Object-oriented modeling and design

Chapter 1

Introduction

Object oriented means a collection of discrete objects that incorporate both data structure and behavior.

The characteristics required by an OO approach include 4 aspects:

Identity, classification, inheritance and polymorphism.

Identity means data is quantized into discrete, distinguishable entities called objects. Objects can be concrete such as a file in a file system. Each object has its own identity. Two objects are distinct even if all there attribute values are identical.

Classification means that objects with the same data structure and behavior are grouped into a class. Each object is said to be an instance of a class. An object has its own value for each attribute but shares the attribute names and operations with other instances of the class.

Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship. A super class has general information that subclasses refine and elaborate. Each subclass inherits all the features of its super class and adds its own unique features.

Polymorphism means that the same operation may behave differently for different classes. The move operation for example behaves differently for a pawn than for the queen in the chess game.

An operation is a procedure or transformation that an object performs or is subject to. An implementation of an operation by a specific class is called a method.

OO Development:

Development refers to the software life cycle: analysis, design and implementation. The essence of OO Development is the identification and organization of application concepts, rather than their final representation in a programming language.

Modeling Concepts: In the past, much of the programming languages focused on implementation rather than analysis and design.

OO development is a conceptual process independent of a programming language until final stages.OO development is fundamentally a way of thinking and not a programming technique.

Its greatest benefits comes from helping specifiers, developers and customers express abstract concepts clearly and communicate them to each other.

OO Methodology:

The process for OO development and graphical notation for representing OO concepts consists of building a model of an application and then adding details to it during design. The methodology has the following stages:

- **System Conception:** Software development begins with business analysis or users conceiving an application and formulating tentative requirements.
- **Analysis:** The analyst scrutinizes and rigorously restates the requirements from system conception by constructing models. The analyst must work with the requestor to understand the problem, because problem statements are rarely complete or correct. The analysis model is a precise abstraction of what the desired system must do, not how it will be done.

It should not contain implementation decisions.

The analysis model has 2 parts:

The **domain model**, a description of the real-world objects reflected within the system and the **application – model**, a description of the parts of the application system itself that are visible to the user.

Eg: domain objects for a stock broker application might include stock, bond, trade and commission.

Application objects might control the execution of trades and present the results.

- **System design:** The development teams devise a high – level strategy – the system architecture for solving the application problem. They also establish policies that will serve as a default for the subsequent, more detailed portions of design. The system designer must decide what performance characteristics to optimize, choose a strategy of attacking the problem and make tentative resource allocations.
- **Class design:** The class designer adds details to the analysis model in accordance with the system design strategy. The focus of class design is the data structures and algorithms needed to implement each class.
- **Implementation:** Implementers translate the classes and relationships developed during class design into a particular programming language database or hardware. Programming should be straight forward, because all of the hard decisions should have already been

made. Some classes are not part of analysis but are introduced during design or implementation

- Eg: data structures such as trees, hash tables and linked lists are rarely resent in the real world and are not visible to users.

Designers introduce them to support particular algorithms. Such objects exist within a computer and are not directly observable.

Testing must be part of an overall philosophy of quality control that occurs throughout the life cycle. Developers must check analysis models against kinds of errors, in addition to the testing implementations for correctness.

Three Models:

Three kinds of models are used to describe a system from different viewpoints:

The Class Model for the objects in the system and their relationships; the State Model for the life history of objects; and the Interaction Model for the interactions among objects.

A complete description of a system requires models from all 3 viewpoints.

The class model describes the static structure of the objects in a system and their relationships. The class model contains class diagrams. A class diagram is a graph whose nodes are classes and whose arcs are relationships among classes.

The state model describes the aspects of an object that change over time. The state diagram is a graph whose nodes are states and whose arcs are transitions between states caused by events.

The interaction model describes how the objects in a system cooperate to achieve broader results.

The interaction model starts with use case that are then elaborated with sequence and activity diagrams. A use case focuses on the functionality of a system i.e, what a system does for users. A sequence diagram elaborates important processing steps.

OO Themes:

- Abstraction means focusing on what an object is and does, before deciding how to implement it.
- Encapsulation separates the external aspects of an object that are accessible to other objects, from the internal implementation details that are hidden from other objects.

- Combining data and behavior: The caller of an operation need not consider how many implementations exist. Operator polymorphism shifts the burden of deciding what implementation to use from the calling code to the class hierarchy.
- Sharing: OO technologies promote sharing at different levels. Inheritance of both data structure and behavior lets subclasses share common code. This sharing via inheritance is one of the main advantages of OO languages.

  OO development not only lets you share information within an application but also offers the prospect of reusing designs and code on future projects.
- Emphasis on the essence of object:

  OO technology stresses what an object is, rather than how it is used. The uses of an object depend on the details of the application and often change during development.
- Synergy: Identity, classification, polymorphism and inheritance characterize OO languages. Each of these concepts can be used in isolaition but together they complement each other synergistically.

Chapter 2

Modeling As A Design Technique

A model is an abstraction of something for the purpose of understanding it before building it. Because a model emits nonessential details it is easier to manipulate than original entity.

Modeling:

Designers build many kinds of models for various purposes before constructing things. Examples include architectural models to show customers, airplane scale models for wind – tunel tests, pencil sketches for composition of all paintings,.. Models serve several purposes.

- **Testing a physical entity before building it:**

  Engineers test scale models of airplanes, cars and boats in wind tunnels and water tanks to improve their dynamics. Recent advances in computation permit the simulation of many physical structures without the need to build physical models. Both physical and computer models are usually cheaper than building a complete system and enable early correction if flaws.

- **Communication with customers:** Architects and product designers build models to show their customers. Mock ups are demonstration products that imitate some or all of the external behavior of a system.

- **Visualization:** Storyboards of movies, television shows and advertisements let writers see how their ideas flow. They can modify awkward transitions, dangling ends and unnecessary segments before detailed writing begins.

- **Reduction of complexity:** The main reason for modeling is to deal with systems that are too complex to understand directly. Models reduce complexity y separating out a small number of important things to deal with at a time.

Abstraction: is the selective examination of certain aspects of a problem. The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant.

A good model captures the crucial aspects of a problem and omits the others.

The 3 models: The class model represents the static, structural, “data” aspects of a system.

The State model represents the temporal, behavioral, “control” aspects of a system.

The Interaction model represents the collaboration of individual objects, the “interaction” aspects of a system.

The 3 kinds of models separate a system into distinct views. The different models are not completely independent but each model can be examined and understood by itself to a large extent.

The different models have limited and explicit interconnection. It is always possible to create bad designs in which the 3 models are so intertwined that they cannot be separated but a good design isolates the different aspects of a system and limits the coupling between them.

Class Model**:** describes the structure of objects in a system, their identity, their relationships to other objects, their attributes and their operations.

Our goal in constructing a class model is to capture those concepts from the real world that are important to an application. In modeling an engineering problem, the class model should contain terms familiar to engineers.

Class diagrams express the class model. Classes define the attribute values carried by each object and the operations that each object performs or undergoes.

State Model: describes those aspects of objects concerned with time and the sequencing of operations – events that mark changes, states that context for events and the organization of events and states.

State diagrams express the state model. Each state diagram shows the state and event sequences permitted in a system for one class of objects.

Actions and events in a state diagram become operations on objects in the class model.

Interaction Model: describes interactions between objects – how individual objects collaborate to achieve the behavior of the system as a whole.

Use cases, sequence diagrams and activity diagrams document the interaction model. Use cases document major themes for interaction between the system and outside actors. Sequence diagrams show the objects that interact and the time sequence of their interactions. Activity diagrams show the flow of control among the processing steps of a computation.

Chapter 3

Class Modeling

Objects and Class Concepts:

Objects: The purpose of class modeling is to describe objects. An object is a concept, abstraction or thing with identity that has meaning for an application.

Some objects have real world counter parts while others are conceptual entities. Still others are introduced for implementation reasons and have no correspondence to physical reality.

All objects have identity and are distinguishable. Identical twins are 2 distinct persons, even though they may look the same. The term identity means that objects are distinguished by their inherit existence and not by descriptive properties that they may have.

Classes: An object is an instance or occurrence of a class. A class describes a group of objects with the same properties (attributes), behavior (operations), kinds of relationships and semantics. Person, company, process are all classes.

Each person has name and birth date and may work at a job. Each process has an owner, priority and list of required resources. Classes often appear as common nouns and noun phrases in problem descriptions and discussions with users.

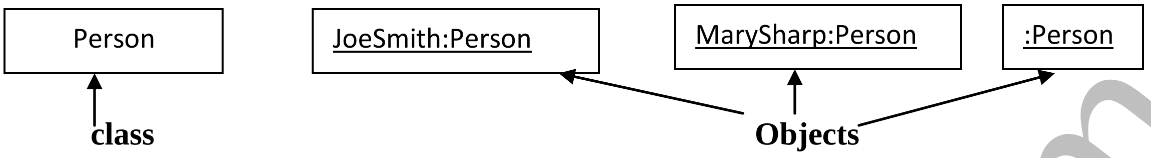
Objects in a class have same attributes and forms of behavior. Most objects derive their individuality from differences in their attribute values and specific relationships to other objects. The objects in a class share a common semantic purpose, above and beyond the requirement of common attributes and behavior.

Eg: a barn and a horse may both have a cost and an age. If both were regarded as purely financial assets, they could belong to the same class. If the developer took into consideration that a person paints a barn and feeds a horse, they would be modeled as distinct classes. The interpretation of semantics depends on the purpose of each application and is a matter of judgment.

Class diagrams: There are two kinds of models of structure – class diagrams and object diagrams.

Class diagrams provide a graphic notation for modeling classes and their relationships thereby describing possible objects.

An object diagram shows individual objects and their relationships. A class diagram corresponds to an infinite set of object diagrams.

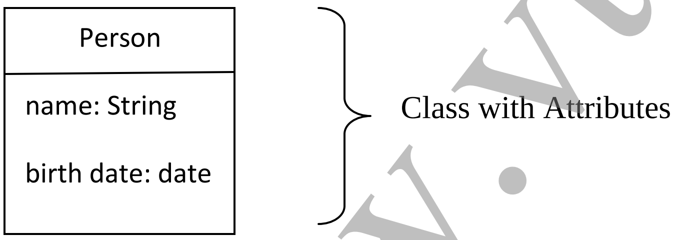


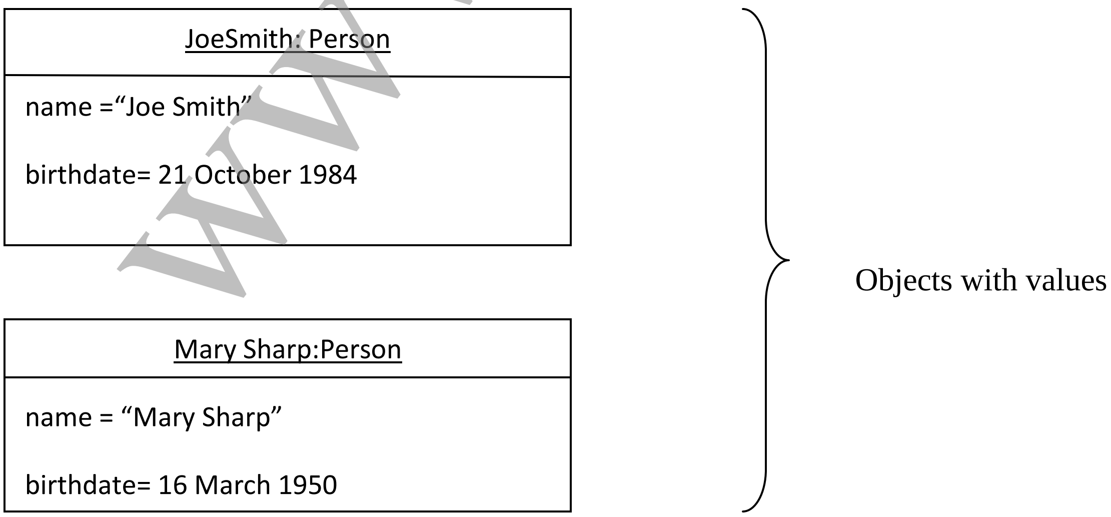
The UML symbol for an object is a box with an object name followed by a colon and the class name. Both the names are underlined. Convention is to list both names in bold face.

The UML symbol for a class also is a box. Our convention is to list the class name in bold face, center the name and capitalize the first letter.

Values and Attributes: A value is a piece of data. An attribute is a named property of a class that describes a value held by each object of the class. You can find attributes by looking for adjectives or by abstracting typical values. Objects is to class as value is to attribute.

Name, birth date and weight are attributes of Person objects. Each attribute has a value for each object. Each attribute name is unique within a class.





Operations and Methods:

An operation is a function or procedure that may be applied to or by objects in a class. Hire , fire and payDividend are operations on class company

All objects in a class share the same operations. Each operation has a target object as an implicit argument. The same operation may apply to many different classes. Such an operation is polymorphic.

A method is the implementation of an operation for a class.

Eg: A class File may have an operation print. You could implement different methods to print ASCII files, binary files and print digitized picture files. All methods logically perform the same task. Thus referred by generic operation print. However a different piece of code may implement each method.

When an operation has methods on several classes, it is important that the methods all have the same signature – the number and types of arguments and the type of result value.

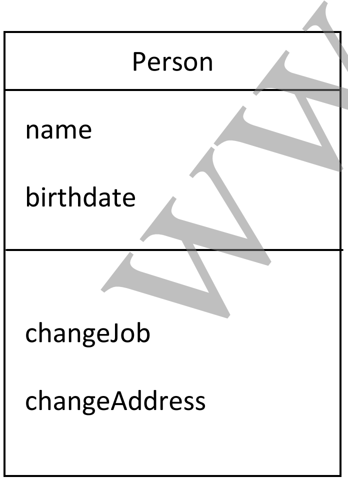
Example: The class Person has attributes name and birth date and operations changeJob and changeAddress. They are the features of Person.

Feature is a generic word for either an attribute or an operation.

The UML notation is to list operations in third compartment of the class box. Our convention is to list the operation name in regular face, left align and use a lower case letter for the first letter.

Optional details such as an argument list; commas separate the arguments. A colon precedes the result type.

An empty argument list in parenthesis shows explicitly that there are no arguments, otherwise conclusions cannot be drawn.



Summary of notations:

A box represents a class and may have as many as 3 compartments from top to bottom: classname, list of attributes and list of operations. Optional details such as type and default value may follow each attribute name and optional details such as argument list and result type may follow each operation name.

The direction indicates whether an argument is an input(in), output(out) or an input argument that can be modified (inout). A colon precedes the type. An equal sign precedes the default value.

The attributes and operation compartments are optional. A missing compartment means they are unspecified.

In contrast an empty compartment means that attributes (operations) are specifid and that there are none.

Link and Association Concepts:

They are the means for establishing relationships among objects and classes.

Links and Associations:

A link is a physical or conceptual connection among objects.

Example: Joe Smith Works-For Simplex Company. Most links relate 2 objects, but some links relates 3 or more objects. A link is an instance of an association.

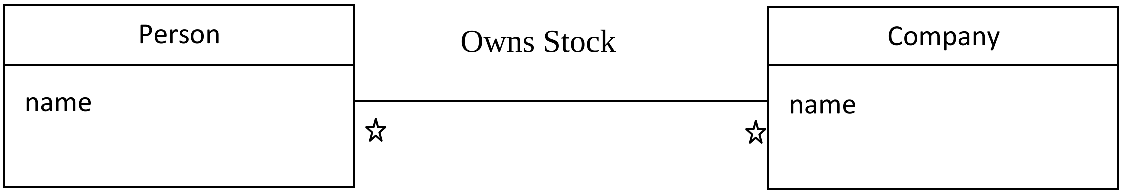
An association is a description of a group of links with common structure and common semantics.

Example: a Person WorksFor a company. The links of an association connect objects from the same classes.

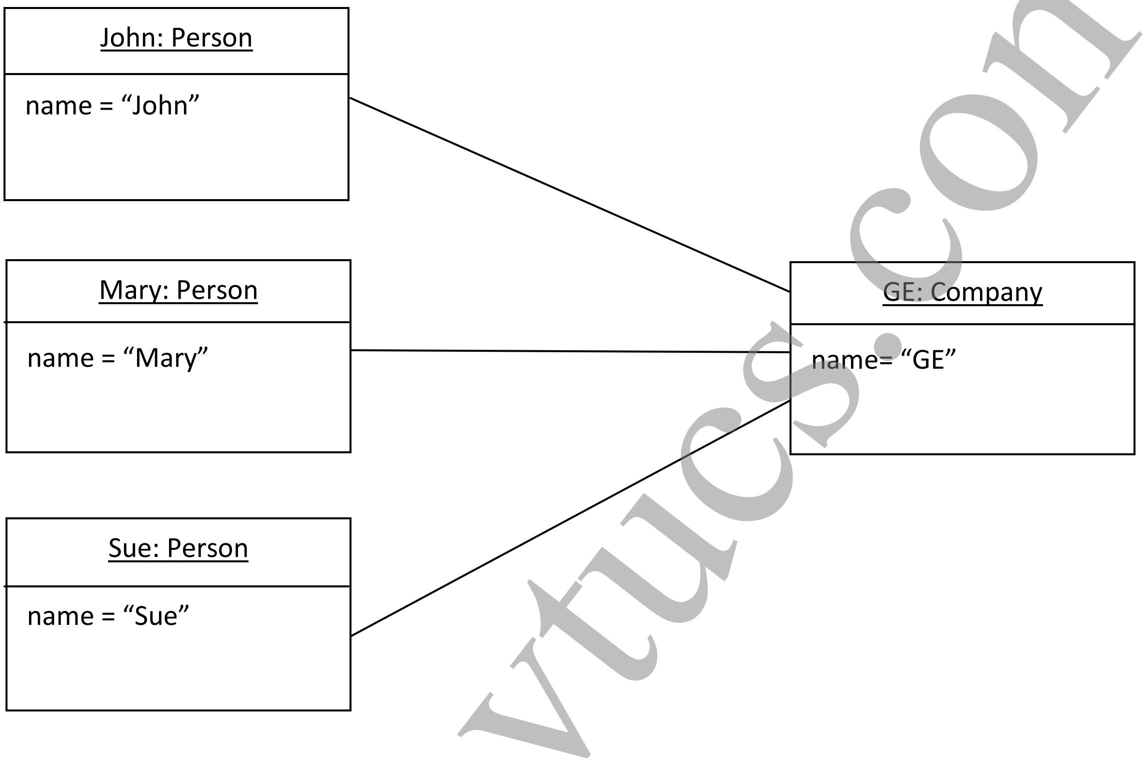
An association describes a set of potential links in the same way that a class describes a set of potential objects.

