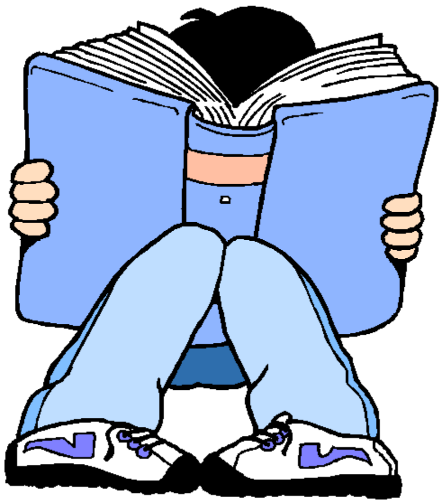


OBJECT ORIENTED MODELING AND DESIGN

OBJECT ORIENTED MODELING AND DESIGN

RT 701 **2+1+0**

Module 1

Introduction: object oriented development-modeling concepts – object oriented methodology – models – object oriented themes-Object Modeling– links and associations – advanced links and association concepts – generalization and inheritance - grouping constructs – a sample object model

Advanced Object Modeling: aggregation – abstract classes – generalization as extension and restriction – multiple inheritance – metadata – candidate keys – constraints.

Module 2

Dynamic modeling: Events and states – Operations – Nested state diagrams – Concurrency – Advanced dynamic modeling concepts – A sample dynamic model – Relationship of Object and Dynamic models.

Functional modeling: Functional models – Data Flow Diagrams - Specifying operations – Constraints – A sample functional model – Relation of functional to Object and Dynamic models.

Module 3

Analysis: Analysis in object modeling, dynamic modeling and functional modeling, Adding operations- Iterating the analysis

System Design: Breaking system into subsystems - Identifying concurrency-allocating subsystems to processors and tasks, managing of data stores. Handling of global resources- handling boundary conditions-Common Architectural Frameworks

Module 4

Object Design: Overview of Object design – Combining the three models – Designing algorithms – Design optimization – Implementation of control – Adjustment of inheritance - Design of association – Object representation – Physical packaging – Documenting design decisions-Comparison of methodologies

Module 5

Other Models: Booch's Methodology- Notations, models, concepts. Jacobson Methodology- architecture, actors and use-cases, requirement model, Analysis Model, Design model, Implementation model and Test Model-Unified Modeling Language (UML).

Text Book

1. Object Oriented Modeling and Design -JamesRumbaugh, Prentice Hall India

2. Object Oriented Analysis and Design with Applications - Grady Booch, Pearson Education Asia

References

1. Object Oriented Software Engineering - Ivan Jacobson, Pearson Education Asia
2. Object Oriented Software Engineering - Berno Bruegge, Allen H. Dutoit, Pearson Education Asia
3. Object Oriented Analysis and Design using UML - H. Srimathi, H. Sriram, A. Krishnamoorthy
4. Succeeding with the Booch OMT Methods -A practical approach - Lockheed Martin, Addison Wesley
5. UML and C++ practical guide to Object Oriented development - Richard C.Lee & William, Prentice Hall India

Module **1**

Introduction: object oriented development-modeling concepts – object oriented methodology – models – object oriented themes-Object Modeling– links and associations – advanced links and association concepts – generalization and inheritance - grouping constructs – a sample object model

Advanced Object Modeling: aggregation – abstract classes – generalization as extension and restriction – multiple inheritance – metadata – candidate keys – constraints.

1 Introduction

(i)Object

Data can be quantized into discrete, distinguishable entities which are called objects.

(ii) Object Oriented Approach means organizing software as a collection of discrete objects that incorporate both data structure and behavior.

2 Object Oriented Development is a new way of thinking about software based on abstractions that exist in the real world as well as in the program.

3 Object Oriented Methodology is a methodology for object oriented development and a graphical notation for representing objects oriented concepts. We can call this methodology as OMT. The methodology has the following stages:

1. Analysis - An analysis model is a concise, precise abstraction of what the desired system must do, not how it will be done. It should not contain any implementation details. The objects in the model should be application domain concepts and not the computer implementation concepts.
2. System design - The designer makes high level decisions about the overall architecture. In system design, the target system is organized as various subsystems based on both the analysis structure and the proposed architecture.
3. Object design - The designer builds a design model based on the analysis model but containing implementation details. The focus of object design is the data structures and algorithms needed to implement each cycle.
4. Implementation - The object classes and relationships developed during object design are finally translated into a particular programming language, database, or hardware implementation. During implementation, it is important to follow good software engineering practice so that the system can remain the traceability, flexibility and extensibility.

The OMT methodology uses three kinds of models

- Object Model-describes the static structure of the objects in a system and their relationships. This model mainly contains object diagrams.
- Dynamic Model-describes the aspects of a system that change over time. This model mainly contains state diagrams.
- Functional Model-describes the data value transformations within a system. This model contains the data flow diagrams

4 <u>Object Oriented Themes</u>:

There are several themes in an object oriented technology. These themes are not unique to object oriented systems. We can see some important themes:

- Abstraction-Abstraction consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental properties. In system development, it focuses only on what an object is and does, before deciding how it should be implemented.
- Encapsulation- It can also be called as information hiding. It consists of separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects. It is not unique to object oriented languages.
- Combining Data and Behavior-The caller of an operation need not consider how many implementations of a given operation exist. Operator polymorphism shifts the burden of deciding what implementation to use from the calling code to the class hierarchy. As an example, let us talk about an object oriented program calling a draw procedure for drawing different figures say a polygon, circle, or text. The decision of which procedure to use is made by each object, based on its class.
- Sharing- Inheritance of both data structure and behavior allows a common structure and behavior allows common structure to be shared among several similar subclasses without redundancy. The sharing of code using inheritance is one of the main advantages of object oriented languages.

(5) <u>Object Modeling:</u> A model is an abstraction of something for the purpose of understanding it before building it. It is easier to manipulate than the original entity.

An object model captures the static structure of a system, relationships between the objects, and the attributes and operations that characterize each class of objects.

This model is the most important one. Some basic concepts covered in object modeling:

(a) Objects: We can define an object as a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. All objects have identity and are distinguishable.eg:- Two apples can be described as 2 different objects even though they have the same descriptive properties.

(b) Classes: An ‘object class’ or ‘class’ describes a group of objects with similar properties, common behavior, common relationships to other objects, and common semantics.

Eg: - Suppose you have a person class –you can term J S, M S, etc. to be objects of that class.

(c)Object Diagrams: They provide a formal graphic notation for modeling objects, classes and their relationships to one another.

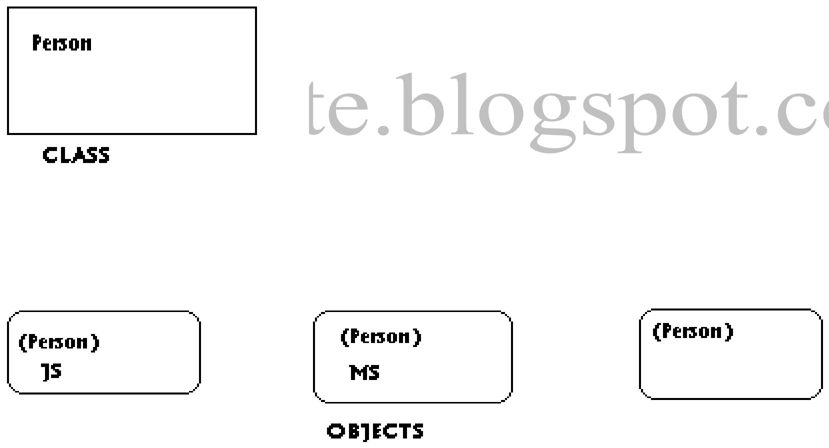


Fig.2.1 Classes and Objects

- Class Diagram: It is a schema, pattern or template for describing many possible instances of data.
- An Instance Diagram describes how a particular set of objects relate to each other. An instance diagram describes object instances.

- As the example above shows the symbol for an object instance, while that of class diagram is a rectangle.
- In the above example, JS, MS are instances of class Person.

(d)Attributes: An attribute is a data value held by objects in a class.

Eg: - In the above example, Name, age, etc.of a person are attributes.

Attributes are listed in the second part of the class box. So in the above example,

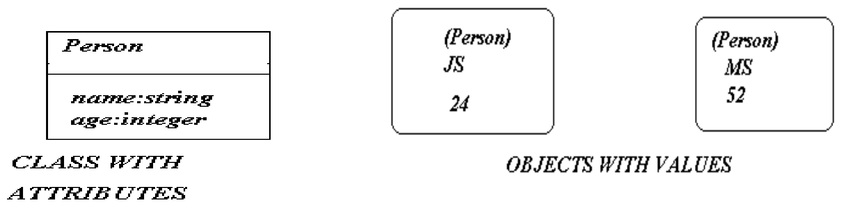


Fig.2.2 Attributes and values

(e) Operations and methods: An operation is a function or transformation that may be applied to or by objects in a class. Operations are listed in the lower third of the class box.

Eg: - consider the above example, change –job and change-address on Person class.

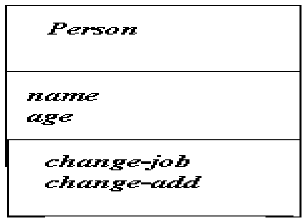


Fig.2.3 Operations

- Each attribute name may be followed by operational details, list of attributes and list of operations.
- Each operation name may be followed by operational details such as operational details such as argument list and result type.

6 Links and associations: They are used for establishing relationships among objects and classes.

Link- A link is a physical or conceptual connection between object instances .We can say that a link is an instance of an association.

Eg: - J S works for Simplex company.

Association –Describes a group of links with common structure and common semantics':- A person works for a company.

They are inherently bidirectional. It means that a binary association can be traversed in both directions. They are implemented as pointers.

The below figure shows a 1-1 association and corresponding links.

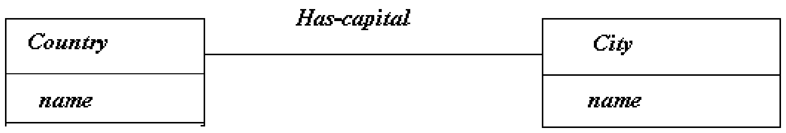


Fig. 2.4 Class Diagram

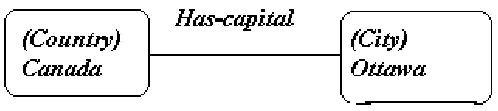


Fig.2.5 Instance Diagram

- Multiplicity –specifies how many instances of one class may relate to each instance of another class. The solid balls,"2+" are multiplicity symbols. Associations may be binary, ternary or higher order.

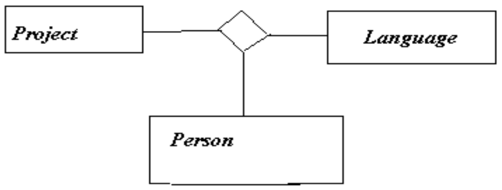


Fig.2.6 Ternary associations and links

7 Advanced Link and Association Concepts:

i) Link Attributes –A link attribute is a property of the links in an association. In the below figure access permission is an attribute of ‘Accessible by’.

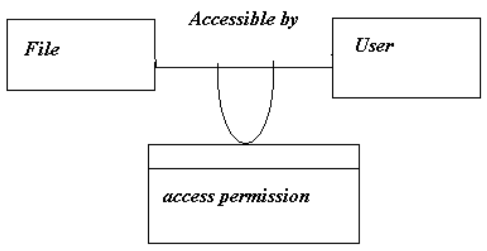


Fig.2.7 Link attribute for a many-many association

* The OMT notation for a link attribute is a box attached to the association by a loop

ii) Role names – A role is an end of an association. Role name is a name that uniquely identifies one end of an association.

Eg: - “Person works for a Company.” Here the person plays the role of an employee and the company the role of the employer.

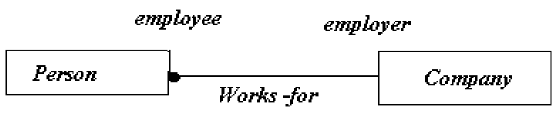


Fig.2.8 Role names for an association

iii) Ordering -Sometimes, the objects are explicitly ordered. For example, the figure below shows a workstation screen containing a number of overlapping windows. The windows are explicitly ordered, so only the topmost window is visible at any point on the screen.

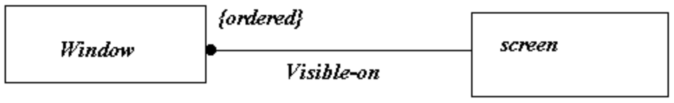


Fig.2.9 Ordered sets in an association

iv) Qualification – Qualifier is a special attribute that reduces the effective multiplicity of an association. It reduces the effective multiplicity of an association. We can qualify a one to many and many to many association.

For eg: - In the below figure a directory has many files .A file may only belong to a single directory. Within the context of a directory, the file name specifies a unique file .A file corresponds to a directory and a file name.

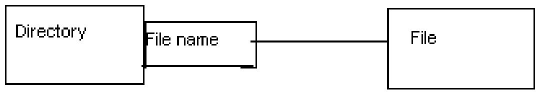


Fig.2.10 A qualified association

(v) Aggregation: Aggregation is the "part-whole" relationship in which objects representing the components of something are associated with an object representing the entire assembly. The figure shows a portion of an object model for a word processing. Here a document consists of many paragraphs, each of which consists of many sentences.

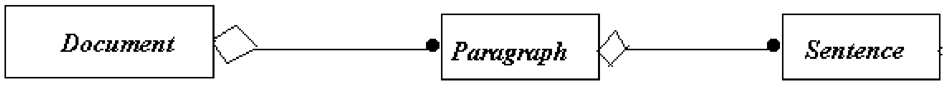
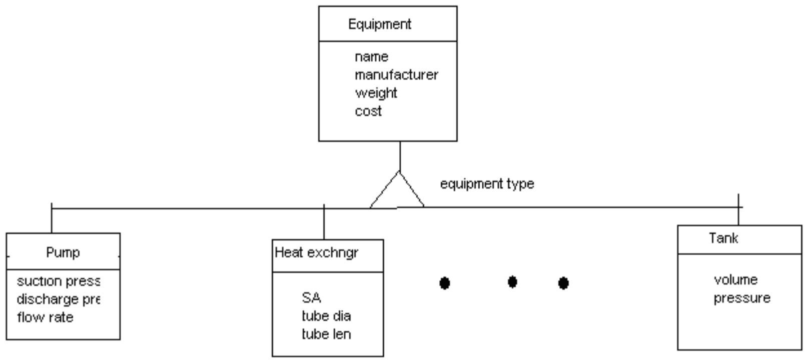


Fig. 2.11 Aggregation

8 Generalization and Inheritance: Generalization is the relationship between a class and one or more refined versions of it. The class being refined is called the 'superclass' and each refined version are called a 'subclass'. For example, Equipment is the superclass of Pump and tank. Attributes and operations that are common to a group of subclasses are attached to the superclass and shared by each subclass. Generalization is sometimes called a ‘is-a’ relationship. Each instance of a subclass is an instance of the superclass. The notation for generalization is a triangle connecting a superclass to



subclass.

Fig. 3.1 An inheritance hierarchy

The dangling ellipsis in the above figure indicates that there are additional subclasses that are not shown in the diagram. This may be since there is no room on the sheet and they are shown elsewhere or may be because enumeration of subclasses is still incomplete. You can also see equipment type written next to the triangle. This is called as a discriminator.

“A discriminator is an attribute of enumeration type that indicates which property of an object is being abstracted by a particular generalization relationship.”

What is the use of generalization? Generalization is a useful construct for both conceptual modeling and implementation.

Overriding Features:

I think you are all familiar with this term. A subclass may override a superclass feature by defining a feature with the same name. Here you can see that the subclass feature refines and replaces the superclass feature. What is the need for this feature? The reasons are many

- to specify behavior that depends on the subclass
- to tighten the specification of a feature.
- better performance

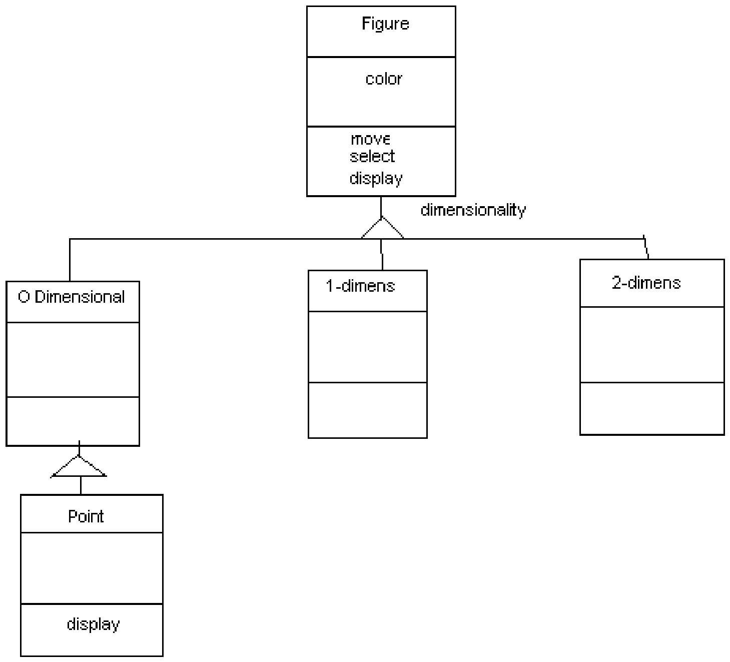


Fig. 3.2 Inheritance for graphic figures

In the above figure, you can see that display must be implemented separately for each kind of figure, although it is defined for any kind of figure.

10 Grouping Constructs:

a) Module- A module is a logical construct for grouping classes, associations, and generalizations. It captures one view of a situation.

For example: Let us say, electrical, plumbing, etc. are different views of a building.

An object model consists of one or more modules. Class names and association names must be unique within a module. Module names are usually listed at the top of each sheet. The same class may be referenced in many modules.

b) Sheet- A sheet is a mechanism for breaking a large object model down into a series of pages. A sheet is a single printed page. Each module consists of one or more sheets.

Each sheet has a title and a name or number. Each association/generalization appears on a single sheet. Classes may appear on multiple sheets.

11 Sample Model: For the sample model you please look through the text book. It shows an object model of a workstation window management system.

12 Aggregation –I think by now, all of you know what is an aggregation? It is a strong form of association in which an aggregate is made of components. Components are part of the aggregate.

- Aggregation vs. Association –Aggregation is a special form of association not an independent concept. If two objects are bound by a part-whole relationship, it is an aggregation .If two objects are usually considered independent, even though they may be linked, it is an association. Some possible tests are:

i) Would you use the phrase part of?

ii) Are some operations on the whole automatically applied to its parts?

iii) Are some attribute values propagated from the whole to all or some parts

iv) Is there an intrinsic asymmetry to the association, where one object class is subordinate to the other?

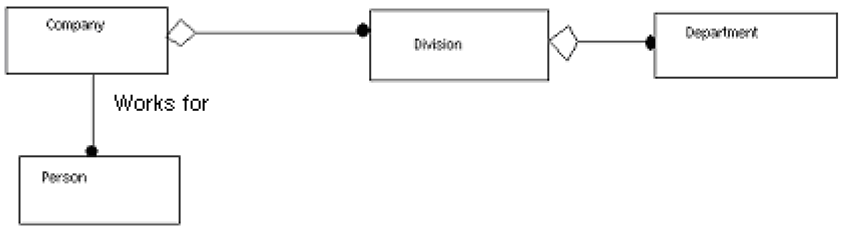


Fig.4.1 Aggregation and association

In the figure, a company is an aggregation of its divisions, which in turns aggregations of departments; a company is indirectly an aggregation of departments.

The decision to use aggregation is a matter of judgment and is often arbitrary.

- *Aggregation vs. Generalization-* Aggregation is not the same as generalization. Aggregation relates instances .Two distinct objects are involved; one of them part of other. While generalization relates classes and is a way of structuring the description of a single object. With generalization, an object is simultaneously an instance of the superclass and an instance of the subclass.

Aggregation is called “a part of” relationship; generalization is often called “a kind of” or “is-a “relationship.

