and redistribute your objects as appropriate.
- Render this allocation in one of two ways:
    1. By nesting objects in the nodes of a deployment diagram
    2. By explicitly indicating the location of the object as a tagged value
- Below figure provides an object diagram that models the distribution of certain objects in a retail system. The value of this diagram is that it lets you visualize the physical distribution of certain key objects. As the diagram shows, two objects reside on a Workstation (the Order and Sales objects), two objects reside on a Server (the ObserverAgent and the Product objects), and one object resides on a DataWarehouse (the ProductTable object).

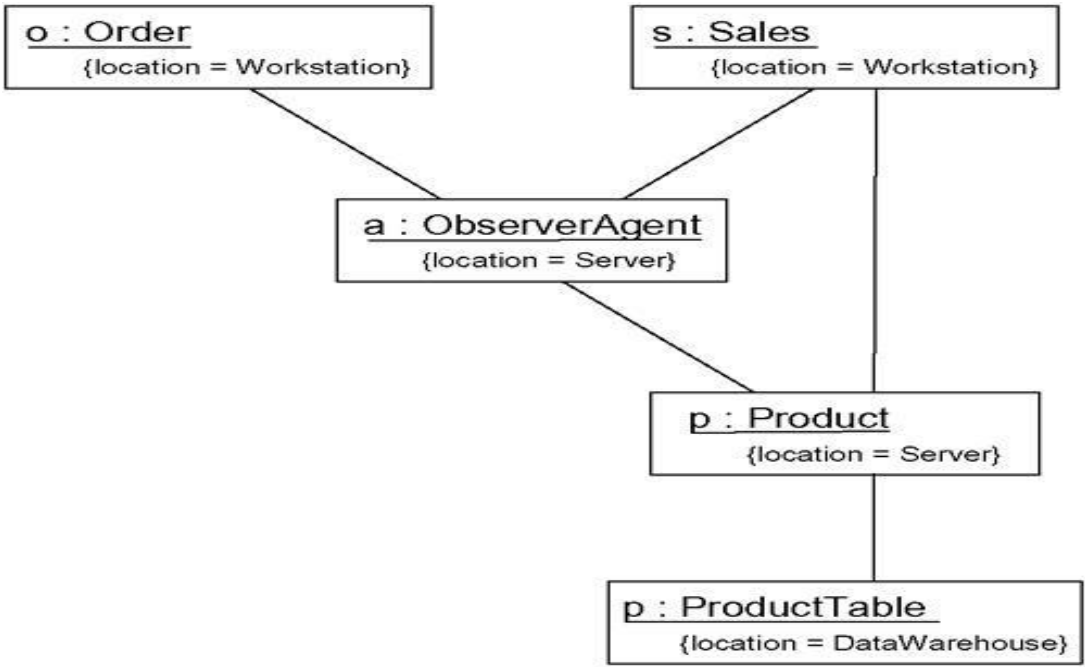


Fig: Modeling the Distribution of Objects

Modeling Objects that Migrate

To model the migration of objects

- Select an underlying mechanism for physically transporting objects across nodes.
- Render the allocation of an object to a node by explicitly indicating its location as a tagged value.
- Using the become and copy stereotyped messages, render the allocation of an object to a new node.
- Consider the issues of synchronization (keeping the state of cloned objects consistent) and identity (preserving the name of the object as it moves).
- The below figure provides a collaboration diagram that models the migration of a Web agent that moves from node to node, collecting information and bidding on resources in order to automatically deliver a lowest-cost travel ticket. Specifically, this diagram shows an instance (named t) of the class TravelAgent migrating from one server to another. Along the way, the object interacts with anonymous Auctioneer instances at each node, eventually delivering a bid for the Itinerary object, located on the client server.

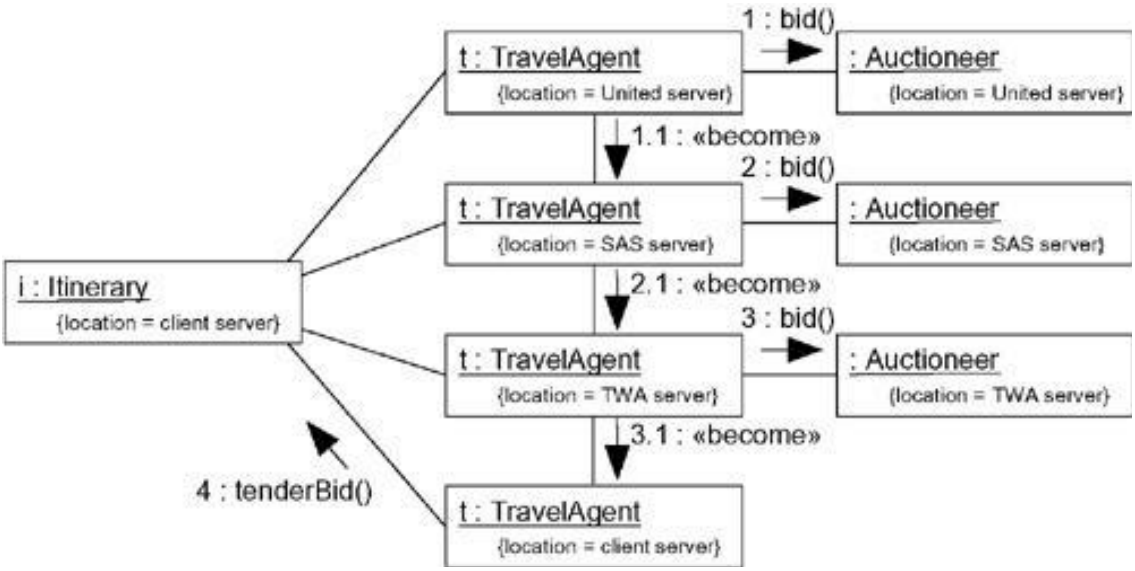


Fig: Modeling Objects that Migrate

State chart Diagrams

- Statechart diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems.
- A statechart diagram shows a state machine.
- An activity diagram is a special case of a statechart diagram in which all or most of the states are activity states and all or most of the transitions are triggered by completion of activities in the source state.
- Both activity and statechart diagrams are useful in modeling the lifetime of an object. However, whereas an activity diagram shows flow of control from activity to activity, a statechart diagram shows flow of control from state to state.
- Statechart diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering.
- In the UML, you model the event-ordered behavior of an object by using statechart diagrams.
- As below figure shows, a statechart diagram is simply a presentation of a state machine, emphasizing the flow of control from state to state.

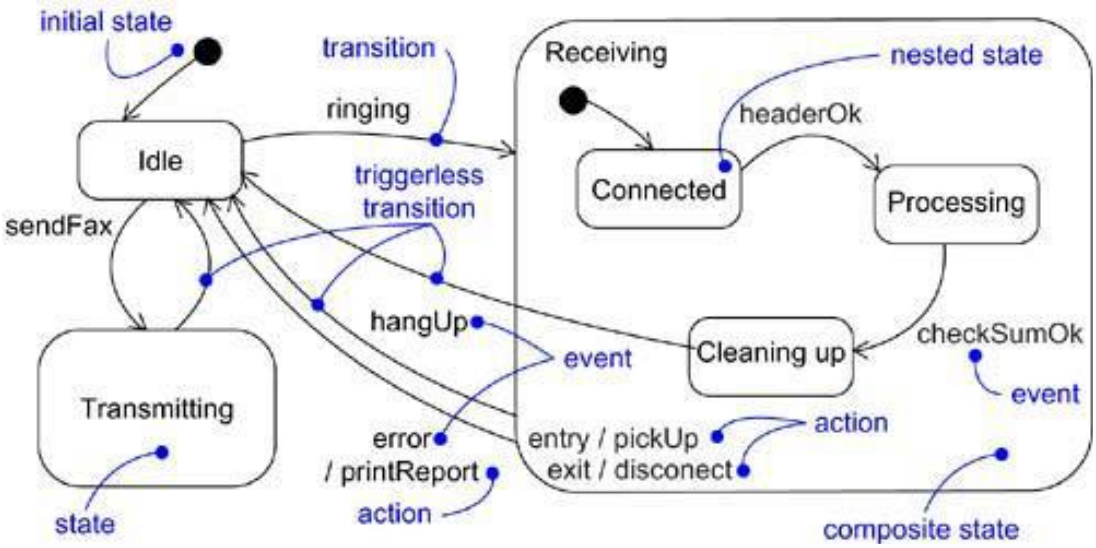


Fig: State chart Diagram

Terms and Concepts

- A statechart diagram shows a state machine, emphasizing the flow of control from state to state.
- A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- A state is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- 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.
- Graphically, a statechart diagram is a collection of vertices and arcs.

Common Properties

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

- What distinguishes a statechart diagram from all other kinds of diagrams is its content.

Contents

Statechart diagrams commonly contain

1. Simple states and composite states
2. Transitions, including events and actions

Note

A statechart diagram is basically a projection of the elements found in a state machine.

This means that statechart diagrams may contain

1. branches,
2. forks,
3. joins,
4. action states,
5. activity states,
6. objects,
7. initial states,
8. final states,
9. History states, and so on.

- A statechart diagram may contain any and all features of a state machine.

Common Uses

- When you model the dynamic aspects of a system, a class, or a use case, you'll typically use statechart diagrams in one way.

To model reactive objects

- A reactive — or event-driven — object is one whose behavior is best characterized by its response to events dispatched from outside its context.

- A reactive object is typically idle until it receives an event. When it receives an event, its response usually depends on previous events.

- After the object responds to an event, it becomes idle again, waiting for the next event.

- For these kinds of objects, you'll focus on the stable states of that object, the events that trigger a transition from state to state, and the actions that occur on each state change.

Common Modeling Technique
Modeling Reactive Objects

To model a reactive object

- Choose the context for the state machine, whether it is a class, a use case, or the system as a whole.

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

- Decide on the stable states of the object by considering the conditions in which the object may exist for some identifiable period of time. Start with the high-level states of the object and only then consider its possible substates.
-
- Decide on the meaningful partial ordering of stable states over the lifetime of the object.

- Decide on the events that may trigger a transition from state to state. Model these events as triggers to transitions that move from one legal ordering of states to another.

- Attach actions to these transitions (as in a Mealy machine) and/or to these states (as in a Moore machine).

- Consider ways to simplify your machine by using substates, branches, forks, joins, and history states.

- Check that all states are reachable under some combination of events.

- Check that no state is a dead end from which no combination of events will transition the object out of that state.

- Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses.

- For example, below figure shows the statechart diagram for parsing a simple context-free language, such as you might find in systems that stream in or stream out messages to XML. In this case, the machine is designed to parse a stream of characters that match the syntax

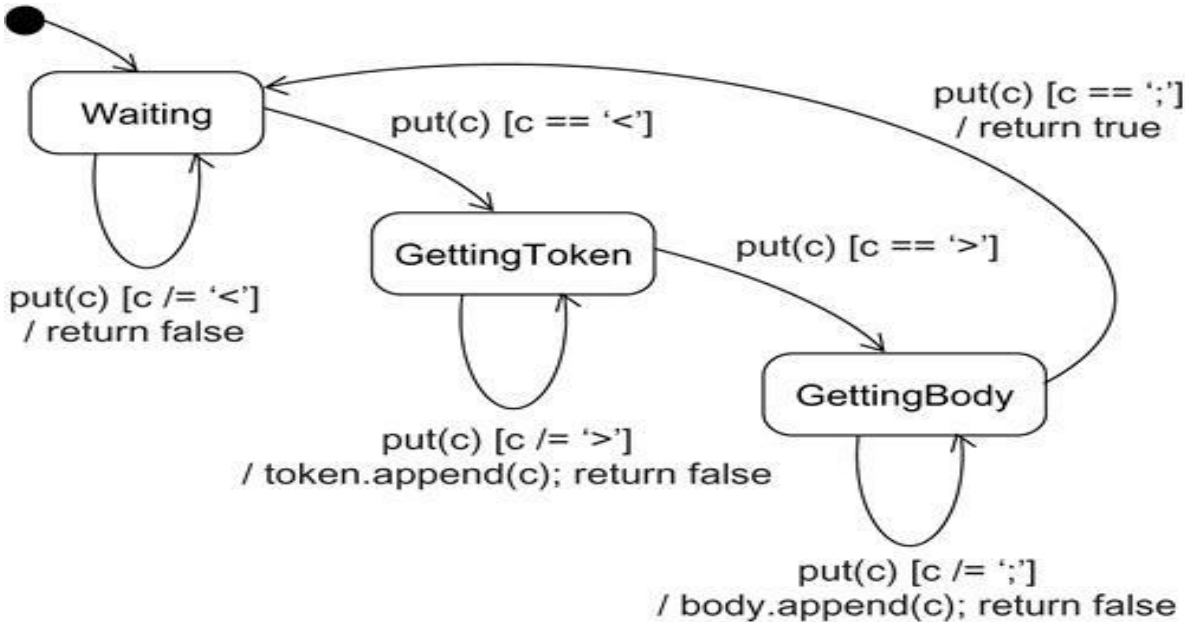


Fig: Modeling Reactive Objects

Forward and Reverse Engineering

- **Forward engineering** (the creation of code from a model) is possible for statechart diagrams, especially if the context of the diagram is a class. For example, using the previous statechart diagram, a forward engineering tool could generate the following Java code for the class

MessageParser.

```
class MessageParser {
public
boolean put(char c) {
switch (state) {
case Waiting:
if (c == '<') {
state = GettingToken;
token = new StringBuffer();
body = new StringBuffer();
}
break;
case GettingToken :
if (c == '>')
state = GettingBody;
else
token.append(c);
break;
case GettingBody :
if (c == ';')
state = Waiting;
else
body.append(c);
return true;
}
return false;
}
StringBuffer getToken() {
return token;
}
StringBuffer getBody() {
return body;
}
private
final static int Waiting = 0;
final static int GettingToken = 1;
final static int GettingBody = 2;
int state = Waiting;
StringBuffer token, body;
}
```

- **Reverse engineering** (the creation of a model from code) is theoretically possible, but practically not very useful. The choice of what constitutes a meaningful state is in the eye of the designer. Reverse engineering tools have no capacity for abstraction and therefore cannot automatically produce meaningful statechart diagrams

UNIT-V
Architectural Modeling: Component, Deployment, Component diagrams and Deployment diagrams.

