Module 1: INTRODUCTION, MODELING CONCEPTS, CLASS MODELING:

Syllabus ---7hr

- What is object orientation?
- What is oo development?
- Oo themes
- Evidence for usefulness of oo development
- Oo modeling history
- Modeling
- Abstraction
- The tree models
- Objects and class concepts
- Link and association concepts
- Generalization and inheritance
- A sample class model
- Navigation of class models
- Practical tips

INTRODUCTION

Note 1:

Intention of this subject (object oriented modeling and design) is to learn how to apply object -oriented concepts to all the stages of the software development life cycle.

Note 2:

Object-oriented modeling and design is a way of thinking about problems using models organized around real world concepts. The fundamental construct is the object, which combines both data structure and behavior.

WHAT IS OBJECT ORIENTATION?

Definition: OO means that we organize software as a collection of discrete objects (that incorporate both data structure and behavior).

There are four**aspects (characteristics)** required by an OO approacho

Identity.

- Classification.
- Inheritance.
- Polymorphism.

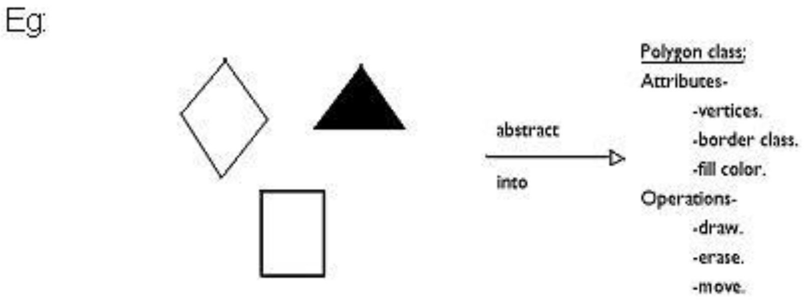
Identity:

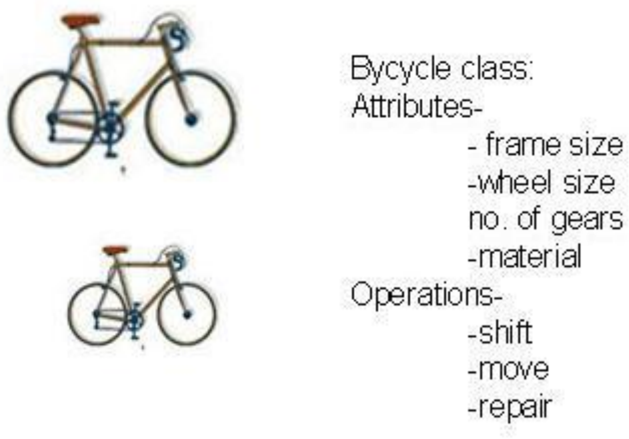
- **Identity** means that data is quantized into discrete, distinguishable entities called objects.
- **E.g. for objects:** personal computer, bicycle, queen in chess etc.

- Objects can be concrete (such as a file in a file system) or conceptual (such as scheduling policy in a multiprocessing OS). Each object has its own inherent identity. (i.e two objects are distinct even if all their attribute values are identical).
- In programming languages, an object is referenced by a unique handle.

Classification:

- **Classification** means that objects with the same data structure (attribute) and behavior (operations) are grouped into a class.
- E.g. paragraph, monitor, chess piece.
- Each object is said to be an instance of its class.
- Fig below shows objects and classes: Each class describes a possibly infinite set of individual objects.





Inheritance:

- It is the sharing of attributes and operations (features) among classes based on a hierarchical relationship. A super class has general information that sub classes refine and elaborate.
- E.g. Scrolling window and fixed window are sub classes of window.

Polymorphism:

- **Polymorphism** means that the same operation may behave differently for different classes.
- For E.g. move operation behaves differently for a pawn than for the queen in a chess game.

Note: An *operation* is a procedure/transformation that an object performs or is subjected to. An implementation of an operation by a specific class is called a *method*.

WHAT IS OO DEVELOPMENT?

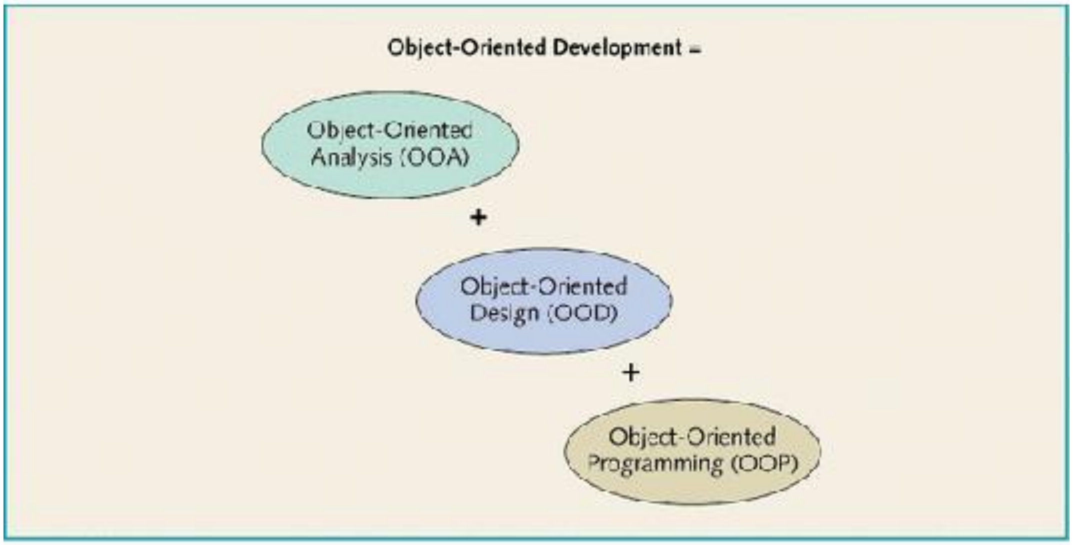

Object-Oriented Development =
Object-Oriented Analysis (OOA)
+
Object-Oriented Design (OOD)
+
Object-Oriented Programming (OOP)

Figure 1-3 Object-oriented development

Development refers to the software life cycle: Analysis, Design and Implementation. The essence of OO Development is the *identification* and *organization* of application concepts, rather than their final representation in a programming language. It's a conceptual process independent of programming languages. OO development is fundamentally a way of thinking and not a programming technique.

OO methodology

Here we present a process for OO development and a graphical notation for representing OO concepts. The process consists of building a model of an application and then adding details to it during design.

The methodology has the following stages

- **System conception:** Software development begins with business analysis or users conceiving an application and formulating tentative requirements.
- **Analysis:** The analyst scrutinizes and rigorously restates the requirements from the system conception by constructing models. The analysis model is a concise, precise abstraction of what the desired system must do, not how it will be done.
- The analysis model has two parts-

- **Domain Model**- a description of real world objects reflected within the system.
- **Application Model**- a description of parts of the application system itself that are visible to the user.
- E.g. In case of stock broker application-
- Domain objects may include- stock, bond, trade & commission.
- Application objects might control the execution of trades and present the results.
- **System Design:** The development teams devise a high-level strategy- The System Architecture- for solving the application problem. The system designer should decide what performance characteristics to optimize, chose a strategy of attacking the problem, and make tentative resource allocations.
- **Class Design:** The class designer adds details to the analysis model in accordance with the system design strategy. His focus is the data structures and algorithms needed to implement each class.
- **Implementation:** Implementers translate the classes and relationships developed during class design into a particular programming language, database or hardware. During implementation, it is important to follow good software engineering practice.

Three models

We use three kinds of models to describe a system from different view points.

1. **Class Model—**for the objects in the system & their relationships.

It describes the static structure of the objects in the system and their relationships.

Class model contains class diagrams- a graph whose nodes are classes and arcs are relationships among the classes.

2. **State model—**for the life history of objects.

It describes the aspects of an object that change over time. It specifies and implements control with state diagrams-a graph whose nodes are states and whose arcs are transition between states caused by events.

3. **Interaction Model**—for the interaction among objects.

It describes how the objects in the system co-operate to achieve broader results. This model starts with use cases that are then elaborated with sequence and activity diagrams.

Use case – focuses on functionality of a system – i.e what a system does for users.

Sequence diagrams – shows the object that interact and the time sequence of their interactions.

Activity diagrams – elaborates important processing steps.

OO THEMES

Several themes pervade OO technology. Few are –

1. **Abstraction**

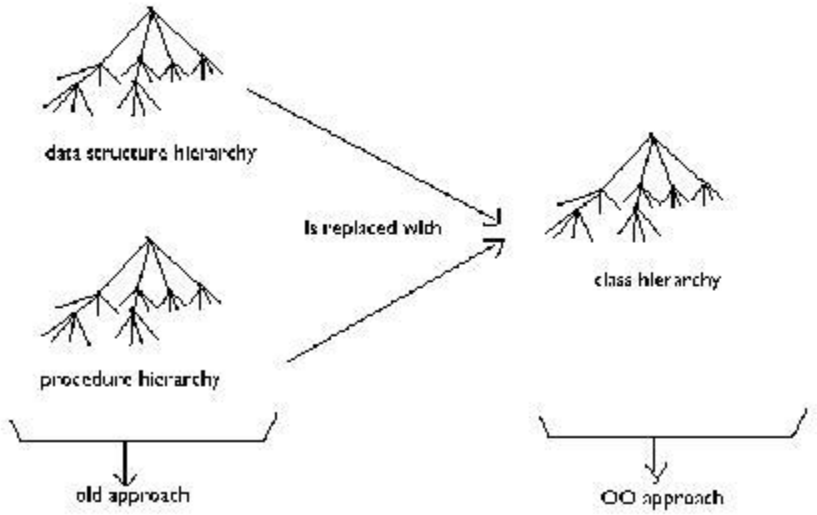
- Abstraction lets you focus on essential aspects of an application while ignoring details i.e focusing on what an object is and does, before deciding how to implement it.
- It's the most important skill required for OO development.

2. **Encapsulation (information hiding)**

- It separates the external aspects of an object (that are accessible to other objects) from the internal implementation details (that are hidden from other objects)
- Encapsulation prevents portions of a program from becoming so interdependent that a small change has massive ripple effects.

3. **Combining data and behavior**

- Caller of an operation need not consider how many implementations exist.
- In OO system the data structure hierarchy matches the operation inheritance
- hierarchy (fig).



4. Sharing

- OO techniques provide sharing at different levels.
- Inheritance of both data structure and behavior lets sub classes share common code.
- OO development not only lets you share information within an application, but also offers the prospect of reusing designs and code on future projects.

5. **Emphasis on the essence of an object**

- OO development pl.s a greater emphasis on data structure and a lesser emphasis on procedure structure than functional-decomposition methodologies.

6. **Synergy**

- Identity, classification, polymorphism and inheritance characterize OO languages.

- Each of these concepts can be used in isolation, but together they complement each other synergistically.

MODELLING AS A DESIGN TECHNIQUE

Note: A model is an abstraction of something for the purpose of understanding it before building it.

MODELLING

Designers build many kinds of models for various purposes before constructing things.

Models serve several purposes–

- **Testing a physical entity before building it:** Medieval built scale models of Gothic Cathedrals to test the forces on the structures. Engineers test scale models of airplanes, cars and boats to improve their dynamics.
- **Communication with customers:** Architects and product designers build models to show their customers (note: mock-ups are demonstration products that imitate some of the external behavior of a system).
- **Visualization:** Storyboards of movies, TV shows and advertisements let writers see how their ideas flow.
- **Reduction of complexity:** Models reduce complexity to understand directly by separating out a small number of important things to do with at a time.

ABSTRACTION

Abstraction is the selective examination of certain aspects of a problem.

The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant.

THE THREE MODELS

1. **Class Model**: represents the static, structural, "data" aspects of a system.

- It describes the structure of objects in a system- their identity, their relationships to other objects, their attributes, and their operations.
- Goal in constructing class model is to capture those concepts from the real world that are important to an application.
- Class diagrams express the class model.

2. **State Model**: represents the temporal, behavioral, "control" aspects of a system.

- State model describes those aspects of objects concerned with time and the sequencing of operations – events that mark changes, states that define the context for events, and the organization of events and states.
- State diagram express the state model.
- Each state diagram shows the state and event sequences permitted in a system for one class of objects.
- State diagram refer to the other models.

- Actions and events in a state diagram become operations on objects in the class model. References between state diagrams become interactions in the interaction model.

3. **Interaction model** – represents the collaboration of individual objects, the "interaction" aspects of a system.

- Interaction model describes interactions between objects – how individual objects collaborate to achieve the behavior of the system as a whole.
- The state and interaction models describe different aspects of behavior, and you need both to describe behavior fully.
- Use cases, sequence diagrams and activity diagrams document the interaction model.

CLASS MODELLING

Note: A class model captures the static structure of a system by characterizing the objects in the system, the relationships between the objects, and the attributes and operations for each class of objects.

OBJECT AND CLASS
CONCEPT Objects

Purpose of class modeling is todescribe objects.

An**object** is a concept, abstraction or thing with identity that has meaning for an application.
Ex: Joe Smith, Infosys Company, process number 7648 and top window are objects.

Classes

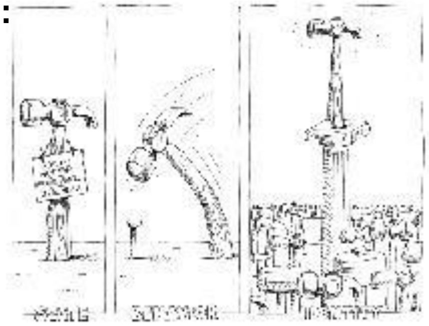
An object is an instance or occurrence of a class.

A**class** describes a group of objects with the same properties (attributes), behavior (operations), kinds of relationships and semantics.
Ex: Person, company, process and window are classes.

Note: All objects have identity and are distinguishable. Two apples with same color, shape and texture are still individual apples: a person can eat one and then the other. The term identity means that the objects are distinguished by their inherent existence and not by descriptive properties that they may have.

CLASS MODELLING

- **OBJECT AND CLASS CONCEPT**
- An **object** has three characteristics: **state**, **behavior** and **a unique identification**. or
- **An object is a concept, abstraction or thing with identity that has meaning for an application. Eg:**
- Note: The term identity means that the objects are distinguished by their inherent existence and not by descriptive properties that they may have.



Class diagrams

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

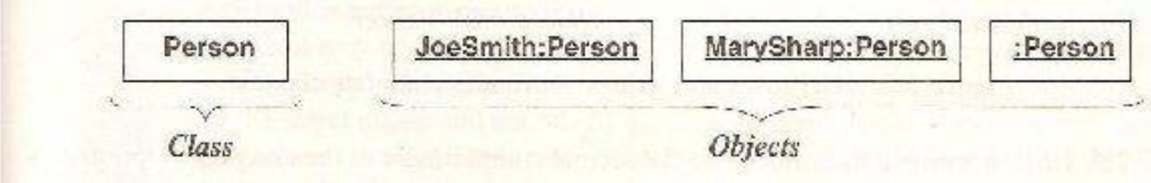
Note: An object diagram shows individual objects and their relationships.

Useful for documenting test cases and discussing examples.

Class diagrams are useful both for abstract modeling and for designing actual programs.

Note: A class diagram corresponds to infinite set of object diagrams.

Figure below shows a class (left) and instances (right) described by it.



Conventions used (UML):

- UML symbol for both classes and objects is box.
- Objects are modeled using box with object name followed by colon followed by class name.
- Use boldf. to list class name, center the name in the box and capitalize the first letter. Use singular nouns for names of classes.
- To run together multiword names (such as JoeSmith), separate the words

with

- intervening capital letter.

Values and Attributes:

Value is a piece of data.

Attribute is a named property of a class that describes a value held by each object of the class.

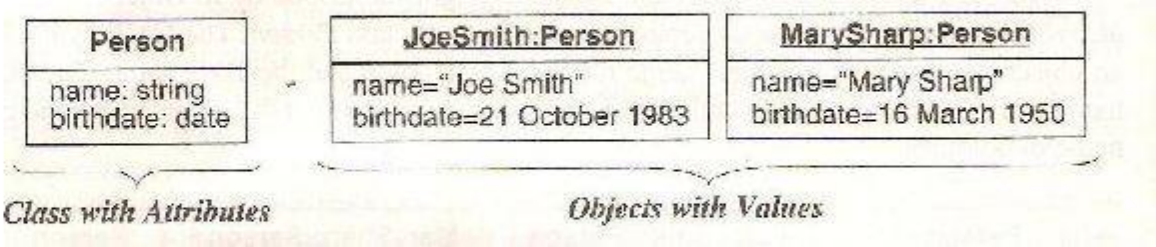
Following analogy holds:

Object is to class as value is to attribute.

E.g. Attributes: Name, bdate, weight.

Values: JoeSmith, 21 October 1983, 64. (Of person object).

Fig shows modeling notation



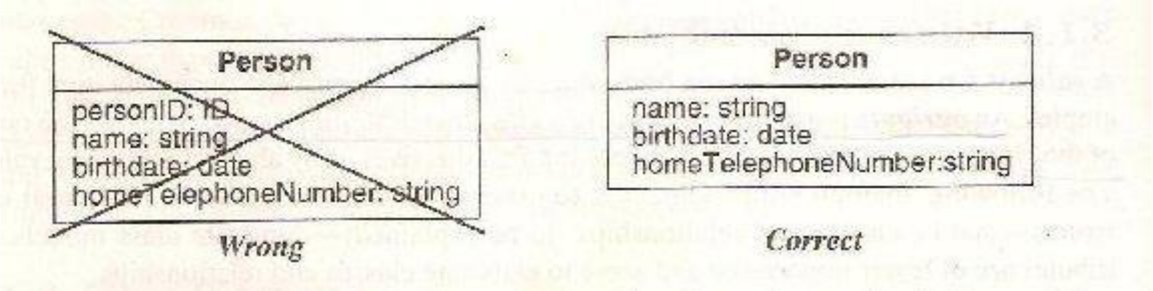
Conventions used (UML):

- List attributes in the 2nd compartment of the class box. Optional details (like default value) may follow each attribute.
- A colon precedes the type, an equal sign precedes default value.
- Show attribute name in regular f., left align the name in the box and use small case for the first letter.

Similarly we may also include attribute values in the 2nd compartment of object boxes with same conventions.

Note: Do not list object identifiers; they are implicit in models.

E.g.



An **operation** is a function or procedure that maybe applied to or by objects in a class.

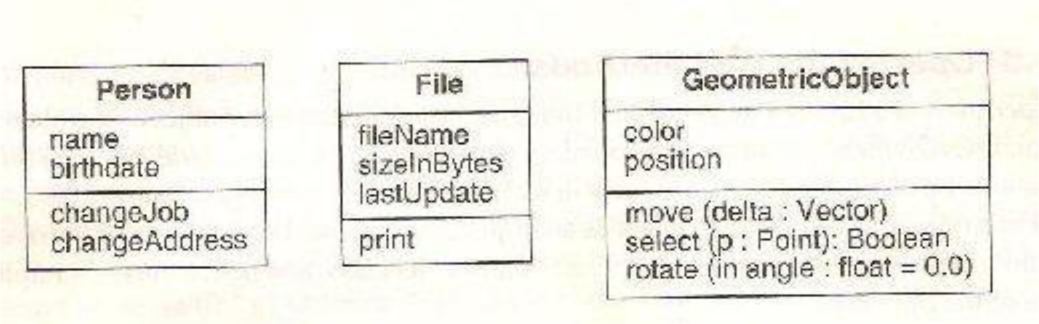
E.g. Hire, fire and pay dividend are operations on Class Company. Open, close, hide and redisplay are operations on class window.

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

E.g. In class file, print is an operation you could implement different methods to print files.

Note: Same operation may apply to many different classes. Such an operation is polymorphic.

Fig shows modeling notation.



UML conventions used –

- List operations in 3rd compartment of class box.
- List operation name in regular f., left align and use lower case for first letter.
- Optional details like argument list and return type may follow each operation name.
- Parenthesis enclose an argument list, commas separate the arguments. A colon precedes the result type.

Note: We do not list operations for objects, because they do not vary among objects of same class.

Summary of Notation for classes

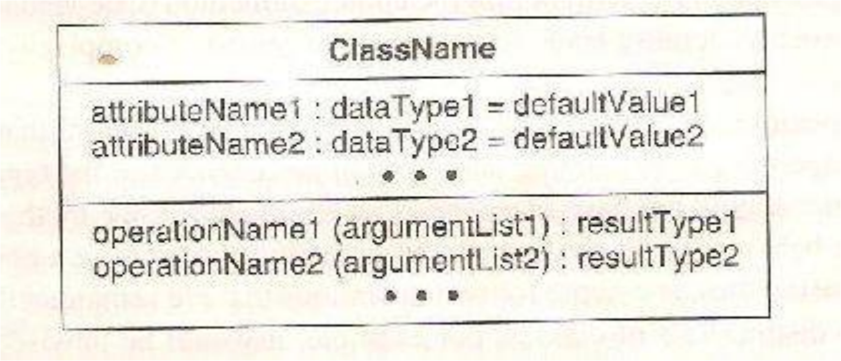

ClassName
attributeName1 : dataType1 = defaultValue1
attributeName2 : dataType2 = defaultValue2
• • •
operationName1 (argumentList1) : resultType1
operationName2 (argumentList2) : resultType2
• • •

Fig: Summary of modeling notation for classes

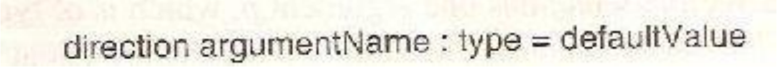
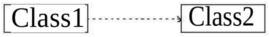

direction argumentName : type = defaultValue

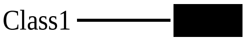
Fig: Notation for an argument of an operation

Class Digarms: Relationships

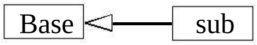
- Classes can related to each other through different relationships:
  - Dependency



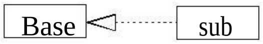
  - Association (delegation)



  - Generalization (inheritance)

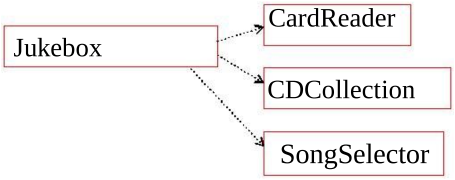


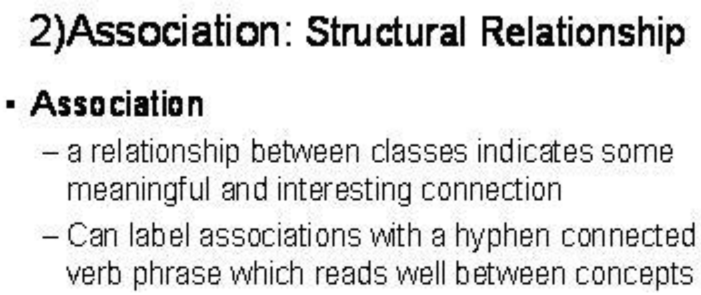
  - Realization (interf.s)



1) Dependency: A Uses Relationship

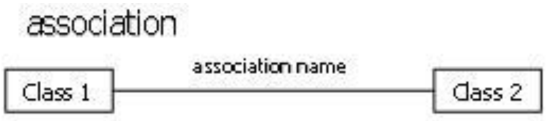
- Dependencies
  - occurs when one object depends on another
  - if you change one object's interf., you need to change the dependent object
  - arrow points from dependent to needed objects





2)Association: Structural Relationship

- **Association**
  - a relationship between classes indicates some meaningful and interesting connection
  - Can label associations with a hyphen connected verb phrase which reads well between concepts



if association name is replaced with "owns>",
it would read "Class 1 owns Class 2"

LINK AND ASSOCIATION CONCEPTS

Note: Links and associations are the means for establishing relationships among objects and classes.

Links and associations

A**link** is a physical or conceptual connection among objects.

E.g. JoeSmith *WorksFor* Simplex Company.

Mathematically, we define a link as a tuple– that is, a list of objects.

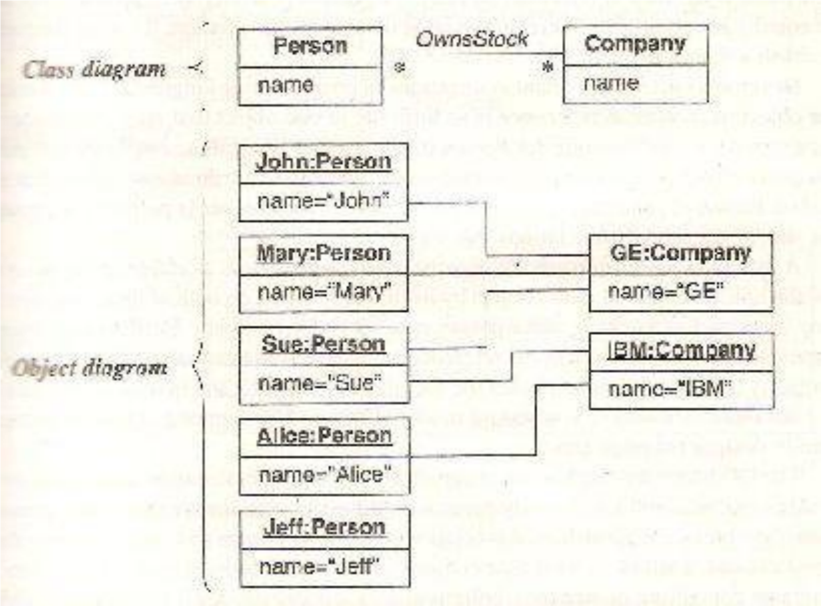
A link is an instance of an**association**.

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

E.g. a person *WorksFor* a company.

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

Fig shows many-to-many association (model for a financial application).



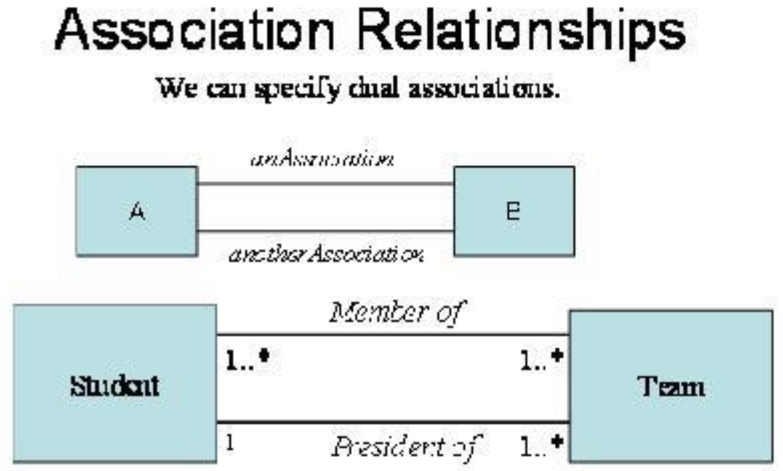
Conventions used (UML):

- Link is a line between objects; a line may consist of several line segments.
- If the link has the name, it is underlined.
- Association connects related classes and is also denoted by a line.
- Show link and association names in italics.

Note:

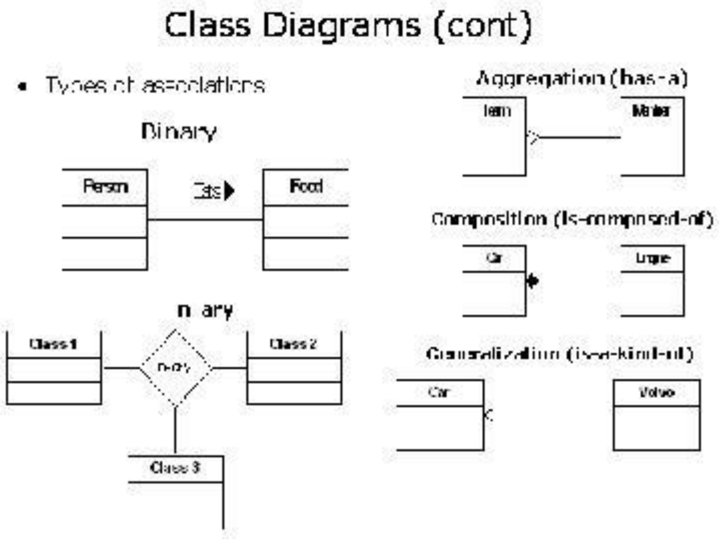
- Association name is optional, if the model is unambiguous. Ambiguity arises when a model has multiple associations among same classes.

- Developers often implement associations in programming languages as references from one object to another. A reference is an attribute in one object that refers to another object.



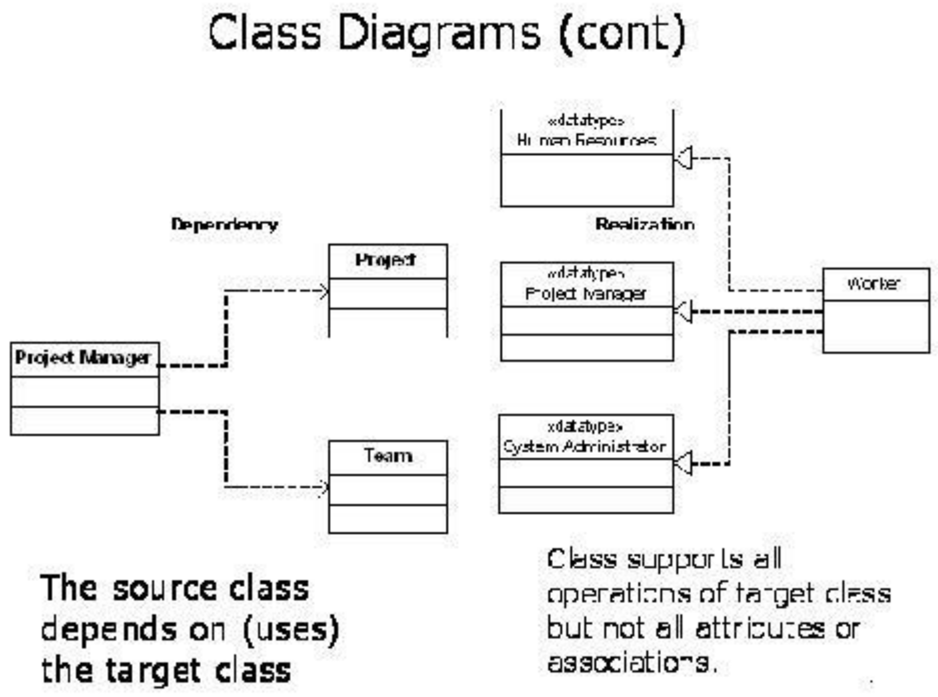
Association Relationships

We can specify dual associations.



Class Diagrams (cont)

- Types of associations



Class Diagrams (cont)

The source class depends on (uses) the target class

Class supports all operations of target class but not all attributes or associations.

Multiplicity

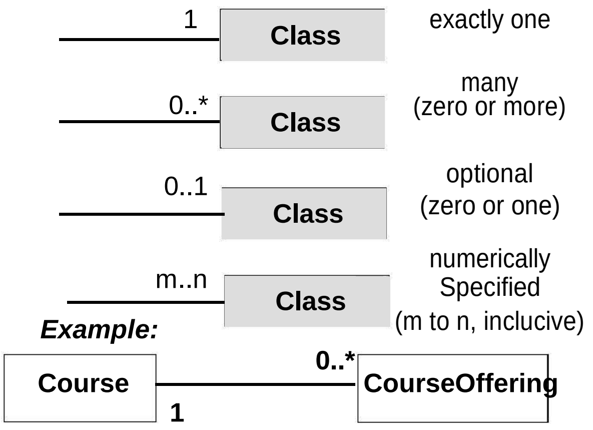
Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class. Multiplicity constrains the number of related objects.

UML conventions:

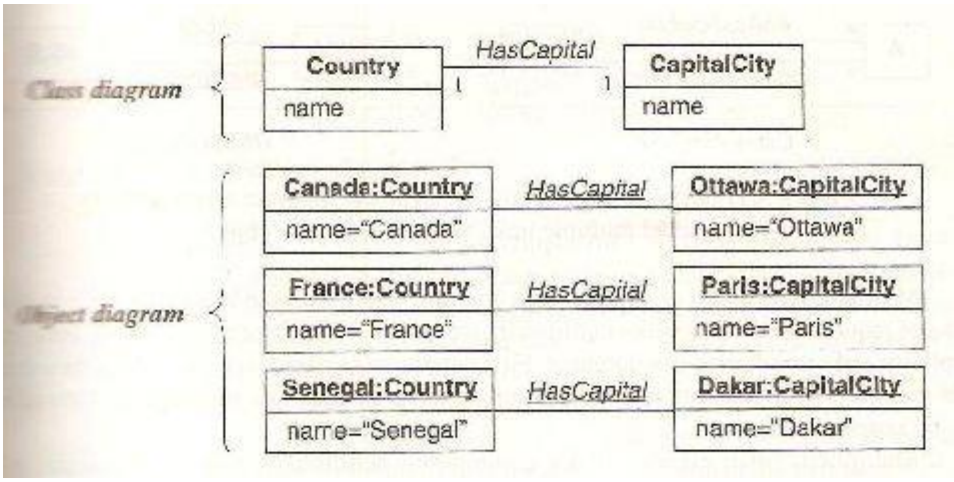
- UML diagrams explicitly lists multiplicity at the ends of association lines.
- UML specifies multiplicity with an interval, such as

“1” (exactly one).
“1..”(one or more).
“3..5”(three to five, inclusive).
“ * ” (many, i.e zero or more).

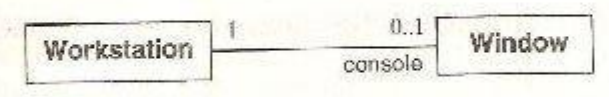
- notations



Previous figure illustrates many-to-many multiplicity. Below figure illustrates **one-to-one multiplicity.**



Below figure illustrates zero-or-one multiplicity.



Note 1: Association vs Link.

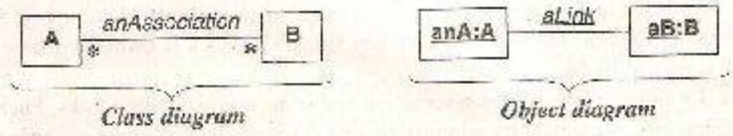


Figure 3.10 Association vs. link. A pair of objects can be instantiated at most once per association (except for bags and sequences).

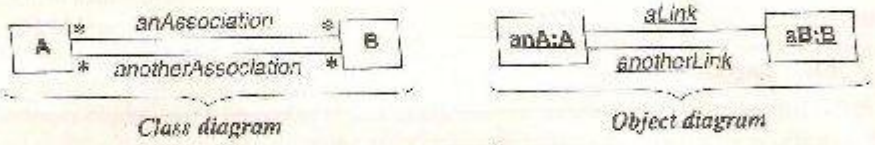
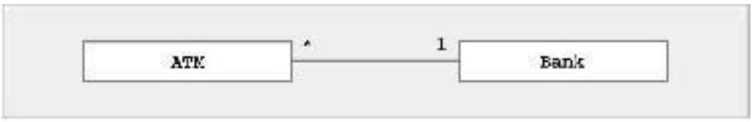


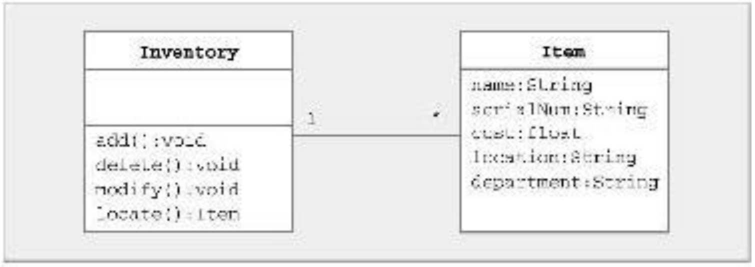
Figure 3.11 Association vs. link. You can use multiple associations to model multiple links between the same objects.

Multiplicity of Associations

- Many-to-one
  - – Bank has many ATMs, ATM knows only 1 bank

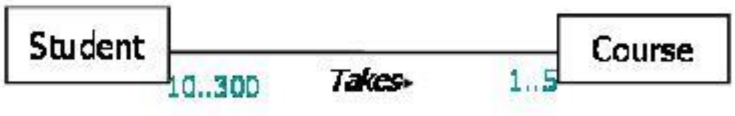


- One-to-many
  - – Inventory has many items, items know 1 inventory



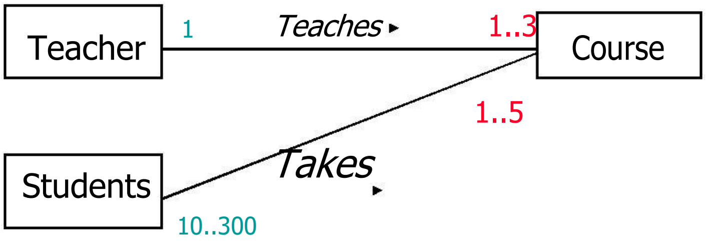
Association - Multiplicity

- A Student can take up to five Courses.
- Student has to be enrolled in at least one course.
- Up to 300 students can enroll in a course.
- A class should have at least 10 students.



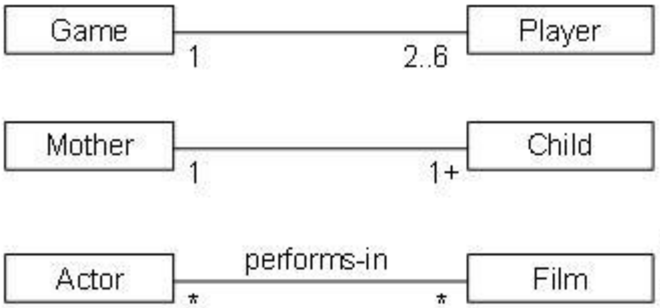
Association ‾ - Multiplicityli ity

- A teacher teaches 1 to 3 courses (subjects)
- Each course is taught by only one teacher.
- A student can take between 1 to 5 courses.
- A course can have 10 to 300 students.



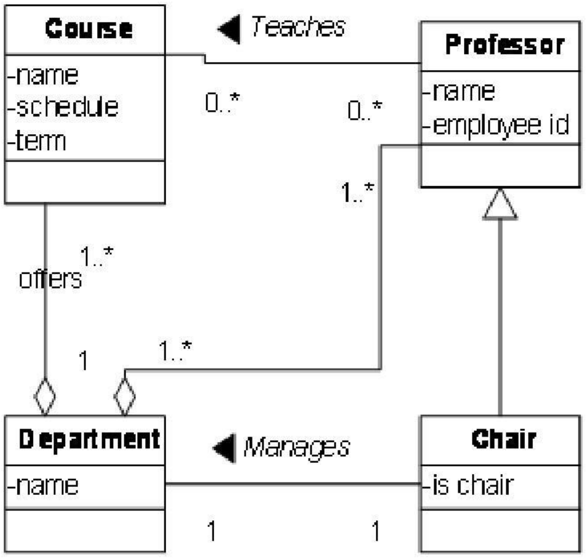
Multiplicity

- Multiplicity defines how many instances of type A can be associated with one instance of type B at some point

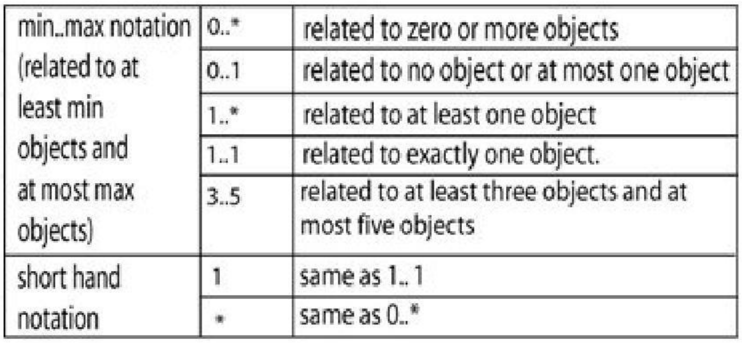


Actor is associated with 0 to many films
A film is associated with 0 to many actors

can label associations



MULTIPLICITIES IN ASSOCIATIONS



min..max notation (related to at least min objects and at most max objects)	0..*	related to zero or more objects
	0..1	related to no object or at most one object
	1..*	related to at least one object
	1..1	related to exactly one object.
	3..5	related to at least three objects and at most five objects
short hand notation	1	same as 1.. 1
	*	same as 0..*

Note 2: Multiplicity vs Cardinality.

- Multiplicity is a constraint on the size of a collection.
- Cardinality is a count of elements that are actually in a collection. Therefore, multiplicity is a constraint on cardinality.

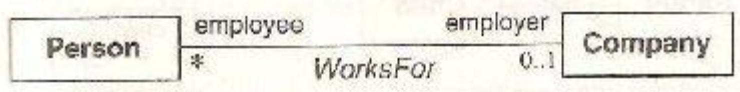
Note 3: The literature often describes multiplicity as being "one" or "many", but more generally it is a subset of the non negative numbers.

Association end names

Multiplicity implicitly refers to the ends of associations. For E.g. A one-to-many association has two ends –

- an end with a multiplicity of "one"
- an end with a multiplicity of "many"

You can not only assign a multiplicity to an association end, but you can give it a name as well.



employee	employer
Joe Doe	Simplex
Mary Brown	Simplex
Jean Smith	United Widgets

Association end names. Each end of an association can have a name.

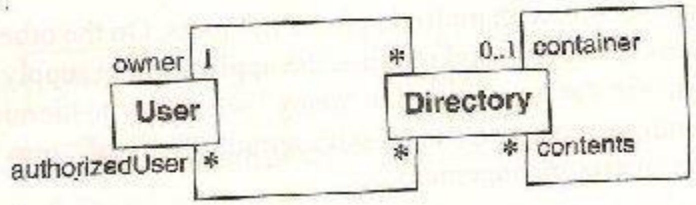
A person is an employee with respect to company.

A company is an employer with respect to a person.

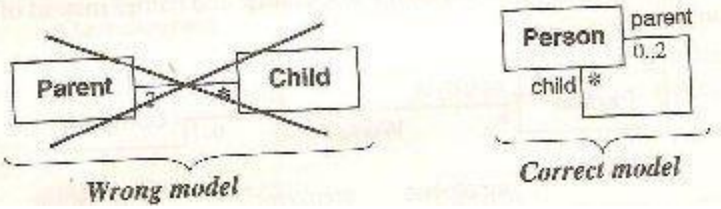
Note 1: Association end names are optional.

Note 2: Association end names are necessary for associations between two objects of the same class. They can also distinguish multiple associations between a pair of classes.

E.g. each directory has exactly one user who is an owner and many users who are authorized to use the directory. When there is only a single association between a pair of distinct classes, the names of the classes often suffice, and you may omit association end names.



Note 3: Association end names let you unify multiple references to the same class. When constructing class diagrams you should properly use association end names and not introduce a separate class for each reference as below fig shows.



Wrong model *Correct model*

Sometimes, the objects on a “many” association end have an explicit order.

E.g. Workstation screen containing a number of overlapping windows. Each window on a screen occurs at most once. The windows have explicit order so only the top most windows are visible at any point on the screen.

Ordering is an inherent part of association. You can indicate an ordered set of objects by writing “{ordered}” next to the appropriate association end.

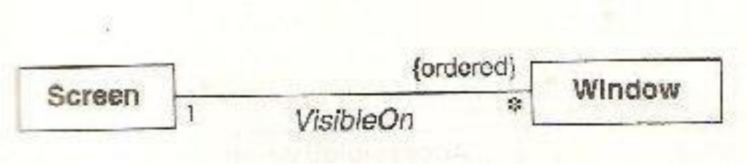


Fig: ordering sometimes occurs for “many” multiplicity

Bags and Sequences

Normally, a binary association has **at most one link** for a pair of objects.

However, you can permit**multiple links** for a pair of objects by annotating an association end with {bag} or {sequence}.

A**bag** is a collection of elements with duplicates allowed.

A**sequence** is an ordered collection of elements with duplicates allowed.

Example:

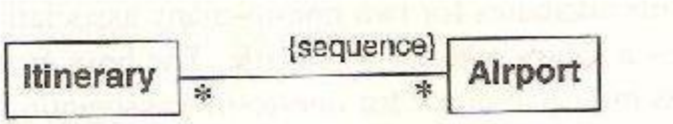


fig: an itinerary may visit multiple airports, so you should use {sequence} and not {ordered}

Note: {ordered} and {sequence} annotations are same, except that the first disallows duplicates and the other allows them.

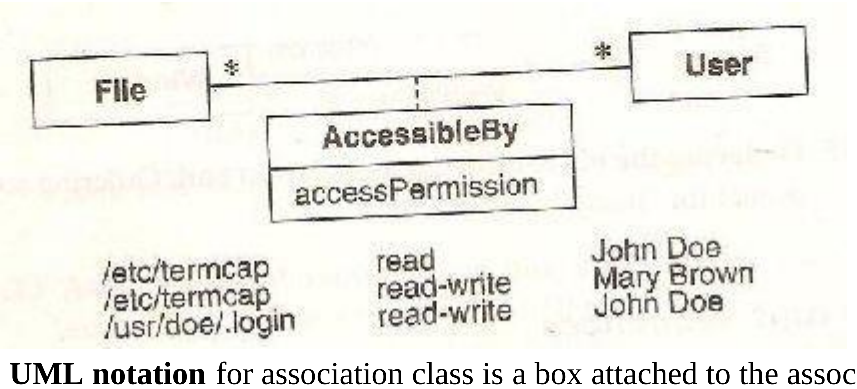
Association classes

An**association class** is an association that is also a class.

Like the links of an association, the instances of an association class derive identity from instances of the constituent classes.

Like a class, an association class can have attributes and operations and participate in associations.

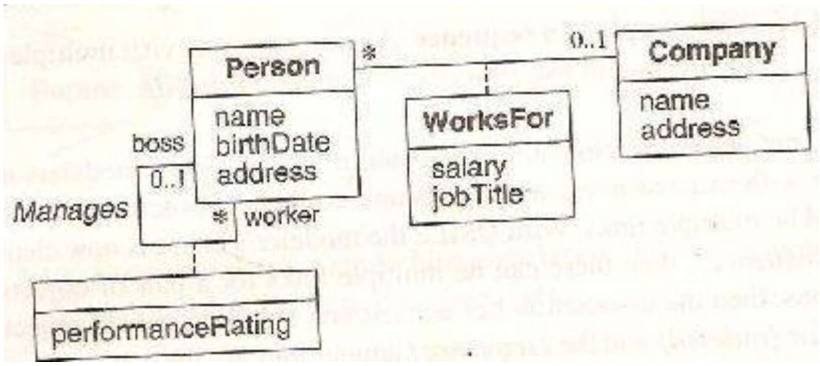
Ex:



UML notation for association class is a box attached to the association by a dashed line.

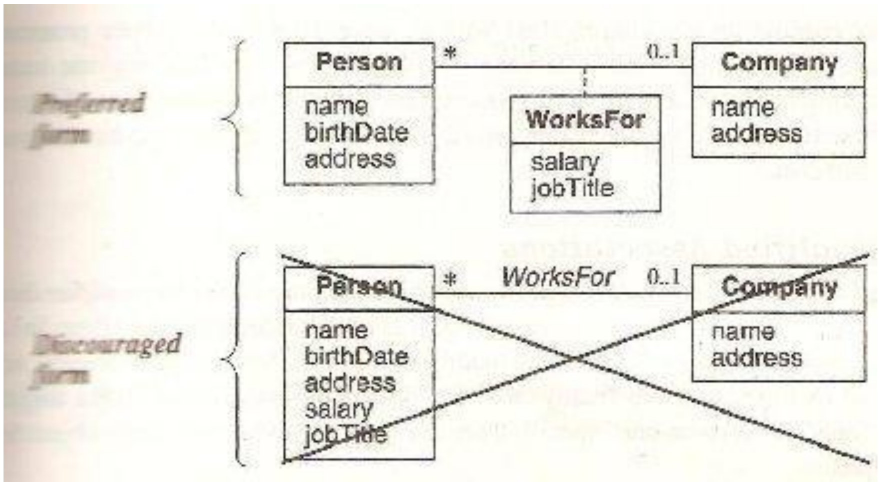
Note: Attributes for association class unmistakably belong to the link and cannot be ascribed to either object. In the above figure, accessPermission is a joint property of File and user cannot be attached to either file or user alone without losing information.

