UNIT-II

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

What is the UML?

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

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

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

Why We Model

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

The Importance of Modeling

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

material, is one that has a sustainable business

What, then, is a model? Simply put,

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

Why do we model? There is one fundamental reason.

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

Through modeling, we achieve four aims

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

Modeling is not just for big systems. Even the software equivalent of a dog house can benefit from some modeling.

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

Principles of Modeling

Four principles of modeling:

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

Object Oriented

Modeling Two

Approaches:

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

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

An Overview of the UML

The UML is a language for

- Visualizing
- Specifying
- Constructing
- Documenting

The UML Is a Language for Documenting

A healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include (but are not limited to)

- Requirements
- Architecture
- Design
- Source code
- Project plans
- Tests
- Prototypes
- Releases

Where Can the UML Be Used?

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

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

A Conceptual Model of the UML

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

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

Building Blocks of the UML:

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

Things in the UML

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

Structural Things

- *Structural things* are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things.
- Classes
- Interface
- Cases
- Active Classes
- Components
- Nodes
- Collaborations

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

Figure : Classes

Interfaces

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

Figure :Interfaces

Collaborations:

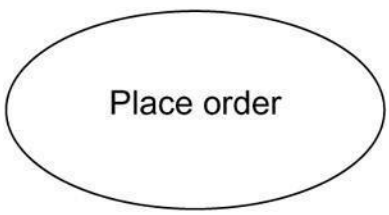
- *A collaboration* defines an interaction. These collaborations therefore represent the implementation of patterns that make up a system. Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its name

Figure: Collaborations

Use Cases:

- A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name

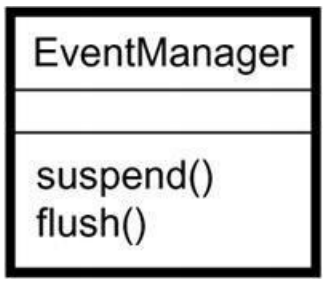
Figure :Use Cases



Active Classes:

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

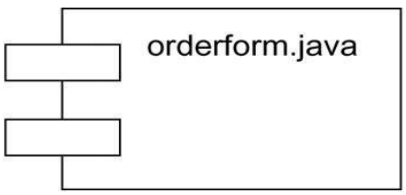
Figure :Active Classes



Components:

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

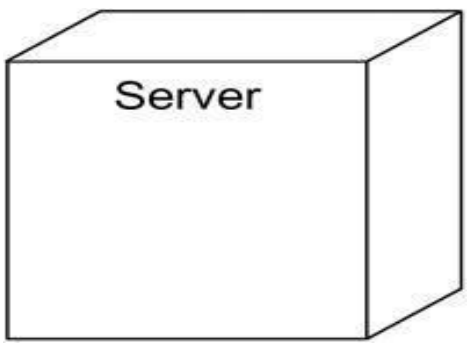
Figure :Components



Nodes:

- A *node* is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability.
- A set of components may reside on a node and may also migrate from node to node. Graphically, a node is rendered as a cube, usually including only its name.

Figure :Nodes



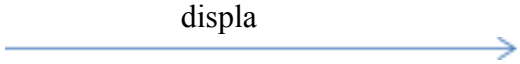
Behavioral Things:

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

1. Messages
2. States

Messages:

- An *interaction* is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose. Graphically, a message is rendered as a directed line, almost always including the name of its operation.



States:

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

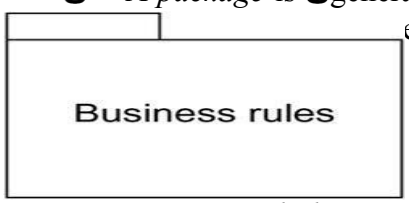
Grouping Things:

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

Packages:

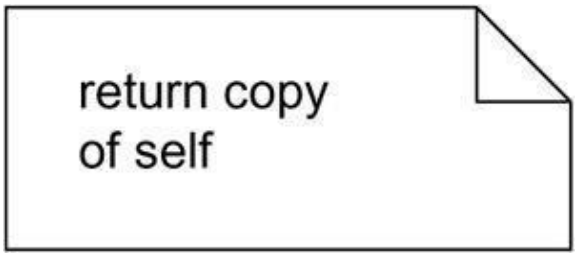
Figure: Packages

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



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

Figure: Notes



There are four kinds of relationships in the UML:

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

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

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

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

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

UML Diagrams

- A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and paths (relationships).
- A diagram represents an elided view of the elements that make up a system.
- In theory, a diagram may contain any combination of things and relationships.

- In practice, a small number of common combinations arise, which are consistent with the five most useful views that comprise the architecture of a software intensive system

The UML includes Nine kinds of diagrams:

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

1. Class diagram shows a set of classes, interfaces, and collaborations and their relationships. These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.
2. Object diagram shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams.
3. Use case diagram shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system.
4. Sequence diagram is an interaction diagram that emphasizes the time-ordering of messages;
5. Collaboration diagram a communication diagram is an interaction diagram that emphasizes the structural organization of the objects or roles that send and receive messages.
6. Statechart diagram shows a state machine, consisting of states, transitions, events, and activities. A state diagrams shows the dynamic view of an object.
7. Activity diagram shows the structure of a process or other computation as the flow of control and data from step to step within the computation. Activity diagrams address the dynamic view of a system.
8. Component diagram is shows an encapsulated class and its interfaces, ports, and internal structure consisting of nested components and connectors. Component diagrams address the static design implementation view of a system.
9. Deployment diagram shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static

deployment view of an architecture

What is Legal UML?