UNIT-V

Components

- A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.
- You use components to model the physical things that may reside on a node, such as executables, libraries, tables, files, and documents.
- A component typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations.
- In the UML, all these physical things are modeled as components. A component is a physical thing that conforms to and realizes a set of interfaces
- The UML provides a graphical representation of a component, as below figure shows.
- This canonical notation permits you to visualize a component apart from any operating system or programming language.
- Using stereotypes, one of the UML's extensibility mechanisms, you can tailor this notation to represent specific kinds of components.

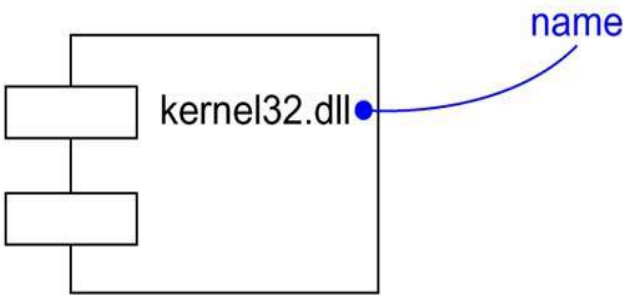


Fig: Components

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

- 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, as in below figure. Just as with classes, you may draw components adorned with tagged values or with additional compartments to expose their details, as you see in the figure.

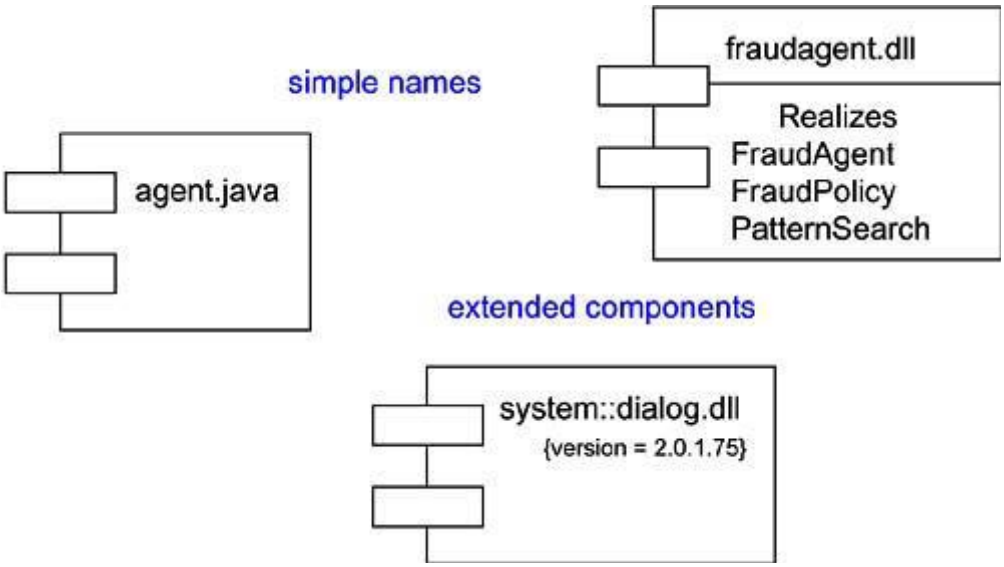


Fig: Simple and Extended Components

Components and Classes

Similarities

- 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.

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.

As below figure 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.

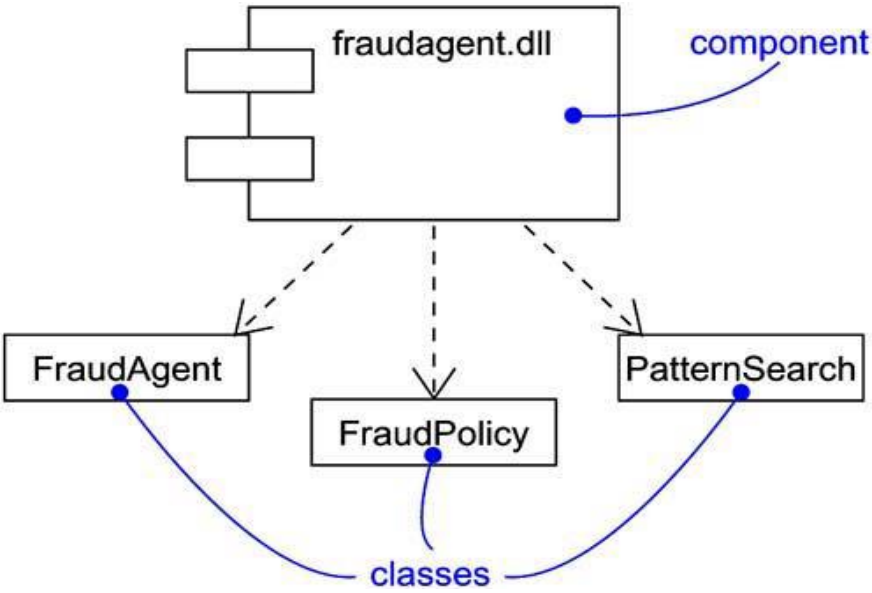


Fig: Components and Classes

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.
- As below figure indicates, you 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.

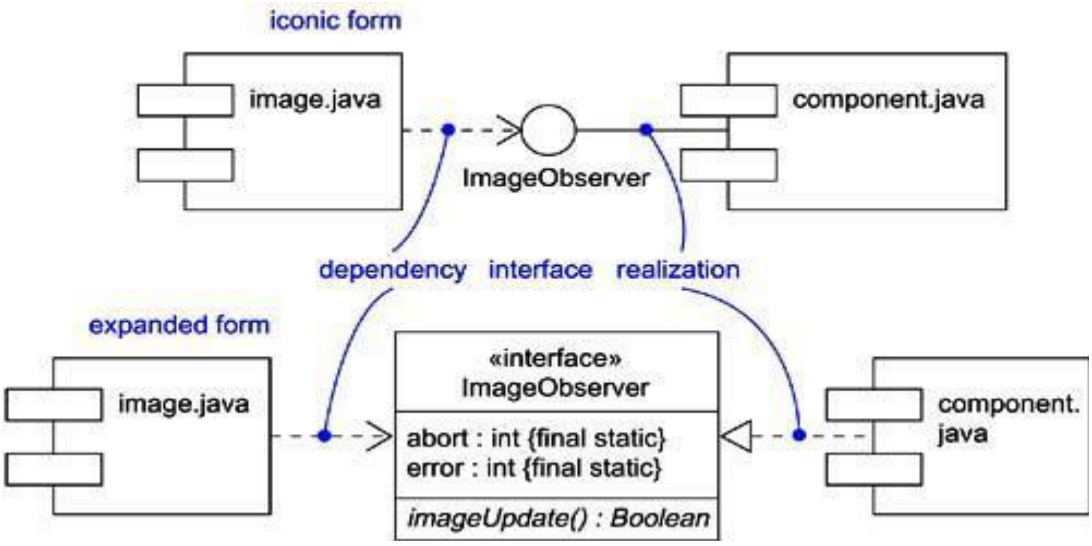


Fig: Components and Interfaces

- 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.

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 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.

- For example, below figure shows a set of components drawn from a personal productivity tool that runs on a single personal computer. This figure includes one executable (animator.exe, with a tagged value noting its version number) and four libraries (dlog.dll, wrfrme.dll, render.dll, and raytrce.dll), all of which use the UML's standard elements for executables and libraries, respectively. This diagram also presents the dependencies among these components.

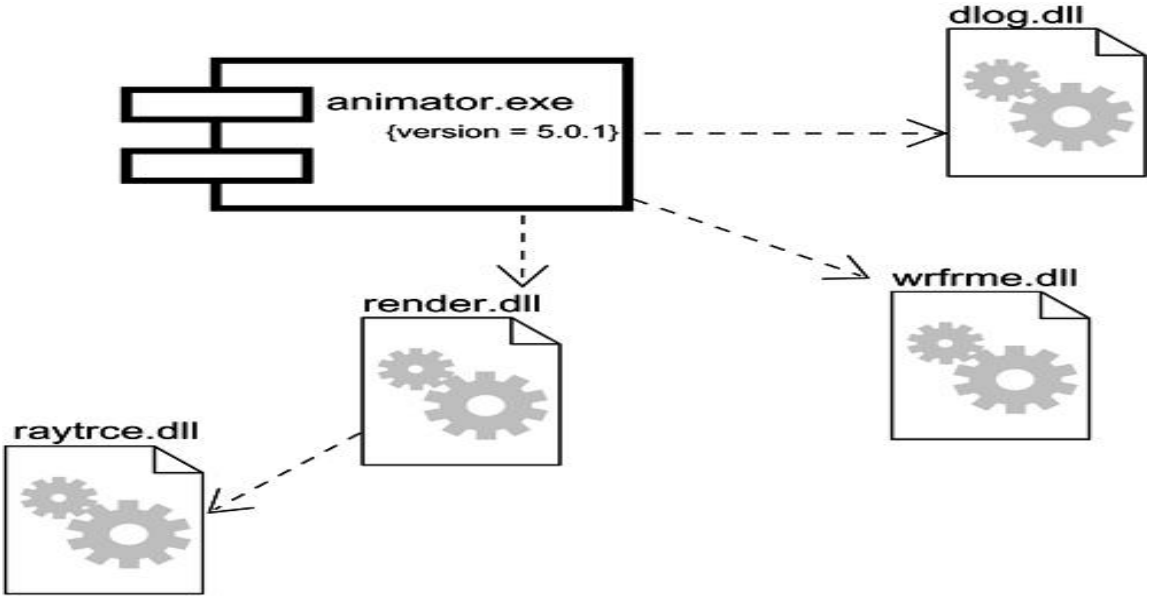


Fig: Modeling Executables and Libraries

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.

- For example, below figure builds on the previous figure and shows the tables, files, and documents that are part of the deployed system surrounding the executable animator.exe. This figure includes one document (animator.hlp), one simple file (animator.ini), and one database table (shapes.tbl), all of which use the UML's standard elements for documents, files, and tables, respectively.

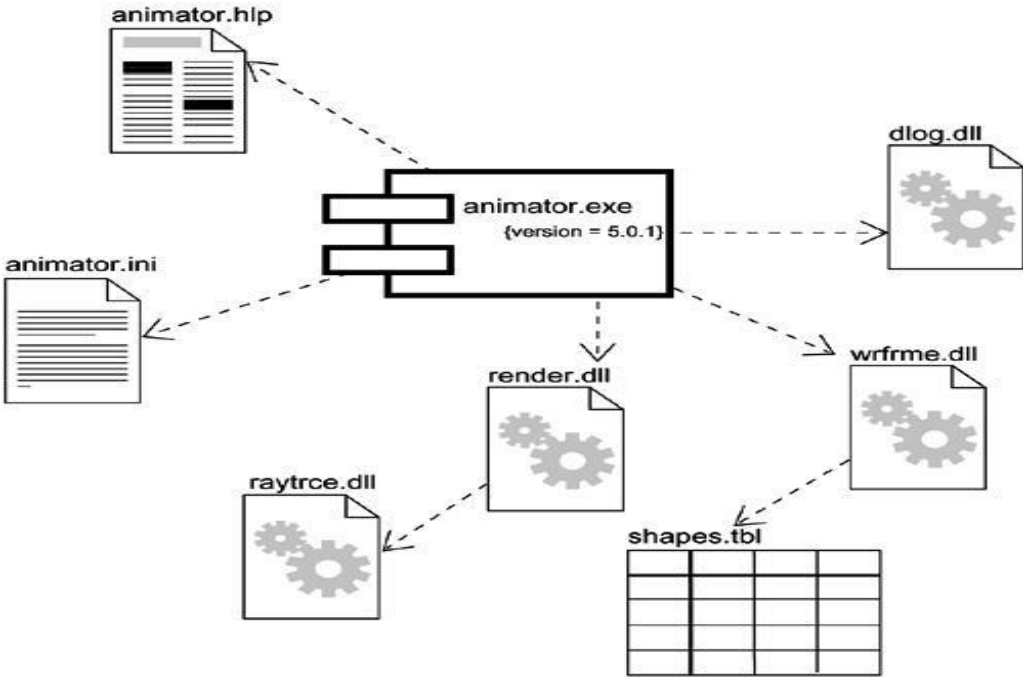


Fig: Modeling Tables, Files, and Documents

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 exposes the APIs of the executable in the previous two figures. You'll see four interfaces that form the API of the executable: IApplication, IModels, IRendering, and IScripts.