Below figure presents attributes for two one-to-many relationships. Each person working for a company receives a salary and has job title. The boss evaluates the performance of each worker. Attributes may also occur for one-to-one associations.

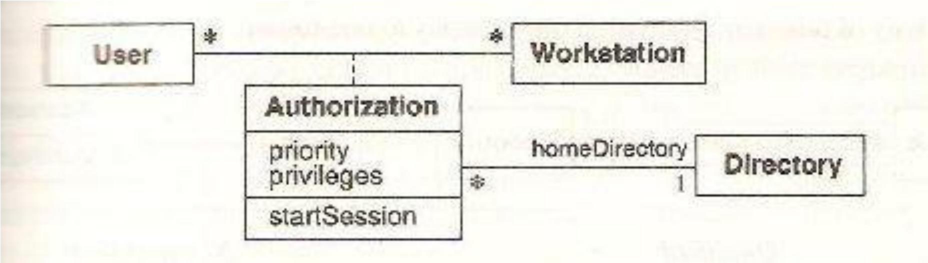


Note 1: Figure shows how it's possible to fold attributes for one-to-one and one-to-many associations into the class opposite a "one" end. This is not possible for many-to-many associations.

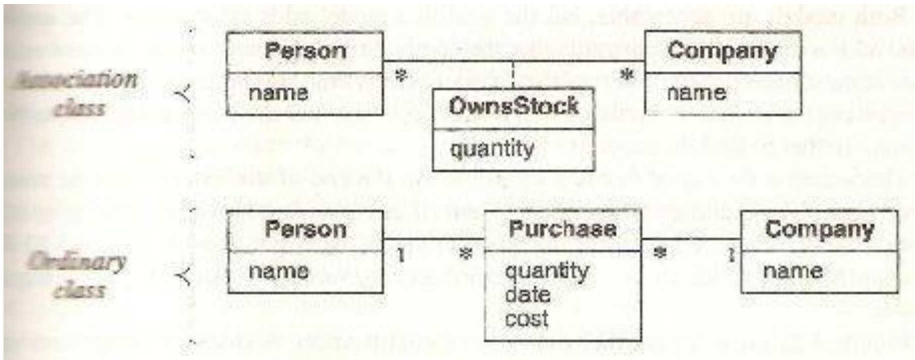
As a rule, you should not fold such attributes into a class because the multiplicity of the association may change.



Note 2: An association class participating in an association.



Note 3: Association class vs ordinary class.



eg:

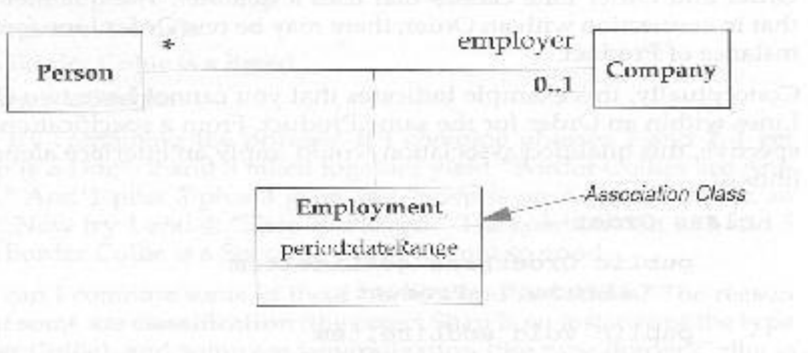
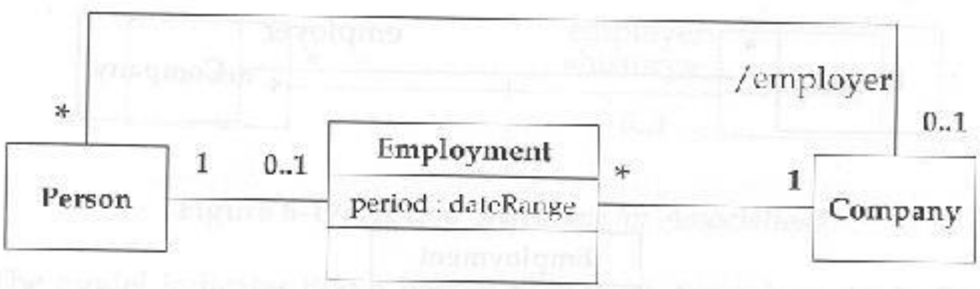


Figure 6-14: *Association Class*



Qualified associations

A **Qualified Association** is an association in which an attribute called the **qualifier** disambiguates the objects for a "many" association ends. It is possible to define qualifiers for one-to-many and many-to-many associations.

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

Ex 1: qualifier for associations with one to many multiplicity. A bank services multiple accounts. An account belongs to single bank. Within the context of a bank, the Account Number specifies a unique account. Bank and account are classes, and Account Number is a qualifier. Qualification reduces effective multiplicity of this association from one-to-many to one-to-one.

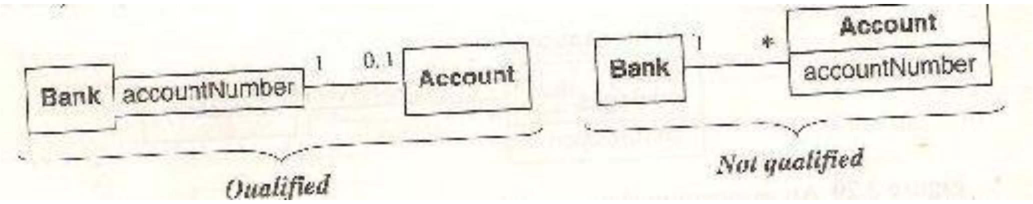
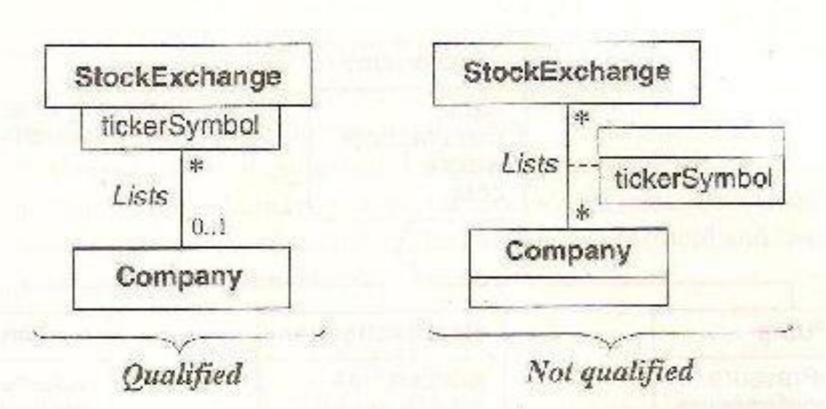
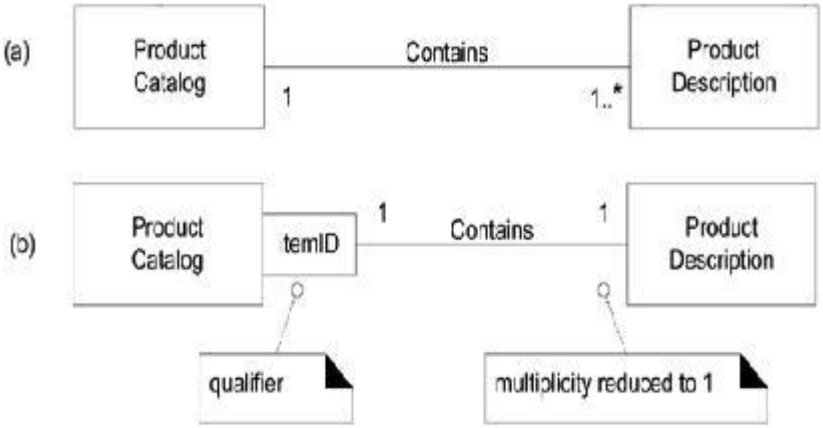


Fig: qualification increases the precision of a model. (note: however, both are acceptable)

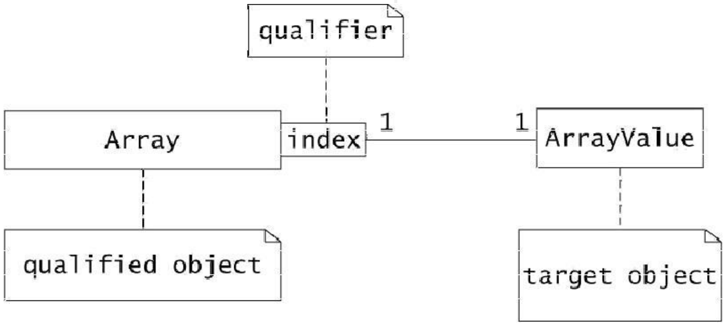
Ex 2: a stock exchange lists many companies. However, it lists only one company with a given ticker symbol. A company maybe listed on many stock exchanges, possibly under different symbols.



Eg 3: Qualified Association



eg 4:



GENERALIZATION AND INHERITANCE

Generalization is the relationship between a class (the superclass) and one or more variations of the class (the subclasses). Generalization organizes classes by their similarities and differences, structuring the description of objects.

The superclass holds common attributes, operations and associations; the subclasses add specific attributes, operations and associations. Each subclass is said to **inherit** the features of its superclass.

There can be**multiple levels** of generalization.

Fig(a) and Fig(b) (given in the following page) shows examples of generalization.

Fig(a) – Example of generalization for equipment.

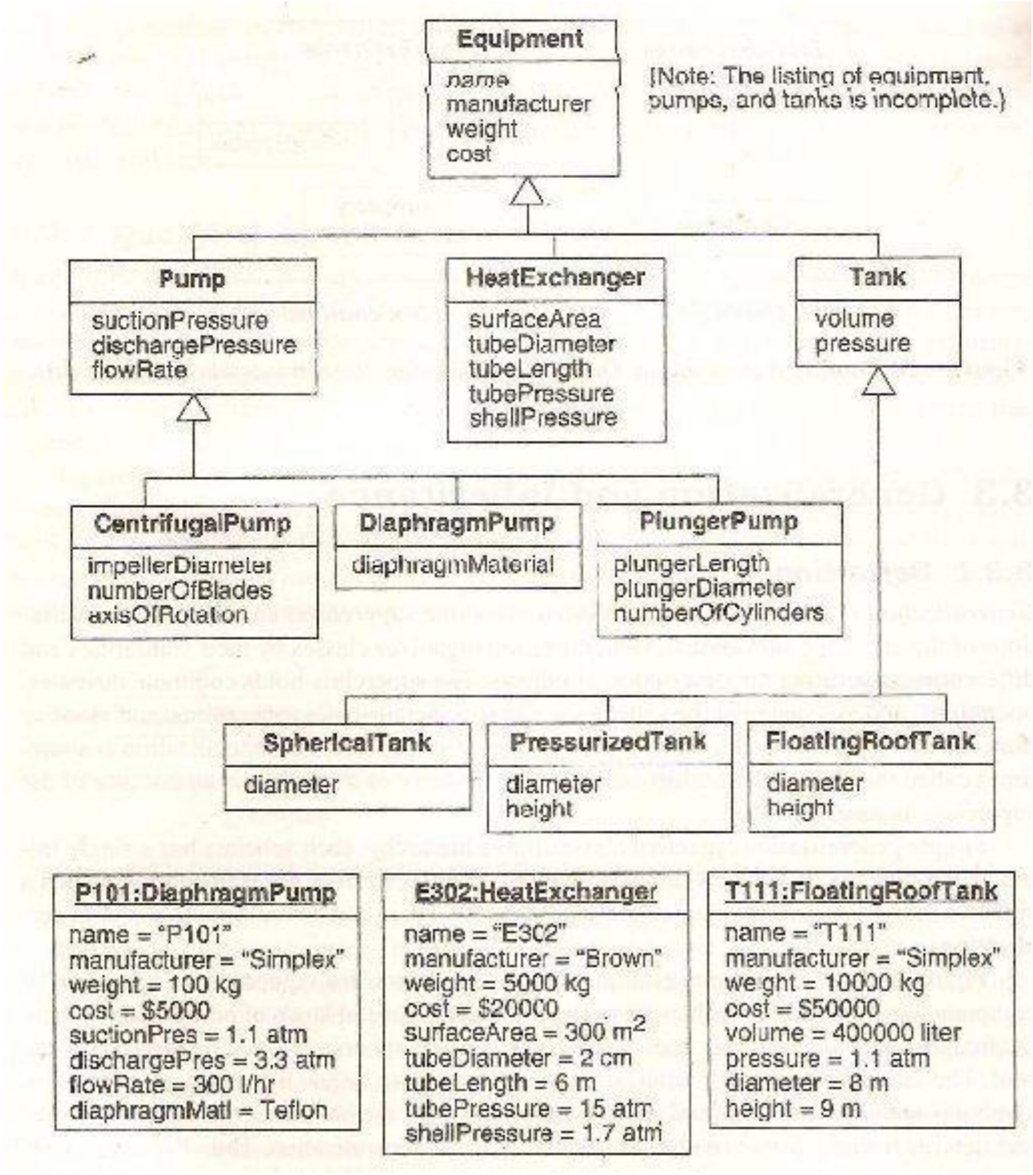
Each object inherits features from one class at each level of generalization.

UML convention used:

Use large hollow arrowhead to denote generalization. The arrowhead points to superclass.

Fig(b) – inheritance for graphic figures.

The word written next to the generalization line in the diagram (i.e dimensionality) is a generalization set name. A generalization set name is an enumerated attribute that indicates which aspect of an object is being abstracted by a particular generalization. It is optional.


Equipment
name
manufacturer
weight
cost
(Note: The listing of equipment, pumps, and tanks is incomplete.)
Pump
suctionPressure
dischargePressure
flowRate
HeatExchanger
surfaceArea
tubeDiameter
tubeLength
tubePressure
shellPressure
Tank
volume
pressure
CentrifugalPump
impellerDiameter
numberOfBlades
axisOfRotation
DiaphragmPump
diaphragmMaterial
PlungerPump
plungerLength
plungerDiameter
numberOfCylinders
SphericalTank
diameter
PressurizedTank
diameter
height
FloatingRoofTank
diameter
height
P101:DiaphragmPump
name = "P101"
manufacturer = "Simplex"
weight = 100 kg
cost = $5000
suctionPres = 1.1 atm
dischargePres = 3.3 atm
flowRate = 300 l/hr
diaphragmMatl = Teflon
E302:HeatExchanger
name = "E302"
manufacturer = "Brown"
weight = 5000 kg
cost = $20000
surfaceArea = 300 m²
tubeDiameter = 2 cm
tubeLength = 6 m
tubePressure = 15 atm
shellPressure = 1.7 atm
T111:FloatingRoofTank
name = "T111"
manufacturer = "Simplex"
weight = 10000 kg
cost = $50000
volume = 400000 liter
pressure = 1.1 atm
diameter = 8 m
height = 9 m

Fig(a)

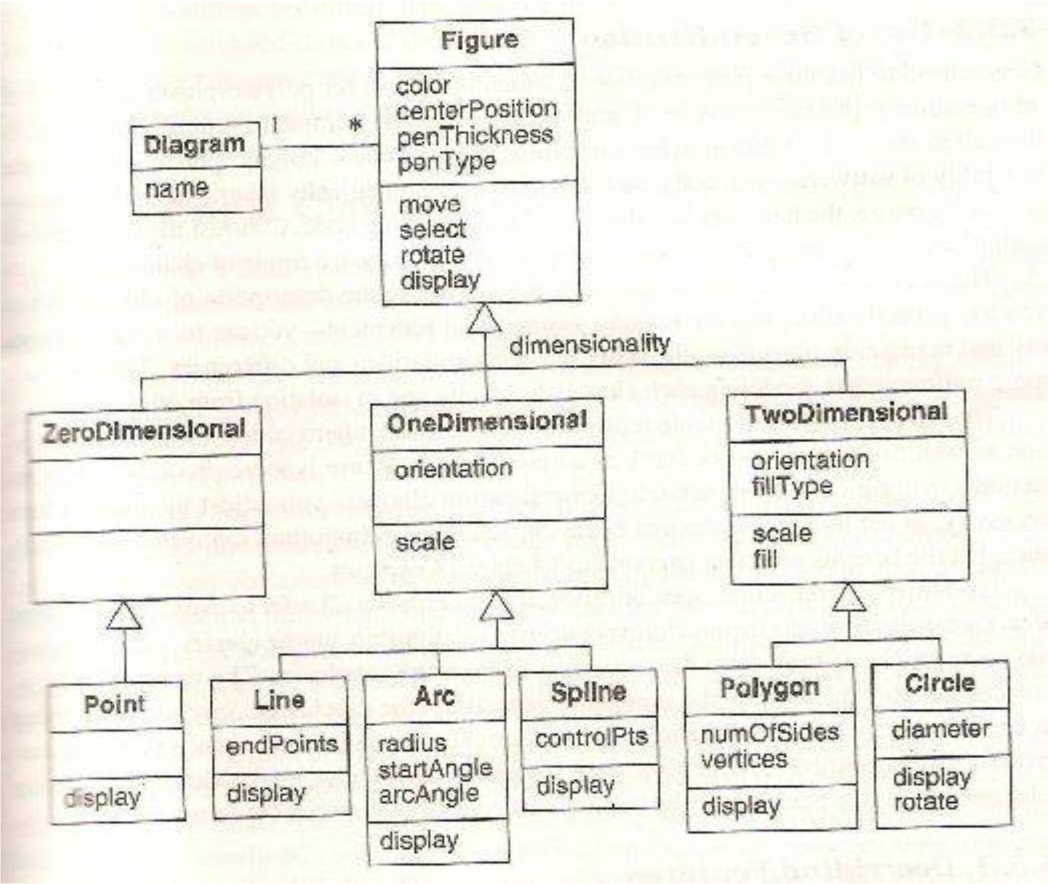


Fig (b)

'move**'**, **'**select**'**, 'rotate', and 'display' are operations that all subclasses inherit.

'scale**'** applies to one-dimensional and two-dimensional figures.

'fill' applies only to two-dimensional figures.

Use of generalization: Generalization has three purposes –

1. **To support polymorphism:** You can call an operation at the superclass level, and the OO language complier automatically resolves the call to the method that matches the calling object's class.
2. **To structure the description of objects:** i.e to frame a taxonomy and organizing objects on the basis of their similarities and differences.
3. **To enable reuse of code:** Reuse is more productive than repeatedly writing code from scratch.

Note: The terms generalization, specialization and inheritance all refer to aspects of the same idea.

Overriding features

A subclass may override a superclass feature by defining a feature with the same name. The overriding feature (subclass feature) refines and repl.s the overridden feature (superclass feature) .

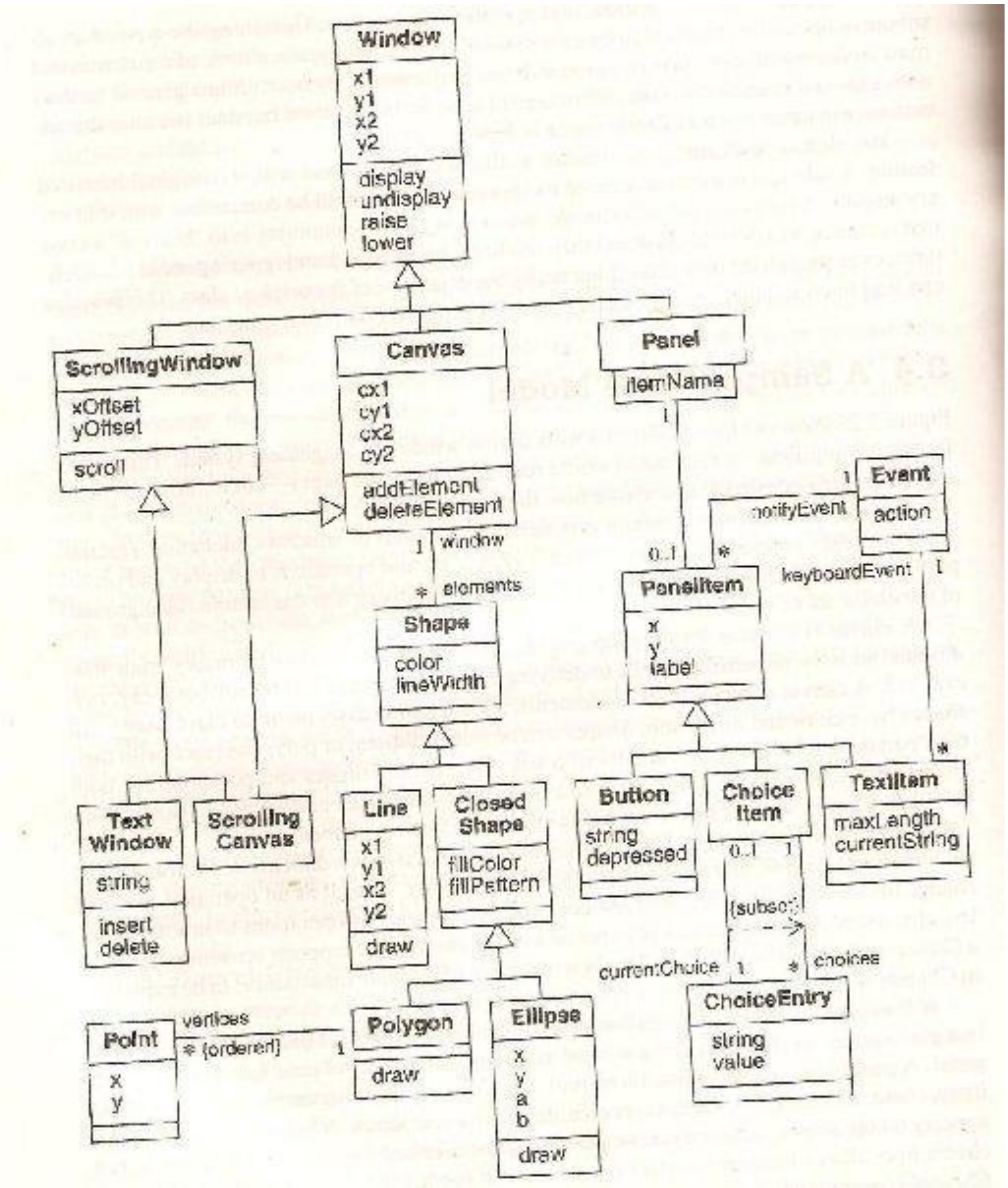
Why override feature?

• To specify behavior that depends on subclass. •
To tighten the specification of a feature.

- To improve performance.

In fig(b) (previous page) each leaf subclasses had overridden ‘display’ feature. **Note:** You may override methods and default values of attributes. You should never override the signature, or form of a feature.

A SAMPLE CLASS MODEL



NAVIGATION OF CLASS MODELS

Class models are useful for more than just data structure. In particular, **navigation of class model** lets you express certain behavior. Furthermore, navigation exercises a class model and uncovers hidden flaws and omission, which you can then repair.

UML incorporates a language that can be used for navigation, the **object constraint language(OCL).**

OCL constructs for traversing class models

OCL can traverse the constructs in class models.

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

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

   Ex: aCreditCardAccount.maximumcredit

2. **Operations: Y**ou can also invoke an operation for an object or collection of objects. Syntax: source object or object collection, followed by dot and then the operation followed by parenthesis even if it has no arguments. OCL has special operations that operate on entire collections (as opposed to operating on each object in a collection). Syntax for collection operation is: source object collection followed by "->", followed by the operation.

3. **Simple associations:** Dot notation is also used to traverse an association to a target end. Target end maybe indicated by an association end name, or class name (if there is no ambiguity).

   Ex: refer fig in next page.

   - aCustomer.MailingAddress yields a set of addresses for a customer (the target end has "many" multiplicity).
   - aCreditCardAccount.MailingAddress yields a single address(the target end has multiplicity of "one").

4. **Qualified associations:** The expression aCreditCardAccount.Statement [30 November 1999] finds the statement for a credit card account with the statement date of November 1999. The syntax is to enclose the qualifier value in brackets.

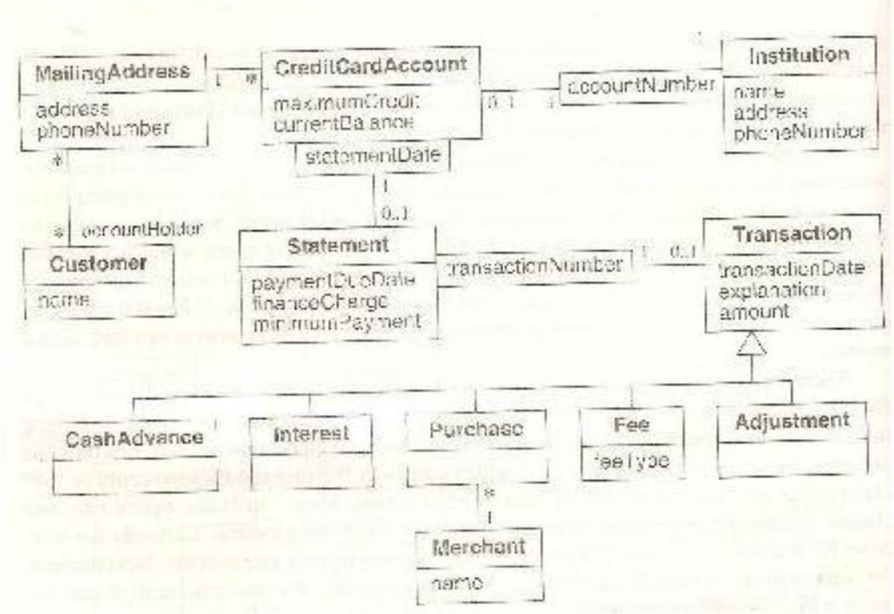
5. **Associations classes:** Given a link of an association class, you can find the constituent objects and vice versa.

6. **Generalization:** Traversal of a generalization hierarchy is implicit for the OCL notation.

7. **Filters:** Most common filter is 'select' operation.

   Ex: aStatement.Transaction->select(amount>$100).

Examples of OCL expressions



Write an OCL expression for–

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

Soln: aCreditCardAccount.Statement.Transaction -> select(aStartDate<=TransactionDate and TransactionDate<=anEndDate)

2. **What volumes of transactions were handled by an institution in the last year?**

Soln: anInstitution.CreditCardAccount.Statement.Transaction -> select(aStartDate<=TransactionDate and TransactionDate<=anEndDate).amount->sum( )

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

Soln: aMerchant.Purchase -> select(aStartDate<=TransactionDate andtransactionDate<=anEndDate).Statement.CreditCardAccount.MailingAddress.Customer ->asset( )

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

Soln: aCustomer.MailingAddress.CreditCardAccount -> size( )

5. **What is the total maximum credit for a customer for all accounts?** Soln: acustomer.MailingAddress.CreditCardAccount.Maximumcredit -> sum( )

Unit 2: Advanced Class Modeling **6 Hours**

Toipics :

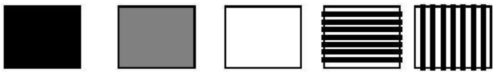
- ***Advanced object and class concepts***
- ***Asoociation ends***
- ***N-ary association***
- ***Aggregation***
- ***Abstract classes***
- ***Multiple inheritance***
- ***Metadata***
- ***Reification***
- ***Constraints***
- ***Derived data***
- ***Packages***

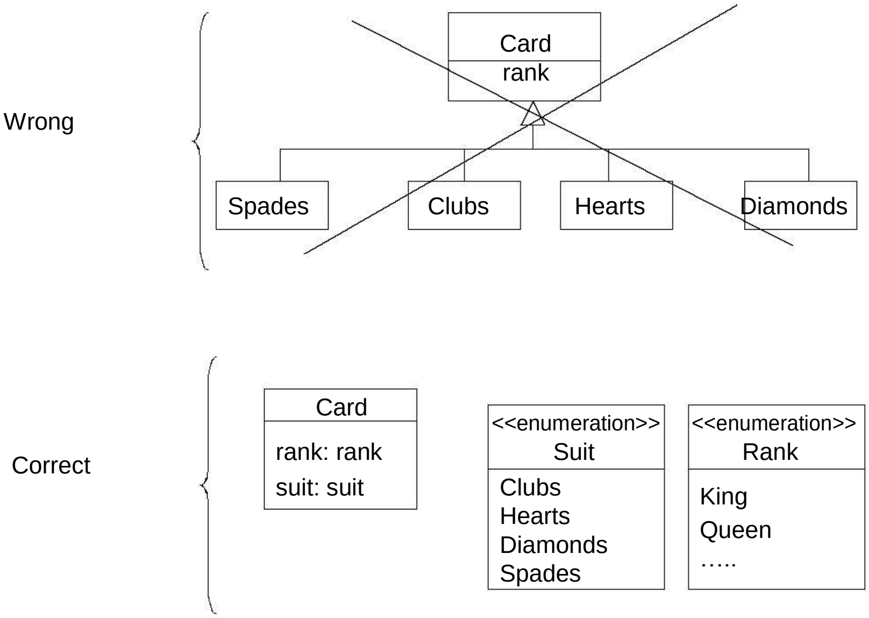
2.1 Advanced object and class concepts

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

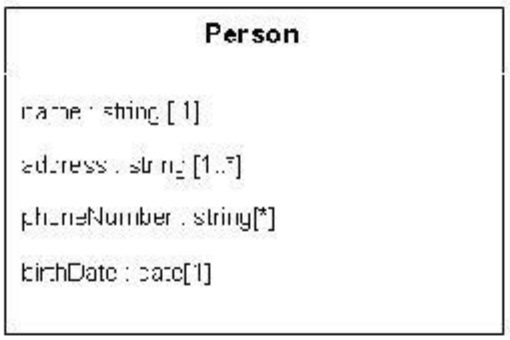




Modeling enumerations. Do not use a generalization to capture the values of an enumerated attribute

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

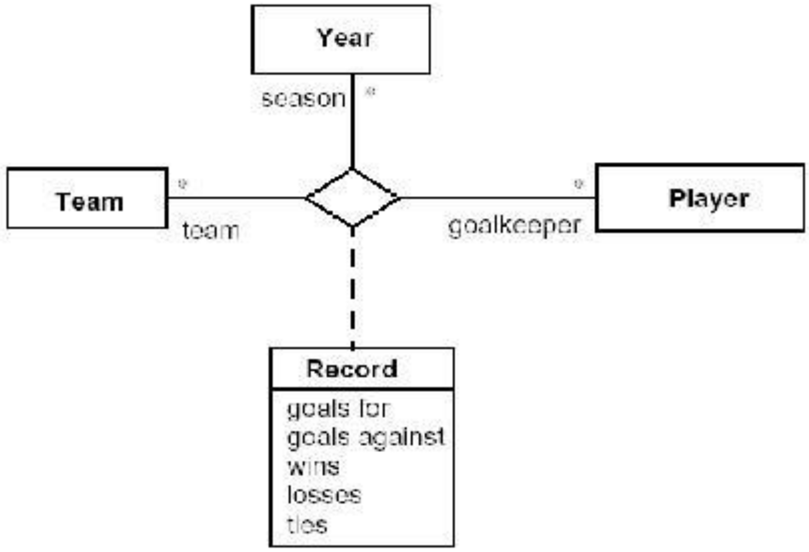
- Visibility refers to the ability of a method to reference a feature from another class and has the possible values of *public, protected, private,* and *package.*
- Any method can access **public** features.
- Only methods of the containing class and its descendants via inheritance can access **protected** 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.

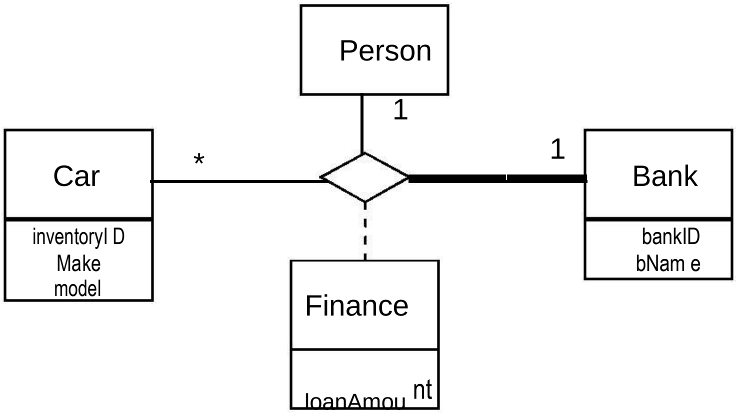
2.2 Associations ends

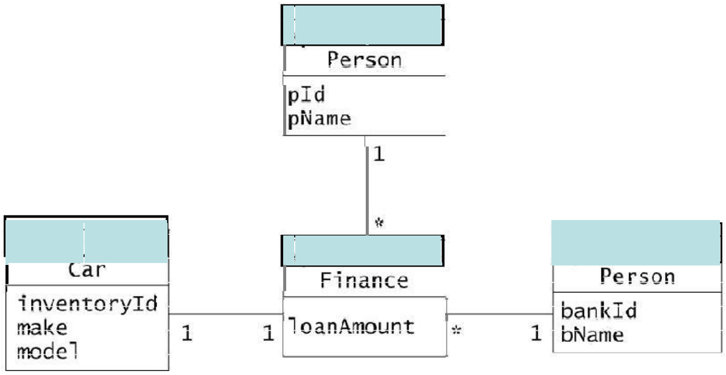
- Association End is an end of association.
- A binary association has 2 ends; a ternary association has 3 ends.

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.

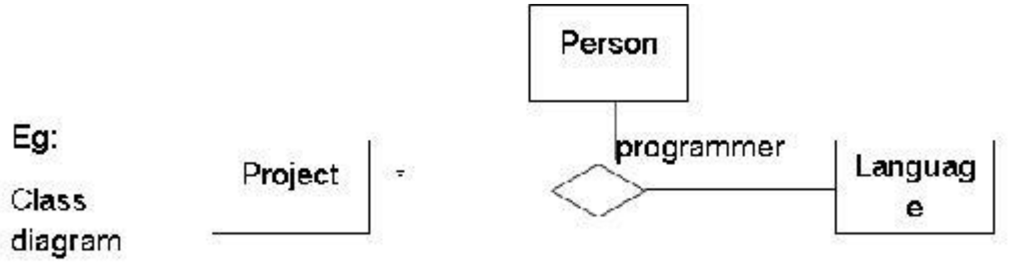






- The UML symbol for n-ary associations is a diamond with lines connecting to related classes. If the association has a name, it is written in italics next to the diamond.
- The OCL does not define notation for traversing n-ary associations.

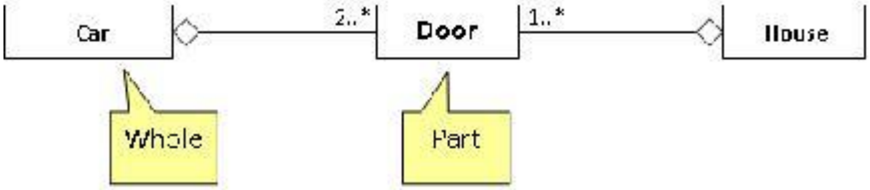
- A typical programming language cannot express n-ary associations. So, promote n-ary associations to classes. Be aware that you change the meaning of a model, when you promote n-ary associations to classes.
- An n-ary association enforces that there is at most one link for each combination.



Instance diagram <u>*see prescribed text book page no. 65 and fing no. 4.6*</u>

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

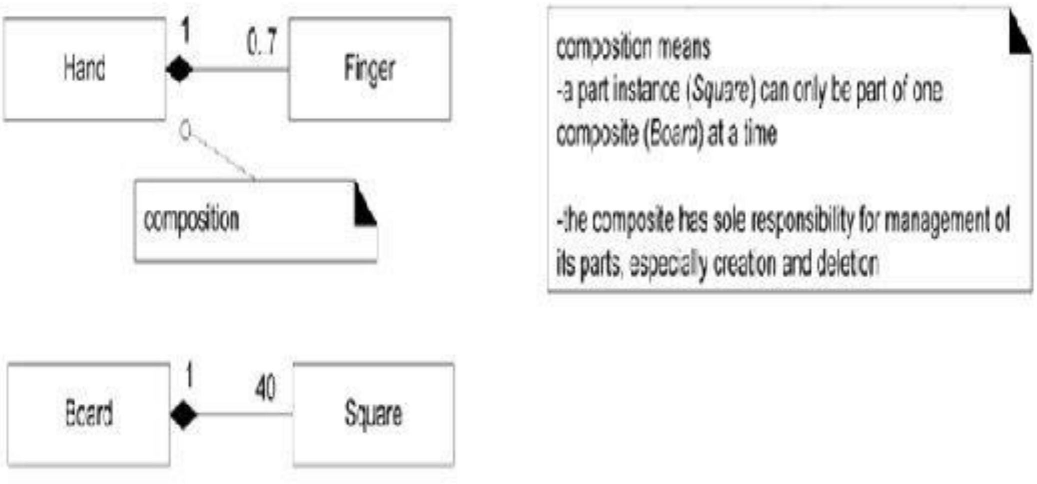
- Aggregation is a special form of association, not an independent concept.
- Aggregation adds semantic connotations.
- If two objects are tightly bound by a part-whole relationship, it is an aggregation. If the two objects are usually considered as independent, even though they may often be linked,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

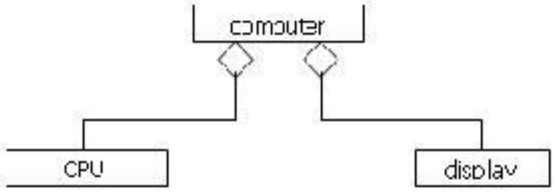
- The decision to use aggregation is a matter of judgment and can be arbitrary.

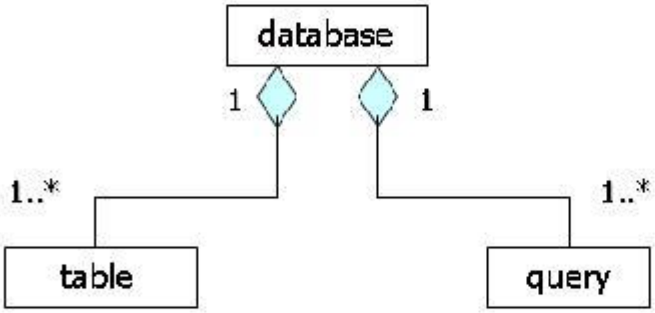
2.4.2 Aggregation versus Composition

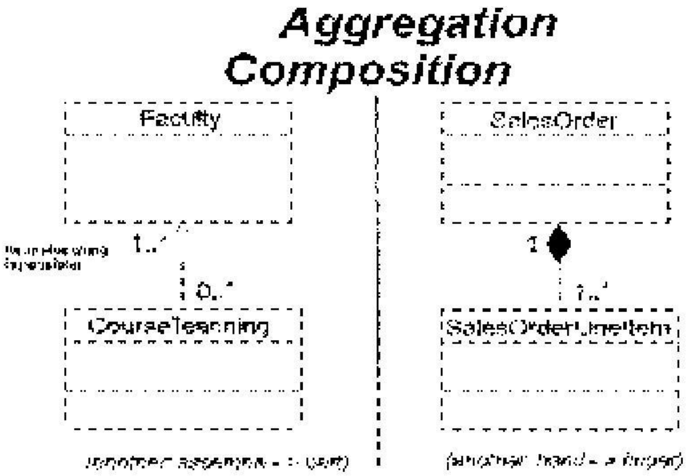
- The UML has 2 forms of part-whole relationships: a general form called Aggregation and a more restrictive form called composition.
- Composition is a form of aggregation with two additional constraints.
- A constitute part can belong to at most one assembly.
- Once a constitute part has been assigned 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

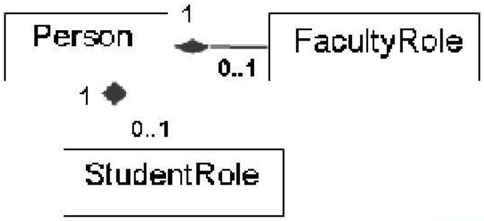
Composition

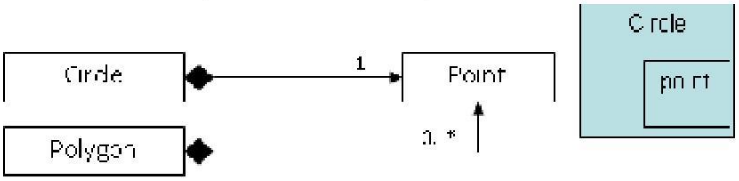

Hand
1
0..7
Finger
composition
composition means
-a part instance (Square) can only be part of one composite (Board) at a time
-the composite has sole responsibility for management of its parts, especially creation and deletion
Board
1
40
Square











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 pl. 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 br.s. 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 repl.s a generalization. This is similar to multiple classification. This approach repl.s 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.

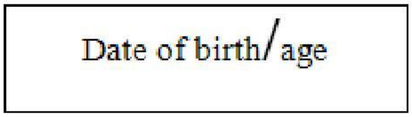
  Eg: see text book page no: 78 fig: 4.24

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 br.s or pl. 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.



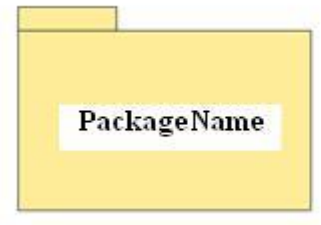
Date of birth/age

- 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.

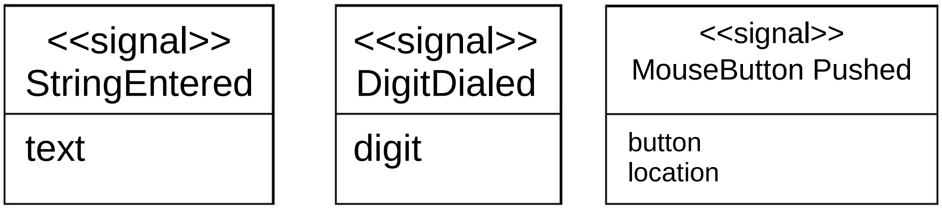
Three types of events:

- 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)

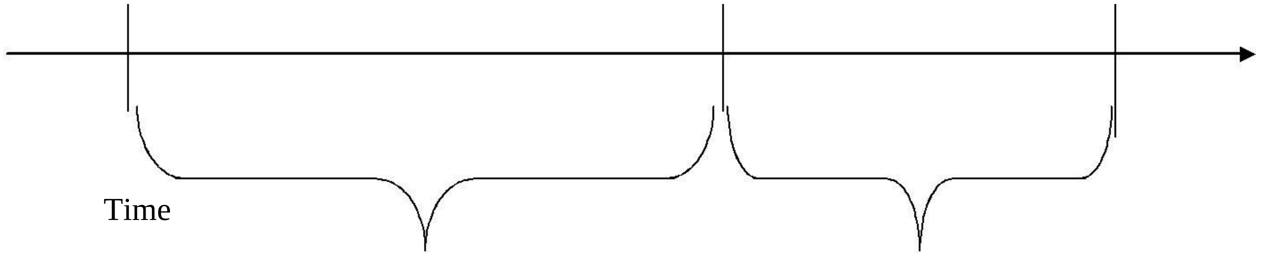
States

- 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 boldf., 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

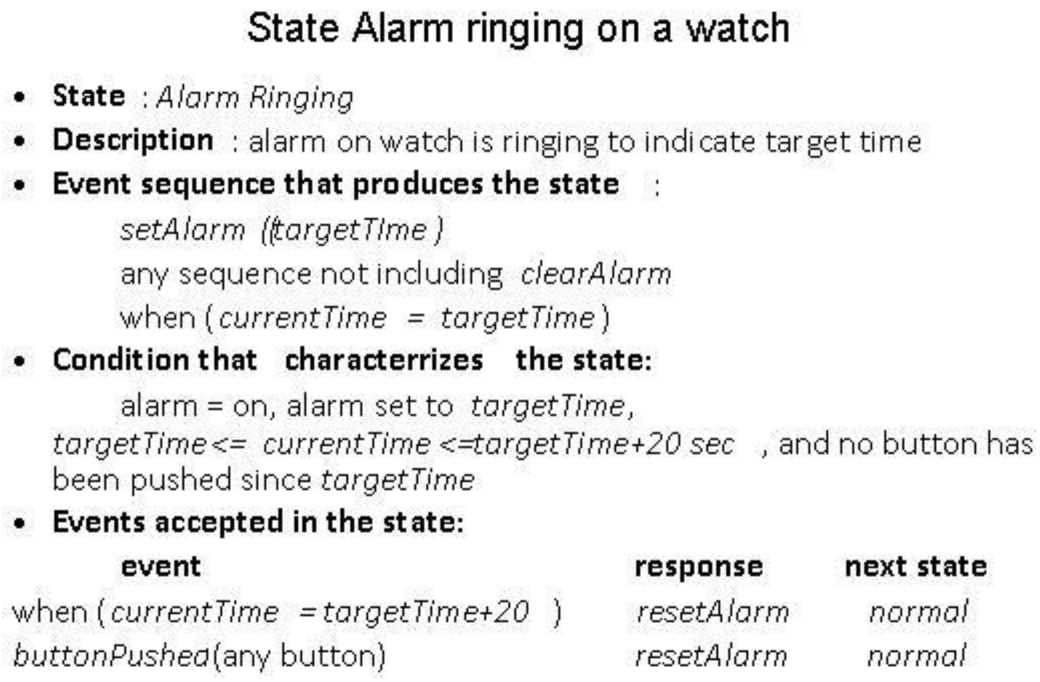
- ➢ Event represents points in time.
- ➢ State represents intervals of time.

- 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.