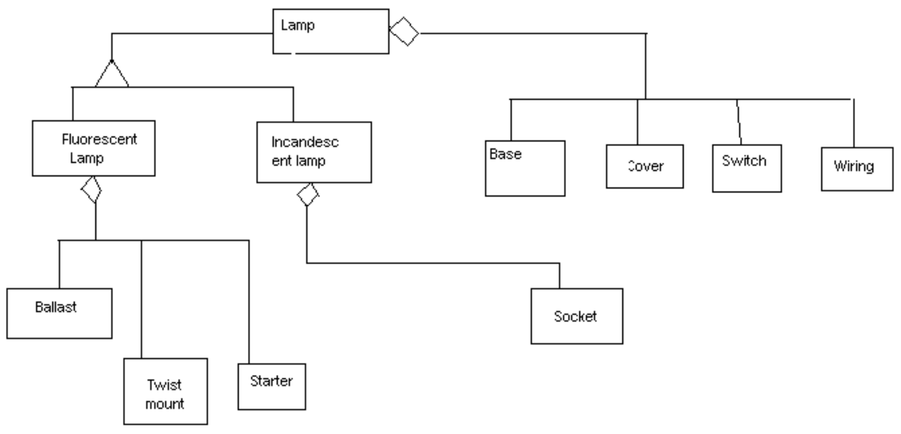


Fig.4.2. Aggregation and generalization

The above figure shows the aggregation and generalization in the case of a desk lamp.

Aggregation is sometimes called an “and-relationship” and generalization an “or-relationship”.

- *Recursive Aggregates* Aggregation can be fixed, variable or recursive.

A fixed aggregate has a fixed structure; the number and types of subparts are predefined. The desk lamp in figure 4.2 is a fixed aggregate.

A variable aggregate has a finite number of levels, but the number of parts may Vary.The Company in Fig.4.1 is a variable aggregate.

A recursive aggregate contains, directly or indirectly, an instance of the same kind of aggregate; the number of potential levels is unlimited.Fig4.3 shows a recursive aggregate.

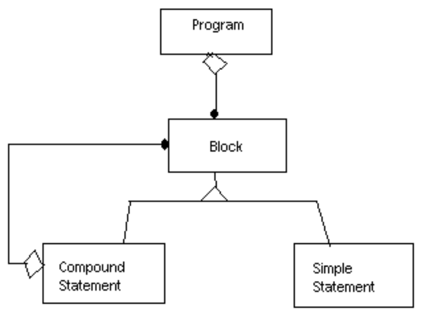


Fig.4.3. Recursive Aggregates

- Propagation of Operations-Propagation (also called triggering) is the automatic application of an operation to a network of objects when the operation is applied to some starting object.

  For example, moving an aggregate moves its parts; the move operation propagates to the parts. The below figure shows propagation.

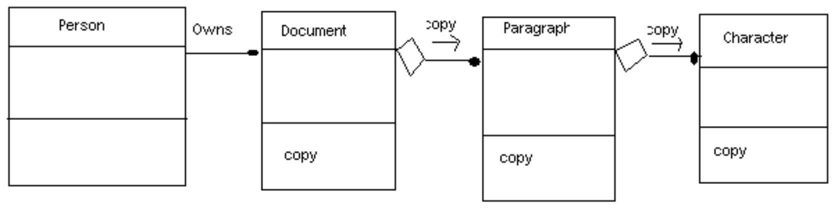


Fig.4.4 Propagation of operations

Here, let us say a person owns multiple documents .Each document is composed of paragraphs that are in turn composed of characters. The copy operation propagates from documents to paragraphs to characters. It means that by copying a paragraph, all the characters in it are copied. It does not propagate in the reverse direction. The propagation is indicated with a small arrow and operation name next to the affected association.

13 Abstract Classes- An abstract class is a class that has no direct instances but whose descendent classes have direct instances.

A concrete class is a class that is an instantiable; that is, it can have direct instances. A concrete class may be leaf classes in the inheritance tree; only concrete classes may be leaf classes in the inheritance tree. Now to get a clear idea about look at the below figure.

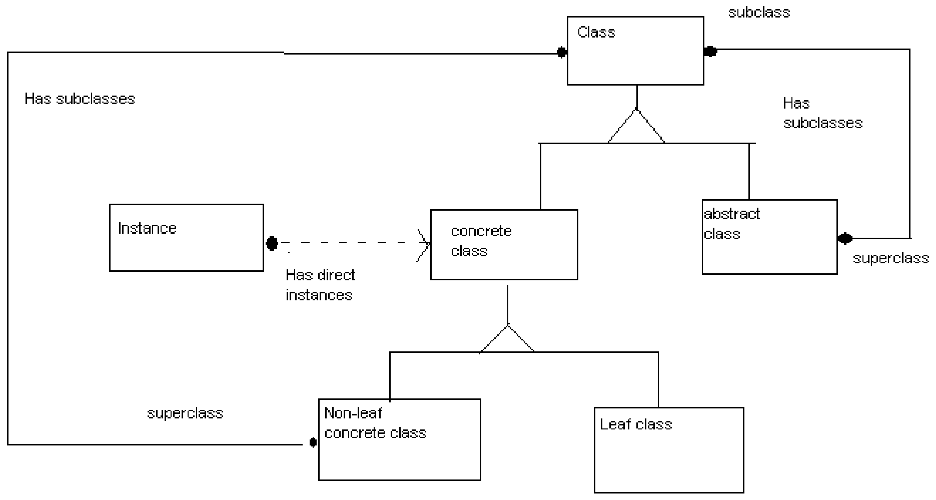


Fig.4.5 Object model defining abstract and concrete class

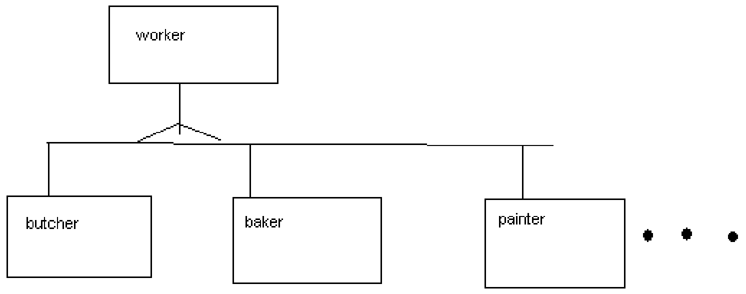
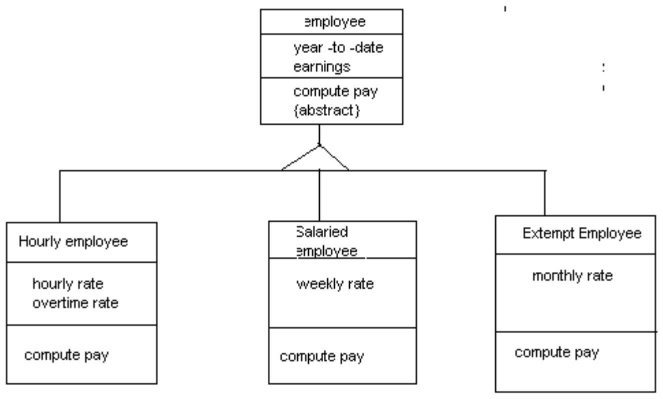


Fig 4.6 Concrete classes

All the classes are concrete classes.



14 <u>Generalization as Extension and Restriction</u>: By now you are all familiar with what is meant by generalization.

In generalization, an instance of a class is an instance of a class is an instance of all ancestors of the class. Therefore you can say that all ancestor class features must apply to the subclass instances. This includes not only the attributes on the ancestor classes but also the operations on the ancestor class.

A subclass may include many features, which is called as an *extension.* For example, fig.4.7 extends class Employee with three subclasses that inherit all Employee features and add new features of their own.

A subclass may also constrain ancestor attributes. This is called *restriction* because it restricts the values that instances can assume. For example, a circle is an ellipse whose major and minor axes are equal.Arbitary changes to the attribute values of a restricted subclass may cause it to violate the constraints, such that the result no longer belongs to the original subclass. This is not a problem from the perspective of the superclass because the result is still a valid superclass instance. For example, a circle that is scaled unequally in the x and y dimensions remains an ellipse but is no longer a circle.

* The inherited features can be renamed in a restriction

For example, the inherited features can be renamed the diameter.

Class membership can be defined in two ways: implicitly by rule or explicitly by enumeration.

Overriding Operations:

Overriding is done for many reasons:

- *Overriding for Extension:* For example, suppose say you have a Window superclass and a Window Labeled subclass. Now this superclass has draw operations that is used to draw the Window boundary and contents. The subclass has an operation draw-Labeled Window. The method could be invoked by invoking the method to draw a window and then adding a code to draw the label.
- *Overriding for Restriction:* This is quite necessary when you have to keep the inherited operations closed within the subclass. For example, the suprclass Set may have the operation add(object).the subclass IntegerSet would then have the more restrictive operation add(integer).
- *Overriding for Optimization:*:
- *Overriding for Convinience:*

15 <u>Multiple Inheritance</u>: It permits a class to have more than one superclass and to inherit features from all parents. It rathers allows mixing of information from two or mores sources. Now what is the advantage of multiple inheritance?

Well it provides increased opportunity for reuse. And to speak about its disadvantage, is a loss of conceptual and implementation simplicity.

Definition:

A class may inherit features from more than one class. A class with more than one superclass is called a join class. A feature from the same ancestor class found along more than one path is inherited only once; it is the same feature. Conflicts among parallel definitions create ambiguities that must be resolved in implementation.

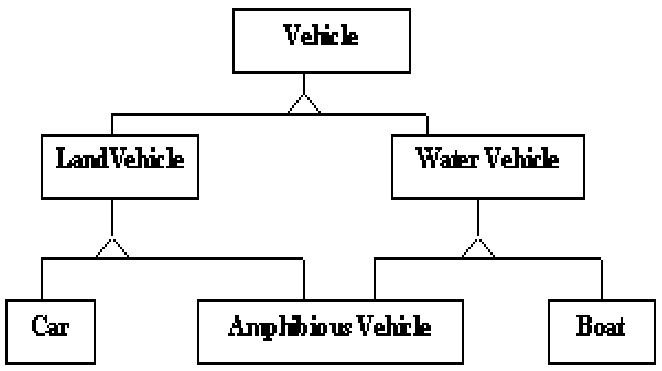


Fig. Multiple inheritance from disjoint classes

Here in the above example, Amphibious Vehicle is a join class. If a class can be refined on several distinct and independent dimensions, then use multiple generalizations.

Accidental Multiple Inheritance

An instance of a join class is inherently an instance of all the ancestors of the join class. For example, an instructor is inherently both faculty and student. But suppose lets say about a professor at HV taking classes at MIT.You can see there is no class to describe the combination .This is a case of accidental multiple inheritance.

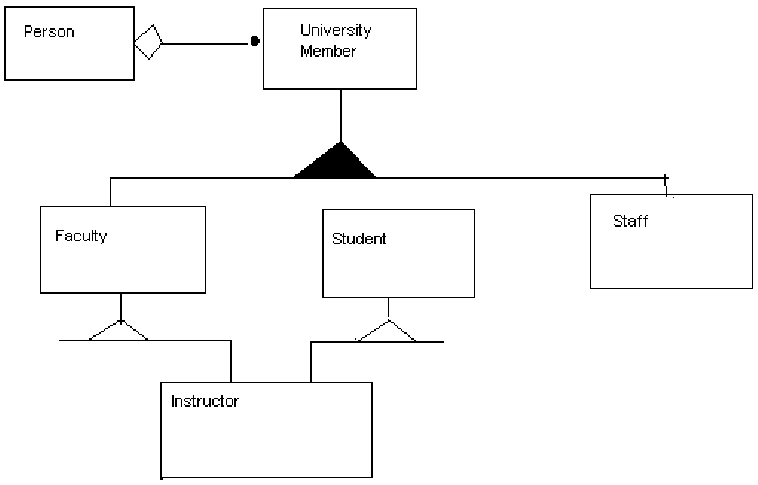


Fig. Workaround for accidental multiple inheritance

16 Metadata-is data that describes other data. For example, the definition of a class is metadata. Models are inherently metadata, since they describe the things being modeled.

17 Candidate Keys – It is a minimal set of attributes that uniquely identifies an object or link. It means you can't discard an attribute from the candidate key and still distinguish all objects and links. A class or association may have one or more candidate keys, each of which may have different combinations and numbers of attributes. The object id is always a candidate key for a class. One or more combinations of related objects are candidate keys for associations.

- A candidate key is delimited with braces in an object model.

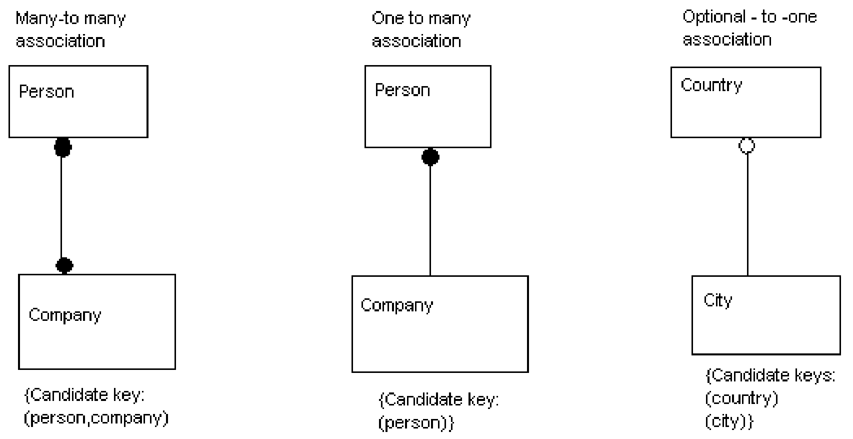

Many-to many association
Person
Company
{Candidate key: (person,company)
One to many association
Person
Company
{Candidate key: (person)}
Optional - to -one association
Country
City
{Candidate keys: (country) (city)}

Fig. Comparison of multiplicity with candidate keys for binary associations

18 Constraints: They are functional relationships between entities of an object model.

Module 2

Dynamic modeling: Events and states – Operations – Nested state diagrams – Concurrency – Advanced dynamic modeling concepts – A sample dynamic model – Relationship of Object and Dynamic models.
Functional modeling: Functional models – Data Flow Diagrams - Specifying operations – Constraints – A sample functional model – Relation of functional to Object and Dynamic models.

What is dynamic modeling?

Temporal relationships are difficult to understand .A system can be best understood by first examining its staitic behavior. Then next we examine the changes to the objects and their relationships over time. Those aspects of a system that are concerned with time and changes are the dynamic model."

Control is that aspect of a system that describes the sequences of operations that occur in response to external stimuli without consideration of what the operations do , what they operate on, or how they are implemented.

The major dynamic modeling concepts are:

- events, states
- state diagrams

<u>Events and States</u>:

"The attribute values and links held by an object are called its state. Over time, the objects stimulate each other, resulting in a series of changes to their states."**"** An individual stimulus from one object to another is an event." The response of an event depends on the state of the object receiving it, and can include a change of state or the sending of another event to the original sender or to a third object.

"The pattern of events, states and state transitions for a given class can be abstracted and represented as a state diagram."

The dynamic model consists of multiple state diagrams, one state diagram for each class with important dynamic behavior, and shows the pattern of activity for an entire system. The state diagrams for the various classes combine into a single dynamic model via shared events.

1) Events:

“An event is something that happens at appoint in time.”

Eg:-Flight 123 departs from Chicago.

An event has no duration. One event may logically precede or follow another, or the two events may be unrelated.

Eg:-Flight 123 must depart Chicago before it can arrive in San Francisco; the two events are casually related.

“Two events that are casually unrelated are said to be concurrent; they have no effect on each other.” Concurrent events can occur in any order.

An event is a one way transmission of information from one object to another. An object sending an event to another object may expect a reply, but the reply is a separate event under the control of the second object, which may or may not choose to send it.

Every event is a unique occurrence but we group them into event classes and give each event class a name to indicate common structure and behavior.

Eg:-Flight 123 departs from Chicago.

Flight 456 departs from Rome.

Both are instances of the event class airplane flight departs.

Some events are signals, but most event classes have attributes indicating the information they convey.eg:-airplane flight departs has attributes airline, flight number and city. The time at which each event occurs is an implicit attribute of an event.

An event conveys information from one object to another. Some classes of events may be simply signals that something has occurred, while other classes of events convey data values. The data values conveyed by events are its attributes, like data values held by its objects. Attributes are shown in the parentheses after the event class name.

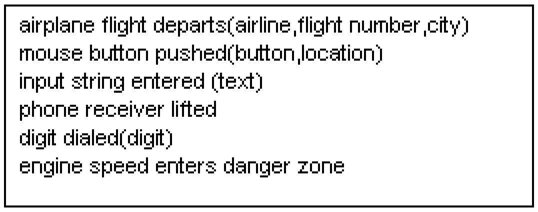

airplane flight departs(airline,flight number,city)
mouse button pushed(button,location)
input string entered (text)
phone receiver lifted
digit dialed(digit)
engine speed enters danger zone

Fig.2.1.1 Event classes and attributes

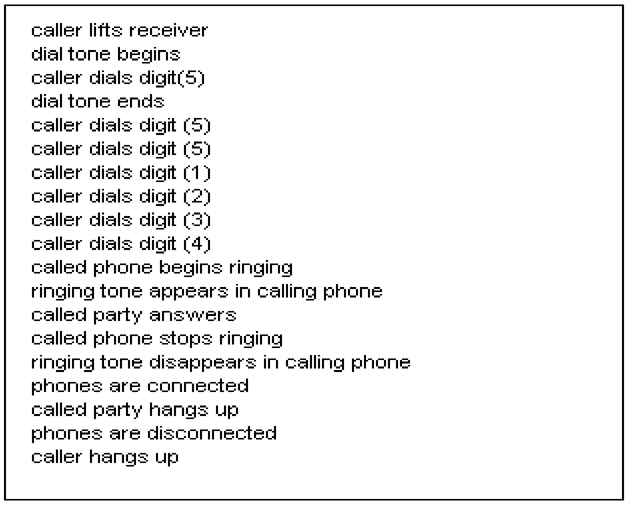
Events include error conditions as well as normal occurrences.

eg:-motor jammed.

2) Scenarios and Event Traces

A scenario is a sequence of events that occurs during one particular execution of system. The scope of a scenario can vary. A scenario can be thought of as a historical record of executing a system or a thought expression of executing a proposed system.

Now let us see the scenario for a telephone system.



caller lifts receiver
dial tone begins
caller dials digit(5)
dial tone ends
caller dials digit (5)
caller dials digit (5)
caller dials digit (1)
caller dials digit (2)
caller dials digit (3)
caller dials digit (4)
called phone begins ringing
ringing tone appears in calling phone
called party answers
called phone stops ringing
ringing tone disappears in calling phone
phones are connected
called party hangs up
phones are disconnected
caller hangs up

Fig.2.1.2 Scenario for phone call

The next step after writing a scenario is to identify the sender and receiver objects of each event.

"The sequence of events and the objects exchanging events can both be shown in an augmented scenario called event trace diagram." The diagram shows each object as a vertical line and each event as a horizontal arrow from the sender object to the receiver object. Time increases from top to bottom.

3) States

A state is an abstraction of the attribute values and links of an object. The set of values are grouped together into a state according to properties that affect the gross behavior of the object.

For example, let us consider the state of a bank. It can either in a solvent or an insolvent state depending on whether it assets exceeds its liabilities.

A state specifies the response of the object to input events. The response to an event may vary quantitatively depending on the exact value of its attributes .The response is the same for values within the same states but different for values in different states.

The response of an object to an event may include an action or a change of state by the object.

Eg:-if a digit is dialed in state dial tone the phone line drops the dial tone and enters dialing.

A state corresponds to the interval between two events received by an object. Events represent points in time; states represent intervals of time.

For example, in the phone example, after the receiver is lifted and before the first digit is dialed, the phone line is in Dial Tone state.

The state of an object depends on the past sequence of events it has received, but in most cases past events are eventually hidden by subsequent events.

For example, Events that has happened before the phone is hung up have no effect on future behavior.