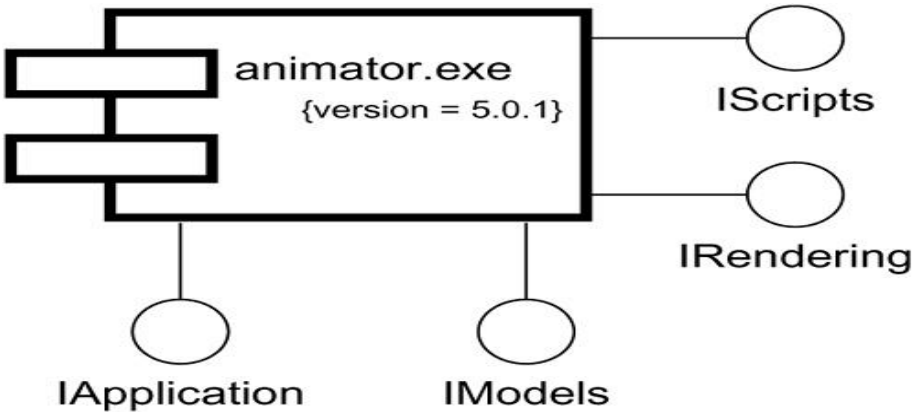


Fig: Modeling an API

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.

- For example, below figure shows some source code files that are used to build the library render.dll from the previous examples. This figure includes four header files (render.h, rengine.h, poly.h, and colortab.h) that represent the source code for the specification of certain classes. There is also one implementation file (render.cpp) that represents the implementation of one of these headers.

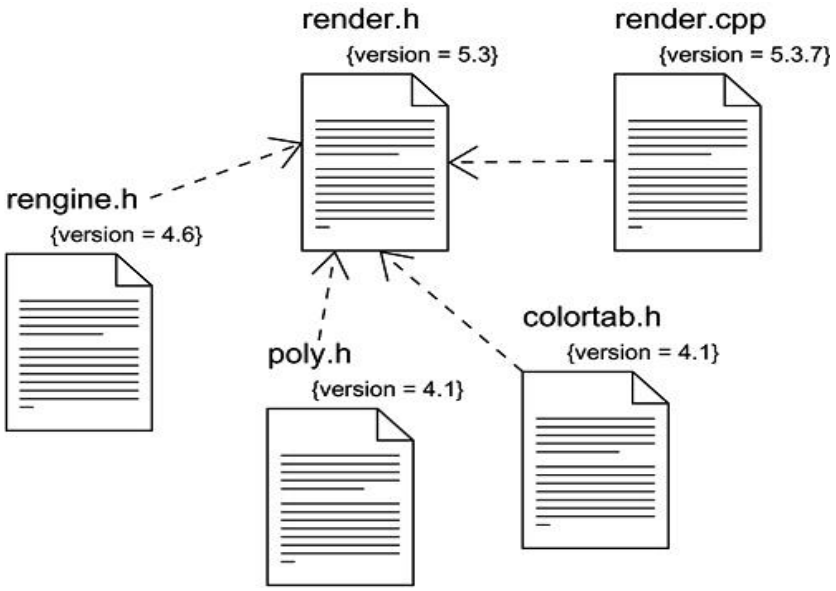


Fig: Modeling Source Code

Deployment

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

- You use nodes to model the topology of the hardware on which your system executes.

- A node typically represents a processor or a device on which components may be deployed.

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

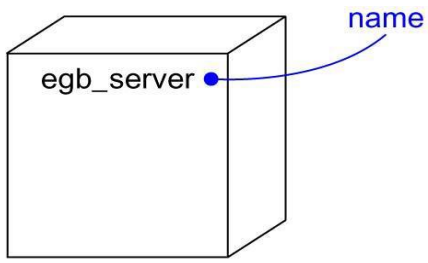


Figure: Nodes

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

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

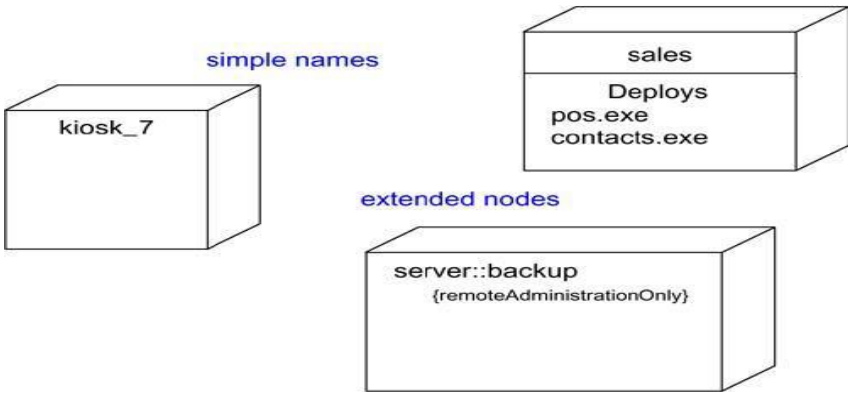


Fig: Simple names and Path names

Nodes and Components

Similarities

- 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.

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; **Codes** 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.

- As below figure shows, 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.

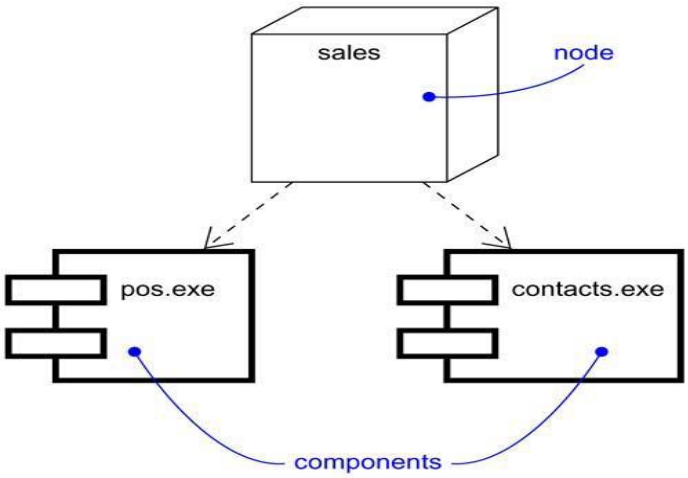


Fig: Nodes and Components

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

Organizing Nodes

- You can organize nodes by grouping them in packages in the same manner in which you can organize classes and components.
- You can also organize nodes by specifying dependency, generalization, and association (including aggregation) relationships among them.

Connections

- The most common kind of relationship you'll use among nodes is an association.

- In this context, an association represents a physical connection among nodes, such as an Ethernet connection, a serial line, or a shared bus,
- As below figure shows. You can even use associations to model indirect connections, such as a satellite link between distant processors.

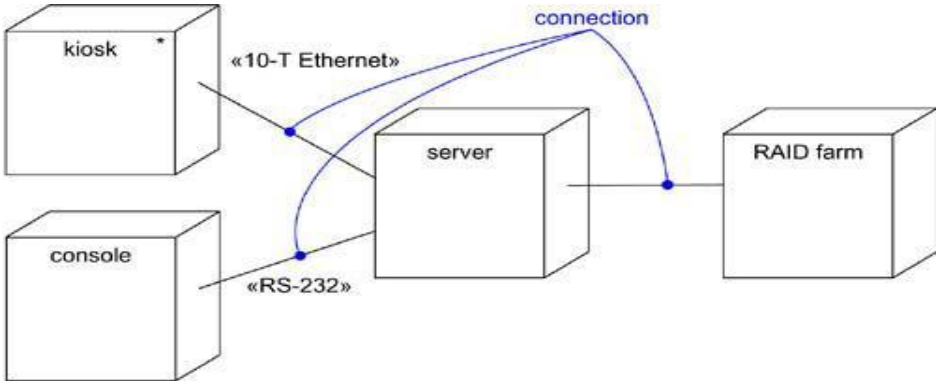


Fig: Connections

- Because nodes are class-like, you have the full power of associations at your disposal.
- This means that you can include roles, multiplicity, and constraints.
- As in the previous figure, you should stereotype these associations if you want to model new kinds of connections for example, to distinguish between a 10-T Ethernet connection and an RS-232 serial connection.

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.
- For example, below figure takes the previous diagram and stereotypes each node. The server is a node stereotyped as a generic processor; the kiosk and the console are nodes stereotyped as special kinds of processors; and the RAID farm is a node stereotyped as a special kind of device.

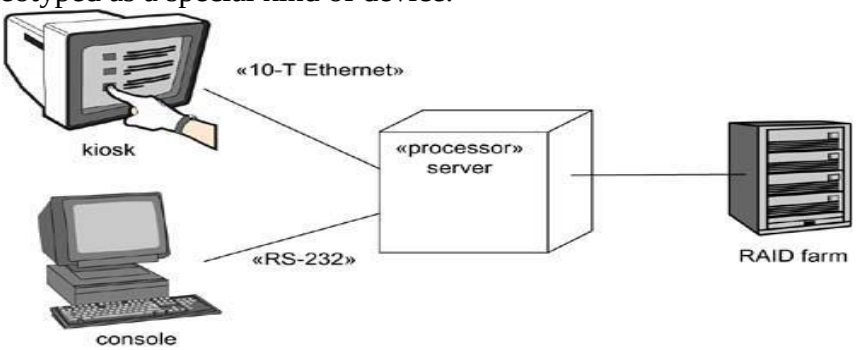


Fig: Processors and Devices

Modeling the Distribution of Components

To model the distribution of components

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

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

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

- Below figure takes the earlier diagrams and specifies the executable components that reside on each node. This diagram is a bit different from the previous ones in that it is an object diagram, visualizing specific instances of each node. In this case, the RAID farm and kiosk instances are both anonymous and the other two instances are named (c for the console and s for the server). Each processor in this figure is rendered with an additional compartment showing the component it deploys. The server object is also rendered with its attributes (processorSpeed and memory) and their values visible.

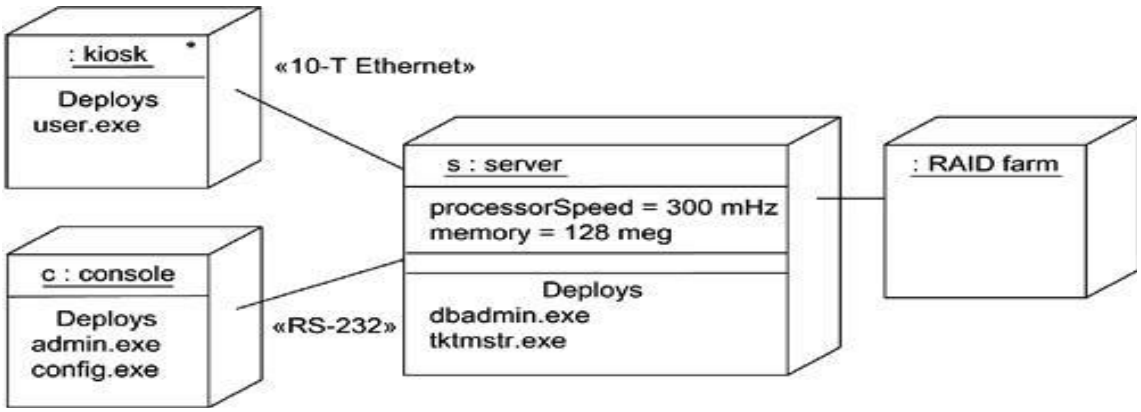


Fig: Modeling the Distribution of Components.

Component Diagrams

- Component diagrams are one of the two kinds of diagrams found in modeling the physical aspects of object-oriented systems.

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

- You use component diagrams to model the static implementation view of a system.

- This involves modeling the physical things that reside on a node, such as executables, libraries, tables, files, and documents.

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

- Component diagrams are not only important for visualizing, specifying, and documenting component-based systems, but also for constructing executable systems through forward and reverse engineering.