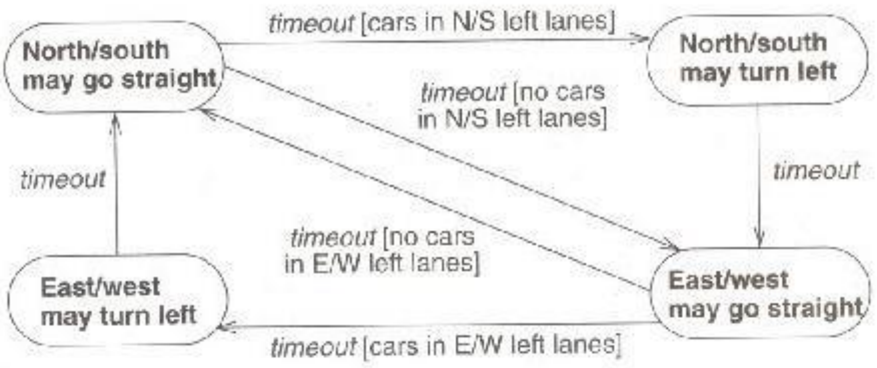

North/south may go straight
North/south may turn left
East/west may turn left
East/west may go straight
timeout [cars in N/S left lanes]
timeout [no cars in N/S left lanes]
timeout
timeout
timeout [no cars in E/W left lanes]
timeout [cars in E/W left lanes]

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

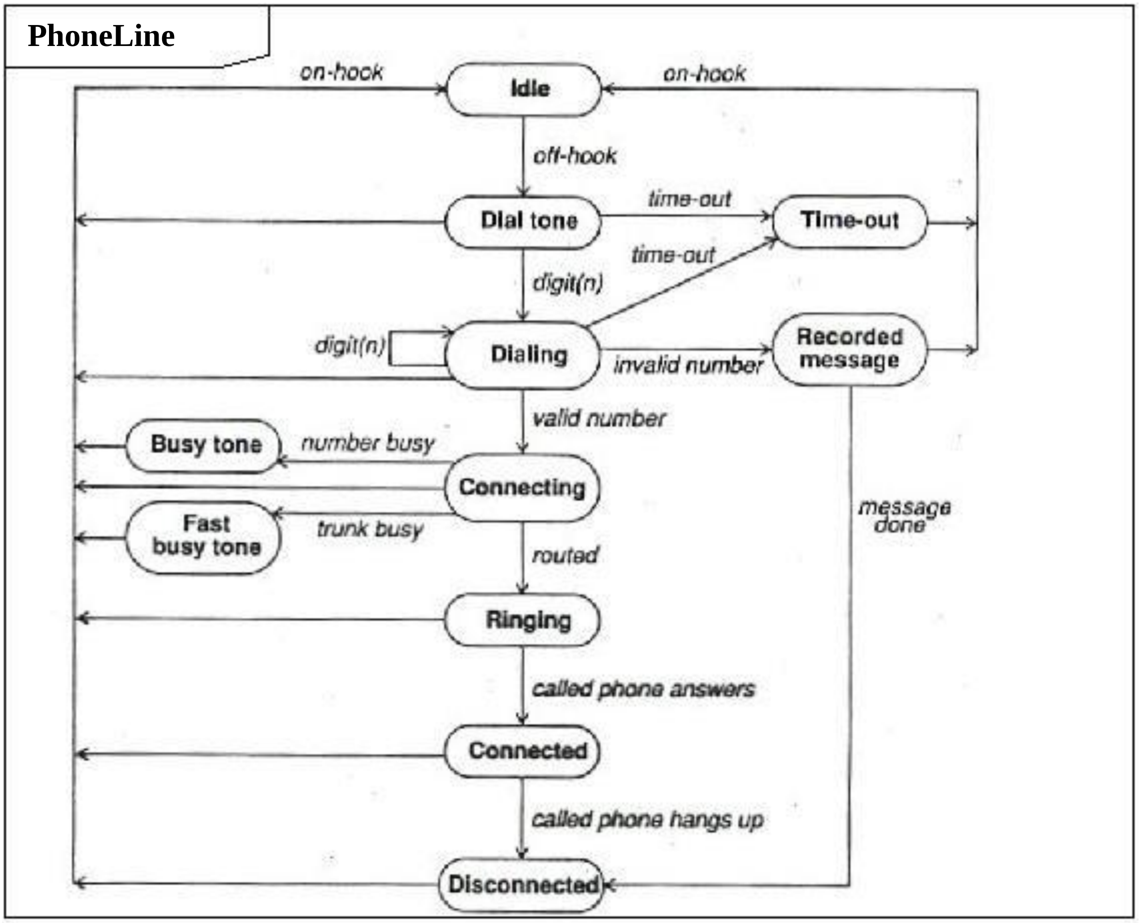


Figure 5.5 State diagram for phone line

<u>One shot state diagrams</u>

- 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.

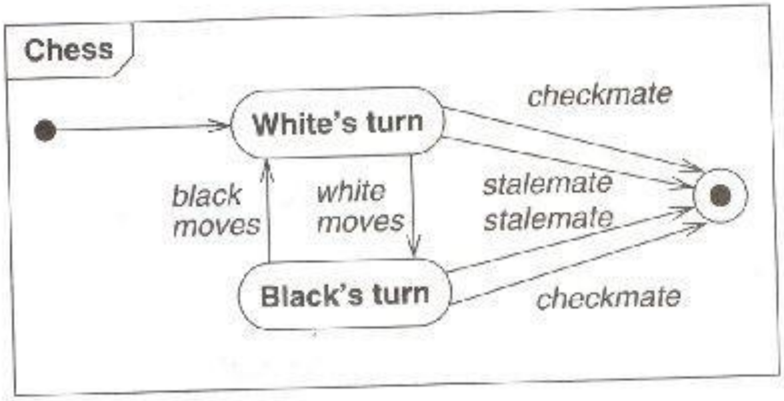


Figure 5.9 State diagram for chess game. One-shot diagrams represent objects with finite lives.

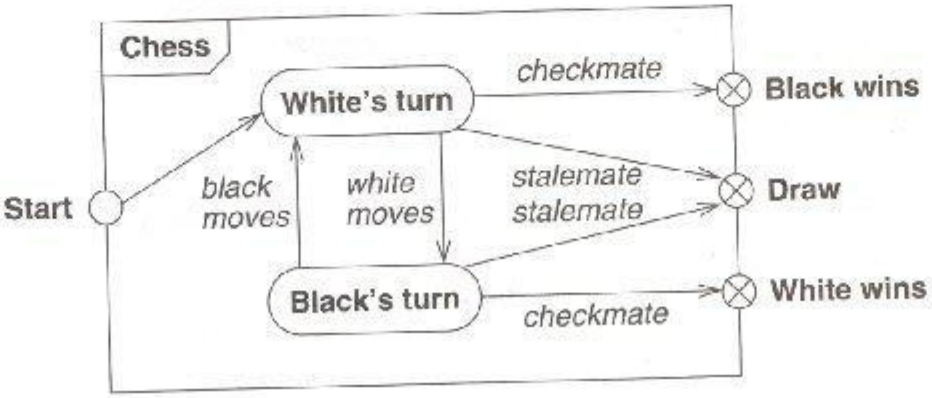


Figure 5.10 State diagram for chess game. You can also show one-shot diagrams by using entry and exit points.

5.4.3 Summary of Basic State Diagram Notation

Figure 5.11 summarizes the basic UML syntax for state diagrams.

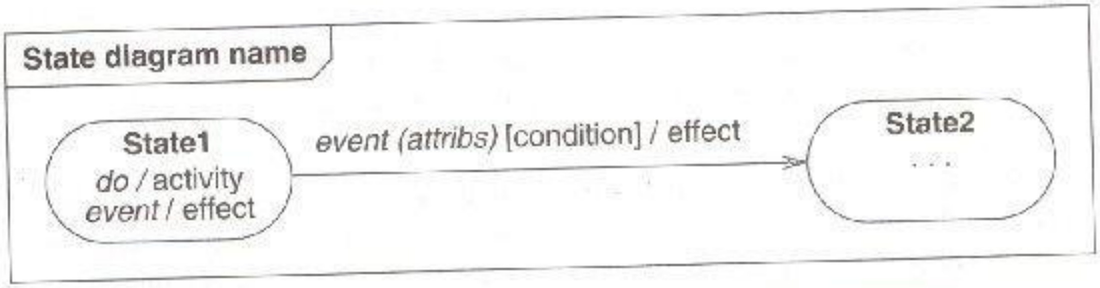


Figure 5.11 Summary of basic notation for state diagrams.

optional name. A special notation is ava

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.

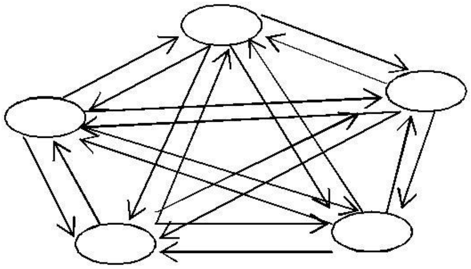
Unit 3: Advanced State Diagrams

Syllabus-------- **7hr**

- **Nested state diagram**
- **Nested states**
- **Signal generalization**
- **Concurrency**
- **A sample state mode**
- **Relation of class and state models**
- **Relation of class and state models**
- **Use case models**
- **Sequence models**
- **Activity models**

Problem with flat state diagrams

- Flat unstructured state diagram are impractical for large problems, because – representing an object with n independent Boolean attribute requires 2^n states. By partitioning the state into n independent sate diagram requires 2n states only.
- Eg:



Above figure requires n2 transition to connect every state to other state. This can be reduced to as low as n by using sub diagrams structure.

Expanding states

- One way to organize a model is by having high level diagram with sub diagrams expanding certain state. This is like a macro substitution in programming language
- A submachine is a state diagram that may be invoked as part of another state diagram

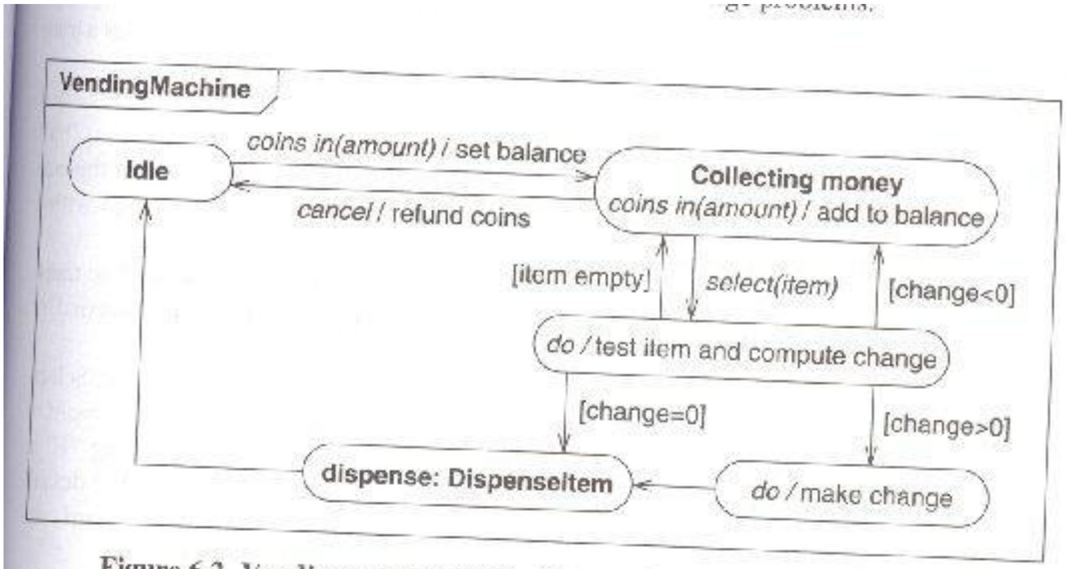


Figure 6.2 **Vending machine state diagram.** You can simplify state diagrams by using subdiagrams.

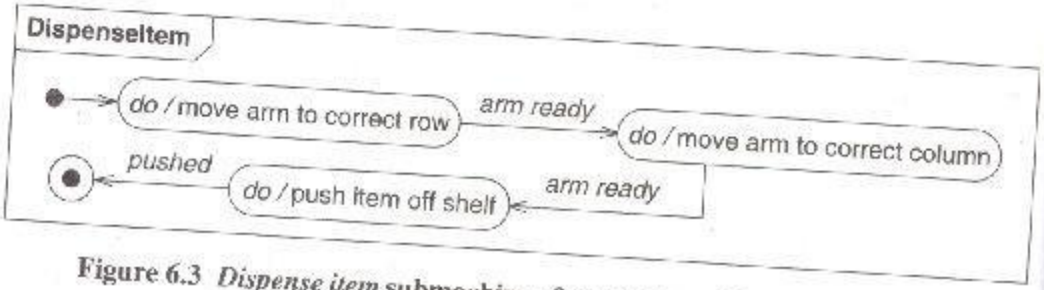


Figure 6.3 ***Dispense item* submachine of vending machine.** A lower-level state diagram can elaborate a state.

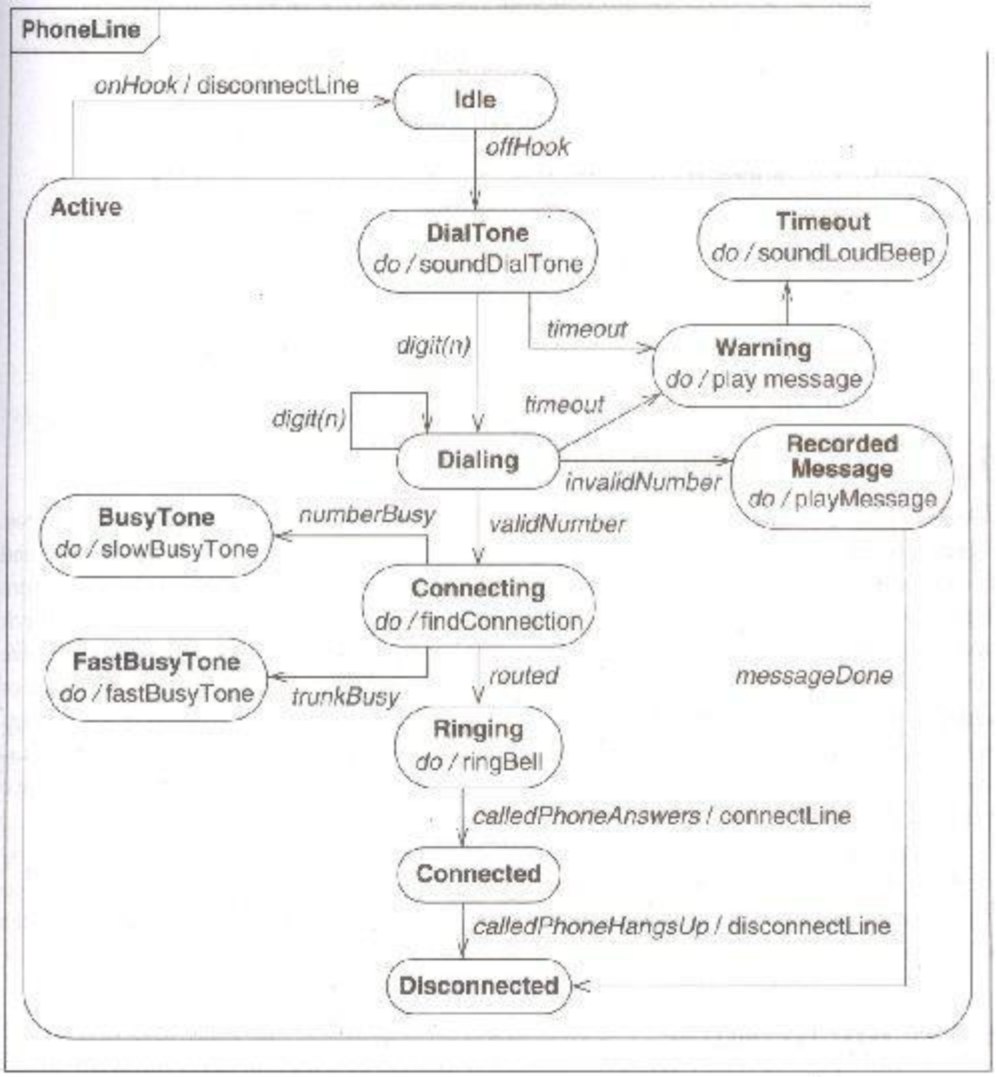


Figure 6.4 Nested states for a phone line. A nested state receives the outgoing transitions of its enclosing state.

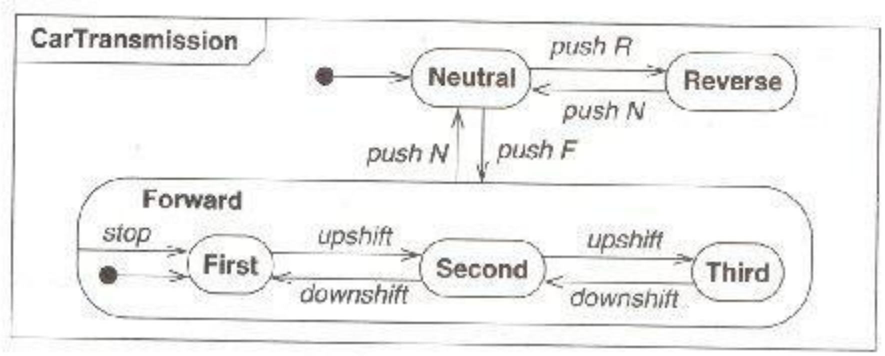


Figure 6.5 **Nested states.** You can nest states to an arbitrary depth.

Signal generalization

You can organize signals into generalization hierarchy with inheritance of signal attributes

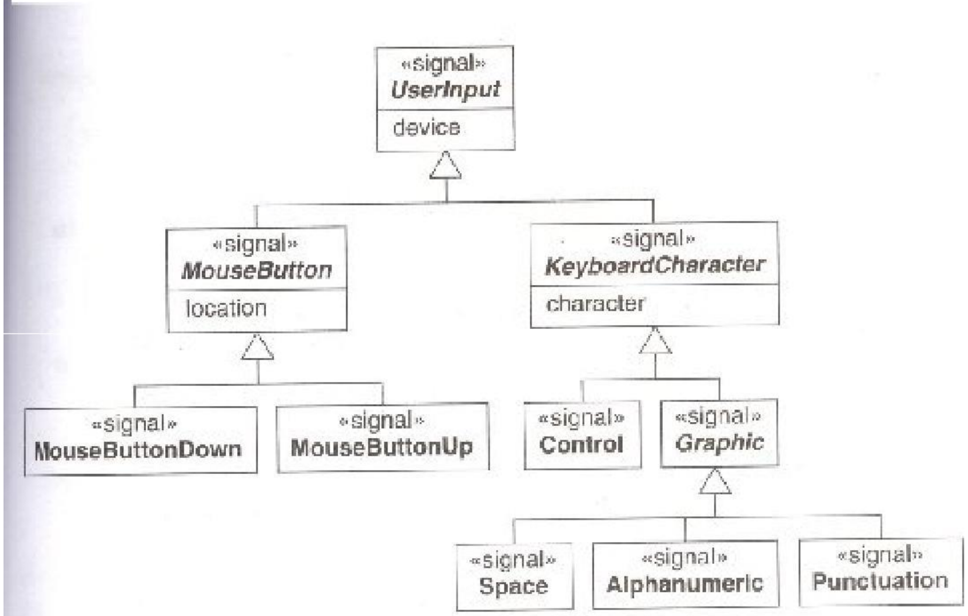
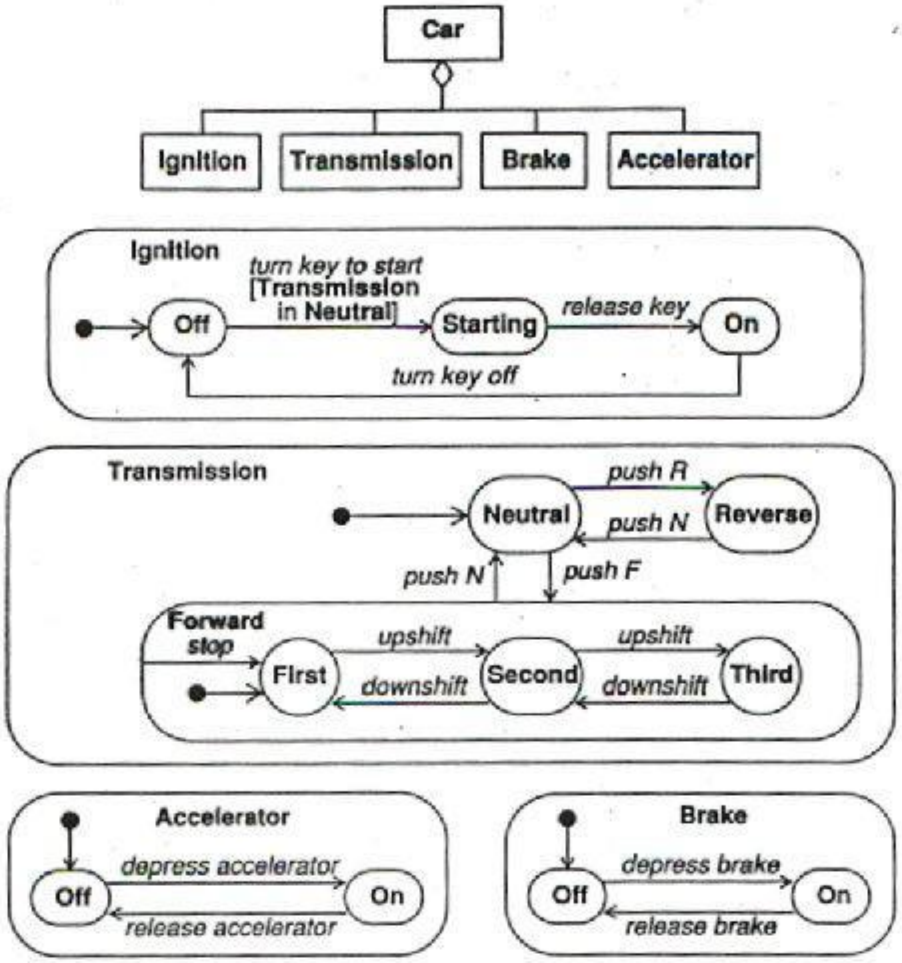


Figure 6.6 **Partial hierarchy for keyboard signals.** You can organize signals using generalization.

- Ultimately, we can view every actual signal as a leaf on a generalization tree of signals

- In a state diagram, a received signal triggers transitions that are defined for any ancestor signal type.
- For eg: typing an ‘a’ would trigger a transition on a signal alphanumeric as well as key board character.
- **Concurrency 1:**
- The state model implicitly supports concurrency among objects.
- In general, objects are autonomous entities that can act and change state independent of one another. However objects need not be completely independent and may be subject to shared constraints that cause some correspondence among their state changes.

1 Aggregation concurrency



2 concurrency within an object

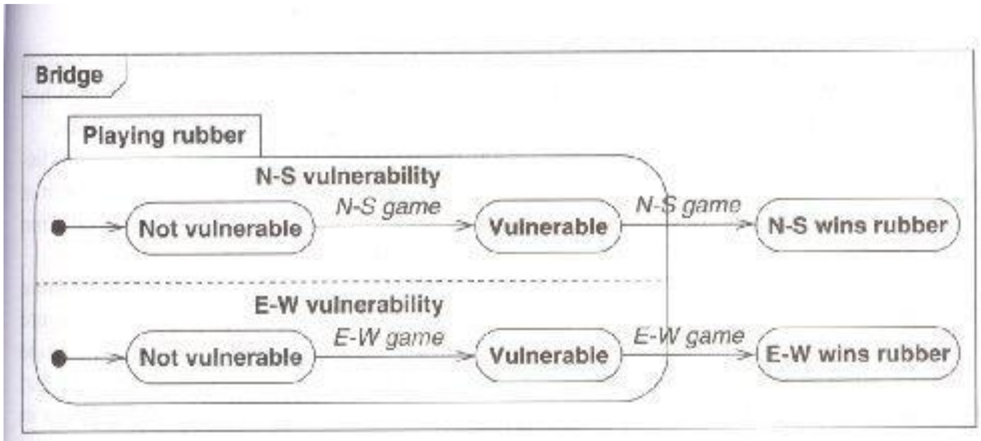


Figure 6.8 Bridge game with concurrent states. You can partition some objects into subsets of attributes or links, each of which has its own subdiagram.

3 synchronization of concurrent activities

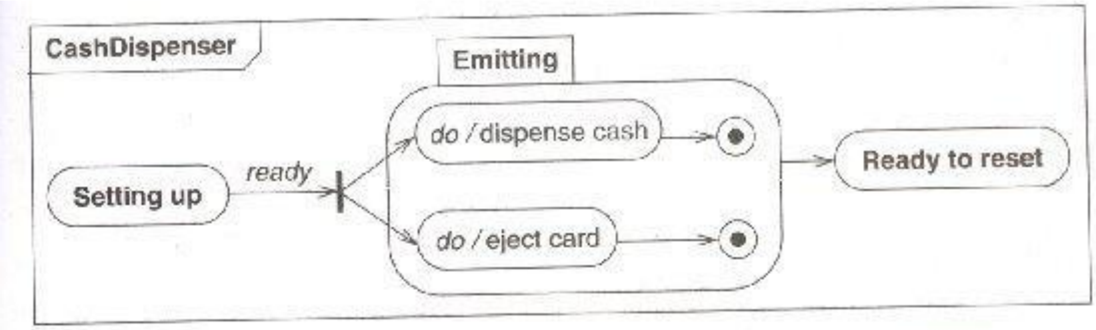


Figure 6.9 Synchronization of control. Control can split into concurrent activities that subsequently merge.

Interaction Models

- The class model describes the objects in a system and their relationship.
- The state model describes the life cycles of the objects.
- The interaction model describes how the objects interact.

The **interaction model** starts with **use cases** that are then elaborated with **sequence** and **activity diagrams**

- **Use case:** focuses on functionality of a system- i.e, what a system does for users
- **Sequence diagrams:** shows the object that interact and the time sequence of their interactions
- **Activity diagrams:** elaborates important processing steps

Use Case models

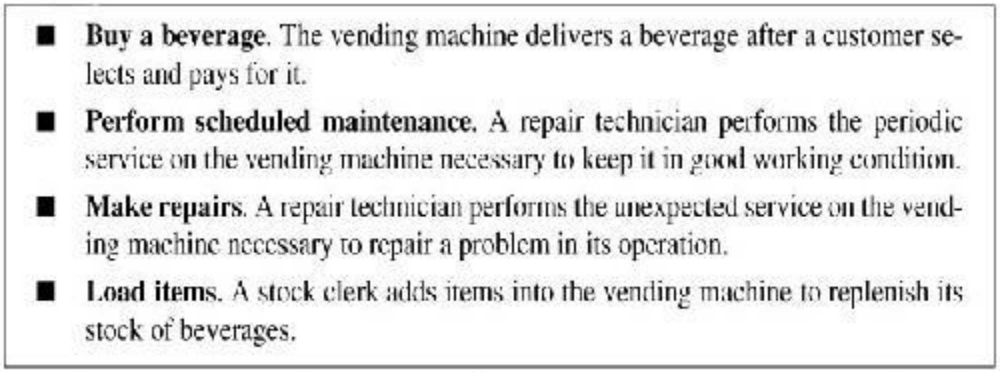
Actors

- A direct external user of a system
- Not part of the system
- For example
- ■ Traveler, agent, and airline for a travel agency system.
- Can be a person, devices and other system
- An actor has a single well-defined purpose

Use Cases

- A use case is a coherent piece of functionality that a system can provide by interacting with actors.
- For example:
- ■ A *customer* actor can *buy a beverage* from a vending machine.
- ■ A *repair technician* can *perform scheduled maintenance* on a vending machine.
- Each use case involves one or more actors as well as the system itself.

A Vending Machine



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

Figure 7.1 Use case summaries for a vending machine. A use case is a coherent piece of functionality that a system can provide by interacting with actors.

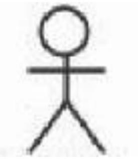
Object Oriented Modeling and Design with UML, Second Edition by Michael Blaha and James Rumbaugh. ISBN 0-13-1-015920-4. © 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

- A use case involves a sequence of messages among the system and its actors.
- Error conditions are also part of a use case.
- A use case brings together all of the behavior relevant to a slice of system functionality.

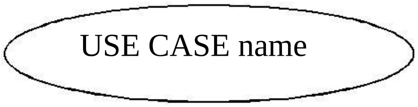
Use Case Description (see text book fug 7.2)

- Use Case Name
- Summary
- Actors
- Preconditions
- Description
- Exception
- Postcondition

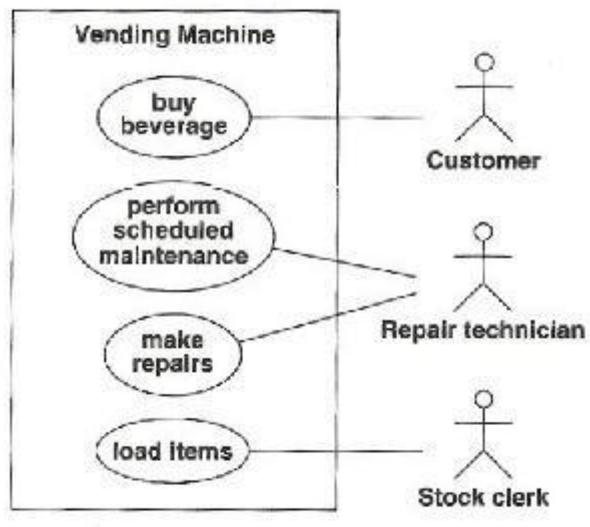
- **Actor**



- **Use Case**



A Vending Machine



Guidelines for Use Case

- First determine the system boundary
- Ensure that actors are focused
- Each use case must provide value to uses
- Relate use cases and actors
- Remember that use cases are informal
- Use cases can be structured

Use Case Relationships

- Include Relationship
- ■ Incorporate one use case within the behavior sequence of another use case. ● Extend Relationship
- ■ Add incremental behavior to a use case.
- Generalization
- ■ Show specific variations on a general use case.

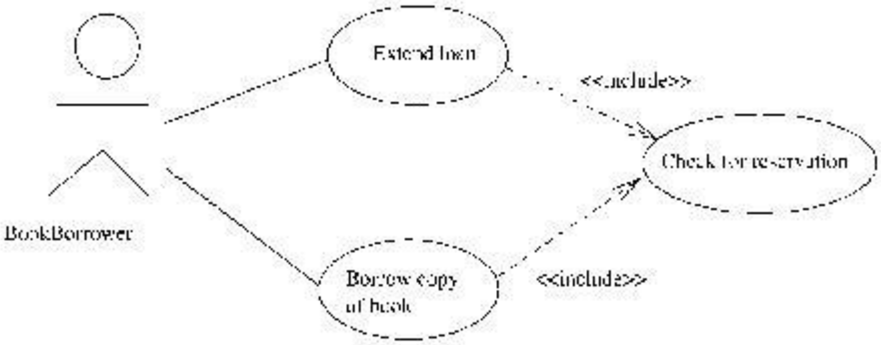
Use case Relationships

Include Relationship Exclude relationship generalization relationship

Examples:

<<include>> for common behavior

(1)



(2)

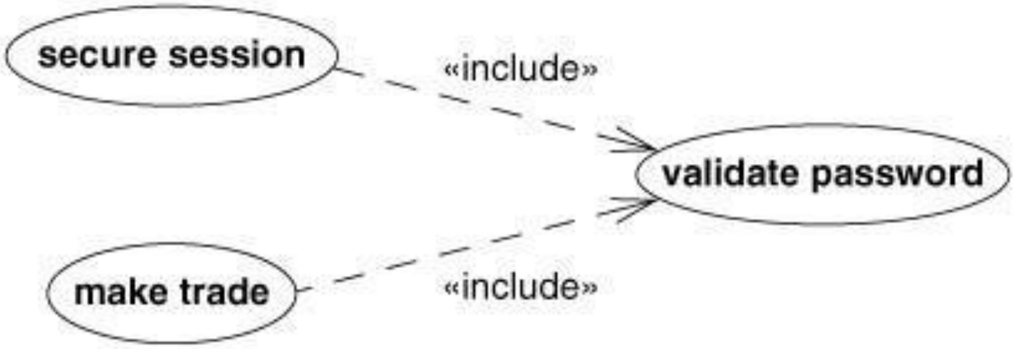
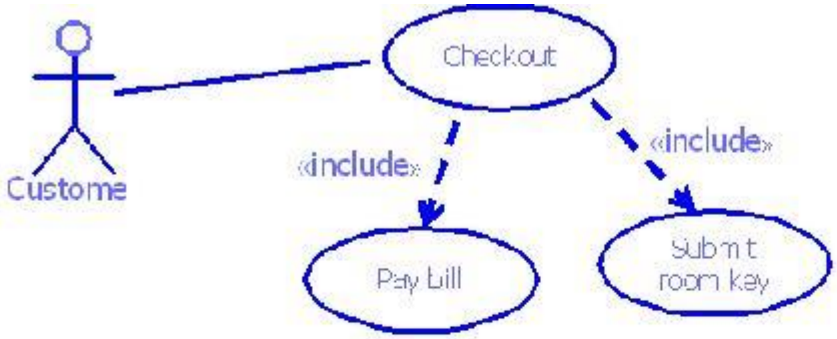


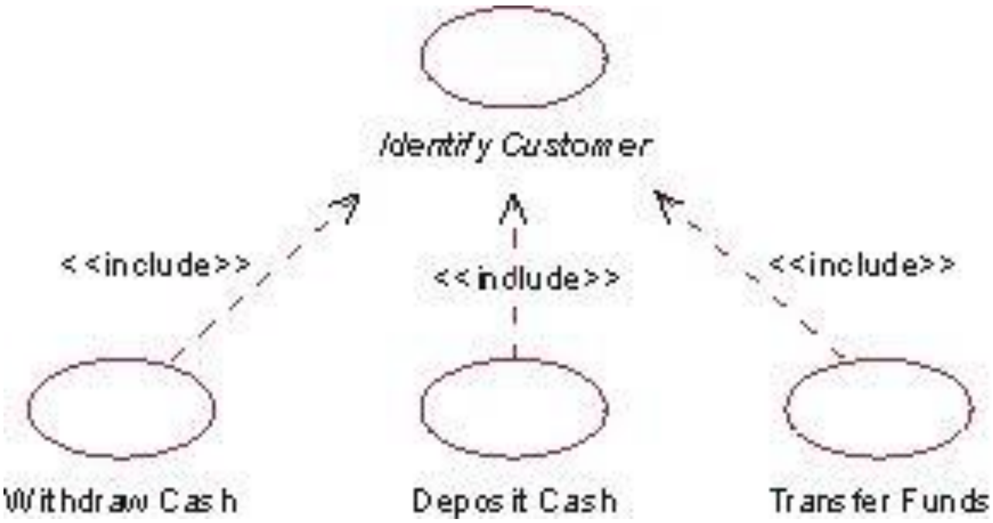
Figure 8.1 Use case inclusion. The *include* relationship lets a base use case incorporate behavior from another use case.

Object-Oriented Modeling and Design with UML, Second Edition by Michael Blaha and James Rumbaugh. ISBN 0-13-1-015920-4. © 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

(3)



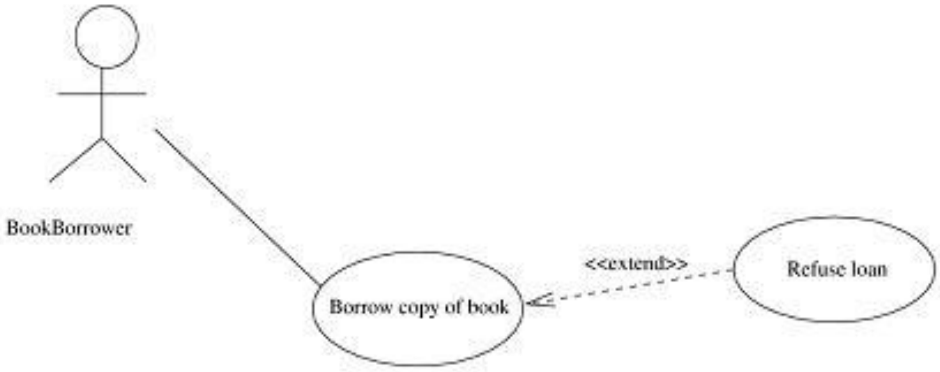
(4)



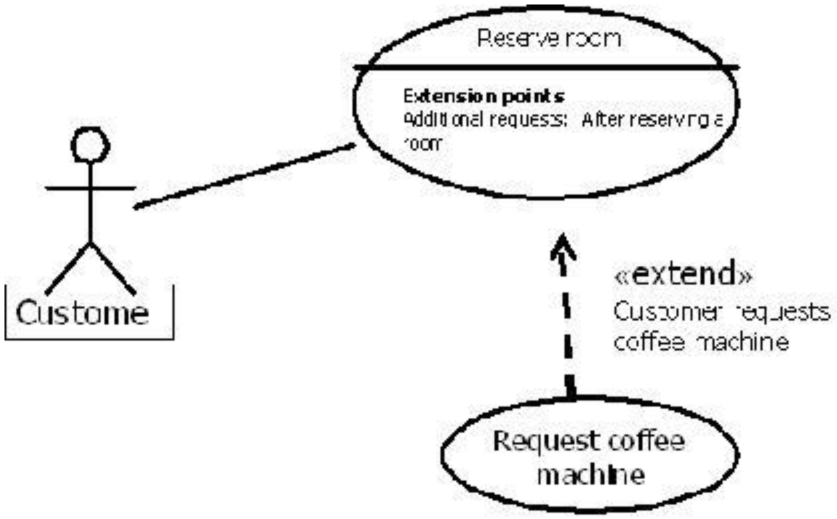
Extend Relationship examples:

<<extend>> for special cases:

(1)



(2)


Reserve room
Extension points
Additional requests: After reserving a room
Custome
«extend»
Customer requests coffee machine
Request coffee machine

(3)

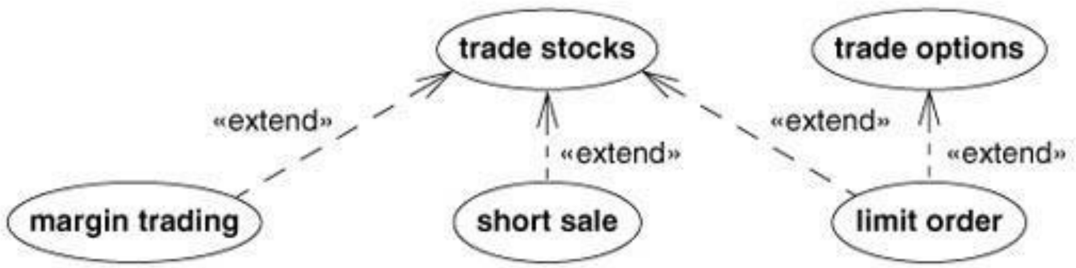
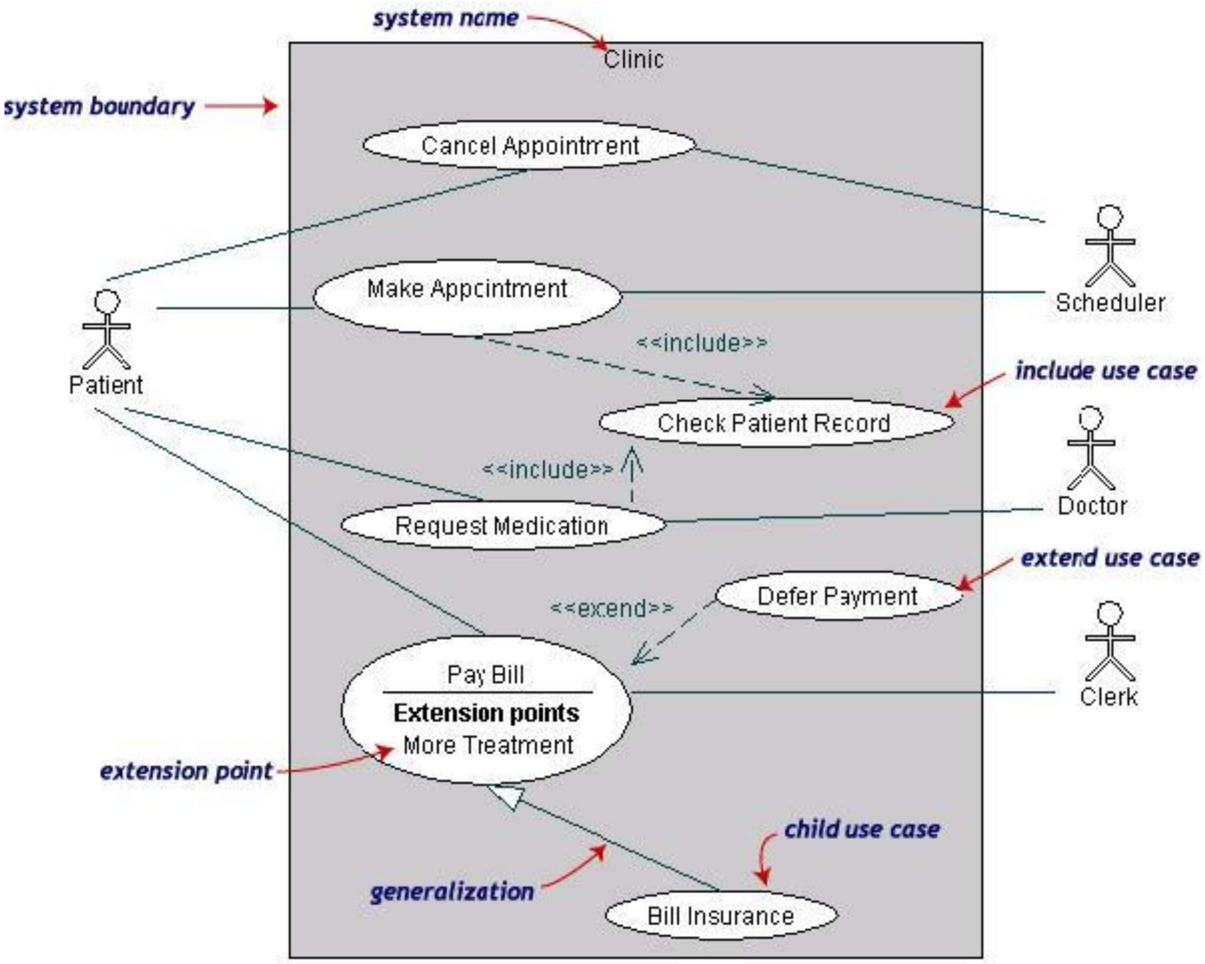


Figure 8.2 Use case extension. The *extend* relationship is like an *include* relationship looked at from the opposite direction. The extension adds itself to the base.

Object-Oriented Modeling and Design with UML, Second Edition by Michael Blaha and James Rumbaugh. ISBN 0-13-1-015920-4. © 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Medical Clinic: «include» and «extend»



Generalization

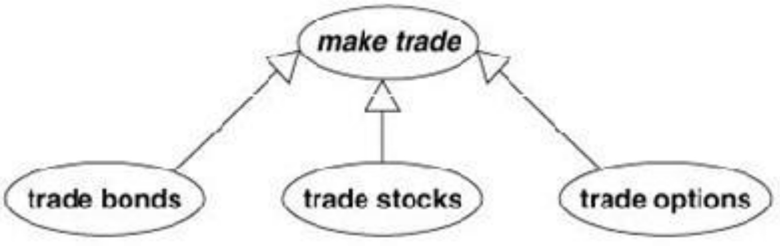
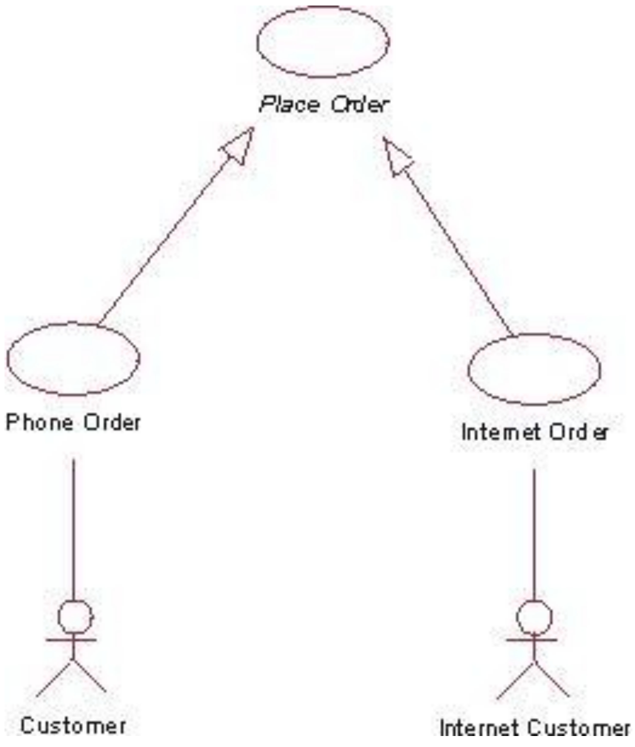


Figure 8.3 Use case generalization. A parent use case has common behavior and child use cases add variations, analogous to generalization among classes.

(2)eg:



Use Case Relationships

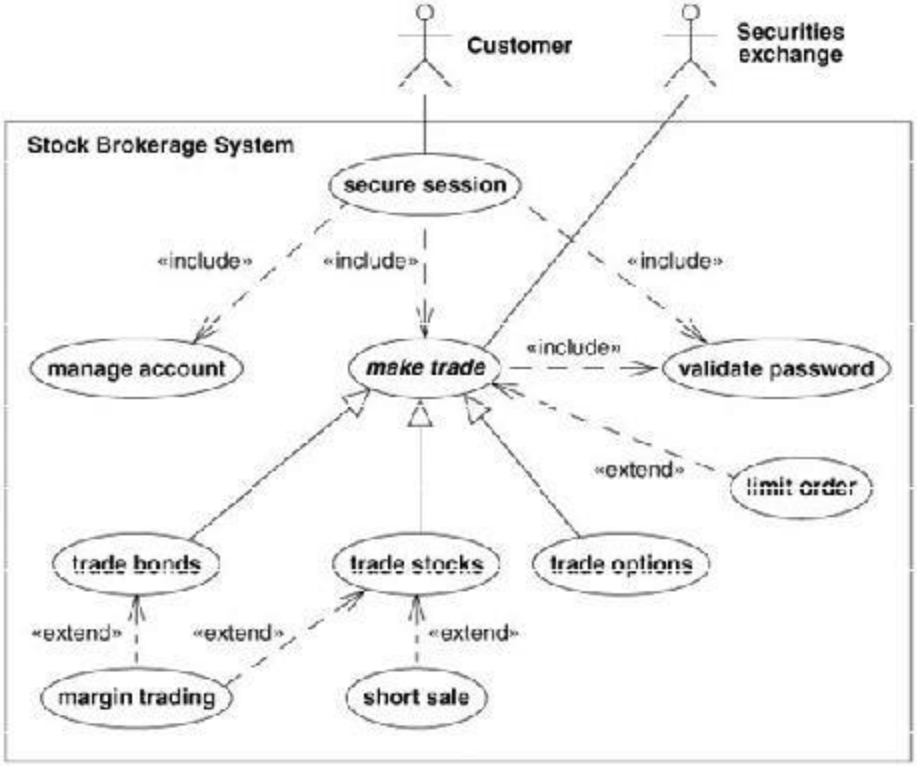


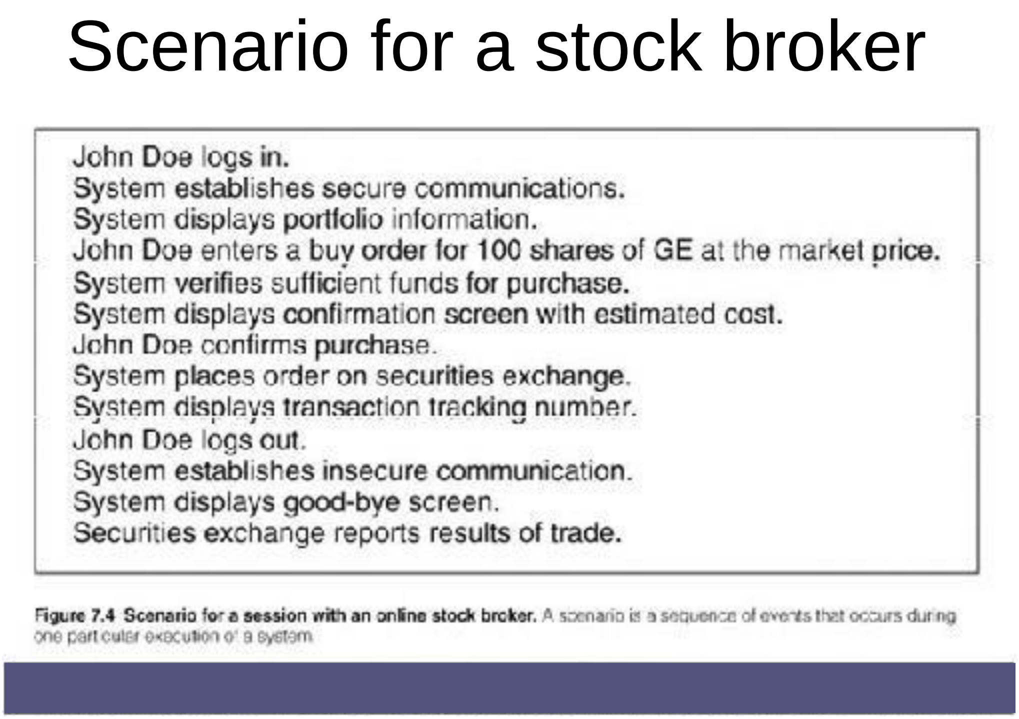
Figure 8.4 Use case relationships. A single use case diagram may combine several kinds of relationships.