The UML has syntactic and semantic rules for

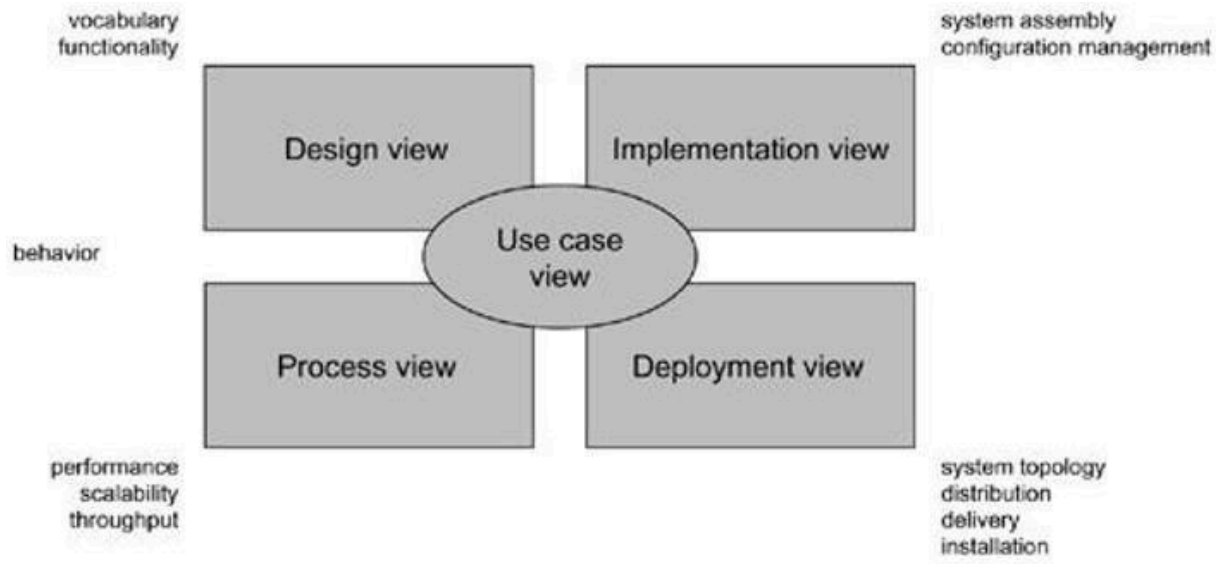
- Names What you can call things, relationships, and diagrams
- Scope The context that gives specific meaning to a name
- Visibility How those names can be seen and used by others
- Integrity How things properly and consistently relate to one another
- Execution What it means to run or simulate a dynamic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

Architecture

- Architecture refers to the different perspectives from which a complex system can be viewed.
- Visualizing, specifying, constructing, and documenting a software-intensive system demands that the system be viewed from a number of perspectives.
- The architecture of a software-intensive system is best described by five interlocking views:

  - Use case view: system as seen by users, analysts and testers.
  - Design view: classes, interfaces and collaborations that make up the system.
  - Process view: active classes (threads).
  - Implementation view: files that comprise the system.
  - Deployment view: nodes on which SW resides.

- Each view is a projection into the organization and structure of the system, focused on a particular aspect of that system.

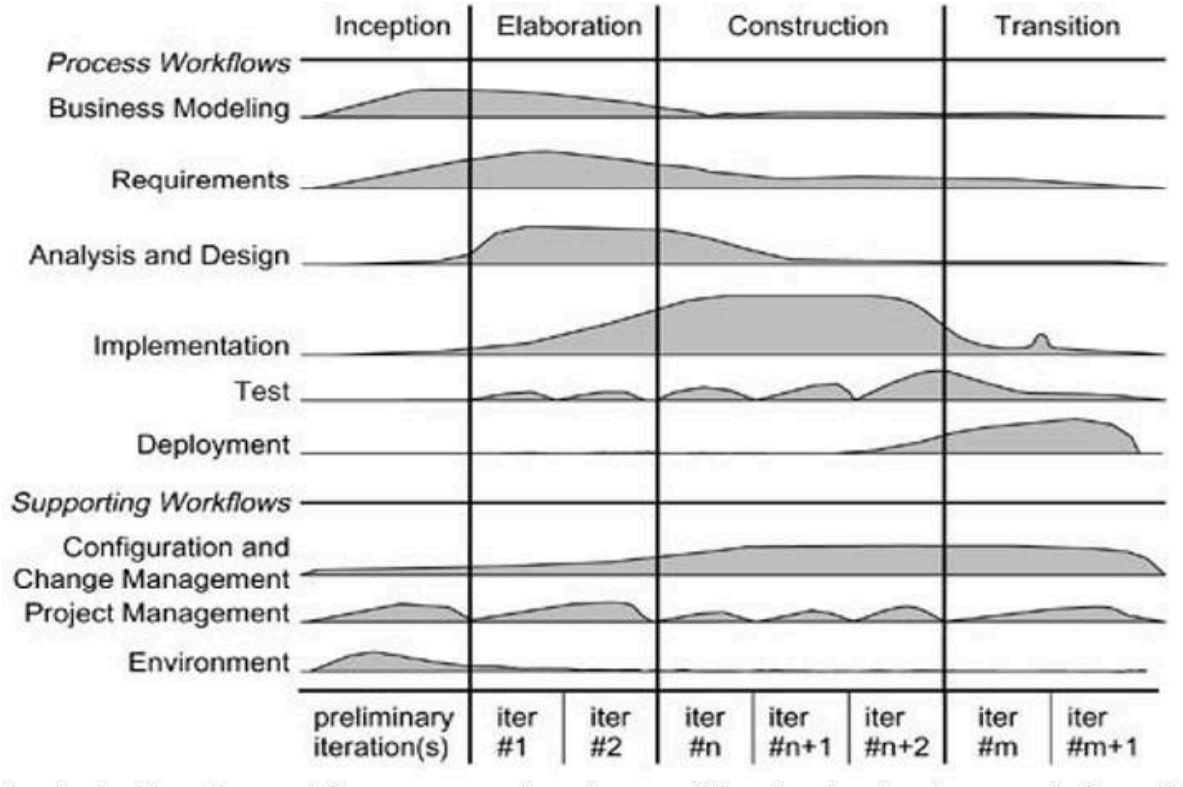


Each of these five views can stand alone so that different stakeholders can focus on the issues of the system's architecture that most concern them.