Example: Model for a financial application:

Stock brokerage firms need to perform tasks such as recording ownership of various stocks, tracking dividends, alerting customers to changes in the market and computing margin requirements.



Class Diagram



Object Diagram

<u>Many- to – many associations</u>

In the class diagram, a person may own stock in zero or more companies; a company may have multiple persons owning its stock.

The asterisk is a multiplicity specifies the number of instances of one class that may relate to a single class instance of another class. The UML notation for a link is a line between objects; a line may consist of several line segments. If a link has a name then it is underlined.

The association name is optional, if the model is unambiguous. Ambiguity arises when a model has multiple associations among the same classes.

Example: Person works for company and person owns stock in company.

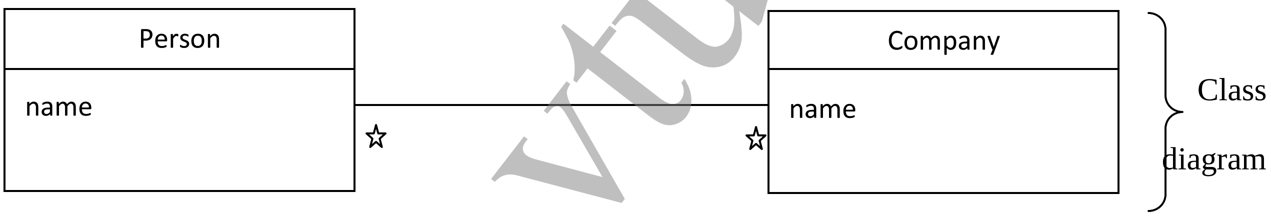
When there are multiple associations names are necessary. Associations are inherently bidirectional. The name of a binary association usually reads in a particular direction but the binary association can be traversed in either direction.

Example: WorksFor connects person to a company. The inverse of WorksFor could be Employs and it connects a company to a person. In reality, both directions of traversal are equally meaningful and refer to the same underlying association.

A reference is an attribute in one object that refers to another object.

Example: a data structure for person might contain an attribute employer that refers to a company object and a company object might contain an attribute employees that refers to a set of Person objects.

Multiplicity: specifies the number of instances of one class that may relate to a single instance of an associated class. UML diagrams explicitly list multiplicity at the ends of association lines. It specifies multiplicity with an interval , such as “/” (exactly one), “/…*” (one or more) or “3..5” (three to five inclusive). The special symbol “*” is a shorthand notation that denotes “many” (zero or more).

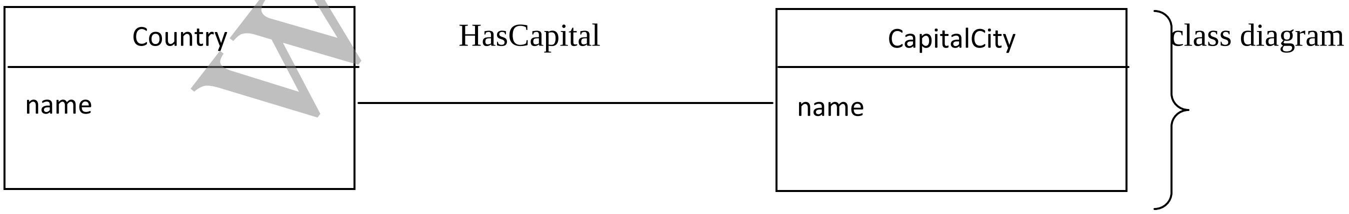


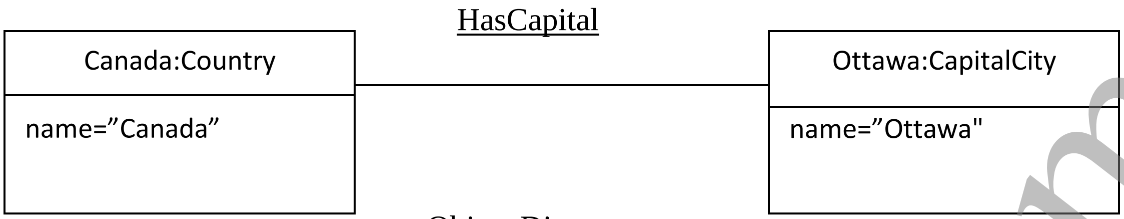
many – to –many multiplicity

A person may own stock in many companies. A company may have multiple persons holding its stock.

One – to- one association and some corresponding links:

Each country has one capital city. A capital city administers one country.



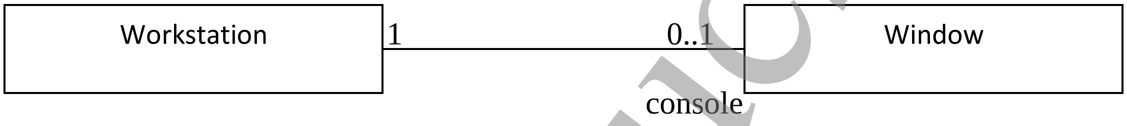


Object Diagram

Zero – or- one multiplicity:

A workstation may have one of its windows designated as the console to receive general error messages. It is possible however, that no console window exists.

The word console is an association end name.



Multiplicity is a constraint on the size of a collection. Cardinality is the count of elements that are actually in a collection.

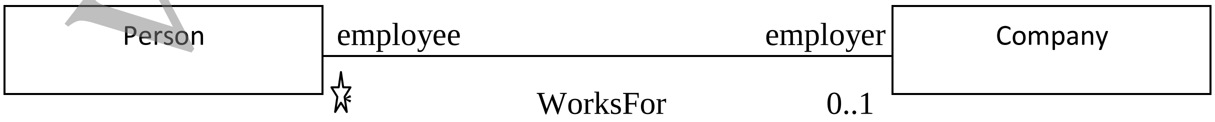
A multiplicity of “many” specifies that an object may be associated with multiple objects. But for each association there is at most one link between a given pair of objects. If you want 2 links between the same objects, you must have 2 associations.

Association End Names:

Multiplicity implicitly referred to the ends of associations.

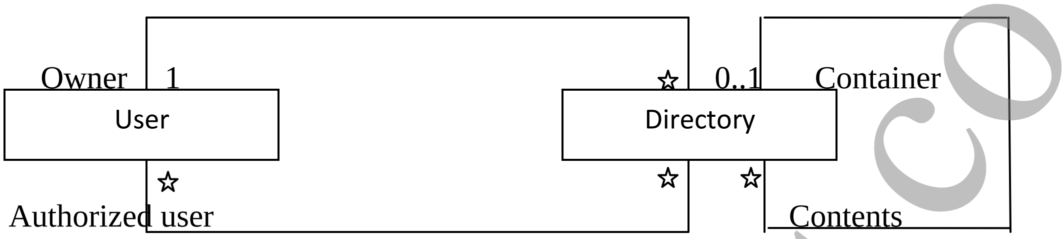
Example: one – to – many associations has 2 ends – an end with a multiplicity “one” and an end with a multiplicity “many”.

Association end names often appear as nouns in problem descriptions. A name appears next to the association end.



Person and company participate in association WorksFor. A person is an employee with respect to a company; a company is an employer with respect to a person. Use of association end names is optional.

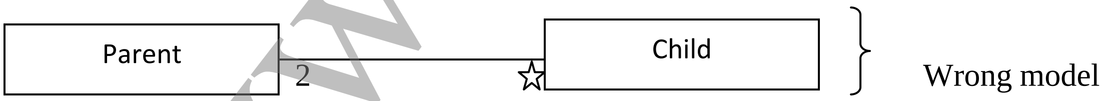
Association end names are necessary for associations between 2 objects of the same class.



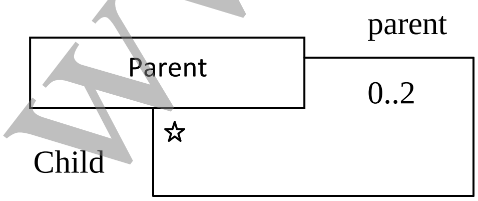
Container and contents distinguish the 2 usages of Directory in the self – association. A directory may contain many lesser directories and may optionally be contained itself. Association end names can also distinguish multiple associations between the same pair of classes.

Example: each directory has exactly one user who is an owner and many users who are authorized to use the directory.

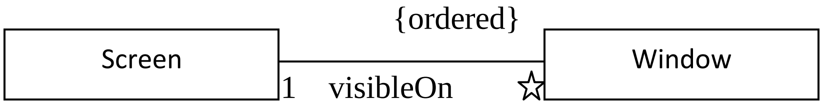
When constructing class diagrams you should properly use association end names and not introduce a separate class for each reference. Two instances represent a person with a child one for child and one for parent.



In the correct model, one person instance participates in2 or more links, twice as a parent and zero or more times as a child.

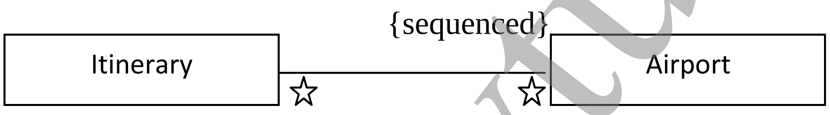


Ordering: Often the objects on a "many" association end have no explicit order and can regard them as a set. However, the objects have explicit order some times.



A workstation screen contains a number of overlapping windows. Each window on a screen occurs at most once. The windows have an explicit order so only the topmost window is visible. The ordering is an inherent part of the association. If objects indicate ordered set objects by writing "{ordered}" next to appropriate association end.

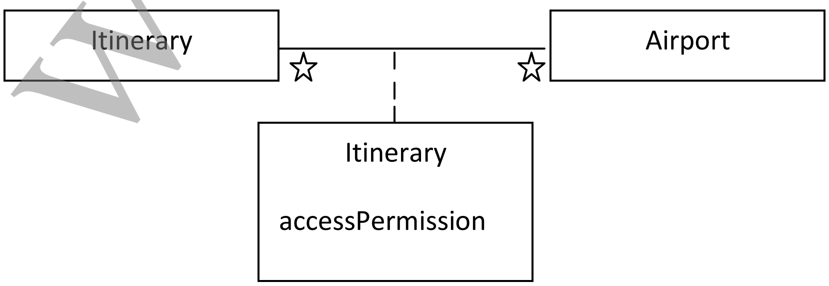
Bags and Sequences: A binary association has at most one link for a pair of objects. However, you can permit multiple links for a pair of objects by annotating an association end with {bag} or {sequence}. A bag is a collection of elements with duplicates allowed. A sequence is a ordered collection of elements with duplicates allowed.



A itinerary is a sequence of airports and the same airport can be visited more than once.

Association classes: As you describe the objects of a class with the attributes, we can describe the links of an association with attributes. The UML represents such information with an association class.

An association class is a association that is also a class. Like a class an association can have attributes and operations and participate in associations.



Many – to -many associations – attributes unmistakably belong to the link and cannot be ascribed to either object.

It is possible to fold attributes for one – to – one and one – to – many associations into the class opposite a “one” end. This is not possible for many – to – many associations.

As a rule, do not fold attributes of an association into a class.

Users may be authorized on many workstations. Each authorization carries a priority and access privileges. A user has a home directory for each authorized work station, but several workstations and users can share the same home directory.

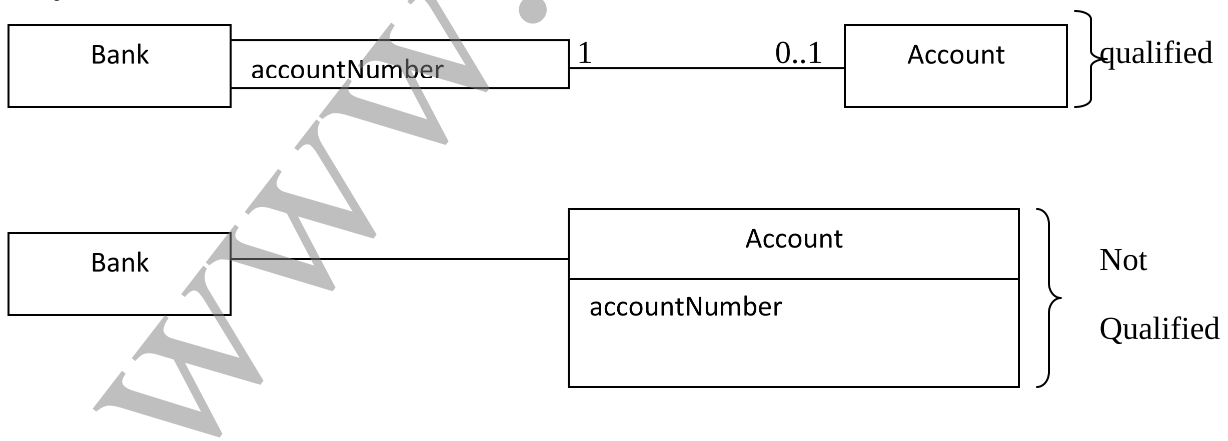
Association classes are an important aspect of class modeling because they let you specify identity and navigation paths.

The association class has only one occurrence for each pairing of person and company. In contrast there can be any number of occurrences of purchase for each person and company.

Qualified Associations: is an association in which an attribute called the qualifier disambiguates the objects for a “many” association end. It is possible to define qualifiers for one – to – many and many –to – many associations.

A qualifier selects among the target objects, reducing the effective multiplicity from”many” to “one”.

Example: a bank services multiple accounts. An account belongs to a single ban. Within the context of a bank, the account number specifies unique account. Bank and Account are classes and accountNumber is the qualifier. Qualification reduces the effective multiplicity from one to many to one to one.



Both models are acceptable, but the qualified model adds information. The qualified model adds multiplicity constraint, that the combination of a bank and an account number yields at most one account. The model conveys the significance of account number in traversing the model, as methods will reflect. You 1st specify the bank and then the account number to find the account.

The notation for qualifier→small box on the end of the association line near the source class. The box may grow out of any side (top, bottom, left, right).

The source class and qualifier yields target class.

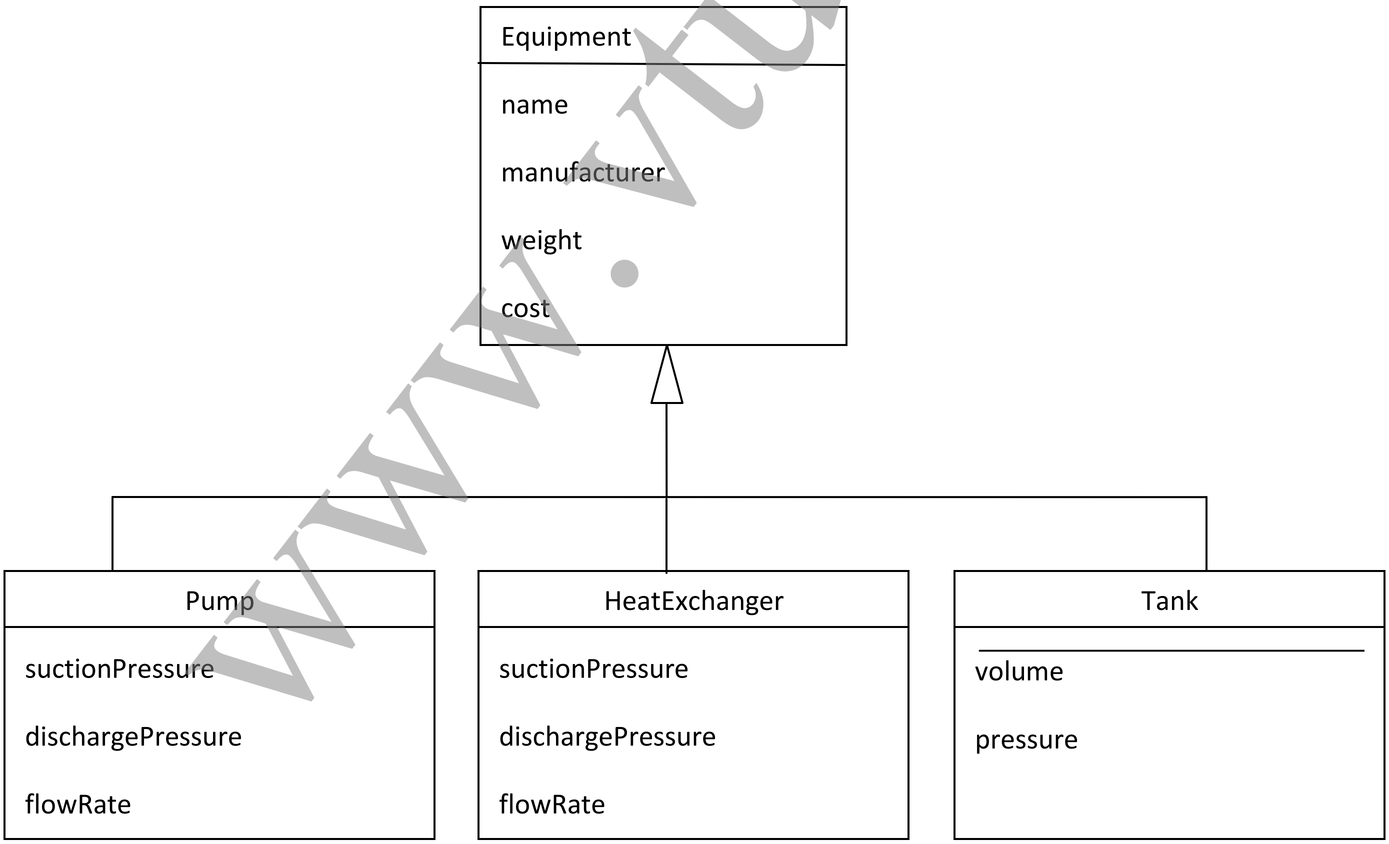
Generalization and Inheritance:

Generalization is the relationship between a class (super class) and one or more variations of the class (subclasses).