Sequence Models

- The sequence model elaborates the themes of use cases.
- Tow kinds of sequences models
- Scenarios
- Sequence diagrams

Scenarios

- A scenario is a sequence of events that occurs during one particular execution of a system.
- For example:
- *John Doe logs in* transmits a message from John Doe to the broker system.



Scenario for a stock broker

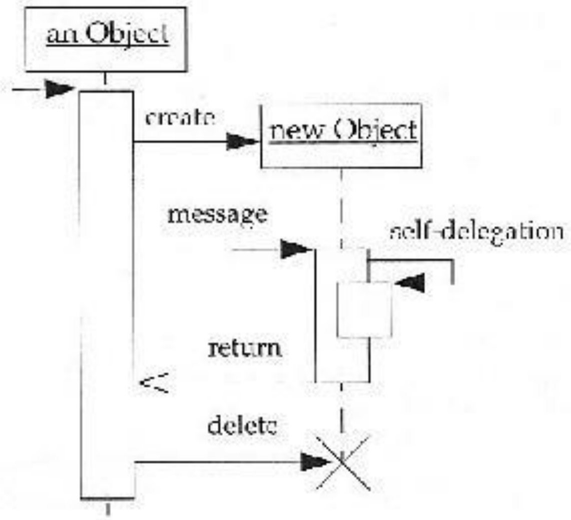
John Doe logs in.
System establishes secure communications.
System displays portfolio information.
John Doe enters a buy order for 100 shares of GE at the market price.
System verifies sufficient funds for purchase.
System displays confirmation screen with estimated cost.
John Doe confirms purchase.
System places order on securities exchange.
System displays transaction tracking number.
John Doe logs out.
System establishes insecure communication.
System displays good-bye screen.
Securities exchange reports results of trade.

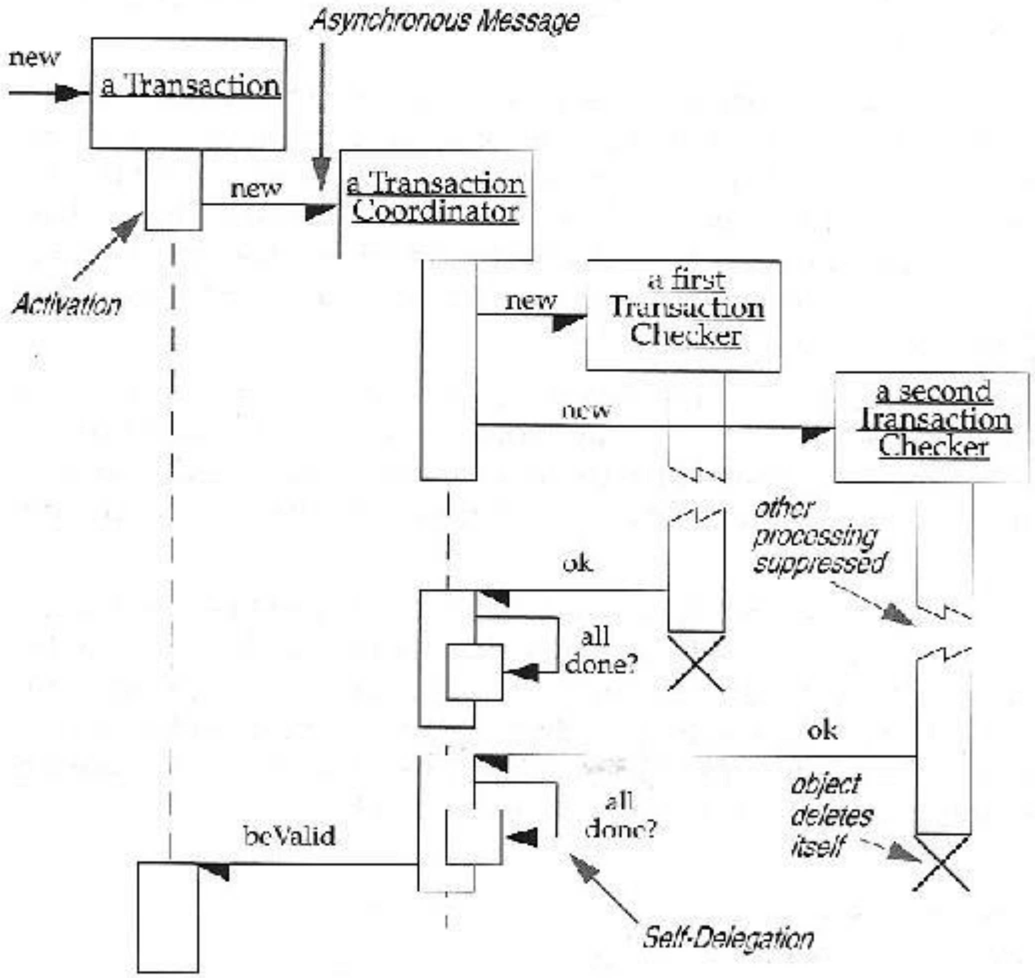
Figure 7.4 Scenario for a session with an online stock broker. A scenario is a sequence of events that occurs during one particular execution of a system.

Sequence Diagram

- A sequence diagram shows the participants in an interaction and the sequence of messages among them.
- A sequence diagram shows the interaction of a system with its actors to perform all or part of a use case.
- Each use case requires one or more sequence diagrams to describe its behavior.

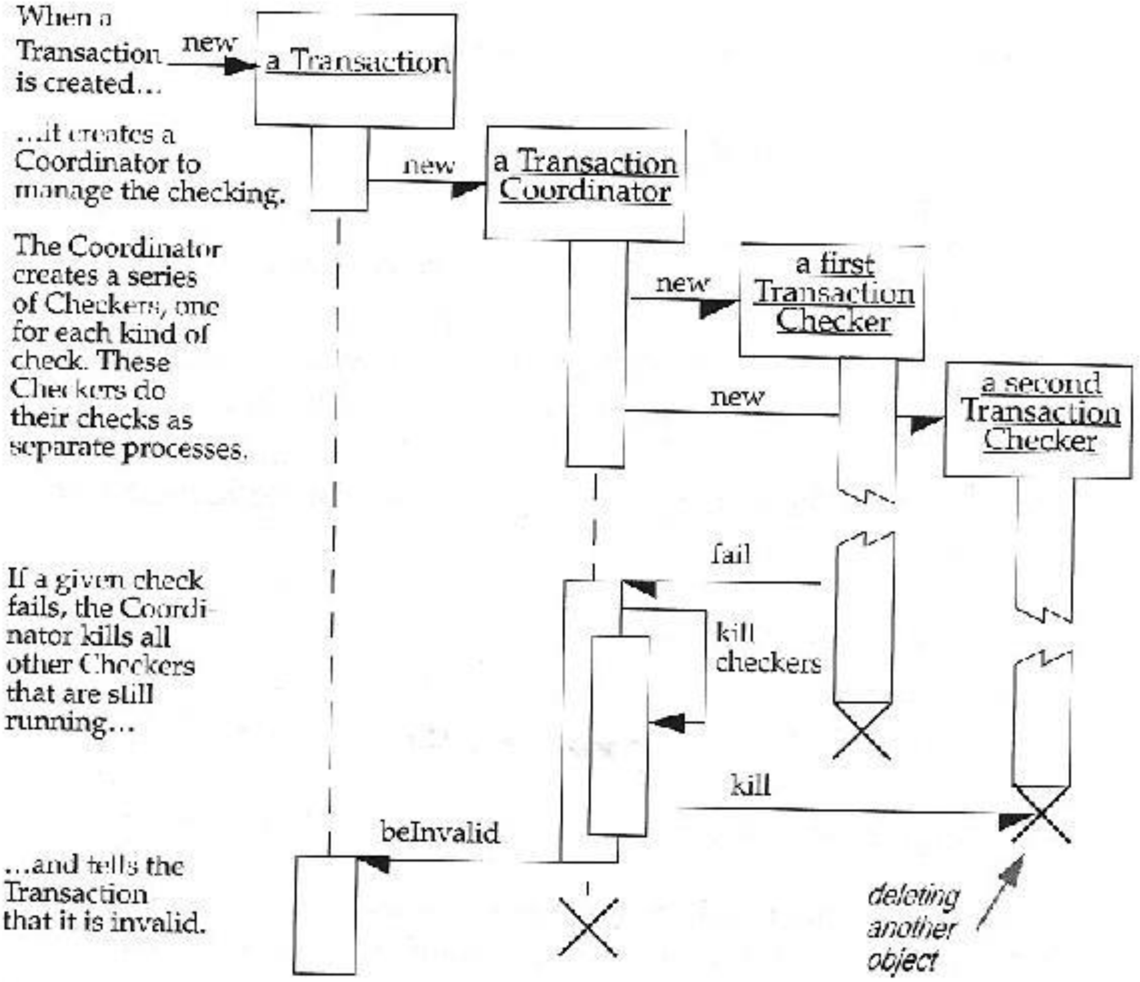
Sequence Diagram





Concurrent Processes

- Activations - show when a method is active – either executing or waiting for a subroutine to return
- Asynchronous Message – (half arrow) a message which does not block the caller, allowing the caller to carry on with its own processing; asynchronous messages can:
- Create a new thread
- Create a new object
- Communicate with a thread that is already running
- Deletion – an object deletes itself
- Synchronous Message – (full arrow) a message that blocks the caller


When a Transaction is created...
new
a Transaction
...it creates a Coordinator to manage the checking.
new
a Transaction Coordinator
The Coordinator creates a series of Checkers, one for each kind of check. These Checkers do their checks as separate processes.
new
a first Transaction Checker
new
a second Transaction Checker
If a given check fails, the Coordinator kills all other Checkers that are still running...
fail
kill checkers
kill
beInvalid
...and tells the Transaction that it is invalid.
deleting another object

Sequence Diagram For a Session

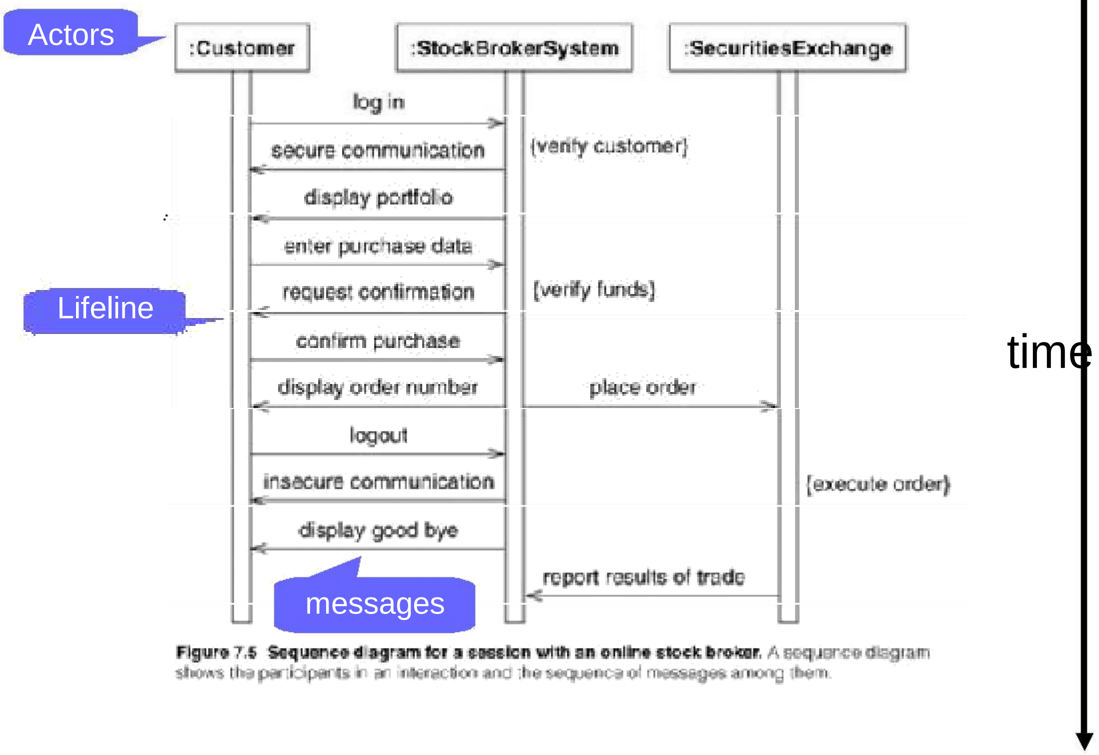


Figure 7.5 Sequence diagram for a session with an online stock broker. A sequence diagram shows the participants in an interaction and the sequence of messages among them.

A stock purchase

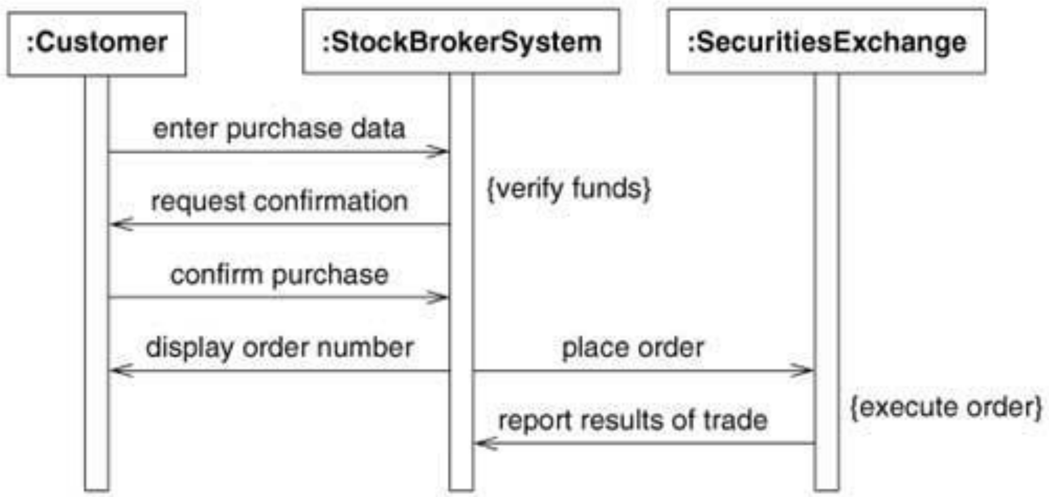


Figure 7.6 Sequence diagram for a stock purchase. Sequence diagrams can show large-scale interactions as well as smaller, constituent tasks.

A stock quote

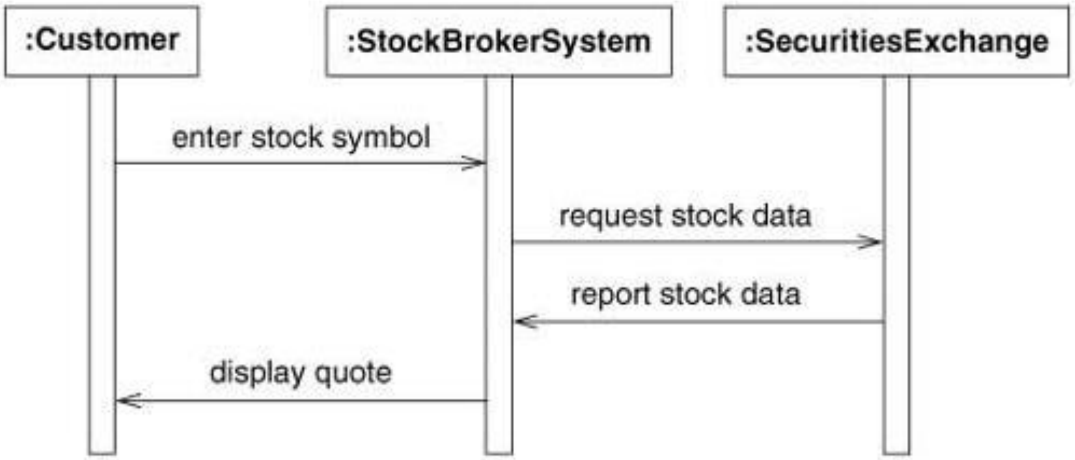


Figure 7.7 Sequence diagram for a stock quote.

A exception case

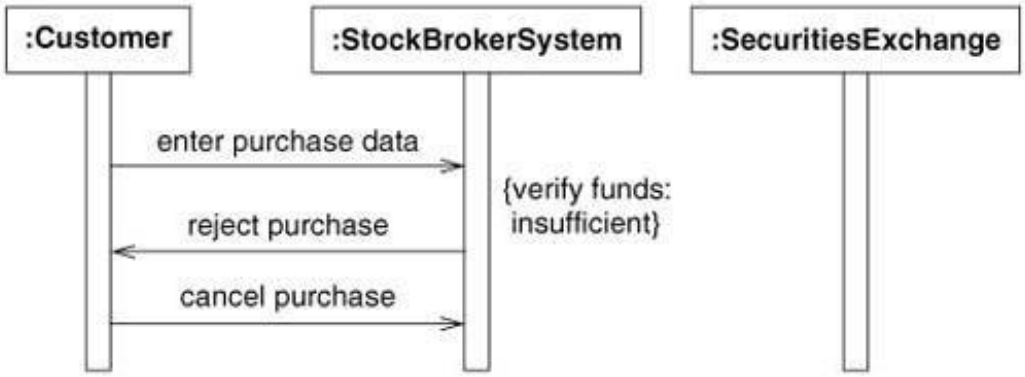


Figure 7.8 Sequence diagram for a stock purchase that fails.

Guidelines

- Prepare at least one scenario per use case
- Abstract the scenarios into sequence diagrams
- Divide complex interactions
- Prepare a sequence diagram for each error condition

Procedural Sequence Models

- Sequence Diagrams with Passive Objects
- ■ A passive object is not activated until it has been called.

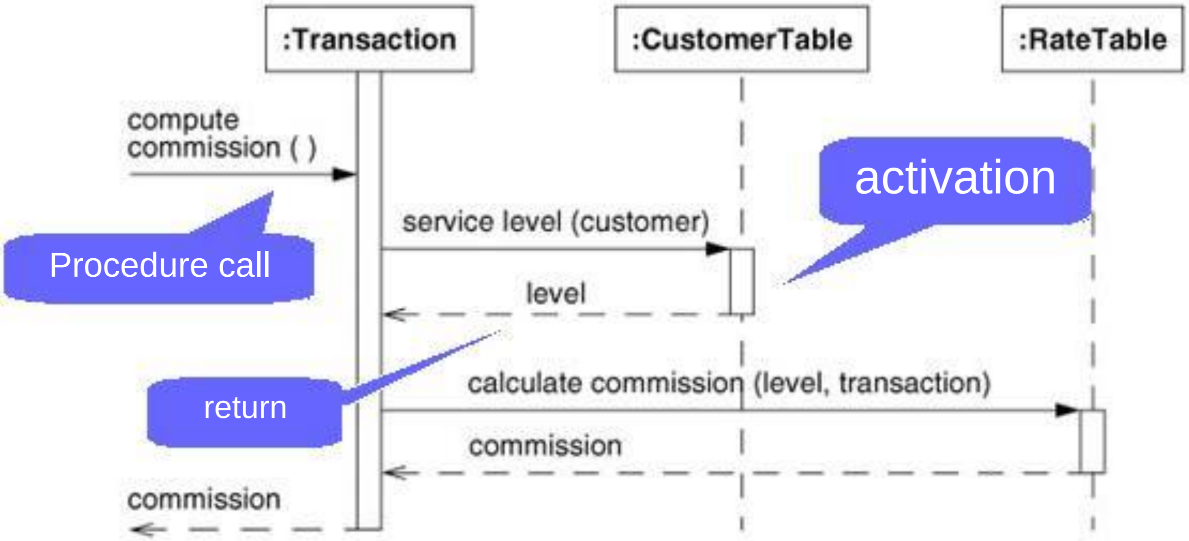


Figure 8.5 Sequence diagram with passive objects. Sequence diagrams can show the implementation of operations.

Sequence Diagrams with Transient Objects

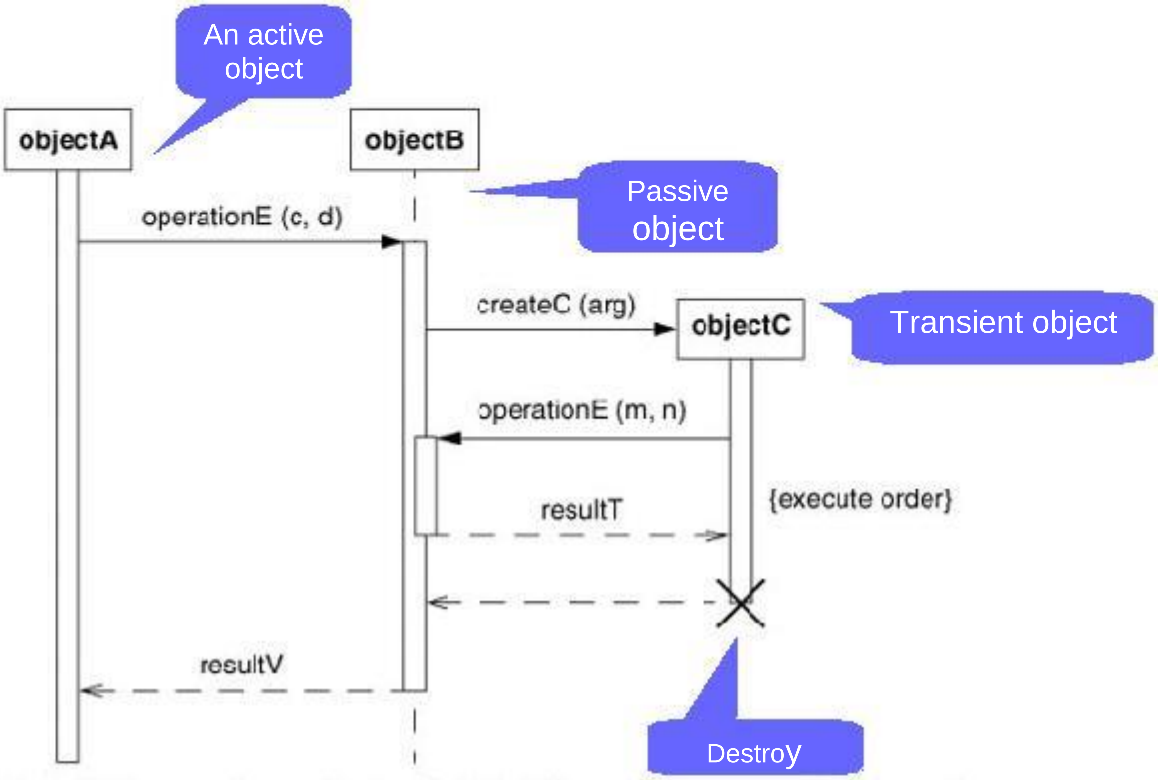


Figure 8.6 Sequence diagram with a transient object. Many applications have a mix of active and passive objects. They create and destroy objects.

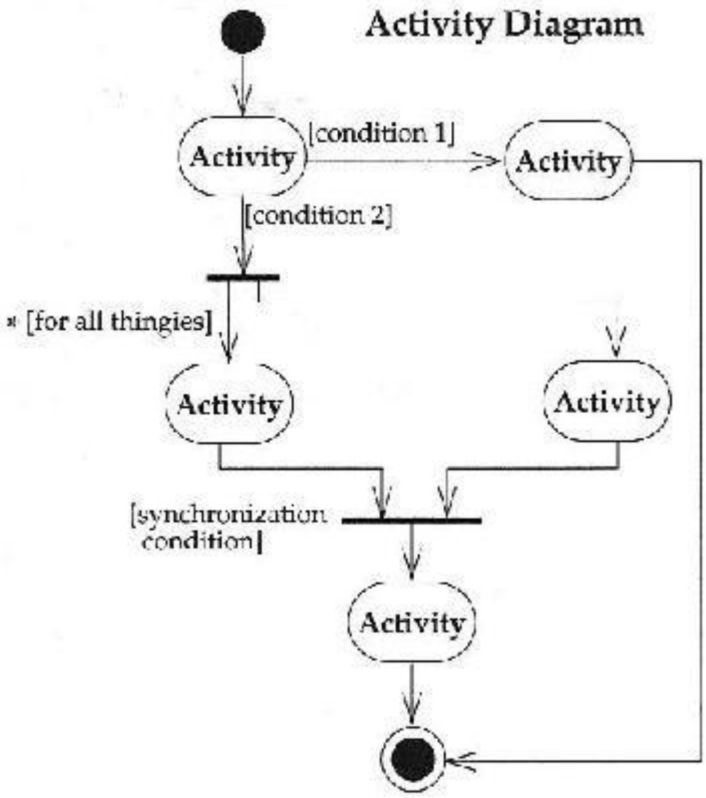
Activity Models

● An activity diagram shows the sequence of steps that make up a complex process, such as an algorithm or workflow.

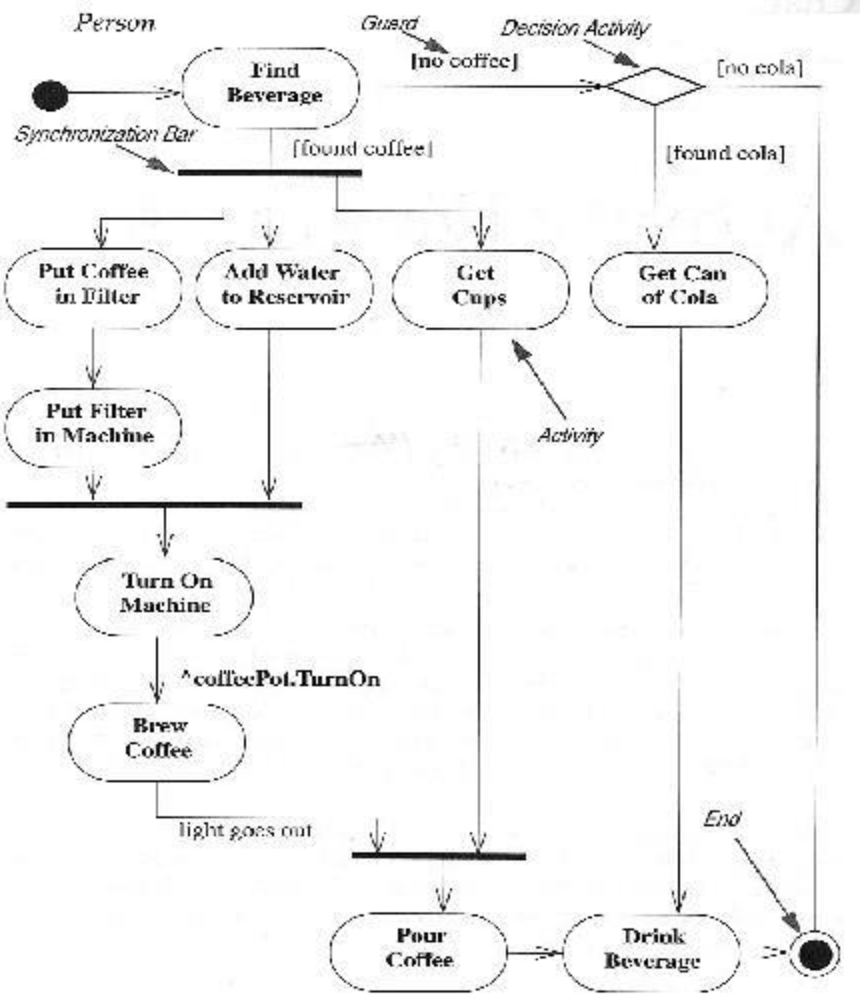
- Activity diagrams are most useful during the early stages of designing algorithms and workflows.
- Activity diagram is like a traditional flowchart in that it shows the flow of control from step to step

Activity diagram Notation

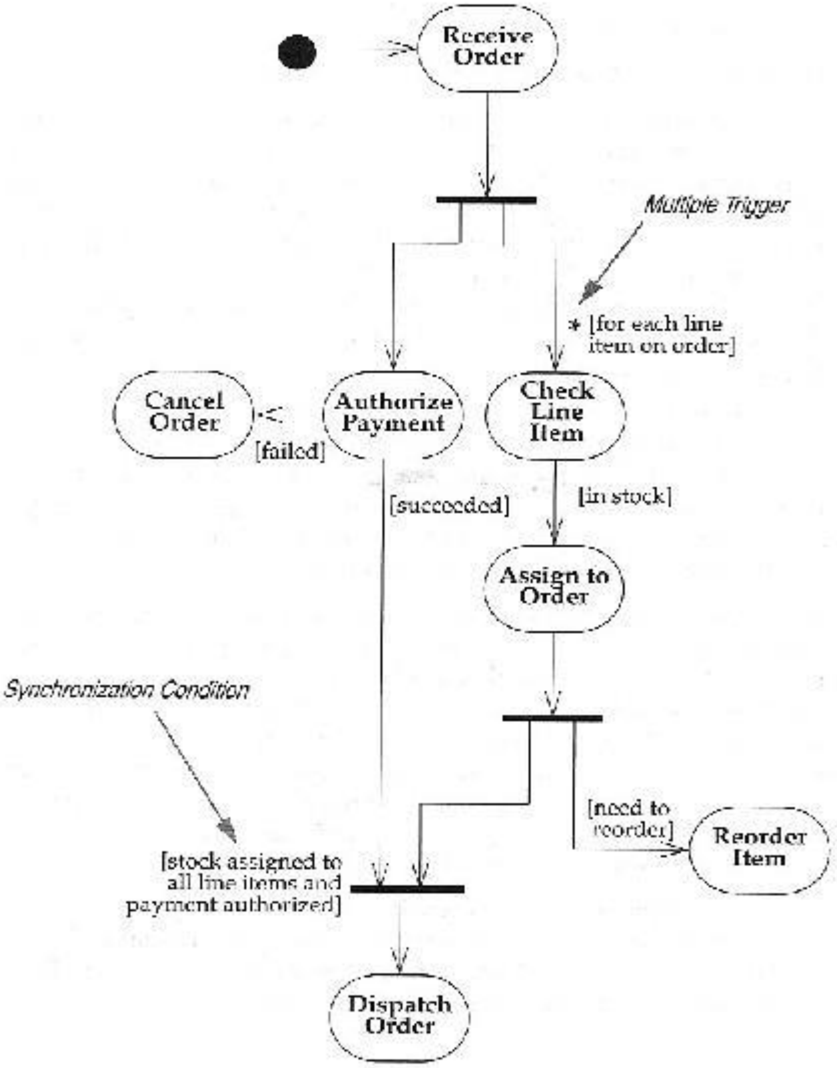
- Start at the top black circle
- If condition 1 is TRUE, go right; if condition 2 is TRUE, go down
- At first bar (a synchronization bar), break apart to follow 2 parallel paths
- At second bar, come together to proceed only when both parallel activities are done
- Activity – an oval
- Trigger – path exiting an activity
- Guard – each trigger has a guard, a logical expression that evaluates to “true” or “false”
- Synchronization Bar – can break a trigger into multiple triggers operating in parallel or can join multiple triggers into one when all are complete
- Decision Diamond – used to describe nested decisions (the first decision is indicated by an activity with multiple triggers coming out of it)



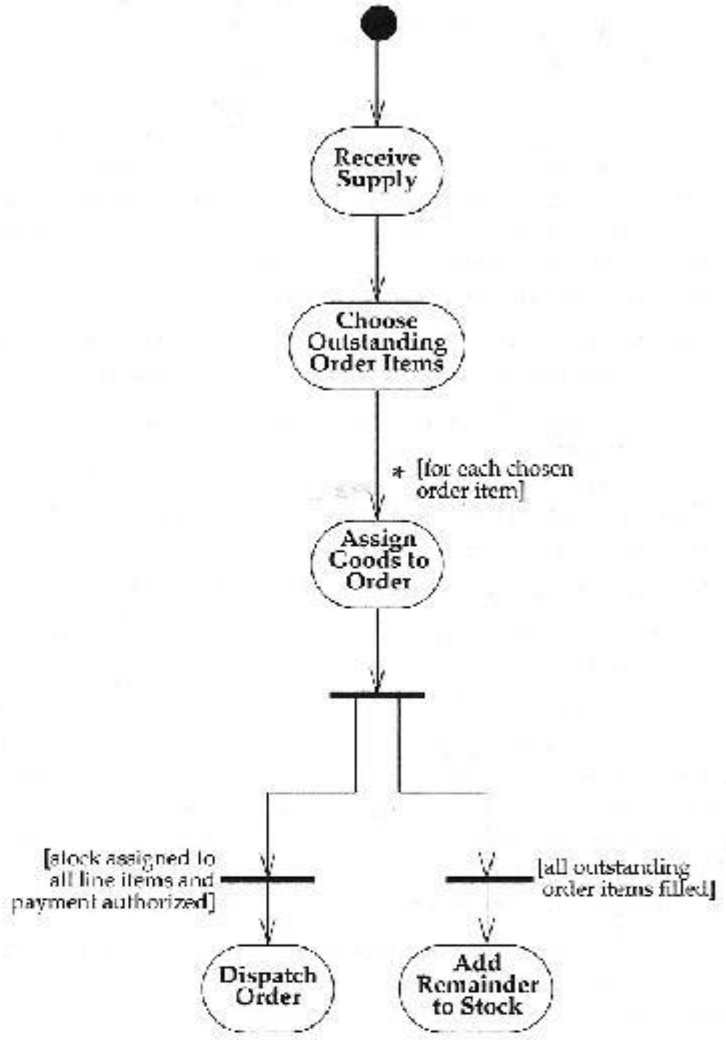
Eg:



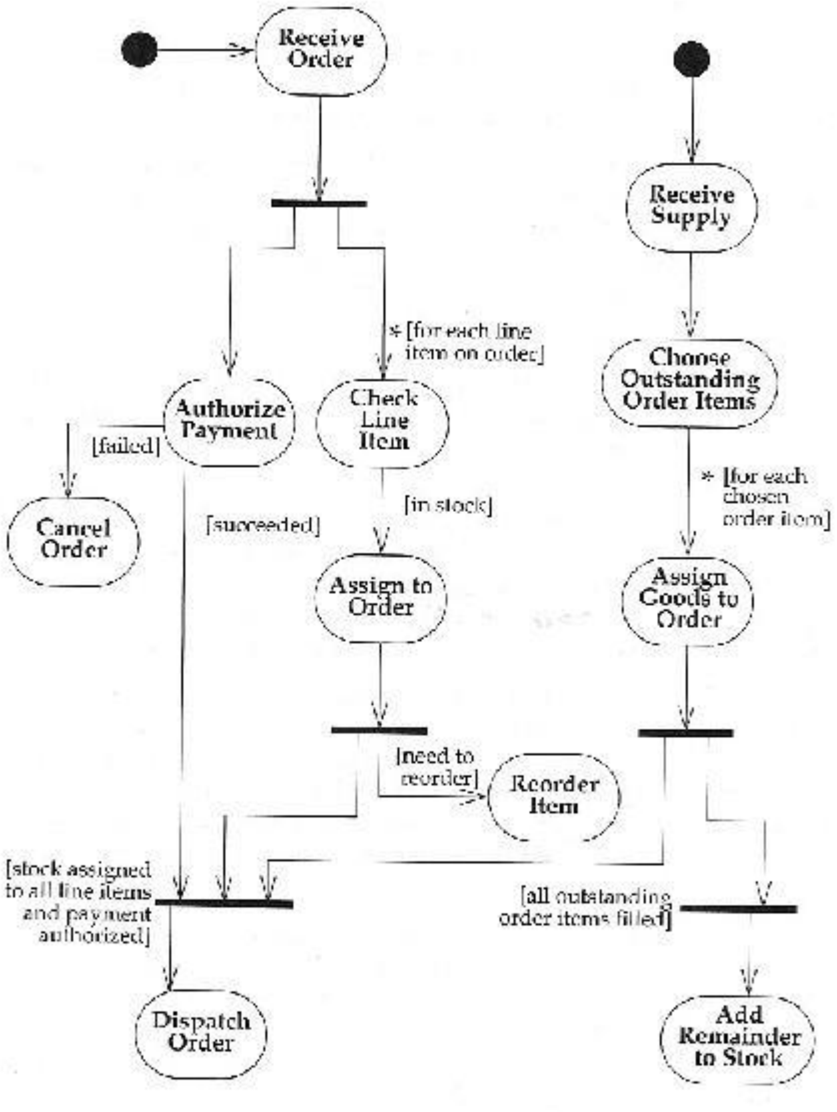
Eg: activity diagram for Use Case: Receiving an Order


Receive Order
Multiple Trigger
* [for each line item on order]
Cancel Order
Authorize Payment
Check Line Item
[failed]
[succeeded]
[in stock]
Assign to Order
Synchronization Condition
[need to reorder]
Reorder Item
[stock assigned to all line items and payment authorized]
Dispatch Order

Activity diagram for Use Case: Receiving a Supply


Receive Supply
Choose Outstanding Order Items
* [for each chosen order item]
Assign Coods to Order
[stock assigned to all line items and payment authorized]
[all outstanding order items filled]
Dispatch Order
Add Remainder to Stock

Activity diagram for Use Case: Receiving an Order and Receiving a Supply



Activity diagram for stock trade processing

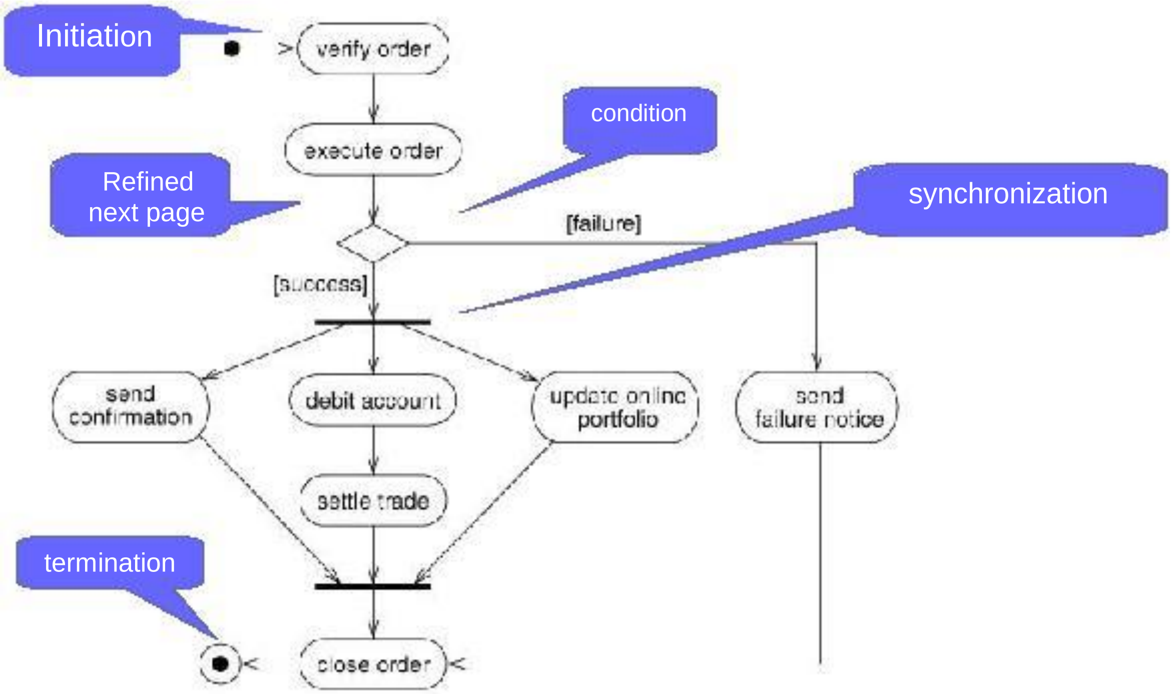


Figure 7.9 Activity diagram for stock trade processing. An activity diagram shows the sequence of steps that make up a complex process.

A Finer Activity for *execute order*

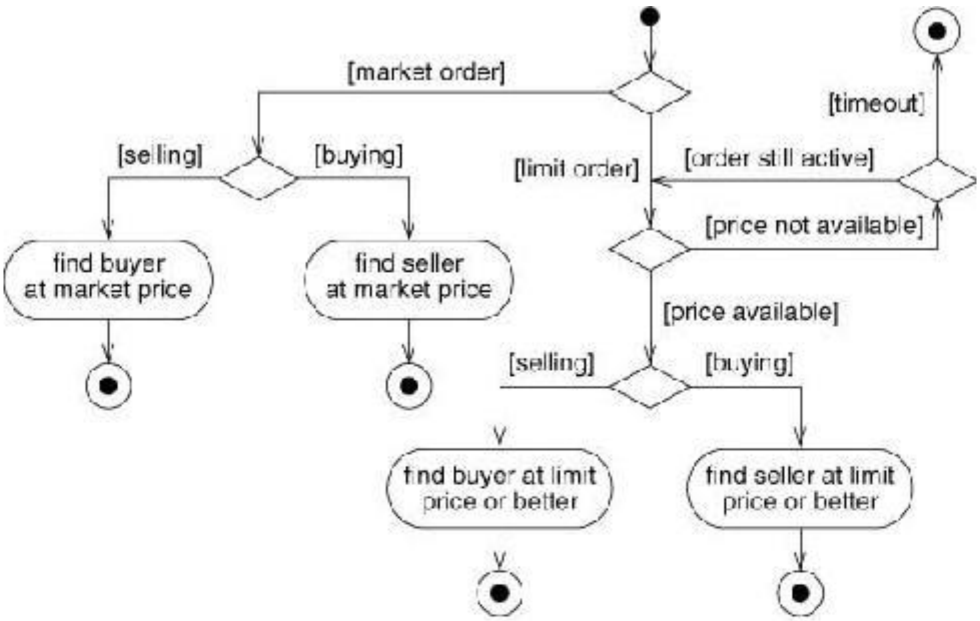


Figure 7.10 Activity diagram for *execute order.* An activity may be decomposed into finer activities.

Guidelines

- Don’t misuse activity diagrams
  - Do no be used as an excuse to develop software via flowcharts.

- Level diagrams
- Be careful with branches and conditions
- Be careful with concurrent activities
- Consider executable activity diagrams

Special constructs for activity diagrams

- Sending and receiving signals
- Swim lanes
- Object flows

Sending and Receiving Signals

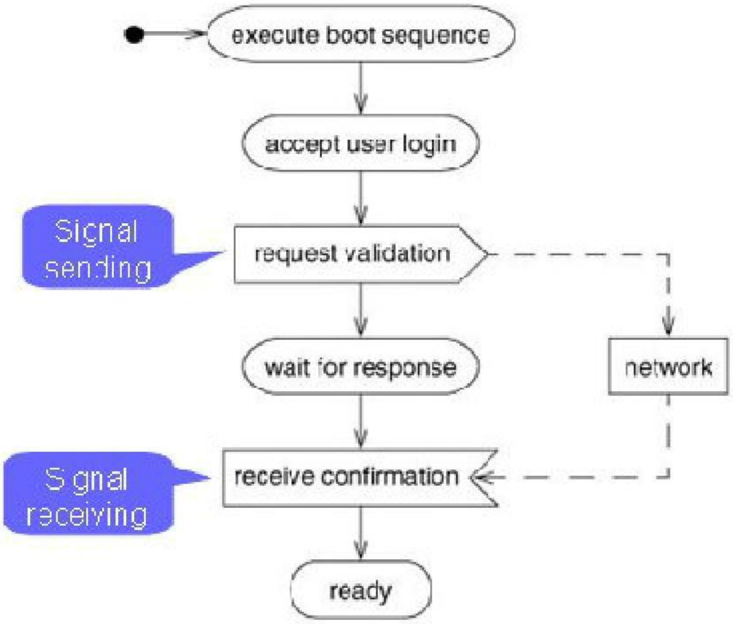


Figure 8.7 Activity diagram with signals. Activity diagrams can show fine control via sending and receiving events.

Swimlanes

- To know which human organization is responsible for an activity.

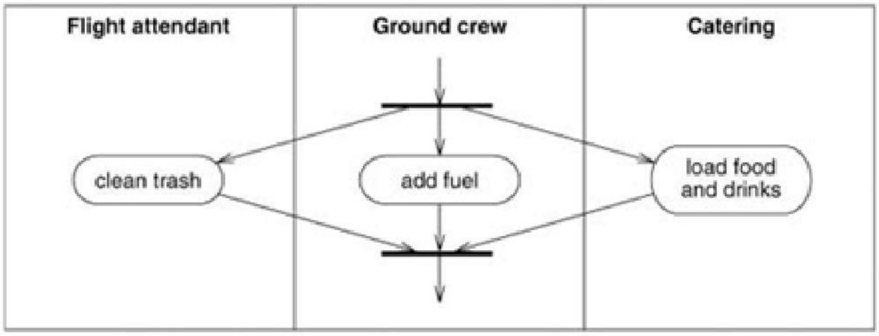
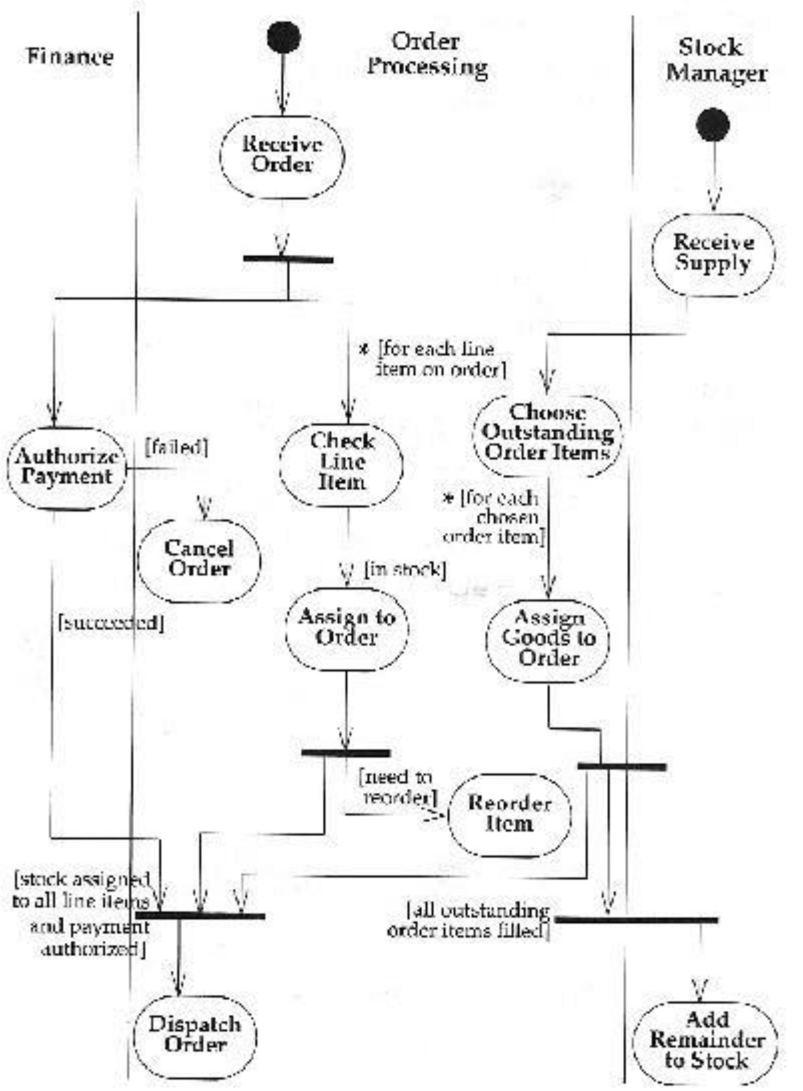


Figure 8.8 Activity diagram with swimlanes. Swimlanes can show organizational responsibility for activities.