- With the UML, you use component diagrams to visualize the static aspect of these physical components and their relationships and to specify their details for construction, as shown in figure

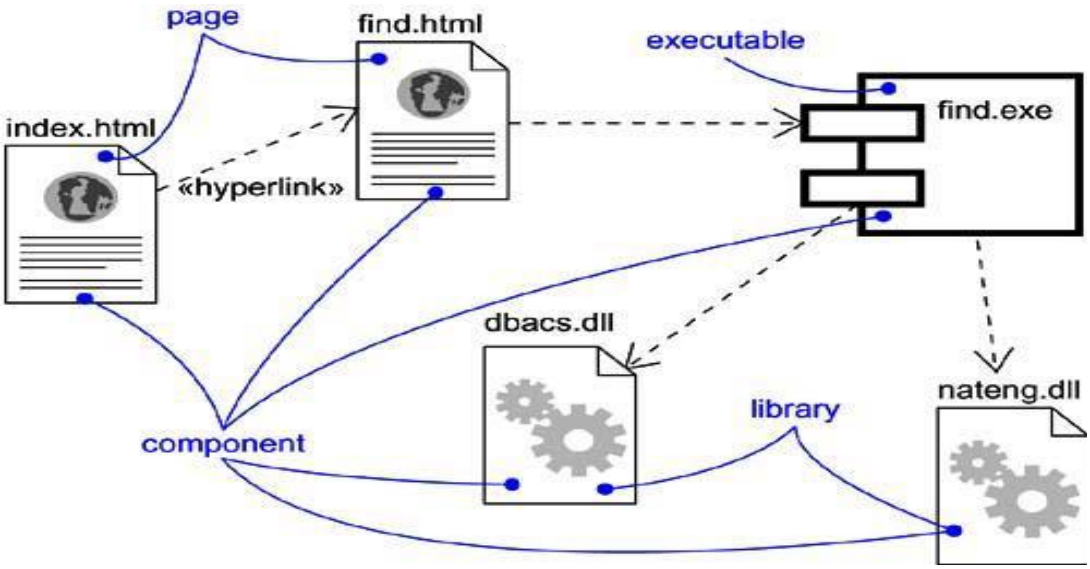


Fig: Component Diagram

Terms and Concepts

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

Common Properties

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

Contents

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.

Common Uses

- You use component diagrams to model the static implementation view of a system.
- This view primarily supports the configuration management of a system's parts, made up of components that can be assembled in various ways to produce a running system.
- When you model the static implementation view of a system, you'll typically use component diagrams in one of four ways.

1 **To model source code**
- With most contemporary object-oriented programming languages, you'll cut code using integrated development environments that store your source code in files.
- You can use component diagrams to model the configuration management of these files, which represent work-product components.

2. **To model executable releases**
- A release is a relatively complete and consistent set of artifacts delivered to an internal or external user.
- In the context of components, a release focuses on the parts necessary to deliver a running system. When you model a release using component diagrams, you are visualizing, specifying, and documenting the decisions about the physical parts that constitute your software• that is, its deployment components.

3. **To model physical databases**
- Think of a physical database as the concrete realization of a schema, living in the world of bits.
- Schemas, in effect, offer an API to persistent information; the model of a physical database represents the storage of that information in the tables of a relational database or the pages of an object-oriented database.
- You use component diagrams to represent these and other kinds of physical databases.

4.**To model adaptable systems**
- Some systems are quite static; their components enter the scene, participate in an execution, and then depart.
- Other systems are more dynamic, involving mobile agents or components that migrate for purposes of load balancing and failure recovery.
- You use component diagrams in conjunction with some of the UML's diagrams for modeling behavior to represent these kinds of 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.

- For example, below figure shows five source code files. signal.h is a header file. Three of its versions are shown, tracing from new versions back to their older ancestors. Each variant of this source code file is rendered with a tagged value exposing its version number.

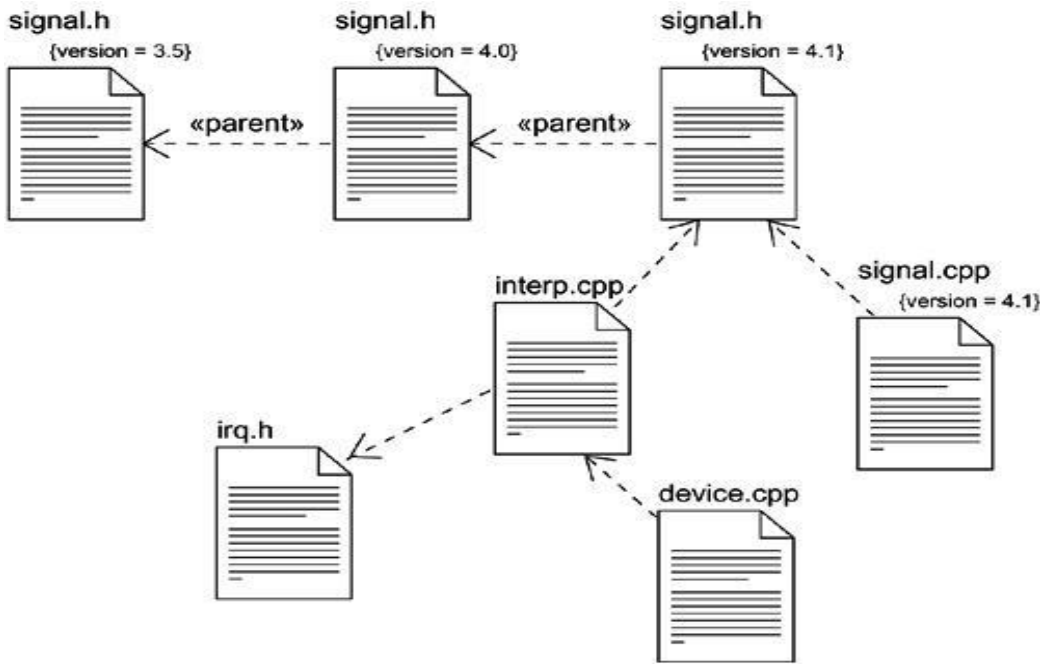


Fig: Modeling Source Code

Modeling an Executable Release

To model an executable release

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

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

- For example, below figure models part of the executable release for an autonomous robot. This figure focuses on the deployment components associated with the robot's driving and calculation functions. You'll find one component (driver.dll) that exports an interface (IDrive) that is, in turn, imported by another component (path.dll). driver.dll exports one other interface (ISelfTest) that is probably used by other components in the system, although they are not shown here. There's one other component shown in this diagram (collision.dll), and it, too, exports a set of interfaces, although these details are elided: path.dll is shown with a dependency directly to collision.dll.

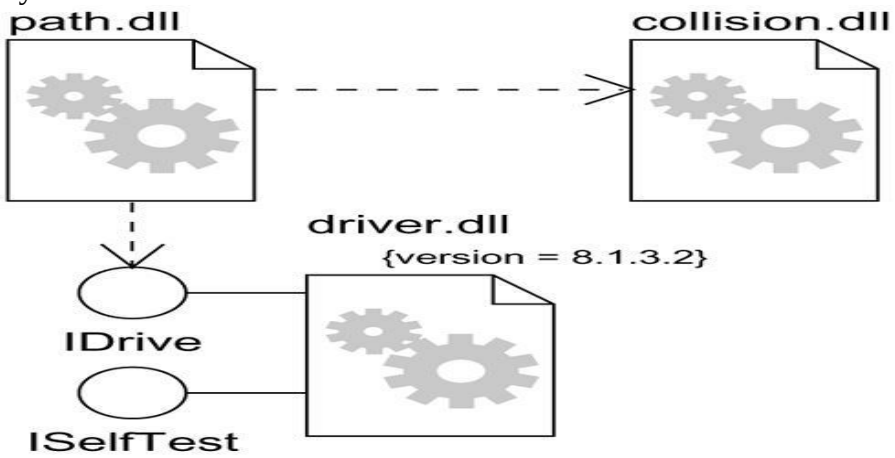


Fig: Modeling an Executable Release

Modeling a Physical Database

To model a physical database

- Identify the classes in your model that represent your logical database schema.
- Select a strategy for mapping these classes to tables. You will also want to consider the physical distribution of your databases. Your mapping strategy will be affected by the location in which you want your data to live on your deployed system.
- To visualize, specify, construct, and document your mapping, create a component diagram that contains components stereotyped as tables.
- Where possible, use tools to help you transform your logical design into a physical design.
- Below figure shows a set of database tables drawn from an information system for a school. You will find one database (school.db, rendered as a component stereotyped as database) that's composed of five tables: student, class, instructor, department, and course (rendered as a component stereotyped as table, one of the UML's standard elements). In the corresponding logical database schema, there was no inheritance, so mapping to this physical database design is straightforward.

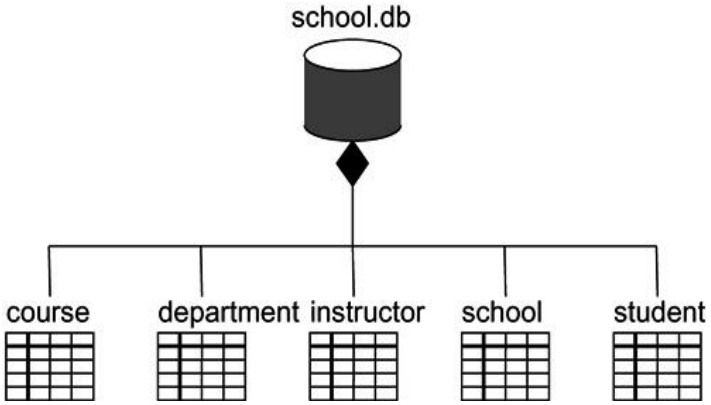


Fig: Modeling a Physical Database

Modeling Adaptable Systems

To model an adaptable system

- Consider the physical distribution of the components that may migrate from node to node.
- You can specify the location of a component instance by marking it with a location tagged value, which you can then render in a component diagram (although, technically speaking, a diagram that contains only instances is an object diagram).
- If you want to model the actions that cause a component to migrate, create a corresponding interaction diagram that contains component instances.
- You can illustrate a change of location by drawing the same instance more than once, but with different values for its location tagged value.
- For example, below figure models the replication of the database from the previous figure. We show two instances of the component school.db. Both instances are anonymous, and both have a different value for their location tagged value. There's also a note, which explicitly specifies which instance replicates the other.

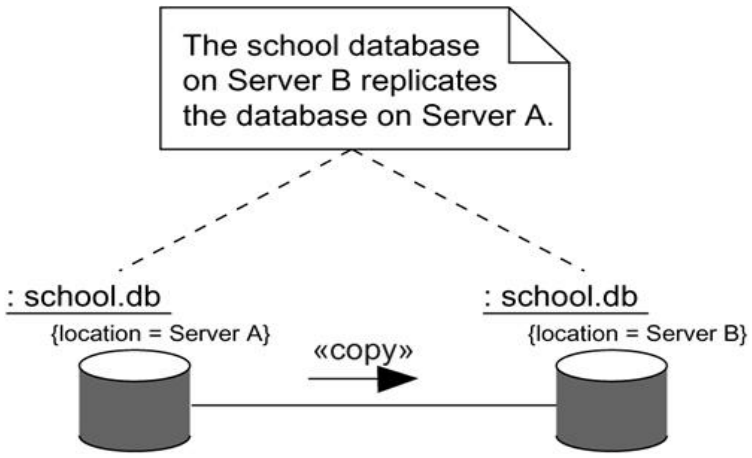


Fig: Modeling Adaptable Systems

Forward and Reverse Engineering: The creation of code from a model

To forward engineer a component diagram

- For each component, identify the classes or collaborations that the component implements.

- Choose the target for each component. Your choice is basically between source code (a form that can be manipulated by development tools) or a binary library or executable (a form that can be dropped into a running system).

- Use tools to forward engineer your models.

Reverse engineering (the creation of a model from code) a component diagram is not a perfect process because there is always a loss of information.

To reverse engineer a component diagram

- Choose the target you want to reverse engineer. Source code can be reverse engineered to components and then classes. Binary libraries can be reverse engineered to uncover their interfaces. Executables can be reverse engineered the least.

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

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

- For example, below figure provides a component diagram that represents the reverse engineering of the ActiveX component vbrun.dll. As the figure shows, the component realizes 11 interfaces. Given this diagram, you can begin to understand the semantics of the component by next exploring the details of its interfaces.

