Unit 2: Advanced Class Modeling 6 Hours

Toipics :

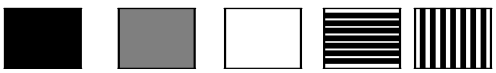
- ***Advanced object and class concepts***
- ***Asoociation ends***
- ***N-ary association***
- ***Aggregation***
- ***Abstract classes***
- ***Multiple inheritance***
- ***Metadata***
- ***Reification***
- ***Constraints***
- ***Derived data***
- ***Packages***

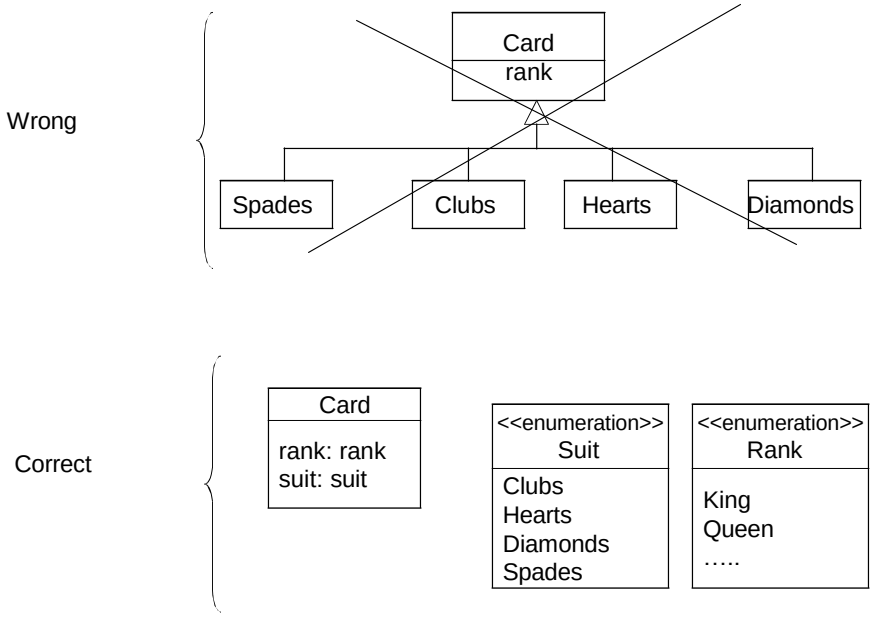
2.1 Advanced object and class concepts

2.1.1 Enumerations

A data type is a description of values, includes numbers, strings, enumerations
Enumerations: A Data type that has a finite set of values.

- When constructing a model, we should carefully note enumerations, because they often occur and are important to users.
- Enumerations are also significant for an implantation; we may display the possible values with a pick list and you must restrict data to the legitimate values.
- Do not use a generalization to capture the values of an Enumerated attribute.
- An Enumeration is merely a list of values; generalization is a means for structuring the description of objects.
- Introduce generalization only when at least one subclass has significant attributes, operations, or associations that do not apply to the superclass.
- In the UML an enumeration is a data type.
- We can declare an enumeration by listing the keyword *enumeration* in guillemets (<< >>) above the enumeration name in the top section of a box. The second section lists the enumeration values.
- Eg: Boolean type= { TRUE, FALSE}
- Eg: figure.pentype _____ - - - - - --------
- Two diml.filltype

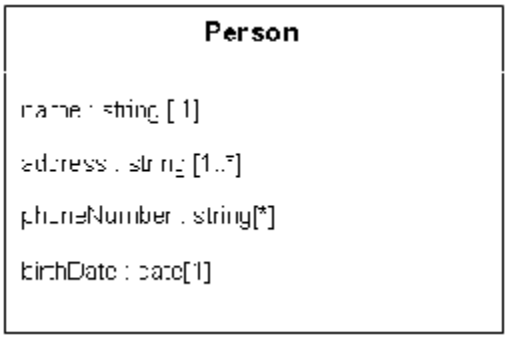




Modeling enumerations. Do not use a generalization to capture the values of an enumerated attribute

2.1.2 Multiplicity

- Multiplicity is a collection on the cardinality of a set, also applied to attributes (database application).
- Multiplicity of an attribute specifies the number of possible values for each instantiation of an attribute. i.e., whether an attribute is mandatory ([1]) or an optional value ([0..1] or * i.e., null value for database attributes) .
- Multiplicity also indicates whether an attribute is single valued or can be a collection.



2.1.3 Scope

- Scope indicates if a feature applies to an object or a class.
- An underline distinguishes feature with class scope (static) from those with object scope.
- Our convention is to list attributes and operations with class scope at the top of the attribute and operation boxes, respectively.

- It is acceptable to use an attribute with class scope to hole the **extent** of a class (the set of objects for a class) - this is common with OO databases. Otherwise, you should avoid attributes with class scope because they can lead to an inferior model.
- It is better to model groups explicitly and assigns attributes to them.
- In contrast to attributes, it is acceptable to define operations of class scope. The most common use of class-scoped operations is to create new instances of a class, sometimes for summary data as well.