Swimlanes - Activity Diagrams that show activities by class

- Arrange activity diagrams into vertical zones separated by lines
- Each zone represents the responsibilities of a particular class (in this example, a particular department)



Object Flows

- Show both the control and the progression of an object from state to state as activities act on it.

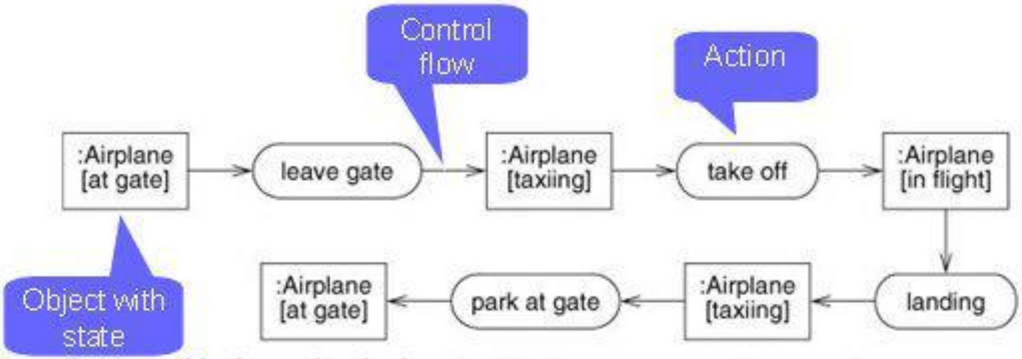


Figure 8.9 Activity diagram with object flows. An activity diagram can show the objects that are inputs or outputs of activities.

UNIT – 4 7 Hours

PROCESS OVERVIEW, SYSTEM CONCEPTION, DOMAIN ANALYSIS

Syllabus :

- ➢ **Process Overview: Development stages; Development life cycle. System Conception:**
- ➢ **Devising a system concept; Elaborating a concept;**
- ➢ **Preparing a problem statement. Domain Analysis: Overview of analysis;**
- ➢ **Domain class model; Domain state model; Domain interaction model;**
- ➢ **Iterating the analysis.**

Process overview

- A ***software development process*** provides a basis for the organized production of software, using a collection of predefined techniques and notations.

Development Stages

- System Conception
  - Conceive an application and formulate tentative requirements
- Analysis
  - Deeply understand the requirements by constructing models
- System design
  - Devise the architecture
- Class design
  - Determine the algorithms for realizing the operations
- Implementation
  - Translate the design into programming code and database structures
- Testing
  - Ensure that the application is suitable for actual use and actually satisfies requirements
- Training
  - Help users master the new application
- Deployment
  - Pl. the application in the field and gr.fully cut over from legacy application
- Maintenance

Preserve the long term viability of the application

Analysis

To specify *what* must be done.

- **Domain analysis** focuses on real-world things whose semantics the application captures.

Application analysis addresses the computer aspects of the application that are visible to users

System Design

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

Class Design

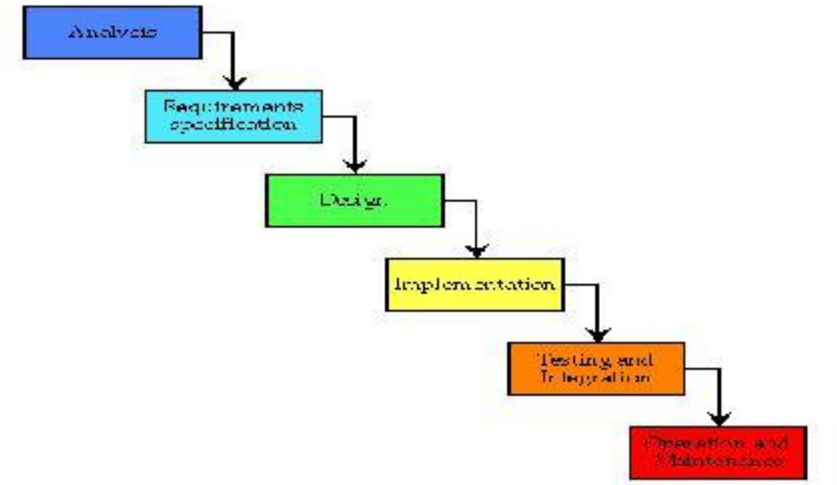
- To emphasis from application concepts toward computer concepts.
- To choose algorithms to implement major system functions.

Development Life Cycle

- Waterfall Development
- Iterative Development

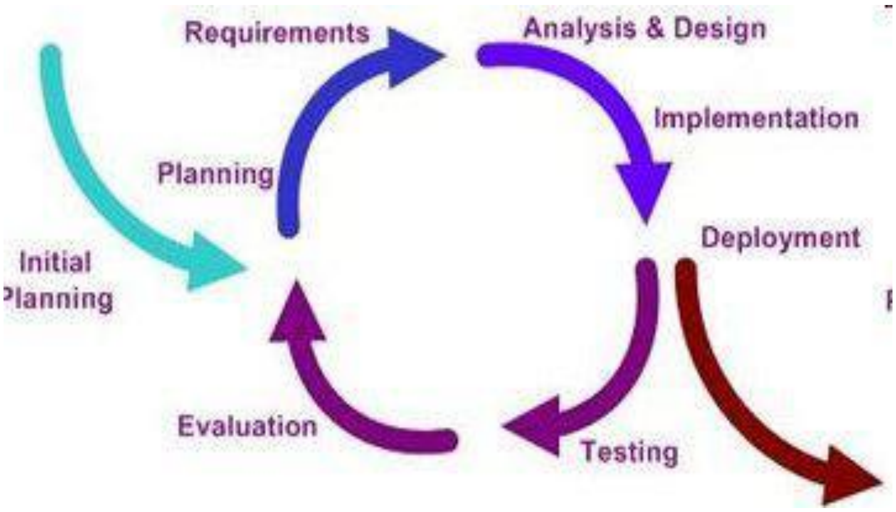
Waterfall Development

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

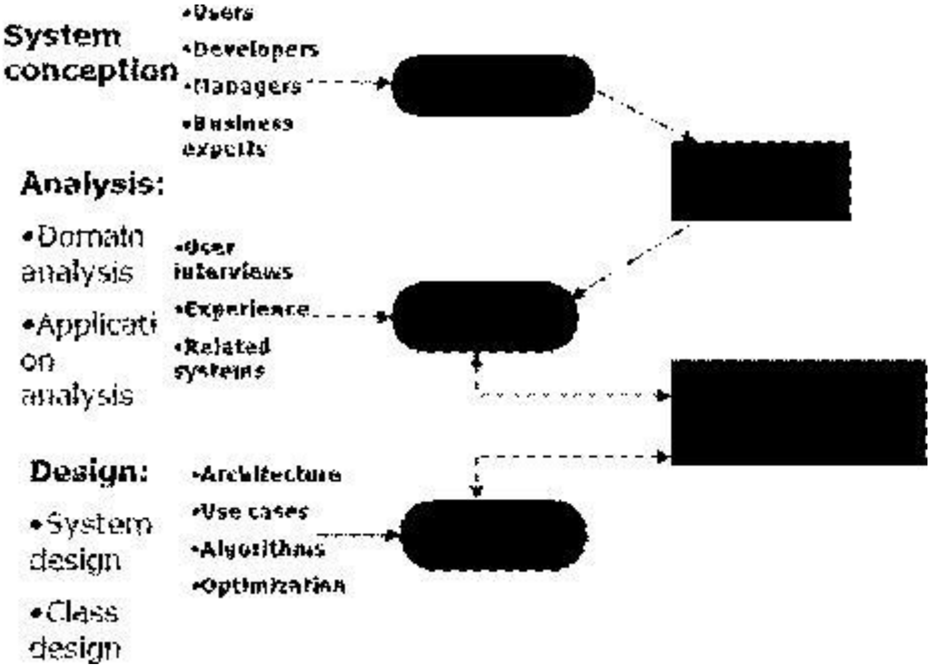


Iterative Development

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



Summary of development process for the organized production of software



System Conception

- System conception deals with the genesis of an application

Devising a System Concept

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

Elaborating a Concept

Good system concept must answer the following questions

- Who is the application for?
  - Stakeholders of the system
- What problems will it solve?
  - Features
- Where will it be used?
  - Compliment the existing base, locally, distributed, customer base
- When is it needed?
  - Feasible time, required time
- Why is it needed?
  - Business case
- How will it work?
  - Brainstorm the feasibility of the problem

The ATM Case Study

Develop software so that customers can access a bank's computers and carry out their own financial transactions without the mediation of a bank employee.

The ATM Case Study

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

Preparing a problem statement

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

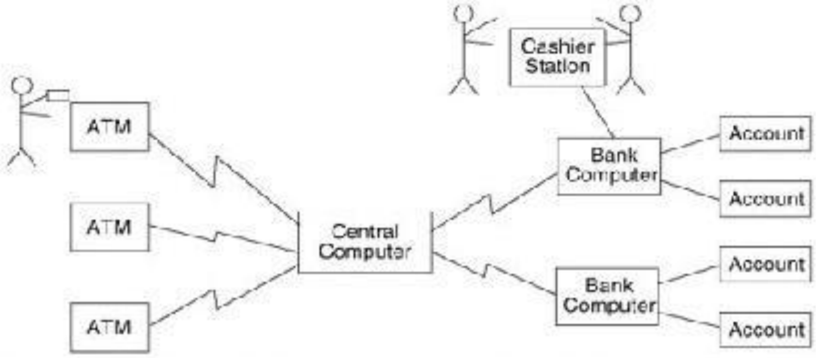


Figure 11.3 ATM network. The ATM case study threads throughout the remainder of this book

PART B

Unit: 5 APPLICATION ANALYSIS, SYSTEM DESIGN

7 Hours

Syllabus:

- ***Application Analysis: Application interaction model; Application class model; Application state model;***
- ***Adding operations. Overview of system design; Estimating performance;***
- ***Making a reuse plan; Breaking a system in to sub-systems;***
- ***Identifying concurrency; Allocation of sub-systems; Management of data***
- ***storage; Handling global resources; Choosing a software control strategy;***
- ***Handling boundary conditions; Setting the trade-off priorities; Common***
- ***Architectural styles; Architecture of the ATM system as the example.***

Application Analysis

Application Interaction Model - steps to construct model

- Determine the system boundary
- Find actors
- Find use cases
- Find initial and final events
- Prepare normal scenarios
- Add variation and exception scenarios
- Find external events
- Prepare activity diagrams for complex use cases.
- Organize actors and use cases
- Check against the domain class model

1. Determine the system boundary

- Determine what the system includes.
- What should be omitted?
- Treat the system as a black box.
- ATM example:
  - For this chapter,
    - Focus on ATM behavior and ignore cashier details.

2. Find actors

- The external objects that interact directly with the system.
- They are not under control of the application.
- Not individuals but archetypical behavior.
- ATM Example:
  - Customer, Bank, Consortium

3. Find use cases

- For each actor, list the different ways in which the actor uses the system.
- Try to keep all of the uses cases at a similar level of detail.
  - apply for loan
  - withdraw the cash from savings account
  - make withdrawal

Use Case for the ATM

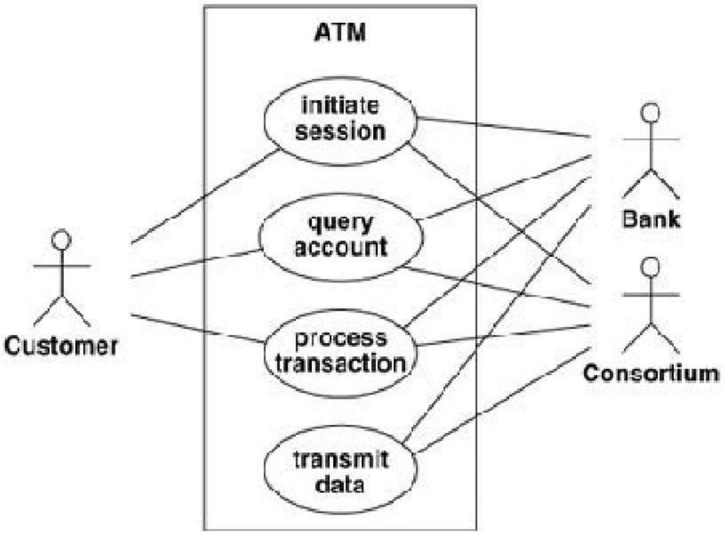
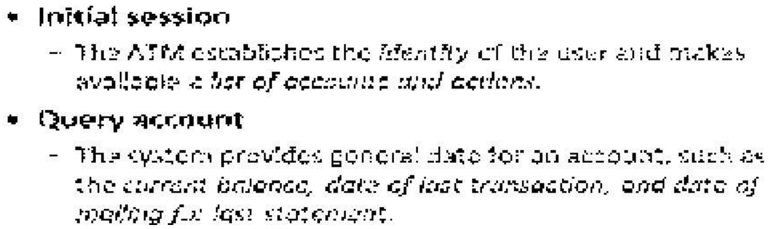
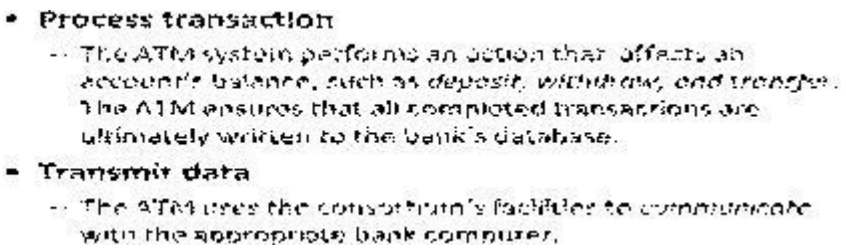


Figure 13.1 Use case diagram for the ATM. Use cases partition the



- Initial session
  - The ATM establishes the *identity* of the user and makes available *a list of accounts and actions.*
- Query account
  - The system provides general data for an account, such as the *current balance, date of last transaction, and date of mailing for last statement.*



- **Process transaction**
  - The ATM system performs an action that affects an account's balance, such as *deposit, withdraw, and transfer*. The ATM ensures that all completed transactions are ultimately written to the bank's database.
- **Transmit data**
  - The ATM uses the consortium's facilities to communicate with the appropriate bank computer.

4. Find initial and final events

Finding the initial and final events for each use case
To understand the **behavior** clearly of system
Execution sequences that cover each use case
Initial events may be

a. A request for the service that the use case provides
b. An occurrence that triggers a chain of activity

ATM example

- Initial session
  - – Initial event
    - The customer’s insertion of a cash card.
  - – final event
    - The system keeps the cash card, or
    - The system returns the cash card.

ATM example

- Query account
  - – Initial event
    - A customer’s request for account data.
  - – final event
    - The system’s delivery of account data to the customer.

ATM example

- Process transaction
  - – Initial event
    - The customer’s initiation of a transaction.
  - – final event
    - Committing or
    - Aborting the transaction

ATM example

- Transmit data
  - – Initial event
    - Triggered by a customer’s request for account data, or
    - Recovery from a network, power, or another kind of failure.
  - – final event
    - Successful transmission of data.

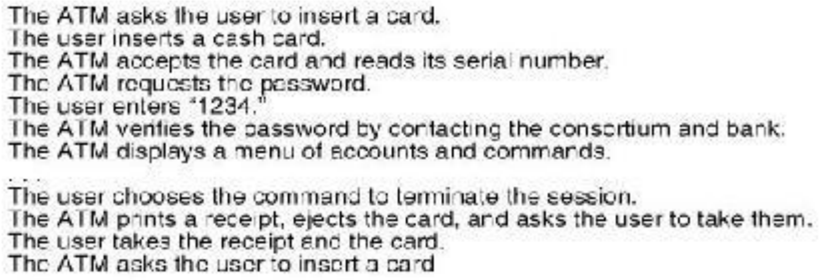
5. Prepare normal scenarios

For each use case, prepare one or more *typical dialogs*.

A scenario is a *sequence of events* among a set of ***interacting*** *objects*.
Sometimes the problem statement describes the full interaction sequence

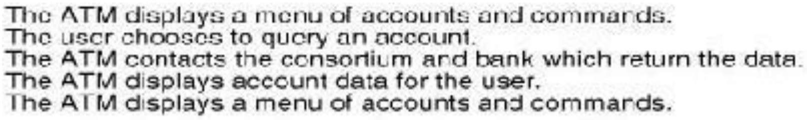
6. Normal ATM scenarios

Initiate session



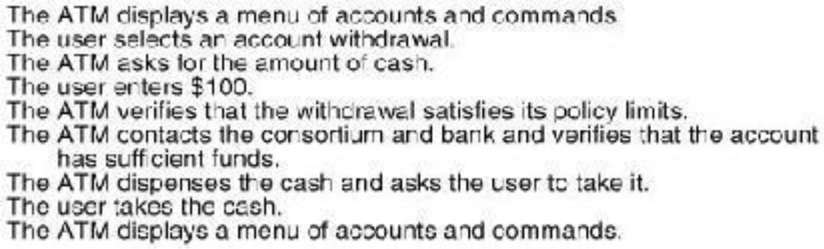
The ATM asks the user to insert a card.
The user inserts a cash card.
The ATM accepts the card and reads its serial number.
The ATM requests the password.
The user enters "1234."
The ATM verifies the password by contacting the consortium and bank.
The ATM displays a menu of accounts and commands.
. . .
The user chooses the command to terminate the session.
The ATM prints a receipt, ejects the card, and asks the user to take them.
The user takes the receipt and the card.
The ATM asks the user to insert a card

- **Query account**



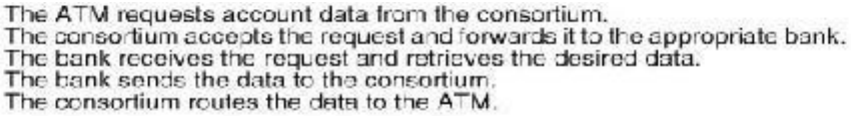
The ATM displays a menu of accounts and commands.
The user chooses to query an account.
The ATM contacts the consortium and bank which return the data.
The ATM displays account data for the user.
The ATM displays a menu of accounts and commands.

- **Process transaction**



The ATM displays a menu of accounts and commands.
The user selects an account withdrawal.
The ATM asks for the amount of cash.
The user enters $100.
The ATM verifies that the withdrawal satisfies its policy limits.
The ATM contacts the consortium and bank and verifies that the account has sufficient funds.
The ATM dispenses the cash and asks the user to take it.
The user takes the cash.
The ATM displays a menu of accounts and commands.

- **Transmit data**



The ATM requests account data from the consortium.
The consortium accepts the request and forwards it to the appropriate bank.
The bank receives the request and retrieves the desired data.
The bank sends the data to the consortium.
The consortium routes the data to the ATM.

7. Add variation and exception scenarios

Special cases

Omitted input E.g., maximum values, minimum value

Error cases

E.g. Invalid values, failures to respond

Other cases

E.g. Help requests, status queries

ATM example

- Variations and exceptions:
  - The ATM can't read the card.
  - The card has expired.
  - The ATM times out waiting for a response.

- The amount is invalid.
- The machine is out of cash or paper.
- The communication lines are down
- The transaction is rejected because of suspicious pattern of card usage.

8. Find external events

- The external events include
  - All inputs,
  - decisions,
  - interrupts, and
  - Interactions to or from users or external devices.
- An event can trigger effects for a target object.
- Use scenarios for normal
  events Sequence diagram
- Prepare a sequence diagram for each scenario.
- The sequence diagram captures the dialog and interplay between actors.
  The sequence diagram clearly shows the sender and receiver of each event
  ATM Example

Sequence diagram of the process transaction

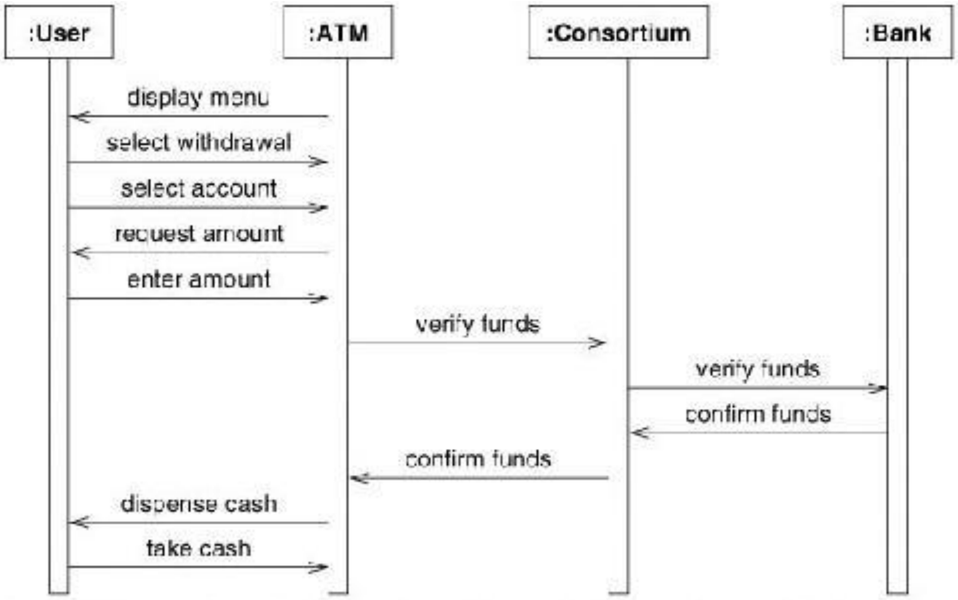


Figure 13.3 Sequence diagram for the *process transaction* scenario. A sequence diagram clearly shows the sender and receiver of each event.

- Events for the ATM case study

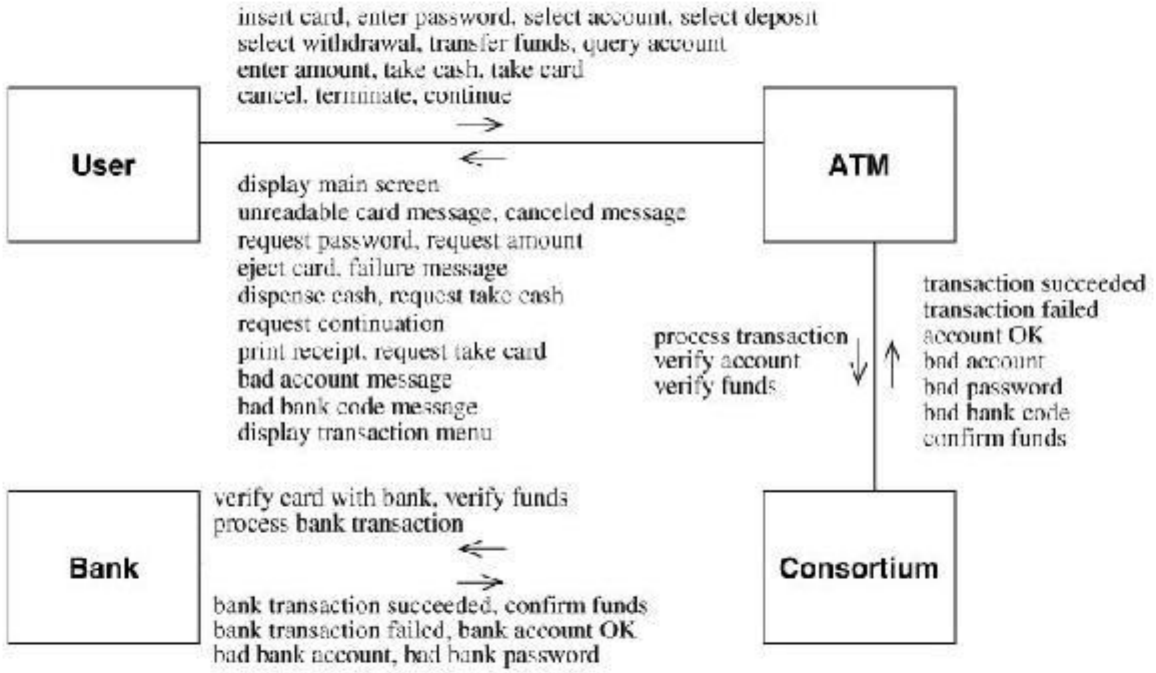

insert card, enter password, select account, select deposit
select withdrawal, transfer funds, query account
enter amount, take cash, take card
cancel, terminate, continue
User
ATM
display main screen
unreadable card message, canceled message
request password, request amount
eject card, failure message
dispense cash, request take cash
request continuation
print receipt, request take card
bad account message
bad bank code message
display transaction menu
process transaction
verify account
verify funds
transaction succeeded
transaction failed
account OK
bad account
bad password
bad bank code
confirm funds
Bank
Consortium
verify card with bank, verify funds
process bank transaction
bank transaction succeeded, confirm funds
bank transaction failed, bank account OK
bad bank account, bad bank password

Figure 13.4 Events for the ATM case study. Tally the events in the scenarios and note the classes that send and receive each event.

9. Activity Diagram

- Activity diagram shows behaviors like alternatives and decisions.
- Prepare activity diagrams for complex use cases.
- Appropriate to document business logic during analysis
- Do not use activity diagram as an excuse to begin implementation.

ATM Example

Activity diagram for card verification

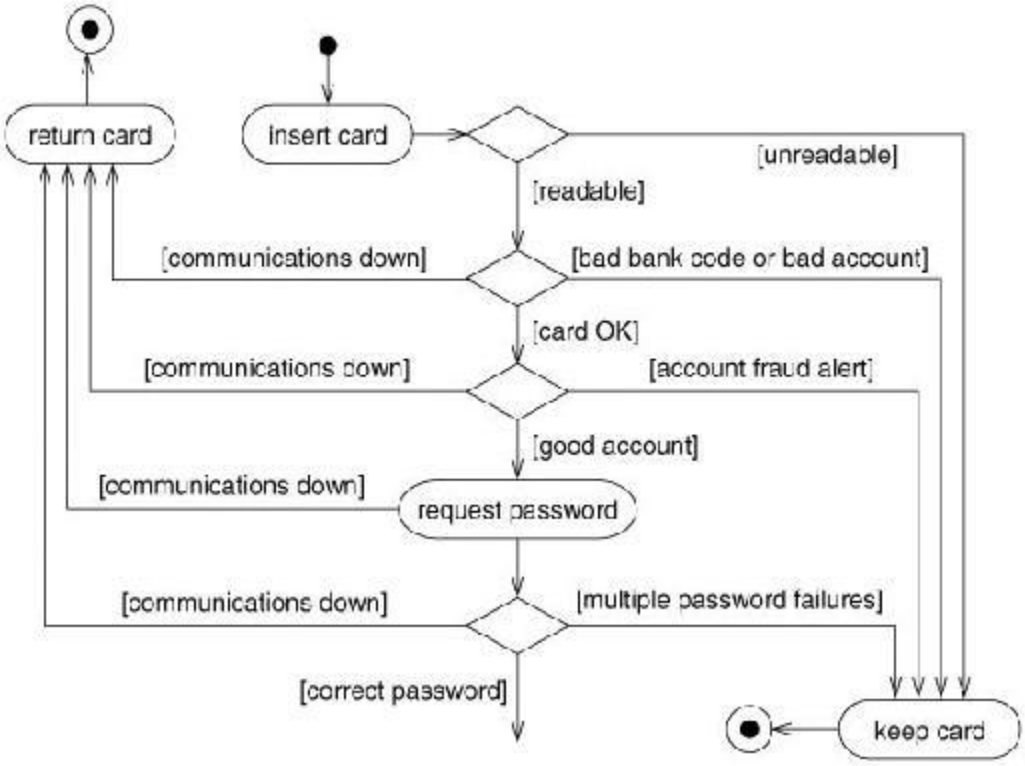


Figure 13.5 Activity diagram for card verification. You can use activity diagrams to document business logic but do not use them as an excuse to begin premature implementation.

10. Organize actors and use cases

- Organize use cases with relationships
  - Include, extend, and generalization
- Organize actors with generalization.

ATM Example

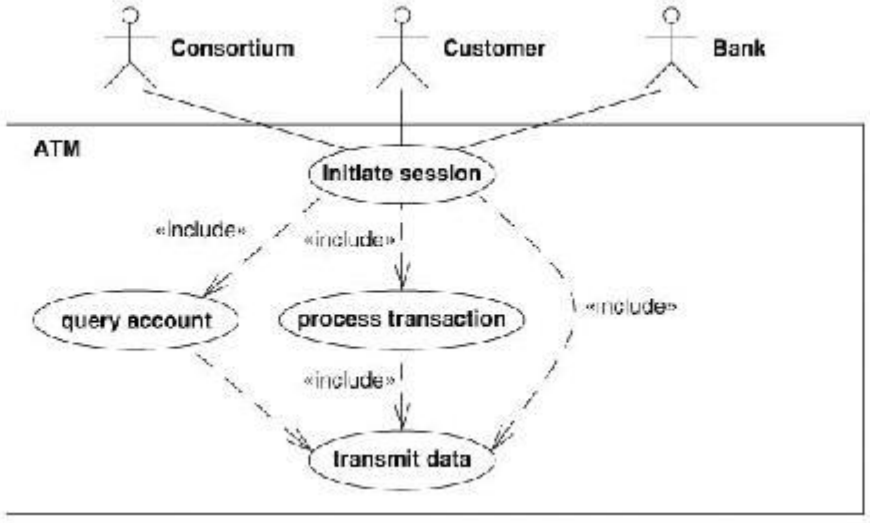
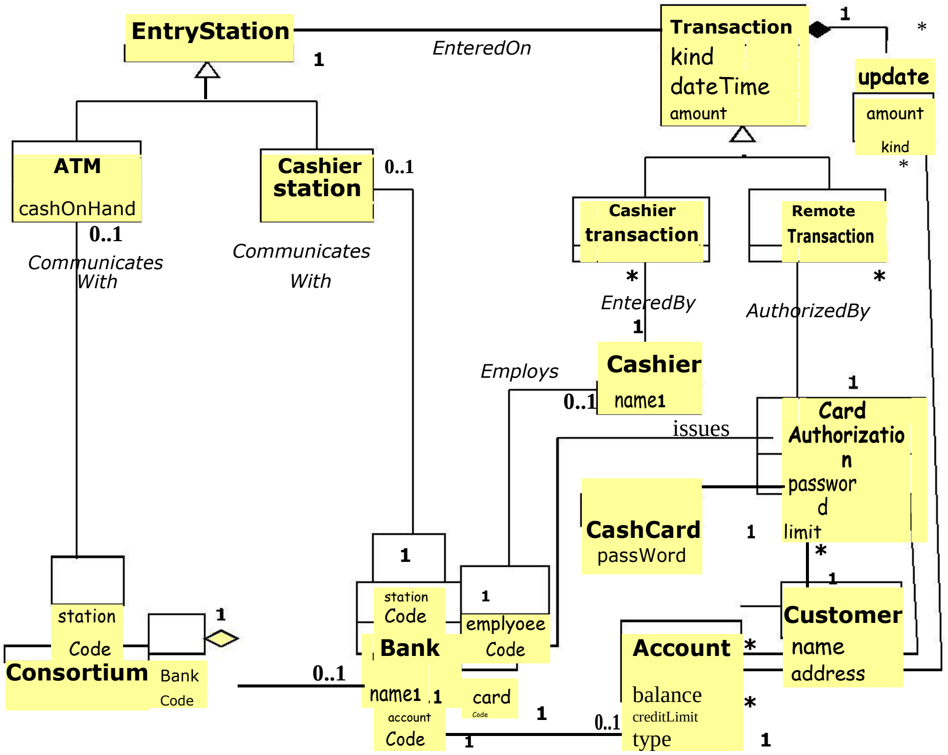


Figure 13.6 Organizing use cases. Once the basic use cases are identified, you can organize them with relationships.

11. Checking Against the Domain Class Model

- The application and domain models should be mostly consistent.
- The actors, use cases, and scenarios are all based on classes and concepts from the domain model.
- Examine the scenarios and make sure that the domain model has all the necessary data.
- Make sure that the domain model covers all event parameters.



Application Class Model

- Application classes define the application itself, rather than the real-world objects that the application acts on
- Most application classes are *computer-oriented* and define the way that users perceive the applications

Application Class Model – steps

1. Specify user interf.s
2. Define boundary classes
3. Determine controllers
4. Check against the interaction model

1. **Specify user interf.s**

   User interf.

   a. Is an object or group of objects

   b. Provide user a way to access system's

      i. domain objects,

      ii. commands, and

      iii. Application options.

   Try to determine the commands that the user can perform.

   A command is a large-scale request for a service,

   c. E.g.

      i. Make a flight reservation

      ii. Find matches for a phrase in a database

   Decoupling application logic from the user interf.. ATM example - The details are not important at this point.

   - The important thing is the information exchanged.

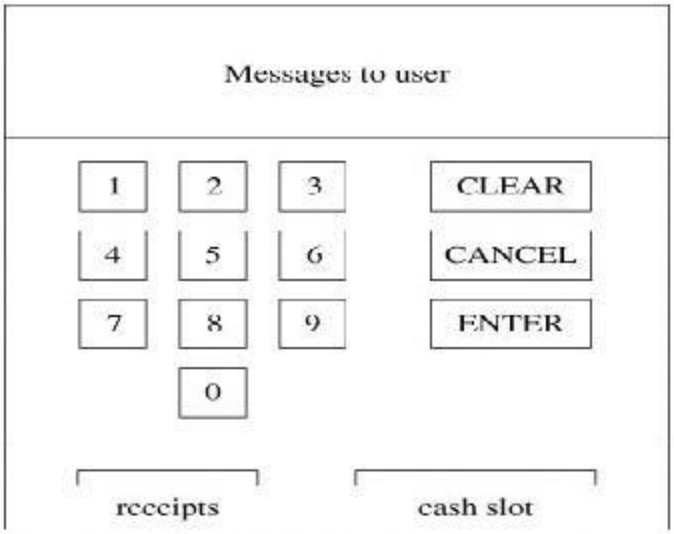


Figure 13.7 Format of ATM Interface. Sometimes a sample interface can help you visualize the operation of an application

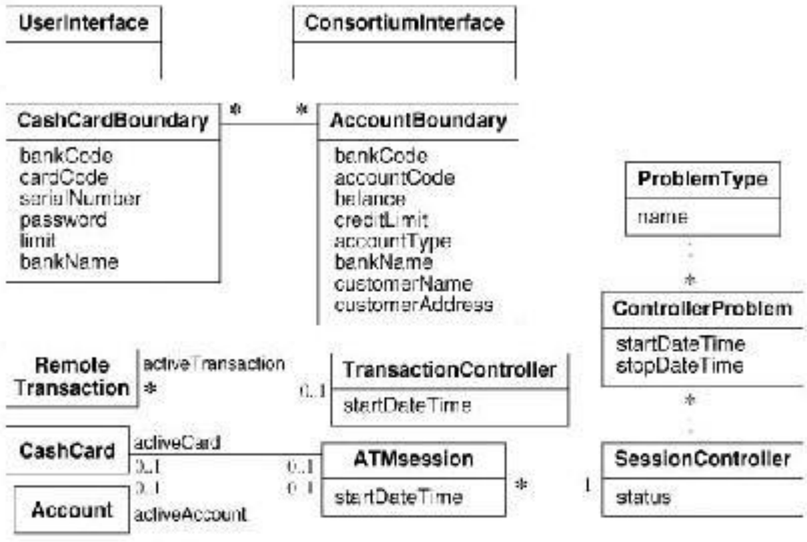


Figure 13.8 ATM application class model. Application classes augment the domain classes and are necessary for development.

2. Defining Boundary Classes

- A boundary class
  - Is an area for communications between a system and external source.
  - Converts information for transmission to and from the internal system.
- ATM example
  - *CashCardBoundary*
  - *AccountBoundary*
  - Between the ATM and the consortium

ATM Example

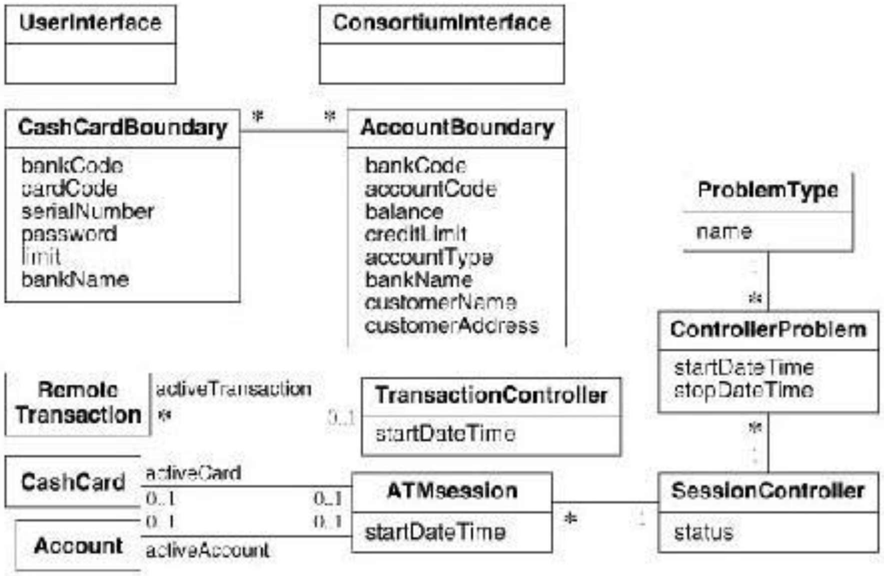


Figure 13.6 ATM application class model. Application classes augment the domain classes and are necessary for development.

3. Determining Controllers

- **Controller** is an active object that manages control within an application.
- Controller
  - *Receives* signals from the outside world or
  - Receives signals from objects within the system,
  - *Reacts* to them,
  - *Invokes* operation on the objects in the system, and
  - *Sends* signals to the outside world.

ATM Example

- There are two controllers
  - The outer loop verifies customers and accounts.
  - The inner loop services transactions.

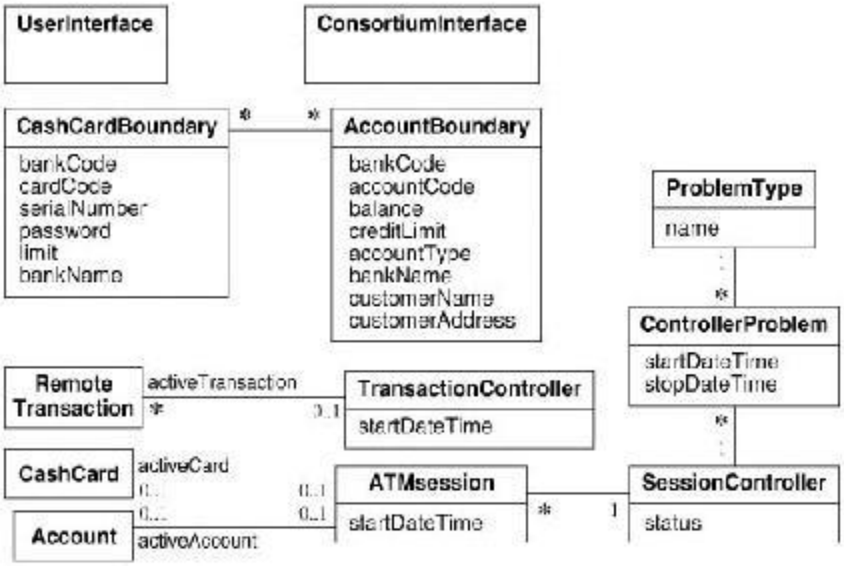
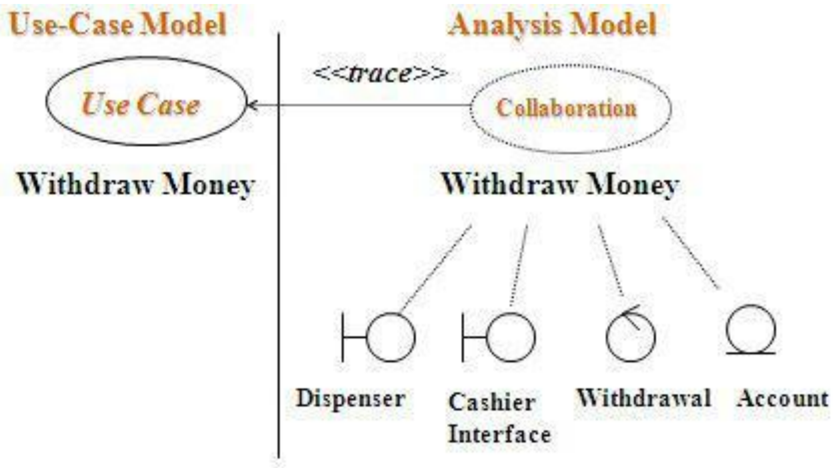


Figure 13.8 ATM application class model. Application classes augment the domain classes and are necessary for development.

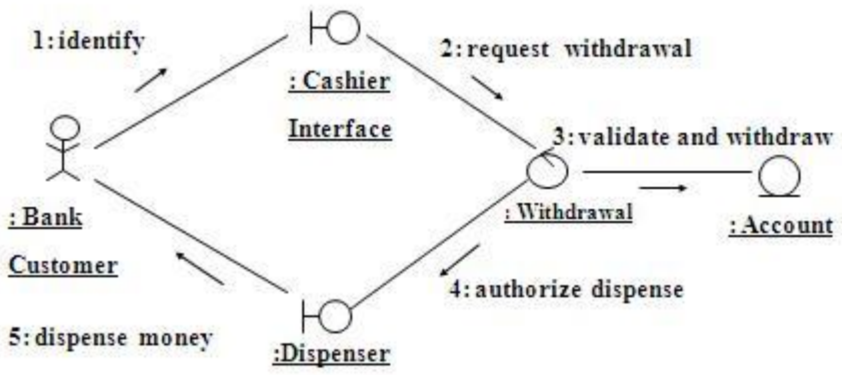
Analysis Stereotypes

- <<boundary>> classes in general are used to model interaction between the system and its actors.
- <<entity>> classes in general are used to model information that is long-lived and often persistent.
- <<control>> classes are generally used to represent coordination, sequencing, transactions, and control of other objects. And it is often used to encapsulate control related to a specific use case.

The Realization of a Use Case in the Analysis Model

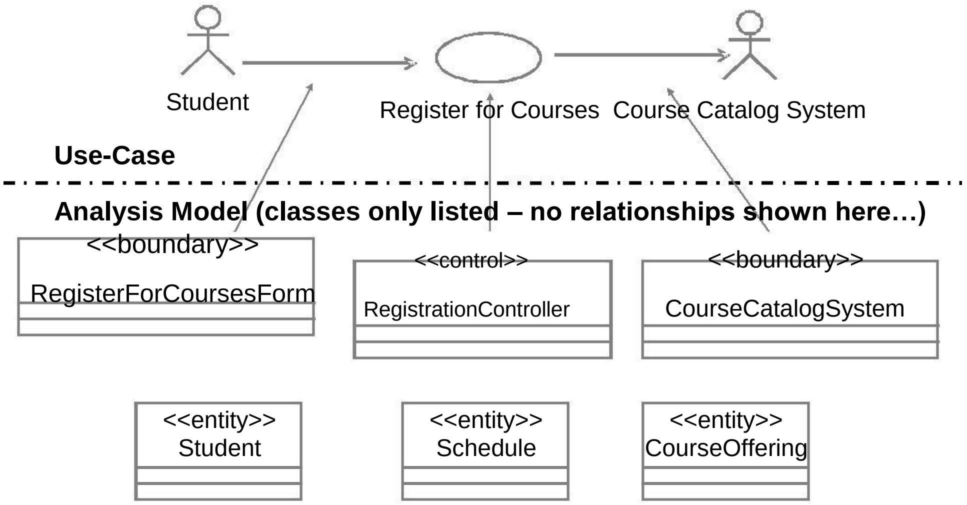


A collaboration diagram for the Withdraw Money use-case realization in the analysis model



Example:Analysis Classes

- The diagram shows the **classes participating** in the Register for Courses use case



Use-Case

Analysis Model (classes only listed – no relationships shown here…)

4. Checking Against the Interaction Model

- Go over the use cases and think about how they would work.
- When the domain and application class models are in pl., you should be able to simulate a use case with the classes.

ATM Example

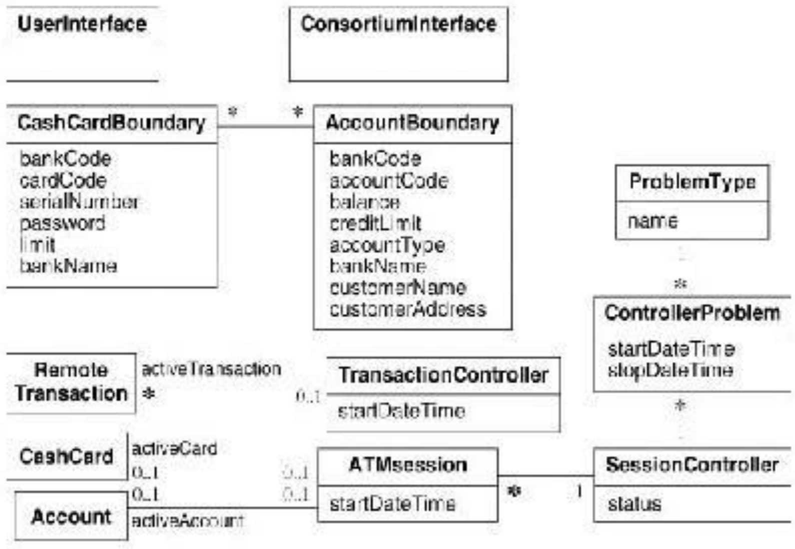


Figure 13.8 ATM application class model. Application classes augment the domain classes and are necessary for development.

Application State Model

- The application state model focuses on application classes
- Augments the domain state model

Application State Model- steps

1. Determine Application Classes with States
2. Find events
3. Build state diagrams
4. Check against other state diagrams
5. Check against the class model
6. Check against the interaction model

1. Determine Application Classes with States

- Good candidates for state models
  - User interf. classes
  - Controller classes
- ATM example
  - The controllers have states that will elaborate.

2. Find events

- Study scenarios and extract events.
- In domain model
  - Find states and then find events
- In application model
  - Find events first, and then find states
- ATM example
  - Revisit the scenarios, some events are:
  - *Insert card, enter password, end session and take card.*

3. Building State Diagrams

- To build a state diagram for each application class with temporal behavior.