Generalization organizes classes by their similarities and different structuring the description of objects. The super class holds common attributes, operations and associations; the sub classes add specific attributes, operations and associations. Each sub class is said to inherits the features of its super class.

Generalization is sometimes called the"is –a" relationship, because each instance of a sub class is a instance of the super class as well.

A large hollow arrowhead denotes generalization. The arrowhead points to the super class. The curly braces denote a UML comment, indicating that there are additional subclasses that the diagram does not show.



The terms ancestor and descendent refer to generalization of classes across multiple levels. Use of Generalization: has 3 purposes, one of which is support for polymorphism. Polymorphism increases the flexibility of software you add a new sub class and automatically inherit super class behavior.

The second purpose of generalization is to structure the description of objects. When generalization is used, you are making a conceptual statement you are forming a taxonomy and organizing objects on the basis of their similarities and differences.

The third purpose is to enable reuse of code inherit code within the application as well as from part work (class library). The terms generalization, specialization and inheritance all refer to aspects of the same idea.

Overriding Features:

A sub class may override a super class feature by defining a feature with the same name. There are several reasons to override a feature: to specify behavior that depends on the sub class, to tighten the specification of a feature or to improve performance.

Navigation of Class Models:

Navigation is important because it lets you exercise a model and uncover hidden flaws and omissions so that you can repair them. You can perform navigation manually or write navigation expressions.

Consider the simple model for credit card accounts:

An institution may issue many credit card accounts, each identified by an account number. Each account has a maximum credit limit, a current balance and a mailing address. The account server one or more customers who reside at the mailing address. The institution periodically issues statement for each account. The statement lists a payment DueDate, financeCharge and minimum Payment. The statement itemizes various transactions that have occurred throughout the billing interval: cash advances, interest charges, purchases, fees and adjustments to the account. The name of the merchant is printed for each purchase.

We pose a variety of questions against the model.

*What transactions occurred for a credit card account within a time interval?

*What volume of transactions were handled by an institution in the last year?

*What customers patronized a merchant in the last year by any kind of credit card?

*How many credit card accounts does a customer currently have?

*What is the total maximum credit for a customer, for all accounts?

The UML incorporates a language that can express these kinds of questions –the object Constraint language (OCL).

OCL Constructs for traversing Class Models

Attributes: You can traverse from an object to an attribute value.

Syntax: source object, followed by a dot and then the attribute name.

Example: aCreditCardAccount. maximumCredit.

Operations: You can also invoke an operation for an object or a collection of objects.

Syntax: source object or object collection, followed by a dot and then the operation. The operation must be followed by parentheses, even if it has no arguments to avoid confusion with attributes.

The OCL has special operations that operate on entire collections. The syntax for a collection operation is the source object collection followed by "→" and then the operation.

Simple associations: A 3rd use of dot notation is to traverse an association to a target end.

Example: aCustomer.MailingAddress yields a set of addresses for a customer. In contrast, aCreditCardAccount.MailingAddress yields a single address.

Qualified associations: A qualifier lets you make a more precise traversal. The expression aCreditCardAccount.Statement[30 November 1999] finds the statement for a credit card account with the statement date 30 November 1999. The syntax is to enclose the qualifier value in brackets.

Generalizations: Traversal is implicit for the OCL notation.

Filters: OCL has several kinds of filters, most common of which is the select operation.

Example: aStatement.transaction → select(amount>$100) finds the transactions for a statement in excess of $100.

Unit 3

ADVANCED STATE MODELING

Nested State diagrams

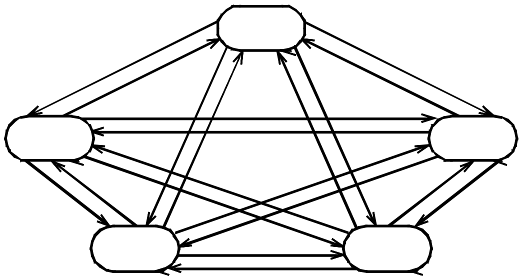
A **state diagram** is a type of diagram used in computer science and related fields to describe the behavior of systems. State diagrams require that the system described is composed of a finite number of states; sometimes, this is indeed the case, while at other times this is a reasonable abstraction. There are many forms of state diagrams, which differ slightly and have different semantics.

State diagrams are used to give an abstract description of the behavior of a system. This behavior is analyzed and represented in series of events that could occur in one or more possible states. Hereby "each diagram usually represents objects of a single class and track the different states of its objects through the system".

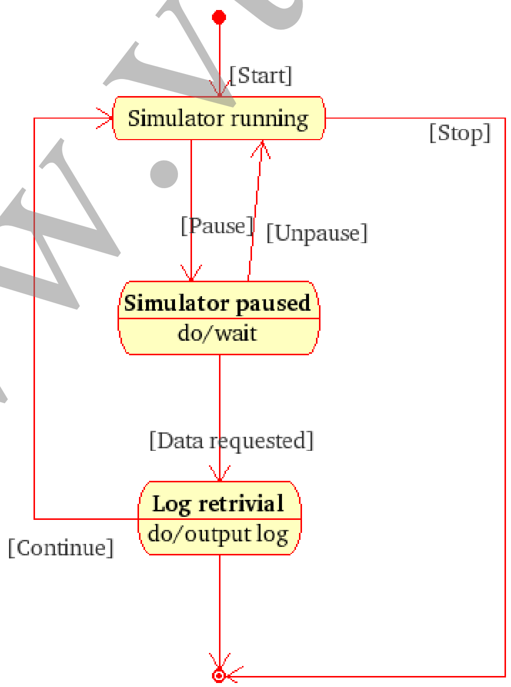
State diagrams can be used to graphically represent finite state machines. This was introduced by Taylor Booth in his 1967 book "Sequential Machines and Automata Theory". Another possible representation is the State transition table.

The UML state diagram is essentially a Harel state chart with standardized notation, which can describe many systems, from computer programs to business processes. The following are the basic notational elements that can be used to make up a diagram:

- Filled circle, pointing to the initial state
- Hollow circle containing a smaller filled circle, indicating the final state (if any)
- Rounded rectangle, denoting a state. Top of the rectangle contains a name of the state. Can contain a horizontal line in the middle, below which the activities that are done in that state are indicated



- Arrow, denoting transition. The name of the event (if any) causing this transition labels the arrow body. A guardexpression may be added before a "/" and enclosed in square-brackets (***eventName*****[guardExpression]**), denoting that this expression must be true for the transition to take place. If an action is performed during this transition, it is added to the label following a "/" (***eventName*****[guardExpression]/action**).
- Thick horizontal line with either x>1 lines entering and 1 line leaving or 1 line entering and x>1 lines leaving. These denote join/fork, respectively.

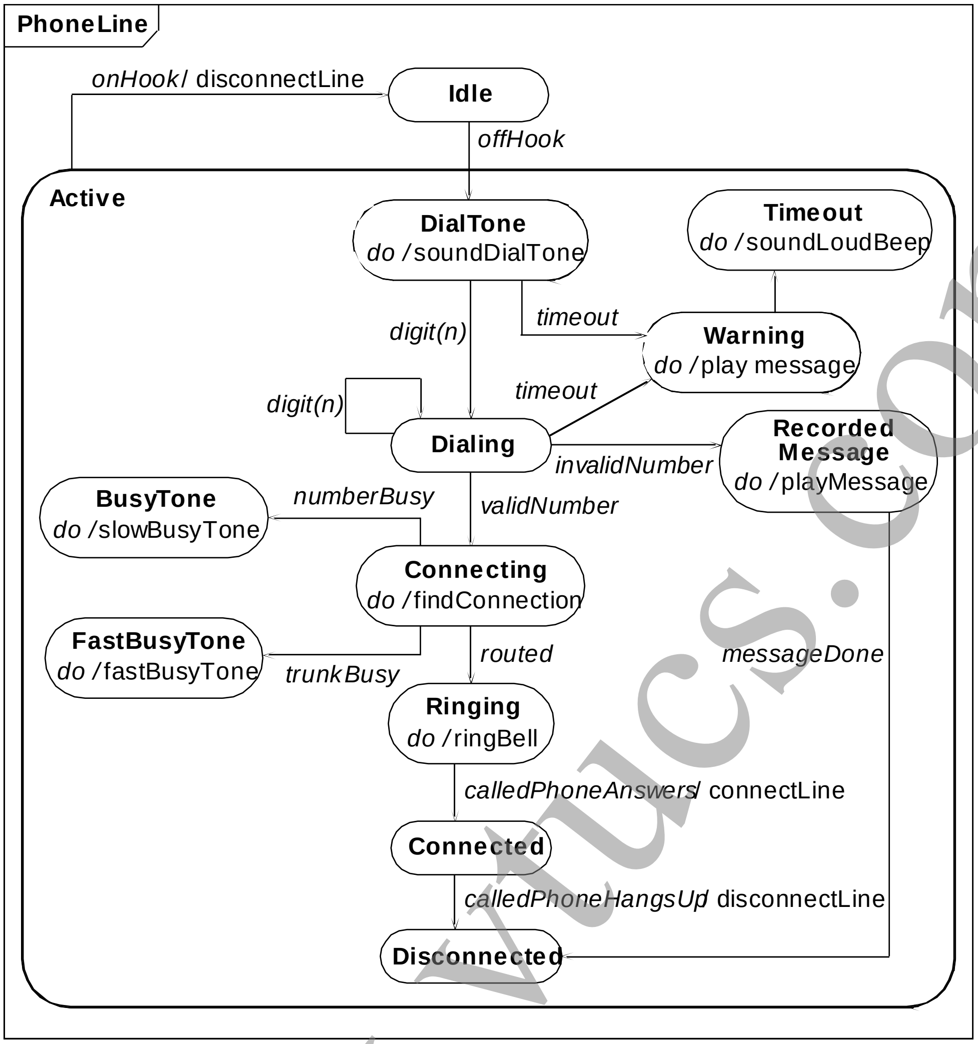


Hierarchically nested states

The most important innovation of UML state machines over the traditional FSMs is the introduction of hierarchically nested states (that is why state charts are also called hierarchical state machines, or HSMs). The semantics associated with state nesting are as follows (see Figure 3): If a system is in the nested state, for example "result" (called the substate), it also (implicitly) is in the surrounding state "on" (called the superstate). This state machine will attempt to handle any event in the context of the substate, which conceptually is at the lower level of the hierarchy. However, if the substate "result" does not prescribe how to handle the event, the event is not quietly discarded as in a traditional "flat" state machine; rather, it is automatically handled at the higher level context of the superstate "on". This is what is meant by the system being in state "result" as well as "on". Of course, state nesting is not limited to one level only, and the simple rule of event processing applies recursively to any level of nesting.

States that contain other states are called *composite states*; conversely, states without internal structure are called *simple states*. A nested state is called a *direct substate* when it is not contained by any other state; otherwise, it is referred to as a *transitively nested substate*.

Because the internal structure of a composite state can be arbitrarily complex, any hierarchical state machine can be viewed as an internal structure of some (higher-level) composite state. It is conceptually convenient to define one composite state as the ultimate root of state machine hierarchy. In the UML specification[1], every state machine has a **top state** (the abstract root of every state machine hierarchy), which contains all the other elements of the entire state machine. The graphical rendering of this all-enclosing top state is optional.



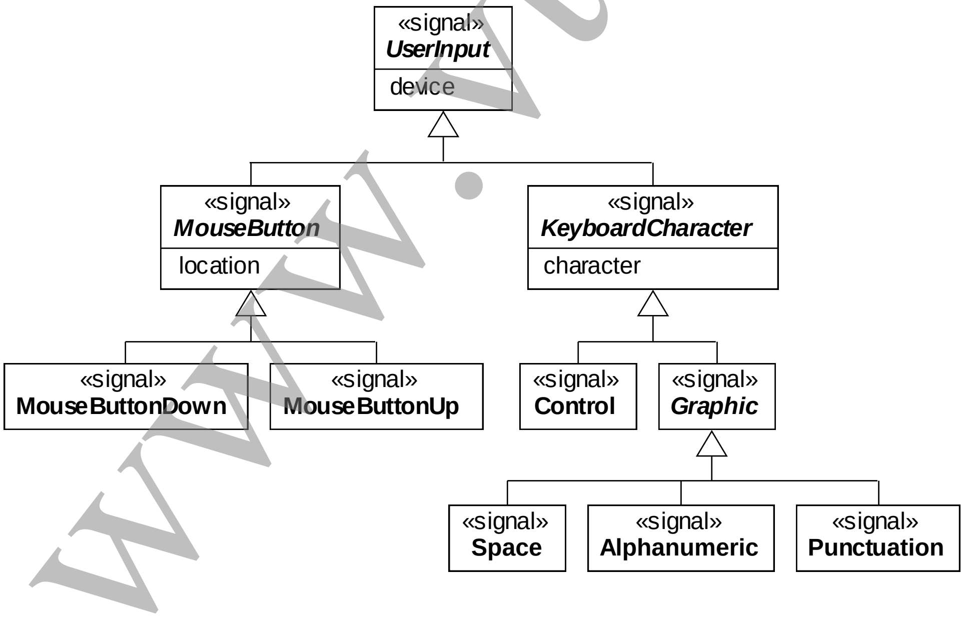
As you can see, the semantics of hierarchical state decomposition are designed to facilitate reusing of behavior. The substates (nested states) need only define the differences from the superstates (surrounding states). A substate can easily inherit[4] the common behavior from its superstate(s) by simply ignoring commonly handled events, which are then automatically handled by higher-level states. In other words, hierarchical state nesting enables programming by difference.

The aspect of state hierarchy emphasized most often is abstraction—an old and powerful technique for coping with complexity. Instead of facing all aspects of a complex system at the same time, it is often possible to ignore (abstract away) some parts of the system. Hierarchical states are an ideal mechanism for hiding internal details because the designer can easily zoom out or zoom in to hide or show nested states.

However, the composite states don't simply hide complexity; they also actively reduce it through the powerful mechanism of hierarchical event processing. Without such reuse, even a moderate increase in system complexity often leads to an explosive increase in the number of states and transitions. For example, the hierarchical state machine representing the pocket calculator (Figure 3) avoids repeating the transitions Clear and Off in virtually every state. Avoiding repetitions allows HSMs to grow proportionally to system complexity. As the modeled system grows, the opportunity for reuse also increases and thus counteracts the explosive increase in states and transitions typical for traditional FSMs.

Signal Generalization

You can organize signals into a generalization hierarchy with inheritance of signal attributes. The below figure shows part of onput signals for a workstation.



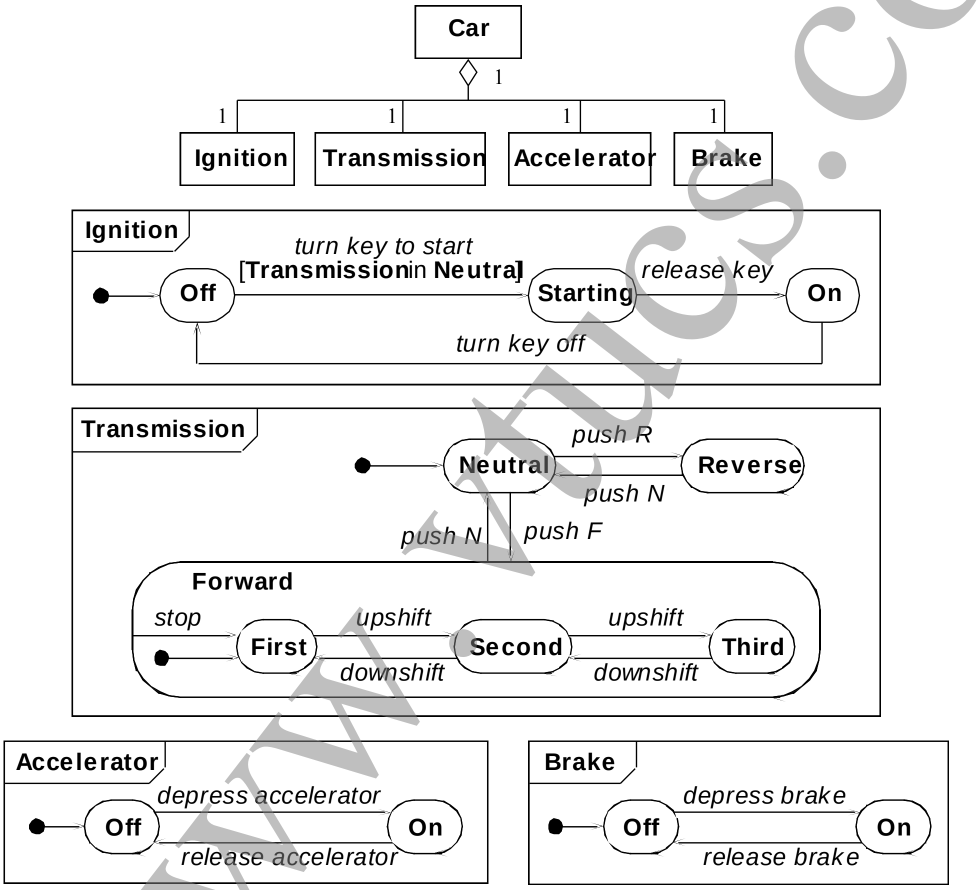
The hierarchy permits different levels of abstraction to be used in a model.

Concurrency

The state model implicitly supports concurrency among objects. But objects need not be completely independent and may be subject to shared constraints that cause some correspondence among their state changes.