Software Development Life Cycle

- UML is involved in each phase of the software development life cycle.
- The UML development process is
  - Use case driven
  - Use case driven means that use cases are used as a primary artifact for establishing the desired behavior of the system, for verifying and validating the system's architecture, for testing, and for communicating among the stakeholders of the project.
  - Architecture-centric
  - Architecture-centric means that a system's architecture is used as a primary artifact for conceptualizing, constructing, managing, and evolving the system under development.
  - Iterative and incremental
  - An iterative process is one that involves managing a stream of executable releases. An is one that involves the continuous integration of the system's architecture to produce these releases, with each new release embodying incremental improvements over the other.

Software Development Life Cycle



- **Inception** is the first phase of the process, when the seed idea for the development is brought up to the point of being at least internally - sufficiently well-founded to warrant entering into the elaboration phase.
- **Elaboration** is the second phase of the process, when the product vision and its architecture are defined. In this phase, the system's requirements are articulated, prioritized, and baselined. A system's requirements may range from general vision statements to precise evaluation criteria, each specifying particular functional or nonfunctional behavior and each providing a basis for testing.
- **Construction** is the third phase of the process, when the software is brought from an executable architectural baseline to being ready to be transitioned to the user community. Here also, the system's requirements and especially its evaluation criteria are constantly reexamined against the business needs of the project, and resources are allocated as appropriate to actively attack risks to the project.
- **Transition** is the fourth phase of the process, when the software is turned into the hands of the user community. Rarely does the software development process end here, for even during this phase, the system is continuously improved, bugs are eradicated, and features that didn't make an earlier release are added.

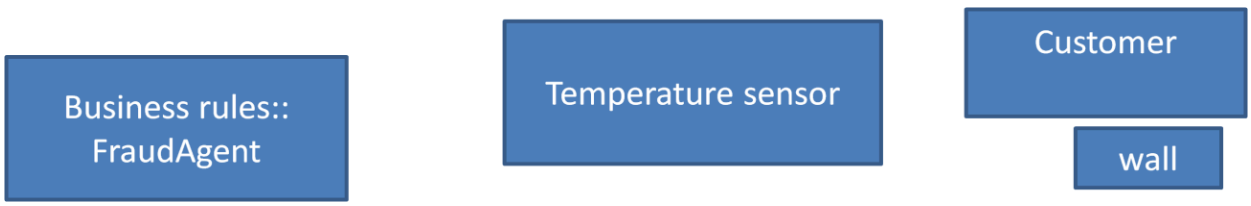
CLASSES

Classes

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

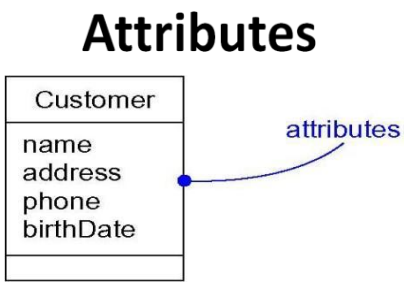
Name

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



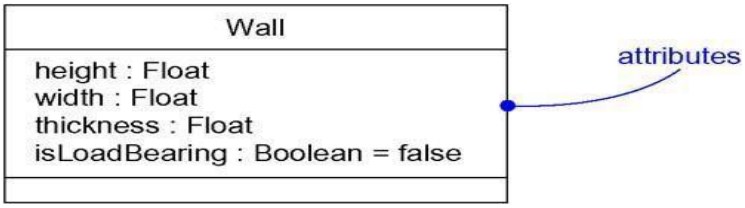
Attributes

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



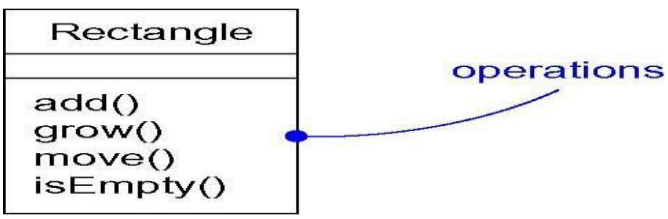
further specify an attribute by stating its class and possibly a default initial value

Attributes and Their Class



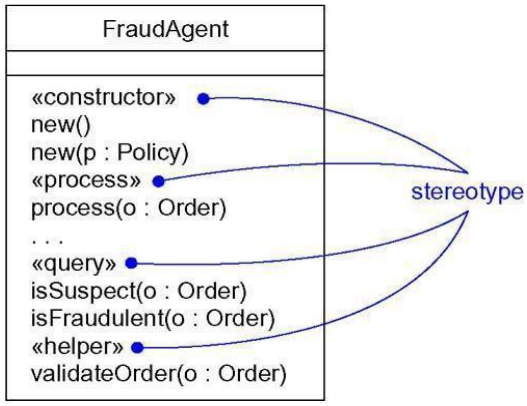
Operations

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



Organizing attributes and relationships

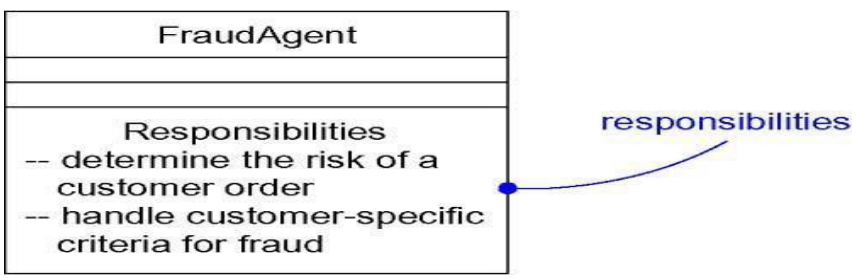
- When drawing a class, you don't have to show every attribute and every operation at once.
- Meaning that you can choose to show only some or none of a class's attributes and operations
- Explicitly specify that there are more attributes or properties than shown by ending each list with an ellipsis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