2.1.4 Visibility

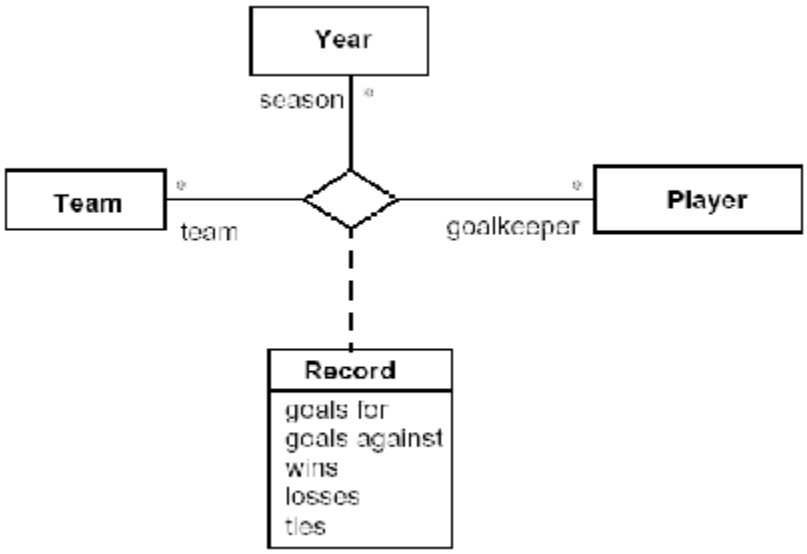
- Visibility refers to the ability of a method to reference a feature from another class and has the possible values of *public, protected, private,* and *package.*
- Any method can access **public** features.
- Only methods of the containing class and its descendants via inheritance can access **protected** features.
- Only methods of the containing class can access **private** features.
- Methods of classes defined in the same package as the target class can access **package** features
- The UML denotes visibility with a prefix. **"+"→ public, "-"→ private, "#"→protected, "~"→ package.** Lack of a prefix reveals no information about visibility.
- Several issues to consider when choosing visibility are
  - **Comprehension**: understand all public features to understand the capabilities of a class. In contrast we can ignore private, protected, package features – they are merely an implementation convince.
  - **Extensibility**: many classes can depend on public methods, so it can be highly disruptive to change their signature. Since fewer classes depend on private, protected, and package methods, there is more latitude to change them.
  - **Context**: private, protected, and package methods may rely on preconditions or state information created by other methods in the class. Applied out of context, a private method may calculate incorrect results or cause the object to fail.

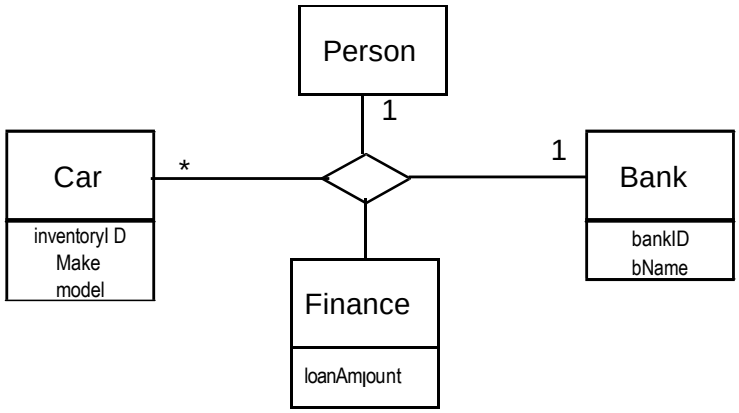
2.2 Associations ends

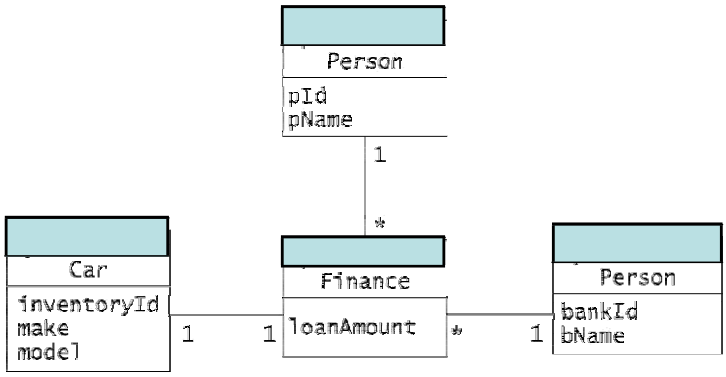
- Association End is an end of association.
- A binary association has 2 ends; a ternary association has 3 ends.

2.3 N-ary Association

- We may occasionally encounter n-ary associations (association among 3 or more classes). But we should try to avoid n-ary associations- most of them can be decomposed into binary associations, with possible qualifiers and attributes.





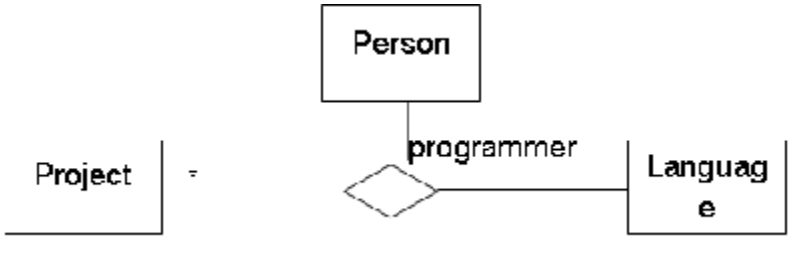


-
- The UML symbol for n-ary associations is a diamond with lines connecting to related classes. If the association has a name, it is written in italics next to the diamond.
- The OCL does not define notation for traversing n-ary associations.

- A typical programming language cannot express n-ary associations. So, promote n-ary associations to classes. Be aware that you change the meaning of a model, when you promote n-ary associations to classes.
- An n-ary association enforces that there is at most one link for each combination.