A state has duration. A state is often associated with a continuous activity .eg:-ringing of phone. Events and states are duals i.e. an event separates two states and a state separates 2 events. A state is often associated with the value of an object satisfying some condition. For example, Water is liquid when temperature of water is greater than zero degrees Celsius and below 100 degree Celsius.

Both events and states depend on the level of abstraction used. For example, travel agent planning an itinerary would treat each segment as a single event, but a flight status board will treat it as 2 events departure and arrival.

A state can be characterized in various ways.

For example,

State: alarm ringing

Description:

.

.

.

.

i.e. we specify the name, natural language description and event sequence that lead to it.

4) State diagrams

"A *state diagram* relates events and states ".

When an event is received the next state depends on the current state as well as the event; a change of state caused by an event is called a transition.

Well you can define a *state diagram* as a graph whose nodes are states and whose directed arcs are transitions labeled by event names .We can use rounded boxes to denote states and it contains an optional name .The transition is drawn as an arrow from the receiving state to the target state; the label on the arrow is the name of the event causing transitions .All transitions leaving a state must be correspond to different events. The state diagram specifies the state sequence caused by an event sequence.

If an object is in a state and an event labeling one of its transition occurs the objects enters the state on the target end of the transition. The transition is said to fire.

-> If more than one transition leaves a state then the first event to occur causes the corresponding transition to fire.

->If an event occurs that no transition leaving the current state, then the event is ignored.

->A sequence of events corresponds to a path through the graph.

-> A state diagram describes the behavior of a single class of objects. But each object has its own attribute values so each object has its own state.

States can represent one shot life cycles or continuous loops

"One shot diagrams represent objects with finite lives." The initial state is entered on the creation of the object. Final state is entered on the destruction of the object. The state diagram in the phone example was that of a continuous system.

Initial state is shown by a solid circle while a final state is shown by a solid circle.

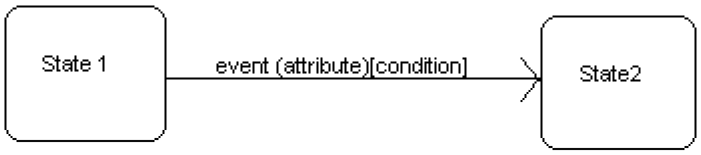
5) Conditions

"A condition is a Boolean function of object values. For example, when we say the temperature was below freezing point from Nov to March.

Here the condition to be temperature below freezing point .It has duration. A state can be defined in terms of a condition; conversely being in a state is a condition. Conditions can be used as guards on transitions "A guarded transition fires when its events occur but only if the condition is true.

For example, let us say a person goes out in the morning (event), if the temperature is below freezing (condition) , then put on your gloves(next state).

A guarded condition on a transition is shown as a Boolean expression in brackets following event name.



Now that we have learnt about states, events, etc. Let us go into the advanced concepts. Let us first discuss on the operations

<u>**Operations**</u>

The state diagrams describe the pattern of events and states for a single object class. How to trigger operations?

i) <u>Controlling Operations</u>

State diagrams could be of little use if they just described the patterns of the events. A behavior description of an object must specify what the object does in response to events .Operations attached to states / transitions are performed in response to corresponding states or events.

"An ***activity*** is an operation that takes time to complete."

* An activity is associated with a state.

* Activities include continuous as well as sequential events.

*A state may control a continuous activity that persists until an event terminates it by calling a transition from the state.

* do: activity within state box indicates that activity starts an entry to the state and stops on exit.

*In the case of sequential events also it is the same thing.

* If an event causes a transition from the state before the activity is complete, it terminates prematurely.

"An ***action*** is an instantaneous operation."

- it is associated with an event
- Actions can also represent internal control operations, such as setting attributes or generating other events.
- Notation ('/' on transition) for action and the name (or description of the action following the name of the event that causes it.

<u>**Nested State Diagrams**</u>

State diagrams can be structured to permit concise descriptions of complex systems. The ways of structuring state machines are similar to the ways of structuring objects. Generalization is equivalent to expanding nested activities.

a) Problems with Flat State Diagrams:

State diagrams lack expressive power and are impractical for large problems. These problems are true of flat, unstructured state diagrams.

b) Nested State Diagrams

An activity in a state can be expanded as a lower level state diagram, each state representing one step of the activity. Nested activities are one shot diagrams with input and output transitions, similar to subroutines.

c) State Generalization

A nested state diagram is actually a form of generalization on states. The states in the nested diagrams are all refinements of the state in the high level diagrams, but in general the states in a nested state diagram may interact with other states. States may have substates that inherit the transitions of superstates, unless overridden.

d) Event Generalization

Events can be organized into generalization hierarchy with inheritance of event attributes.

<u>**(iv) Concurrency**</u>

a) Aggregation Concurrency

A dynamic model describes a set of concurrent objects. A state of an entire system can't be represented by a single state in a single object. It is the product of the states of all objects in it. A state diagram for an assembly is a collection of state diagram, one for each component. By using aggregation we can represent concurrency. Aggregate state corresponds to combined state of all component diagrams.

b) Concurrency within an object

Concurrency within a single composite state of an object is shown by partitioning the composite state into sub diagrams with dotted lines.

(v)Advanced dynamic modeling concepts.

i) Entry and Exit Actions

As an alternative to showing actions on transitions, actions can be associated with entering or exiting a state. Exit actions are less common. What about the order of execution of various actions and events associated with states and events?

Of all the action on the incoming transition will be executed first, then the entry action, followed by the activity within the state, followed by the exit action and finally the action on the outgoing action.

The activities can be interrupted by events causing transitions out of state but entry and exit can't be interrupted. The entry/exit actions are useful in state diagrams .A state can be expressed in terms of matched entry-exit actions.

ii) Internal Actions

An event can cause an action to be performed without causing a state change .The event name is written inside state box followed by '/' . Now to differentiate betw
een a self-transition and an internal action. Self-transition causes an entry and exit action to be executed but an internal action does not.

iii) Automatic Transition

A state performs a sequential activity. When it is over, the transition to another fires. An arrow without an event name indicates an automatic transition that fires when the activity associates with the source state is completed.

If there is no activity, the unlabeled transitions fires as soon as the state is entered. Such transitions are called as '*lambda*' transitions.

iv) Sending Events

An object can perform action of sending an event to another object. A system interacts by events. An event can be directed to a single or a group of objects. Any and all objects can accept it concurrently.

“If a state can accept events from more than one object, the order in which concurrent events are received may affect final state. This is called a race condition. Unwanted race conditions should be avoided.

v) Synchronization of Concurrent Activities

Sometimes an object must perform two or more activities concurrently. Both activities must be completed before the object can progress to its next state. The target state occurs after both events happen in any order.

Functional Models

The functional model specifies the results of a computation without specifying how or when they are computed. That means it specifies the meaning of the operations in the object model and the actions in the dynamic model.

Spreadsheet is a kind of functional model. The purpose of the spreadsheet is to specify values in terms of other values.

Data Flow Diagrams

The functional model consists of multiple data flow diagrams. A Data Flow Diagram (DFD) shows the functional relationships of the values computed by a system, including the input values, output values, and internal data stores. Or we can say that a DFD is a graph showing the flow of data values from their sources in objects through processes that transform them to their destinations in other objects.

A DFD contains **processes** that transform data, **data flows** that move data, **actor** objects that produce and consume data, and **data store** objects that store data passively.

The DFD shows the sequence of transformations performed, as well as the external values and objects that affect the computation.

i) <u>Processes</u>

A process transforms data values. The lowest level processes are pure functions without side effects. An entire data flow graph is a high level process. A process may have side effects if it contains nonfunctional components. A process is drawn as an ellipse.

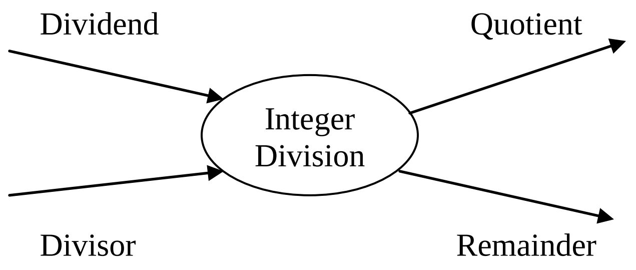


Fig: process

ii) <u>Data Flows</u>

A data flow connects the output of an object or process to the input of another object or process. It represents an intermediate data value within a computation. A data flow is drawn as an arrow. Sometimes an aggregate data value is split into its components, each of which goes to a different process. Flows on the boundary of a data flow diagram are its inputs and outputs.

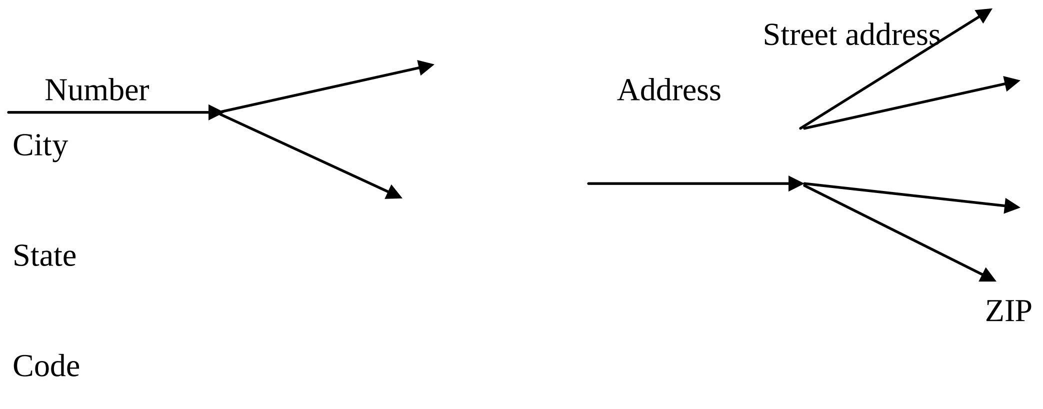


Fig: Data flows to copy a value and split an aggregate value

iii) Actors

An actor is an active object that drives the data flow graph by producing or consuming values. Actors are attached to the inputs and outputs of a data flow graph. The actors are lie on the boundary of the data flow graph. An actor is drawn as a rectangle.

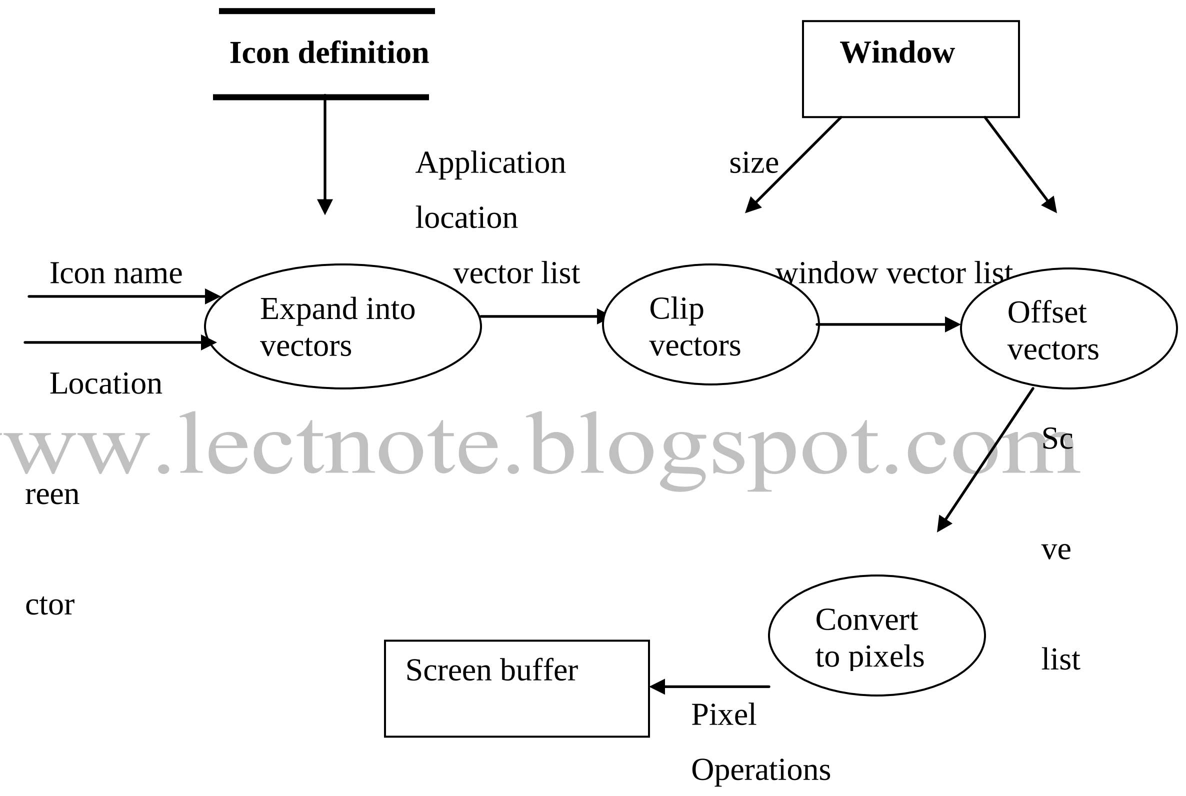


Fig: Data flow diagram for windowed graphics display

iv) Data Stores

A data store is a passive object within a data flow diagram that stores data for later access. A data store is drawn as a pair of parallel lines containing the name of the store. Input arrows indicate information or operations that modify the stored data. Output arrows indicate information retrieved from the store. Both actors and data stores are objects.

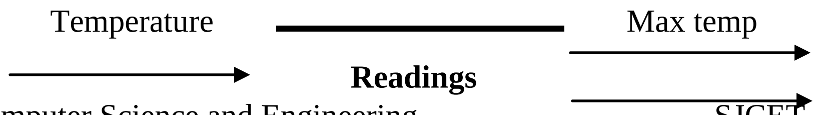


Fig: Data stores

v) Nested Data Flow Diagrams

A process can be expanded into another data flow diagram. Each input and output of the process is an input or output of the new diagram. The new diagram may have data stores. Diagrams can be nested to an arbitrary depth, and the entire set of nested diagrams forms a tree. A diagram that references itself represents a recursive computation.

vi) Control Flows

A control flow is a Boolean value that affects whether a process is evaluated. The control flow is not an input value to the process itself. A control flow is shown by a dotted line from a process producing a Boolean value to the process being controlled.

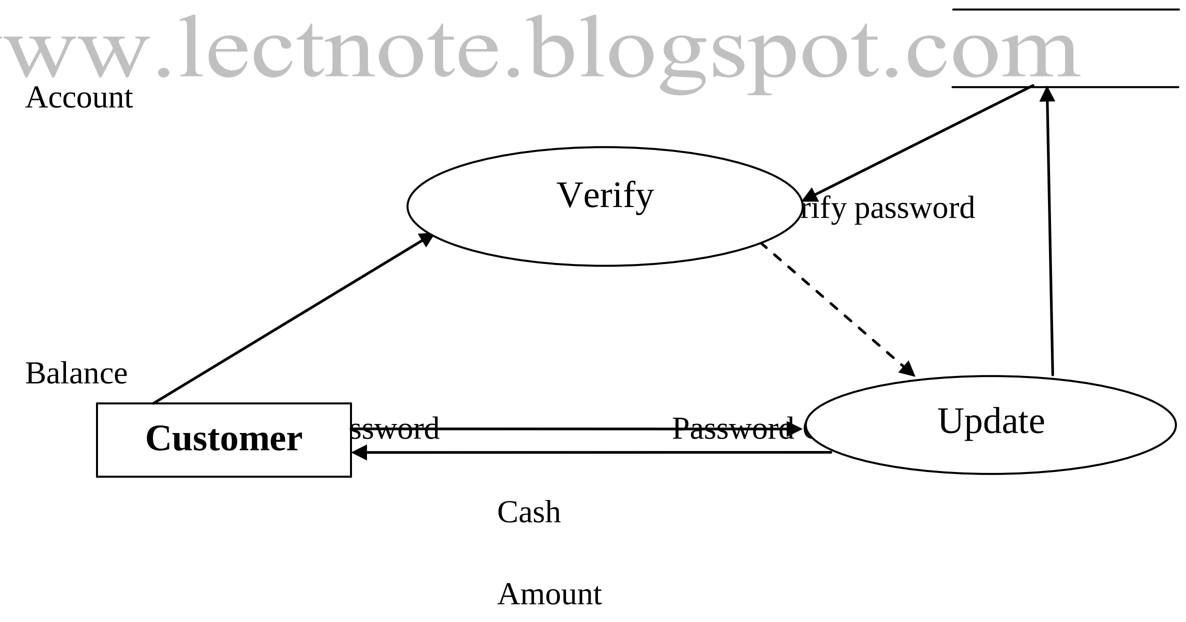


Fig: Control Flow

Specifying Operations

Each operation can be specified in various ways. These are :

- Mathematical functions, such as trigonometric functions
- Table of input and output values for small finite sets
- Equations specifying output in terms of input
- Pre- and post- conditions
- Decision tables
- Pseudocode
- Natural language

Specification of an operation includes a signature and a transformation. The external specification of an operation only describes changes visible outside the operation. Access operations are operations that read or write attributes or links of an object. Nontrivial operations can be divided into three categories: queries, actions, and activities. A query is an operation that has no side effects, it is a pure function. An action is a transformation that has side effects on the target object. An activity is an operation to or by an object that has duration in time

Constraints

A constraint shows the relationship between two objects at the same time or between different values of the same object at different times. A constraint may be expressed as a total function or a partial function. Object constraints specify that some objects depend entirely or partially on other objects. Dynamic constraints specify relationships among the states or events of different objects. Functional constraints specify restrictions on operations.

Relation of Functional to Object and Dynamic Models.

The functional model shows what has to be done by a system. The leaf processes are the operations on objects. The processes in the functional model correspond to operations in the object model. Processes in the functional model show objects that are related by function.

A process is usually implemented as a method.

Module 3

Analysis: Analysis in object modeling, dynamic modeling and functional modeling, Adding operations- Iterating the analysis

System Design: Breaking system into subsystems - Identifying concurrency- allocating subsystems to processors and tasks, managing of data stores. Handling of global resources- handling boundary conditions-Common Architectural Frameworks

Problem statement

The first step to develop anything is to state the requirements. Being vague about your objective only postpones decisions to a later stage where changes are much costly. The problem statement should state what is to be done and not how it is to be done. It should be a statement of needs, not a proposal for a solution. The requestor should indicate which features are mandatory and which are optional; to avoid overly constraining design decisions. The requestor should avoid describing system internals.

A problem statement may have more or less detail. Most problem statements are ambiguous, incomplete or even inconsistent. Some requirements are wrong. Some requirements have unpleasant consequences on the system behavior or impose unreasonable implementation costs. Some requirements seem reasonable first. The problem statement is just a starting point for understanding problem. Subsequent analysis helps in fully understanding the problem and its implication.

Object Modeling

The first step in analyzing the requirements is to construct an object model. It describes real world object classes and their relationships to each other. Information for the object model comes from the problem statement, expert knowledge of the application domain, and general knowledge of the real world. If the designer is not a domain expert, the information must be obtained from the application expert and checked against the model repeatedly. The object model diagrams promote communication between computer professionals and application domain experts.

The steps for object modeling are:

1. ***Identifying Object Classes***

The first step in constructing an object model is to identify relevant object classes from the application domain. The objects include physical entities as well as concepts. All classes must make sense in the application domain; avoid computer implementation constructs such as linked lists. Not all classes are explicit in the problem statement; some are implicit in the application domain or general knowledge.

2.Keeping the Right classes

Now discard the unnecessary and incorrect classes according to the following criteria:

- Redundant Classes - If two classes express the same information, the most descriptive name should be kept.
- Irrelevant Classes - If a class has little or nothing to do with the problem, it should be eliminated .this involves judgment, because in another context the class could be important.
- Vague Classes - A class should be specific .Some tentative classes may have ill-defined boundaries or be too broad in scope.

- Attributes - Names that primarily describe individual objects should be restated as attributes. If independent existence of a property is important, then make it a class and not an attribute.
- Operations - If a name describes an operation that is applied to objects and not manipulated in its own right, then it is not a class.
- Roles - The name of a class should reflect its intrinsic nature and not a role that it plays in an association.
- Implementation Constructs - Constructs extraneous to real world should be eliminated from analysis model. They may be needed later during design, but not now.