1. Aggregation concurrency

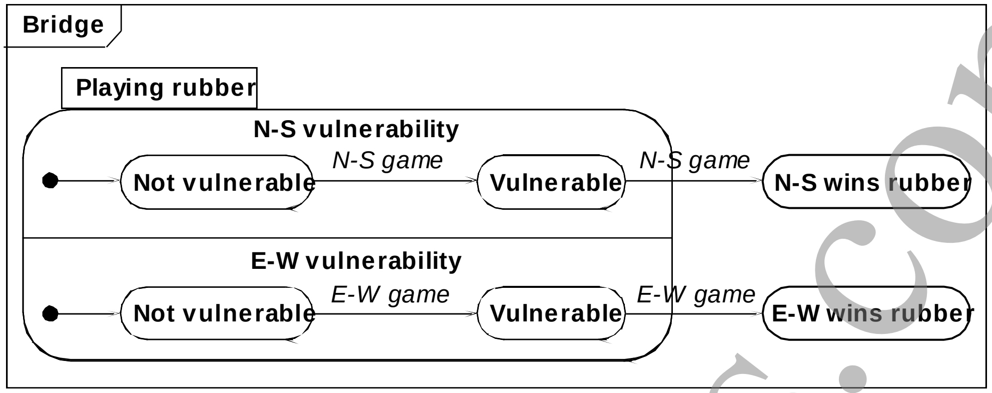
A state diagram for an assembly is a collection of state diagrams, one for each part. Aggregation is the and relationship. The below figure shows the state of a *car* as an aggregation of several parts.



2. Concurrency within an object

You can patition some objects into subsets of attributes or links, each of which has its own sub diagram. The UML shows concurrency within an object by partitioning the composite state in a separate tab so that it does not become confused with the concurrent region.

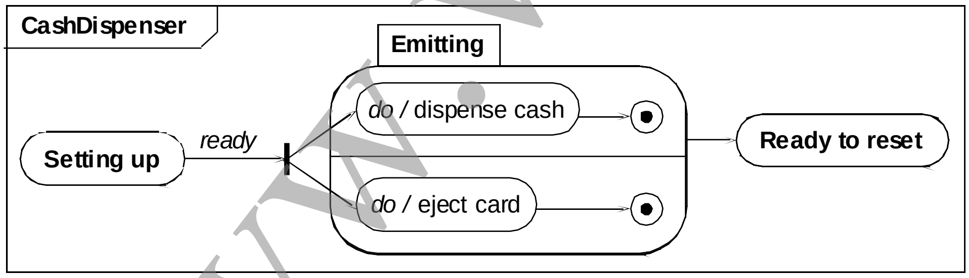
The below figure shows state diagram for the play of a bridge rubber. When a side wins a game, it becomes”vulnerable”; the first side to win two games wins the rubber.



3. synchronoisation of concurrent activities

Sometimes one object must perform two or more activities concurrently. The object does not synchronize the initial steps of the activities but most complete both activities before it can progress to next state.

The below figure shows a concurrent state diagram for the emitting activity.



Relation of class and state model

A **class diagram** in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes.

A relationship is a general term covering the specific types of logical connections found on class and object diagrams

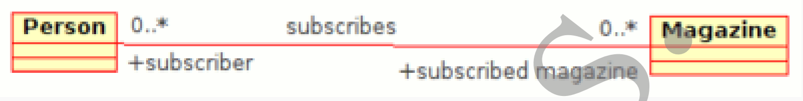
UML shows the following relationships:

Instance Level Relationships

External links

A *Link* is the basic relationship among objects. It is represented as a line connecting two or more object boxes. It can be shown on an object diagram or class diagram. A link is an instance of an association. In other words, it creates a relationship between two classes.

Association

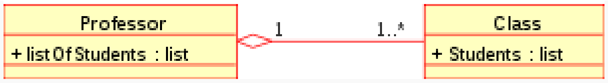


Class diagram example of association between two classes

An Association represents a family of links. Binary associations (with two ends) are normally represented as a line, with each end connected to a class box. Higher order associations can be drawn with more than two ends. In such cases, the ends are connected to a central diamond.

An association can be named, and the ends of an association can be adorned with role names, ownership indicators, multiplicity, visibility, and other properties. There are five different types of association. Bi-directional and uni-directional associations are the most common ones. For instance, a flight class is associated with a plane class bi-directionally. Associations can only be shown on class diagrams. Association represents the static relationship shared among the objects of two classes. Example: "department offers courses", is an association relation.

Aggregation



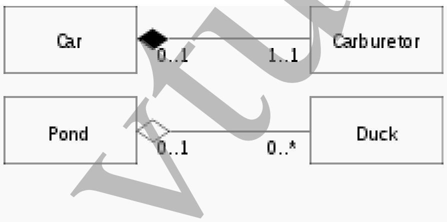
Class diagram showing Aggregation between two classes

Aggregation is a variant of the "has a" or association relationship; aggregation is more specific than association. It is an association that represents a part-whole or part-of relationship. As a type of association, an aggregation can be named and have the same adornments that an association can. However, an aggregation may not involve more than two classes.

Aggregation can occur when a class is a collection or container of other classes, but where the contained classes do not have a strong *life cycle dependency* on the container—essentially, if the container is destroyed, its contents are not.

In UML, it is graphically represented as a *hollow* diamond shape on the containing class end of the tree of lines that connect contained class(es) to the containing class.

Composition



Class diagram showing Composition between two classes at top and Aggregation between two classes at bottom

Composition is a stronger variant of the "owns a" or association relationship; composition is more specific than aggregation. It is represented with a solid diamond shape.

Composition usually has a strong *life cycle dependency*between instances of the container class and instances of the contained class(es): If the container is destroyed, normally every instance that it contains is destroyed as well. Note that a part can (where allowed) be removed from a composite before the composite is deleted, and thus not be deleted as part of the composite.

The UML graphical representation of a composition relationship is a *filled* diamond shape on the containing class end of the tree of lines that connect contained class(es) to the containing class.

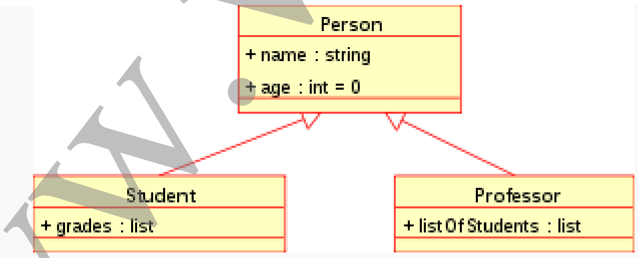
Differences between Composition and Aggregation

When attempting to represent real-world whole-part relationships, e.g., an engine is part of a car, the composition relationship is most appropriate. However, when representing a software or database relationship, e.g., car model engine ENG01 is part of a car model CM01, an aggregation relationship is best, as the engine, ENG01 may be also part of a different car model. Thus the aggregation relationship is often called "catlog" containment to distinguish it from composition's "physical" containment.

The whole of a composition must have a multiplicity of 0..1 or 1, indicating that a part must belong to only one whole; the part may have any multiplicity. For example, consider University and Department classes. A department belongs to only one university, so University has multiplicity 1 in the relationship. A university can (and will likely) have multiple departments, so Department has multiplicity 1..*.

Class Level Relationships

Generalization



Class diagram showing generalization between one superclass and two subclasses

The Generalization relationship indicates that one of the two related classes (the *subtype*) is considered to be a specialized form of the other (the *super type*) and supertype is considered as '***Generalization'*** of subtype. In practice, this means that any instance of the subtype is also an instance of the supertype. An exemplary tree of generalizations of this form is found in binomial nomenclature:human beings are a subtype of simian, which are a subtype of mammal, and so on. The relationship is most easily understood by the phrase 'A is a B' (a human is a mammal, a **mammal** is an animal).

The UML graphical representation of a Generalization is a hollow triangle shape on the supertype end of the line (or tree of lines) that connects it to one or more subtypes.

The generalization relationship is also known as the *inheritance* or *"is a"* relationship.

The *supertype* in the generalization relationship is also known as the *"parent"*, *superclass*, *base class*, or *base type*.

The *subtype* in the specialization relationship is also known as the *"child"*, *subclass*, *derived class*,*derived type*, *inheriting class*, or *inheriting type*.

Note that this relationship bears no resemblance to the biological parent/child relationship: the use of these terms is extremely common, but can be misleading.

- Generalization-Specialization relationship

  A is a type of B

  E. g. "an oak is a type of tree", "an automobile is a type of vehicle"

Generalization can only be shown on class diagrams and on Use case diagrams.

Interaction Modeling

The interaction model describes how objects interact to produce useful results. Interactions can be modeled at different level of abstraction. At higher level use cases describe how a s/m interacts with outside actors. The next level, sequence diagrams are used. Finally, activity diagrams are used.

Use case

Use case in software engineering and systems engineering is a description of a system's behavior as it responds to a request that originates from outside of that system. In other words, a use case describes "who" can do "what" with the system in question. The use case technique is used to capture a system's behavioral requirements by detailing scenario-driven threads through the functional requirements.

"Each use case focuses on describing how to achieve a goal or a task. For most software projects, this means that multiple, perhaps dozens of use cases are needed to define the scope of the new system. The degree of formality of a particular software project and the stage of the project will influence the level of detail required in each use case." [*cite this quote*]

Use cases should not be confused with the features of the system under consideration. A use case may be related to one or more features, and a feature may be related to one or more use cases.

A use case defines the interactions between external actors and the system under consideration to accomplish a goal. An actor specifies a role played by a person or thing when interacting with the system.The same person using the system may be represented as different actors because they are playing different roles. For example, "Joe" could be playing the role of a Customer when using an Automated Teller Machine to withdraw cash, or playing the role of a Bank Teller when using the system to restock the cash drawer.

Use cases treat the system as a black box, and the interactions with the system, including system responses, are perceived as from outside the system. This is a deliberate policy, because it forces the author to focus on what the system must do, not how it is to be done, and avoids the trap of making assumptions about how the functionality will be accomplished.

Use cases may be described at the abstract level (business use case, sometimes called essential use case), or at the system level (system use case). The differences between these are the scope.

- A **business use case** is described in technology-free terminology which treats system as a black box and describes the business process that is used by its business actors (people or systems external to the business) to achieve their goals (e.g., manual payment processing, expense report approval, manage corporate real estate). The business use case will describe a process that provides value to the business actor, and it describes *what* the process does. Business Process Mapping is another method for this level of business description.

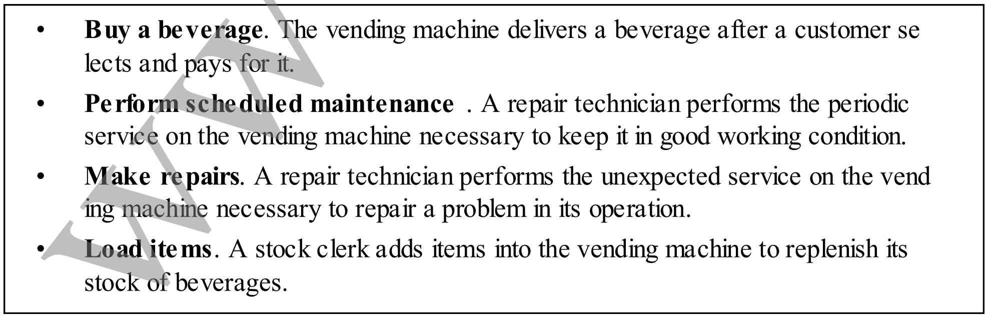
- A **system use case** is normally described at the system functionality level (for example, create voucher) and specifies the function or the service that the system provides for the user. The system use case will describe *what* the actor achieves interacting with the system. For this reason it is recommended that system use case specification begin with a verb (e.g., *create* voucher, *select*payments, *exclude* payment, *cancel* voucher). Generally, the actor could be a human user or another system interacting with the system being defined.

A use case should:

- Describe what the system shall do for the actor to achieve a particular goal.
- Include no implementation-specific language.
- Be at the appropriate level of detail.
- Not include detail regarding user interfaces and screens. This is done in user-interface design.

Use case notation

In Unified Modeling Language, the relationships between all (or a set of) the use cases and actors are represented in a use case diagram or diagrams, originally based upon Ivar Jacobson's Objectorynotation. SysML, a UML profile, uses the same notation at the system block level. Use case summaries for a vending machine.



- **Buy a beverage**. The vending machine delivers a beverage after a customer selects and pays for it.
- **Perform scheduled maintenance** . A repair technician performs the periodic service on the vending machine necessary to keep it in good working condition.
- **Make repairs**. A repair technician performs the unexpected service on the vending machine necessary to repair a problem in its operation.
- **Load items**. A stock clerk adds items into the vending machine to replenish its stock of beverages.

Use case description can be given as below.

Use Case: Buy a beverage

Summary: The vending machine delivers a beverage after a customer selects and pays for it.

Actors: Customer

Preconditions: The machine is waiting for money to be inserted.

Description: The machine starts in the waiting state in which it displays the message "Enter coins." A customer inserts coins into the machine. The machine displays the total value of money entered and lights up the buttons for the items that can be purchased for the money inserted. The customer pushes a button. The machine dispenses the corresponding item and makes change, if the cost of the item is less than the money inserted.

Exceptions:

Canceled: If the customer presses the cancel button before an item has been selected, the customer 's money is returned and the machine resets to the waiting state.

Out of stock: If the customer presses a button for an out-of-stock item, the message "That item is out of stock" is displayed. The machine continues to accept coins or a selection.

Insufficient money: If the customer presses a button for an item that costs more than the money inserted, the message "You must insert $ *nn.nn* more for that item" is displayed, where *nn.nn* is the amount of additional money needed. The machine continues to accept coins or a selection.

No change: If the customer has inserted enough money to buy the item but the machine cannot make the correct change, the message "Cannot make correct change" is displayed and the machine continues to accept coins or a selection.

Postconditions: The machine is waiting for money to be inserted.

Use cases and the development process

The specific way use cases are used within the development process will depend on which development methodology is being used. In certain development methodologies, a brief use case survey is all that is required. In other development methodologies, use cases evolve in complexity and change in character as the development process proceeds. In some methodologies, they may begin as brief business use cases, evolve into more detailed system use cases, and then eventually develop into highly detailed and exhaustive test cases.

Use case templates

There is no standard template for documenting detailed use cases. A number of competing schemes exist, and individuals are encouraged to use templates that work for them or the project they are on. Standardization within each project is more important than the detail of a specific template. There is, however, considerable agreement about the core sections; beneath differing terminologies and orderings there is an underlying similarity between most use cases. Different templates often have additional sections, e.g., assumptions, exceptions, recommendations, technical requirements. There may also be industry specific sections.

Use case name

A use case name provides a unique identifier for the use case. It should be written in verb-noun format (e.g., *Borrow Books*, *Withdraw Cash*), should describe an achievable goal (e.g., *Register User* is better than *Registering User*) and should be sufficient for the end user to understand what the use case is about.

Goal-driven use case analysis will name use cases according to the actor's goals, thus ensuring use cases are strongly user centric. Two to three words is the optimum. If more than four words are proposed for a name, there is usually a shorter and more specific name that could be used.

Version

Often a version section is needed to inform the reader of the stage a use case has reached. The initial use case developed for business analysis and scoping may well be very different from the evolved version of that use case when the software is being developed. Older versions of the use case may still be in current documents, because they may be valuable to different user groups.

Goal

Without a goal a use case is useless. There is no need for a use case when there is no need for any actor to achieve a goal. A goal briefly describes what the user intends to achieve with this use case.

Summary

A summary section is used to capture the essence of a use case before the main body is complete. It provides a quick overview, which is intended to save the reader from having to read the full contents of a use case to understand what the use case is about. Ideally, a summary is just a few sentences or a paragraph in length and includes the goal and principal actor.

Actors

An actor is someone or something outside the system that either acts on the system – a primary actor – or is acted on by the system – a secondary actor. An actor may be a person, a device, another system or sub-system, or time. Actors represent the different roles that something outside has in its relationship with the system whose functional requirements are being specified. An individual in the real world can be represented by several actors if they have several different roles and goals in regards to a system. These interact with system and do some action on that.

Stakeholders

A stakeholder is an individual or department that is affected by the outcome of the use case.Individuals are usually agents of the organization or department for which the use case is being created. A stakeholder might be called on to provide input, feedback, or authorization for the use case.The stakeholder section of the use case can include a brief description of which of these functions the stakeholder is assigned to fulfill.

Preconditions

A *preconditions* section defines all the conditions that must be true (i.e., describes the state of the system) for the *trigger* (see below) to meaningfully cause the initiation of the use case. That is, if the system is not in the state described in the preconditions, the behavior of the use case is indeterminate. Note that the preconditions are *not* the same thing as the "trigger" (see below): the mere fact that the preconditions are met does NOT initiate the use case.

However, it is theoretically possible *both* that a use case should be initiated whenever condition X is met *and* that condition X is the only aspect of the system that defines whether the use case can meaningfully start. If this is really true, then condition X is *both* the precondition and the trigger, and would appear in both sections. But this is *rare*, and the analyst should check carefully that they have not overlooked some preconditions which are part of the trigger. If the analyst has erred, the module based on this use case will be triggered when the system is in a state the developer has not planned for, and the module may fail or behave unpredictably.

Triggers

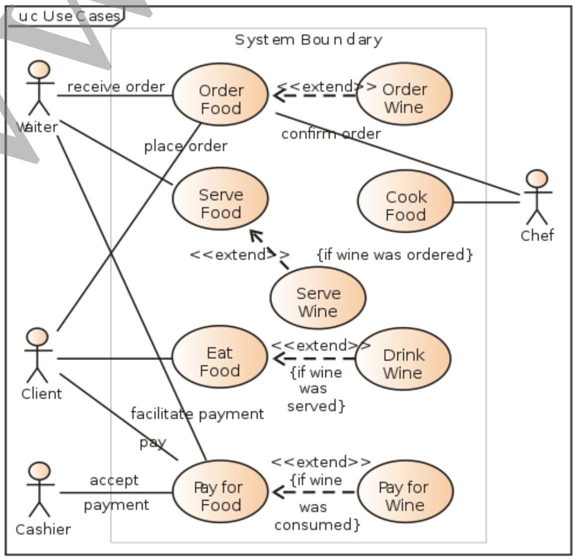
A 'triggers' section describes the event that causes the use case to be initiated. This event can be external, temporal or internal. If the trigger is not a simple true "event" (e.g., the customer presses a button), but instead "when a set of conditions are met", there will need to be a triggering process that continually (or periodically) runs to test whether the "trigger conditions" are met: the "triggering event" is a signal from the trigger process that the conditions are now met.