Eg:

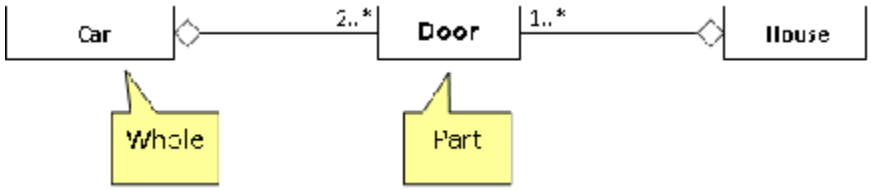
Class diagram



Instance diagram *see prescribed text book page no. 65 and fing no. 4.6*

2.4 Aggregation

- Aggregation is a strong form of association in which an aggregate object is made of constituent parts.
- Constituents are the parts of aggregate.
- The aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects.
- We define an aggregation as relating an assembly class to one constituent part class.
- An assembly with many kinds of constituent parts corresponds to many aggregations.
- We define each individual pairing as an aggregation so that we can specify the multiplicity of each constituent part within the assembly. This definition emphasizes that aggregation is a special form of binary association.
- The most significant property of aggregation is transitivity (if A is part of B and B is part of C, then A is part of C) and antisymmetric (if A is part of B then B is not part of A)



2.4.1 Aggregation versus Association

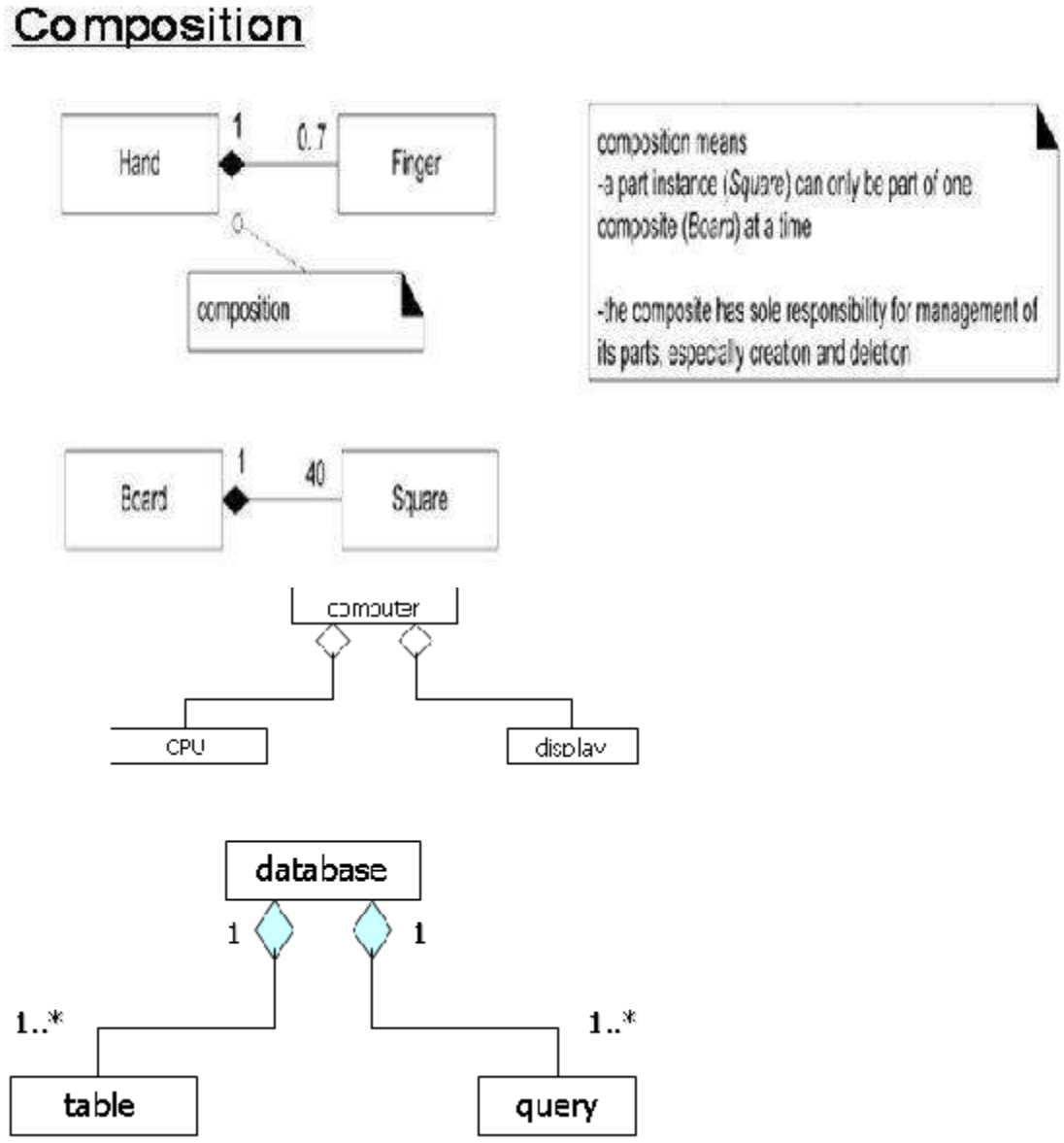
- Aggregation is a special form of association, not an independent concept.
- Aggregation adds semantic connotations.
- If two objects are tightly bound by a part-whole relationship, it is an aggregation. If the two objects are usually considered as independent, even though they may often be linked, it is an association.
- Aggregation is drawn like association, except a small (hollow) diamond indicates the assembly end.