Responsibilities

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

Fig: Responsibilities



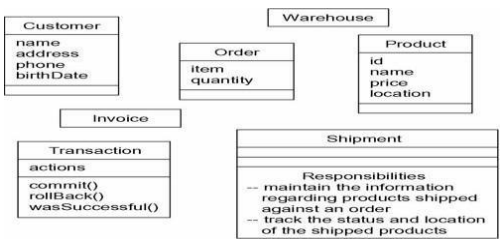
Common modeling techniques

1. Modeling the vocabulary of a system

To model the vocabulary of a system

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

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

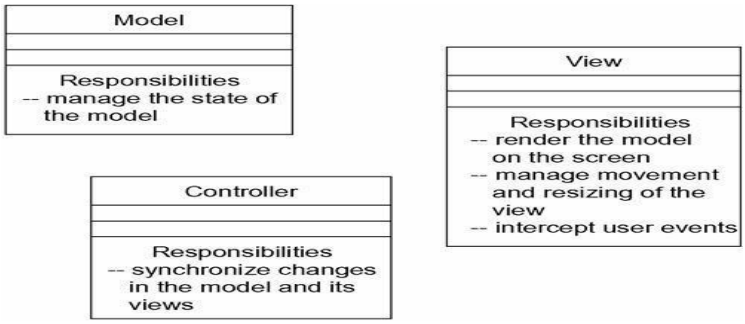


2. Modeling the Distribution of responsibilities in a System

To model the distribution of responsibilities in a System

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

Modeling the Distribution of responsibilities in a System



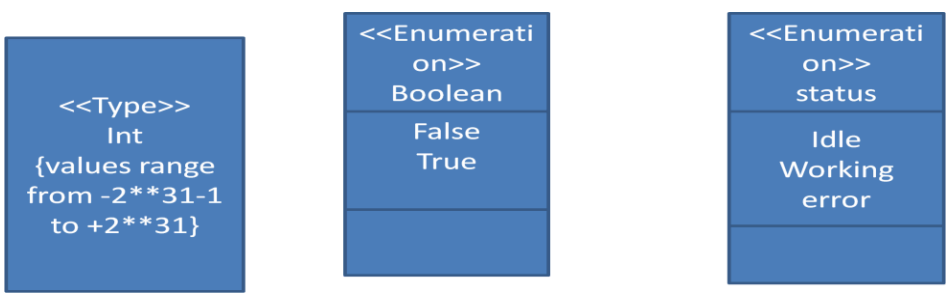
3. Modeling Non software things

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

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

Modeling Non software Things

Modeling Non Software things



4. Modeling Primitive Types

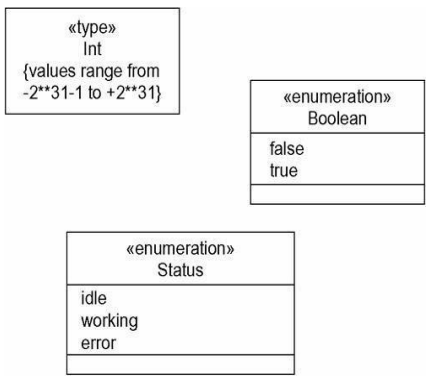
- At the other extreme, the things you model may be drawn directly from the programming language you are using to implement a solution.
- Typically, these abstractions involve primitive types, such as integers, characters, strings, and even enumeration types, that you might create

yourself.

To model primitive types,

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

Fig: Modeling Primitive Types



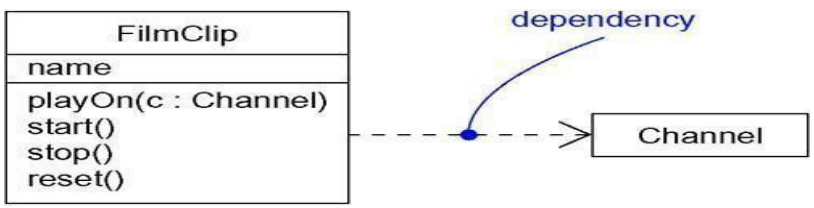
Relationships

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

Dependency

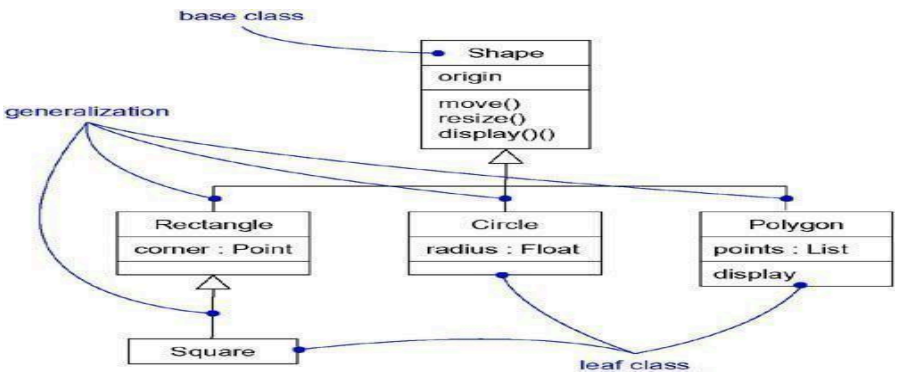
A *dependency* is a using relationship that states that a change in specification of one thing (for example, class **Event**) may affect another thing that uses it (for example, class **Window**), but not necessarily the reverse.