There is varying practice over how to describe what to do when the trigger occurs but the preconditions are not met.

- One way is to handle the "error" within the use case (as an exception). Strictly, this is illogical, because the "preconditions" are now not true preconditions at all (because the behavior of the use case is determined even when the preconditions are not met).
- Another way is to put all the preconditions in the trigger (so that the use case does not run if the preconditions are not met) and create a different use case to handle the problem. Note that if this is the local standard, then the use case template theoretically does not need a preconditions section!

A **use case diagram** in the Unified Modeling Language (UML) is a type of behavioral diagram defined by and created from a Use-case analysis. Its purpose is to present a graphical overview of the functionality provided by a system in terms of actors, their goals (represented as use cases), and any dependencies between those use cases.

The main purpose of a use case diagram is to show what system functions are performed for which actor. Roles of the actors in the system can be depicted.



Sequence Models

A **sequence diagram** in Unified Modeling Language(UML) is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart.

Sequence diagrams are sometimes called event diagrams, event scenarios, and timing diagrams.

UML sequence diagrams model the flow of logic within your system in a visual manner, enabling you both to document and validate your logic, and are commonly used for both analysis and design purposes. Sequence diagrams are the most popular UML artifact for dynamic modeling, which focuses on identifying the behavior within your system. Other dynamic modeling techniques include activity diagramming, communication diagramming, timing diagramming, and interaction overview diagramming. Sequence diagrams, along with class diagrams and physical data models are in my opinion the most important design-level models for modern business application development.

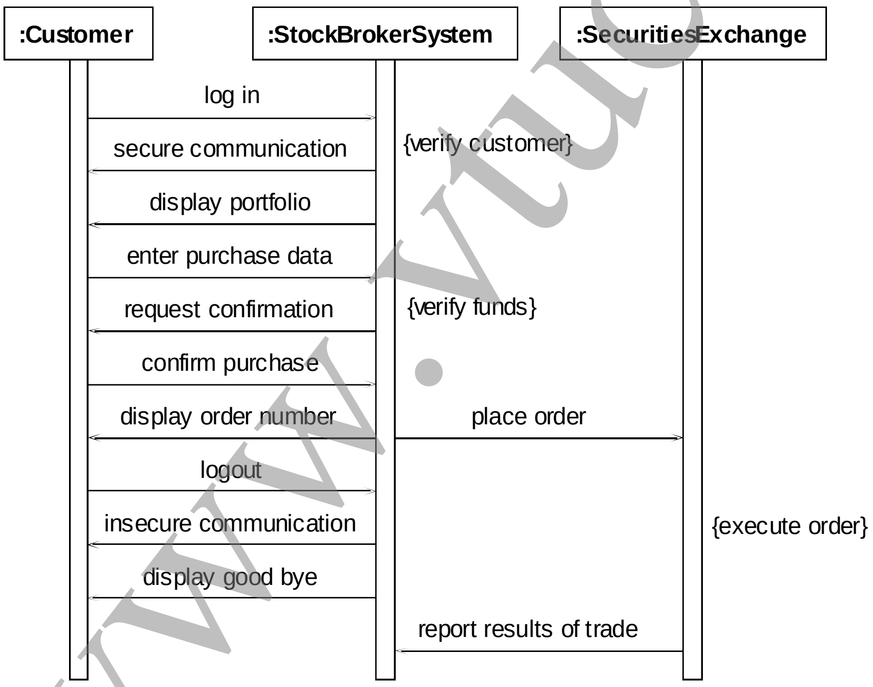
Sequence diagrams are typically used to model:

1. **Usage scenarios**. A usage scenario is a description of a potential way your system is used. The logic of a usage scenario may be part of a use case, perhaps an alternate course. It may also be one entire pass through a use case, such as the logic described by the basic course of action or a portion of the basic course of action, plus one or more alternate scenarios. The logic of a usage scenario may also be a pass through the logic contained in several use cases. For example, a student enrolls in the university, and then immediately enrolls in three seminars.
2. **The logic of methods**. Sequence diagrams can be used to explore the logic of a complex operation, function, or procedure. One way to think of sequence diagrams, particularly highly detailed diagrams, is as **visual object code**.
3. **The logic of services**. A service is effectively a high-level method, often one that can be invoked by a wide variety of clients. This includes web-services as well as business

transactions implemented by a variety of technologies such as CICS/COBOL or CORBA-compliant object request

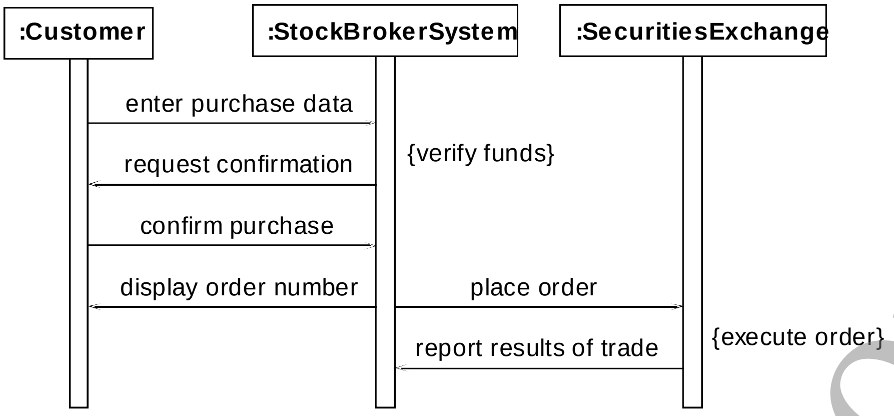
Let's start with three simple examples. Figure 1 depicts a UML sequence diagram for the Enroll in University use case, taking a system-level approach where the interactions between the actors and the system are show. Figure 2 depicts a sequence diagram for the detailed logic of a service to determine if an applicant is already a student at the university**.** Figure 3 shows the logic for how to enroll in a seminar. I will often develop a system-level sequence diagram with my stakeholders to help to both visualize and validate the logic of a usage scenario. It also helps me to identify significant methods/services, such as checking to see if the applicant already exists as a student, which my system must support.

Figure 1. sequence diagram for a session with an online stock broker.



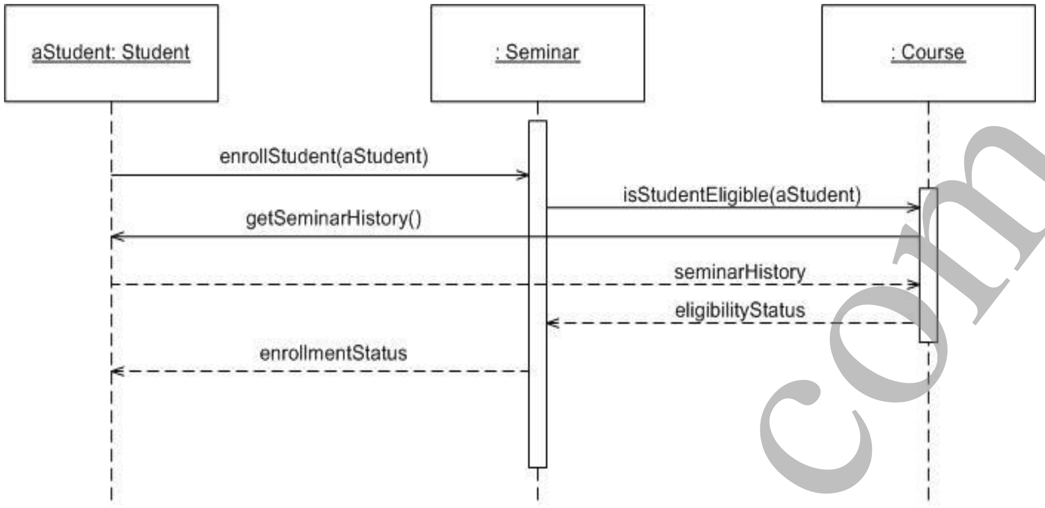
The reason why they're called sequence diagrams should be obvious: the sequential nature of the logic is shown via the ordering of the messages (the horizontal arrows). The first message starts in the top left corner, the next message appears just below that one, and so on.

Figure 2. sequence diagram for stock purchase.



The boxes across the top of the diagram represent classifiers or their instances, typically use cases, objects, classes, or actors. Because you can send messages to both objects and classes, objects respond to messages through the invocation of an operation and classes do so through the invocation of static operations, it makes sense to include both on sequence diagrams. Because actors initiate and take an active part in usage scenarios, they can also be included in sequence diagrams. Objects have labels in the standard UML format *name: ClassName*, where "name" is optional (objects that haven't been given a name on the diagram are called anonymous objects). Classes have labels in the format *ClassName*, and actors have names in the format *Actor Name*. Notice how object labels are underlined, classes and actors are not. For example, in Figure 3, you see the *Student* object has the name *aStudent*, this is called a named object, whereas the instance of*Seminar* is an anonymous object. The instance of *Student* was given a name because it is used in several places as a parameter in messages, whereas the instance of the *Seminar* didn't need to be referenced anywhere else in the diagram and thus could be anonymous. In Figure 2 the *Student* class sends messages to the*PersistenceFramework* class (which could have been given the stereotype <<infrastructure>> but wasn't to keep the diagram simple). Any message sent to a class is implemented as a static method, more on this later.

Figure 3. Enrolling in a seminar (method).

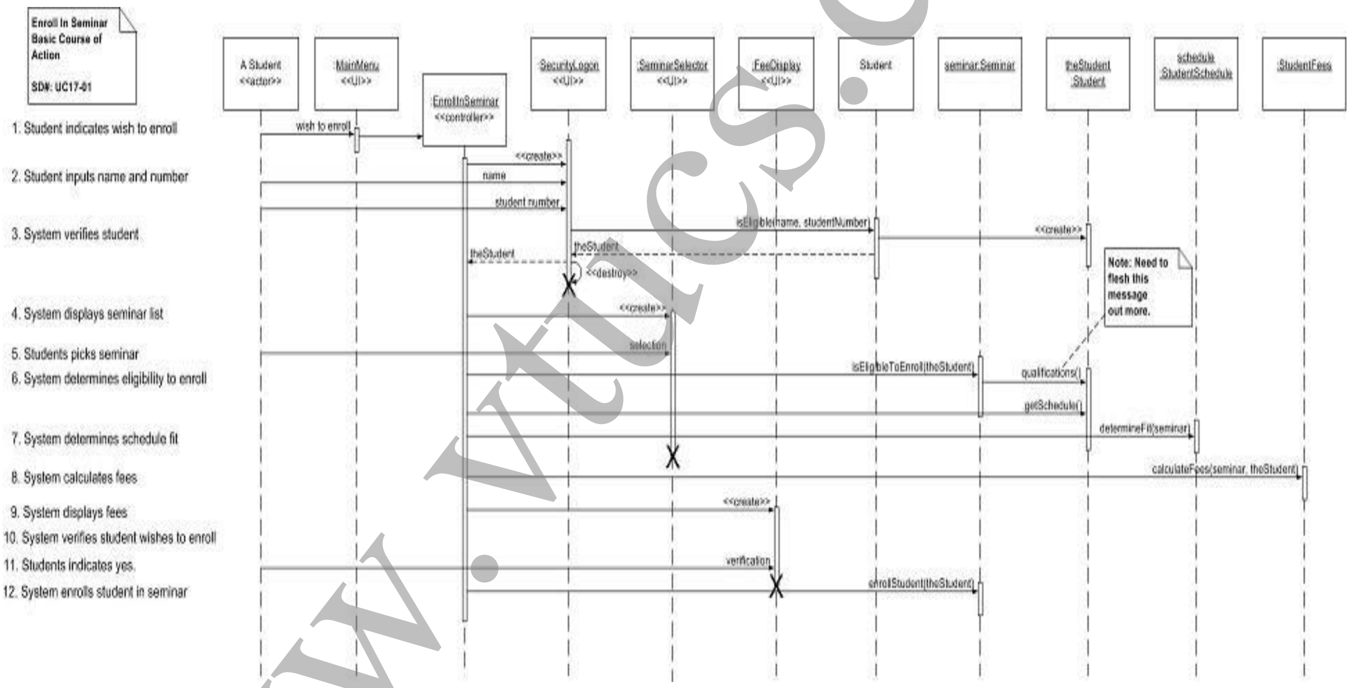


The dashed lines hanging from the boxes are called object lifelines, representing the life span of the object during the scenario being modeled. The long, thin boxes on the lifelines are activation boxes, also called method-invocation boxes, which indicate processing is being performed by the target object/class to fulfill a message. I will only draw activation boxes when I'm using a tool that natively supports them, such as a sophisticated CASE tool, and when I want to explore performance issues. Activation boxes are too awkward to draw on whiteboards or with simple drawing tools such that don't easily support them.

The *X* at the bottom of an activation box, an example of which is presented in Figure 4, is a UML convention to indicate an object has been removed from memory. In languages such as C++ where you need to manage memory yourself you need to invoke an object's destructor, typically modeled a message with the stereotype of <<destroy>>. In languages such as Java or C# where memory is managed for you and objects that are no longer needed are automatically removed from memory, something often referred to as garbage collection, you do not need to model the message. I generally don't bother with modeling object destruction at all and will instead trust that the programmers, often myself, will implement low-level details such as this appropriately.

Figure 4 presents a complex UML sequence diagram for the basic course of action for the *Enroll in Seminar* use case. This is an alternative way for modeling the logic of a usage scenario, instead of doing it at the system-level such as Figure 1 you simply dive straight into modeling the detailed logic at the object-level. I'll take this approach when I'm working with developers who are experienced sequence diagrammers and I have a large working space (either a huge whiteboard or a CASE tool installed on a workstation with a very large screen and good graphic card). Most of the time I'll draw system-level diagrams first and then create small diagrams also the lines of what is shown in Figures **2** and **3**.

Figure 4. Basic course of action for the Enroll in Seminar use case.


Enroll In Seminar
Basic Course of
Action
SD#: UC17-01
A Student
<<actor>>
:MainMenu
<<UI>>
:EnrollInSeminar
<<controller>>
:SecurityLogon
<<UI>>
:SeminarSelector
<<UI>>
:FeeDisplay
<<UI>>
Student
seminar:Seminar
theStudent
:Student
schedule
:StudentSchedule
:StudentFees
1. Student indicates wish to enroll
2. Student inputs name and number
3. System verifies student
4. System displays seminar list
5. Students picks seminar
6. System determines eligibility to enroll
7. System determines schedule fit
8. System calculates fees
9. System displays fees
10. System verifies student wishes to enroll
11. Students indicates yes.
12. System enrolls student in seminar
wish to enroll
<<create>>
name
student number
isEligible(name, studentNumber)
<<create>>
theStudent
theStudent
<<destroy>>
<<create>>
selection
Note: Need to flesh this message out more.
isEligibleToEnroll(theStudent)
qualifications()
getSchedule()
determineFit(seminar)
calculateFees(seminar, theStudent)
<<create>>
verification
enrollStudent(theStudent)

Messages are indicated on UML sequence diagrams as labeled arrows, when the source and target of a message is an object or class the label is the signature of the method invoked in response to the message. However, if either the source or target is a human actor, then the message is labeled with brief text describing the information being communicated. For example, in Figure 4 the *EnrollInSeminar* object sends the message *isEligibleToEnroll(theStudent)* to the instance of *Seminar*. Notice how I include both the method's name and the name of the parameters, if any, passed into it. The *Student* actor provides information to the *SecurityLogon* object via the messages labeled *name* and *student number* (these really aren't messages, they are actually user interactions).

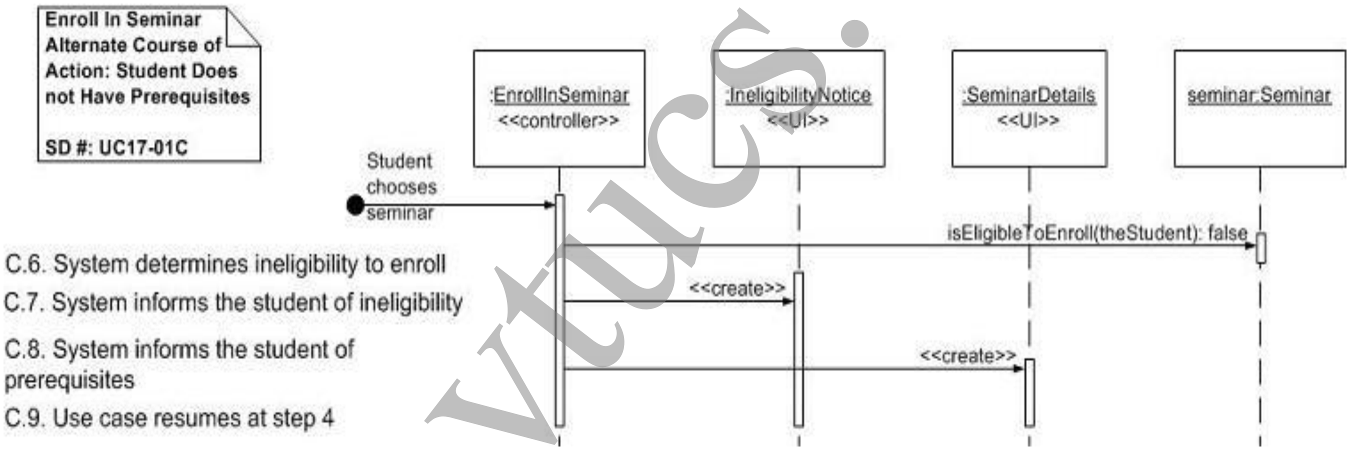
Return values are optionally indicated using a dashed arrow with a label indicating the return value. For example, the return value *theStudent* is indicated coming back from the *Student* class as the result of invoking a message, whereas no return value is indicated as the result of sending the message *isEligibleToEnroll(theStudent)* to*Seminar*. My style is not to indicate the return values when it's obvious what is being returned, so I don't clutter my sequence diagrams (as you can see, sequence diagrams get complicated fairly quickly). Figure 5 shows an alternate way to indicate return values using the format *message: returnValue* for messages, as you with*isEligibleToEnroll(theStudent): false.*

Notice the use of stereotypes throughout the diagram. For the boxes, I applied the stereotypes <<actor>>, <<controller>>, and <<UI>> indicating they represent an actor, a controller class, or a user interface (UI) class, respectively. I've also used visual stereotypes on some diagrams – a stick figure for actors; the robustness diagram visual stereotypes for controller, interface, and entity objects; and a drum for the database. Stereotypes are also used on messages. Common practice on UML diagrams is to indicate creation and destruction messages with the stereotypes of <<create>> and <<destroy>>, respectively. For example, you see the*SecurityLogon* object is created in this manner (actually, this message would likely be sent to the class that would then result in a return value of the created object, so I cheated a bit). This object later destroys itself in a similar manner, presumably when the window is closed.