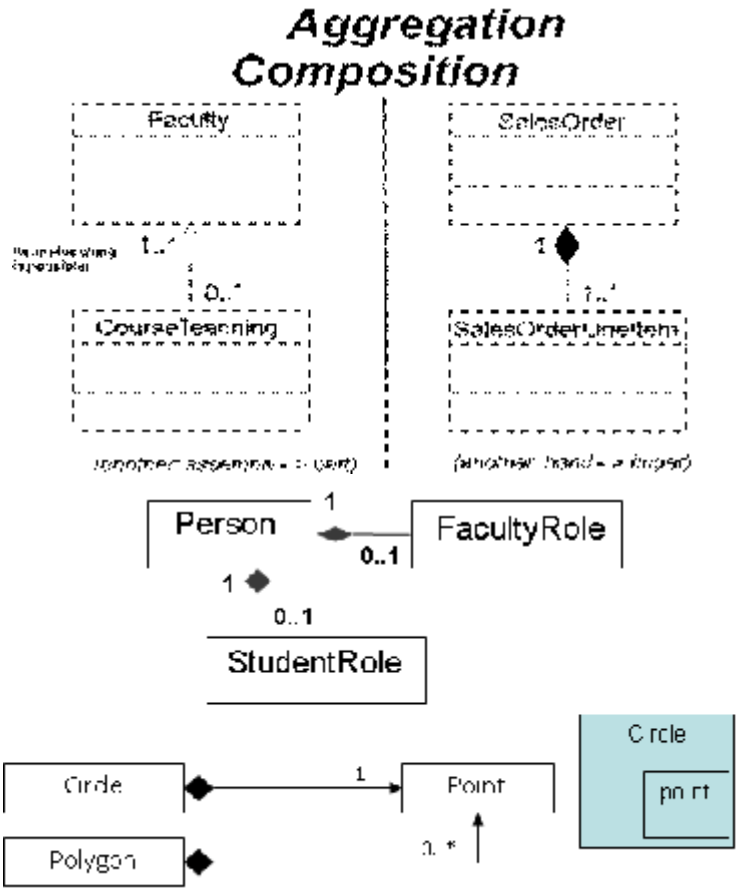
- The decision to use aggregation is a matter of judgment and can be arbitrary.

2.4.2 Aggregation versus Composition

- The UML has 2 forms of part-whole relationships: a general form called Aggregation and a more restrictive form called composition.
- Composition is a form of aggregation with two additional constraints.
- A constitute part can belong to at most one assembly.
- Once a constitute part has been assigned an assembly, it has a coincident lifetime with the assembly. Thus composition implies ownership of the parts by the whole.
- This can be convenient for programming: Deletion of an assembly object triggers deletion of all constituent objects via composition.
- Notation for composition is a small solid diamond next to the assembly class.

Eg: see text book examples also





2.4.3 Propagation of Operations

- Propagation (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.
- Provides concise and powerful way of specifying a continuum behavior.
- Propagation is possible for other operations including save/restore, destroy, print, lock, display.
- Notation (not an UML notation): a small arrow indicating the direction and operation name next to the affected association.

  Eg: see page no: 68 fig: 4.11

2.5 Abstract Classes

- Abstract class is a class that has no direct instances but whose descendant classes have direct instances.
- A concert class is a class that is insatiable; that is, it can have direct instances.
- A concrete class may have abstract class.
- Only concrete classes may be leaf classes in an inheritance tree.

  Eg: see text book page no: 69, 70 fig: 4.12, 4.13,4.14

- In UML notation an abstract class name is listed in an italic (or place the keyword {abstract} below or after the name).
- We can use abstract classes to define the methods that can be inherited by subclasses.
- Alternatively, an abstract class can define the signature for an operation with out supplying a corresponding method. We call this an abstract operation.
- Abstract operation defines the signature of an operation for which each concrete subclass must provid4 its own implementation.
- A concrete class may not contain abstract operations, because objects of the concrete class would have undefined operations.

2.6 Multiple Inheritance

- Multiple inheritance permits a class to have more than one superclass and to inherit features from all parents.
- We can mix information from 2 or more sources.
- This is a more complicated from of generalization than single inheritance, which restricts the class hierarchy to a tree.
- The advantage of multiple inheritance is greater power in specifying classes and an increased opportunity for reuse.
- The disadvantage is a loss of conceptual and implementation simplicity.
- The term multiple inheritance is used somewhat imprecisely to mean either the conceptual relationship between classes or the language mechanism that implements that relationship.

2.6.1 Kinds of Multiple Inheritance

- The most common form of multiple inheritance is from sets of disjoint classes. Each subclass inherits from one class in each set.
- The appropriate combinations depend on the needs of an application.
- Each generalization should cover a single aspect.
- We should use multiple generalizations if a class can be refined on several distinct and independent aspects.
- A subclass inherits a feature from the same ancestor class found along more than one path only once; it is the same feature.
- Conflicts among parallel definitions create ambiguities that implementations must resolve. In practice, avoid such conflicts in models or explicitly resolve them, even if a particular language provides a priority rule for resolving conflicts.
- The UML uses a constraint to indicate an overlapping generalization set; the notation is a dotted line cutting across the affected generalization with keywords in braces.
  Eg: see text book page no: 71,72 fig: 4.15,4.16

2.6.2 Multiple Classification

- An instance of a class is inherently an instance of all ancestors of the class.
- For example, an instructor could be both faculty and student. But what about a Harvard Professor taking classes at MIT? There is no class to describe the

combination. This is an example of multiple classification, in which one instance happens to participate in two overlapping classes.