- Initial state diagram
  - Choose one of these classes and consider a sequence diagram.
  - The initial state diagram will be a sequence of events and states.
  - Every scenario or sequence diagram corresponds to a path through the state diagram.
- Find loops
  - If a sequence of events can be repeated indefinitely, then they form a loop.
- Merge other sequence diagrams into the state diagram.
- After normal events have been considered, add variation and exception cases.
- The state diagram of a class is finished when the diagram covers all scenarios and the diagram handles all events that can affect a state.
- Identify the classes with multiple states
- Study the interaction scenarios to find events for these classes
- Reconcile the various scenarios
- Detect overlap and closure of loops

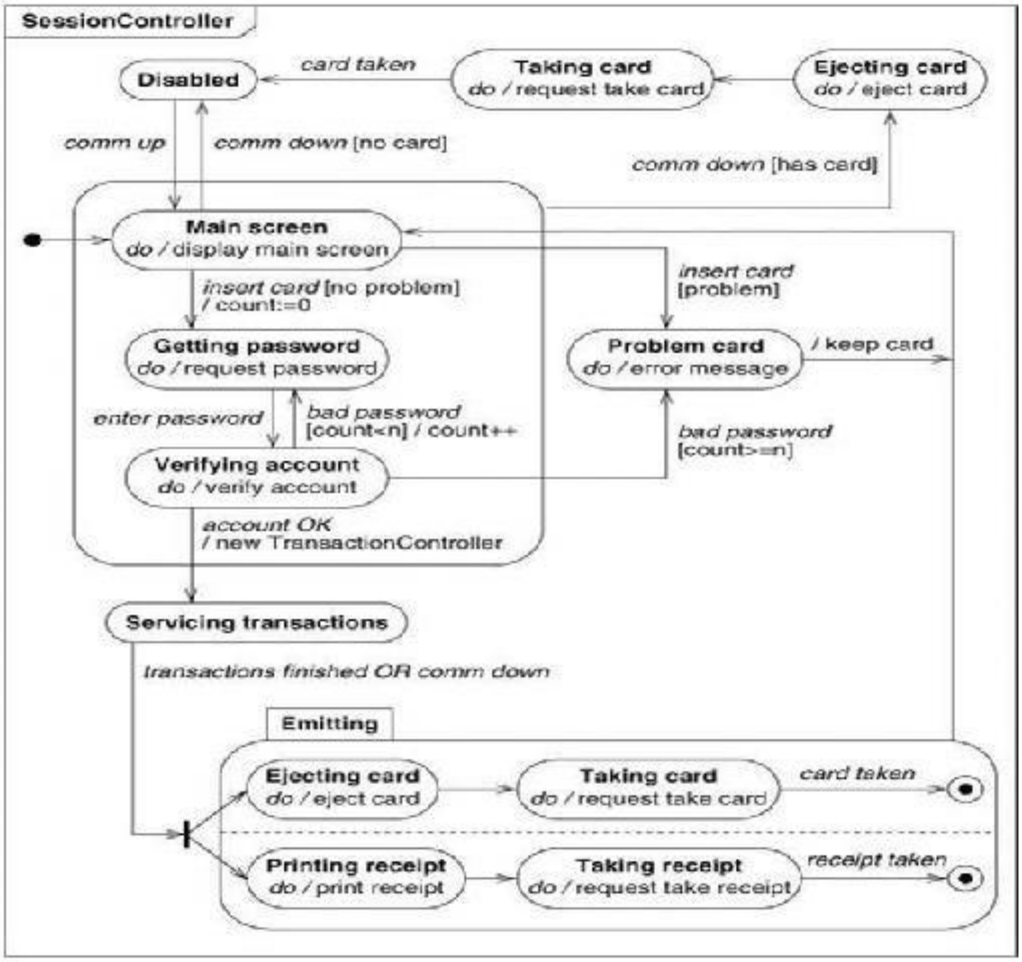


Figure 13.9 State diagram for *SessionController*. Build a state diagram for each application class with temporal behavior.

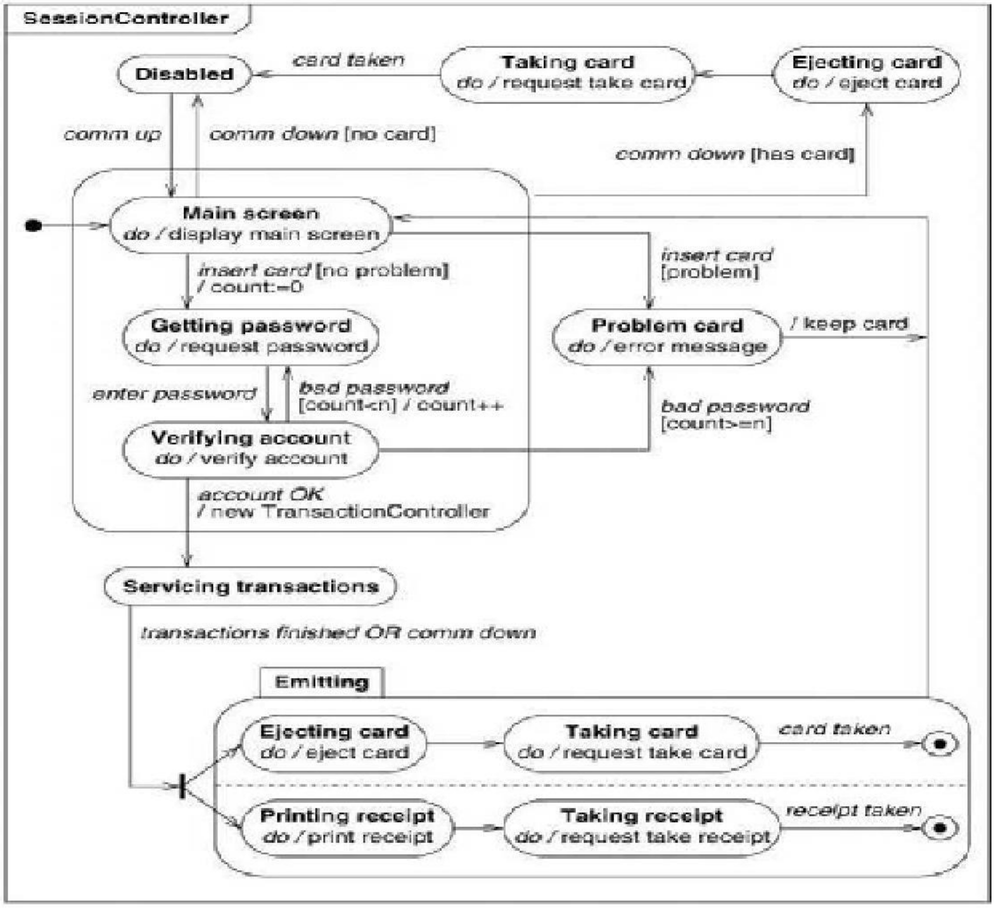


Figure 13.9 State diagram for *SessionController*. Build a state diagram for each application class with temporal behavior.

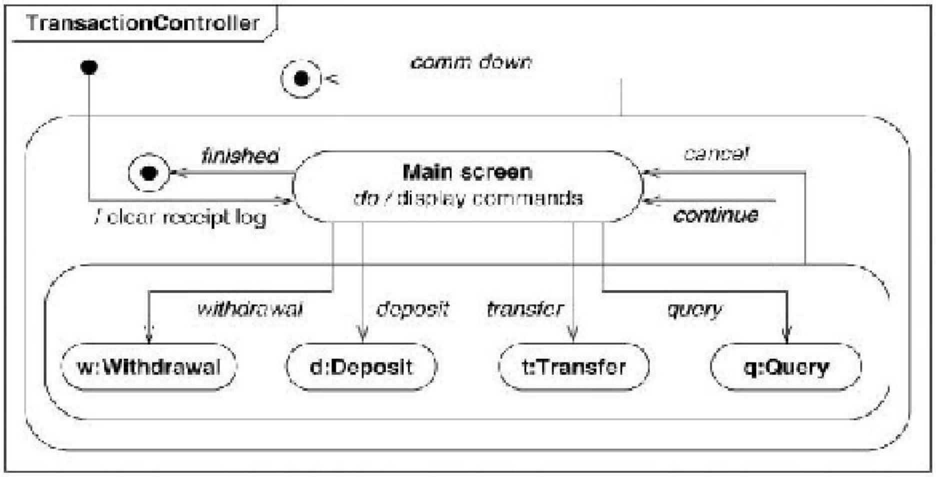


Figure 13.10 State diagram for *TransactionController*. Obtain information from the scenarios of the interaction model.

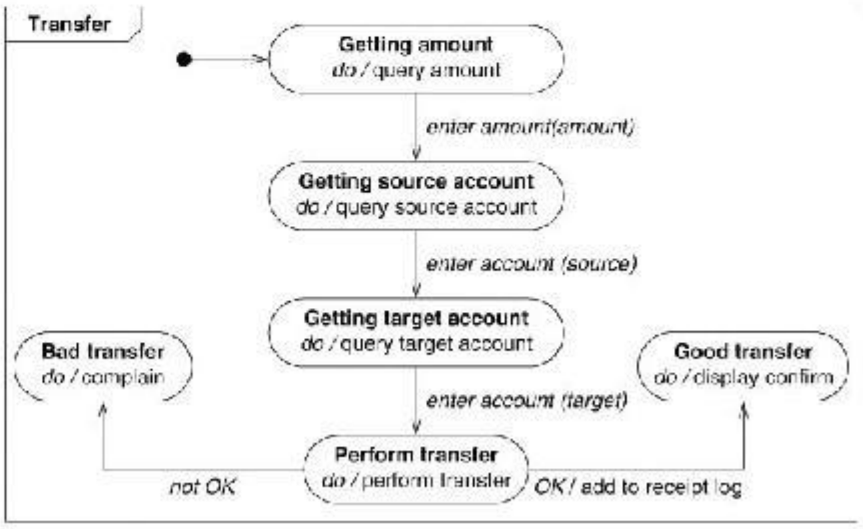


Figure 13.11 **State diagram for *Transfer*.** This diagram elaborates the *Transfer* state in Figure 13.10

4. **check against other state diagrams**

- Every event should have a sender and a receiver.
- Follow the effects of an input event from object to object through the system to make sure that they match the scenarios.
- Objects are inherently concurrent.
- Make sure that corresponding events on different state diagrams are consistent.
- ATM example
  - The *SessionController* initiates the *TransactionController*,
  - The termination of the *TransactionController* causes the *SessionController* to resume.

5. **Check against the class model**

- ATM example
  – Multiple ATMs can potentially concurrently access an account.
  – Account access needs to be controlled to ensure that only one update at a time is applied.

6. **Check against the interaction model**

- Check the state model against the scenarios of the interaction model.
- Simulate each behavior sequence by hand and verify the state diagrams.
- Take the state model and tr. out legitimate paths.

Adding Operations

- Operations from the class model
- Operations from use cases
- Shopping-list operations
- Simplifying operations

Operations from the class model

- The reading and writing of attribute values and association links.
- Need not show them explicitly.

Operations from use cases

- Use cases lead to activities.
- Many of these activities correspond to operations on the class model.
- ATM example
  - *Consortium → verifyBankCode.*
  - *Bank → verifyPassword.*
  - *ATM → verifyCashCard*

Shopping-List Operations

- The real-world behavior of classes suggests operations.
- Shopping-list operations provide an opportunity to broaden a class definition.

ATM example

- Account.close()
- Bank.createSavingsAccount(customer):account
- Bank.createCheckingAccount(customer):account
- Bank.createCashCardAuth(customer);cashCardAuthorization

Simplifying Operations

- Try to broaden the definition of an operation to encompass similar operations.
- Use inheritance to reduce the number of distinct operations.

ATM domain class model

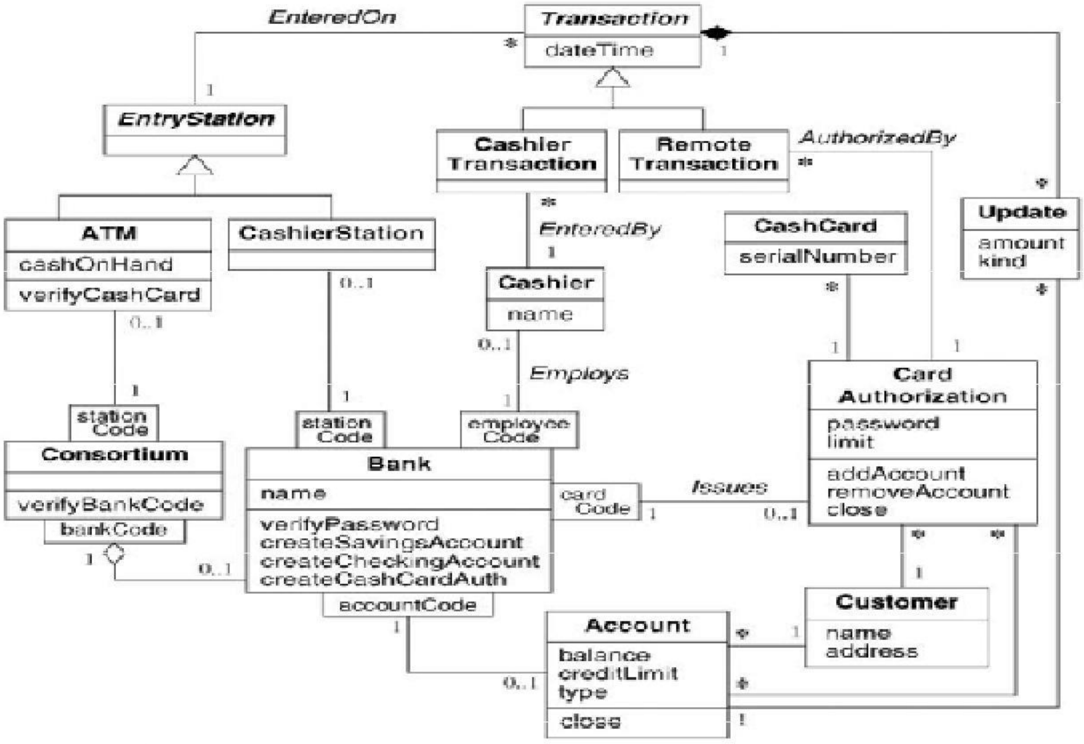
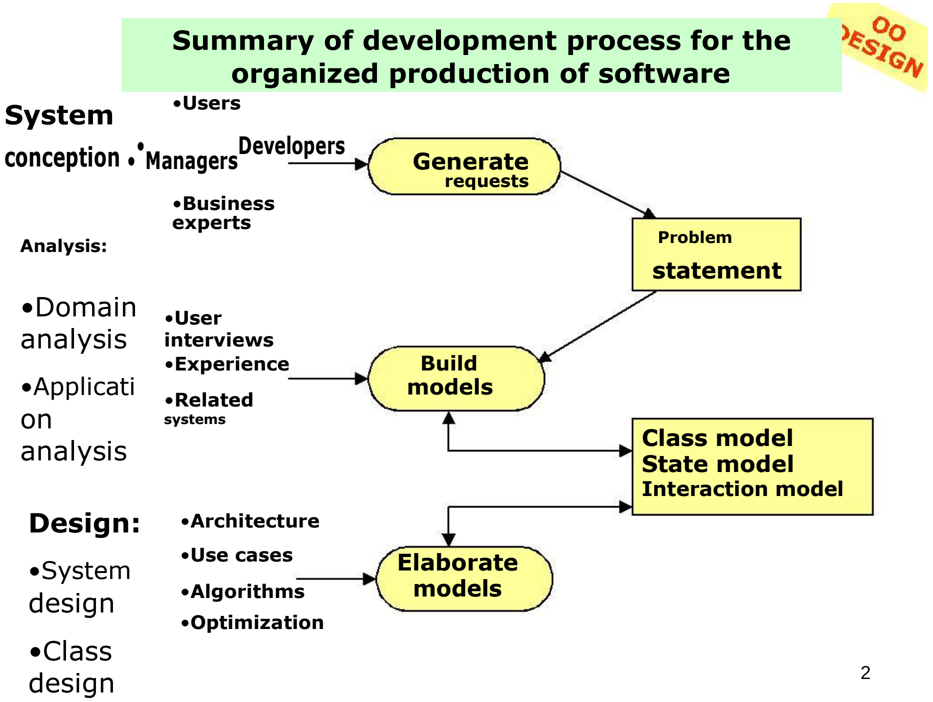


Figure 13.12 ATM domain class model with some operations.

Overview of System Design



Summary of development process for the organized production of software

OO DESIGN

System conception

- Users
- Managers
- Developers
- Business experts

Generate requests

Problem statement

Analysis:

- Domain analysis
- Application analysis

- User interviews
- Experience
- Related systems

Build models

Class model
State model
Interaction model

Design:

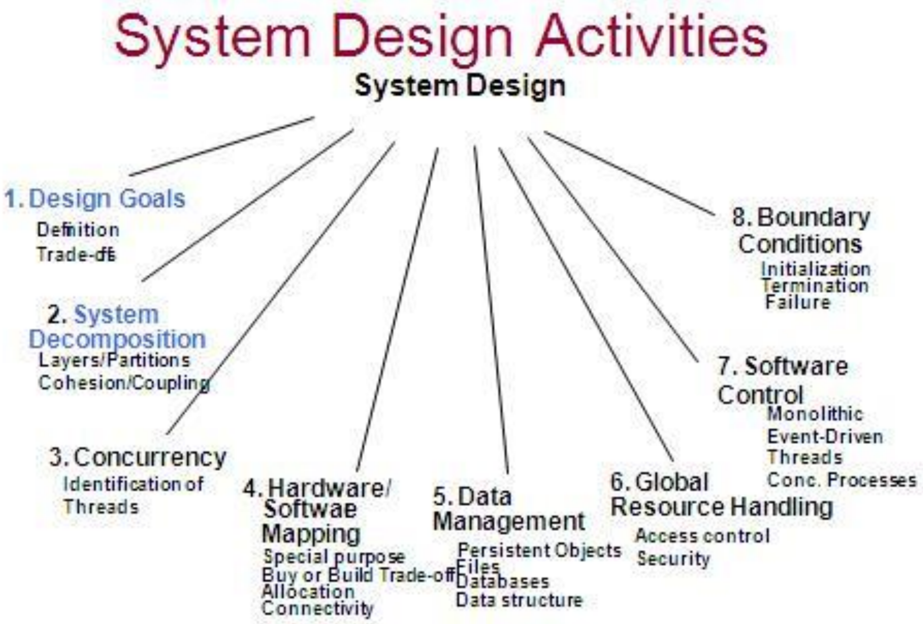
- System design
- Class design

- Architecture
- Use cases
- Algorithms
- Optimization

Elaborate models

2

- Analysis – focus is on *what* needs to be done; independent of *how it* is done
- Design – focus is on decisions about how the problem will be solved
    - First at high level
    - Then with more detail
- *System Design –*
    - first design stage
    - Overall structure and style
    - Determines the organization of the system into subsystems
    - Context for detailed decisions about how the problem will be solved



System Design Activities

System Design

1. Design Goals
Definition
Trade-offs

2. System Decomposition
Layers/Partitions
Cohesion/Coupling

3. Concurrency
Identification of Threads

4. Hardware/ Softwae Mapping
Special purpose
Buy or Build Trade-off
Allocation
Connectivity

5. Data Management
Persistent Objects
Files
Databases
Data structure

6. Global Resource Handling
Access control
Security

7. Software Control
Monolithic
Event-Driven
Threads
Conc. Processes

8. Boundary Conditions
Initialization
Termination
Failure

Estimate system performance

- To determine if the system is feasible
- To make simplifying assumptions

ATM Example

- Suppose
  - The bank has 40 branches, also 40 terminals.
  - On a busy day half the terminals are busy at once.
  - Each customer takes one minute to perform a session.
  - A peak requirement of about 40 transactions a minute.
  - *storage*
  - Count the number of customers.
  - Estimate the amount of data for each customer.
  - :
  - :

Make a reuse plan

- Two aspects of reuse:
  - Using existing things
  - Creating reusable new things
- Reusable things include:
  - Models
  - Libraries
  - Frameworks
  - Patterns

Reusable Libraries

- A library is a collection of classes that are useful in many contexts.
- Qualities of “Good” class libraries:
  - *Coherence* – well focused themes
  - *Completeness* – provide complete behavior
  - *Consistency* - polymorphic operations should have consistent names and signatures across classes
  - *Efficiency* – provide alternative implementations of algorithms
  - *Extensibility* – define subclasses for library classes
  - *Genericity* – parameterized class definitions
- Problems limit the reuse ability:
  - Argument validation
    - Validate arguments by collection or by individual
  - Error Handling
    - Error codes or errors
  - Control paradigms
    - Event-driven or procedure-driven control
  - Group operations
  - Garbage collection

  - Name collisions

Reusable Frameworks

- A framework is a skeletal structure of a program that must be elaborated to build a complete application.
- Frameworks class libraries are typically application specific and not suitable for general use.

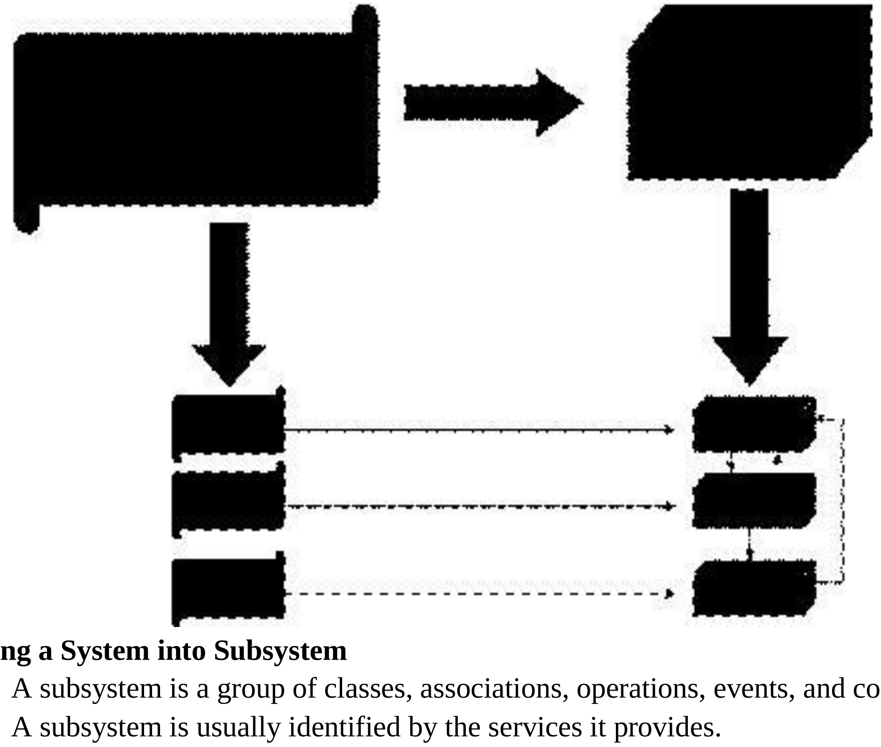
Reusable Patterns

- A pattern is a proven solution to a general problem.
- There are patterns for analysis, architecture, design, and implementation.
- A pattern is more likely to be correct and robust than an untested, custom solution.
- Patterns are prototypical model fragments that distill some of the knowledge of experts. Pattern vs. Framework
- A pattern is typically a small number of classes and relationships.
- A framework is much broader in scope and covers an entire subsystem or application.

ATM example

- Transaction
- Communication line
  Breaking a System into Subsystem
- Each subsystem is based on some common theme, such as
  - Similar functionality
  - The same physical location, or
  - Execution on the same kind of hardware.

Software Architecture



Breaking a System into Subsystem

- A subsystem is a group of classes, associations, operations, events, and constrains.
- A subsystem is usually identified by the services it provides.
- Each subsystem has a well-defined interf. to the rest of the system.

- The relation between two subsystems can be
  - Client-server relationship
  - Peer-to-peer relationship

The decomposition of systems

- Subsystems is organized as a sequence of
  - Horizontal layers,
  - Vertical partitions, or
  - Combination of layers and partitions.

Layered system

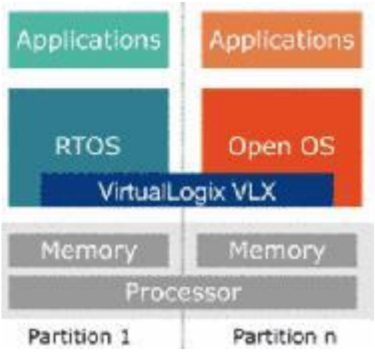
- Each built in terms of the ones below it.
- The objects in each layer can be independent.
- E.g.
  - A client-server relationship
- Problem statement specifies only the top and bottom layers:
  - The top is the desired system.
  - The bottom is the available resources.
- The intermediate layers is than introduced.
- Two forms of layered architectures:
  - Closed architecture
    - Each layer is built only in terms of the immediate lower layer.
  - Open architecture
    - A layer can use features on any lower layer to any depth.
    - Do not observe the principle of information hiding.

Partitioned System

- Vertically divided into several subsystems
- Independent or weakly coupled
- Each providing one kind of service.
- E.g. A computer operating system includes
  - File system
  - Process control
  - Virtual memory management
  - Device control

Partitions vs. Layers

- Layers vary in their level of abstraction.
- Layers depend on each other.
- Partitions divide a system into pieces.
- Partitions are peers that are independent or mutually dependent. (peer-to-peer relationship)



Combining Layers and Partitions

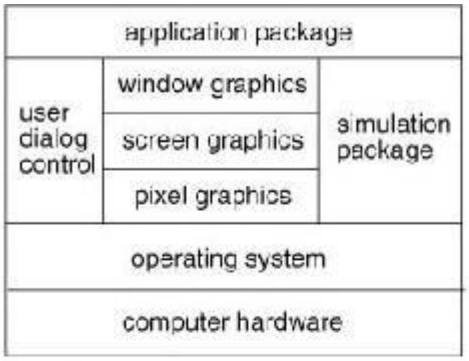


Figure 14.1 Block diagram of a typical application. Most large systems mix layers and partitions.

ATM Example

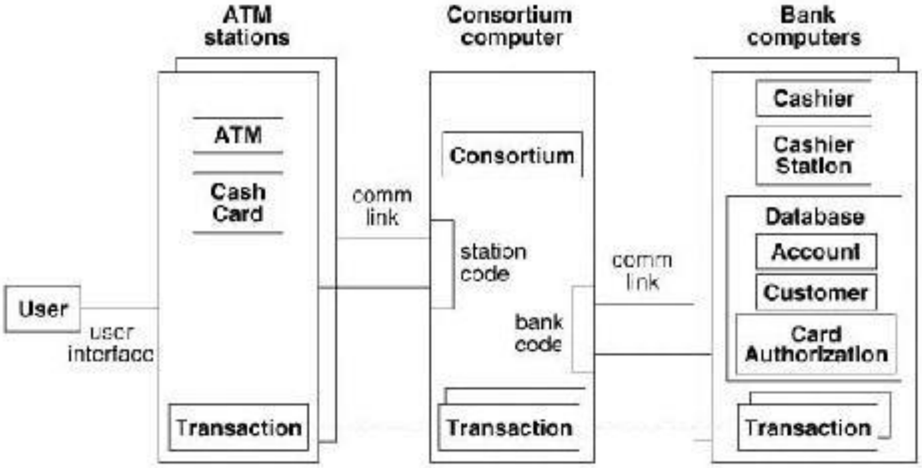


Figure 14.2 Architecture of ATM system. It is often helpful to make an informal diagram showing the organization of a system into subsystems.

Identifying Concurrency

- To identify
  - The objects that must be active concurrently.
  - The objects that have mutually exclusive activity

Inherent Concurrency

- By exam the state model
- Two objects are inherently concurrent if they can receive events at the same time without interacting.
- If the events are unsynchronized, you cannot fold the objects onto a single thread of control.

Defining Concurrent Tasks

- By examining the state diagrams, you can fold many objects onto 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.
- ATM example:
  - Combine the ATM object with the bank transaction object as a single task.

Allocation of Subsystems

- Allocate each concurrent subsystem to a hardware unit by
  - Estimating hardware resource requirements
  - Making hardware-software trade-offs
  - Allocating tasks to processors
  - Determining physical connectivity

Estimating hardware resource requirements

- The number of processors required depends on the volume of computations and the speed of the machine
- Example: military radar system generates too much data in too short a time to handle in single CPU, many parallel machines must digest the data
- Both steady-state load and peak load are important

Making hardware-software trade-offs

- You must decide which subsystems will be implemented in hardware or software
- Main reasons for implementing subsystems in hardware
  - Cost -
  - Performance – most efficient hardware available

Allocating tasks to processors

- Allocating software subsystems to processors
- Several reasons for assigning tasks to processors.
  - Logistics – certain tasks are required at specified physical locations, to control hardware or permit independent operation
  - Communication limits
  - Computation limits – assigning highly interactive systems to the same processor, independent systems to separate processors

Determining physical connectivity

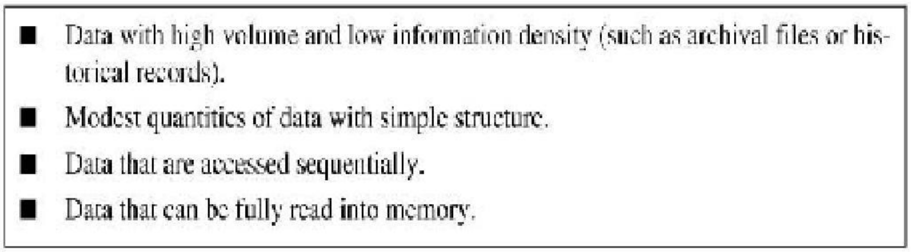
- Determine the arrangement and form of the connections among the physical units
  - Connection topology- choose an topology for connecting the physical units
  - Repeated units-choose a topology of repeated units
  - Communications- choose the form of communication channels and communication protocols

Management of Data Storage

- Alternatives for data storage:
  - Data structures,
  - Files,
  - Databases

Data Suitable for Files

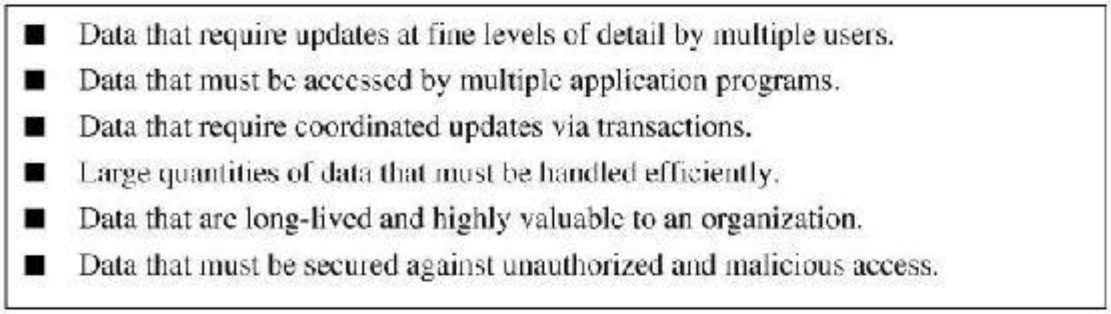
- Files are cheap, simple, and permanent, but operations are low level.



- Data with high volume and low information density (such as archival files or historical records).
- Modest quantities of data with simple structure.
- Data that are accessed sequentially.
- Data that can be fully read into memory.

Data Suitable for Databases

- Database make applications easier to port, but interf. is complex.



- Data that require updates at fine levels of detail by multiple users.
- Data that must be accessed by multiple application programs.
- Data that require coordinated updates via transactions.
- Large quantities of data that must be handled efficiently.
- Data that are long-lived and highly valuable to an organization.
- Data that must be secured against unauthorized and malicious access.

Figure 14.4 Data suitable for databases. Databases provide heavyweight data management and are used for most important business applications.

Handling Global Resources

- The system designer must identify global resources and determine *mechanisms for controlling access* to them.
- Kinds of global resources:
  - Physical units

    - Processors, tape drivers…
  - Sp.s
    - Disk sp.s, workstation screen…
  - Logical name
    - Object ID, filename, class name…
  - Access to shared data
    - Database

- Some common mechanisms are:
  - Establishing "**guardian**" object that serializes all access
  - **Partitioning** global resources into disjoint subsets which are managed at a lower level, and
  - **Locking**

ATM example

- Bank codes and account numbers are global resources.
- Bank codes must unique within the context of a consortium.
- Account codes must be unique within the context of a bank.

Choosing a Software Control Strategy

- To choose a single control style for the whole system.
- Two kinds of control flows:
  - External control
  - Internal control

Software External Control

- Concerns the flow of externally visible events among the objects in the system.
- Three kinds:
  - Procedure-driven sequential
  - Event-driven sequential
  - Concurrent

Procedure-driven Control

- Control resides within the program code
- Procedure request external input and then wait for it
- When input arrives, control resumes with in the procedure that made the call.
- Advantage:
  - Easy to implement with conventional languages
- Disadvantage:
  - The concurrency inherent in objects are to mapped into a sequential flow of control.
- Suitable only if the state model shows a regular alternation of input and output events.
- C++ and Java are procedural languages.

- They fail to support the concurrency inherent in objects.

Event-driven Control

- Control resides within a dispatcher or monitor that the language, subsystem, or operating system provides.
- The dispatcher calls the procedures when the corresponding events occur.

Software Internal Control

- Refer to the flow of control within a process.
- To decompose a process into several tasks for logical clarity of for performance.
- Three kinds:
  - Procedure calls,
  - Quasi-concurrent intertask call,
    - Multiple address sp.s or call stacks exist but only a single thread of control can be active at once.
  - Current intertask calls

Handling Boundary Conditions

- Most of system design is concerned with steady-state behavior, but boundary conditions are also important
- Boundary conditions are
  - Initialization
  - Termination, and
  - Failure
- Initialization
  - The system must initialize constant data, parameters, global variables, …
- Termination
  - Release any external resources that it had reserved.
- Failure
  - Unplanned termination of a system. The good system designer plans for orderly failure

Setting Trade-off Priorities

- The priorities reconcile desirable but incompatible goals.
  - E.g memory vs. cost
- Design trade-offs affect the entire character of a system.
- The success of failure of the final product may depend on how well its goal s are chosen.
- Essential aspect of system architecture is making trade-offs between
  - time and sp.
  - Hardware and software
  - Simplicity and generality, and
  - Efficiency and maintainability
- The system design must state the priorities

Common Architectural Styles

- Several prototypical architectural styles are common in existing system.
- Some kinds of systems:
  - Batch transformation } Functional transformations

- Continuous transformation
- Interactive interf.
- Dynamic simulation } Time-dependent systems
- Real-time system
- Transaction manager -> Database system

Batch transformation

- Perform sequential computation.

- The application receives the inputs, and the goal is to compute an answer.
- Does not interact with the outside world
- E.g.
  - Compiler
  - Payroll processing
  - VLSI automatic layout
  - :
- The most important aspect is to define a clean series of steps
- Sequence of steps for a compiler

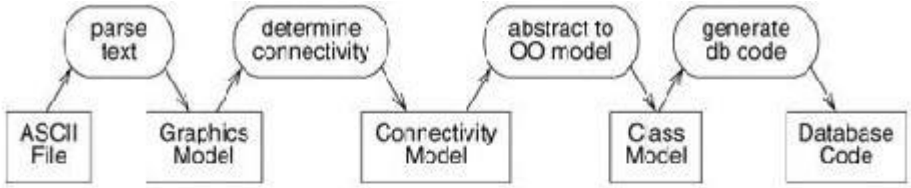


Figure 14.5 Sequence of steps for a compiler. A batch transformation is a sequential input-to-output transformation that does not interact with the outside world

The steps in designing a batch transformation are as follows

- Break the overall transformation into stages, with each stage performing one part of the transformation.
- Prepare class models for the input, output and between each pair of successive stages. Each stage knows only about the models on either side of it.
- Expand each stage in turn until the operations are straightforward to implement.
- Restructure the final pipeline for optimization.

Continuous transformation

- The outputs actively depend on changing inputs.
- Continuously updates the outputs (in practice discretely)
- E.g.
  - Signal processing
  - Windowing systems
  - Incremental compilers
  - Process monitoring system
- Sequence of steps for a graphics application

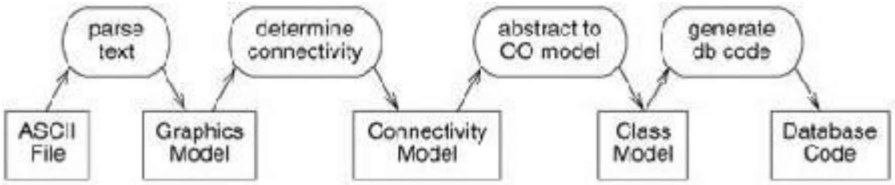


Figure 14.5 Sequence of steps for a compiler. A batch transformation is a sequential input-to-output transformation that does not interact with the outside world.

- Steps in designing a pipeline for a continuous transformation are as follows
  - Break the overall transformation into stages, with each stage performing one part of the transformation.
  - Define input, output and intermediate models between each pair of successive stages as for the batch transformation
  - Differentiate each operation to obtain incremental charges to each stage.
  - Add additional intermediate objects for optimization.

Interactive interf.

– Dominated by interactions between the system and external agents.

Steps in designing an interactive interf. are as follows

- ✓ Isolate interf. classes from the application classes
- ✓ Use predefined classes to interact with external agents
- ✓ Use the state model as the structure of program
- ✓ Isolate physical events from logical events.
- ✓ Fully specify the application functions that are invoked by the interf.

Dynamic simulation

– Models or tracks real-world objects.

– Steps in designing a dynamic simulation

- Identify active real-world objects from the class model.
- Identify discrete events
- Identify continuous dependencies
- Generally simulation is driven by a timing loop at a fine time scale

Real-time system

– An interactive system with tight time constraints on actions.

Transaction manager

– Main function is to store and retrieve data.

– Steps in designing an information system are as follows

- Map the class model to database structures.
- Determine the units of concurrency
- Determine the unit of transaction
- Design concurrency control for transactions

Architecture of the ATM system

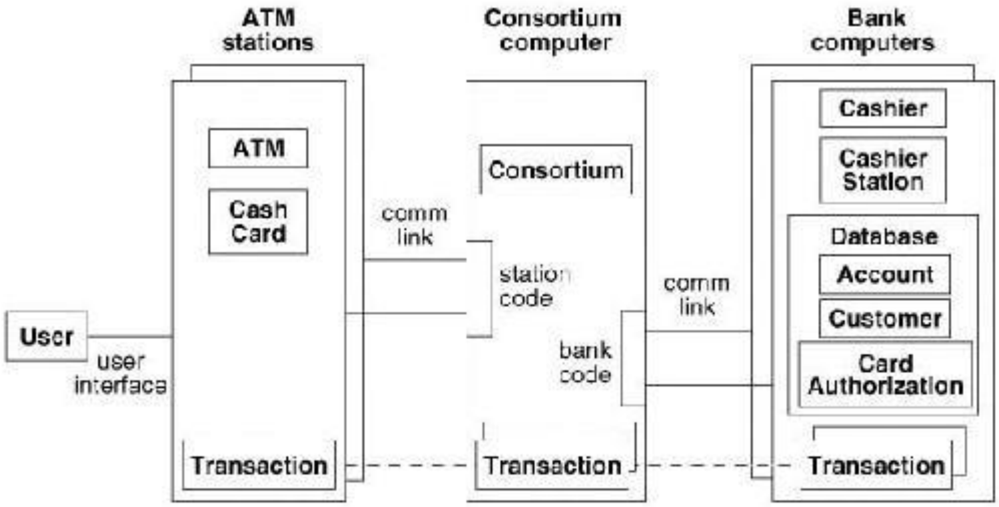


Figure 14.2 Architecture of ATM system. It is often helpful to make an informal diagram showing the organization of a system into subsystems.

Unit-6: Class Design, Implementation modeling

7 Hours

Syllabus:

- *Class Design: Overview of class design;*
- *Bridging the gap; Realizing use cases; Designing algorithms; Recursing downwards, Refactoring;*
- *Design optimization; Reification of behavior; Adjustment of inheritance; Organizing a class design;*
- *ATM example.*
- *Implementation Modeling: Overview of implementation; Fine-tuning classes; Fine-tuning generalizations; realizing associations; Testing.*
- *Legacy Systems: Reverse engineering;*
- *Building the class models; Building the interaction model;*
- *Building the state model; Reverse engineering tips; Wrapping; Maintenance.*

Class design

- The analysis phase determines *what* the implementation must do
- The system design phase *determines the plan of attack*
- The purpose of the class design is to complete the *definitions of the classes* and *associations* and choose *algorithms* for operations

Overview of Class Design – steps

1. Bridging the gap
2. Realizing Use Cases
3. Designing Algorithms
4. Recursing Downward
5. Refactoring
6. Design Optimization
7. Reification of Behavior
8. Adjustment of Inheritance
9. Organizing a Class Design

1. **<u>Bridging the gap</u>**

   Bridge the gap from high-level requirements to low-level services

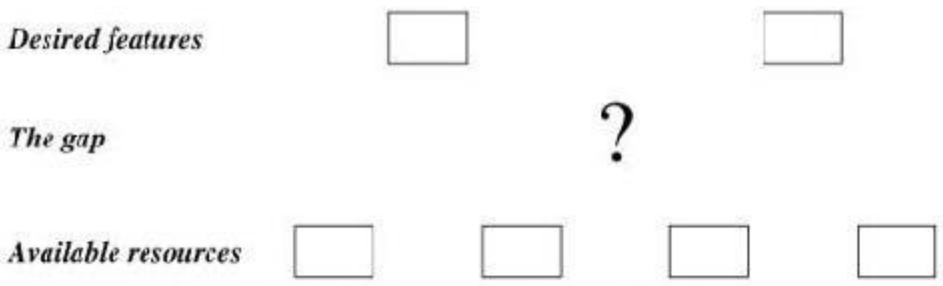


Figure 15.1 The design gap. There is often a disconnect between the desired features and the available resources.

- Salesman can use a spreadsheet to construct formula for his commission – readily build the system
- Web-based ordering system – cannot readily build the system because too big gap between the resources and features

- The intermediate elements may be operations, classes or other UML constructs.
- You must invent intermediate elements to bridge the gap.

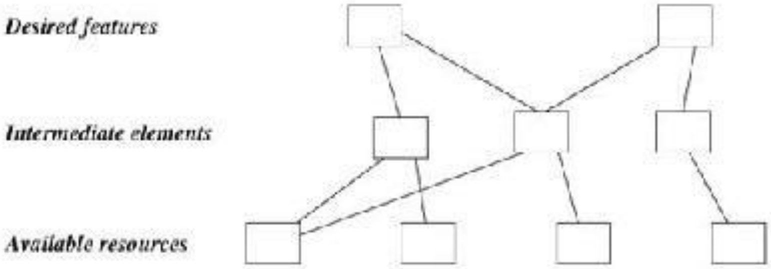


Figure 15.2 Bridging the gap. You must invent intermediate elements to bridge the gap between the desired features and the available resources.

2. **Realizing Use Cases**

- Realize use cases with operations.
- The cases define system-level behavior.
- During design you must invent new operations and new objects that provide this behavior.

- Step1: List the **responsibilities** of a use case or operation.

- A ***responsibility*** is something that an object knows or something it must do.
- For Example:
    - An **online theater ticket system**
    - Making a reservation has the **responsibility** of
        - Finding unoccupied seats to the desired show,
        - Marking the seats as occupied,
        - Obtaining payment from the customer,
        - Arranging delivery of the tickets, and
        - Crediting payment to the proper account.

- Step2: Each operation will have various responsibilities.
    - Group the responsibilities into **clusters** and try to make each cluster coherent.
- Step3: Define an operation for each responsibility cluster.
- Step4: Assign the new lower-level operations to classes.

ATM Example

- *Process transaction* includes:
    - Withdrawal includes responsibilities:
        - Get amount from customer, verify that amount is covered by the account balance, verify that amount is within the bank’s policies, verify that ATM has sufficient cash, ….
        - A database transaction ensures all-or-nothing behavior.
    - Deposit
    - Transfer

Use Case for the ATM

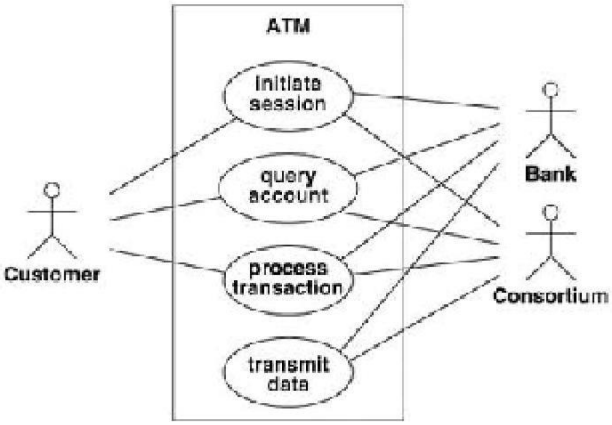


Figure 15.1 Use case diagram for the ATM. Use cases partition the

- Process transaction includes:
    - Deposit includes responsibilities:
        - Get amount from customer, accept funds envelope from customer, …
    - Transfer includes responsibilities:
        - Get source account, get target account, get amount, verify that source account covers amount, …
- There is some overlap between the operations.
- A reasonable design would coalesce this behavior and build it once.