I used a UML note in Figure 4; notes are basically free-form text that can be placed on any UML diagram, to provide a header for the diagram ,indicating its title and identifier (as you may have noticed, I give unique identifiers to all artifacts that I intend to keep). Notes are depicted as a piece of paper with the top-right corner folded over. I also used a note to indicate future work that needs to be done, either during analysis or design, in this diagram□ the*qualifications()* message likely represents a series of messages sent to the student object. Common UML practice is to anchor a note to another model element with a dashed line when appropriate, in this case the note is attached to the message.

Although Figure 4 models the logic of the basic course of action for the *Enroll in Seminar* use case how would you go about modeling alternate courses? The easiest way to do so is to create a single sequence diagram for each alternate course, as you see depicted in Figure 5. This diagram models only the logic of the alternate course, as you can tell by the numbering of the steps on the left-hand side of the diagram, and the header note for the diagram indicates it is an alternate course of action. Also notice how the ID of this diagram includes that this is alternate course *C,* yet another modeling rule of thumb I have found useful over the years.

Figure 5. An alternate course of action for the Enroll in Seminar use case.



Let's consider other sequence diagramming notation. Figure 5 includes an initial message, *Student chooses seminar*, which is indicated by the filled in circle. This could easily have been indicated via a method invocation, perhaps *enrollIn(seminar)*. Figure 6 shows another way to indicate object creation – sending the *new*message to a class. We've actually seen three ways to achieve this, the other two being to send a message with the <<create>> stereotype and/or to send a message into the side of the classifier symbol (for example in Figure 4 the message going into the side of *EnrollInSeminar* or in Figure 6 the message going into the side of*StudentInfoPage*. My advice is to choose one style and stick to it.

Figures 6 and 7 each depict a way to indicate looping logic. One way is to show a frame with the label loop and a constraint indicating what is being looped through, such as *for each seminar* in Figure 6. Another approach is to simply precede a message that will be invoked several times with an asterisk, as you see in Figure 7 with the inclusion of the *Enroll in Seminar* use case.

Figure 6. Outputting transcripts.

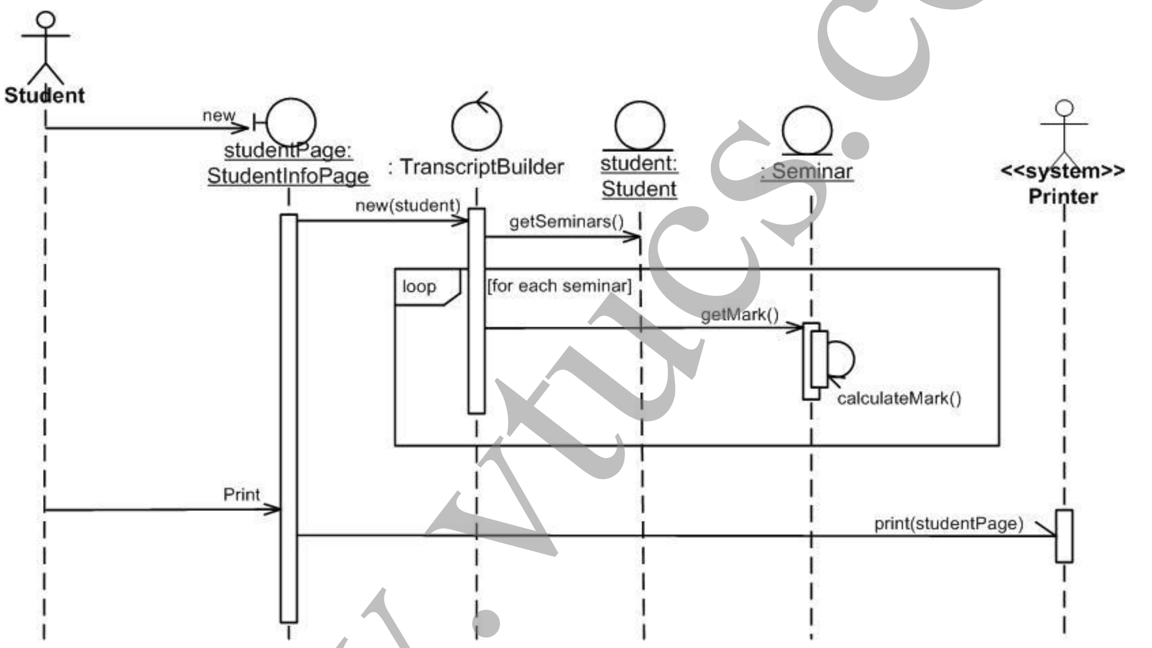
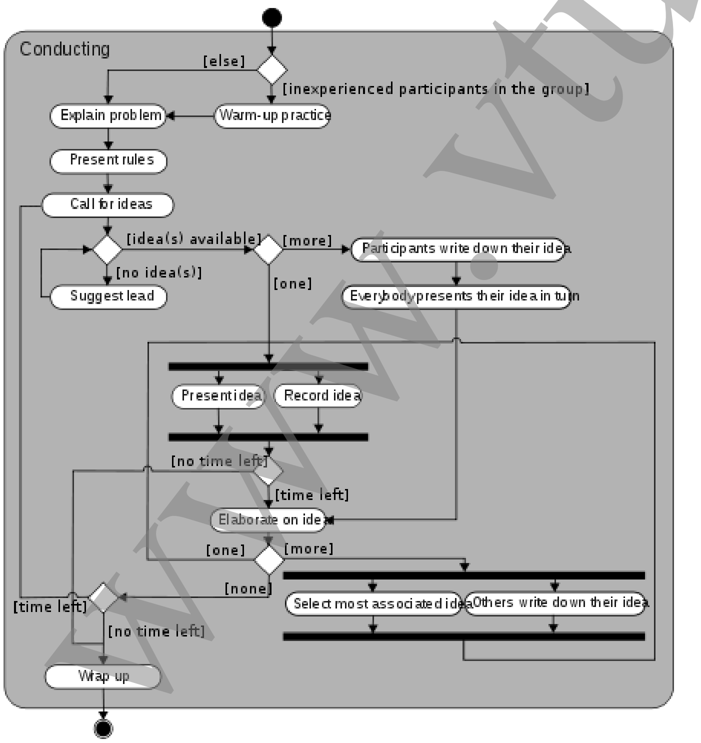


Figure 6 includes an asynchronous message, the message to the system printer which has the partial arrowhead. An asynchronous message is one where the sender doesn't wait for the result of the message, instead it processes the result when and if it ever comes back. Up until this point all other messages have been synchronous, messages where the sender waits for the result before continuing on. It is common to send asynchronous messages to hardware devices or autonomous software services such as message buses.

Activity model

Activity diagramsare graphical representations ofworkflows of stepwise activities and actions with support for choice, iteration and concurrency.[1] In theUnified Modeling Language, activity diagrams can be used to describe the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.



Activity diagrams are constructed from a limited repertoire of shapes, connected with arrows. The most important shape types:

- *rounded rectangles* represent *activities*;
- *diamonds* represent *decisions*;
- *bars* represent the start (*split*) or end (*join*) of concurrent activities;
- a *black circle* represents the start (*initial state*) of the workflow;
- an *encircled black circle* represents the end (*final state*).

Arrows run from the start towards the end and represent the order in which activities happen.

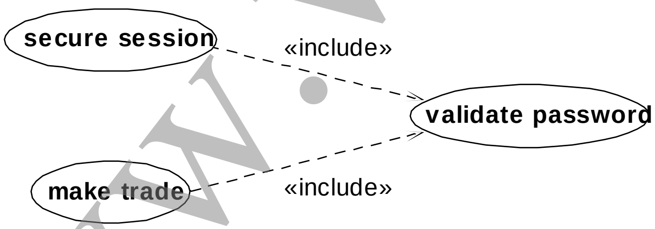
Hence they can be regarded as a form of flowchart. Typical flowchart techniques lack constructs for expressing concurrency. However, the join and split symbols in activity diagrams only resolve this for simple cases; the meaning of the model is not clear when they are arbitrarily combined with decisions or loops.

While in UML 1.x, activity diagrams were a specialized form of state diagrams, in UML 2.x, the activity diagrams were reformalized to be based on Petri net-like semantics, increasing the scope of situations that can be modeled using activity diagrams. These changes cause many UML 1.x activity diagrams to be interpreted differently in UML 2.x.

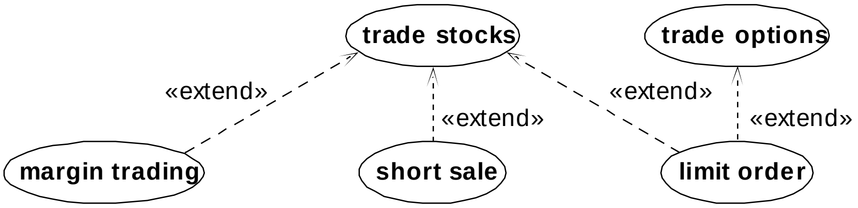
Use Case Relationships

This section describes how to relate use cases to each other. A dashed line between use cases is used to indicate these relationships.

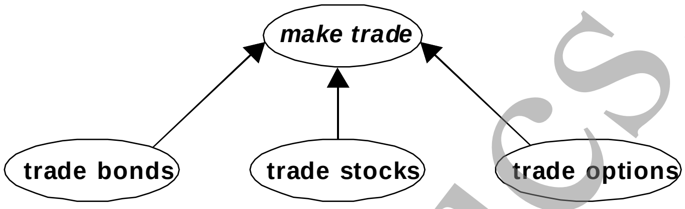
- **Include** - Subroutine - Factors out and organizes common subtasks. Extra behavior is added into a base use case. This behavior describes the insertion explicitly. The included use case is not a complete process. Use "include" when multiple use cases have a common function that can be used by all. Dashed line with arrow points to subroutine use case.



- **Extend** - Rarely used - Must perform a pre-task (Used only for critical order). The base and extended use cases are complete processes on their own. The base use case does not know about the extended use case. Arrow points to event that comes first.



- **Generalization-specialization (Gen-Spec)** - The gen-spec use case adds features to a generic use case. The gen-spec use case inherits features of the base use case. The gen spec can be used for use cases and actors since both can be specialized.



Include relationships

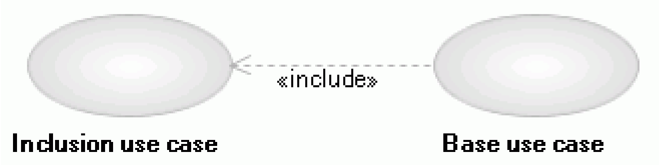
In UML modeling, an include relationship is a relationship in which one use case (the base use case) includes the functionality of another use case (the inclusion use case). The include relationship supports the reuse of functionality in a use-case model.

You can add include relationships to your model to show the following situations:

- The behavior of the inclusion use case is common to two or more use cases.
- The result of the behavior that the inclusion use case specifies, not the behavior itself, is important to the base use case.

Include relationships usually do not have names. If you name an include relationship, the name is displayed beside the include connector in the diagram.

As the following figure illustrates, an include relationship is displayed in the diagram editor as a dashed line with an open arrow pointing from the base use case to the inclusion use case. The keyword «include» is attached to the connector.



Extend relationships

In UML modeling, you can use an extend relationship to specify that one use case (extension) extends the behavior of another use case (base). This type of relationship reveals details about a system or application that are typically hidden in a use case.

The extend relationship specifies that the incorporation of the extension use case is dependent on what happens when the base use case executes. The extension use case owns the extend relationship. You can specify several extend relationships for a single base use case.

While the base use case is defined independently and is meaningful by itself, the extension use case is not meaningful on its own. The extension use case consists of one or several behavior sequences (segments) that describe additional behavior that can incrementally augment the behavior of the base use case. Each segment can be inserted into the base use case at a different point, called an extension point.

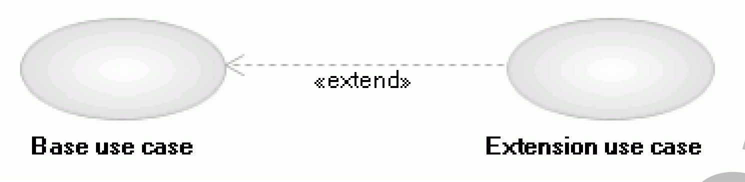
The extension use case can access and modify the attributes of the base use case; however, the base use case is not aware of the extension use case and, therefore, cannot access or modify the attributes and operations of the extension use case.

You can add extend relationships to a model to show the following situations:

- A part of a use case that is optional system behavior
- A subflow is executed only under certain conditions
- A set of behavior segments that may be inserted in a base use case

Extend relationships do not have names.

As the following figure illustrates, an extend relationship is displayed in the diagram editor as a dashed line with an open arrowhead pointing from the extension use case to the base use case. The arrow is labeled with the keyword «extend».



Generalization relationships

In UML modeling, a generalization relationship is a relationship in which one model element (the child) is based on another model element (the parent). Generalization relationships are used in class, component, deployment, and use-case diagrams to indicate that the child receives all of the attributes, operations, and relationships that are defined in the parent.

To comply with UML semantics, the model elements in a generalization relationship must be the same type. For example, a generalization relationship can be used between actors or between use cases; however, it cannot be used between an actor and a use case.

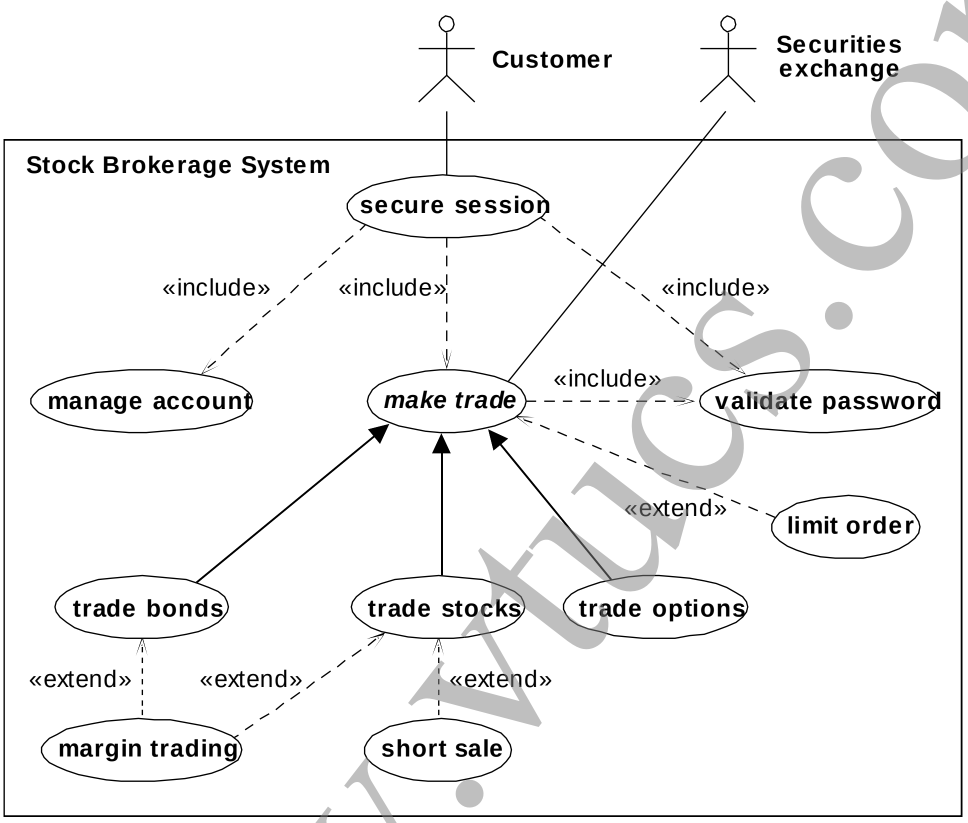
You can add generalization relationships to capture attributes, operations, and relationships in a parent model element and then reuse them in one or more child model elements. Because the child model elements in generalizations inherit the attributes, operations, and relationships of the parent, you must only define for the child the attributes, operations, or relationships that are distinct from the parent.

The parent model element can have one or more children, and any child model element can have one or more parents. It is more common to have a single parent model element and multiple child model elements.

Generalization relationships do not have names.

As the following figures illustrate, a generalization relationship is displayed in the diagram editor as a solid line with a hollow arrowhead that points from the child model element to the parent model element.