Eg: see text book page no: 73 fig: 4.17

2.6.3 Workarounds

- Dealing with lack of multiple inheritance is really an implementation issue, but early restructuring of a model is often the easiest way to work around its absence.
- Here we list 2 approaches for restructuring techniques (it uses delegation)
- Delegation is an implementation mechanism by which an object forwards an operation to another object for execution.

1. **Delegation using composition of parts:** Here we can recast a superclass with multiple independent generalization as a composition in which each constituent part replaces a generalization. This is similar to multiple classification. This approach replaces a single object having a unique ID by a group of related objects that compose an extended object. Inheritance of operations across the composition is not automatic. The composite must catch operations and delegate them to the appropriate part.

   In this approach, we need not create the various combinations as explicit classes. All combinations of subclasses from the different generalization are possible.
2. **Inherit the most important class and delegate the rest:**
   Fig 4.19 preserves identity and inheritance across the most important generalization. We degrade the remaining generalization to composition and delegate their operations as in previous alternative.
3. **Nested generalization:** this approach multiplies out all possible combinations. This preserves inheritance but duplicates declarations and code and violets the spirit of OO programming.
4. **Superclasses of equal importance:** if a subclass has several superclasses, all of equal importance, it may be best to use delegation and preserve symmetry in the model.
5. **Dominant superclass:** if one superclass clearly dominates and the others are less important, preserve inheritance through this path.
6. **Few subclasses:** if the number of combinations is small, consider nested generalization. If the number of combinations is large, avoid it.
7. **Sequencing generalization sets:** if we use generalization, factor on the most important criterion first, the next most important second, and so forth.
8. **Large quantities of code:** try to avoid nested generalization if we must duplicate large quantities of code.
9. **Identity:** consider the importance of maintaining strict identity. Only nested generalization preserves this.

2.7 Metadata

- Metadata is data that describes other data. For example, a class definition is a metadata.
- Models are inherently metadata, since they describe the things being modeled (rather than being the things).
- Many real-world applications have metadata, such as parts catalogs, blueprints, and dictionaries. Computer-languages implementations also use metadata heavily.
- We can also consider classes as objects, but classes are meta-objects and not real-world objects. Class descriptor object have features, and they in turn have their own classes, which are called ***metaclasses.***

  Eg: see text book page no: 75 fig: 4.21

2.8 Reification

- Reification is the promotion of something that is not an object into an object.
- Reification is a helpful technique for Meta applications because it lets you shift the level of abstraction.
- On occasion it is useful to promote attributes, methods, constraints, and control information into objects so you can describe and manipulate them as data.
- As an example of reification, consider a database manager. A developer could write code for each application so that it can read and write from files. Instead, for many applications, it is better idea to reify the notion of data services and use a database manager. A database manager has abstract functionality that provides a general-purpose solution to accessing data reliably and quickly for multiple users.

  Eg: see text book page no: 75 fig: 4.22

2.9 Constraints

- Constraint is a condition involving model elements, such as objects, classes, attributes, links, associations, and generalization sets.
- A Constraint restricts the values that elements can assume by using OCL.

2.9.1 Constraints on objects

Eg: see text book page no: 77 fig: 4.23

2.9.2 Constraints on generalization sets

- Class models capture many Constraints through their very structure. For example, the semantics of generalization imply certain structural constraints.
- With single inheritance the subclasses are mutually exclusive. Furthermore, each instance of an abstract superclass corresponds to exactly one subclass instance. Each instance of a concrete superclass corresponds to at most one subclass instance.
- The UML defines the following keyword s for generalization.
  - **Disjoint:** The subclasses are mutually exclusive. Each object belongs to exactly one of the subclasses.
  - **Overlapping:** The subclasses can share some objects. An object may belong to more than one subclass.

- **Complete:** The generalization lists all the possible subclasses.
- **Incomplete:** The generalization may be missing some subclasses.

2.9.3 Constraints on Links

- Multiplicity is a constraint on the cardinality of a set. Multiplicity for an association restricts the number of objects related to a given object.
- Multiplicity for an attribute specifies the number of values that are possible for each instantiation of an attribute.
- Qualification also constraints an association. A qualifier attribute does not merely describe the links of an association but is also significant in resolving the "many" objects at an association end.
- An association class implies a constraint. An association class is a class in every right; for example, it can have attribute and operations, participate in associations, and participate in generalization. But an association class has a constraint that an ordinary class does not; it derives identity from instances of the related classes.
- An ordinary association presumes no particular order on the object of a "many" end. The constraint {ordered} indicates that the elements of a "many" association end have an explicit order that must be preserved.

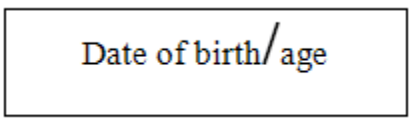
  Eg: see text book page no: 78 fig: 4.24

2.9.4 Use of constraints

- It is good to express constraints in a declarative manner. Declaration lets you express a constraint's intent, without supposing an implementation.
- Typically, we need to convert constraints to procedural form before we can implement them in a programming language, but this conversion is usually straightforward.
- A "good" class model captures many constraints through its structure. It often requires several iterations to get the structure of a model right from the prospective of constraints. Enforce only the important constraints.
- The UML has two alternative notations for constraints; either delimit a constraint with braces or place it in a "dog-earned" comment box. We can use dashed lines to connect constrained elements. A dashed arrow can connect a constrained element to the element on which it depends.