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



Generalization

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



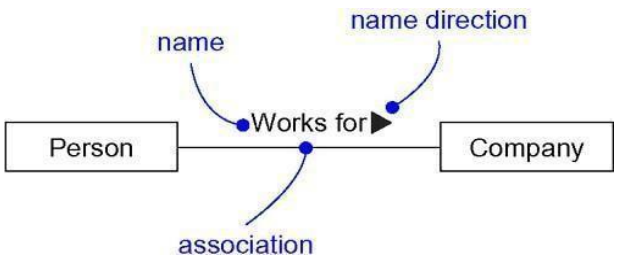
Association

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

There are four adornments that apply to associations.

Name

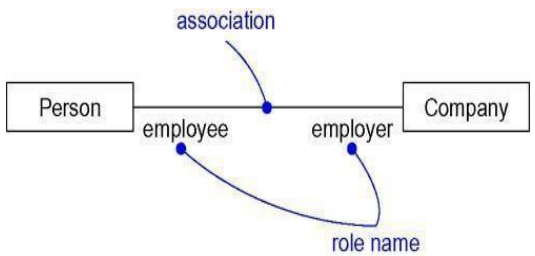
An association can have a name, and you use that name to describe the nature of the relationship. So that there is no ambiguity about its meaning, you can give a direction to the name by providing a direction triangle that points in the direction you intend to read the name, as shown in Figure



There are four adornments that apply to associations.

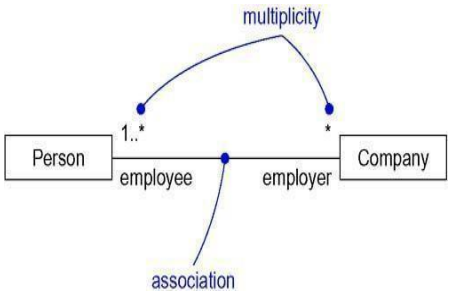
Role

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



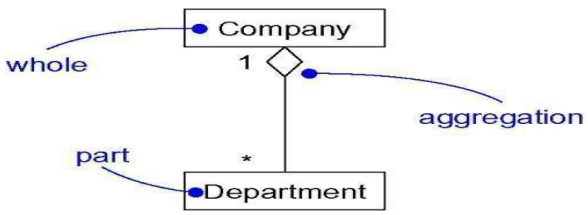
Multiplicity

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



Aggregation

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

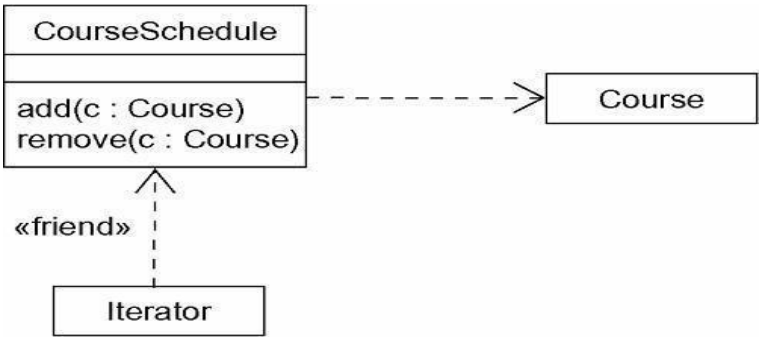


Common Modeling Techniques

1. Modeling simple dependencies

-To model this using relationship

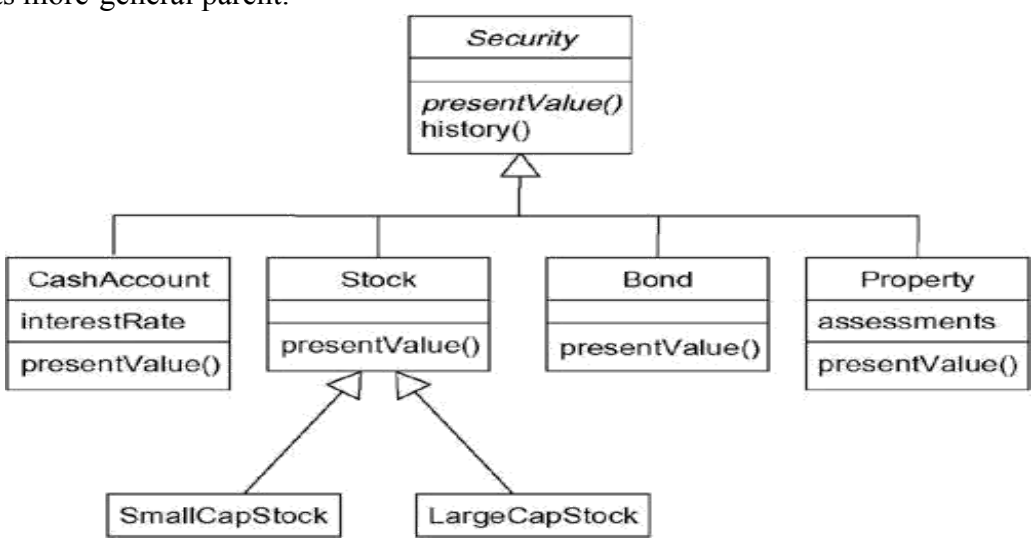
1) Create a dependency pointing from the class with the operation to the class used as a parameter in the operation.