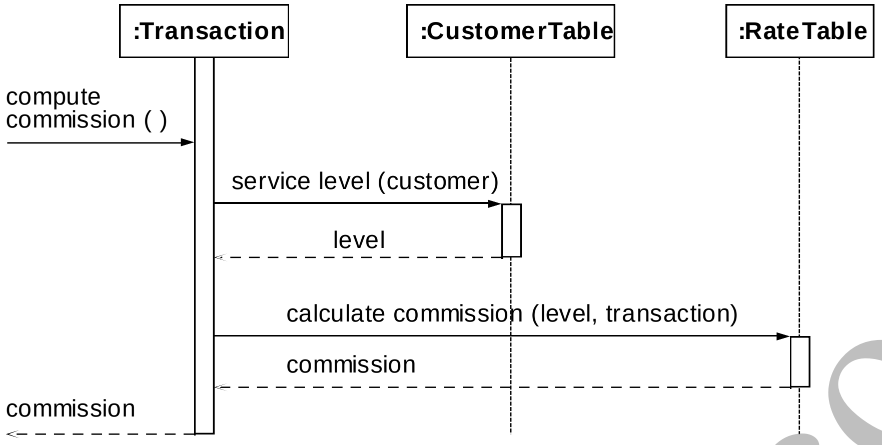
Combination of use case relationship



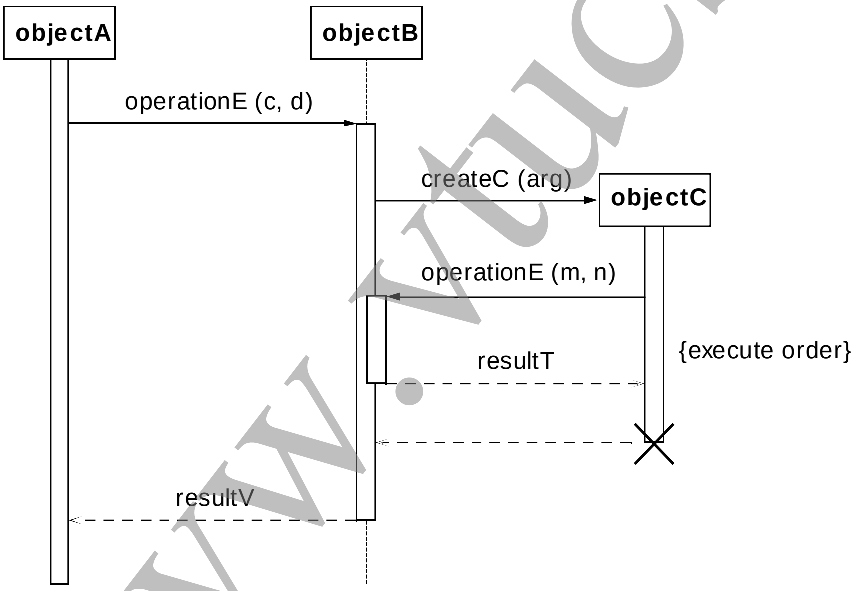
Procedural Sequence Models

1. Sequence diagrams with passive objects

A passive object is not activated until it has been called. Once the execution of an operation completes and control returns to the caller, the passive object becomes inactive.



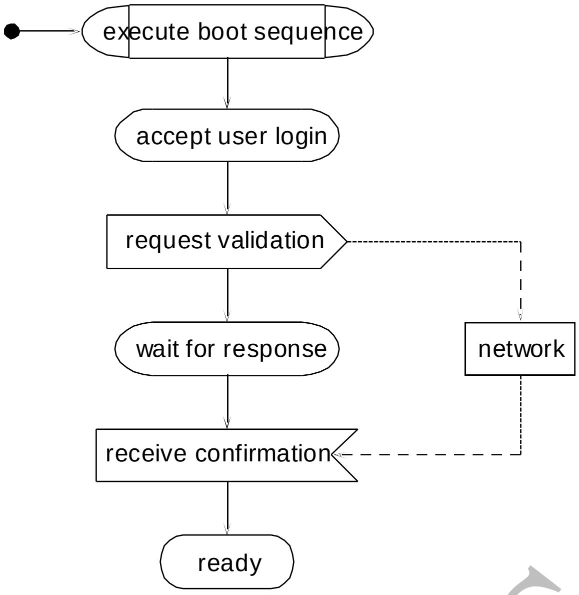
2. Sequence diagrams with transient objects



Special constructs for activity models

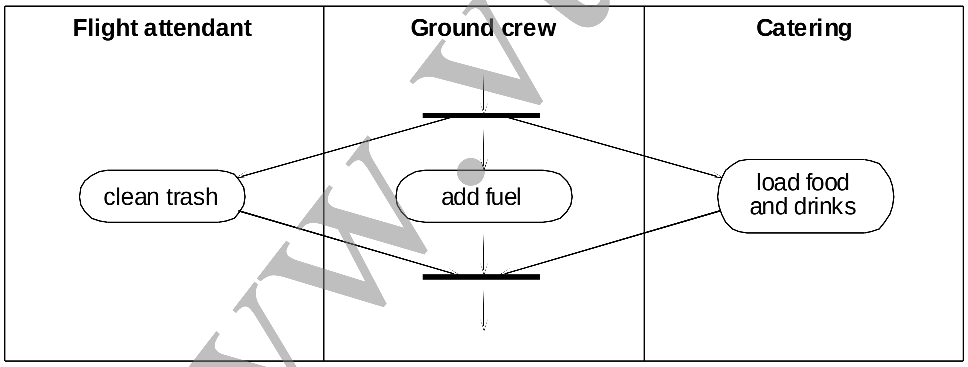
1. Sending and receiving signals.

The figure shows the sending of a signal as a convex pentagon and receiving a signal as a concave pentagon.



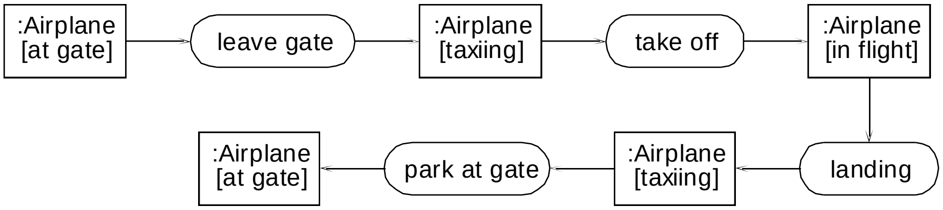
2. Swimlanes

Every column in partition of an activity diagram is called a swimlane as shown in the below figure



3. Object flows

frequently the same pbject goes through several states during execution of an activity diagram.



An activity diagram showing object flows among different object states has most of the advantages of a data flow diagram without most of their disadvantages.

UNIT 4

PROCESS OVERVIEW, SYSTEM CONCEPTION, DOMAIN ANALYSIS

Since conception is the primary part in the realization of a computer system and in order to help designers describe their software, several languages and tools such the UML modeling language have been proposed in the literature. UML knew an important success for the conception of object oriented systems. In this paper, we propose a new approach of conception and implementation of object oriented expert system based on the UML. For this we introduce our approach of design of the object oriented expert system based on UML, then we define an extension of the CLIPS, called VCLIPS_UML, in

order to support UML. VCLIPS_UML brings two main improvements to CLIPS. The first improvement permits an easy access and modification of the CLIPS knowledge base. The user introduces his knowledge base described with the UML class and the object diagram; VCLIPS_UML gives the corresponding script directly. The second improvement concerns the ease of its utilization by making the syntactic and semantics aspects of the CLIPS programming language more transparent. The implementation of VCLIPS_UML is carried out in a way to make it expandable and portable.

Development stages

- System Conception
  - Conceive an application and formulate tentative requirements
- Analysis
  - Deeply understand the requirements by constructing models
- System design
  - Devise the architecture
- Class design

  - Determine the algorithms for realizing the operations
- Implementation
  - Translate the design into programming code and database structures
- Testing
  - Ensure that the application is suitable for actual use and actually satisfies requirements
- Training
  - Help users master the new application
- Deployment
  - Place the application in the field and gracefully cut over from legacy application
- Maintenance
  - Preserve the long term viability of the application

System Conception

Analysis

To specify *what* must be done.

- Domain analysis focuses on real-world things whose semantics the application captures.
- Application analysis addresses the computer aspects of the application that are visible to users.

System Design

- Devise a high-level strategy — the architecture — for solving the application problem.
- The choice of architecture is based on the requirements as well as past experience.

Class Design

- To emphasis from application concepts toward computer concepts.
- To choose algorithms to implement major system functions.

DEVELOPMENT LIFE CYCLE

Waterfall Development

- The stages in a rigid linear sequence with no backtracking.
- Suitable for well-understood applications with predictable outputs from analysis and design.

Iterative Development

- First develop the nucleus of a system, then grow the scope of the system…
- There are multiple iterations as the system evolves to the final deliverable.
- Each iteration includes a full complement of stages:
  - analysis, design, implementation, and testing.

SYSTEM CONCEPTION

- System conception deals with the genesis of an application.

DEVISING A SYSTEM CONCEPT

- New functionality
- Streamlining
- Simplification automate manual process
- Integration
- Analogies
- Globalization

ELABORATING A CONCEPT

Good system concept must answer the following questions

- Who is the application for?
  - Stakeholders of the system
- What problems will it solve?
  - Features
- Where will it be used?

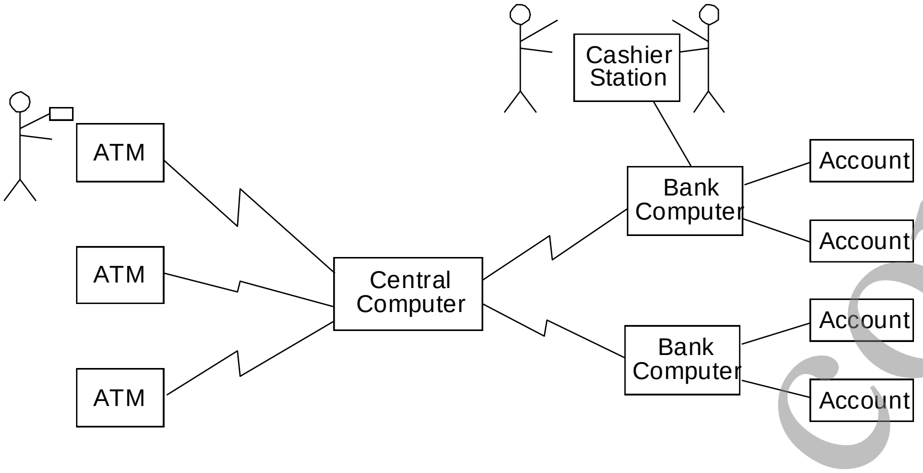
  - Compliment the existing base, locally, distributed, customer base
- When is it needed?
  - Feasible time, required time
- Why is it needed?
  - Business case
- How will it work?
  - Brainstorm the feasibility of the problem.

ATM CASE STUDY

- Who is the application for?
  - We are vendor building the software
- What problems will it solve?
  - Serve both bank and user
- Where will it be used?
  - Locations throughout the world
- When is it needed?
  - Revenue , investment
- Why is it needed?
  - Economic incentive. We have to demonstrate the techniques in the book
- How will it work
  - N-tier architecture, 3-tier architecture

PREPARING A PROBLEM STATEMENT

Design the software to support a computerized banking network including both human cashiers and automatic teller machines (ATMs)to be shared by a consortium of banks. Each bank provides its own computer to maintain own accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank's computers. Human cashiers enter account and transaction data.


Cashier Station
ATM
ATM
ATM
Central Computer
Bank Computer
Bank Computer
Account
Account
Account
Account

Analysis

Purpose

To produce a complete, consistent, and unambiguous

description of

- the problem domain and

- the functional requirements of the system.

In analysis, the models concentrate on describing *what* a system does, rather than *how* it does it.

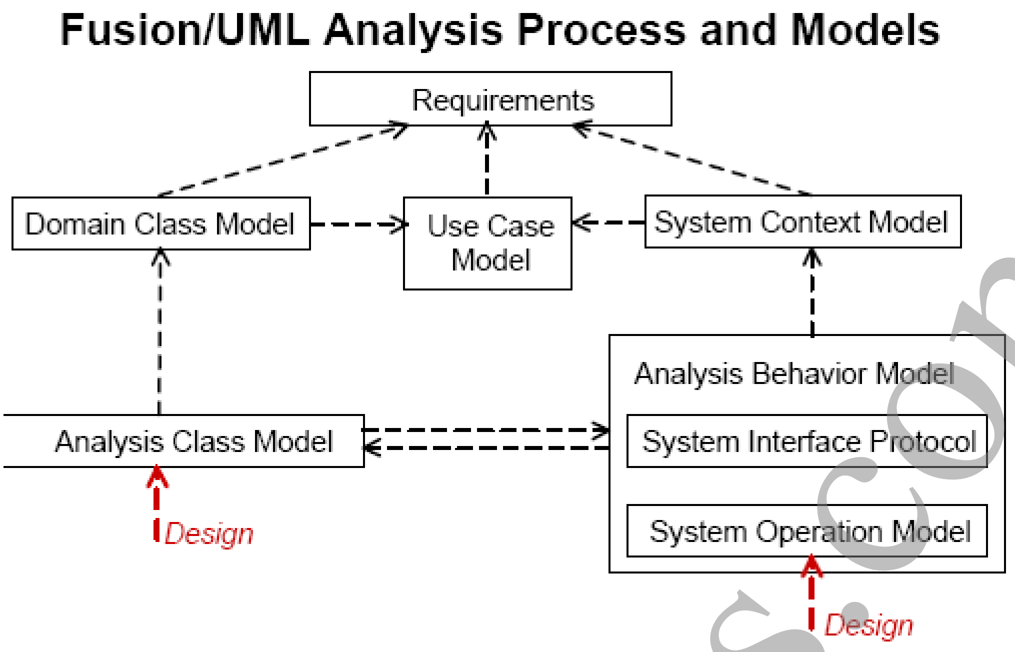
Results

Class Model: Defines the static structure of the information

in the system.

Behavior Model: Defines the input and output

communication of the system.



Process of Analysis

0. Develop a Use Case Model

1. Develop the Domain Class Model for the problem

domain

2. Develop the System Context Model: identify actors,

events and system operations.

3. Produce the Analysis Class Model by adding the system

boundary to the Domain Class Model.

4. Develop the Analysis Behavior Model

4.1 Develop the System Interface Protocol

4.2 Develop the System Operation Model

5. Check the Analysis Models for consistency and completeness

Domain Class Model

The *Domain Class Model* captures the *concepts* in the *domain* of the problem, and the relationships between them. It establishes the vocabulary of the problem domain.

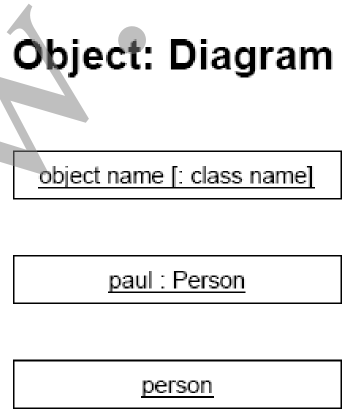
The Class Model *notation* is similar to the entity relationship model notation.

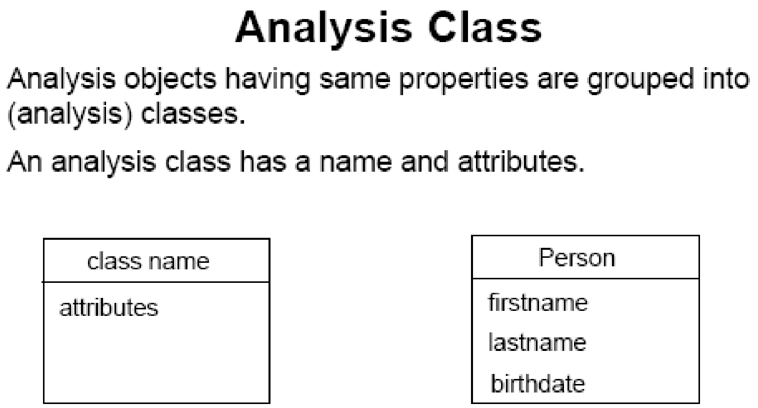
Analysis Object

An object is a thing or concept that can be distinctly *identified*, e.g. a specific person, organization, machine, or event.

The identity of an object cannot be changed. An object can have values associated with it, called *(value) attributes*, e.g. a person object could have the attributes name, address and occupation. The values of the attributes can change, but not their number and names.

Attributes of an analysis object are not allowed to be objects. An analysis object does not have a method interface.





Analysis Class

Analysis objects having same properties are grouped into (analysis) classes.

An analysis class has a name and attributes.

class name
attributes

Person
firstname lastname birthdate

Association

Associations are the “glue” that holds together a system.Without associations, there is only a set of unconnected classes.

An association models a relationship between objects.The existence of an association is conditional: all related classes must exist. Similarly, an occurrence of an association can only exist if the connected objects all exist.

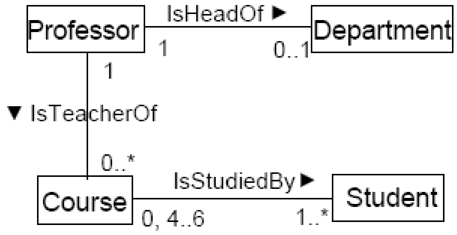
Association: Multiplicity

The multiplicity (or cardinality) defines the number of objects which are allowed to be associated with each other in an association. Multiplicity is shown by annotating the line end, called the association end, connecting the class. The full form is a range (or even a set of ranges), e.g. 0..1, 1..1, 1..4, 0..*, 1..*, an asterisk * meaning that the upper

limit is unlimited. Sometimes short forms are used: 1 for 1..1, 2 for 2..2, etc.,

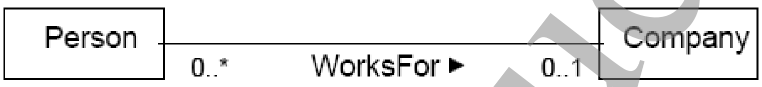
and * for 0..*. Multiplicity in UML is written on the opposite branch of the association compared with other (especially French) entityrelationship model notations.

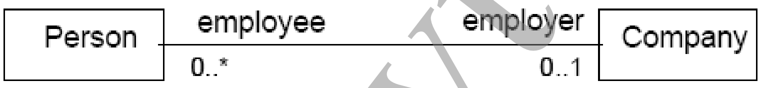
Association: Multiplicity



Association: Roles

An association need not have a name. Usually, role names are more convenient because they avoid the problem of which way to read the name and they provide names for navigation and code generation.