3. ***Preparing a data dictionary***

Prepare a data dictionary for all modeling entities .Write a paragraph precisely describing each object class. Describe the scope of the class within the current problem, including any assumptions or restrictions on its membership or use. Data dictionary also describes associations, attributes and operations.

4. Identifying Associations

Identify associations between classes .Any dependency between two or more classes is an association. A reference from one class to another is an association. Associations often correspond to stative verbs or verb phrases. These include physical location (next to, part of), directed actions (drives), communication (talks to), ownership (has, part of) or satisfaction of some kind (works for). Extract all the candidates from the problem statement and get them down on to the paper first, don't try to refine things too early. Don't spend much time trying to distinguish between association and aggregation. Use whichever seems most natural at the time and moves on.

Majority of the associations are taken from verb phrases in the problem statement. For some associations, verb phrases are implicit. Some associations depend on real world knowledge or assumptions. These must be verified with the requestor, as they are not in the problem statement.

5. Keeping the right Associations

Discard unnecessary and incorrect associations using the following criteria:

i. Associations between eliminated classes

If one of the classes in the association has been eliminated, then the association must be eliminated or restated in terms of other classes.

ii. Irrelevant or implementation Associations

Eliminate any association that are outside the problem domain or deal with implementation constructs.

iii. Actions

An association should describe a structural property of the application domain, not a transient event.

iv. Ternary Association

Most associations between three or more classes can be decomposed into binary associations or phrased as qualified associations. If a term ternary association is purely descriptive and has no features of its own, then the term is a link attribute on a binary association.

v. Derived Association

Omit associations that can be defined in terms of other associations because they are redundant. Also omit associations defined by conditions on object attributes.

vi. Misnamed Associations

Don't say how or why a situation came about, say what it is. Names are important to understanding and should be chosen with great care.

vii. Role names

Add role names where appropriate. The role name describes the role that a class in the association plays from the point of view of other class.

viii. Qualified Associations

Usually a name identifies an object within some context; mostly names are not globally unique. The context combines with the name to uniquely identify the object.

A qualifier distinguishes objects on the "many" side of an association.

ix. Multiplicity

Specify multiplicity, but don't put too much effort into getting it right, as multiplicity often changes during analysis.

x. Missing Associations

Add any missing associations that are discovered.

6. Identifying Attributes

Identify object attributes. Attributes are properties of individual objects, such as name, weight, velocity, or color. Attributes shouldn't be objects, use an association to show any relationship between two objects. Attributes usually correspond to nouns followed by possessive phrases, such as the 'color of the car'. Adjectives often represent specific enumerated attribute values, such as red. Unlike classes and associations, attributes are less likely to be fully described in the problem statement. You must draw on your knowledge of the application domain and the real world to find them. Attributes seldom affect the basic structure of the problem. Only consider attributes that directly relate to a particular application. Get the most important attributes first, fine details can be added later. Avoid attributes which are solely for implementation. Give each attribute a meaningful name. Derived attributes are clearly labeled or should be omitted. e.g.:- age.

Link attributes should also be identified. They are sometimes mistaken for object attributes.

7. Keeping the Right Attributes

Eliminate unnecessary and incorrect attributes with the following criteria:

- Objects: If the independent existence of an entity is important, rather than just its value, then it is an object.

  e.g.:- Boss is an object and salary is an attribute.

The distinction often depends on the application .An entity that has features of its own within the given application is an object.

- Qualifiers: If the value of an attribute depends on a particular context, then consider restating the attribute as a qualifier. e.g.:- Employee number is not a unique property of a person with two jobs, it qualifies the association company employs person.
- Nouns: Names are often better modeled as qualifiers rather than attributes.
- Identifiers: Object Oriented Languages incorporate the notion of an object id for unambiguously referencing an object. Do not list these object identifiers in object models, as object identifiers are implicit in object models. One list attributes which exist in application domain.
- Link Attributes: If a property depends on the presence of a link, then the property is an attribute of the link and not of related objects. Link attributes are usually obvious on many to many associations, they can't be attached to the "many" objects without losing information. Link attributes are also subtle on one-one association.
- Internal values: If an attribute describes the internal state that is invisible outside the object, then eliminate it from the analysis.
- Fine detail: Omit minor attributes which are unlikely to affect most operations.
- Discordant attributes: An attribute that seems completely different from and unrelated to all other attributes may indicate a class that should be split into two distinct classes.

8. Refining with Inheritance

The next step to organize classes by using inheritance to share common structure .inheritance can be added in two directions:- by generalizing common aspects of existing classes into a superclass (bottom up),refining existing classes into specialized subclasses (top down).

9. Testing Access Paths

Trace access paths through object model diagram to see if they yield sensible results. Where a unique value is expected is there a path yielding a unique result? For multiplicity is there a way to pick out unique values when needed? Are there useful questions which can't be answered? Thus we will be able to identify if there is some missing information.

10. Iterating Object Model

Object model is rarely correct after a single pass. Continuous iteration is needed. Different parts of a model are iterated often at different stages of completion. If there is any deficiency, go aback to an earlier stage to correct it. Some refinements are needed only after the functional and dynamic models.

11. Grouping Classes into Modules

Group the classes into sheets and modules. Reuse module from previous design.

Dynamic Modeling:

Dynamic Modeling shows the time-dependent behavior of the system and the objects in it. Begin dynamic analysis by looking for events-externally visible stimuli and responses. Summarize permissible event sequences for each object with a state diagram. Dynamic model is insignificant for a purely static data repository. It is important in the case of interactive systems. For most problems, logical correctness depends on the sequences of interactions, not the exact time of interactions. Real time systems do have specific timing requirements on interactions that must be considered during analysis.

The following steps are performed in constructing a dynamic model:

(i) Preparing a Scenario

Prepare one or more typical dialogs between user and system to get a feel for expected system behavior. These scenarios show the major interactions, external display format and information exchanges.

Approach the dynamic model by scenarios, to ensure that important steps are not overlooked and that the overall flow of the interaction is smooth and correct. Sometimes the problem statement describes the full interaction sequence, but most of the time you will have to invent the interaction format.

- The problem statement may specify needed information but leaves open the manner in which it is obtained.
- In many applications gathering information is a major task or sometimes the only major task. The dynamic model is critical in such applications.
- First prepare scenarios for "normal " cases , interactions without any unusual inputs or error conditions
- Then consider "special "cases, such as omitted input sequences, maximum and minimum values, and repeated values.
- Then consider user error cases, including invalid values and failures to respond. For many interactive applications, error handling is the most different part of the implementation. If possible, allow the user to abort an operation or roll back to a well defined starting point at each step.
- Finally consider various other kinds of interactions that can be overlaid on basic interactions, such as help requests and status queries.
- A scenario is a sequence of events .An event occurs when information is exchanged between an object in the system and an outside agent, such as user. The values exchanged are parameters of the event. The events with no parameters are meaningful and even common. The information in such event is the fact that it has occurred a pure signal. Anytime information is input to the system or output from the system.
- For each event identify the actor (system, user or other external agent) that caused the event and the parameters of the event.
- The screen layout or output format generally doesn't affect the logic of the interaction or the values exchanged.

- Don't worry about output formats for the initial dynamic model; describe output formats during refinement of the model.

(ii) Interface Formats

Most interactions can be separated into two parts:

- application logic
- user interface

Analysis should concentrate first on the information flow and control, rather than the presentation format. Same program logic can accept input from command lines, files, mouse buttons, touch panels, physical push buttons, or carefully isolated. Dynamic model captures the control logic of the application. It is hard to evaluate a user interface without actually testing it .Often the interface can be mocked up so that users can try it. Application logic can often be simulated with dummy procedures. Decoupling application logic from user interface allows "look and feel" of the user interface to be evaluated while the application is under development.

(iii) Identifying Events

Examine the scenario to identify all external events. Events include all signals, inputs, decisions, interrupts, transitions, and actions to or from users or external devices. Internal computation steps are not events, except for decision points that interact with the external world .Use scenarios to find normal events, but don't forget error conditions and unusual events. An action by an object that transmits information is an event. Most object to object interaction and operations correspond to events. Some information flows are implicit. Group together under a single name event that have the same effect on flow of control, even if their parameter values differ. You must decide when differences in quantitative values are important enough to distinguish. The distinction depends on the application. You may have to construct the state diagram before you can classify all events; some distinctions between events may have no effect on behavior and can be ignored. Allocate each type of event to the object classes that send it and receive it. The event is an output event for the sender and an input event for the receiver. Sometimes an object sends an event to itself, in which case the event is both an output and input for the same class. Show each scenario as an event trace – an ordered list of events between different objects assigned to columns in a table. If more than one object of the same class participates in the scenario, assign a separate column to each object. By scanning a particular column in the trace, you can see the events that directly affect a particular object. Only these events can appear in the state diagram for the object. Show the events between a group of classes (such as a module) on an event flow diagram. This diagram summarizes events between classes, without regard for sequence. Include events from all scenarios, including error events. The event flow diagram is a dynamic counterpart to an object diagram. Paths in the object diagram show possible information flows; paths in the event flow diagram show possible control flows.

(iv) Building a State Diagram

Prepare a state diagram for each object class with non trivial dynamic behavior , showing the events the object receives and sends .Every scenario or event trace corresponds to path through the state diagram .Each branch in a control flow is represented by a state with more than one exit transition. The hardest thing is deciding at which state an alternative path rejoins the existing diagram. Two paths join at a state if the object “forgets” which the one was taken. In many cases, it is obvious from your knowledge of the application that two states are identical. Beware of two paths that appear identical but which can be distinguished under some circumstances. The judicious use of parameters and conditional transitions can simplify state diagrams considerably but at the cost of mixing together state information and data. State diagrams with too much data dependency can be confusing and counterintuitive. Another alternative is to partition a state diagram into two concurrent subdiagrams, using one subdiagram for the main line and the other for the distinguishing information. After normal events have been considered, add boundary cases and special cases. Handling user errors needs more thought and code than normal case. You are finished with the state diagram of a class when the diagram handles all events that can affect an object of the class in each of its states. If there are complex interactions with independent inputs, we use nested state diagrams. Repeat the above process of building state diagrams for each class of objects. Concentrate on classes with important interactions. Not all classes need state diagrams. Many objects respond to input events independent of their past history, or capture all important history as parameters that do not affect control. Such objects may receive and send events. List the input events for each object and output event sent in response to each input event, but there will be no further state structure. Eventually you may be able to write down state diagram directly without preparing event traces.

(v) Matching Events between Objects.

Check for completeness and consistency at the system level when the state diagram for each class is complete .Every event should have a sender and a receiver, occasionally the same object .States without predecessors or successors are suspicious; make sure they represent starting or termination points of the interaction sequences .Follow the effects of an input event from object to object through the system to make sure that they match the scenario. Objects are inherently concurrent; beware of synchronization errors where an input occurs at an awkward time. Make sure that corresponding events on different state diagrams are consistent. The set of state diagrams for object classes with important dynamic behavior constitute the dynamic model for the application.

Functional Modeling

The functional model shows how values are computed, without regard for sequencing, decisions or object structure. It shows which values depend on which other values and which function relate them. The dfds are useful for showing functional dependencies .Functions are expressed in various ways, including natural language, mathematical equations and pseudo code. Processes on dfd corresponds to activities or actions in the state diagrams of the class. Flows on dfd correspond to objects or attribute values on attribute values in object diagram. Construct the functional model after the two models. The following steps are performed in constructing functional model:

(i) Identifying input and output values

List input and output values. The input and output values are parameters of events between system and outside world. Find any input or output values missed.

(ii) Building DFD

Now construct a DFD, show each output value is completed from input values. A DFD is usually constructed in layers. The top layer may consist of a single process, or perhaps one process to gather inputs, compute values, and generate outputs. Within each dfd layer, work backward from each output value to determine the function that computes it. If the inputs are operations are all inputs of the entire diagram, you are done; otherwise some of the operation inputs are intermediate values that must be traced backward in turn. You can also trace forward from inputs to outputs, but it is usually harder to identify all users of an input, than to identify all the sources of an output.

- Expand each non trivial process in the top level DFD into a lower level DFD.
- Most systems contain internal storage objects that retain values between iterations. An internal store can be distinguished from a data flow or a process because it receives values that don't result in immediate outputs but are used at some distant time in the future.
- The DFD show only dependencies among operations. They don't show decisions.(control flows)

iii) Describing Functions

When the dfd has been refined enough, write a description of each function. The description can be in natural language, mathematical equations, pseudocode, decision tables, or some appropriate form.

- Focus on what the function does, and not how it is implemented.
- The description can be declarative or procedural.

- A declarative description specifies the relationships between input and output values and the relationships among output values.

For example, "sort and remove duplicate values"

Declarative: "Every value in the input list appears exactly once in the output list, and the values in the output list are in strictly increasing order."

Procedural:-specifies function by giving an algorithm to compute it. The purpose of the algorithm is only to specify what the function does.

A declarative description is preferable; they don't imply implementation but if procedural is easy it should be used.

iv) Identify constraints between objects

Identify constraints between objects. Constraints are functional dependencies between objects that are not related by an input output dependency. Constraints can be on two objects at the same time, between instances of the same object at different times (an invariant) or between instances of different object at different times.

-> Preconditions on functions are constraints that the input values must satisfy.

-> Post conditions are constraints that the output values are guaranteed to hold.

-> State the times or conditions under which the constraints hold.

v) Specifying Optimization criteria

Specify values to be maximized, minimized or optimized. If there is several optimization criteria that conflict, indicate how the trade-off is to be decided.

System Design

System design is the high-level strategy for solving the problem and building a solution. The steps involved in system design are:

<u>1. estimating system performance</u>

Prepare a rough performance estimate. The purpose is not to achieve high accuracy, but merely to determine if the system is feasible.

<u>2. making a reuse plan</u>

There are 2 different aspects of reuse – using existing things and creating reusable new things. It is much easier to reuse existing things than to design new things. Reusable things include models, libraries, patterns, frameworks etc.

<u>Breaking a system into subsystems</u>

First step of system design is to divide the system in a small number of subsystems. Each subsystem encompasses aspects of the system that share some common property such as similar functionality, same physical location, execution on same kind of hardware.

- A subsystem is a package of classes, associations, operations, events and constraints interrelated and have a well defined small interface with other subsystems. Subsystems are identified by the services provided. Determining subsystems allow a division of labor so each can be designed by a separate team.

- Most interactions should be within subsystems rather than across boundaries and it reduces dependencies among subsystems.

- Relationship between two subsystems can be client-supplier or peer-to-peer. In a client-supplier relationship, the client calls on the supplier for service. Client must know the interface of the supplier but not vice versa. In a peer-to-peer relationship, each can call on the other and therefore they must know each others interfaces. One way communication (client-supplier) is much easier to build, understand and change than two way (Peer-to-Peer).

- The decomposition of systems into subsystems may be organized as a sequence of layers or partitions.

- Layered system is an ordered set of virtual worlds, a layer knows about the layer below but not above. A client supplier relationship exists between lower layers and upper layers.

- Partitions divide into several independent SS, each providing a type of service. A system can be a hybrid of layered and partitioned

<u>**Choosing the system topology**</u>

System Topology defines the flow of information among subsystems. The flow may be a pipeline (compiler) or star which has a master that controls all other interactions. Try to use simple topologies when possible to reduce the number of interactions between subsystems.

- In analysis as in the real world and in hardware all objects are concurrent. In implementation not all software objects are concurrent because one processor may support many objects. An important goal of system design is to identify which object must be active concurrently and which objects have activity that is mutually exclusive (not active at the same time). The latter objects can be folded into a single thread of control or task (runs on a single processor).

Identifying Concurrency

- Dynamic Modeling is the guide to concurrency - Two objects are said to be inherently concurrent if they can receive events at the same time without interacting. If the events are unsynchronized, the objects cannot be folded into a single thread of control. (engine and wing control on an airplane must operate concurrently)

- Two subsystems that are inherently concurrent need not necessarily be implemented as separate hardware units. The purpose of interrupts, Operating System and tasking mechanism is to simulate logical concurrency in a uniprocessor - so if there are no timing constraints on the response then a multitasking Operating System can handle the computation.

- All objects are conceptually concurrent, but many objects are interdependent. By examining the state diagram of individual objects and the exchange of events between them, many objects can be folded into a single thread of control. A thread of control is a path through a set of state diagrams on which only a single object at a time is active - thread remains in a state diagram until an object sends an event to another object and waits for another event.

- On each thread of control only a single object is active at a time. Threads of control are implemented as tasks in computer systems.

Estimating Hardware Resource Requirements

- The decision to use multiple processors or hardware functional units is based on the need for higher performance than a single CPU can provide (or fault tolerant requests). Numbers of processors determine the volume of computations and speed of the machine.

- The system designer must estimate the required CPU power by computing the steady state load as the product of the number of transactions per second and the time required to process a transaction. Experimentation is often useful to assist in estimating.

Hardware/Software Tradeoffs

Hardware can be viewed as a rigid but highly optimized form of software. The system designer must decide which subsystems will be implemented in hardware and which in software.

Subsystems are in hardware for 2 reasons:

- Existing hardware provides exactly the functionality required – e.g. floating point processor
- Higher performance is required than a general purpose CPU can provide.

Compatibility, cost, performance, flexibility are considerations whether to use hardware or software. Software in many cases may be more costly than hardware because the cost of people is very high. Hardware of course is much more difficult to change than software.

Allocating Tasks To Processors

Tasks are assigned to processors because:

- Certain tasks are required at specific physical locations
- Response time or information flow rate exceeds the available communication bandwidth between a task and a piece of hardware.
- Computation rates are too great for single processor, so tasks must be spread among several processors.

8. Management Of Data Stores

- The internal and external data stores in a system provide clean separation points between subsystems with well defined interfaces. Data stores may combine data structures, files, and database implemented in memory or on secondary storage devices. The different data stores provide trade-offs between cost, access time, capacity and reliability.

- Files are cheap, simple and permanent form of data store. File operations are low level and applications must include additional code to provide a suitable level of abstraction. File implementations vary from machine to machine so portability is more difficult.

- Databases are powerful and make applications easier to port. They do have a complex interface and may work awkwardly with programming languages.

The kind of data that belongs in a Database:

- Data that requires access at fine levels of detail by multiple users
- Data that can be efficiently managed by DBMS commands
- Data that must be ported across many hardware and operating system platforms
- Data that must be accessible by more than one application program.

The kind of data that belongs in a file:

- Data that is voluminous but difficult to structure within the confines of a Database (graphics bit map)
- Data that is voluminous and of low information density (debugging data, historical files)
- Raw data that is summarized in the Database
- Volatile data

Advantages Of Using A Database

- Handles crash recovery, concurrency, integrity
- Common interface for all applications
- Provides a standard access language - SQL

Disadvantages Of Using A Database

- Performance overhead
- Insufficient functionality for advanced applications - most databases are structured for business applications.
- Awkward interface with programming languages.

Handling Global Resources

- The system designer must identify the global resources and determine mechanisms for controlling access to them - processors, tape drives, etc

- If the resource is a physical object, it can control itself by establishing a protocol for obtaining access within a concurrent system. If the resource is logical entity (object ID for example) there is a danger of conflicting access. Independent tasks could simultaneously use the same object ID. Each global resource must be owned by a guardian object that controls access to it.

Choosing Software Control Implementation

There are two kinds of control flows in a software system: external control and internal control. Internal control is the flow of control within a process. It exists only in the implementation and therefore is not inherently concurrent or sequential. External control is the flow of externally visible events among the objects in the system.

There are three kinds of external control:

- Procedure driven systems: control resides within program code; it is easy to implement but requires that concurrency inherent in objects be mapped into a sequential flow of control.

- Event driven systems - control resides within a dispatcher or monitor provided by the language, subsystem or operating system. Application procedures are attached to events and are called by the dispatcher when the corresponding events occur.

- Concurrent systems: control resides concurrently in several independent objects, each a separate task.

11. Handling Boundary Conditions

We must be able to handle boundary conditions such as initialization, termination and failure.

1. Initialization: The system must be brought from a quiescent initial state to a sustainable steady state condition. Things to be initialized include constant data, parameters, global variables, tasks, guardian objects and possibly the class hierarchy itself.