2. Modeling Single Inheritance

To model inheritance relationships,

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



3. Modeling Structural Relationships

To model structural relationships,

1. For each pair of classes, if you need to navigate from objects of one to objects of another, specify an association between the two. This is a data-driven view of

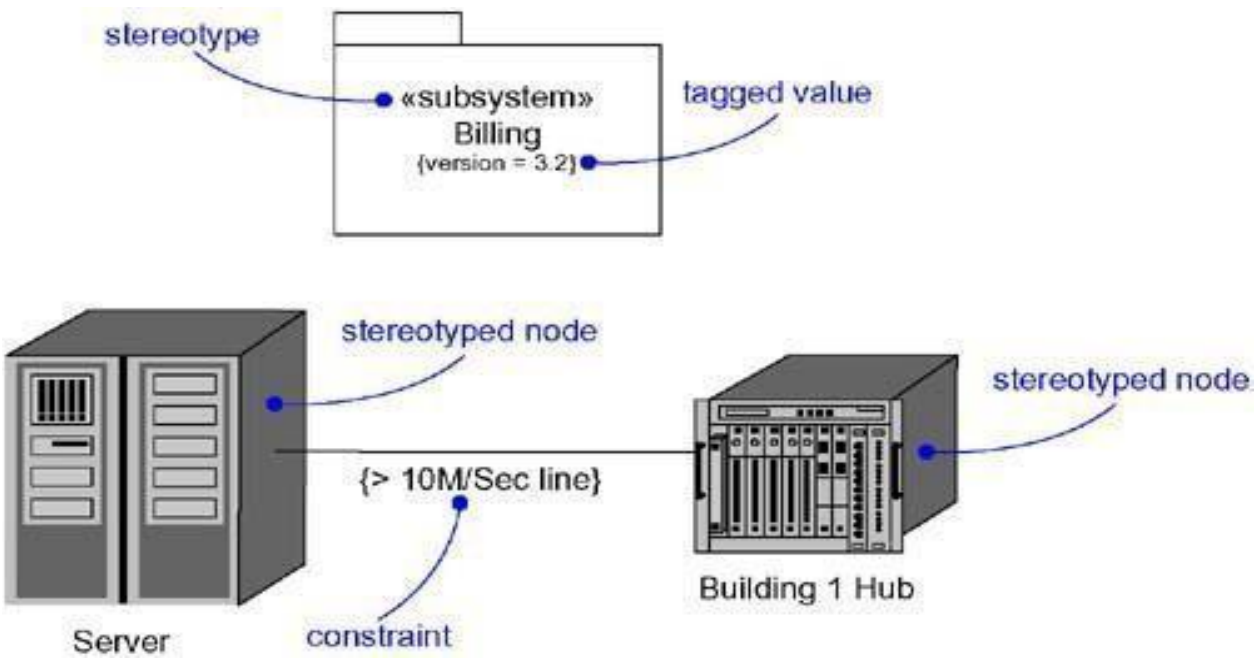
associations.

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

COMMON MECHANISMS

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

Fig: Stereotypes, Tagged Values, and constraints



A ***note*** is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a

dog-eared corner, together with a textual or graphical comment.

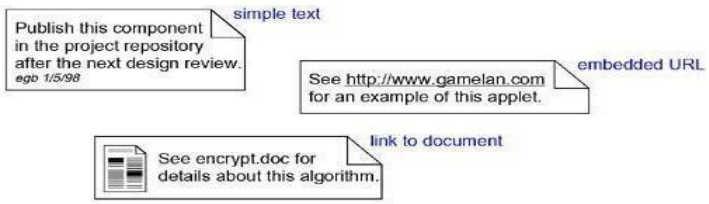
A ***stereotype*** is an extension of the vocabulary of the UML, allowing to create new kinds of building blocks similar to existing ones but specific to problem. Graphically, a stereotype is rendered as a name enclosed by guillemets (<< >>) and placed above the name of another element.

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

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

Notes

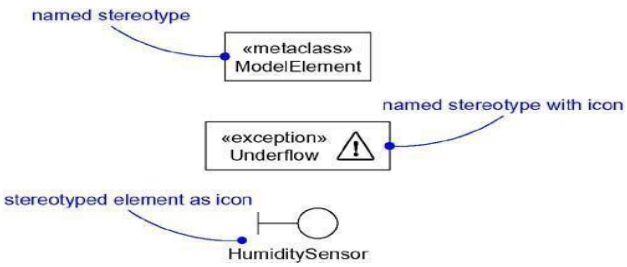
- A note that renders a comment has no semantic impact, meaning that its contents do not alter the meaning of the model to which it is attached. Notes are used to specify things like requirements, observations, reviews, and explanations, in addition to rendering constraints.
- A note may contain any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



Other Adornments

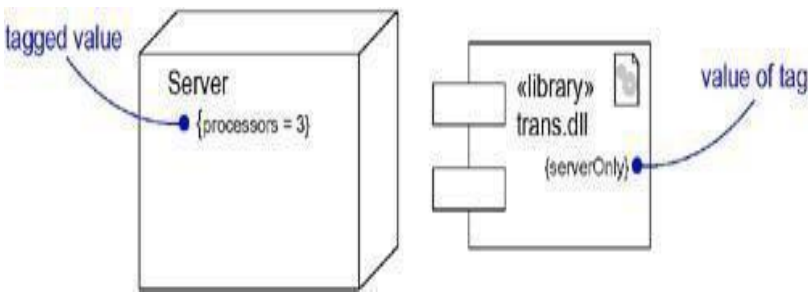
Adornments are textual or graphical items that are added to an element's basic notation and are used to visualize details from the element's specification.