2.10. Derived Data

- A derived element is a function of one or more elements, which in turn may be derived. A derived element is redundant, because the other elements completely determine it. Ultimately, the derivation tree terminates with base elements. Classes, associations, and attributes may be derived. The notation for a derived element is a slash in front of the element name along with constraint that determines the derivation.

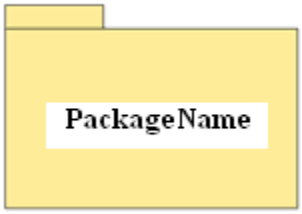


- A class model should generally distinguish independent base attributes from dependent derived attributes.

  Eg: see text book page no: 79 fig: 4.25

2.11 Packages

- A package is a group of elements (classes, association, generalization, and lesser packages) with a common theme.
- A package partitions a model, making it easier to understand and manage.
- A package partitions a model making it easier to understand and manage. Large applications my require several tiers of packages.
- Packages form a tree with increasing abstraction toward the root, which is the application, the top-level package.
- Notation for pakage is a box with a tab.



❖ Tips for devising packages

- Carefully delineate each packages's scope
- Define each class in a single package
- Make packages cohesive.

State Modeling

State model describes the sequences of operations that occur in response to external stimuli.

The state model consists of multiple state diagrams, one for each class with temporal behavior that is important to an application.

The state diagram is a standard computer science concept that relates events and states.

Events represent external stimuli and states represent values objects.

Events

An event is an occurrence at a point in time, such as user depresses left button or Air Deccan flight departs from Bombay.

An event happens instantaneously with regard to time scale of an application.

One event may logically precede or follow another, or the two events may be unrelated (concurrent; they have no effect on each other).

Events include error conditions as well as normal conditions.

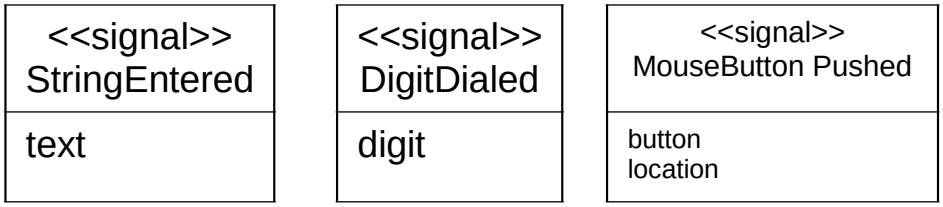
Three types of events:

- signal event,
- change event,

- time event.

Signal Event

- A signal is an explicit one-way transmission of information from one object to another.
- It is different form a subroutine call that returns a value.
- An object sending a signal to another object may expect a reply, but the reply is a separate signal under the control of the second object, which may or may not choose to send it.
- A signal event is the event of sending or receiving a signal (concern about receipt of a signal).
- Eg:



The difference between signal and signal event

a signal is a message between objects

a signal event is an occurrence in time.

Change Event

- A change event is an event that is caused by the satisfaction of a Boolean expression.
- UML notation for a change event is keyword when followed by a parenthesized Boolean expression.

Eg:

- when (room temperature < heating set point)
- when (room temperature > cooling set point)
- when (battery power < lower limit)
- when (tire pressure < minimum pressure)

Time Event

- Time event is an event caused by the occurrence of an absolute time or the elapse of a time interval.
- UML notation for an absolute time is the keyword when followed by a parenthesized expression involving time.
- The notation for a time interval is the keyword after followed by a parenthesized expression that evaluates to a time duration.

Eg:

- when (date = jan 1, 2000)
- after (10 seconds)

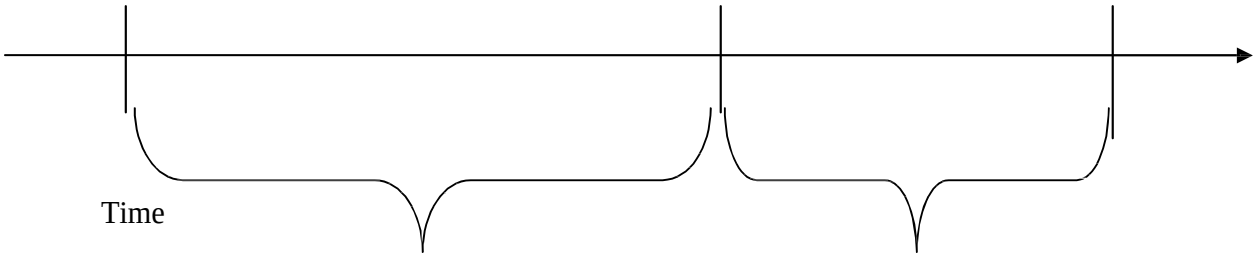
States

- A state is an abstraction of the values and links of an object.
- Sets of values and links are grouped together into a state according to the gross behavior of objects
- UML notation for state- a rounded box Containing an optional state name, list the state name in boldface, center the name near the top of the box, capitalize the fist letter.
- Ignore attributes that do not affect the behavior of the object.
- The objects in a class have a finite number of possible states.
- Each object can be in one state at a time.
- A state specifies the response of an object to input events.
- All events are ignored in a state, except those for which behavior is explicitly prescribed.

Event vs. States

- Event represents points in time.
- State represents intervals of time.

• Eg: **power turned on** **power turned off** **power turned on**



• **Powered** **Not powered**

A state corresponds to the interval between two events received by an object.
The state of an object depends on past events.
Both events and states depend on the level of abstraction.