2. Termination: It is usually simpler than initialization because many internal objects can simply be abandoned. The task must release any external resources that it had reserved. In a concurrent system, one task must notify other tasks of its termination.

3. Failure: It is the unplanned termination of a system. Failure can arise from user errors, from the exhaustion of system resources, or from an external breakdown.

Setting Trade-Off Priorities

Cost, maintainability, portability, understandability, speed, may be traded off during various parts of system design.

Common Architectural Frameworks

There are several prototypical architectural frameworks that are common in existing systems. Each of these is well-suited to a certain kind of system. If we have an application with similar characteristics, we can save effort by using the corresponding architecture, or at least use it as a starting point for our design. The kinds of systems include:

1. Batch transformation: It is a data transformation executed once on an entire input set. A batch transformation is a sequential input-to-output transformation, in which inputs are supplied at the start, and the goal is to compute an answer. There is no ongoing interaction with the outside world. The most important aspect of the batch transformation is the functional model which specifies how input values are transformed into output values. The steps in designing a batch transformation are:

1) Break the overall transformation into stages, each stage performing one part of the transformation. The system diagram is a data flow diagram. This can usually be taken from the functional model.
2) Define intermediate object classes for the data flows between each pair of successive stages.
3) Expand each stage in turn until the operations are straightforward to implement.
4) Restructure the final pipeline for optimization.

2. Continuous Transformation: It is a system in which the outputs actively depend on changing inputs and must be periodically updated. Because of severe time constraints, the entire set of outputs cannot usually be recomputed each time an input changes. Instead, the new output values must be computed incrementally. The transformation can be implemented as a pipeline of functions. The effect of each incremental change in an input value must be propagated through the pipeline. The steps in designing a pipeline for continuous transformation are:

1. Draw a DFD for the system. The input and output actors correspond to data structures whose value change continuously. Data stores within the pipeline show parameters that affect the input-to-output mapping.
2. Define intermediate objects between each pair of successive stages.
3. Differentiate each operation to obtain incremental changes to each stage.
4. Add additional intermediate objects for optimization.

3. Interactive Interface: An interactive interface is a system that is dominated by interactions between the system and external agents, such as humans, devices or other programs. The external agents are independent of the system, so their inputs cannot be controlled. Interactive interfaces are dominated by the dynamic model. The steps in designing an interactive interface are:

1. Isolate the objects that form the interface from the objects that define the semantics of the application.
2. Use predefined objects to interact with external agents, if possible.
3. Use the dynamic model as the structure of the program.
4. Isolate physical events from logical events.
5. Fully specify the application functions that are invoked by the interface. Make sure that the information to implement them is present.

4. Dynamic Simulation: A dynamic simulation models or tracks objects in the real world. The steps in designing a dynamic simulation are:

1. Identify actors, real-world objects from the object model. The actors have attributes that are periodically updated. The actors have attributes that are periodically updated.
2. Identify discrete events. Discrete events correspond to discrete interactions with the object. Discrete events can be implemented as operations on the object.
3. Identify continuous dependencies.
4. Generally a simulation is driven by a timing loop at a fine time scale. Discrete events between objects can often be exchanged as part of the timing loop.

5. Transaction Manager: A transaction manager is a database system whose main function is to store and access information. The information comes from the application domain. Most transaction managers must deal with multiple users and concurrency. The steps in designing a transaction management system are:

1. Map the object model directly into a database.
2. Determine the units of concurrency. Introduce new classes as needed.
3. Determine the unit of transaction.
4. Design concurrency control for transactions.

Module 4

Object Design: Overview of Object design – Combining the three models – Designing algorithms – Design optimization – Implementation of control – Adjustment of inheritance - Design of association – Object representation – Physical packaging – Documenting design decisions-Comparison of methodologies

Object Design

The object design phase determines the full definitions of the classes and associations used in the implementation, as well as the interfaces and algorithms of the methods used to implement operations. The object design phase adds internal objects for implementation and optimizes data structures and algorithms.

Overview of Object Design

During object design, the designer carries out the strategy chosen during the system design and fleshes out the details. There is a shift in emphasis from application domain concepts toward computer concepts. The objects discovered during analysis serve as the skeleton of the design, but the object designer must choose among different ways to implement them with an eye toward minimizing execution time, memory and other measures of cost. The operations identified during the analysis must be expressed as algorithms, with complex operations decomposed into simpler internal operations. The classes, attributes and associations from analysis must be implemented as specific data structures. New object classes must be introduced to store intermediate results during program execution and to avoid the need for recomputation. Optimization of the design should not be carried to excess, as ease of implementation, maintainability, and extensibility are also important concerns.

Steps of Design:

During object design, the designer must perform the following steps:

1. Combining the three models to obtain operations on classes.
2. Design algorithms to implement operations.
3. Optimize access paths to data.
4. Implement control for external interactions
5. Adjust class structure to increase inheritance.
6. Design associations.
7. Determine object representation.
8. Package classes and associations into modules.

a) <u>Combining the three models to obtain operations on classes.</u>

After analysis, we have object, dynamic and functional model, but the object model is the main framework around which the design is constructed. The object model from analysis may not show operations. The designer must convert the actions and activities of the dynamic model and the processes of the functional model into operations attached to classes in the object model. Each state diagram describes the life history of an object. A transition is a change of state of the object and maps into an operation on the object. We can associate an operation with each event received by an object. In the state diagram, the action performed by a transition depends on both the event and the state of the object. Therefore, the algorithm implementing an operation depends on the state of the object. If the same event can be received by more than one state of an object, then the code implementing the algorithm must contain a case statement dependent on the state. An event sent by an object may represent an operation on another object .Events often occur in pairs , with the first event triggering an action and the second event returning the result on indicating the completion of the action. In this case, the event pair can be mapped into an operation performing the action and returning the control provided that the events are on a single thread. An action or activity initiated by a transition in a state diagram may expand into an entire dfd in the functional model .The network of processes within the dfd represents the body of an operation. The flows in the diagram are intermediate values in operation. The designer convert the graphic structure of the diagram into linear sequence of steps in the algorithm .The process in the dfd represent suboperations. Some of them, but not necessarily all may be operations on the original target object or on other objects. Determine the target object of a suboperation as follows:

* If a process extracts a value from input flow then input flow is the target.

* Process has input flow or output flow of the same type, input output flow is the target.

* Process constructs output value from several input flows, then the operation is a class operation on output class.

* If a process has input or an output to data store or actor, data store or actor is the target.

b) <u>Designing algorithms</u>

Each operation specified in the functional model must be formulated as an algorithm. The analysis specification tells what the operation does from the view point of its clients, but the algorithm shows how it is done. The analysis specification tells what the operation does from the view point of its clients, but the algorithm shows how it is done. An algorithm may be subdivided into calls on simpler operations, and so on recursively, until the lowest-level operations are simple enough to implement directly without refinement .The algorithm designer must decide on the following:

i) Choosing algorithms

Many operations are simple enough that the specification in the functional model already constitutes a satisfactory algorithm because the description of what is done also shows how it is done. Many operations simply traverse paths in the object link network or retrieve or change attributes or links.

Non trivial algorithm is needed for two reasons:

a) To implement functions for which no procedural specification

b) To optimize functions for which a simple but inefficient algorithm serves as a definition.

Some functions are specified as declarative constraints without any procedural definition. In such cases, you must use your knowledge of the situation to invent an algorithm. The essence of most geometry problems is the discovery of appropriate algorithms and the proof that they are correct. Most functions have simple mathematical or procedural definitions. Often the simple definition is also the best algorithm for computing the function or else is also so close to any other algorithm that any loss in efficiency is the worth the gain in clarity. In other cases, the simple definition of an operation would be hopelessly inefficient and must be implemented with a more efficient algorithm.

For example, let us consider the algorithm for search operation .A search can be done in two ways like binary search (which performs log n comparisons on an average) and a linear search (which performs n/2 comparisons on an average).Suppose our search algorithm is implemented using linear search , which needs more comparisons. It would be better to implement the search with a much efficient algorithm like binary search.

Considerations in choosing among alternative algorithm include:

a) Computational Complexity:

It is essential to think about complexity i.e. how the execution time (memory) grows with the number of input values.

For example: For a bubble sort algorithm, time ∞ n^2

Most other algorithms, time ∞ n log n

b) Ease of implementation and understandability:

It is worth giving up some performance on non critical operations if they can be implemented quickly with a simple algorithm.

c) Flexibility:

Most programs will be extended sooner or later. A highly optimized algorithm often sacrifices readability and ease of change. One possibility is to provide two implementations of critical applications, a simple but inefficient algorithm that can be implemented, quickly and used to validate the system, and a complicated but efficient algorithm whose correct implementation can be checked against the simple one.

d) Fine Timing the Object Model:

We have to think, whether there would be any alternatives, if the object model were structured differently.

ii) Choosing Data Structures

Choosing algorithms involves choosing the data structures they work on. We must choose the form of data structures that will permit efficient algorithms. The data structures do not add information to the analysis model, but they organize it in a form convenient for the algorithms that use it.

iii) Defining Internal Classes and Operations

During the expansion of algorithms, new classes of objects may be needed to hold intermediate results. New, low level operations may be invented during the decomposition of high level operations. A complex operation can be defined in terms of lower level operations on simpler objects. These lower level operations must be defined during object design because most of them are not externally visible. Some of these operations were found from "shopping –list". There is a need to add new internal operations as we expand high –level functions. When you reach this point during the design phase, you may have to add new classes that were not mentioned directly in the client's description of the problem. These low-level classes are the implementation elements out of which the application classes are built.

iv) Assigning Responsibility for Operations

Many operations have obvious target objects, but some operations can be performed at several places in an algorithm, by one of the several places, as long as they eventually get done. Such operations are often part of a complex high-level operation with many consequences. Assigning responsibility for such operations can be frustrating, and they are easy to overlook in laying out object classes because they are easy to overlook in laying out object classes because they are not an inherent part of one class. When a class is meaningful in the real world, then the operations on it are usually clear. During implementation, internal classes are introduced.

How do you decide what class owns an operation?

When only one object is involved in the operation, tell the object to perform the operation. When more than one object is involved, the designer must decide which object plays the lead role in the operation. For that, ask the following questions:

- Is one object acted on while the other object performs the action? It is best to associate the operation with the target of the operation, rather than the initiator.
- Is one object modified by the operation, while other objects are only queried for the information they contain? The object that is changed is the target.
- Looking at the classes and associations that are involved in the operation, which class is the most centrally-located in this subnetwork of the object model? If the classes and associations form a star about a single central class, it is the target of the operation.
- If the objects were not software, but the real world objects represented internally, what real world objects would you push, move, activate or manipulate to initiate operation?

Assigning an operation within a generalization hierarchy can be difficult. Since the definitions of the subclasses within the hierarchy are often fluid and can be adjusted during design as convenient. It is common to move an operation up and down in the hierarchy during design, as its scope is adjusted.

Design Optimization

The basic deign model uses the analysis model as the framework for implementation .

The analysis model captures the logical information about the system, while the design model must add details to support efficient information access. The inefficient but semantically correct analysis model can be optimized to make the implementation more efficient, but an optimized system is more obscure and less likely to be reusable in another context. The designer must strike an appropriate balance between efficiency and clarity. During design optimization, the designer must

i) Add Redundant Associations for Efficient Access

During analysis, it is undesirable to have redundancy in association network because redundant associations do not add any information. During design, however we evaluate the structure of the object model for an implementation. For that, we have to answer the following questions:

* Is there a specific arrangement of the network that would optimize critical aspects of the completed system?

* Should the network be restructured by adding new associations?

* Can existing associations be omitted?

The associations that were useful during analysis may not form the most efficient network when the access patterns and relative frequencies of different kinds of access are considered. In cases where the number of hits from a query is low because only a fraction of objects satisfy the test, we can build an index to improve access to objects that must be frequently retrieved.

Analyze the use of paths in the association network as follows:

- Examine each operation and see what associations it must traverse to obtain its information. Note which associations are traversed in both directions, and which are traversed in a single direction only, the latter can be implemented efficiently with one way pointers.

  For each operation note the following items:
- How often is the operation called? How costly is to perform?
- What is the "fan-out" along a path through the network? Estimate the average count of each "many" association encountered along the path. Multiply the individual fan-outs to obtain the fan-out of the entire path; which represents the number of accesses on the last class in the path. Note that "one" links do not increase the fan-out, although they increase the cost of each operation slightly, don't worry about such small effects.

- What is the fraction of "hits" on the final class , that is , objects that meets selection criteria (if any) and is operated on? If most objects are rejected during the traversal for some reason, then a simple nested loop may be inefficient at finding target objects. Provide indexes for frequent, costly operations with a low hit ratio because such operations are inefficient to implement using nested loops to traverse a path in the network.

ii)Rearranging Execution Order for Efficiency

After adjusting the structure of the object model to optimize frequent traversal, the next thing to optimize is the algorithm itself. Algorithms and data structures are directly related to each other, but we find that usually the data structure should be considered first. One key to algorithm optimization is to eliminate dead paths as early as possible. Sometimes the execution order of a loop must be inverted.

iii) Saving Derived Attributes to Avoid Recomputation:

Data that is redundant because it can be derived from other data can be "cached" or store in its computed form to avoid the overhead of recomputing it. The class that contains the cached data must be updated if any of the objects that it depends on are changed.

Derived attributes must be updated when base values change. There are 3 ways to recognize when an update is needed:

- Explicit update: Each attribute is defined in terms of one or more fundamental base objects. The designer determines which derived attributes are affected by each change to a fundamental attribute and inserts code into the update operation on the base object to explicitly update the derived attributes that depend on it.
- Periodic Recomputation: Base values are updated in bunches. Recompute all derived attributes periodically without recomputing derived attributes after each base value is changed. Recomputation of all derived attributes can be more efficient than incremental update because some derived attributes may depend on several base attributes and might be updated more than once by incremental approach. Periodic recomputation is

simpler than explicit update and less prone to bugs. On the other hand, if the data set changes incrementally a few objects at a time, periodic recomputation is not practical because too many derived attributes must be recomputed when only a few are affected.

- Active values: An active value is a value that has dependent values. Each dependent value registers itself with the active value, which contains a set of dependent values and update operations. An operation to update the base value triggers updates all dependent values, but the calling code need not explicitly invoke the updates. It provides modularity.

<u>Implementation of Control</u>

The designer must refine the strategy for implementing the state – event models present in the dynamic model. As part of system design, you will have chosen a basic strategy for realizing dynamic model, during object design flesh out this strategy. There are three basic approaches to implementing the dynamic model:

i) *State as Location within a Program:*

This is the traditional approach to representing control within a program. The location of control within a program implicitly defines the program state. Any finite state machine can be implemented as a program. Each state transition corresponds to an input statement. After input is read, the program branches depending on the input event received. Each input statement need to handle any input value that could be received at that point. In highly nested procedural code, low –level procedures must accept inputs that they may know nothing about and pass them up through many levels of procedure calls until some procedure is prepared to handle them.

One technique of converting state diagram to code is as follows:

1. Identify the main control path. Beginning with the initial state, identify a path through the diagram that corresponds to the normally expected sequence of events. Write the name of states along this path as a linear sequence of events. Write the names of states along this path as a linear sequence .This becomes a sequence of statements in the program.

2. Identify alternate paths that branch off the main path and rejoin it later. These become conditional statements in the program.
3. Identify backward paths that branch off the main loop and rejoin it earlier .These become loops in program. If multiple backward paths that do not cross, they become nested loops. Backward paths that cross do not nest and can be implemented with goto if all else fails, but these are rare.
4. The status and transitions that remain correspond to exception conditions. They can be handled using error subroutines , exception handling supported by the language , or setting and testing of status flags. In the case of exception handling, use goto statements.

ii) State machine engine

The most direct approach to control is to have some way of explicitly representing and executing state machine. For example, state machine engine class helps execute state machine represented by a table of transitions and actions provided by the application. Each object instance would contain its own independent state variables but would call on the state engine to determine next state and action. This approach allows you to quickly progress from analysis model to skeleton prototype of the system by defining classes from object model state machine and from dynamic model and creating “stubs” of action routines. A stub is a minimal definition of function /subroutine without any internal code. Thus if each stub prints out its name , technique allows you to execute skeleton application to verify that basic flow of control is correct. This technique is not so difficult.

iii) Control as Concurrent Tasks

An object can be implemented as task in programming language /operating system. It preserves inherent concurrency of real objects. Events are implemented as inter task calls using facilities of language/operating system. Concurrent C++/Concurrent Pascal support concurrency. Major Object Oriented languages do not support concurrency.

Adjustment of Inheritance

The definitions of classes and operations can often be adjusted to increase the amount of inheritance.

The designer should:

i) Rearrange classes and operations

Sometimes the same operation is defined across several classes and can easily be inherited from a common ancestor, but more often operations in different classes are similar but not identical. By slightly modifying the definitions of the operations or the classes , the operations can often be made to match so that they can be covered by a single inherited operation. Before inheritance can be used , each operation must have the same interface and the types of arguments and results. If the signatures match, then the operations must be examined to see if they have the same semantics. The following kinds of adjustments can be used to increase the chance of inheritance.

- Some operations may have fewer arguments than others .The missing arguments can be added but ignored.
- Some operations may have few arguments because they are special cases of more general arguments .Implement the special operations by calling the general operation with appropriate parameter values.
- Similar attributes in different classes may have different names. Give the attributes the same name and move them to a common ancestor class. These operations that access the attributes will match better.
- Any operation may be defined on several different classes in a group but undefined on the other classes. Define it on the common ancestor class and define it as no operation on the values that do not care about it.

ii) Abstracting Out Common Behavior

Reexamine the object model looking for commonality between classes. New classes and operations are often added during design. If a set of operations / attributes seems to be repeated in two classes , it is possible that the two classes are specialized variations of the sane thing. When common behavior has been recognized , a common superclass can be created that implements the shared features , specialized features in subclass. This transformation of the object model is called abstracting out a common superclass / common behavior .usually the superclass is abstract meaning no direct instances. Sometimes a superclass is abstracted even when there is only one subclass; here there is no need of sharing. Superclass may be reusable in future projects. It is an addition to the class library. When a project is completed , the reusable classes should be collected, documented and generalized so that they may be used in future projects.. Another advantage of abstract superclasses other than sharing and reuse is modularity. Abstract superclasses improve the extensibility of a software product. It helps in the configuration management of software maintenance and distribution.

iii) Use Delegation to Share Implementation

Sometimes programmers use inheritance as an implementation technique with no intention of guaranteeing the same behavior. Sometimes an existing class implements some of the behavior that we want to provide in a newly defined class, although in other respects the two classes are different. The designer may inherit from the existing class to achieve part of the implementation of the new class. This can lead to problems –unwanted behavior.

Design of Associations

During object design phase, we must formulate a strategy for implementing all associations in the object model. We can either choose a global strategy for implementing all associations uniformly , or a particular technique for each association.

i) Analyzing Association Traversal

Associations are inherently bidirectional. If association in your application is traversed in one direction, their implementation can be simplified. The requirements on your application may change, you may need to add a new operation later that needs to traverse the association in reverse direction. For prototype work, use bidirectional association so that we can add new behavior and expand /modify. In the case of optimization work, optimize some associations.

ii) One-way association

* If an association is only traversed in one direction it may be implemented as pointer.

* If multiplicity is 'many' then it is implemented as a set of pointers.

* If the "many" is ordered , use list instead of set .

* A qualified association with multiplicity one is implemented as a dictionary object(A dictionary is a set of value pairs that maps selector values into target values.

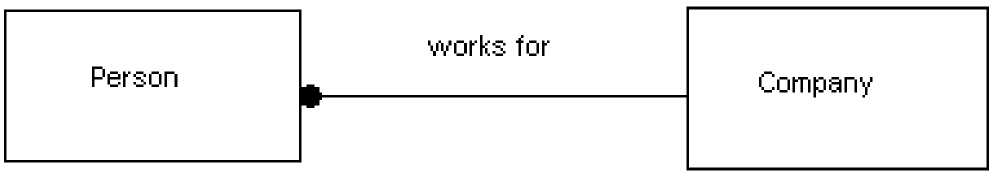
* Qualified association with multiplicity "many" are rare.(it is implemented as dictionary set of objects).