3. **<u>Designing Algorithms</u>**

- Formulate an ***algorithm*** for each operation
- The analysis specification tells *what* the operation does for its clients
- The algorithm show *how* it is done

Designing Algorithms- steps

i. Choose *algorithms* that minimize the cost of implementing operations.
ii. Select *data structures* appropriate to the algorithms
iii. Define new internal classes and operations as necessary.
iv. Assign operations to appropriate classes.

i. Choosing algorithms (Choose algorithms that minimize the cost of implementing operations)

- ➢ When efficiency is not an issue, you should use simple algorithms.
- ➢ Typically, 20% of the operations consume 80% of execution time.
- ➢ Considerations for choosing alternative algorithms
    - o Computational complexity
    - o Ease of implementation and
understandability o Flexibility

        - Simple but inefficient
        - Complex efficient
- ATM Example
    - Interactions between the consortium computer and bank computers could be complex.
    - Considerations:
        - Distributed computing
        - The scale of consortium computer (scalability)
        - The inevitable conversions and compromises in coordinating the various data formats.
    - All these issues make the choice of algorithms for coordinating the consortium and the banks important

The ATM Case Study

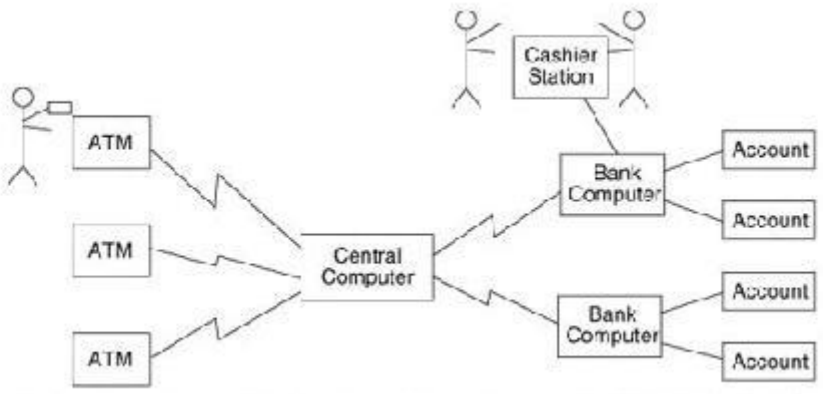


Figure 11.3 ATM network. The ATM case study threads throughout the remainder of this book.

ii. **Choosing Data Structures (select data structures appropriate to the algorithm)**

  a. Algorithms require data structures on which to work.
  b. They organize information in a form convenient for algorithms.
  c. Many of these data structures are instances of ***container classes***.
  d. Such as arrays, lists, queues, stacks, set…etc.

iii. **Defining New Internal Classes and Operations**

  a. To invent new, low-level operations during the decomposition of high-level operations.
  b. The expansion of algorithms may lead you to create new classes of objects to hold intermediate results.
  c. ATM Example:
     i. *Process transaction* uses case involves a customer receipt.
     ii. A *Receipt* class is added.

iv. **Assigning Operations to Classes (assign operations to appropriate classes)**

  a. How do you decide what class owns an operation?
     i. Receiver of action

        1. To associate the operation with the *target* of operation, rather than the *initiator*.

    b. Query vs. update
        i. The object that is changed is the target of the operation
    c. Focal class
        i. Class centrally located in a star is the operation's target
    d. Analogy to real world

[redacted]

ATM Example

- *Process transaction* includes:
    - Withdrawal includes responsibilities:
        - Get amount from customer, verify that amount is covered by the account balance, verify that amount is within the bank's policies, verify that ATM has sufficient cash, ….
        - A database transaction ensures all-or-nothing behavior.
    - Deposit
    - Transfer
- Customer.getAccount(), account.verifyAmount(amount), bank.verifyAmount(amount), ATM.verifyAmount(amount)

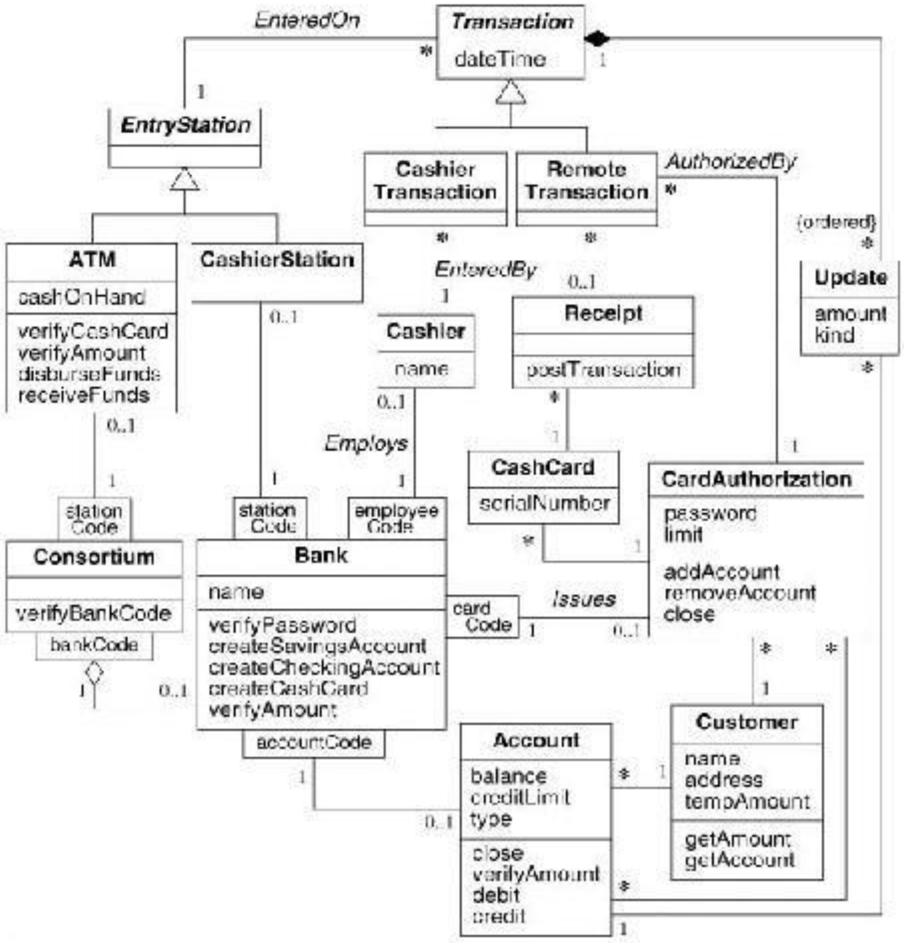


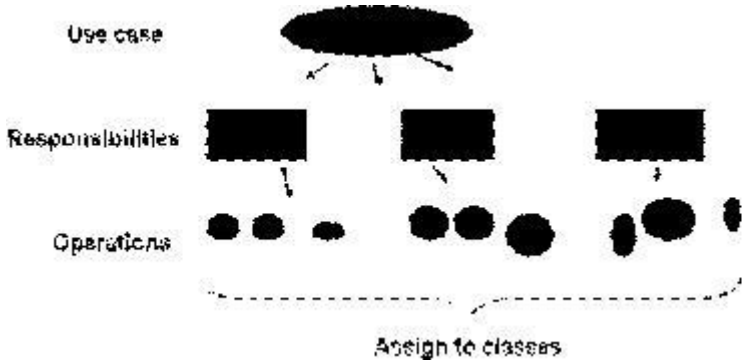
Figure 15.4 ATM domain class model with some class design elaborations.

4. Recursing Downward

- To organize operations as layers.
  - Operations in higher layers invoke operations in lower layers.
- Two ways of downward recursion:
  - By functionality
  - By mechanism
- Any large system mixes functionality layers and mechanism layers.

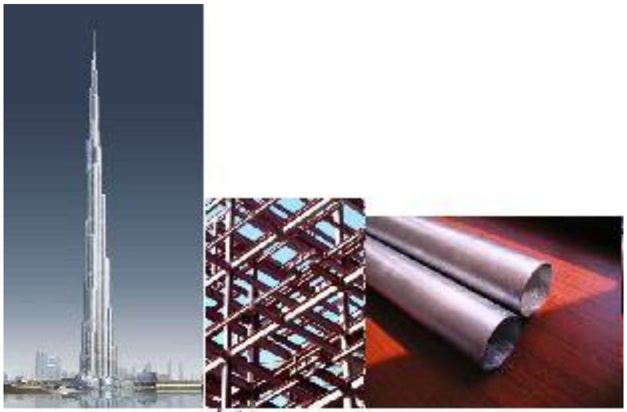
Functionality Layers

- Take the required high-level functionality and break it into lesser operations.
- Make sure you combine similar operations and attach the operations to classes.
- An operation should be coherent meaningful, and not an arbitrary portion of code.
- ATM eg., **use case** decomposed into **responsibilities** (see sec 15.3). Resulting **operations** are assigned to classes (see sec 15.4.4). If it is not satisfied rework them



Mechanism Layers

- Build the system out of layers of needed support mechanisms.
- These mechanisms don't show up explicitly in the high-level responsibilities of a system, but they are needed to make it all work.
- E.g. Computing architecture includes
- Data structures, algorithms, and control patterns.
- A piece of software is built in terms of other, more mechanisms than itself.



Ethernet
CP-134U-I V2
Air Conditioner Control
Windows NT Server
Elevator Control
Entrance Control
Fire Alarm System

5. Refactoring

- Refactoring
  - Changes to the internal structure of software to improve its design without altering its external functionality.
- You must revisit your design and rework the classes and operations so that they clean satisfy all their uses and are conceptually sound.

ATM Example

- Operations of process transaction
  - *Account.credit(amount)*
  - *Account.debit(amount)*
- Combine into
  - *Account.post(amount)*

6. Design Optimization

- To design a system is to first get the logic correct and then optimize it.
- Often a small part of the code is responsible for most of the time or sp. costs.
- It is better to focus optimization on the **critical areas,** than to spread effort evenly.

Design Optimization

- Optimized system is **more obscure** and less **likely to be reusable**.
- You must strike an appropriate **balance between efficiency and clarity.**
- **Tasks** to optimization:
  1. Provide efficient access paths.
  2. Rearrange the computation for greater efficiency.
  3. Save intermediate results to avoid recomputation.

i. Adding Redundant Associations for Efficient Access

- ✓ Rearrange the associations to optimize critical aspects of the system.
- ✓ Consider employee skills database

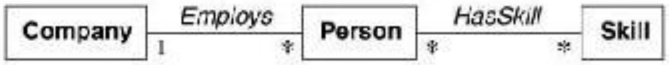


Figure 15.5 Analysis model for person skills. Derived data is undesirable during analysis because it does not add information.

- ✓ *Company.findSkill*( ) returns a set of persons in the company with a given skill.
- ✓ Suppose the company has 1000 employees,.
- ✓ In case where the number of hits from a query is low because few objects satisfy the test, an *index* can improve access to frequently retrieved objects.

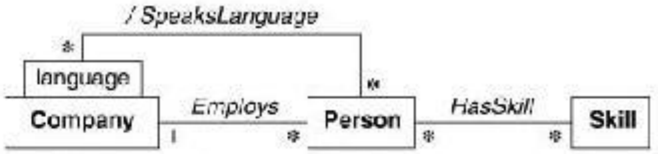


Figure 15.6 Design model for person skills. Derived data is acceptable during design for operations that are significant performance bottlenecks.

- ✓ Examine each operations and see what associations it must traverse to obtain its information.
- ✓ Next, for each operation, note the following,
  - Frequency of access
  - Fan-out
  - Selectivity

ATM Example

- Banks must report cash deposits and withdrawals greater than $10,000 to the government.

- Tr. from
  - *Bank* to *Account*,
  - *Account* to *Update*,
  - Then filter out the updates that are cash and greater than $10,000
- A derived association from *Bank* to *Update* would speed this operation.

ii. Rearranging Execution Order for Efficiency

- ✓ After adjusting the structure of class model to optimize frequent traversals, the next thing is
- ✓ To optimize the algorithm
  1. To eliminate dead paths as early as possible
  2. To narrow the search as soon as possible
  3. Sometimes, invert the execution order of a loop

iii. Saving Derived Values to Avoid Recomputation

- ✓ There are three ways to handle updates
  1. Explicit update
  2. Periodic recomputation
  3. Active values

Reification behavior

- Behavior written in code is rigid; you can execute but cannot manipulate it at run time
- If you need to store, pass, or modify the behavior at run time, you should reify it

Adjustment of Inheritance

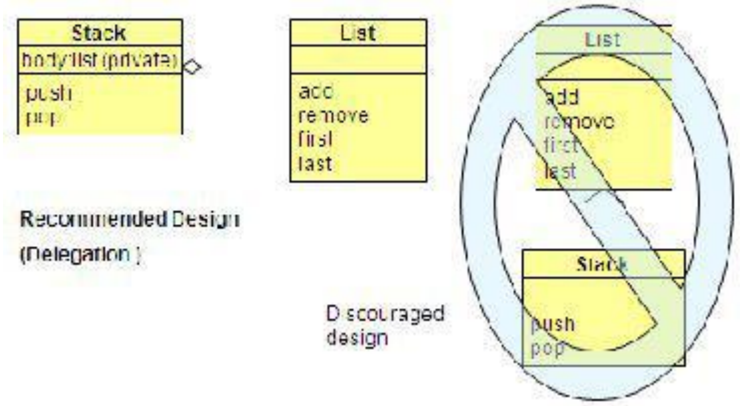
- To increase inheritance perform the following steps
  - Rearrange classes and operations to increase inheritance
  - Abstract common behavior out of groups of clusters
  - Use delegation to share behavior when inheritance is semantically invalid

Rearrange classes and operations to increase inheritance

- Use the following kinds of adjustments to increase the chance of inheritance
  - Operations with optional arguments
  - Operations that are special cases
  - Inconsistent names
  - Irrelevant operations

Use delegation to share behavior when inheritance is semantically invalid

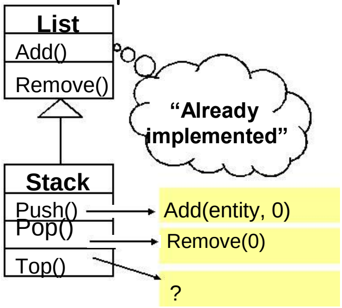
- When class B inherits the specification of class A, you can assume that every instance of class B is an instance of class A because it behaves the same
- ***Inheritance of implementation*** – discourage this
- One object can selectively invoke the desired operations of another class, using delegation rather than inheritance
- Delegation consists of catching operation on one object and sending it to a related object
- Delegate only meaningful operations, so there is no danger of inheriting meaningless operations by accident



Implementation Inheritance

- A very similar class is already implemented that does almost the same as the desired class implementation.

❖ Example: I have a **List** class, I need a **Stack** class. How about subclassing the **Stack** class from the **List** class and providing three methods, **Push()** and **Pop(), Top()**?



❖ Problem with implementation inheritance:

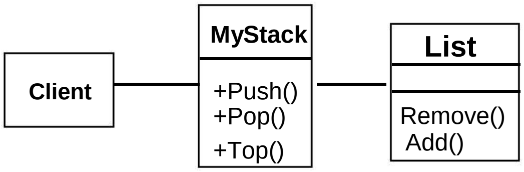
- Some of the inherited operations might exhibit unwanted behavior. What happens if the Stack user calls Remove() instead of Pop()?
- Close coupling – what happens if the Add() method is changed?

Problem with implementation inheritance

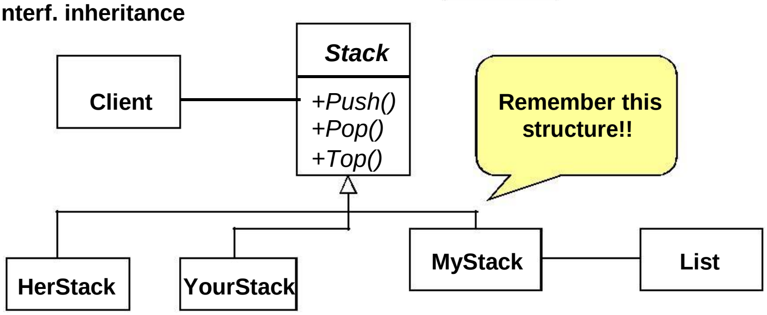
- How to avoid the following problem?

Some of the inherited operations might exhibit unwanted behavior.
What happens if the Stack user calls Remove() instead of Pop()?

1. **Delegation**



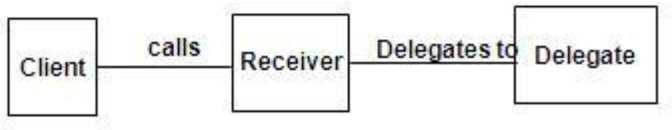
2. **Interf. inheritance**



Delegation as alternative to Implementation Inheritance

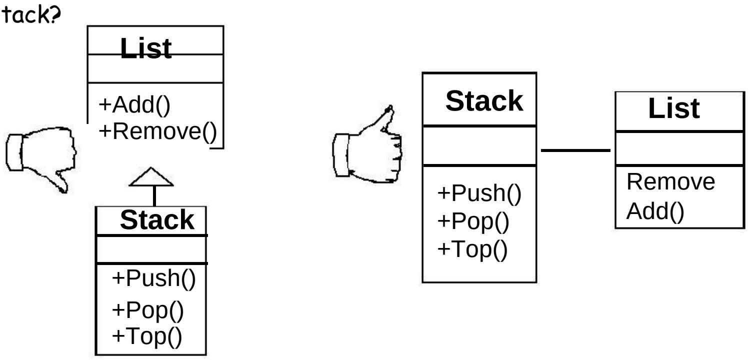
- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance
- In Delegation two objects are involved in handling a request
  - A receiving object delegates operations to its delegate.

The developer can make sure that the receiving object does not allow the client to misuse the delegate object



Delegation instead of Implementation Inheritance

- **Inheritance**: Extending a Base class by a new operation or overwriting an operation.
- **Delegation**: Catching an operation and sending it to another object.
- Which of the following models is better for implementing a stack?



Organization of Class Design

- We can improve the organization of a class design with the following steps:
  - Information hiding
  - Coherence of Entities
  - Fine-tuning packages

Information hiding

- Carefully separating external specification from internal specification
- There are several ways to hide information:
  - Limit the scope of class-model traversals
  - Do not directly access foreign attributes
  - Define interf.s at a high level of abstraction
  - Hide external objects
  - Avoiding cascading method calls

Coherence of Entities

➔ An entity, such as a class, an operation or a package is coherent if it is organized on a consistent plan and all its parts fit together toward a common goal.

➔ An entity should habve a single major theme

➔ It should not be a collection of unrelated parts.

Fine – Tuning Packages

- **Overview of Implementation**
- **Fine-tuning Classes**
- **Fine-tuning Generalization**
- **Realizing Associations**
- **Testing**

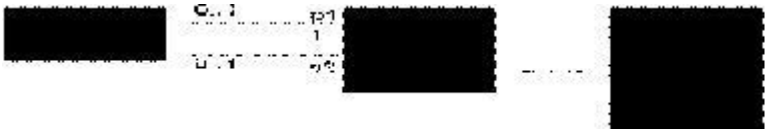
Fine-tuning classes

- Fine tune classes before writing code in order to simplify development or to improve performance
- Partition a class
- Merge classes
- Partition / merge attributes
- Promote an attribute / demote a class

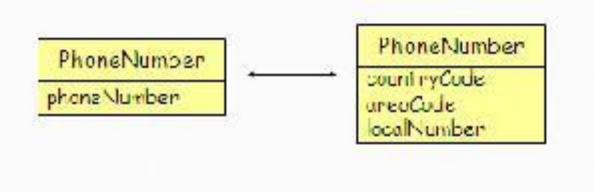
Fine-tuning classes – partition a class

- Sometimes it is helpful to fine-tune a model by partitioning or merging classes
- partitioning of a class can be complicated by generalization and association

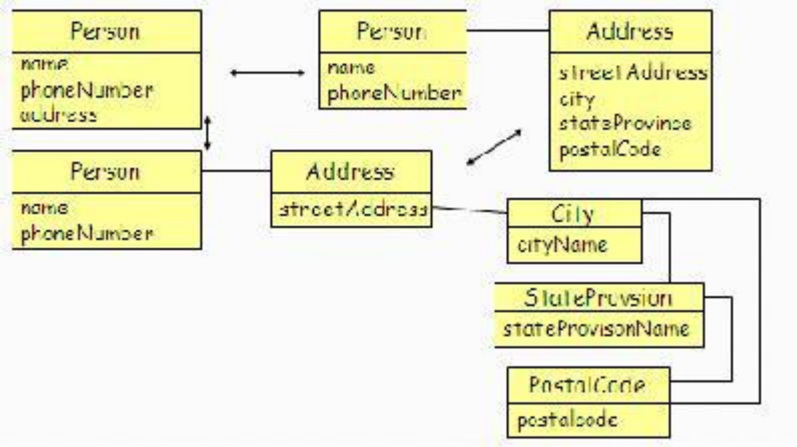
Fine-tuning classes – merge classes



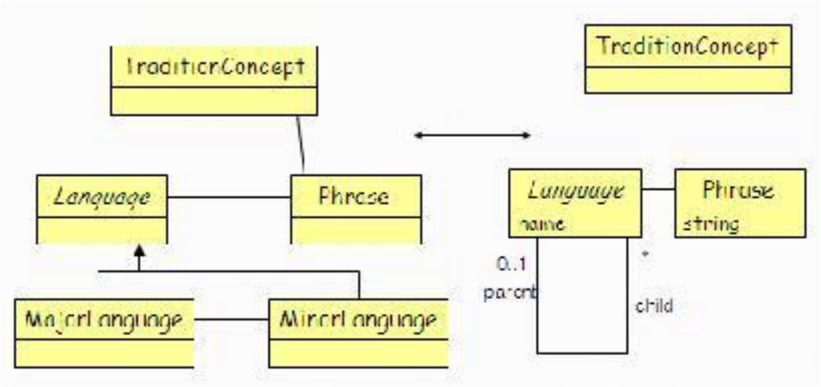
Fine-tuning classes – partition / merge attributes



Fine-tuning classes – promoting an attribute / demote a class



Fine-tuning generalizations



Realizing associations

- Associations are “glue” of the class model, providing access paths between objects
- Analyzing associations by traversing associations

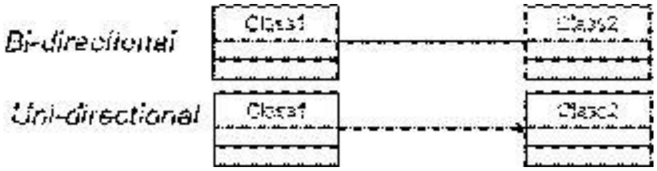
Class1
Class2

Analyzing Association Traversal

- Until now we assumed that associations are bidirectional
- But some applications are traversed in only one direction
- We may add another operation that make traversal in reverse direction

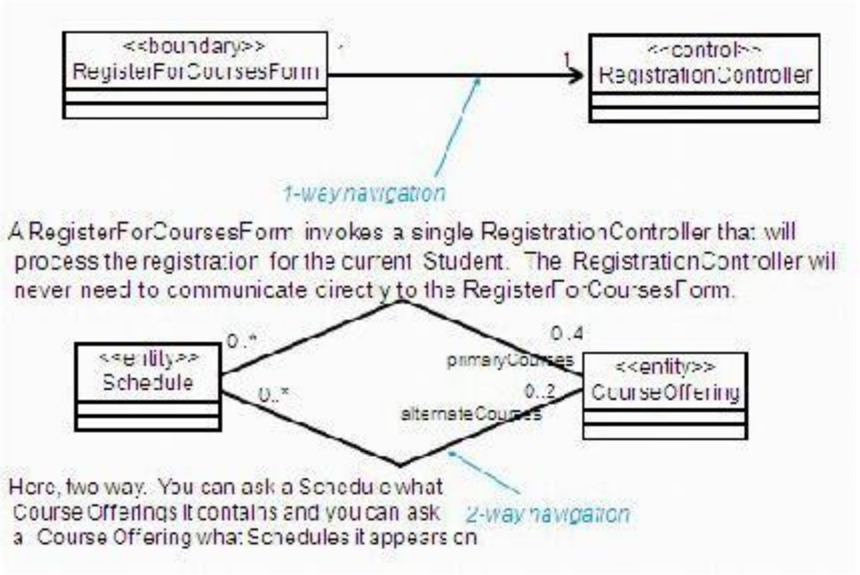
Navigability

- Possible to navigate from an associating class to the target class – indicated by arrow which is pl.d on the target end of the association line next to the target class (the one being navigated to).
- Associations are bi-directional by default – suppress arrows.
- Arrows only drawn for associations with one-way navigability.



- ✓ Navigability is inherently a design and implementation property.
- ✓ Can be specified in Analysis, but with expectation of refining in Class Design.
- ✓ In analysis, associations are usually bi-directional; design, we really check this.

Example: Navigability

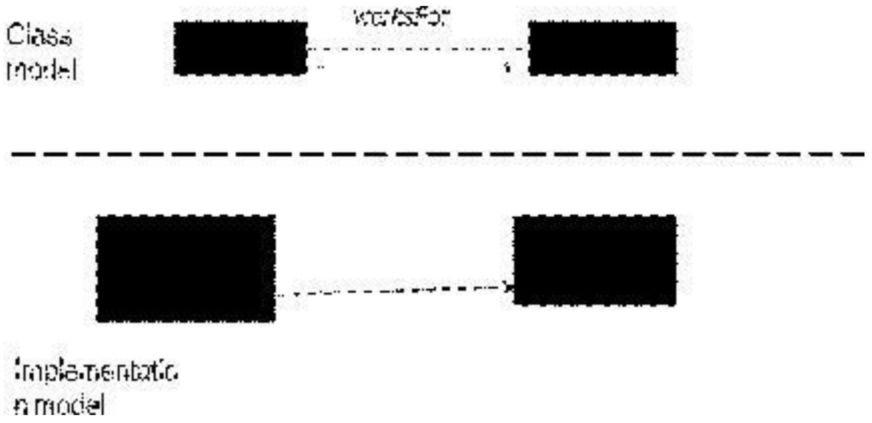


A RegisterForCoursesForm invokes a single RegistrationController that will process the registration for the current Student. The RegistrationController will never need to communicate directly to the RegisterForCoursesForm.

Here, two way. You can ask a Schedule what Course Offerings it contains and you can ask a Course Offering what Schedules it appears on

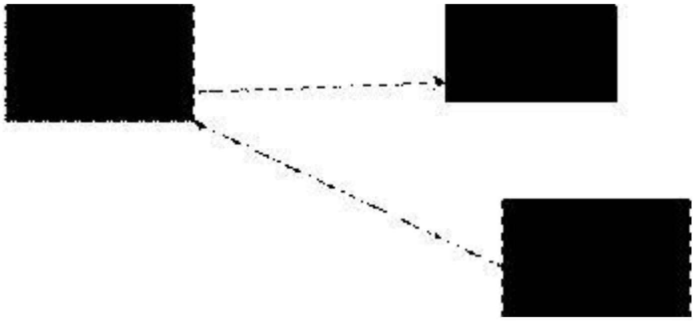
One-way Associations

- Implement one-way associations using ***pointer***- an attribute that contains the object reference
- Actual implementation of pointer using
  - Programming language pointer or
  - Database foreign key
- If the multiplicity is "one" then it is a *simple pointer*
- If the multiplicity is "many" then it is a *set of pointers*



Two-way Association

- Many associations are traversed in both directions, not usually with equal frequencies
- Three approaches for implementation
  - Implement one-way
  - Implement two-way
  - Implement with an association object



Testing

- Unit testing
- System testing

UNIT - 7 DESIGN PATTERNS – 1:

Syllabus : - 6hrs

- **What is a pattern**
- **what makes a pattern?**
- **Pattern categories;**
- **Relationships between patterns;**
- **Pattern description.**
- **Communication Patterns:**
- **Forwarder-Receiver;**
- **Client-Dispatcher-Server;**
- **Publisher-Subscriber.**

Patterns

❖ Patterns help you build on the collective experience of skilled software engineers.

❖ They capture existing, well-proven experience in software development and help to promote good design practice.

❖ Every pattern deals with a specific, recurring problem in the design or implementation of a software system.

❖ Patterns can be used to construct software architectures with specific properties

What is a Pattern?

▪ Abstracting from specific problem-solution pairs and distilling out common factors leads to patterns.

▪ These problem-solution pairs tend to fall into families of similar problems and solutions with each family exhibiting a pattern in both the problems and the solutions.

Definition :

The architect Christopher Alexander defines the term pattern as

❖ Each pattern is a three-part rule, which expresses a relation between a
certain context,
a problem, and
a solution.

- As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.

- As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.

- The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing. And when we must create it. It is both a process and a thing: both a description of a thing which is alive, and a description of the process which will generate that thing.

Properties of patterns for Software Architecture

- ❖ A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it.
- ❖ Patterns document existing, well-proven design experience.
- ❖ Patterns identify & and specify abstractions that are above the level of single classes and instances, or of components.
- ❖ Patterns provide a common vocabulary and understanding for design principles
- ❖ Patterns are a means of documenting software architectures.
- ❖ Patterns support ***the*** construction of software with defined properties.
- ❖ Patterns help you build complex and heterogeneous software

architectures ❖ Patterns help you to manage software complexity

Putting all together we can define the pattern as:

Conclusion or final definition of a Pattern:

A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.

What Makes a Pattern?

Three-part schema that underlies every pattern:

Context: a situation giving rise to a problem.

Problem: the recurring problem arising in that context.

Solution: a proven resolution of the problem.

Context:

- The Contest extends the plain problem-solution dichotomy by describing the situations in which the problems occur
- Context of the problem may be fairly general. For eg: "developing software with a human-computer interf.". On the other had, the contest can tie specific patters together.
- Specifying the correct context for the problem is difficult. It is practically impossible to determine all situations in which a pattern may be applied.

Problem:

- This part of the pattern description schema describes the problem that arises repeatedly in the given context.
- It begins with a general problem specification (capturing its very essence what is the concrete design issue we must solve?)
- This general problem statement is completed by a set of forces
- Note: The term 'force denotes any aspect of the problem that should be considered while solving it, such as
  - Requirements the solution must fulfill
  - Constraints you must consider
  - Desirable properties the solution should have.
- Forces are the key to solving the problem. Better they are balanced, better the solution to the problem

Solution:

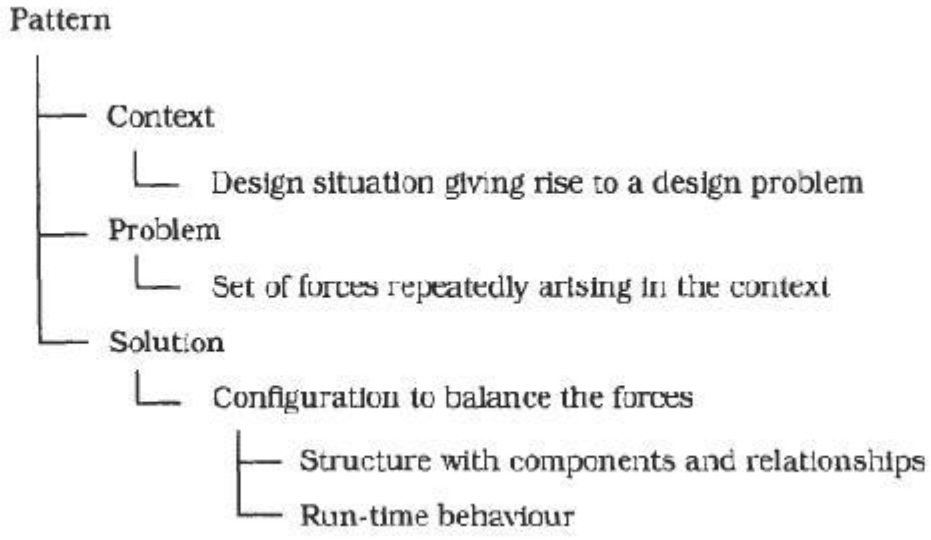
- The solution part of the pattern shows how to solve the recurring problem(or how to balance the forces associated with it)
- In software architectures, such a solution includes two aspects:

Every pattern specifies a certain structure, a spatial configuration of elements. This structure addresses the static aspects of the solution. It consists of both components and their relationships.

Every pattern specifies runtime behavior. This runtime behavior addresses the dynamic aspects of the solution like, how do the participants of the patter collaborate? How work is organized between then? Etc.

- The solution does not necessarily resolve all forces associated with the Problem.
- A pattern provides a solution schema rather than a full specified artifact or blue print.
- No two implementations of a given pattern are likely to be the same.

- The following diagram summarizes the whole schema.


Pattern
Context
Design situation giving rise to a design problem
Problem
Set of forces repeatedly arising in the context
Solution
Configuration to balance the forces
Structure with components and relationships
Run-time behaviour

Pattern Categories
we group patterns into three categories:

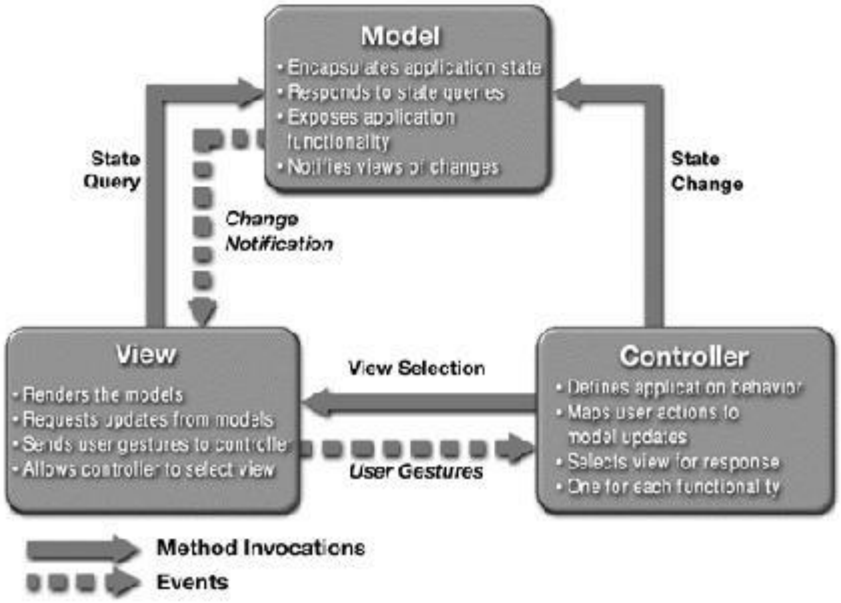
- Architectural patterns
- Design patterns
- Idioms

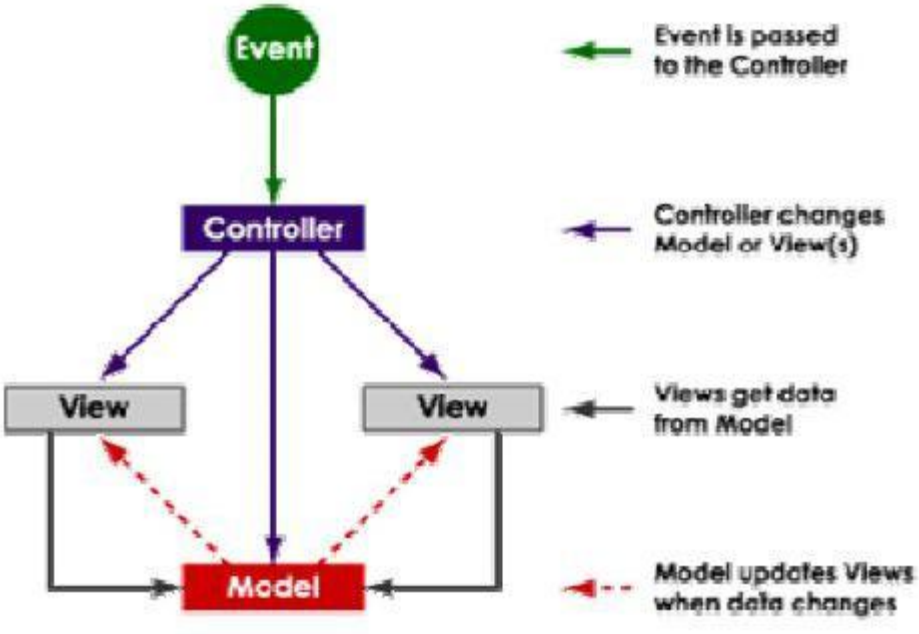
Each category consists of patterns having a similar range of scale or abstraction.

Architectural patterns

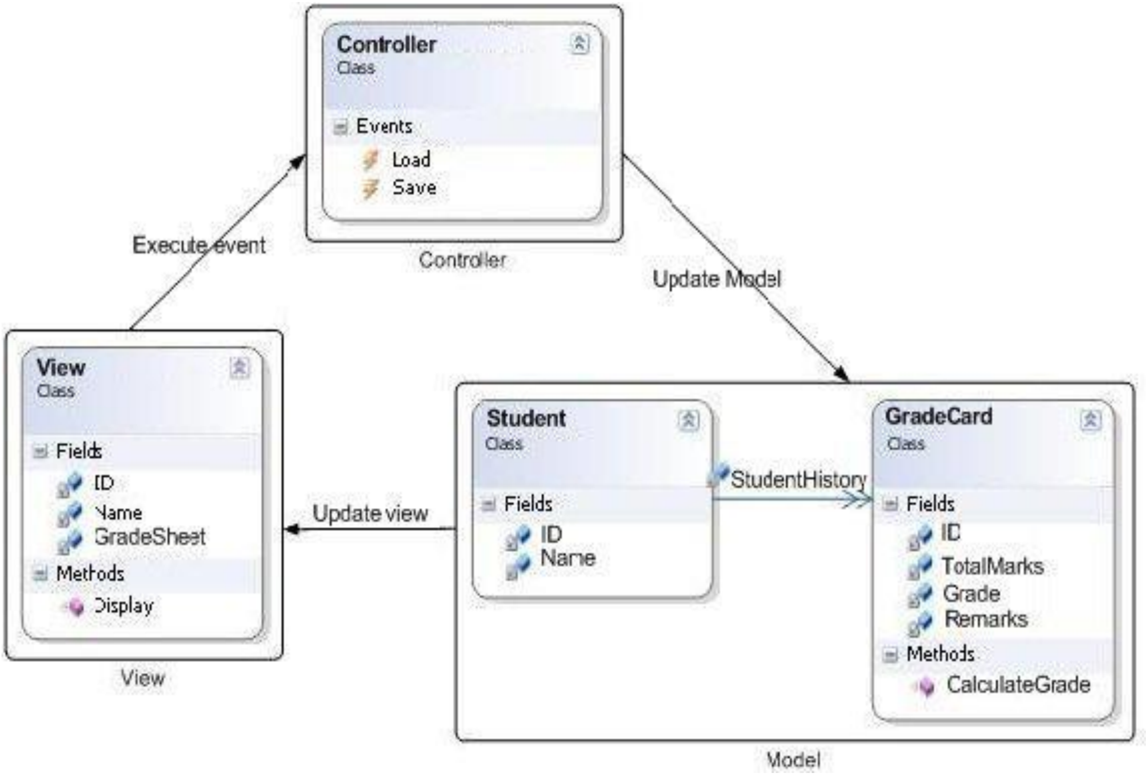
- Architectural patterns are used to describe viable software architectures that are built according to some overall structuring principle.
- Definition: An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- Eg: Model-view-controller pattern.

Structure→


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events


Event
Controller
View
View
Model
Event is passed to the Controller
Controller changes Model or View(s)
Views get data from Model
Model updates Views when data changes

Eg:



Design patterns

- Design patterns are used to describe subsystems of a software architecture as well as the relationships between them (which usually consists of several smaller architectural units)
- Definition: **A** design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them.It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular Context.
- They are medium-scale patterns. They are smaller in scale than architectural patterns, but tend to be independent of a particular programming language or programming paradigm.
- Eg: Publisher-Subscriber pattern.

Idioms

- Idioms deals with the implementation of particular design issues.
- Definition: ***An idiom*** is a low-level pattern specific to a programming language. ***An*** idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.
- Idioms represent the lowest- level patterns. They address aspects of both design and implementation.
- Eg: counted body pattern.

Pattern description

- **Name :**The name and a short summary of the pattern
- **Also known as:**Other names for the pattern, if any are known
- **Example :**A real world example demonstrating the existence of the problem and the need for the pattern
- **Context :**The situations in which the patterns may apply
- **Problem :**The problem the pattern addresses, including a discussion of its associated forces.
- **Solution :**The fundamental solution principle underlying the pattern
- **Structure :**A detailed specification of the structural aspects of the pattern, including CRC – cards for each participating component and an OMT class diagram.
- **Dynamics :**Typical scenarios describing the run time behavior of the pattern
- **Implementation:** Guidelines for implementing the pattern. These are only a suggestion and not a immutable rule.
- **Examples resolved**: Discussion for any important aspects for resolving the example that are not yet covered in the solution , structure, dynamics and implementation sections.
- **Variants:**A brief description of variants or specialization of a pattern
- **Known uses:**Examples of the use of the pattern, taken from existing systems
- **Consequences:**The benefits the pattern provides, and any potential liabilities.
- **See Also:**References to patterns that solve similar problems, and the patterns that help us refine the pattern we are describing.

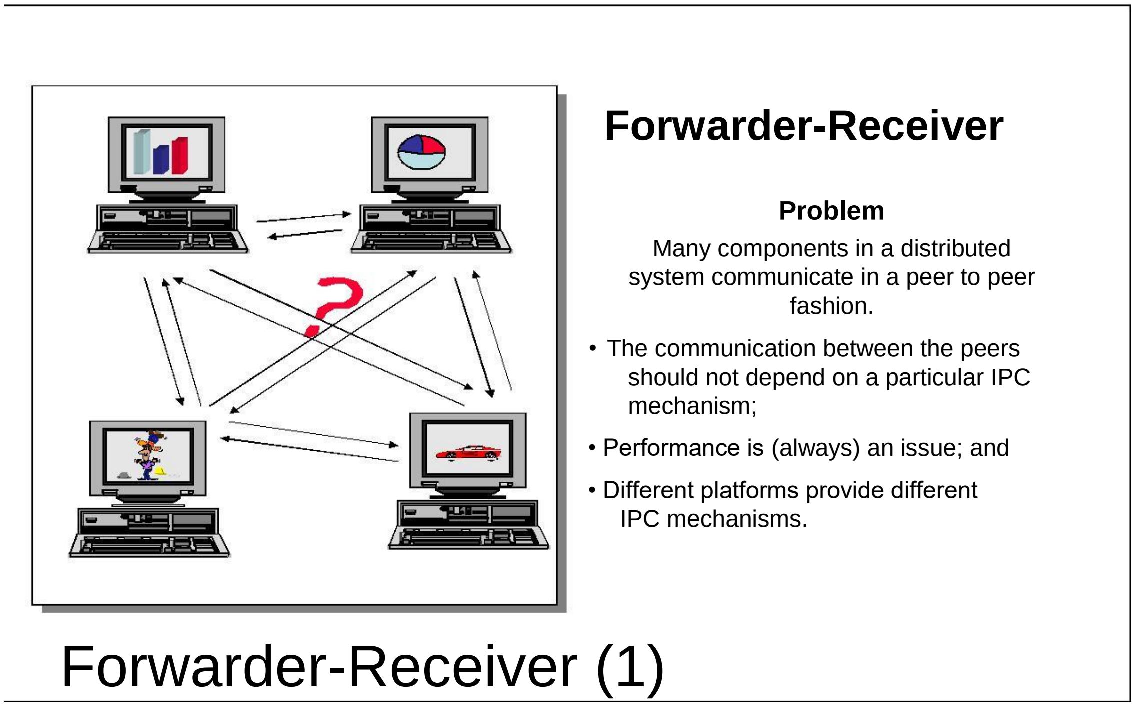
Communication pattern:

Most of the today’s software systems run on distributed systems. These distributed systems need a means for communication.

- Problems:
  - Many communication mechanisms to choose from.
  - The use of communication facilities is often hard-wired into existing applications, leading to various problems.
    - Difficult to change the communication mechanism later.
    - Portability

- Migration of sub systems from one network node to another is only possible if the communication facility allows it.

- Solution:
  - Loosen the coupling between components of a distributed system and the mechanism it uses for communication, eg: by using
    - Encapsulation
    - Location transparency
- We discuss two patterns that addresses these topics:
  - The **Forwarder – Receiver** design pattern (provides encapsulation)
  - The **Client – Dispatcher – Server** design pattern (provides location transparency)
- Keeping cooperating component consistent is another problem in communication. We discuss one pattern that addresses this issue:
  - The **Publisher – Subscriber pattern**



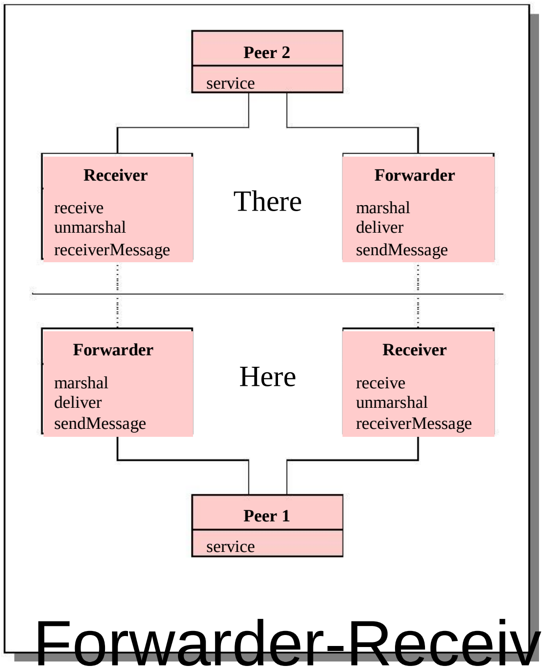
Forwarder-Receiver

Problem

Many components in a distributed system communicate in a peer to peer fashion.

- The communication between the peers should not depend on a particular IPC mechanism;
- Performance is (always) an issue; and
- Different platforms provide different IPC mechanisms.

Forwarder-Receiver (1)



Solution

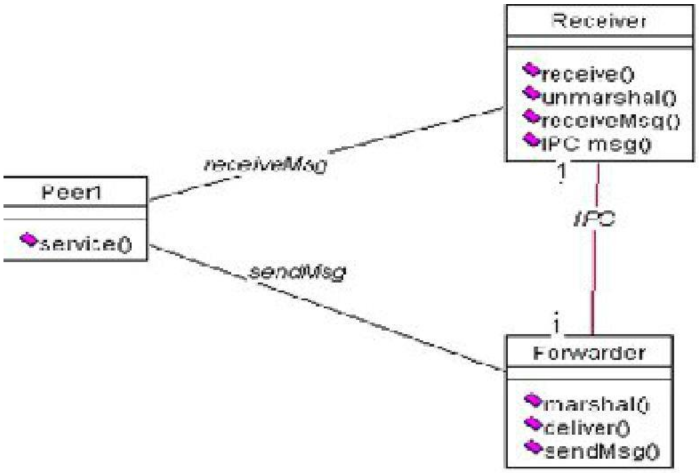
Encapsulate the inter-process communication mechanism:

- *Peers* implement application services.
- *Forwarders* are responsible for sending requests or messages to remote peers using a specific IPC mechanism.
- *Receivers* are responsible for receiving IPC requests or messages sent by remote peers using a specific IPC mechanism and dispatching the appropriate method of their intended receiver.

Forwarder-Receiver (2)

- **Intent**
- "The Forwarder-Receiver design pattern provides transparent interprocess communication for software systems with a peer-to-peer interaction model.
- It introduces forwarders and receivers to decouple peers from the underlying communication mechanisms."
- **Motivation**
- Distributed peers collaborate to solve a particular problem.
- A peer may act as a client - requesting services- as a server, providing services, or both.
- The details of the underlying IPC mechanism for sending or receiving messages are hidden from the peers by encapsulating all system-specific functionality into separate components. Examples of such functionality are the mapping of names to physical locations, the establishment of communication channels, or the marshaling and unmarshaling of messages.

Structure



- F-R consists of three kinds of components, Forwarders, receivers and peers.
- Peer components are responsible for application tasks.
- Peers may be located in different process, or even on a different machine.
- It uses a forwarder to send messages to other peers and a receiver to receive messages form other peers.
- They continuously monitor network events and resources, and listen for incoming messages form remote agents.
- Each agent may connect to any other agent to exchange information and requests.
- To send a message to remote peer, it invokes the method sendmsg of its forwarder.
- It uses marshal.sendmsg to convert messages that IPC understands.
- To receive it invokes receivemsg method of its receiver to unmarshal it uses unmarshal.receivemsg.
- Forwarder components send messages across peers.
- When a forwarder sends a message to a remote peer, it determines the physical location of the recipient by using its name-to-address mapping.
- Kinds of messages are
- Command message- instruct the recipient to perform some activities.
- Information message- contain data.
- Response message- allow agents to acknowledge the arrival of a message.

- It includes functionality for sending and marshaling
- Receiver components are responsible for receiving messages.
- It includes functionality for receiving and unmarshaling

messages. Dynamics

- P1 requests a service from a remote peer P2.
- It sends the request to its forwarder forw1 and specifies the name of the recipient.
- Forw1 determines the physical location of the remote peer and marshals the message.
- Forw1 delivers the message to the remote receiver recv2.
- At some earlier time p2 has requested its receiver recv2 to wait for an incoming request.
- Now recv2 receives the message arriving from forw1.
- Recv2 unmarshals the message and forwards it to its peer p2.
- Meanwhile p1 calls its receiver recv1 to wait for a response.
- P2 performs the requested service and sends the result and the name of the recipient p1 to the forwarder forw2.
- The forwarder marshals the result and delivers it recv1.
- Recv1 receives the response from p2, unmarshals it and delivers it to p1.