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



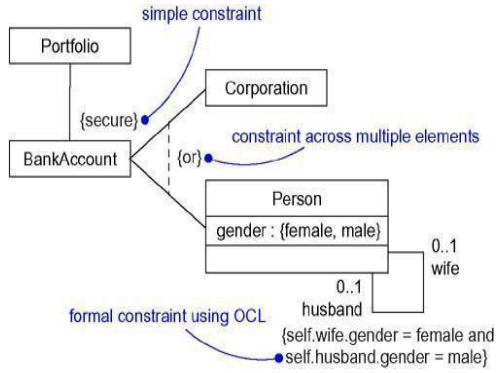
Tagged Values

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



Constraints

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



Standard Elements

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

Stereotype

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

Documentation

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

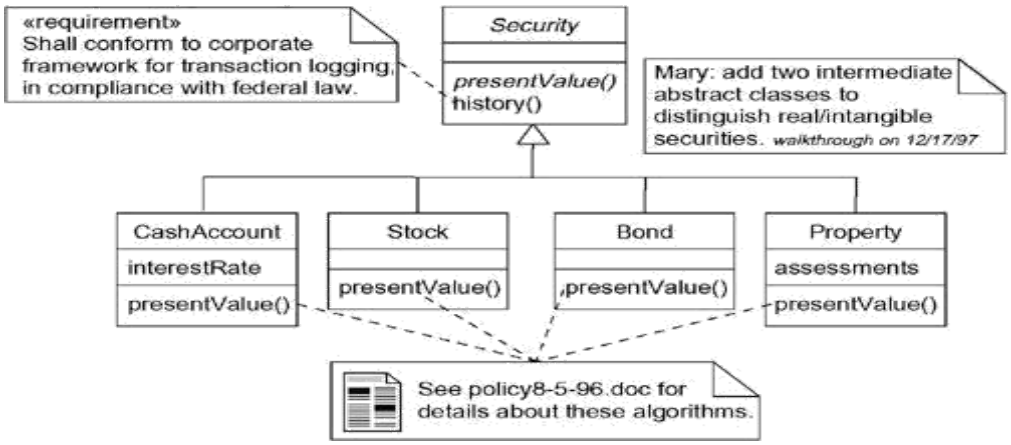
Common Modeling Techniques

1. Modeling Comments

- To model a comment

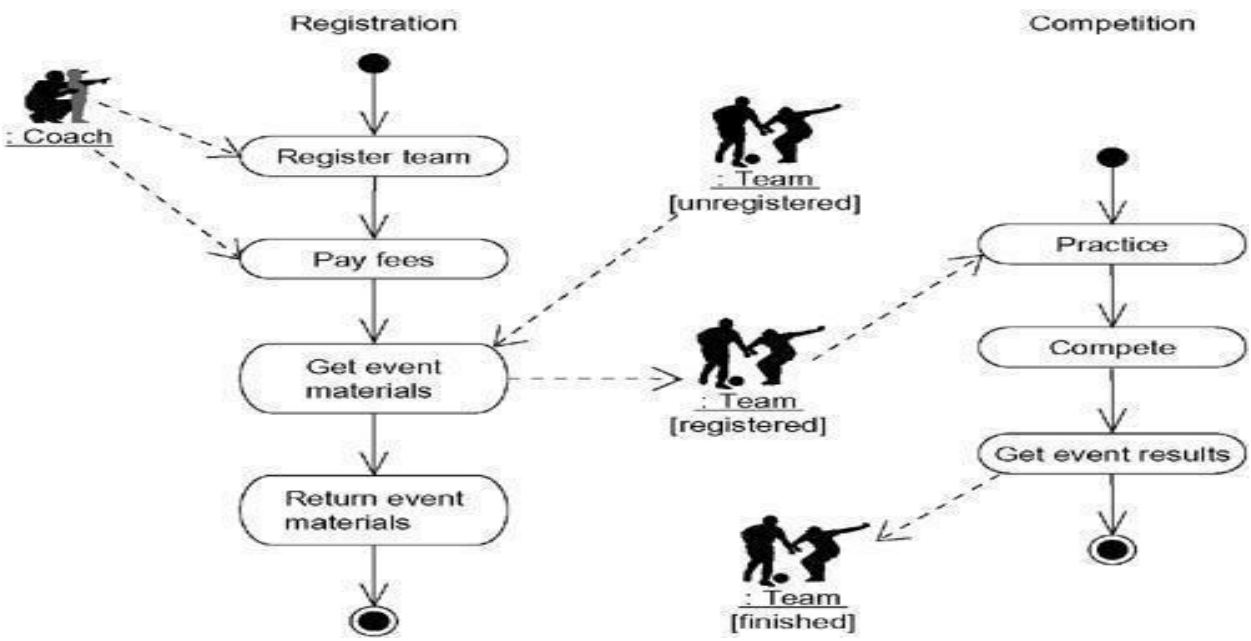
1) Put your comment as text in a note and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a note to its elements using a dependency relationship.
2) Remember that you can hide or make visible the elements of your model as you see fit. This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

the others.



2. Modeling New Building Blocks

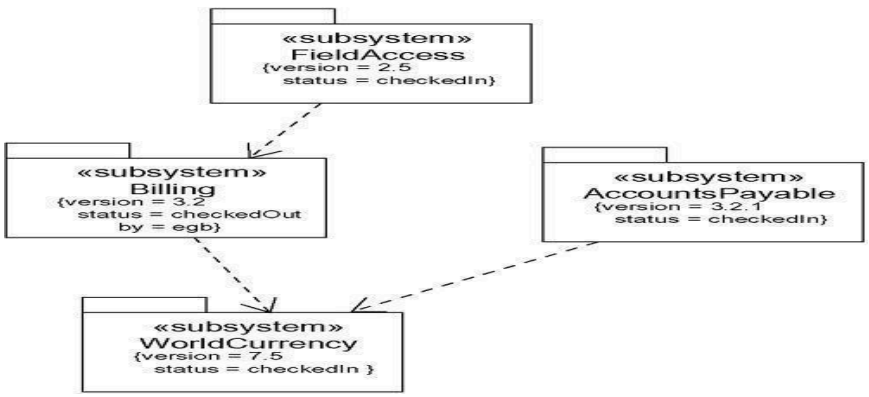
1) Make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are there's already some standard stereotype that will do what you want.
2) If you're convinced there's no other way to express these semantics, identify the primitive thing in the UML that's most like what you want to model (for example, class, interface, component, node, association, and so on) and define a new stereotype for that thing.
3) Specify the common properties and semantics that go beyond the basic element being stereotyped by defining a set of tagged values and constraints for the stereotype.
4) If you want these stereotype elements to have a distinctive visual cue, define a new icon for the stereotype.