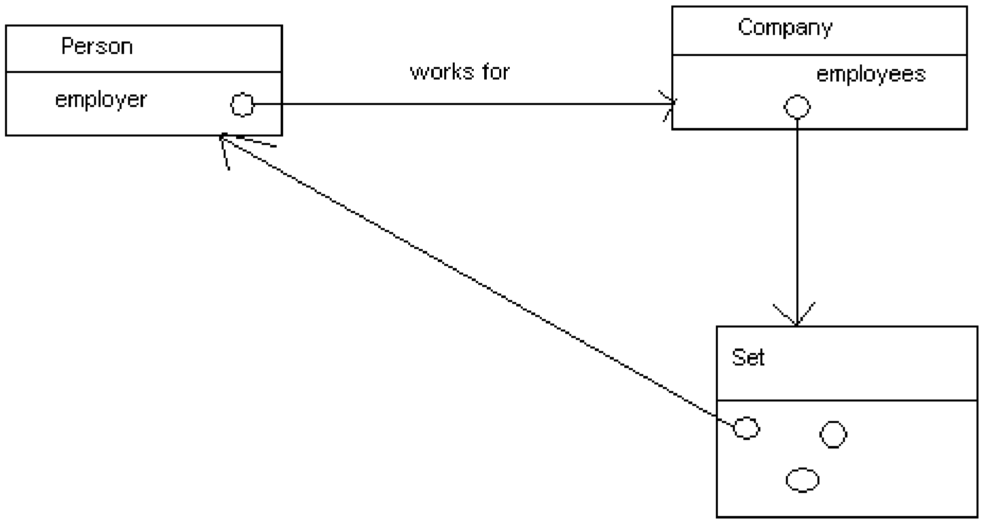
iii) Two-way associations

Many associations are traversed in both directions, although not usually with equal frequency. There are three approaches to their implementation:

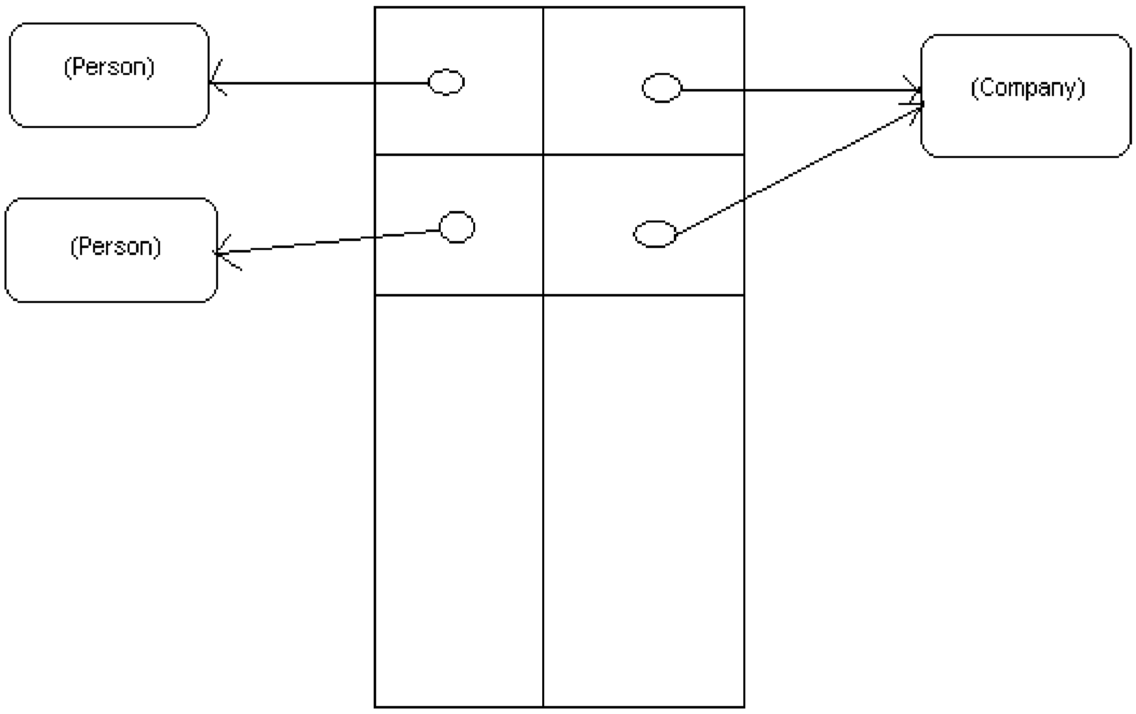
- Implement as an attribute in one direction only and perform a search when a backward traversal is required. This approach is useful only if there is great disparity in traversal frequency and minimizing both the storage cost and update cost are important.

- Implement as attributes in both directions. It permits fast access, but if either attribute is updated then the other attribute must also be updated to keep the link consistent .This approach is useful if accesses outnumber updates.





- Implement as a distinct association object independent of either class. An association object is a set of pairs of associated objects stored in a single variable size object. An association object can be implemented using two dictionary object one for forward direction and other for reverse direction.



iv) Link Attributes

Its implementation depends on multiplicity .

- If it is a one-one association , link attribute is stored in any one of the classes involved.
- If it is a many-one association, the link attribute can be stored as attributes of many object ,since each " many object appears only once in the association.
- If it is a many-many association, the link attribute can't be associated with either object; implement association as distinct class where each instance is one link and its attributes.

Object Representation

Implementing objects is mostly straight forward, but the designer must choose when to use primitive types in representing objects and when to combine groups of related objects. Classes can be defined in terms of other classes, but eventually everything must be implemented in terms of built-in-primitive data types, such as integer strings, and enumerated types. For example, consider the implementation of a social security

number within an employee object. It can be implemented as an attribute or a separate class.
Defining a new class is more flexible but often introduces unnecessary indirection. In a similar vein, the designer must often choose whether to combine groups of related objects.

Physical Packaging

Programs are made of discrete physical units that can be edited, compiled, imported, or otherwise manipulated. In C and Fortran the units are source files; In Ada, it is packages. In object oriented languages, there are various degrees of packaging. In any large project, careful partitioning of an implementation into packages is important to permit different persons to cooperatively work on a program.

Packaging involves the following issues:

i) Hiding internal information from outside view.

One design goal is to treat classes as 'black boxes' , whose external interface is public but whose internal details are hidden from view. Hiding internal information permits implementation of a class to be changed without requiring any clients of the class to modify code. Additions and changes to the class are surrounded by "fire walls" that limit the effects of any change so that changes can be understood clearly. Trade off between information hiding and optimization activities. During analysis , we are concerned with information hiding. During design , the public interface of each class must be defined carefully. The designer must decide which attributes should be accessible from outside the class. These decisions should be recorded in the object model by adding the annotation {private} after attributes that are to be hidden , or by separating the list of attributes into 2 parts. Taken to an extreme a method on a class could traverse all the associations of the object model to locate and access another object in the system .This is appropriate during analysis , but methods that know too much about the entire model are fragile because any change in representation invalidates them. During design we try to limit the scope of any one method. We need top define the bounds of visibility that each method requires.

Specifying what other classes a method can see defines the dependencies between classes. Each operation should have a limited knowledge of the entire model, including the structure of classes, associations and operations. The fewer things that an operation knows about, the less likely it will be affected by any changes. The fewer operations know about details of a class, the easier the class can be changed if needed.

The following design principles help to limit the scope of knowledge of any operation:

- Allocate to each class the responsibility of performing operations and providing information that pertains to it.
- Call an operation to access attributes belonging to an object of another class
- Avoid traversing associations that are not connected to the current class.
- Define interfaces at as high a level of abstraction as possible.
- Hide external objects at the system boundary by defining abstract interface classes, that is, classes that mediate between the system and the raw external objects.
- Avoid applying a method to the result of another method, unless the result class is already a supplier of methods to the caller. Instead consider writing a method to combine the two operations.

ii) Coherence of entities.

One important design principle is coherence of entities. An entity ,such as a class , an operation , or a module , is coherent if it is organized on a consistent plan and all its parts fit together toward a common goal. It shouldn't be a collection of unrelated parts. A method should do one thing well .a single method should not contain both policy and implementation.

"A policy is the making of context dependent decisions."

"Implementation is the execution of fully specified algorithms."

Policy involves making decisions, gathering global information, interacting with outside world and interpreting special cases. Policy methods contain input output statements, conditionals and accesses data stores. It doesn't contain complicated algorithms but instead calls various implementation methods. An implementation method does exactly one operation without making any decisions, assumptions, defaults or deviations .All information is supplied as arguments(list is long). Separating policy and implementation increase reusability. Therefore implementation methods don't contain any context dependency. So they are likely to be reusable Policy method need to be rewritten in an application , they are simple and consists of high level decisions and calls on low-level methods. A class shouldn't serve too many purposes.

iii) Constructing physical modules.

During analysis and system design phases we partitioned the object model into modules.

* The initial organization may not be suitable for final packaging of system implementation

new classes added to existing module or layer or separate module.

- Modules should be defined so that interfaces are minimal and well defined.
- Connectivity of object model can be used as a guide for partitioning modules. Classes that are closely connected by associations should be in the same module. Loosely connected classes should be grouped in separate modules.
- Classes in a module should represent similar kinds of things in the application or should be components of the same composite object.
- Try to encapsulate strong coupling within a module. Coupling is measured by number of different operations that traverse a given association. The number expresses the number of different ways the association is used, not the frequency.

Documenting Design Decisions

The above design decisions must be documented when they are made, or you will become confused. This is especially true if you are working with other developers. It is impossible to remember design details for any non trivial software system, and documentation is the best way of transmitting the design to others and recording it for reference during maintenance.

The design document is an extension of the Requirements Analysis Document.

-> The design document includes revised and much more detailed description of the object model-both graphical and textual. Additional notation is appropriate for showing implementation decisions, such as arrows showing the traversal direction of associations and pointers from attributes to other objects.

-> Functional model will also be extended. It specifies all operation interfaces by giving their arguments, results, input-output mappings and side effects.

-> Dynamic model – if it is implemented using explicit state control or concurrent tasks then the analysis model or its extension is adequate. If it is implemented by location within program code, then structured pseudocode for algorithms is needed.

Keep the design document different from analysis document .The design document includes many optimizations and implementation artifacts. It helps in validation of software and for reference during maintenance. Traceability from an element in analysis to element in design document should be straightforward. Therefore the design document is an evolution of analysis model and retains same names.

Module 5

Other Models: Booch's Methodology- Notations, models, concepts. Jacobson Methodology- architecture, actors and use-cases, requirement model, Analysis Model, Design model, Implementation model and Test Model-Unified Modeling Language (UML).

The Booch Method - The Booch Method is a well-documented, proven method for object-oriented analysis and design.

What makes the Booch Method different?

The Booch method notation differs from other OO methodologies because it centers on the development of *four* fundamental models of the system in the problem domain that will be implemented primarily in software. These models are:

- The Logical Model
- The Physical Model
- The Static Model
- The Dynamic Model

These models document *components:* classes, class categories, objects, operations, subsystems, modules, processes, processors, devices and the relationships between them. Just as an architect has multiple views of a skyscraper under construction, so the software engineer has multiple view of the software system undergoing design.

Views of the Booch Method models

The Booch method recognizes that an engineer can view the four models from several different perspectives. In the Booch method, an engineer can document the models with several different diagrams:

- The Class Diagram
- The Object Diagram
- The Module Diagram
- The State Diagram

- The Interaction Diagram
- The Process Diagram

The engineer augments these diagrams with specifications that textually complement the icons in the diagrams. It's important to realize that there is *not* a one-to-one mapping between a model and a particular view. Some diagrams and/or specifications can contain information from several models, and some models can include information that ends up in several diagrams. Each model component and their relationships can appear in none, one, or several of a models diagram.

i) Class Diagram

A class diagram shows the existence of classes and their relationships in the logical view of the system. This is the diagram that most people think of when they think of the Booch notation, because the 'clouds' that represent classes are so unique.

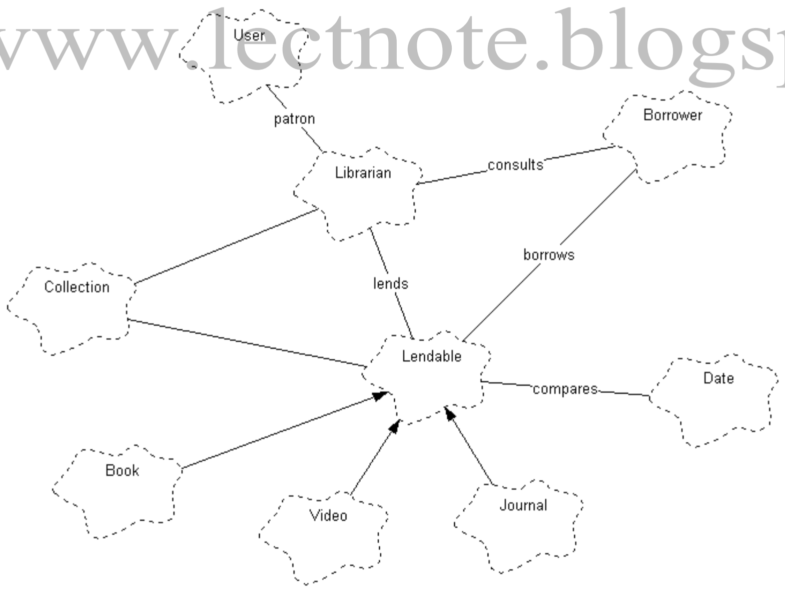


Figure: Initial Class Diagram for the Library Application

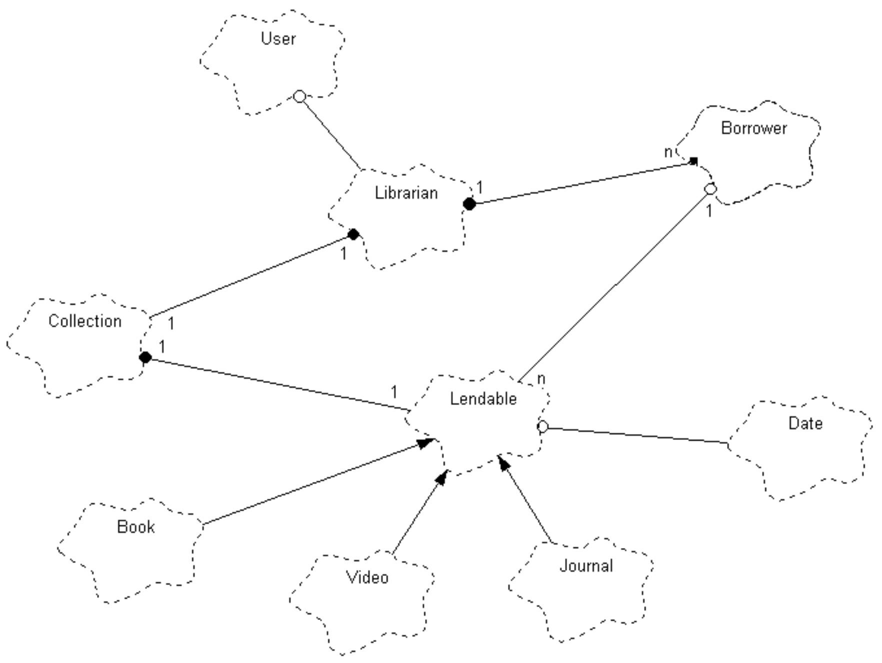


Figure: Second pass Class Diagram for the Library Application

In the above figure, the relationship with a dot on the end indicated a ``has'' relationship. The class on the ``dot'' side *owns* or uses the facilities of the class on the other end of the line.

ii) The Object Diagram

An object diagram, or for our purposes here an *object-scenario diagram* shows the existence of objects, their relationships in the logical view of the system, and how they execute a particular scenario or use-case.

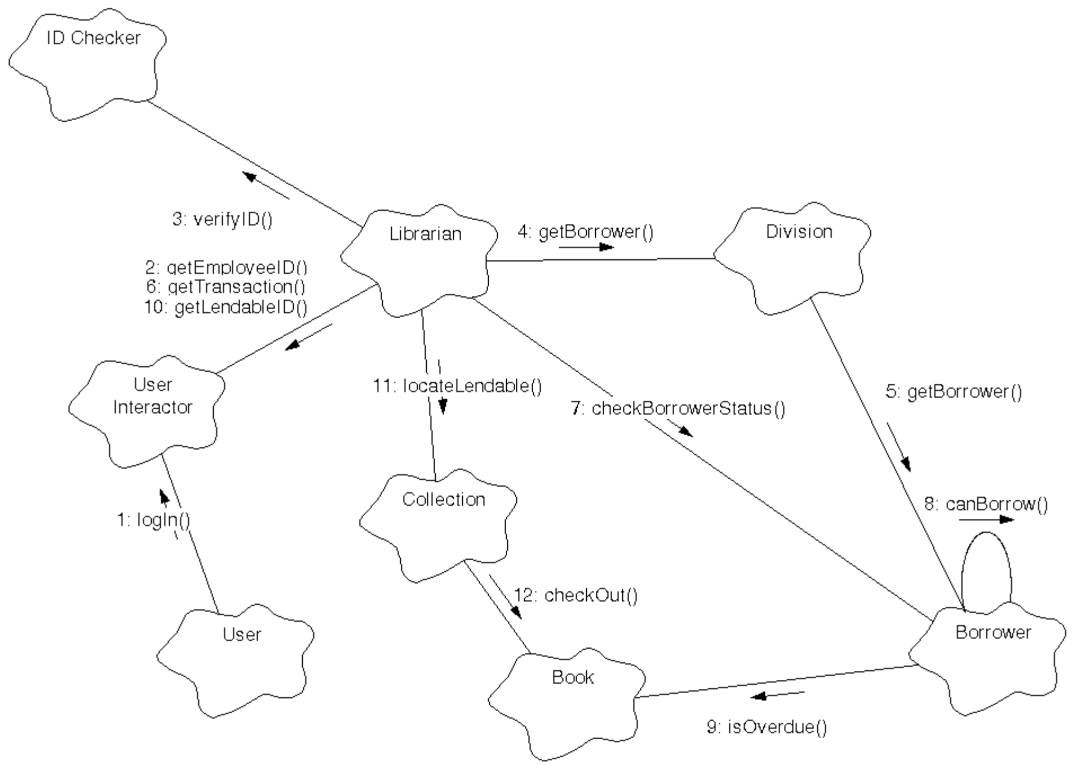


Figure: Object Diagram for checking out a book

iii) The Module Diagram

A module diagram shows the allocation of classes and objects to modules in the physical view of a system.

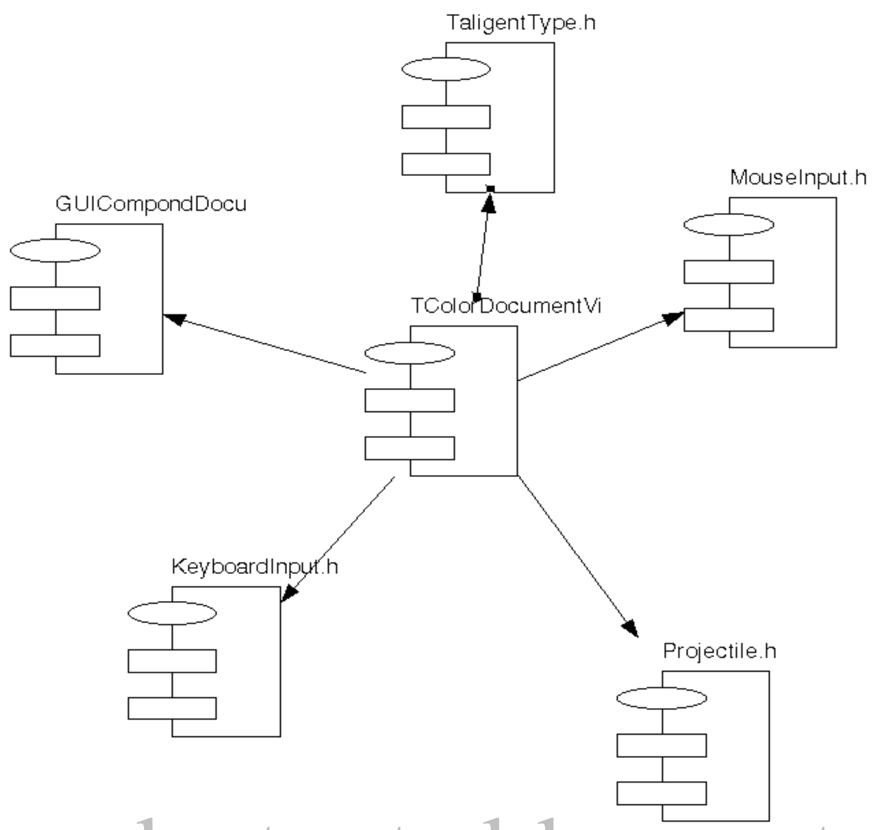


Figure: A Module Diagram for the ColorDocumentView class header file

iv) The State Transition Diagram

The *State Transition diagram* shows the state space of a given context, the events that cause a transition from one state to another, and the actions that result.