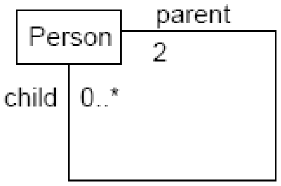
Note: Without the arrow showing the read direction or the role names, the meaning of the association could be that the company works for the person.

Association: Roles

Role names clarify the semantics of an association, especially when reflexive.

As a child, a person has exactly two parents.

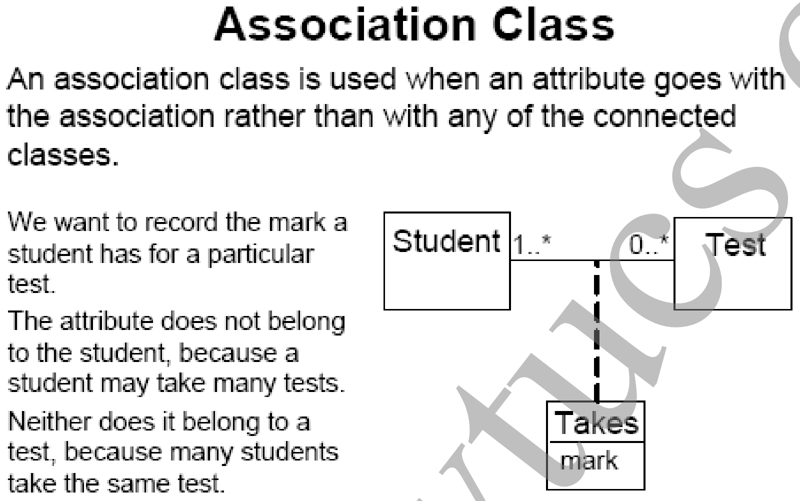
As a parent, a person may have several children.



Association Class

An association class is an association that is also a class. An association class has both association and class properties: it connects two or more classes, and it also has attributes and sometimes operations. Like for all associations, the identity of an occurrence

stems from the connected objects. Like all classes, an association class can have attributes, and sometimes operations. In the UML, an association class can participate in an association. We do not recommend this practice.



Association Class

An association class is used when an attribute goes with the association rather than with any of the connected classes.

We want to record the mark a student has for a particular test.

The attribute does not belong to the student, because a student may take many tests.

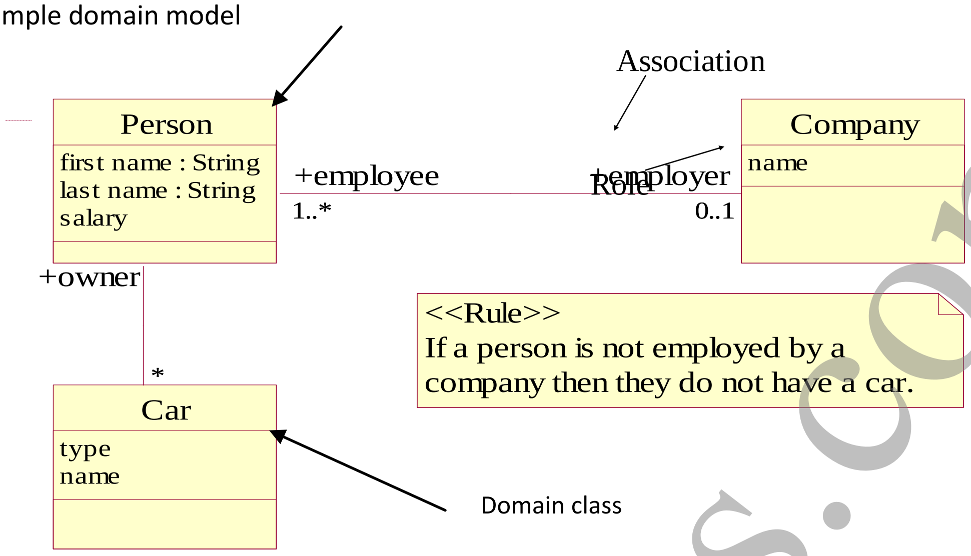
Neither does it belong to a test, because many students take the same test.

“A domain model captures the most important types of objects in the context of the business. The domain model represents the ‘things’ that exist or events that transpire in the business environment.”

- Gives a conceptual framework of the things in the problem space
- Helps you think – focus on semantics
- Provides a glossary of terms – noun based
- It is a static view - meaning it allows us convey time invariant business rules
- Foundation for use case/workflow modelling

- Based on the defined structure, we can describe the state of the problem domain at any time.

Simple domain model



Features of a domain model

- The following features enable us to express time invariant static business rules for a domain:-
  - **Domain classes** – each domain class denotes a type of object.
  - **Attributes** – an attribute is the description of a named slot of a specified type in a domain class; each instance of the class separately holds a value.
  - **Associations** – an association is a relationship between two (or more) domain classes that describes links between their object instances. Associations can have roles, describing the multiplicity and participation of a class in the relationship.
  - **Additional rules** – complex rules that cannot be shown with symbology can be shown with attached notes.

Domain classes

- Each domain class denotes a type of object. It is a descriptor for a set of things that share common features. Classes can be:-
  - *Business objects* - represent things that are manipulated in the business e.g. *Order*.
  - *Real world objects* – things that the business keeps track of e.g. *Contact*, *Site*.
  - *Events that transpire* - e.g. *sale* and *payment*.
- A domain class has attributes and associations with other classes (discussed below). It is important that a domain class is given a good description

Perform the following in very short iterations:

- Make a list of candidate domain classes.
- Draw these classes in a UML class diagram.
- If possible, add brief descriptions for the classes.
- Identify any associations that are necessary.
- Decide if some domain classes are really just attributes.
- Where helpful, identify role names and multiplicity for associations.
- Add any additional static rules as UML notes that cannot be conveyed with UML symbols.
- Group diagrams/domain classes by category into packages.

Concentrate more on just identifying domain classes in early iterations !

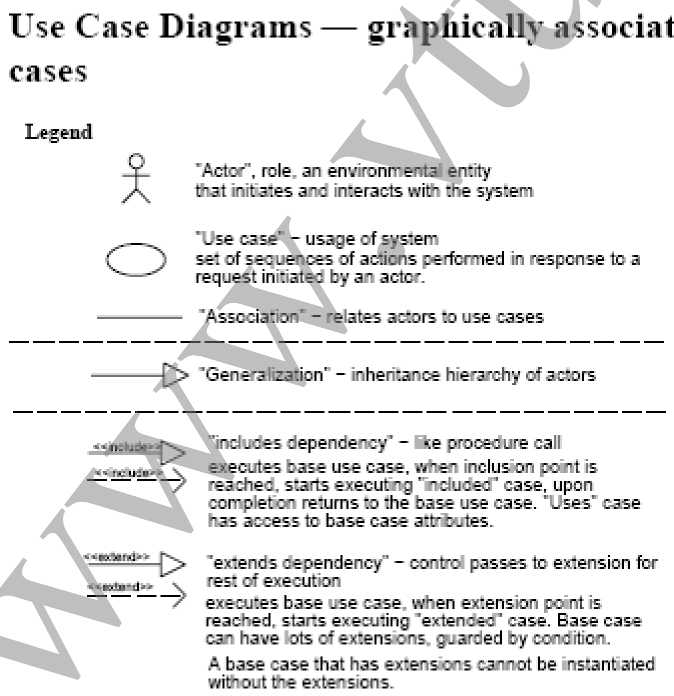
- An obvious way to identify domain classes is to identify nouns and phrases in textual descriptions of a domain.
- Consider a use case description as follows:-
- 1. **Customer** arrives at a **checkout** with **goods** and/or **services** to purchase.
- 2. **Cashier** starts a new **sale**.
- 3. **Cashier** enters **item identifier**.
- 4. System records the **sale line item** and presents the **item description, price** and running **total**.

- A domain class sounds like an attribute if: -

- It relies on an associated class for it's identity – e.g. 'order number' class associated to an 'order' class. The 'order number' sounds suspiciously like an attribute of 'order'.
- It is a simple data type – e.g. 'order number' is a simple integer. Now it really sounds like an attribute!

Use Cases—sets of sequences of actions

By a sequence of actions, I mean an interaction, a sequence of messages between the system and things outside of the system(users). By a set of sequences of actions, I mean that a single use case covers several related scenarios and specifies the user's view of essential system behaviour, including variants. This is important. Use cases do not model all tasks that the system performs. They are written from the user's perspective, and model just system functionality that can be initiated by the environment. We want to think about the system's functionality in terms of the different ways the system can be used.



Use Case Diagrams — graphically associate users with use cases

Legend

"Actor", role, an environmental entity
that initiates and interacts with the system

"Use case" – usage of system
set of sequences of actions performed in response to a request initiated by an actor.

"Association" – relates actors to use cases

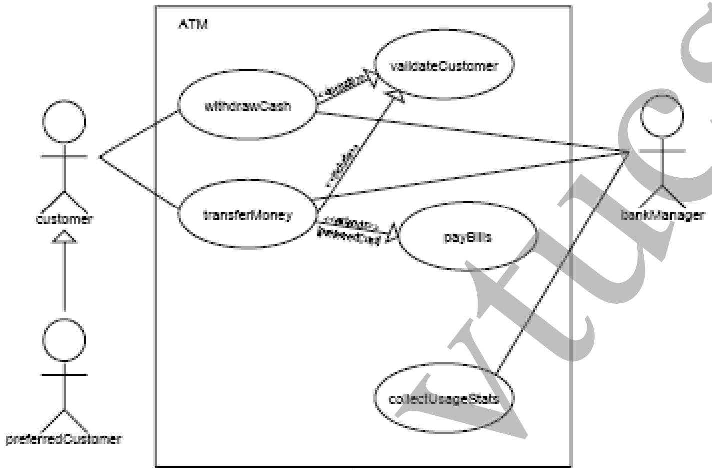
"Generalization" – inheritance hierarchy of actors

"includes dependency" – like procedure call
executes base use case, when inclusion point is reached, starts executing "included" case, upon completion returns to the base use case. "Uses" case has access to base case attributes.

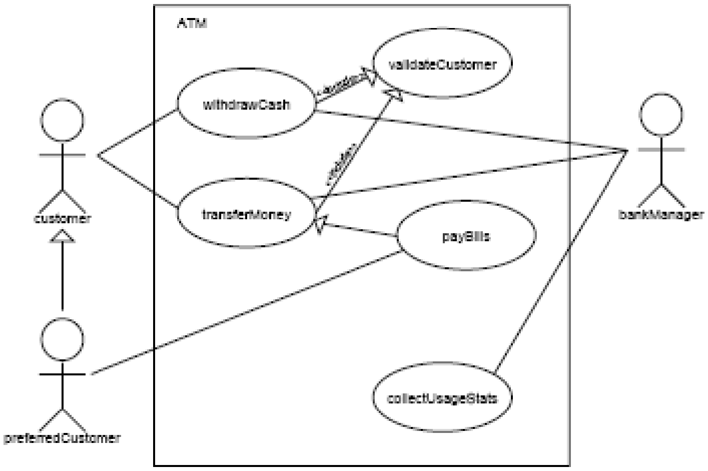
"extends dependency" – control passes to extension for rest of execution
executes base use case, when extension point is reached, starts executing "extended" case. Base case can have lots of extensions, guarded by condition.
A base case that has extensions cannot be instantiated without the extensions.

The system is drawn as a box. Use cases are inside and actors are outside the box. A use case diagram will usually show all of the system's use cases, all of the actors that might use the system, and associate actors with the use cases they might be involved in.

One way:



Another way:



Use Case Relationships

As Fowler and Scott say:

_ Use *include* when you are repeating yourself in two or more separate use cases and you want to avoid repetition.

_ Use *generalization* when you are describing a variation on normal behavior and you wish to describe it casually.

_ Use *extend* when you are describing a variation on normal behavior and you wish to use the more controlled form, declaring your extension points in your base use case.

Sequence Diagrams = event traces

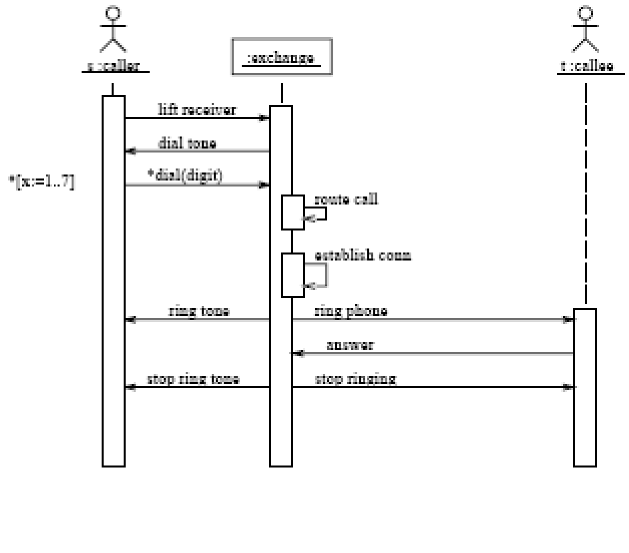
Scenarios may be depicted using sequence diagrams. _ column = object; give name and class, and underline the name.

_ vertical line is life line of object.

_ rectangle on life line; called several things. _ horizontal arrow expresses event conveyed by source object to target object.

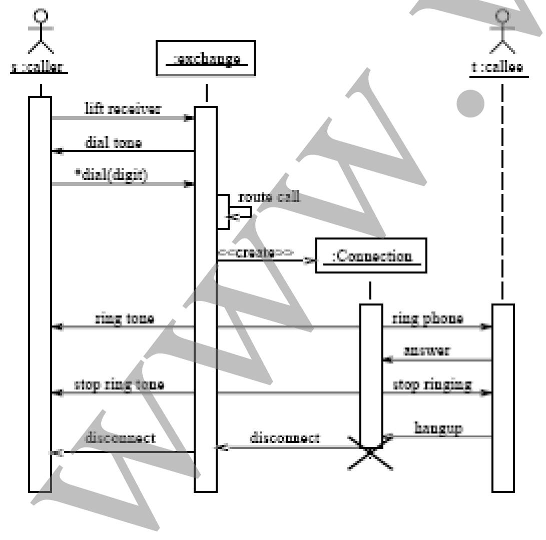
_ “*” or note in the margin expresses iteration._ looping arrow expresses recursion or calls to self.

Ex: Telephone Switch



As you elaborate the problem domain and specify in more detail the entities that the system senses and controls, the self calls become messages to these other entities._ create transient objects in response to a <<create>>._ destroy transient objects either because it receives a <<destroy>> message or because it destroys itself. Now for an expansion of the previous sequence diagram:

Ex: Telephone Switch



Class Diagram

Class Diagram—depicts the types of objects involved in the problem and their relationships Before we begin looking at class diagrams, I would like to point out an important subtlety in the way people use them. There are really three perspectives from which a class diagram can be read or written.

Three Perspectives of UML diagrams

1. Conceptual—the diagram represents the problem that the software should solve In conceptual diagrams, classes model real-world entities in the system's environment. It may very well be that these concepts will also relate to the software classes that implement them. E.g., customers, from the problem domain, are clearly related to customer records, from the system. Phone numbers and bills are real-world entities that might be modelled as classes in a software system, but there need not be a direct mapping from a conceptual class to a software class. Whether there is would be a decision for the software designer.

2. Design—the diagram depicts only the interfaces of software classes.

Design diagrams are supposed to be used as the medium in which to carry out design without having to consider implementation details. Class interfaces were one of the key

advances of ADTs and effective OO programming, namely, that we can abstract from a class to its interface, which describes the services that class offers to other classes

without describing how that class is implemented. This enables us to visualize, understand, and reason about larger designs, in terms of classes and their services, and not get bogged down in implementation details.

3. Implementation—the diagram depicts interfaces and implementations of classes.

An implementation class diagram shows all of the operations implemented by a class, regardless of whether the operations are accessible to other classes and whether classes have implementations attached to them. The perspective of a diagram has a huge impact on how one thinks about a class diagram and what decisions one makes when

constructing the diagram.

UML defines several models for representing systems:

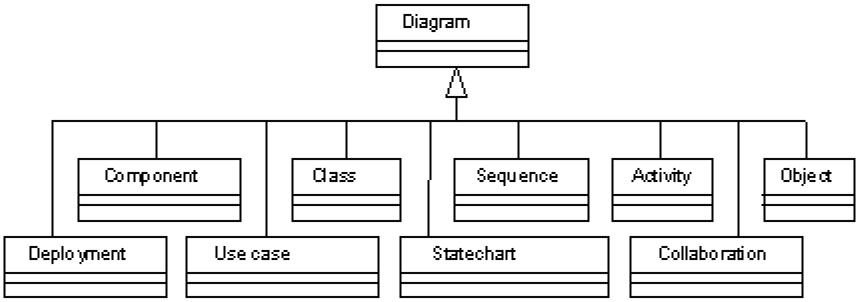
- The **class model** captures the **static structure**
- The **state model** expresses the **dynamic behavior of objects.**
- The **use case model** describes the **requirements of the user.**
- The **interaction model** represents the **scenarios and messages flows**
- The **implementation model** shows the **work units**
- The **deployment model** provides details that pertain to **process allocation**

UML Diagrams

UML defines nine different types of diagram:

1. **Use case diagrams:** represent the **functions** of a system from the **user's point of view**.
2. **Sequence diagrams**: are a temporal representation of **objects and their interactions.**
3. **Collaboration diagrams**: spatial representation of **objects, links**, and **interactions.**
4. **Object diagrams :**represent **objects** and their **relationships** and correspond to simplified collaboration diagrams that do not represent message broadcasts.
5. **Class diagrams** represent the static structure in terms of classes and relationships
6. **Statechart diagrams**: represent the **behavior of a class** in terms of states at **run time**.
7. **Activity diagrams:** represent the **behavior of an operation** as a set of actions
8. **Component diagrams:** represent the **physical components** of an application
9. **Deployment diagrams:** represent the deployment of **components** on particular pieces of **hardware**

The different types of diagrams defined by UML



Relationship among various UML Diagrams in OOAD (Object Oriented Analysis and design) is illustrated in the following diagrams of **Business Model**, **Use Case** Diagram , Sequence Diagram , Class Diagram and Code generation..