3) Modeling New Properties

1) First, make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there's already some standard tagged value that will do what you want.
2) If you're convinced there's no other way to express these semantics, add this new property to an individual element or a stereotype. The rules of generalization apply -- tagged values defined for one kind of element apply to its children.

Fig: Modeling New Properties



4) To model new semantics

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

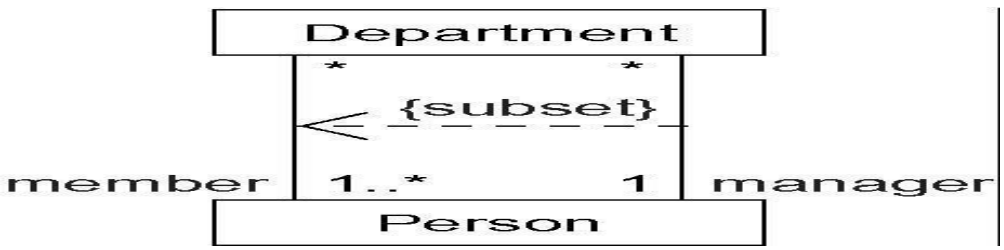


Fig: Modeling New Semantics

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

DIAGRAMS

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

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

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

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

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

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

Common Modeling Techniques

1. Modeling Different Views of a System

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

2) Modeling Different Levels of Abstraction

- Consider the needs of your readers, and start with a given model.
- If your reader is using the model to construct an implementation, she'll need diagrams that are at a lower level of abstraction, which means that they'll need to

reveal a lot of detail. The model to present a conceptual model to an end user, then use the diagrams that are at a higher level of abstraction, which means that they'll hide a lot of detail.

- Depending on where you land in this spectrum of low-to-high levels of abstraction, create a diagram at the right level of abstraction by hiding or revealing the following four categories of things from the model.

Building blocks and relationships:

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

Adornments:

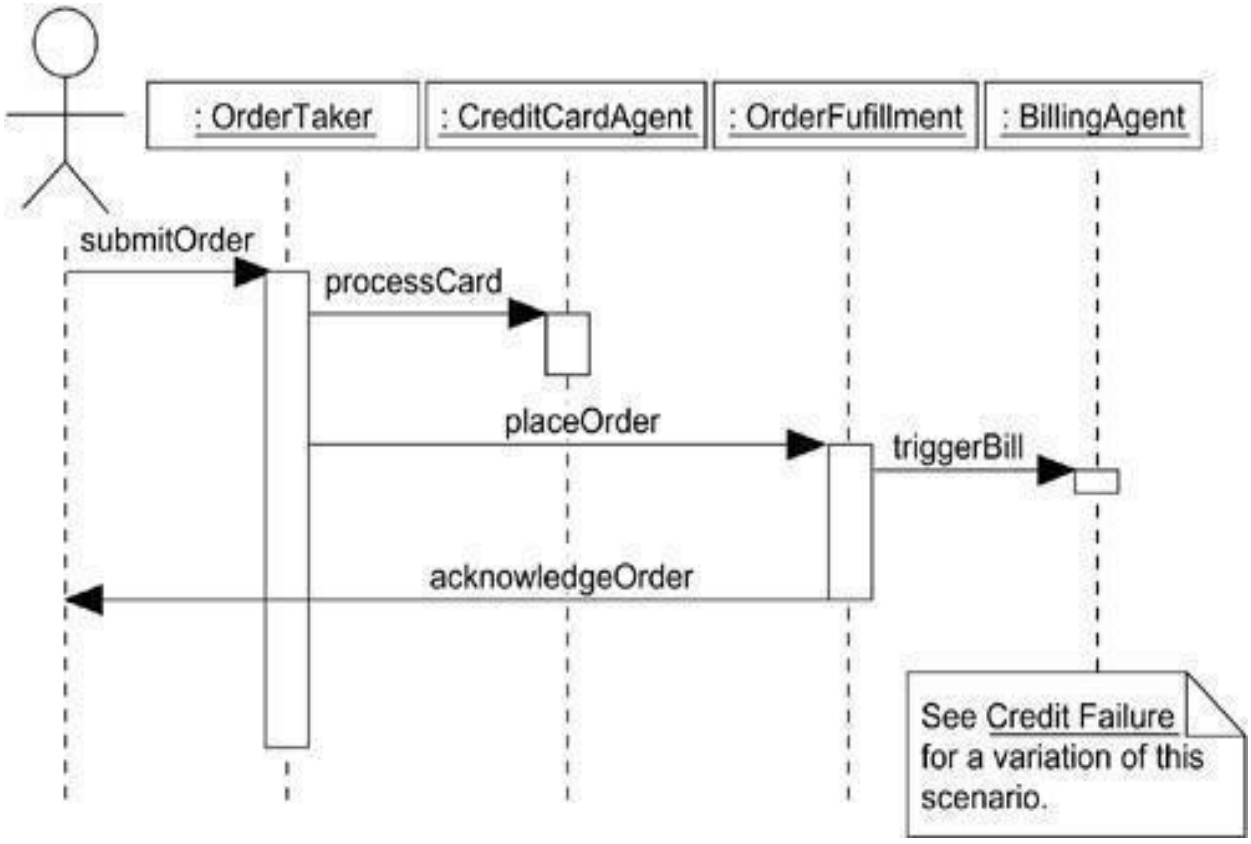
- Reveal only the adornments of these building blocks and relationships that are essential to understanding the intent.

Flow:

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

Stereotypes:

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



3) Modeling Complex Views

To model complex views,

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