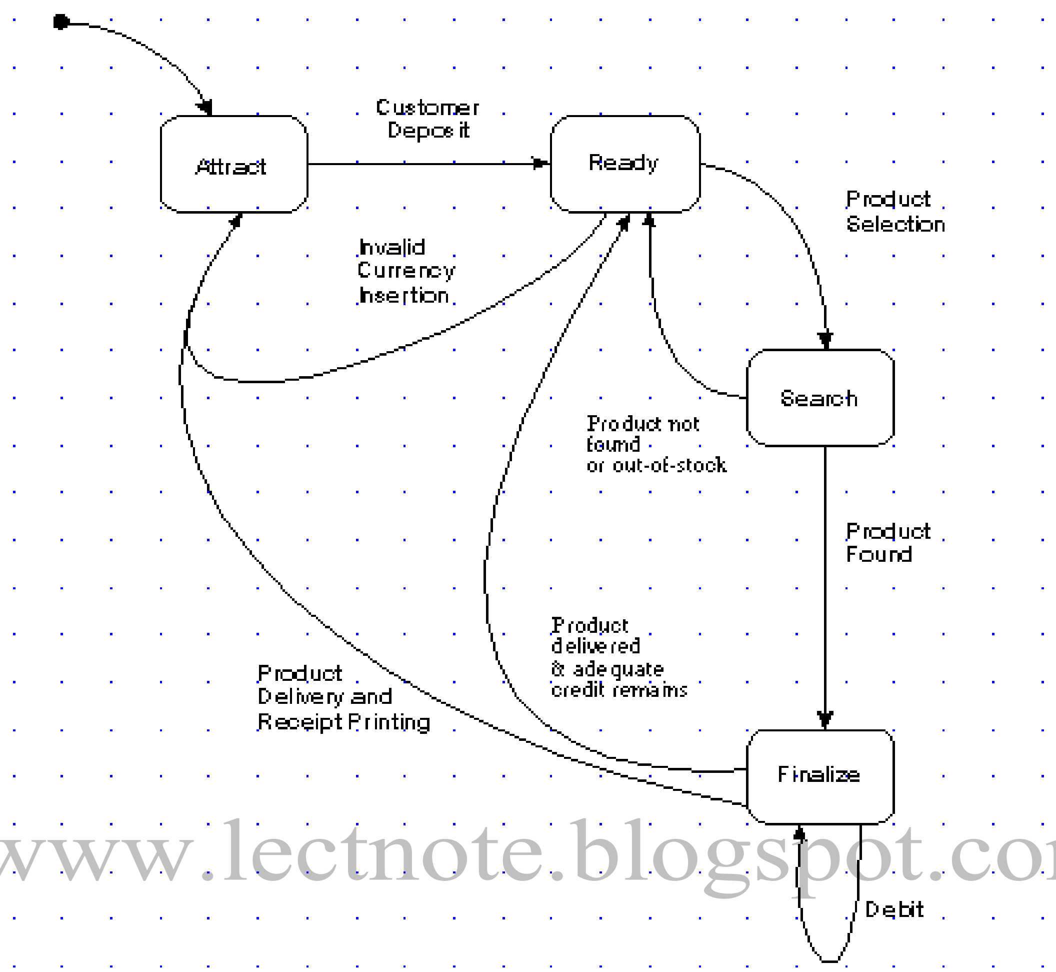

Attract
Customer Deposit
Ready
Product Selection
Invalid Currency Insertion
Search
Product not found or out-of-stock
Product Found
Product delivered & adequate credit remains
Product Delivery and Receipt Printing
Finalize
Debit

Figure: State Transition Diagram for the Postal Machine CPU

v) Interaction Diagrams

The interaction diagrams illustrate how objects interact via messages. They are used for dynamic object modeling.

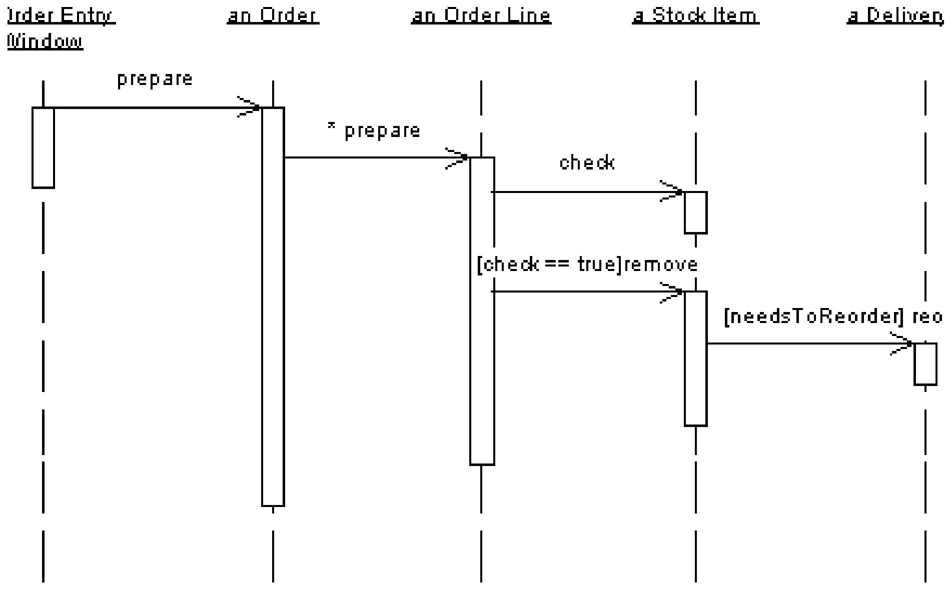
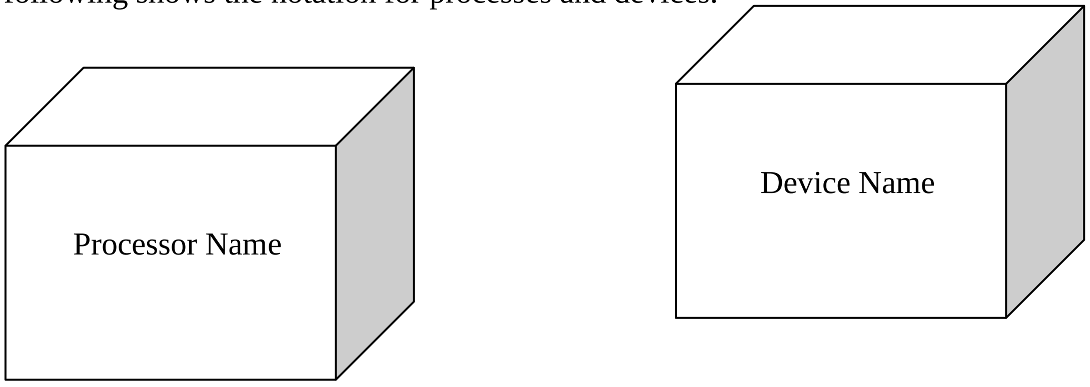


Figure: An interaction diagram

vi) Process Diagram

It is used to show allocation of processes to processors in the physical design. The following shows the notation for processes and devices.



Process 1 Process 2

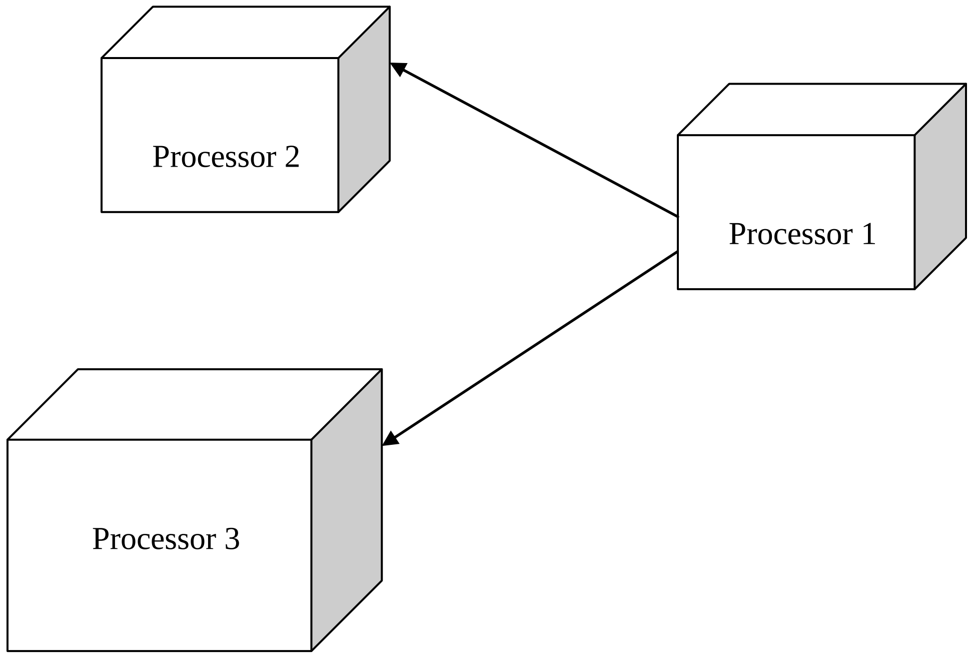


Figure: An example process diagram

The Booch's method prescribes a macro development process and a micro development process:

a) *Macro Development Process*

The macro process serves as a controlling framework for the micro process and can take weeks or even months. The primary concern of the macro process is technical management of the system .Such management is interested less in the actual OOD than in how well the project corresponds to the requirements set for it and whether it is produced on time. The macro development process consists of the following steps:

i) Conceptualization: You establish the core requirements of the system .You establish a set of goals and develop a prototype to prove the concept.

ii) Analysis and Development of the Model: You use the class diagram to describe the roles and responsibilities objects are to carry out in performing the desired behavior of the system. Then you use the object diagram to describe the desired behavior of the system in terms of scenario or use interaction diagram to describe behavior of system in terms of scenarios.

iii) Design or create the system architecture: You use the class diagram to decide what classes exist and how they relate to each other. You use object diagram to decide what mechanisms are used to regulate how objects collaborate .You use module diagram to map out where each class and object should be declared. You use the process diagram to determine to which processor .Also determine the schedules for multiple processes on each relevant processor.

iv) Evolution or implementation: Successively refine the system through many iterations .Produce a stream of software implementations, each of which is a refinement of the prior one.

v) Maintenance: Making localized changes to the system to add new requirements and eliminate bugs.

b) *Micro Development Process:* Each macro development process has its own micro development processes. It is a description of the day-to-day activities by a single or small group of software developers, which looks blurry to an outsider, since analysis and design phases are not clearly defined. It consists of the following:

i) Identify classes and objects

ii) Identify class and object semantics

iii) Identify class and object relationships

iv) Identify class and object interfaces and implementation.

The Jacobson's Methodology

The Jacobson's methodologies (e.g.: - Object oriented Business Engineering (OOBE), Object Oriented Software Engineering (OOSE) and Objectory) cover the entire life cycle and stress traceability between the different phases, both forward and backward. This traceability enables reuse of analysis and design work, possibly much bigger factors in the reduction of development time than reuse of code. At the heart of their methodologies is the use-case concept, which evolved with Objectory.

i) *Use cases*

Use cases are scenarios for understanding system requirements .A use case is an interaction between users and a system. The use case model captures the goal of the user and the responsibility of the system to its users.

In the requirement analysis, the use case is described as one of the following:

- Non-formal text with no clear flow of events.
- Text, easy to read but with a clear flow of events to follow (this is a recommended style).
- Formal style using pseudocode.

The use case description must contain:

- How and when the use case begins and ends.
- The interaction between the use case and its actors, including when the interaction occurs and what is exchanged.
- How and when the use case will need data stored in the system or will store data in the system.
- Exceptions to the flow of events.
- How and when concepts of the problem domain are handled.

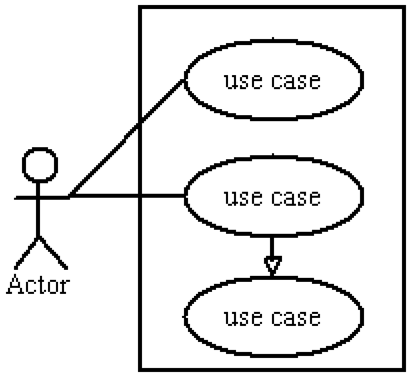
Every single use case should describe one main flow of events. An exceptional or additional flow of events could be added. The exceptional or additional flow of events could be added. The exceptional use case extends another use case to include the additional one. The use case model employs extends and uses relationships.

- The extends relationship is used when you have one use case that is similar to another use case but does a bit more .It extends the functionality of the original use case.

- The uses relationship senses common behavior in different use cases.

Use cases could be viewed as concrete or abstract.

- An abstract use case is not complete and has no actors that initiate it but is used by another use case .This inheritance could be in several levels.
- Abstract use cases are the ones that have uses or extend relationships.



ii) *Architecture*

System development includes development of different models. We need to find modeling technique notation for each model. Such a set of modeling techniques defines the architecture. Each model shows certain aspects of the system. Collection of models is known as architecture.

iii) *Object Oriented Software Engineering:*

It is also called as Objectory. It is a method of object oriented development with the specific aim to fit the development of large, real time systems. The development process also use case driven development , stresses that use cases are involved in several phases of the development , including analysis , design ,validation and testing. The use case scenario begins with a user of the system initiating a sequence of interrelated events. The system development method based on OOSE, Objectory is a disciplined process for the industrialized development of software based on a use case driven design. It is an approach to OOAD that centers on understanding the ways in which a system actually is used. By organizing the analysis and design models around sequences of user interaction and actual usage scenarios, the method produces systems that are both usable and more robust adapting more easily to changing usage. Objectory has been developed and applied to numerous application areas and embodied in CASE tool systems.

Objectory is built around several different models:

- Use Case Model: The use case model defines the outside (actors) and inside (use case) of the system' behavior.
- Domain Object Model: The objects of the real world are mapped into the domain object model.
- Analysis object model: The analysis model presents how the source code (implementation) should be carried out and written.
- Implementation Model: The implementation model represents the implementation of the system.
- Test Model: The test model constitutes the test plans, specifications and reports.

The maintenance of each model is specified in its associated process. A process is created when the first development project starts and is terminated when the developed system is taken out of service.

iv) *Object Oriented Business Engineering:*

OOBE is object modeling at the enterprise level .Use cases again are the central vehicle for modeling, providing traceability throughout the software engineering processes.

- Analysis Phase: It defines the system to be built in terms of the problem domain object model, the requirement model and analysis model. It shouldn't take into account the actual implementation model. It reduces complexity and promotes maintainability over system. Jacobson doesn't insist on development of problem domain object model, but refers to other models, who suggest that customers draw his view of system. This model should be developed just understanding for requirements model. The analysis process is iterative but requirements and analysis model should be stable before moving on to subsequent models. Prototyping is also useful.

- Design and Implementation Phases: Implementation environment should be identified for design model. The various factors affecting the design model are DBMS, distribution of process, constraints due to programming language, available component libraries and incorporation of GUI. It may be possible to identify implementation environment concurrently with analysis. The analysis objects are translated into design objects that fit current implementation environment.
- Testing Phase: It describes testing, levels and techniques (unit, integration and system.)

What is UML?

UML stands for Unified Modeling Language. This object-oriented system of notation has evolved from the work of Grady Booch, James Rumbaugh, Ivar Jacobson, and the Rational Software Corporation. These renowned computer scientists fused their respective technologies into a single, standardized model. Today, UML is accepted by the Object Management Group (OMG) as the standard for modeling object oriented programs. The primary goals in the design of the UML were as follows:

1. Provide users a ready to use expressive visual modeling language so they can develop and exchange meaningful models.

2. Provide extensibility and specialization mechanisms to extend the core concepts.

3. Be independent of particular programming languages and development processes.

4. Provide a formal basis for understanding the modeling language.

5. Encourage the growth of the OO tools market.

6. Support higher-level development concepts.

7. Integrate best practices and methodologies.

Types of UML Diagrams

UML defines nine types of diagrams: Class (package), Object, Use case, Sequence, Collaboration, State chart, Activity, Component, and Deployment.

i) UML Class Diagram

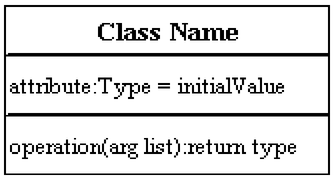
Class diagrams are the backbone of almost every object-oriented method including UML. They describe the static structure of a system.

Basic Class Diagram Symbols and Notations

Classes represent an abstraction of entities with common characteristics. Associations represent the relationships between classes.

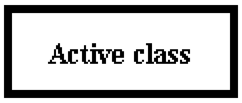
Classes

Illustrate classes with rectangles divided into compartments. Place the name of the class in the first partition (centered, bolded, and capitalized), list the attributes in the second partition, and write operations into the third.


Class Name
attribute:Type = initialValue
operation(arg list):return type

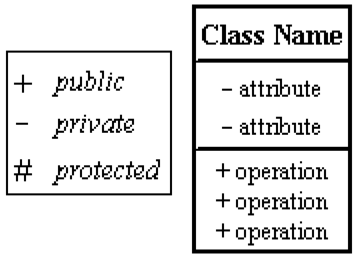
Active Class

Active classes initiate and control the flow of activity, while passive classes store data and serve other classes. Illustrate active classes with a thicker border.

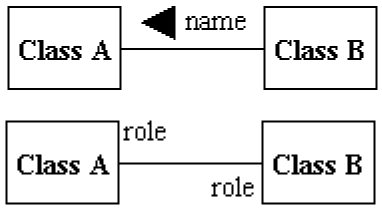


Visibility

Use visibility markers to signify who can access the information contained within a class. Private visibility hides information from anything outside the class partition. Public visibility allows all other classes to view the marked information. Protected visibility allows child classes to access information they inherited from a parent class.



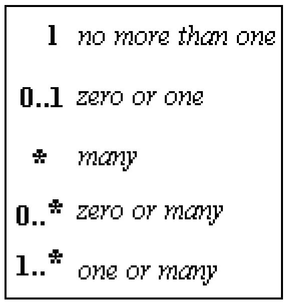
Associations - Associations represent static relationships between classes. Place association names above, on, or below the association line. Use a filled arrow to indicate the direction of the relationship. Place roles near the end of an association. Roles represent the way the two classes see each other.



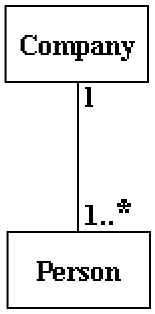
More Basic Class Diagram Symbols and Notations

Multiplicity(Cardinality)

Place multiplicity notations near the ends of an association. These symbols indicate the number of instances of one class linked to one instance of the other class. For example, one company will have one or more employees, but each employee works for one company only.

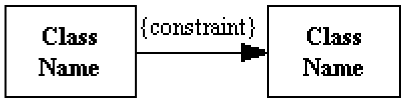


1	*no more than one*
0..1	*zero or one*
*	*many*
0..*	*zero or many*
1..*	*one or many*

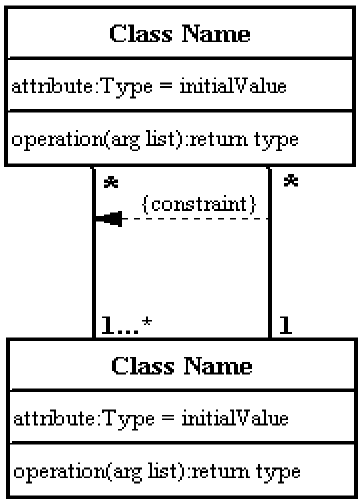


Constraint

Place constraints inside curly braces {}.