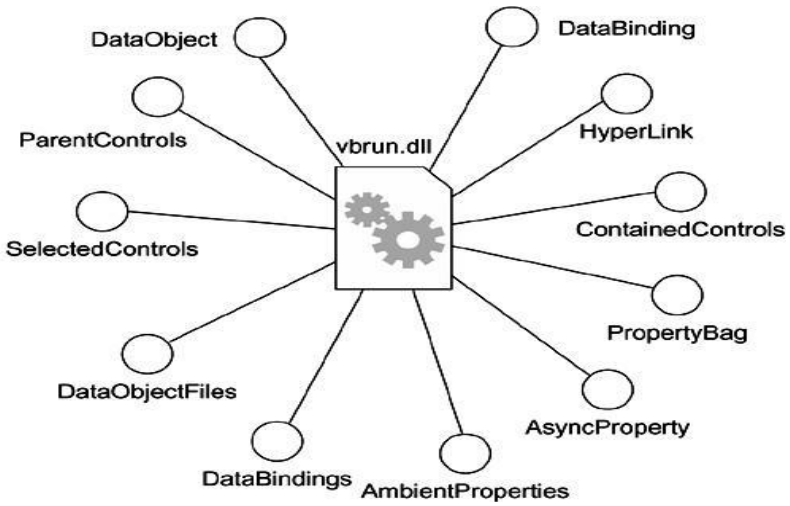


Fig: Reverse Engineering

Deployment Diagrams

- Deployment diagrams are one of the two kinds of diagrams used in modeling the physical aspects of an object-oriented system.
- A deployment diagram shows the configuration of run time processing nodes and the components that live on them.
- You use deployment diagrams to model the static deployment view of a system.
- For the most part, this involves modeling the topology of the hardware on which your system executes.
- Deployment diagrams are essentially class diagrams that focus on a system's nodes.
- Deployment diagrams are not only important for visualizing, specifying, and documenting embedded, client/server, and distributed systems, but also for managing executable systems through forward and reverse engineering.

- With the UML, you use deployment diagrams to visualize the static aspect of these physical nodes and their relationships and to specify their details for construction, as in below figure.

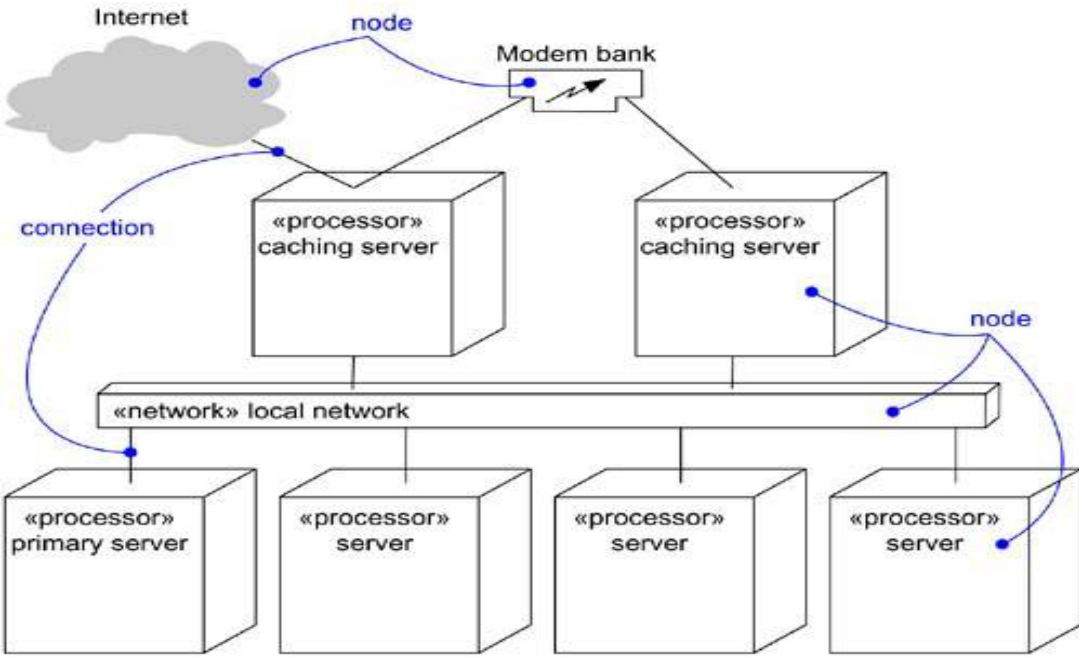


Fig: A Deployment Diagram

Terms and Concepts

- 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.

Common Properties

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

Contents

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,

Common Uses

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

1. **To model embedded systems**
- An embedded system is a software-intensive collection of hardware that interfaces with the physical world.
- Embedded systems involve software that controls devices such as motors, actuators, and displays and that, in turn, is controlled by external stimuli such as sensor input, movement, and temperature changes.
- You can use deployment diagrams to model the devices and processors that comprise an embedded system.

2. **To model client/server systems**
- A client/server system is a common architecture focused on making a clear separation of concerns between the system's user interface (which lives on the client) and the system's persistent data (which lives on the server).
- Client/ server systems are one end of the continuum of distributed systems and require you to make decisions about the network connectivity of clients to servers and about the physical distribution of your system's software components across the nodes.
- You can model the topology of such systems by using deployment diagrams.

3**. To model fully distributed systems**
- At the other end of the continuum of distributed systems are those that are widely, if not globally, distributed, typically encompassing multiple levels of servers
- Such systems are often hosts to multiple versions of software components, some of which may even migrate from node to node.
- Crafting such systems requires you to make decisions that enable the continuous change in the system's topology. You can use deployment diagrams to visualize the

system's current topology and distribution of components to reason about the impact of changes on that topology.

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.
- For example, below figure shows the hardware for a simple autonomous robot. You'll find one node (Pentium motherboard) stereotyped as a processor.

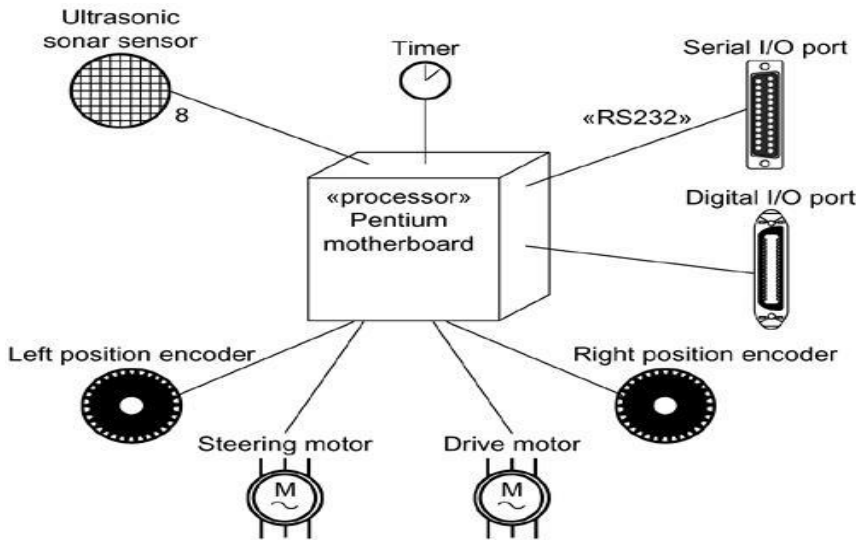


Fig: Modeling an Embedded System

Modeling a Client/Server System

To model a client/server system

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

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

- Provide visual cues for these processors and devices via stereotyping.

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

- For example, below figure shows the topology of a human resources system, which follows a classical client/server architecture. This figure illustrates the client/server split explicitly by using the packages named client and server. The client package contains two nodes (console and kiosk), both of which are stereotyped and are visually distinguishable. The server package contains two kinds of nodes (caching server and server), and both of these have been adorned with some of the components that reside on each. Note also that caching server and server are marked with explicit multiplicities, specifying how many instances of each are expected in a particular deployed configuration. For example, this diagram indicates that there may be two or more caching servers in any deployed instance of the system.

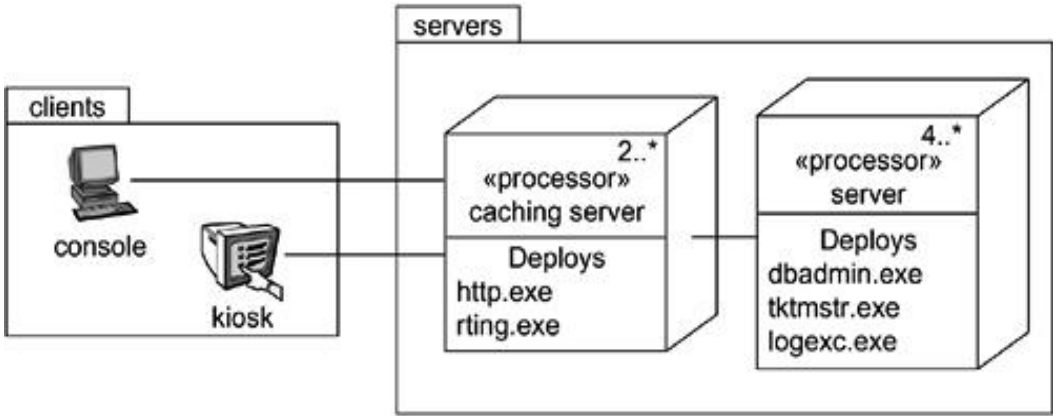


Fig: Modeling a Client/Server System

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.

- Below figure shows the topology of a fully distributed system. This particular deployment diagram is also an object diagram, for it contains only instances. You can see three consoles (anonymous instances of the stereotyped node console), which are

linked to the Internet (clearly a singleton node). In turn, there are three instances of regional servers, which serve as front ends of country servers, only one of which is shown. As the note indicates, country servers are connected to one another, but their relationships are not shown in this diagram.

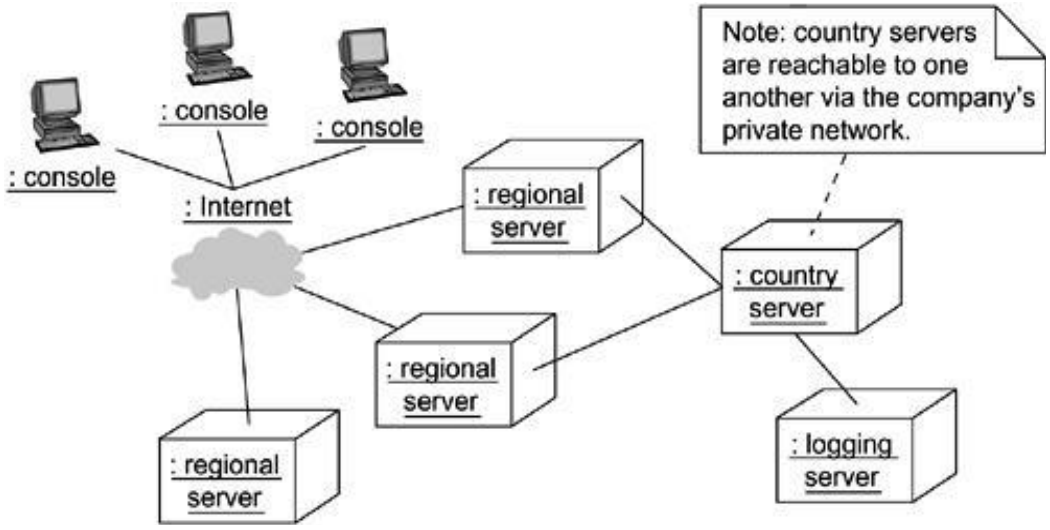


Fig: Modeling a Fully Distributed System

Forward and Reverse Engineering

- There's only a modest amount of forward engineering (the creation of code from models) that you can do with deployment diagrams.
- For example, after specifying the physical distribution of components across the nodes in a deployment diagram, it is possible to use tools that then push these components out to the real world. For system administrators, using the UML in this way helps you visualize what can be a very complicated task.

Reverse engineering (the creation of models from code) from the real world back to deployment diagrams is of tremendous value, especially for fully distributed systems that are under constant change.

- You'll want to supply a set of stereotyped nodes that speak the language of your system's network administrators, in order to tailor the UML to their domain. The advantage of using the UML is that it offers a standard language that addresses not only their needs, but the needs of your project's software developers, as well.

To reverse engineer a deployment diagram

- Choose the target that you want to reverse engineer. In some cases, you'll want to sweep across your entire network; in others, you can limit your search.

- Choose also the fidelity of your reverse engineering. In some cases, it's sufficient to reverse engineer just to the level of all the system's processors; in others, you'll want to reverse engineer the system's networking peripherals, as well.

- Use a tool that walks across your system, discovering its hardware topology. Record that topology in a deployment model.

- Along the way, you can use similar tools to discover the components that live on each node, which you can also record in a deployment model. You'll want to use an intelligent search, for even a basic personal computer can contain gigabytes of components, many of which may not be relevant to your system.

- Using your modeling tools, create a deployment diagram by querying the model.

- For example, you might start with visualizing the basic client/server topology, then expand on the diagram by populating certain nodes with components of interest that live on them. Expose or hide the details of the contents of this deployment diagram as necessary to communicate your intent.

UNIT - VI
Case Study: The Unified Library application, ATM application.