State Alarm ringing on a watch

- **State** : *Alarm Ringing*
- **Description** : alarm on watch is ringing to indicate target time
- **Event sequence that produces the state** :

  setAlarm (targetTime)

  any sequence not including *clearAlarm*

  when (*currentTime = targetTime*)
- **Condition that characterrizes the state:**

  alarm = on, alarm set to *targetTime*,

  targetTime<= currentTime <=targetTime+20 sec , and no button has been pushed since *targetTime*
- **Events accepted in the state:**

event	response	next state
when (*currentTime = targetTime+20*)	*resetAlarm*	*normal*
buttonPushed(any button)	*resetAlarm*	*normal*

Fig: various characterizations of a state. A state specifies the response of an object to input events

Transitions & Conditions

- A transition is an instantaneous change from one state to another.
- The transition is said to fire upon the change from the source state to target state.
- The origin and target of a transition usually are different states, but sometimes may be the same.
- A transition fires when its events (multiple objects) occurs.
- A guard condition is a Boolean expression that must be true in order for a transition to occur.
- A guard condition is checked only once, at the time the event occurs, and the transition fires if the condition is true.

Guard condition Vs. change event

Guard condition	change event
a guard condition is checked only once	a change event is checked continuously
UML notation for a transition is a line	may include event label in italics
followed by guard condition in square brackets	from the origin state to the target state an arrowhead points to the target state.

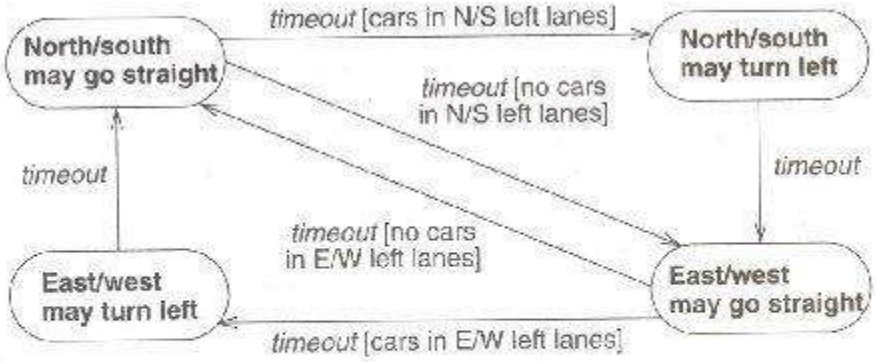


Figure 5.7 Guarded transitions. A transition is an instantaneous change from one state to another. A guard condition is a boolean expression that must be true in order for a transition to occur.

State Diagram

- A state diagram is a graph whose nodes are states and whose directed arcs are transitions between states.
- A state diagram specifies the state sequence caused by event sequences.
- State names must be unique within the scope of a state diagram.
- All objects in a class execute the state diagram for that class, which models their common behavior.
- A state model consists of multiple state diagrams one state diagram for each class with important temporal behavior.
- State diagrams interact by passing events and through the side effects of guard conditions.
- UML notation for a state diagram is a rectangle with its name in small pentagonal tag in the upper left corner.
- The constituent states and transitions lie within the rectangle.
- States do not totally define all values of an object.
- If more than one transition leaves a state, then the first event to occur causes the corresponding transition to fire.
- If an event occurs and no transition matches it, then the event is ignored.
- If more than one transition matches an event, only one transition will fire, but the choice is nondeterministic.