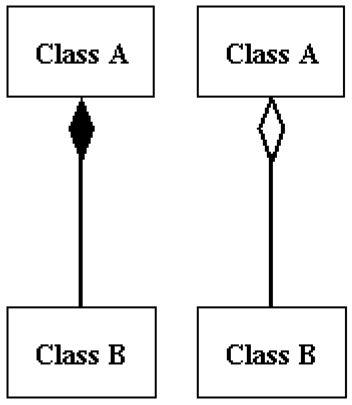


Simple Constraint


Class Name
attribute:Type = initialValue
operation(arg list):return type
*
{constraint}
*
1...*
1
Class Name
attribute:Type = initialValue
operation(arg list):return type

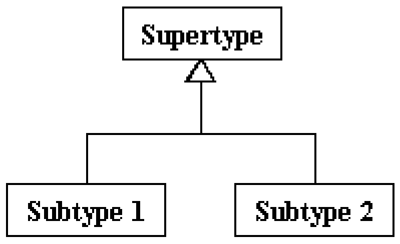
Composition and Aggregation

Composition is a special type of aggregation that denotes a strong ownership between Class A, the whole, and Class B, its part. Illustrate composition with a filled diamond. Use a hollow diamond to represent a simple aggregation relationship, in which the "whole" class plays a more important role than the "part" class, but the two classes are not dependent on each other. The diamond end in both a composition and aggregation relationship points toward the "whole" class or the aggregate.



Generalization

Generalization is another name for inheritance or an "is a" relationship. It refers to a relationship between two classes where one class is a specialized version of another. For example, Honda is a type of car. So the class Honda would have a generalization relationship with the class car.



In real life coding examples, the difference between inheritance and aggregation can be confusing. If you have an aggregation relationship, the aggregate (the whole) can access only the PUBLIC functions of the part class. On the other hand, inheritance allows the inheriting class to access both the PUBLIC and PROTECTED functions of the superclass.

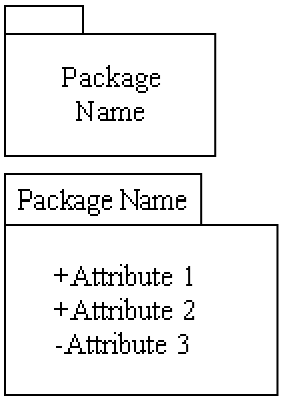
ii) UML Package Diagram

Package diagrams organize the elements of a system into related groups to minimize dependencies among them

Basic Package Diagram Symbols and Notations

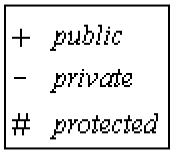
Packages

Use a tabbed folder to illustrate packages. Write the name of the package on the tab or inside the folder. Similar to classes, you can also list the attributes of a package.



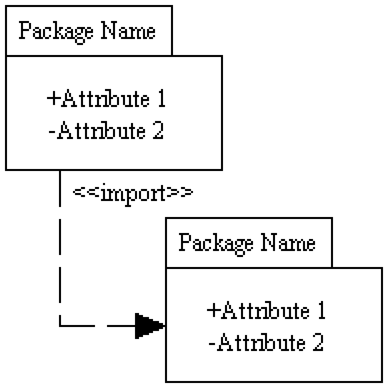
Visibility

Visibility markers signify who can access the information contained within a package. Private visibility means that the attribute or the operation is not accessible to anything outside the package. Public visibility allows an attribute or an operation to be viewed by other packages. Protected visibility makes an attribute or operation visible to packages that inherit it only.



Dependency

Dependency defines a relationship in which changes to one package will affect another package. Importing is a type of dependency that grants one package access to the contents of another package.



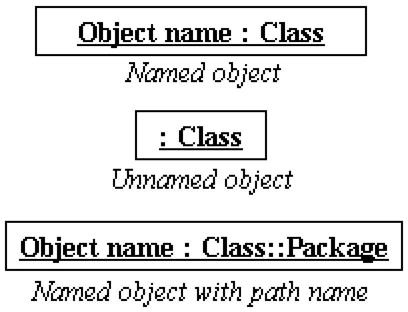
iii) UML Object Diagram

Object diagrams are also closely linked to class diagrams. Just as an object is an instance of a class, an object diagram could be viewed as an instance of a class diagram. Object diagrams describe the static structure of a system at a particular time and they are used to test the accuracy of class diagrams.

Basic Object Diagram Symbols and Notations

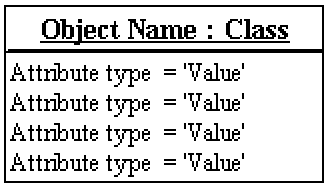
Object names

Each object is represented as a rectangle, which contains the name of the object and its class underlined and separated by a colon.



Object attributes

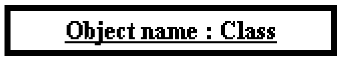
As with classes, you can list object attributes in a separate compartment. However, unlike classes, object attributes must have values assigned to them.



Object with attributes

Active object

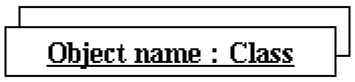
Objects that control action flow are called active objects. Illustrate these objects with a thicker border.



Active object

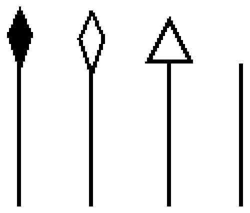
Multiplicity

You can illustrate multiple objects as one symbol if the attributes of the individual objects are not important.



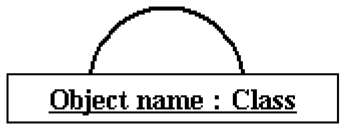
Multiple objects

Links - Links are instances of associations. You can draw a link using the lines used in class diagrams.


Links

Self-linked

Objects that fulfill more than one role can be self-linked. For example, if Mark, an administrative assistant, also fulfilled the role of a marketing assistant, and the two positions are linked, Mark's instance of the two classes will be self-linked.



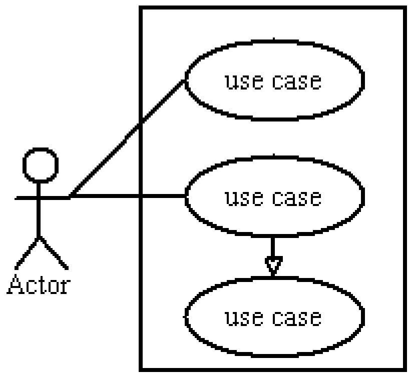
iv) UML Use Case Diagram

Use case diagrams model the functionality of a system using actors and use cases. Use cases are services or functions provided by the system to its users.

Basic Use Case Diagram Symbols and Notations

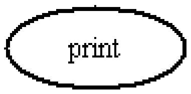
System

Draw your system's boundaries using a rectangle that contains use cases. Place actors outside the system's boundaries.



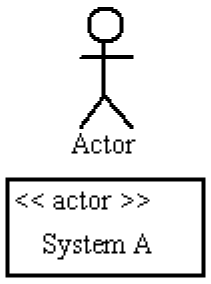
Use Case

Draw use cases using ovals. Label with ovals with verbs that represent the system's functions.



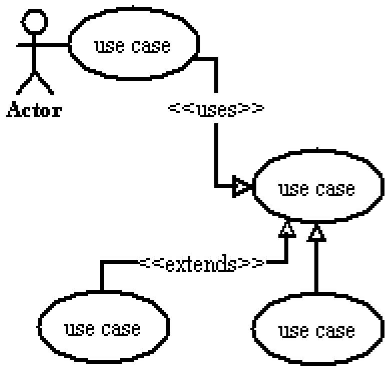
Actors

Actors are the users of a system. When one system is the actor of another system, label the actor system with the actor stereotype.



Relationships

Illustrate relationships between an actor and a use case with a simple line. For relationships among use cases, use arrows labeled either "uses" or "extends." A "uses" relationship indicates that one use case is needed by another in order to perform a task. An "extends" relationship indicates alternative options under a certain use case.

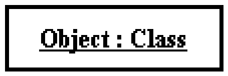


v) UML Sequence Diagram

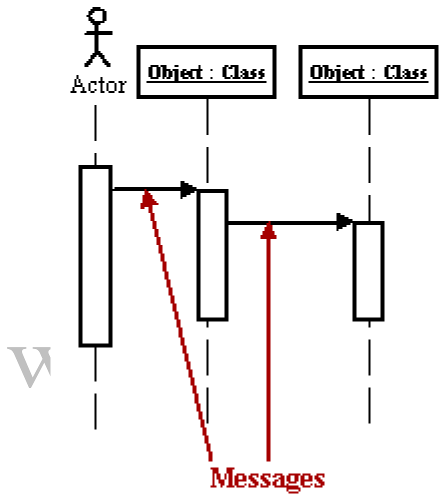
Sequence diagrams describe interactions among classes in terms of an exchange of messages over time.

Basic Sequence Diagram Symbols and Notations

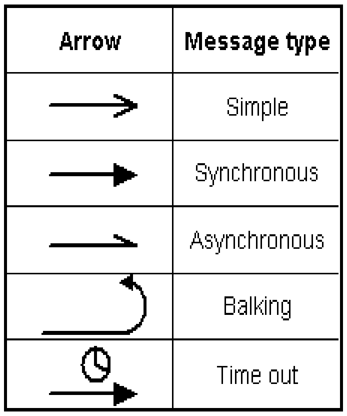
Class roles
Class roles describe the way an object will behave in context. Use the UML object symbol to illustrate class roles, but don't list object attributes.



Activation
Activation boxes represent the time an object needs to complete a task.



Messages - Messages are arrows that represent communication between objects. Use half-arrowed lines to represent asynchronous messages. Asynchronous messages are sent from an object that will not wait for a response from the receiver before continuing its tasks.

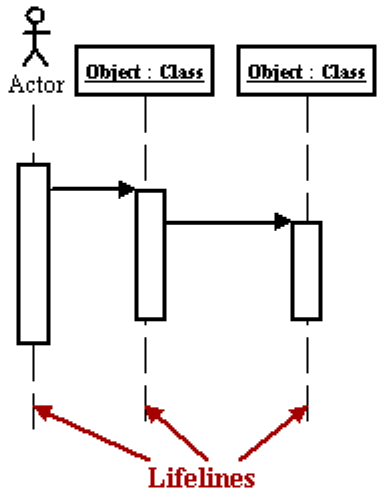


Arrow	Message type
→	Simple
➔	Synchronous
⇀	Asynchronous
⤴	Balking
◷ ➔	Time out

Various message types for Sequence and Collaboration diagrams

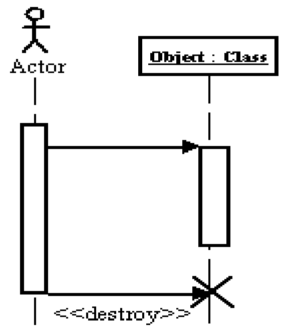
Lifelines

Lifelines are vertical dashed lines that indicate the object's presence over time.



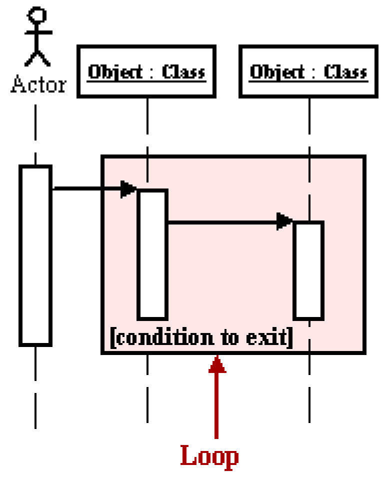
Destroying Objects

Objects can be terminated early using an arrow labeled "< < destroy > >" that points to an X.



Loops

A repetition or loop within a sequence diagram is depicted as a rectangle. Place the condition for exiting the loop at the bottom left corner in square brackets [ ].



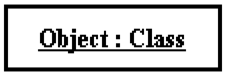
vi) UML Collaboration Diagram

A collaboration diagram describes interactions among objects in terms of sequenced messages. Collaboration diagrams represent a combination of information taken from class, sequence, and use case diagrams describing both the static structure and dynamic behavior of a system.

Basic Collaboration Diagram Symbols and Notations

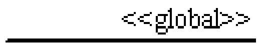
Class roles

Class roles describe how objects behave. Use the UML object symbol to illustrate class roles, but don't list object attributes.



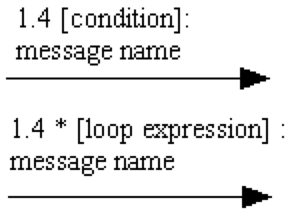
Association roles

Association roles describe how an association will behave given a particular situation. You can draw association roles using simple lines labeled with stereotypes.



Messages

Unlike sequence diagrams, collaboration diagrams do not have an explicit way to denote time and instead number messages in order of execution. Sequence numbering can become nested using the Dewey decimal system. For example, nested messages under the first message are labeled 1.1, 1.2, 1.3, and so on. The condition for a message is usually placed in square brackets immediately following the sequence number. Use a * after the sequence number to indicate a loop.



vii) UML State chart Diagram A state chart diagram shows the behavior of classes in response to external stimuli. This diagram models the dynamic flow of control from state to state within a system.

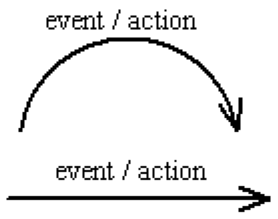
Basic State chart Diagram Symbols and Notations

States

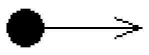
States represent situations during the life of an object. You can easily illustrate a state in SmartDraw by using a rectangle with rounded corners.

Transition

A solid arrow represents the path between different states of an object. Label the transition with the event that triggered it and the action that results from it.

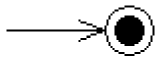


More Basic Collaboration Diagram Symbols and Notations



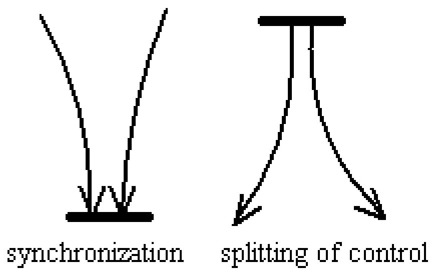
Initial State

A filled circle followed by an arrow represents the object's initial state.



Final State

An arrow pointing to a filled circle nested inside another circle represents the object's final state.



Synchronization and Splitting of Control

A short heavy bar with two transitions entering it represents a synchronization of control. A short heavy bar with two transitions leaving it represents a splitting of control that creates multiple states.

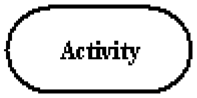
viii) UML Activity Diagram

An activity diagram illustrates the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Typically, activity diagrams are used to model workflow or business processes and internal operation. Because an activity diagram is a special kind of state chart diagram, it uses some of the same modeling conventions.

Basic Activity Diagram Symbols and Notations

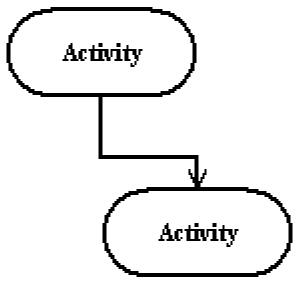
Action states

Action states represent the non interruptible actions of objects. You can draw an action state in SmartDraw using a rectangle with rounded corners.



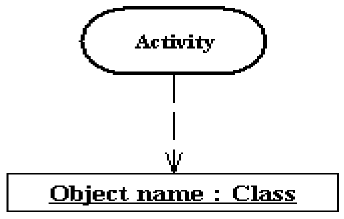
Action Flow

Action flow arrows illustrate the relationships among action states.



Object Flow

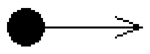
Object flow refers to the creation and modification of objects by activities. An object flow arrow from an action to an object means that the action creates or influences the object. An object flow arrow from an object to an action indicates that the action state uses the object.



More Basic Collaboration Diagram Symbols and Notations

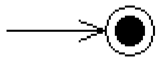
Initial State

A filled circle followed by an arrow represents the initial action state.



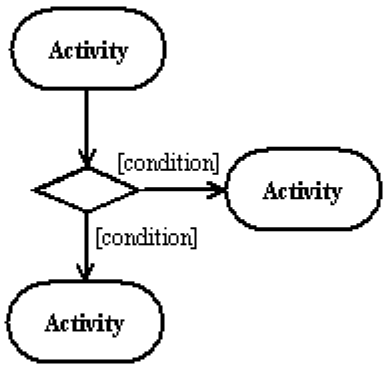
Final State

An arrow pointing to a filled circle nested inside another circle represents the final action state.



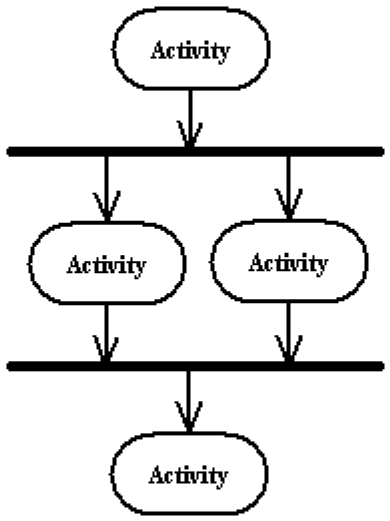
Branching

A diamond represents a decision with alternate paths. The outgoing alternates should be labeled with a condition or guard expression. You can also label one of the paths "else."



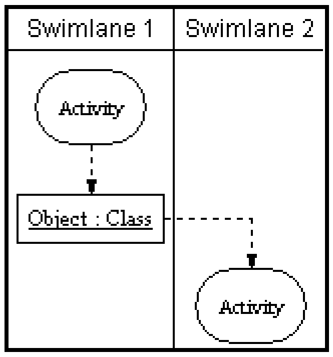
Synchronization

A synchronization bar helps illustrates parallel transitions. Synchronization is also called forking and joining.



Swimlanes

Swimlanes group related activities into one column.

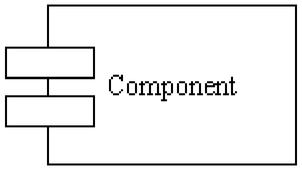


ix) UML Component Diagram

A component diagram describes the organization of the physical components in a system.

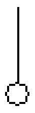
Basic Component Diagram Symbols and Notations

Component A component is a physical building block of the system. It is represented as a rectangle with tabs.



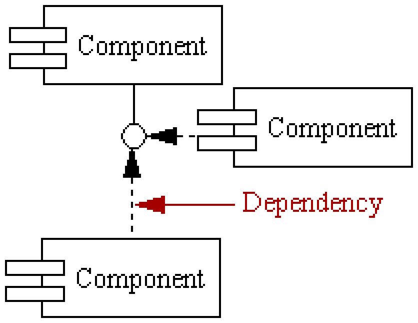
Interface

An interface describes a group of operations used or created by components.



Dependencies

Draw dependencies among components using dashed arrows.



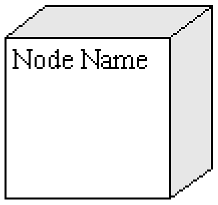
x) UML Deployment Diagram

Deployment diagrams depict the physical resources in a system including nodes, components, and connections.

Basic Deployment Diagram Symbols and Notations

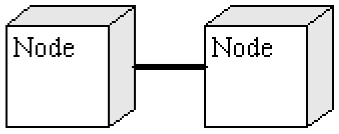
Node

A node is a physical resource that executes code components.



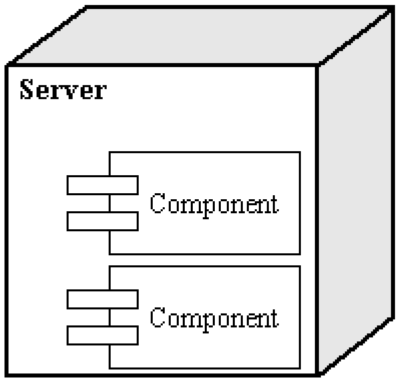
Association

Association refers to a physical connection between nodes, such as Ethernet.



Components and Nodes

Place components inside the node that deploys them.


Server
Component
Component