UNIT - VI

The 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:

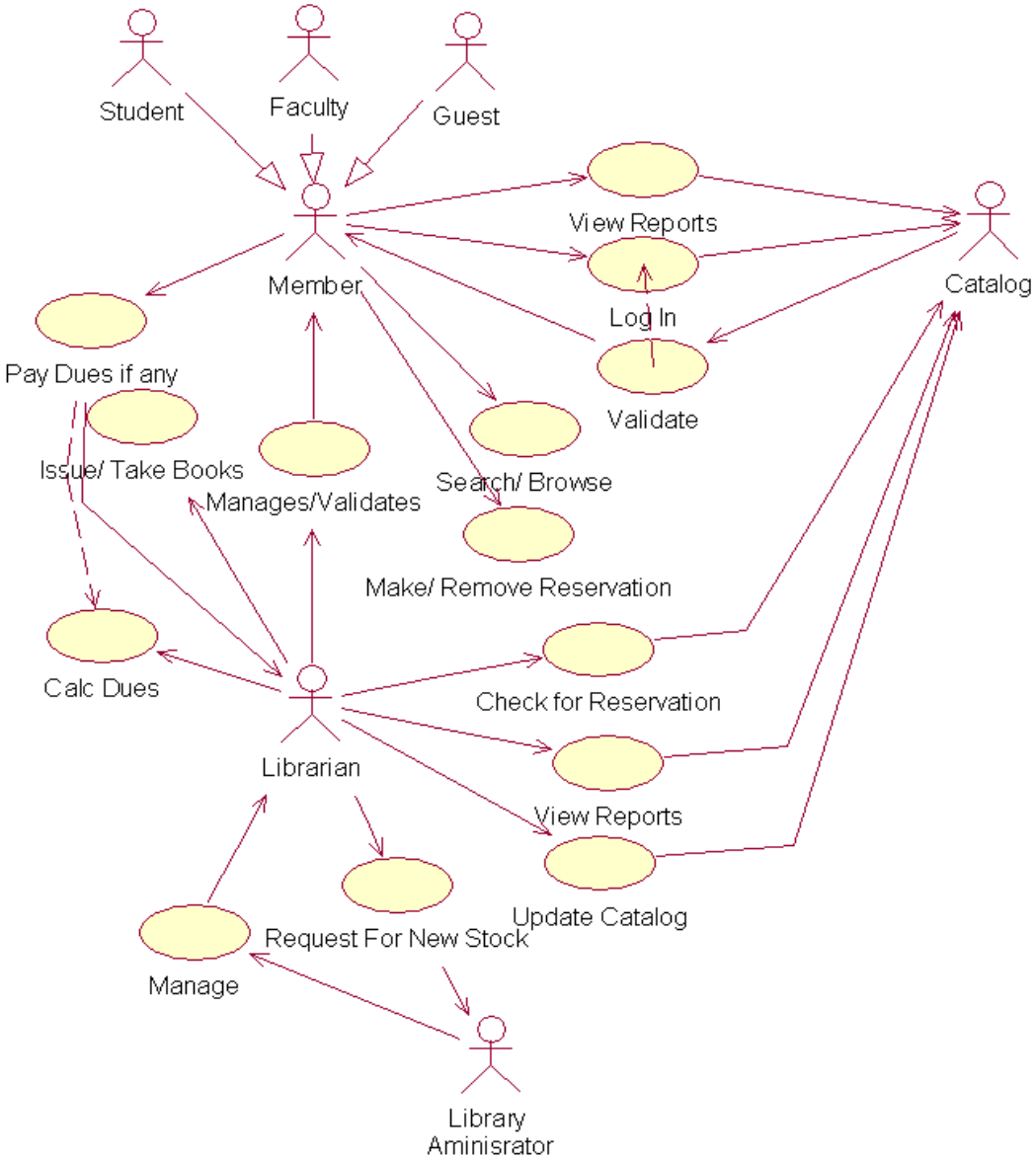
1. Manages and validates members
2. View reports from the system
3. Issues books
4. Calculates dues
5. Takes books
6. Places orders to the master librarian
7. Maintains list of books and magazine

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.

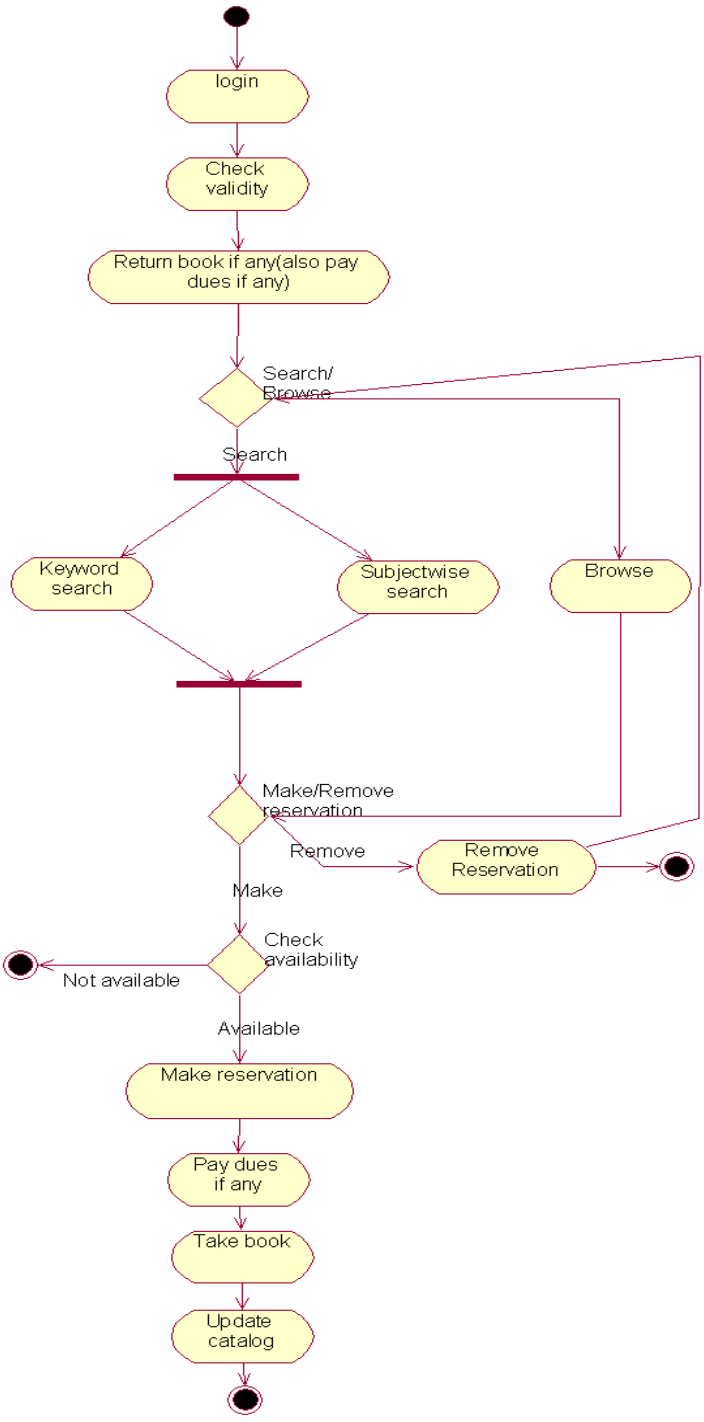


Obj

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



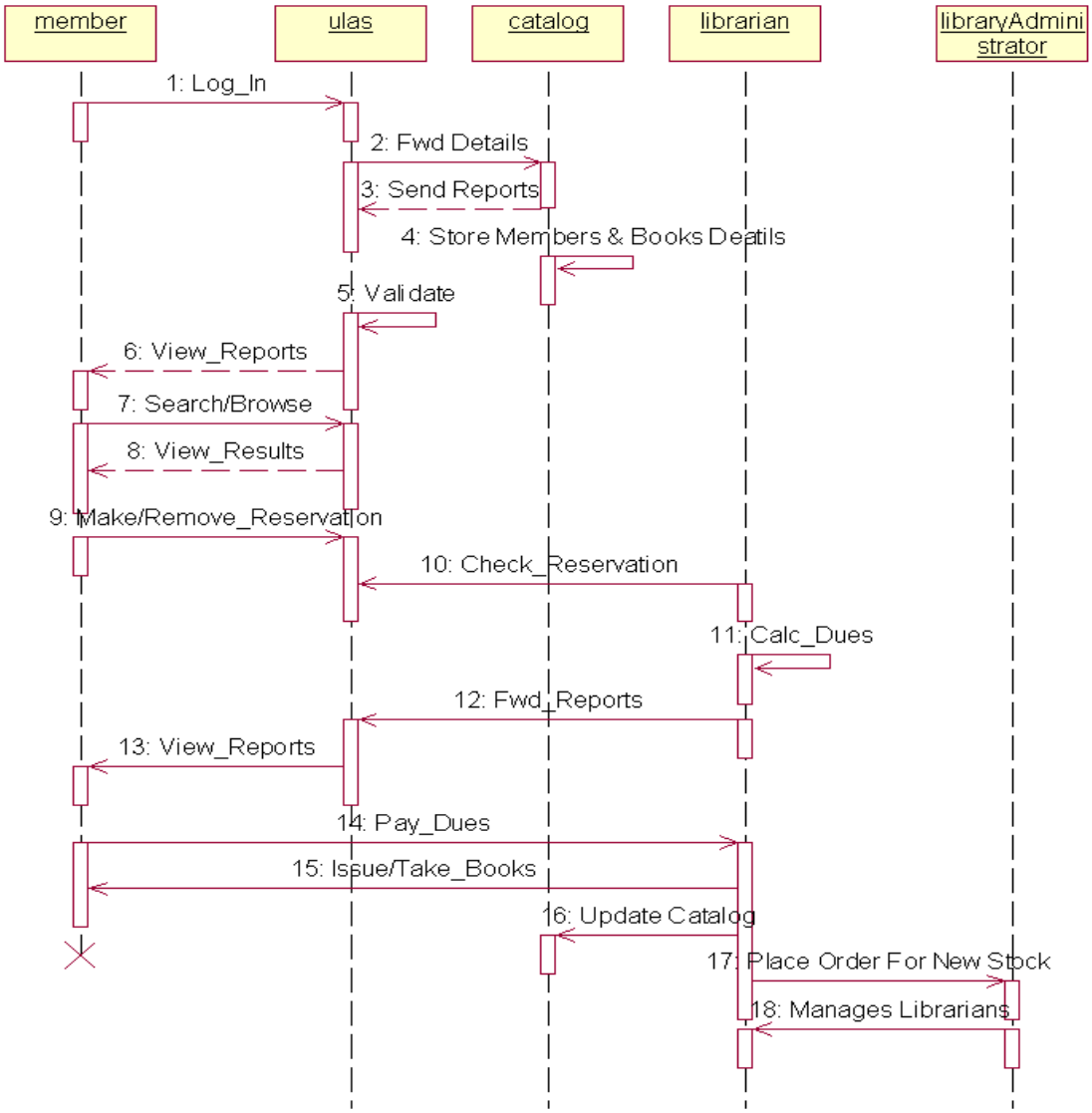
INTERACTION DIAGRAM:

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



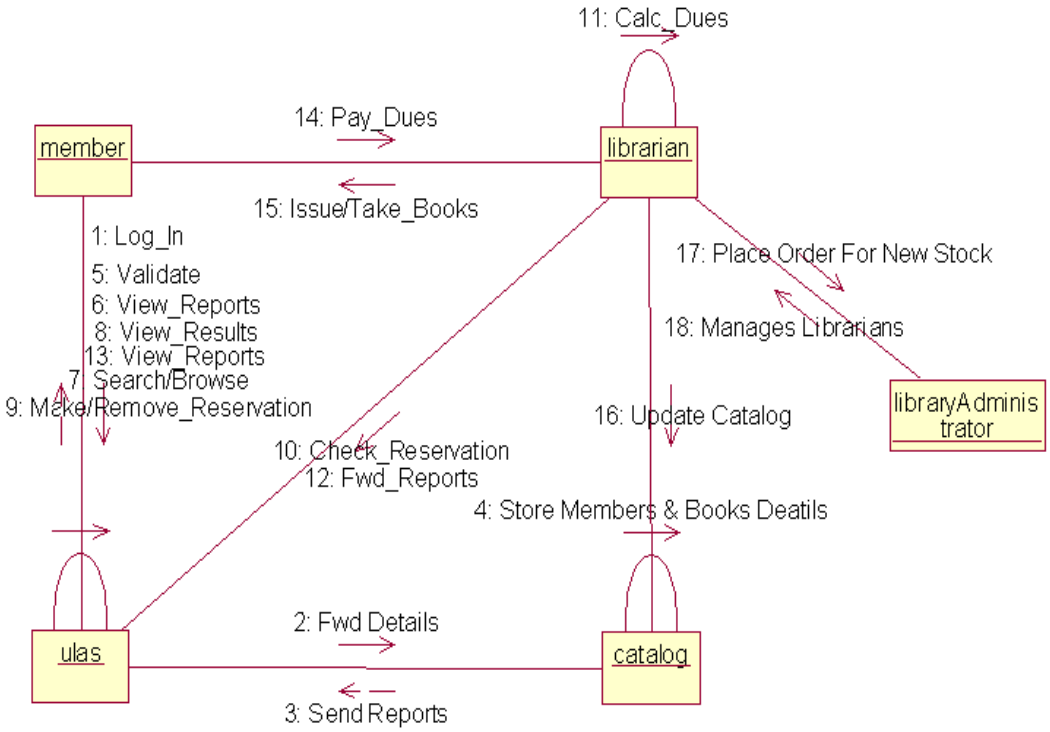
COLLABORATION DIAGRAM*:*

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:

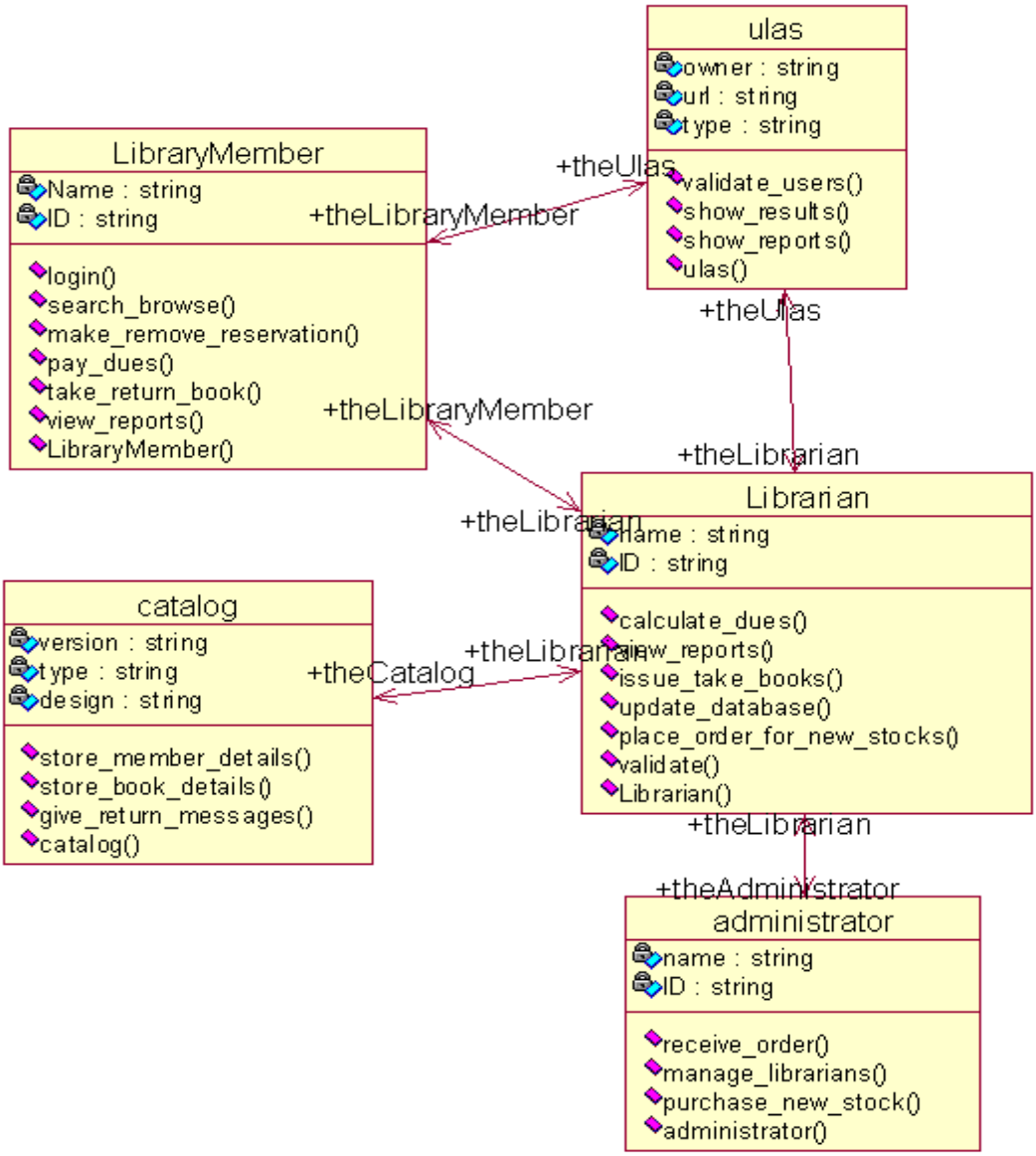
Class diagram shows structure of the software system. The class diagram shows a set of classes, interfaces and their relationships. The components are:

a) Class

b) Relationship:

The forms of relationship are:

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



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:

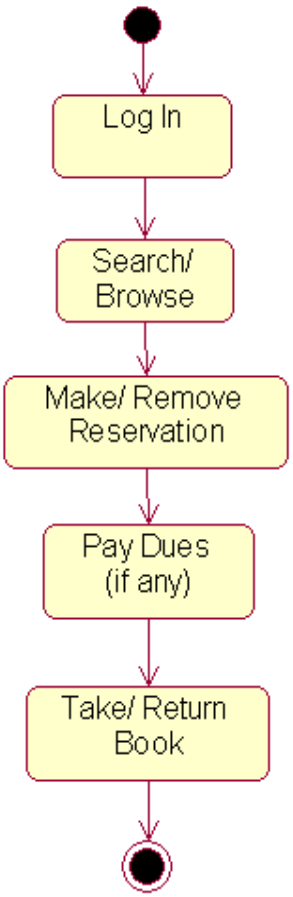
a) Start state

b) End state

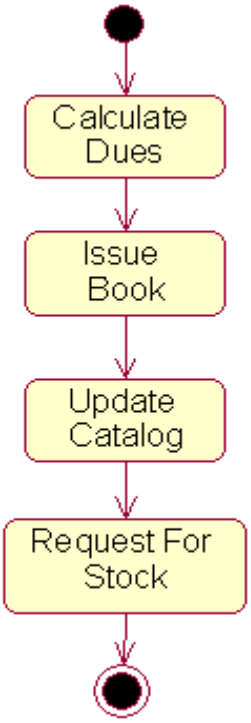
c) State

d) Transition

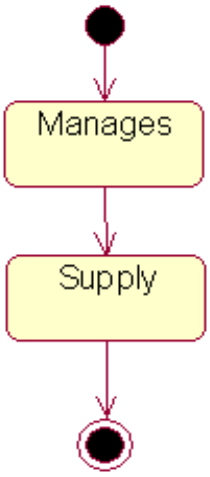
Member:



Librarian:



Library Administrator:



ATM application

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:

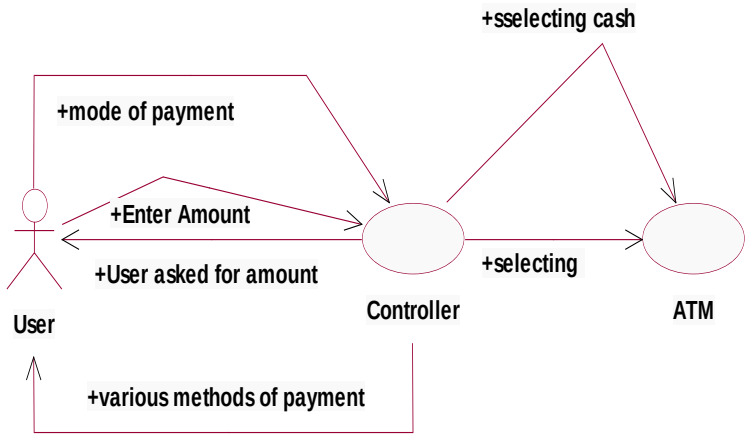
The scope of the project of the ATM system is to understand the working of the system and also to provide more security to the user as it facilitates with easy transaction.

Problem Statement :

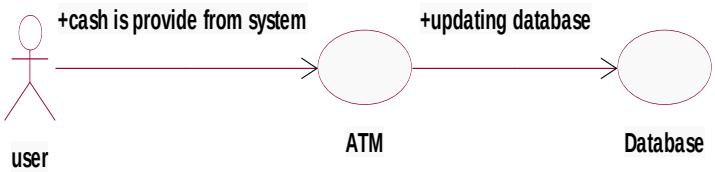
The operations on the function of the ATM are as follows:

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.