Eg: Sample state diagram

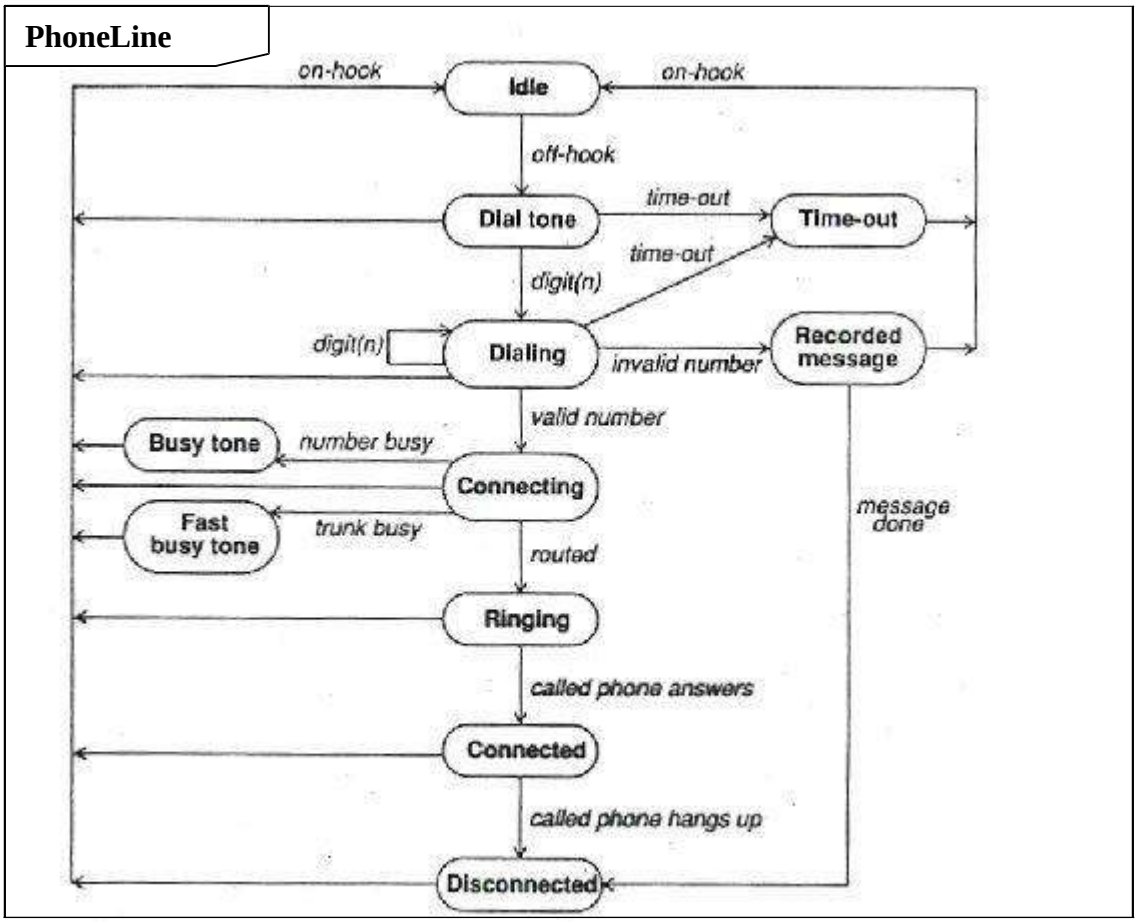


Figure 5.5 State diagram for phone line

One shot state diagrams

- State diagrams can represent continuous loops or one-shot life cycles

Diagram for the [hone line is a continuous loop

One – shot state diagrams represent objects with finite lives and have initial and final states.

- Tine initial state is entered on creation of an object
- Entry of the final state implies destruction of the object.

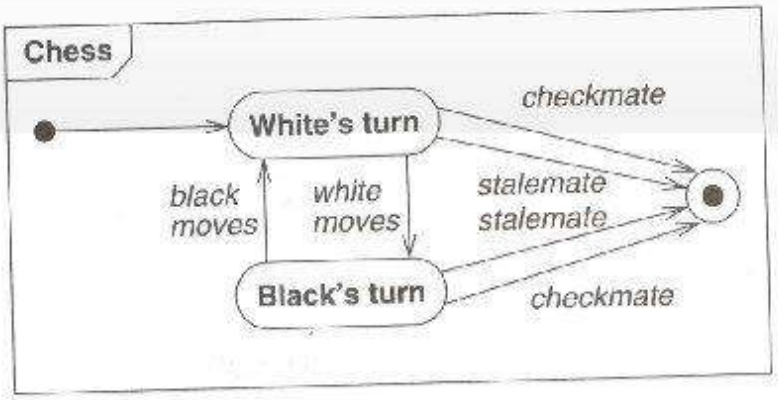


Figure 5.9 State diagram for chess game. One-shot diagrams represent objects with finite lives.

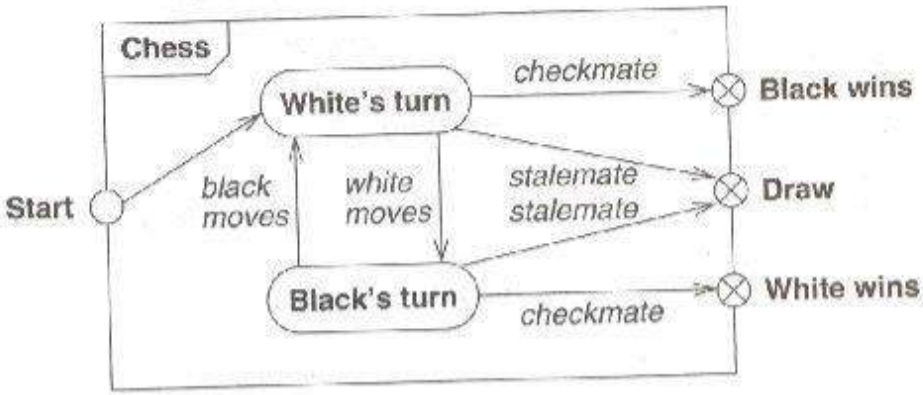


Figure 5.10 State diagram for chess game. You can also show one-shot diagrams by using entry and exit points.

5.4.3 Summary of Basic State Diagram Notation

Figure 5.11 summarizes the basic UML syntax for state diagrams.

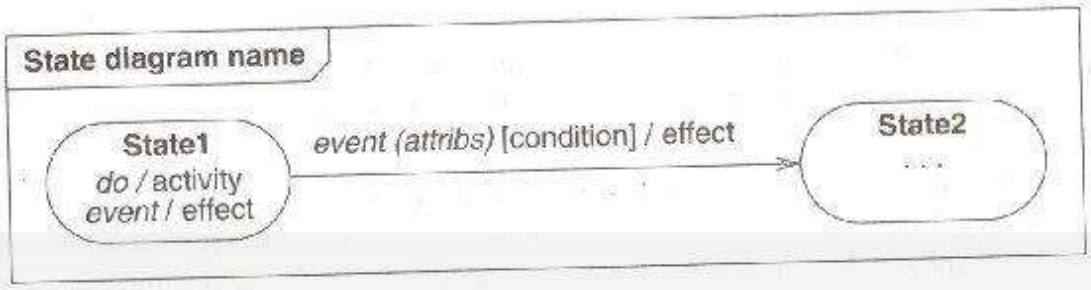


Figure 5.11 Summary of basic notation for state diagrams.

State diagram Behaviour

Activity effects

- ➔ An effect is a reference to a behavior that is executed in response to an event.
- ➔ An activity is the actual behavior that can be invoked by any number of effects.
- ➔ Eg: disconnectPhoneLine might be an activity that executed in response to an onHook event for Figure5.8.