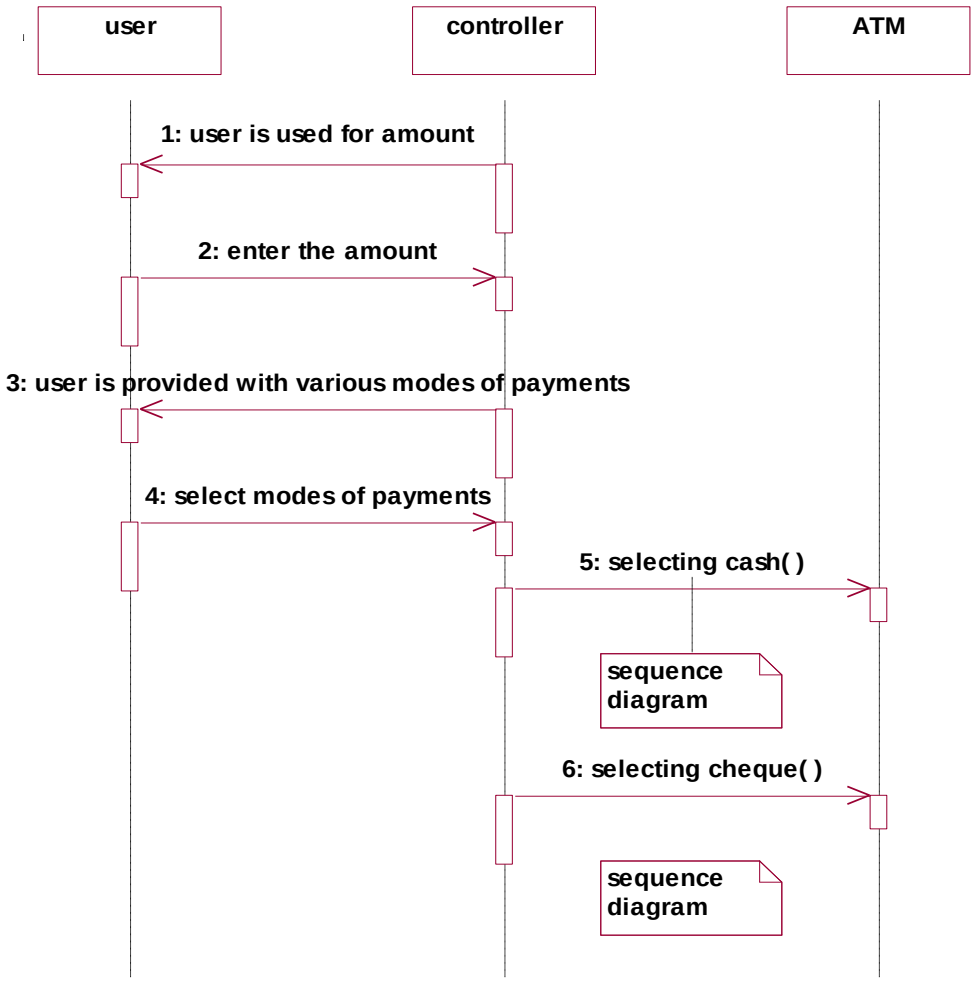
Withdraw usecase diagram :-



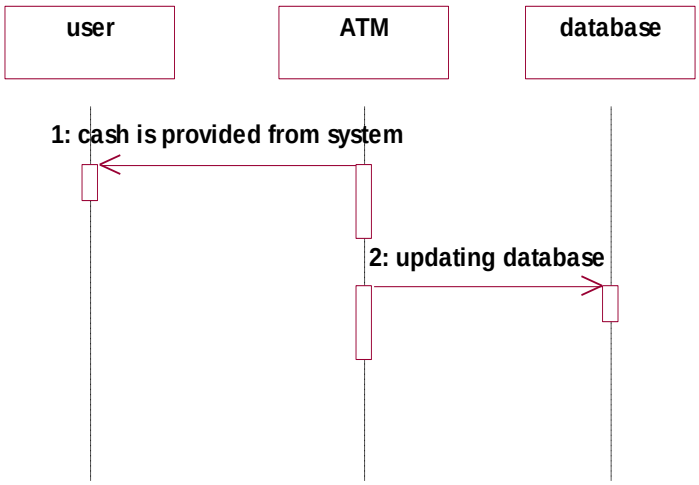
Withdraw cash usecase diagram :-



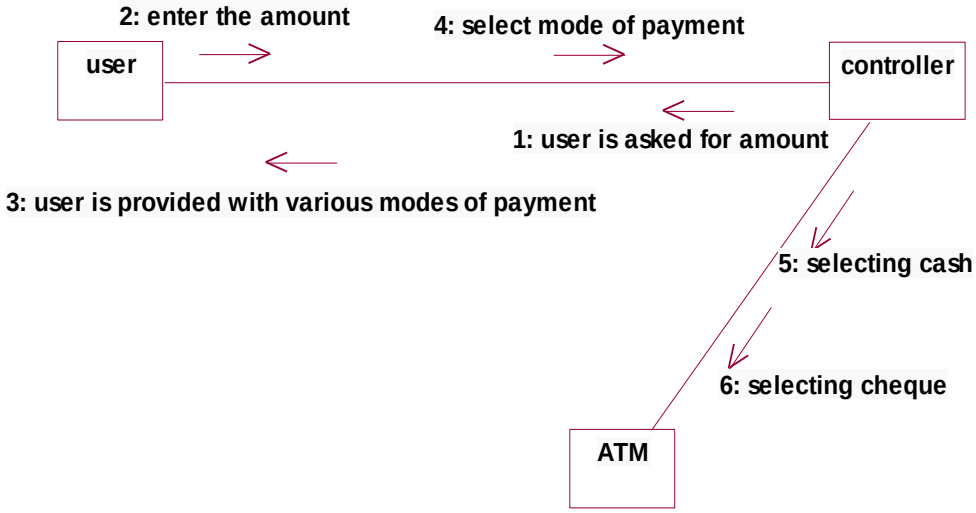
Withdraw sequence diagram :-



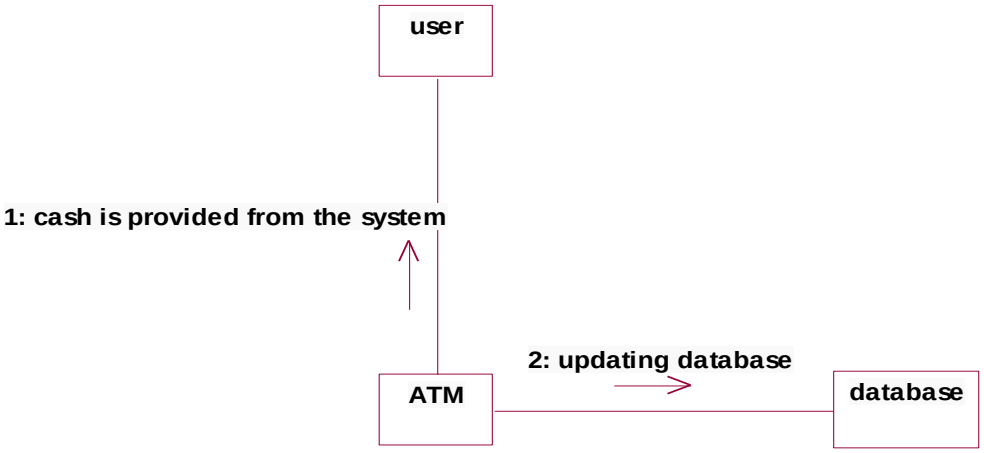
Withdraw cash sequence diagram :-



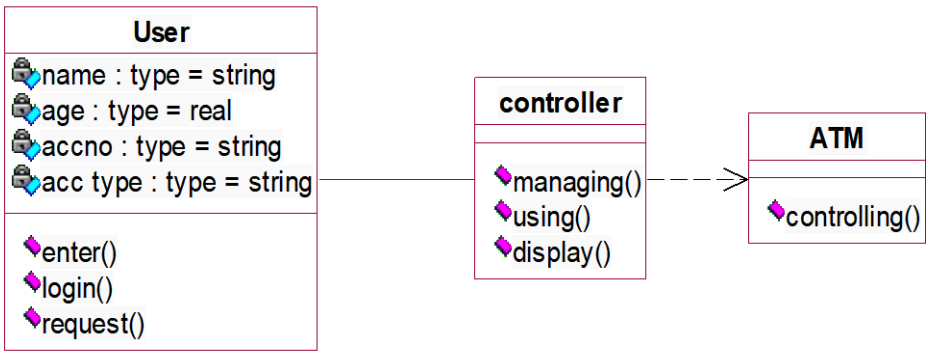
Withdraw collaboration:-



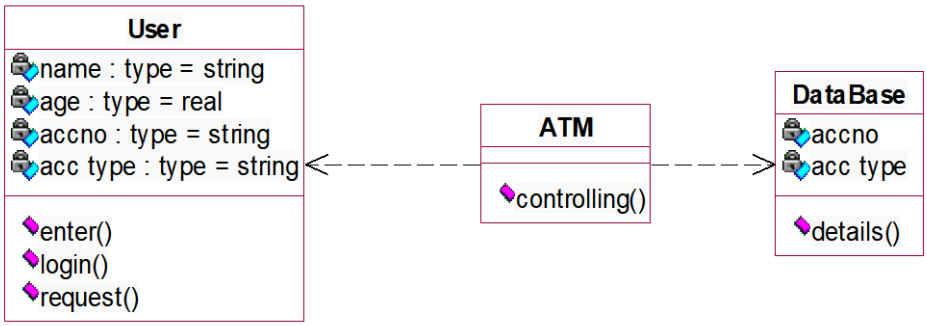
Withdraw cash collaboration :-



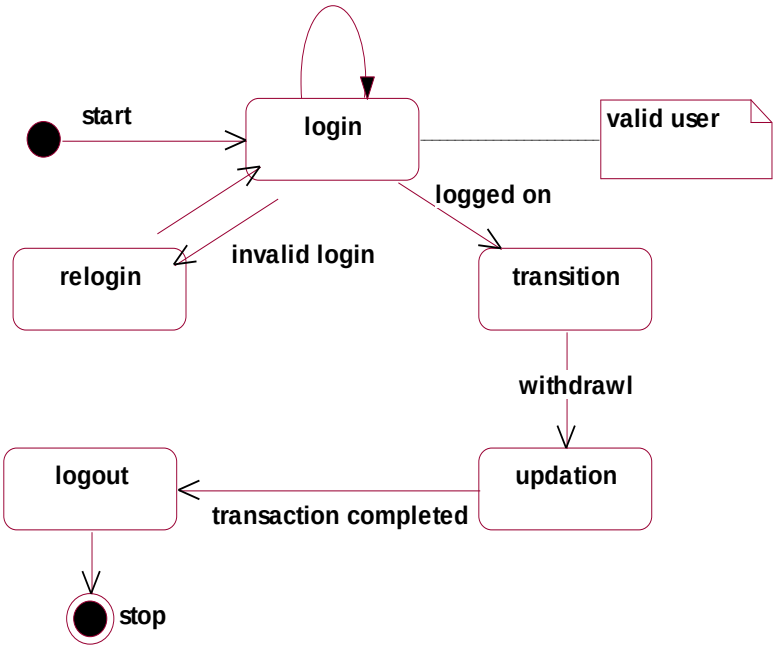
Withdraw class diagram :-



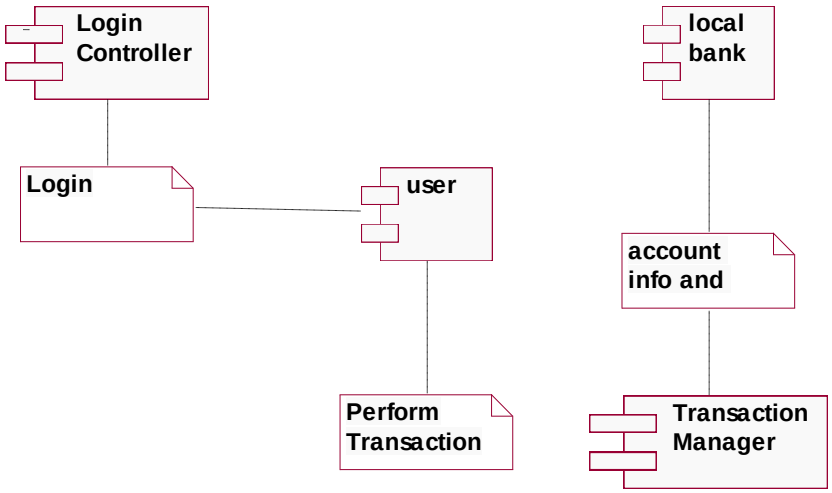
Withdraw cash class diagram:-



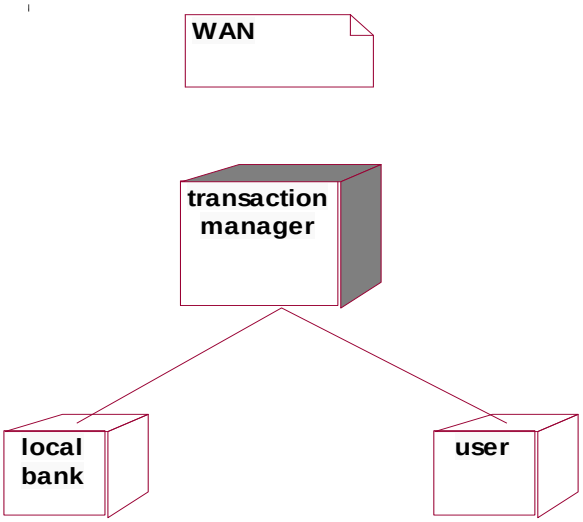
State Transition diagram :



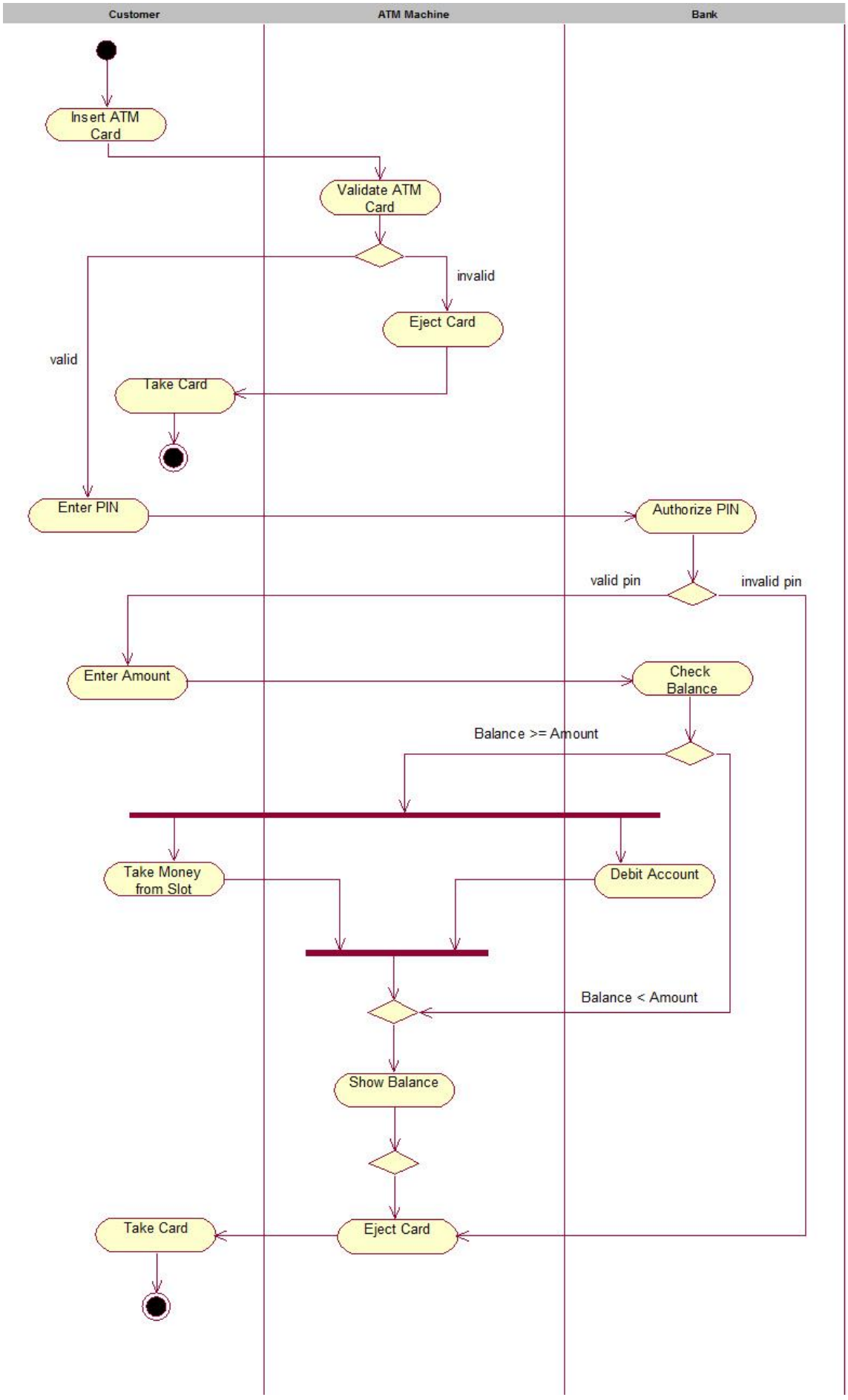
Component Diagram: -



Deployment Diagram:-



Activity diagram:-


Customer
ATM Machine
Bank
Insert ATM Card
Validate ATM Card
invalid
Eject Card
valid
Take Card
Enter PIN
Authorize PIN
valid pin
invalid pin
Enter Amount
Check Balance
Balance >= Amount
Take Money from Slot
Debit Account
Balance < Amount
Show Balance
Take Card
Eject Card