Implmentation

- Specify a name to address mapping.-/server/cvramanserver/…..
- Specify the message protocols to be used between peers and forwarders.-class message consists of sender and data.
- Choose a communication mechanism-TCP/IP sockets
- Implement the forwarder.- repository for mapping names to physical addresses-desitination Id, port no.

sendmsg(dest, marshal(the mesg))

- Implement the receiver – blocking and non blocking

recvmsg() unmarshal(the msg)

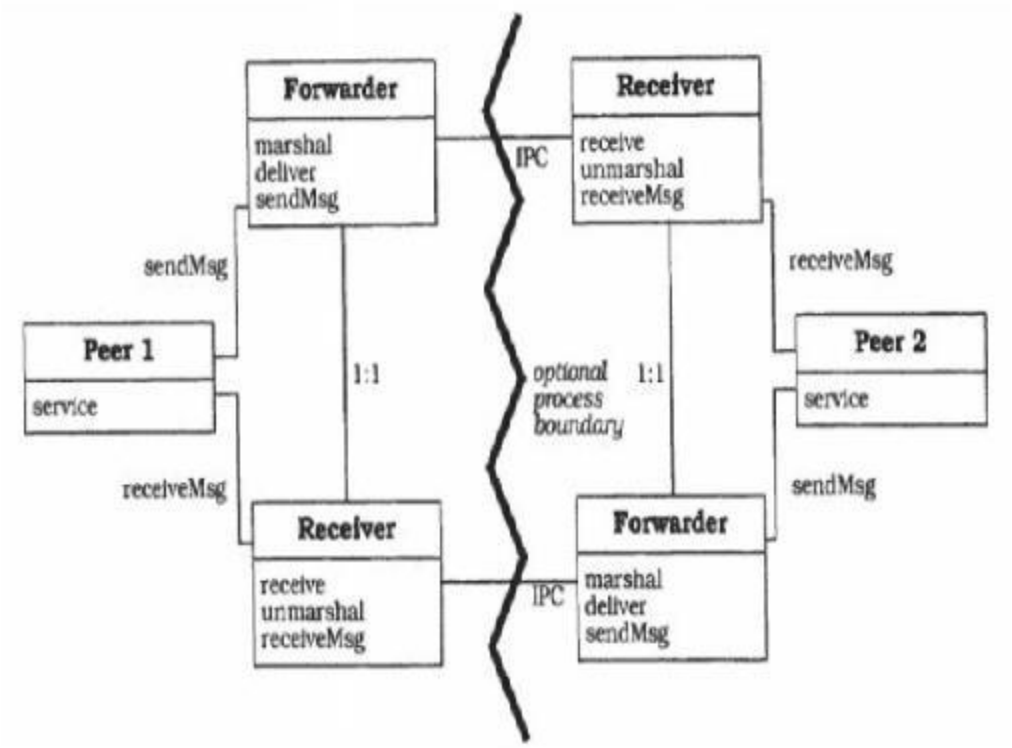
- Implement the peers of the application – partitioning into client and servers.
- Implement a start up configuration- initialize F-R with valid name to address mapping

Benefits and liability

- Efficient inter-process communication
- Encapsulation of IPC facilities

- No support for flexible re-configuration of components.
- **Known Uses**
- This pattern has been used on the following systems: TASC, a software development toolkit for factory automation systems, supports the implementation of Forwarder-Receiver structures within distributed applications.

- Part of the REBOOT project uses Forwarder-Receiver structures to facilitate an efficient IPC in the material flow control software for flexible manufacturing.
- ATM-P implements the IPC between statically-distributed components using the Forwarder-Receiver pattern..)
- In the Smalltalk environment BrouHaHa, the Forwarder-Receiver pattern is used to implement interprocess communication.

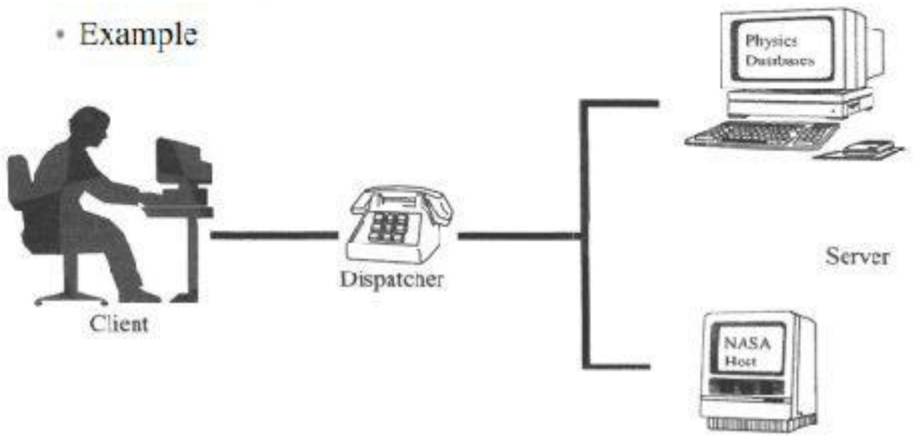


Client-Dispatcher-Server

- **Goals**
  - **Introduce an intermediate layer between clients and servers : the dispatcher**
  - **Provide location transparency**
  - **Hides details of establishment of communication**
- **Applicability**
  - **A software system integrating a set of distributed servers, with the servers running locally or distributed over a network.**

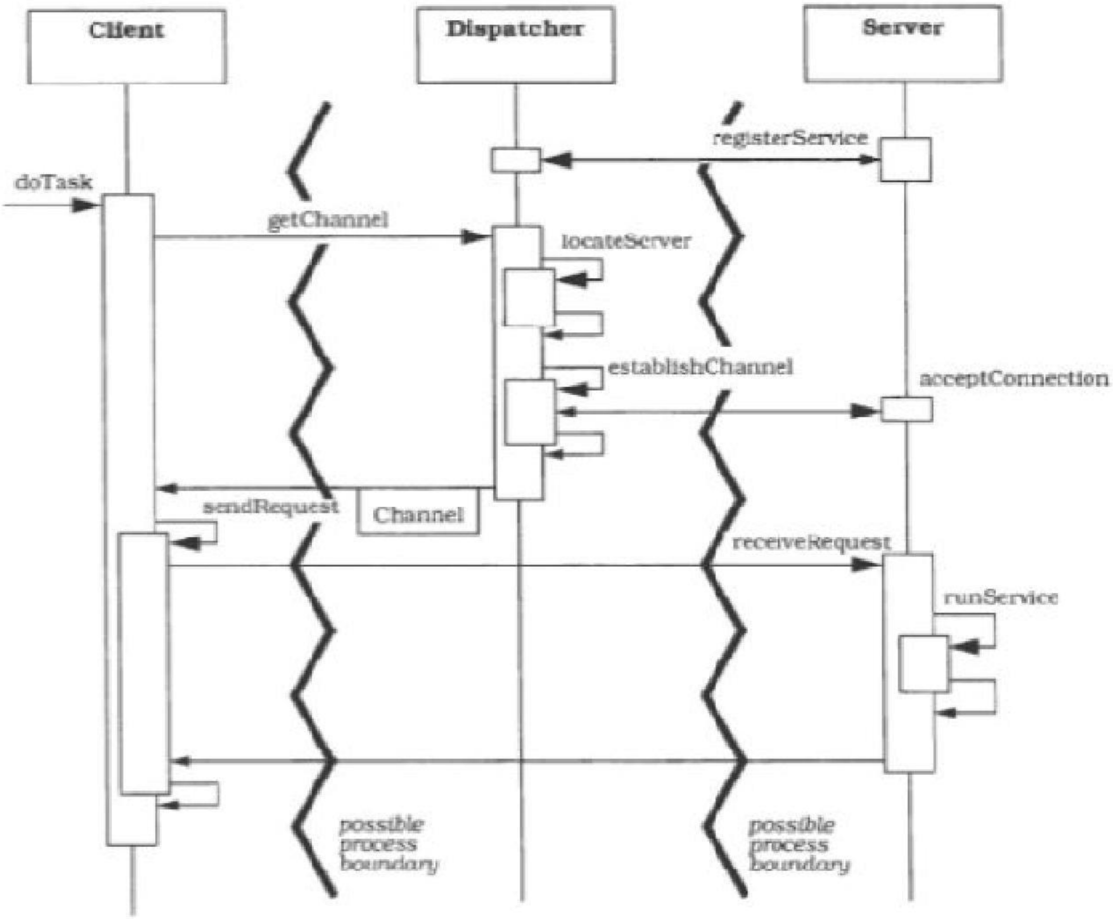
Client-Dispatcher-Server

- Example

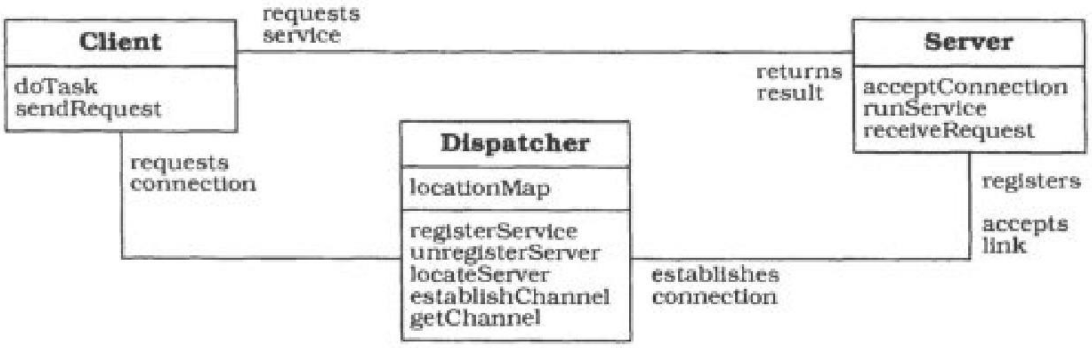


- **Components**
  - **Client**
    - **Performs some domain-specific tasks**
    - **Accesses operations offered by servers**
      - **Ask the dispatcher for a communication channel**
      - **Send its request to the server by this channel**
  - **Server**
    - **Provides services to clients**
    - **Registers itself with the dispatcher**
  - **Dispatcher**
    - **Establishes communications channels**
    - **Locates servers**
    - **(Un-)Registers servers**
    - **Maintains a map of server locations and name**

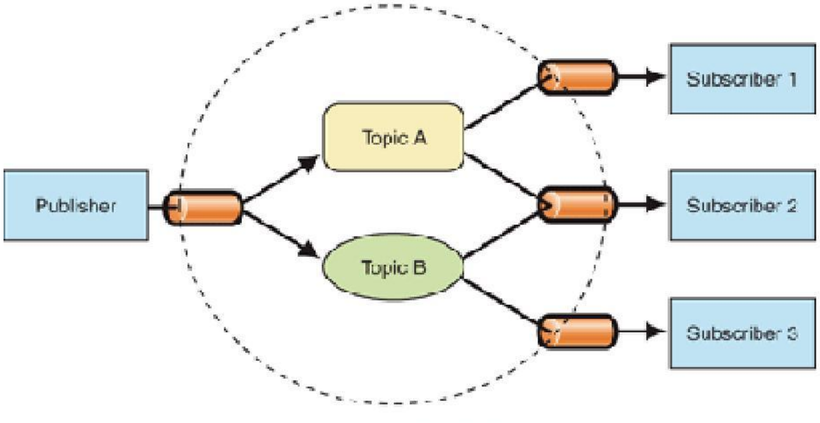
Interaction protocol



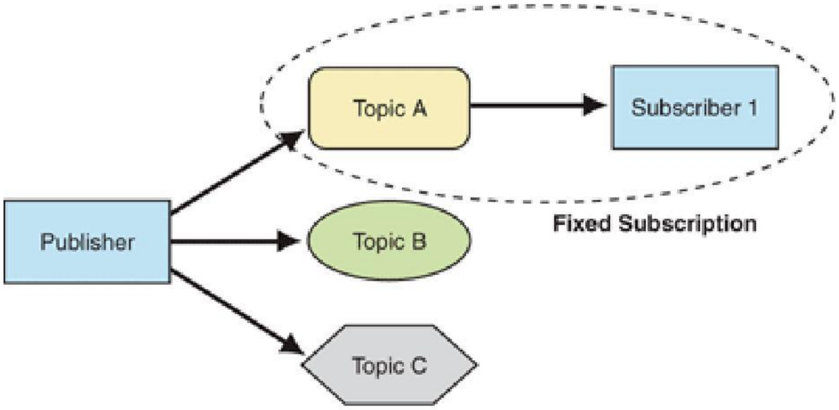
- Component structure and inter-relationships

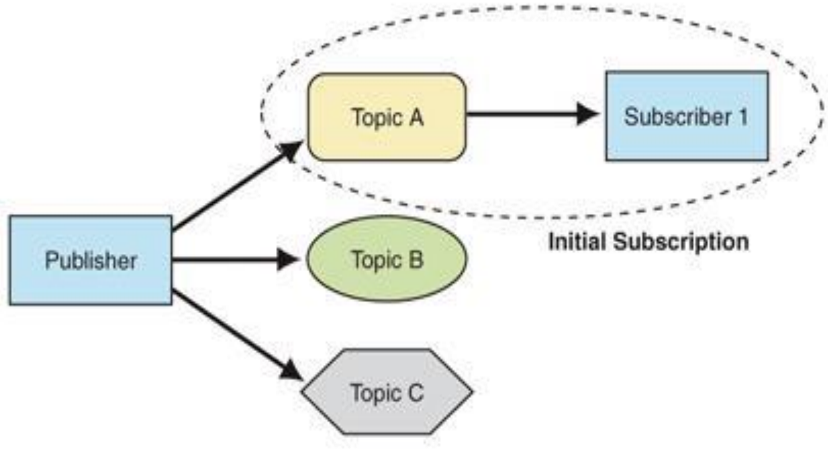


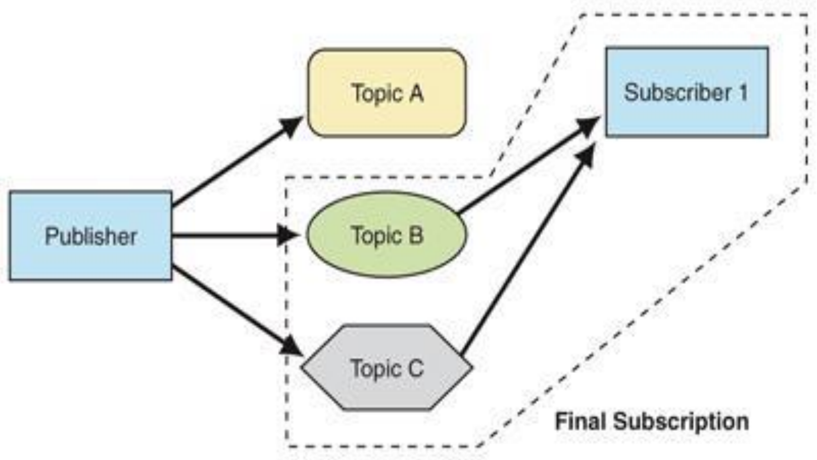
Publisher-Subscriber



Communication Infrastructure







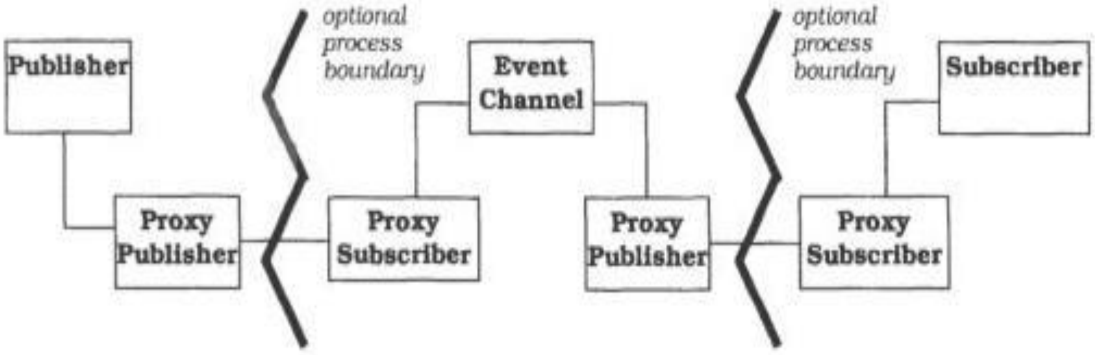
Publisher-Subscriber

- Goal
  - Help to keep the state of cooperation components synchronized
  - One publisher notifies any number of subscribers about changes to its state
- Applicability
  - Applications in which data changes in one pl. but many other components depend on this data
  - Number and identities of dependant components may changeover time
- Example : graphical user interf.s

Components

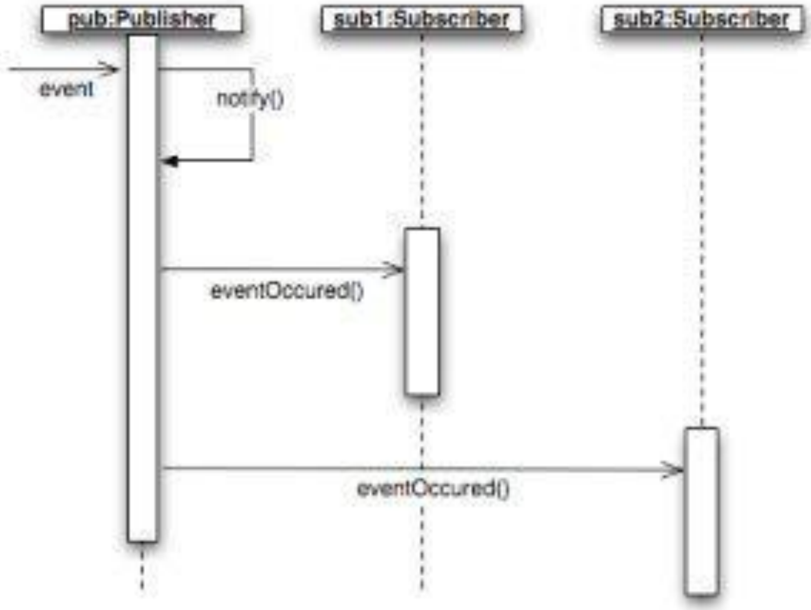
- Publisher
  - Maintains registry of currently-subscribed components
  - Sends notification to subscribers when its state has changed
- Subscriber
  - Can use the (un)subscribe interf. of the publisher
  - Retrieve changed data from publisher

- **Push model**
  - Publisher sends all changed data when it notifies the subscriber
  - Rigid dynamic behavior
  - Poor choice for complex data changes
  - Useful when subscribers need published information most of the time
- **Pull model**
  - Publisher only sends minimal information when sending a change notification
  - Subscribers are responsible for retrieving the data they need
  - Offers more flexibility but higher number of messages between publisher and subscriber
  - Useful when only individual subscribers can decide if and when they need a specific piece of information
- **Strengths**
  - Loosely-coupled
  - Publishers are loosely coupled to subscribers
  - Scalable in small installations
- **Weaknesses**
  - Not so scalable in large installations
  - Publisher assumes that subscriber is listening
- **Variants**
  - **Gatekeeper**
    - Publisher notifies remote subscribers
  - **Event Channel**
    - Strongly decouples publishers and subscribers
    - Possible to have more than one publisher
    - Subscribers only wish to be notified about changes, don’t care in which component changes occurred
    - Publishers are not interested in which components are subscribing
    - Event channel created and pl.d between publishers and subscribers
    - Appears as a subscriber to publishers
    - Appears as a publisher to subscribers
    - Event channel, subscriber and publisher can be in different processes
    - Can use buffers, can be chained (Unix pipes)



- **Variants**
  - Use of Producer-Consumer style of cooperation
    - Producer supplies information, consumer accepts it
    - Strongly decoupled thanks to a buffer
    - Only synchronization is for buffer under/overflow
    - Event-Channel pattern can simulate a P-C with more than one producer or consumer
- **Known uses**
  - **Java Swing, GUIs**

- Interaction protocol



Unit 8: DESIGN PATTERNS-2

SYLLABUS: --------- 6 hr

- **Management Patterns**
  - ○ **Command processor**
  - ○ **View handler**

Idioms

- **Introduction**
- **What can idioms**

provide? ➢ **Idioms and style**

- **Where to find idioms**

Counted pointer example

Design Patterns Management

Systems must often handle collections of objects of similar kinds, of service, or even complex components.

E.g.1 Incoming events from users or other systems, which must be interpreted and scheduled approximately.

e.g.2 When interactive systems must present application-specific data in a variety of different way, such views must be handled approximately, both individually and collectively.

- In well-structured s/w systems, separate manager components are used to handle such homogeneous collections of objects.

For this two design patterns are described

- The Command processor pattern
- The View Handler pattern

Command Processor

- The command processor design pattern separates the request for a service from its execution. A command processor component manages requests as separate objects, schedules their execution, and provides additional services such as the storing of request objects for later undo.

Context:

Applications that need flexible and extensible user interf.s or Applications that provides services related to the execution of user functions, such as scheduling or undo.

Problem:

- Application needs a large set of features.
- Need a solution that is well-structured for mapping its interf. to its internal functionality
- Need to implement pop-up menus, keyboard shortcuts, or external control of application via a scripting language
- We need to balance the following forces:
- Different users like to work with an application in different ways.
- Enhancement of the application should not break existing code.

- ▪ Additional services such as undo should be implemented consistently for all requests.

Solution:

• Use the command processor pattern •
Encapsulate requests into objects

- Whenever user calls a specific function of the application, the request is turned into a command object.
- The central component of our pattern description, the command processor component takes care of all command objects.
- It schedules the execution of commands, may store them for later undo and may provide other additional services such as logging the sequences of commands for testing purposes.

Example : Multiple undo operations in Photosho

Structure:

- Command processor pattern consists of following components:
  - o The abstract command component
  - o A command component
  - o The controller component
  - o The command processor component
  - o The supplier component

Components

- **Abstract command Component:**
- Defines a uniform interf. of all commands objects
- At least has a procedure to execute a command
- May have other procedures for additional services as undo, logging,…

Class Abstract Command	***Collaborators***
Responsibility • Defines a uniform interf. Interf. to execute commands • Extends the interf. for services of the command processor such as undo and logging	

- **A Command component:**
- For each user function we derive a command component from the abstract command.
- Implements interf. of abstract command by using zero or more supplier components.
- Encapsulates a function request
- Uses suppliers to perform requests
- E.g. undo in text editor : save text + cursor position

Class	***Collaborators***
Command	▪ Supplier
Responsibility • Encapsulates a function request • Implements interf. of abstract command • Uses suppliers to perform requests	

- **The Controller Component:**
- Represents the interf. to the application
- Accepts service requests (e.g. bold text, paste text) and creates the corresponding command objects
- The command objects are then delivered to the command processor for execution

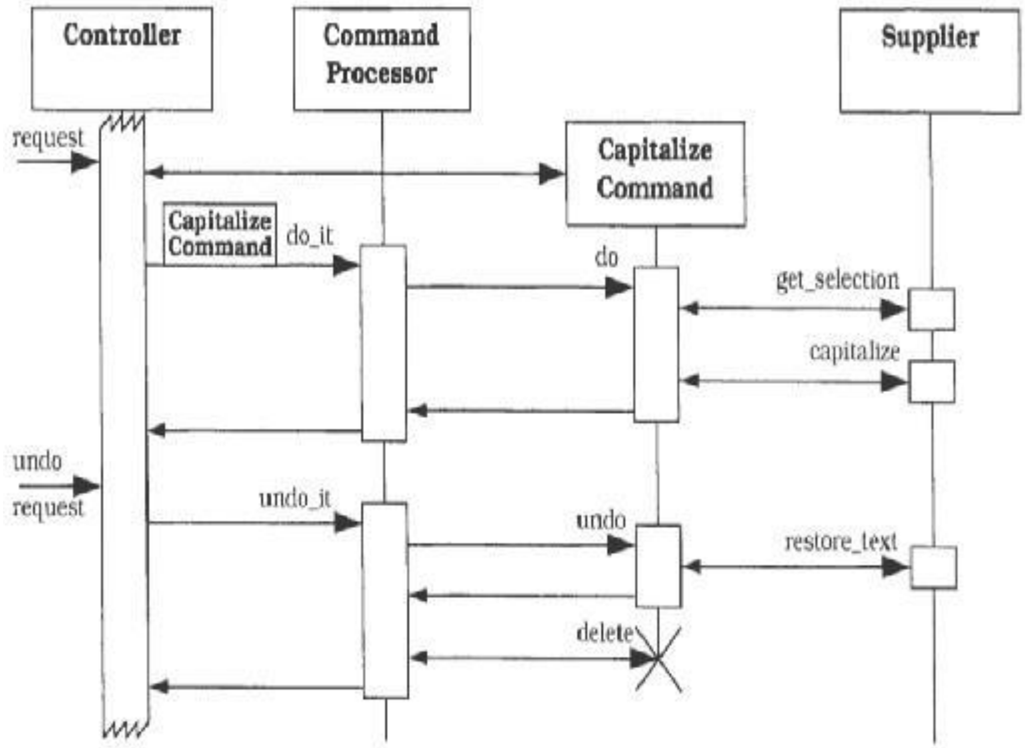
Class	***Collaborators***
Controller	▪ Command Processor ▪ Command
Responsibility • Accepts service requests • Translates requests into Commands • Transfer commands to command processor	

- **Command processor Component:**
- Manages command objects, schedule them and start their execution
- Key component that implements additional services (e.g. stores commands for later undo)
- Remains independent of specific commands (uses abstract command interf.)

Class	***Collaborators***
Command Processor	▪ Abstract Command
Responsibility • Activates command execution • Maintains command objects • Provides additional services related to command execution	

- **The Supplier Component:**
- Provides functionality required to execute concrete commands
- Related commands often share suppliers
- E.g. undo : supplier has to provide a means to save and restore its internal state

Class Supplier	***Collaborators***
Responsibility • Provides application specific functionality	



The following steps occur:

The controller accepts the request from the user within its event loop and creates a capitalize' command object.

The controller transfers the new command object to the command processor for execution and further handling.

The command processor activates the execution of the command and stores it for later undo.

The capitalize command retrieves the currently-selected text from its supplier, stores the text and its position in the document, and asks the supplier to actually capitalize the selection.

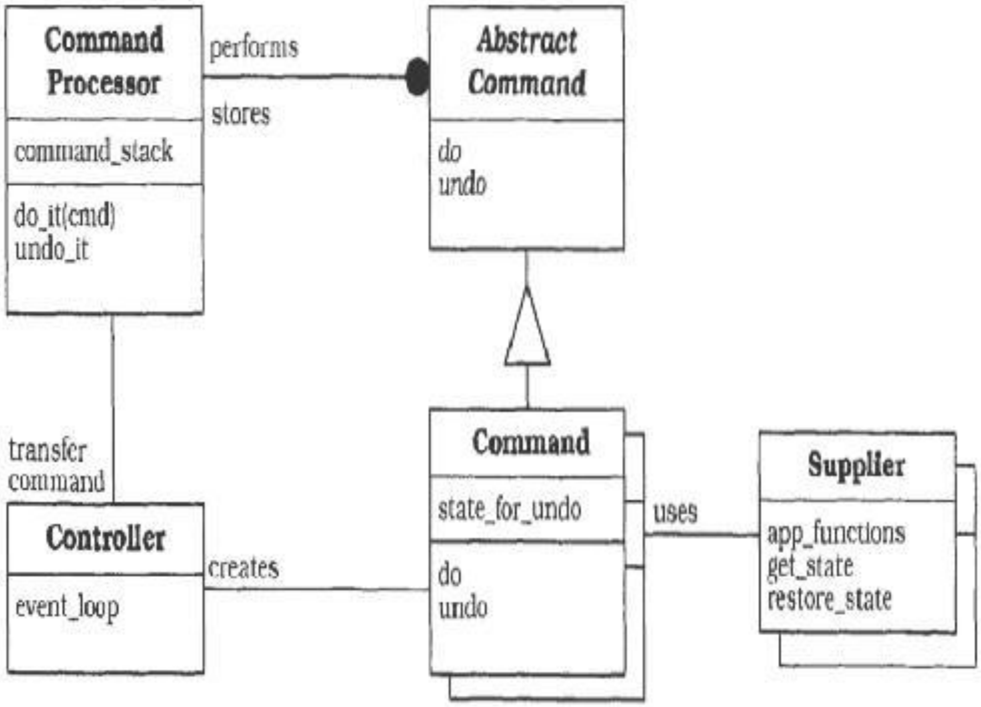
After accepting an undo request, the controller transfers this request to the command processor.

The command processor invokes the undo procedure of the most recent command.

The capitalize command resets the supplier to the previous state, by replacing the saved text in its original position

If no further activity is required or possible of the command, the command processor deletes the command object.

Component structure and inter-relationships



Strengths

▫ Flexibility in the way requests are activated

- Different requests can generate the same kind of command object (e.g. use GUI or keyboard shortcuts)

▫ Flexibility in the number and functionality of requests

- Controller and command processor implemented independently of functionality of individual commands

- ▪ Easy to change implementation of commands or to introduce new ones

▫ Programming execution-related services

- Command processor can easily add services like logging, scheduling,…

▫ Testability at application level

- Regression tests written in scripting language

▫ Concurrency

- Commands can be executed in separate threads
- Responsiveness improved but need for synchronization

Weaknesses

▫ Efficiency loss

▫ Potential for an excessive number of command classes

- Application with rich functionality may lead to many command classes
- Can be handled by grouping, unifying simple commands

▫ Complexity in acquiring command parameters

Variants

▫ Spread controller functionality

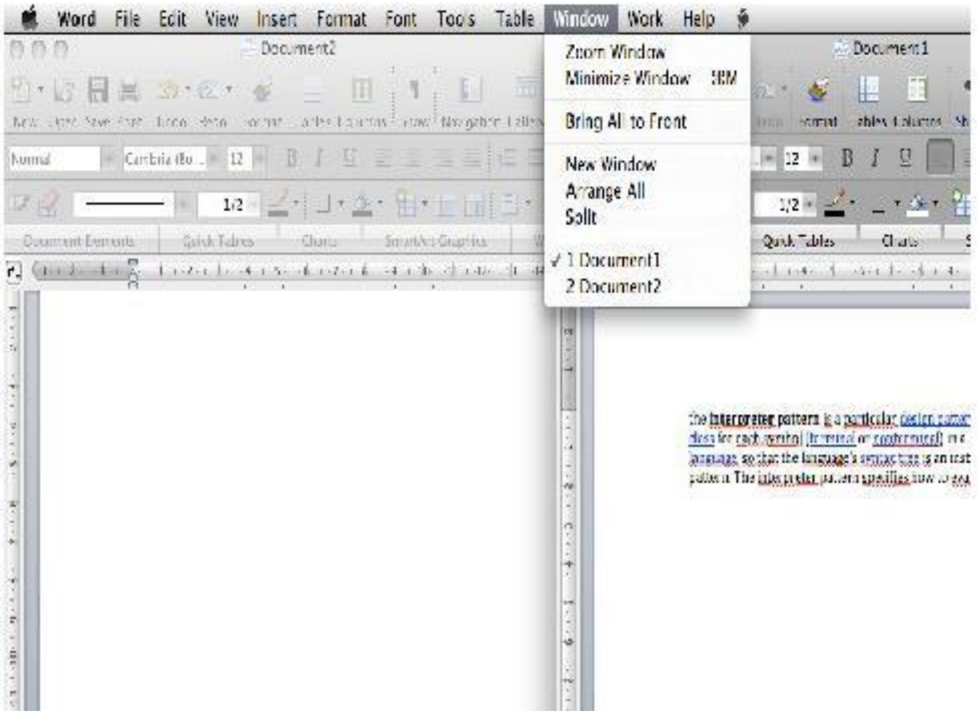
- Role of controller distributed over several components (e.g. each menu button creates a command object)

▫ Combination with Interpreter pattern

- Scripting language provides programmable interf.
- Parser component of script interpreter takes role of controller

View Handler

- **Goals**
- Help to manage all views that a software system provides
- Allow clients to open, manipulate and dispose of views
- Coordinate dependencies between views and organizes their update
- **Applicability**
- Software system that provides multiple views of application specific data, or that supports working with multiple documents
- Example : Windows handler in Microsoft Word



the Interpreter pattern is a particular design pattern
class for each symbol (terminal or nonterminal) in a
language, so that the language's syntax tree is an inst
pattern. The interpreter pattern specifies how to eva

View Handler and other patterns

• MVC

• View Handler pattern is a refinement of the relationship between the model and its associated views.

- PAC

• Implements the coordination of multiple views according to the principles of the View Handler pattern.

❖ **Components**

▫ View Handler

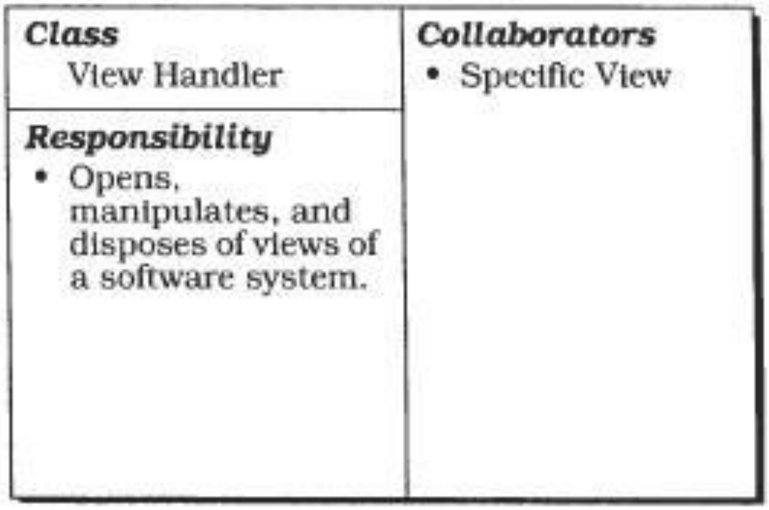
- Is responsible for opening new views, view initialization

- Offers functions for closing views, both individual ones and all currently-open views

- View Handlers patterns adapt the idea of separating presentation from functional core.

- The main responsibility is to Offers view management services (e.g. bring to foreground, tile all view, clone views)

- Coordinates views according to dependencies



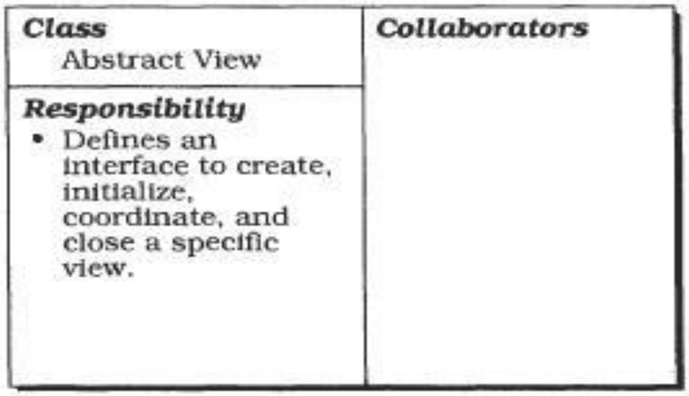
Class View Handler	*Collaborators* • Specific View
Responsibility • Opens, manipulates, and disposes of views of a software system.	

❖ **Components**

▫ Abstract view

- Defines common interf. for all views

- Used by the view handler : create, initialize, coordinate, close, etc.



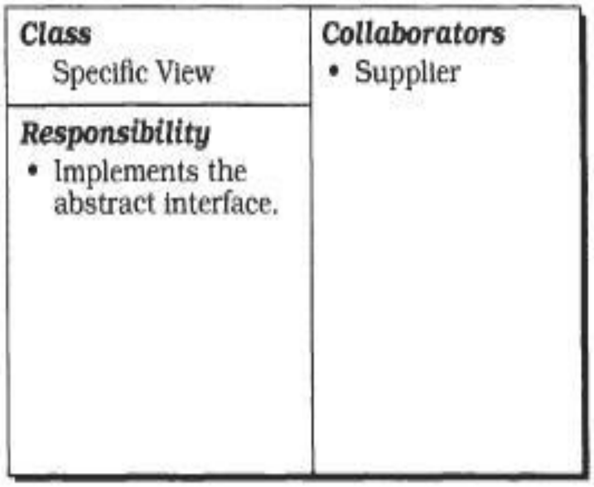
Class Abstract View	*Collaborators*
Responsibility • Defines an interface to create, initialize, coordinate, and close a specific view.	

❖ **Components**

- Specific view
- Implements Abstract view interf.

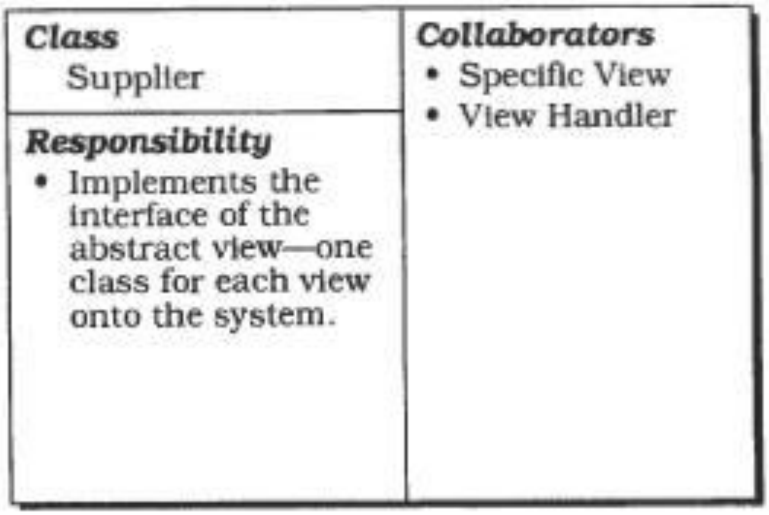
- Knows how to display itself
- Retrieves data from supplier(s) and change data
- Prepares data for display
- Presents them to the user

- Display function called when opening or updating a view



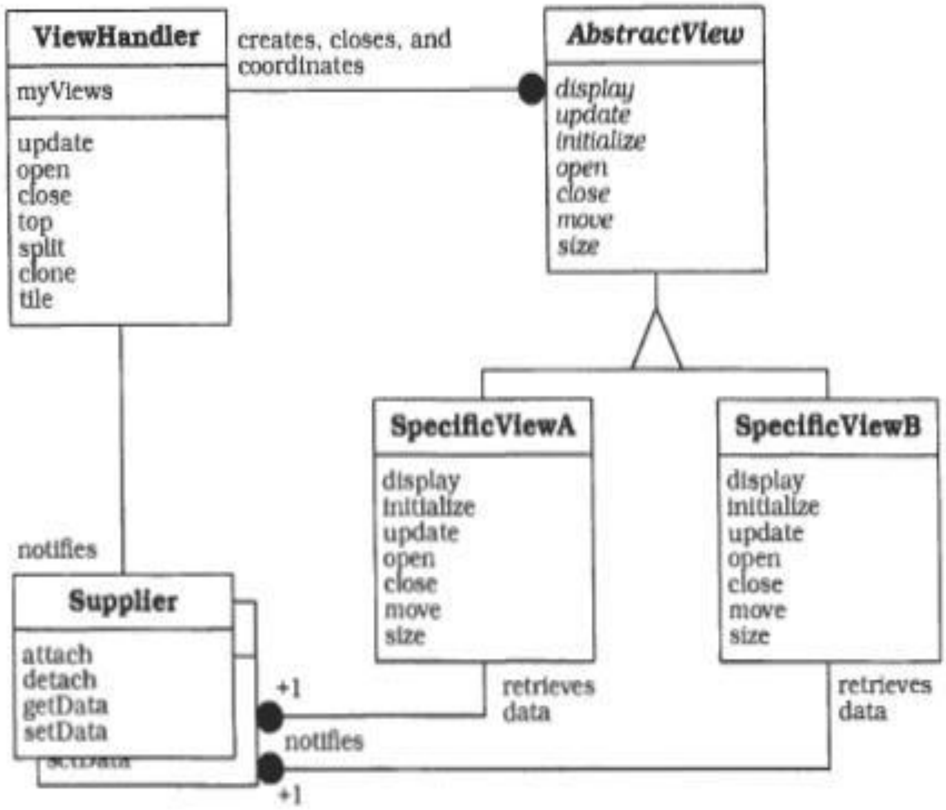
Class Specific View	**Collaborators** • Supplier
Responsibility • Implements the abstract interface.	

- ❖ **Components**
- • **Supplier**
- ▪ Provides the data that is displayed by the view components
- ▪ Offers interf. to retrieve or change data
- ▪ Notifies dependent component about changes in data



Class Supplier	**Collaborators** • Specific View • View Handler
Responsibility • Implements the interface of the abstract view—one class for each view onto the system.	

The OMT diagram that shows the structure of view handler pattern Component structure and inter-relationships



Two scenarios to illustrate the behavior of the View Handler

- View creation
- View tiling

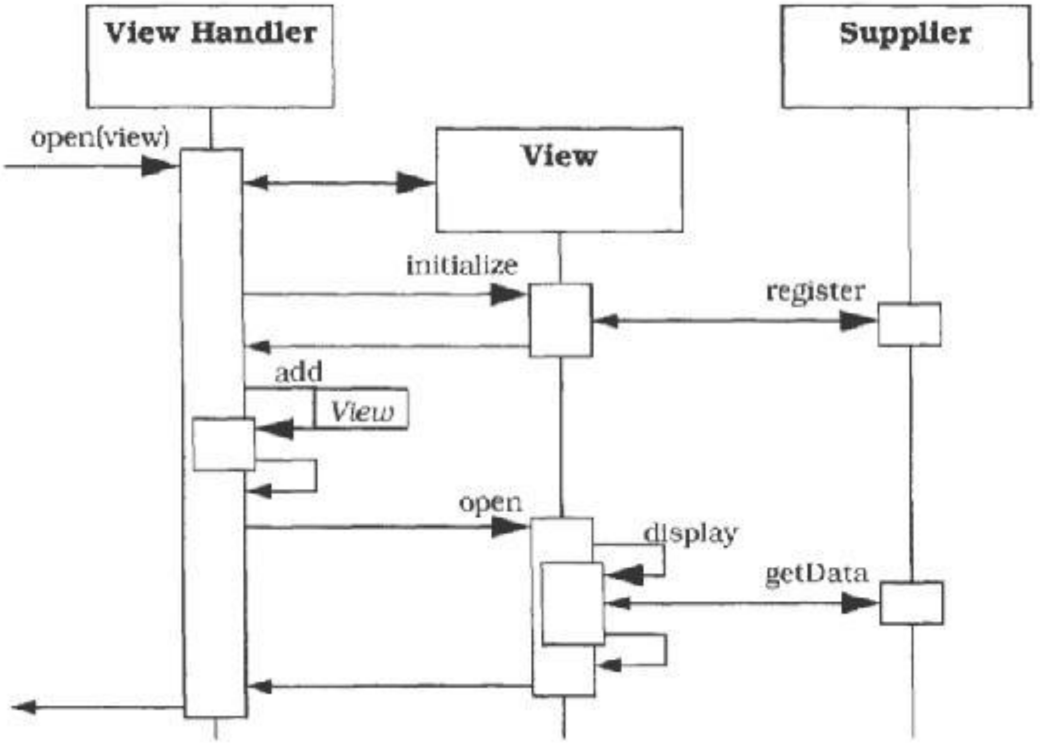
Both scenarios assume that each view is displayed in its own window.

Scenario I : View creation

Shows how the view handler creates a new view. The scenario comprises four phases:

- A client-which may be the user or another component of the system-calls the view handler to open a particular view.
- The view handler instantiates and initializes the desired view. The view registers with the change-propagation mechanism of its supplier, as specified by the Publisher-Subscriber pattern.
- The view handler adds the new view to its internal list of open views.
- The view handler calls the view to display itself. The view opens a new window, retrieves data from its supplier, prepares this data for display, and presents it to the user.

Interaction protocol

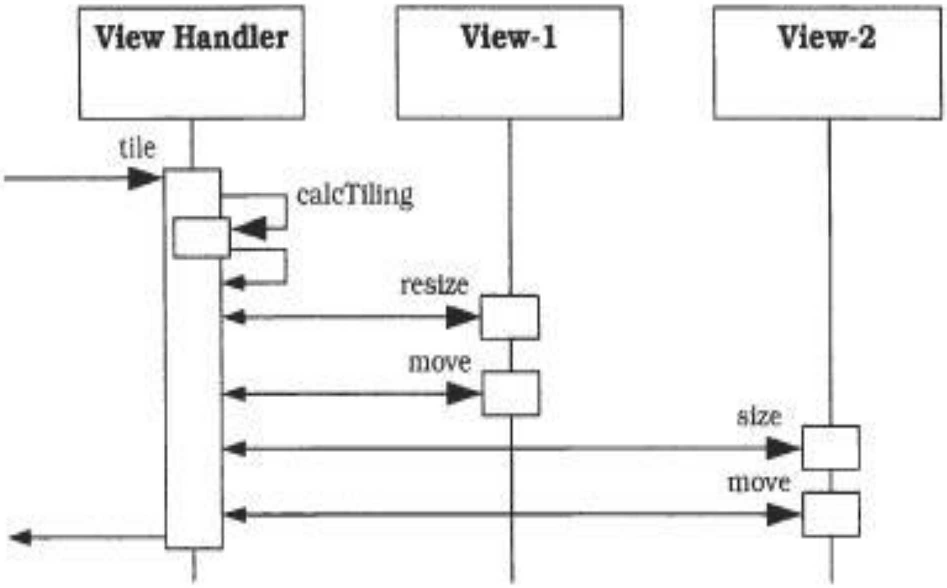


Scenario II : View Tiling

Illustrates how the view handler organizes the tiling of views. For simplicity, we assume that only two views are open. The scenario is divided into three phases:

- The user invokes the command to tile all open windows. The request is sent to the view handler.
- For every open view, the view handler calculates a new size and position, and calls its resize and move procedures.
- Each view changes its position and size, sets the corresponding clipping area, and refreshes the image it displays to the user. We assume that views cache the image they display. If this is not the case, views must retrieve data from their associated suppliers

Interaction protocol



Implementation

The implementation of a View Handler structure can be divided into four steps. We assume that the suppliers already exist, and include a suitable change-propagation mechanism.

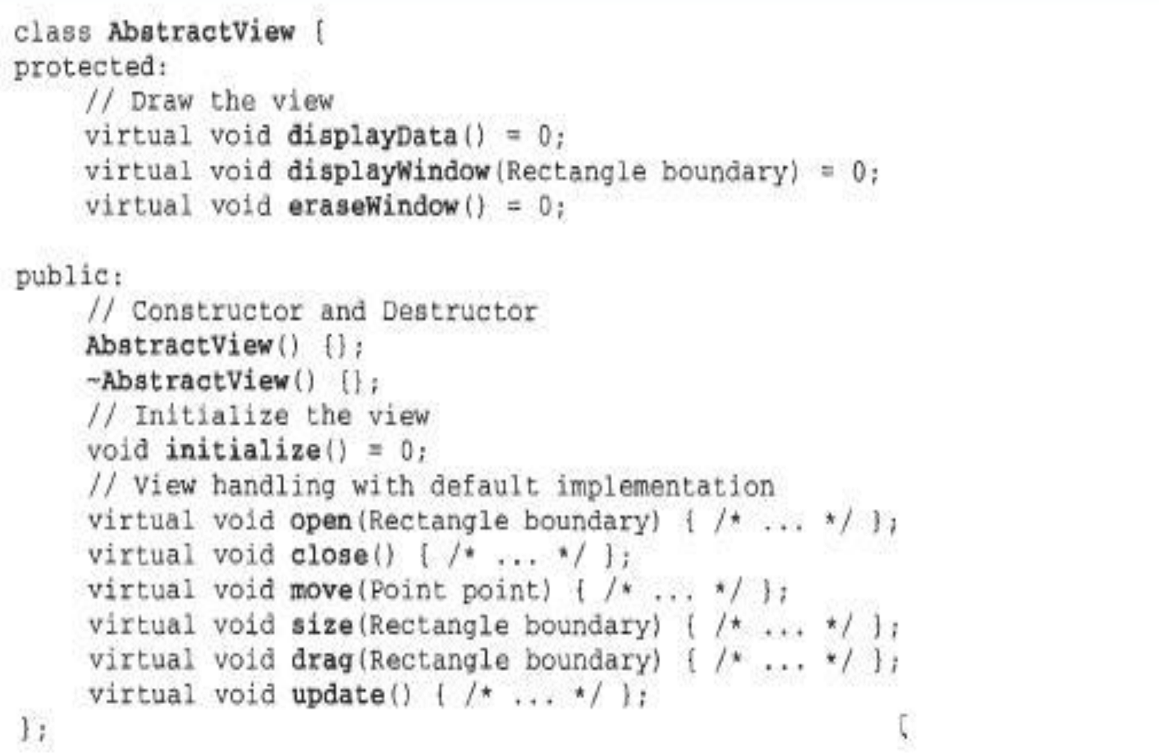
1. Identify the ***views.***
2. Specify *a common interf. for all* ***views.***
3. ***Implement the views.***
4. *Define the* ***view handler***

Identify the ***views.*** Specify the types of views to be provided and how the user controls each individual view.

Specify *a common interf. for all* ***views.*** This should include functions to open, close, display, update, and manipulate a view. The interf. may also offer a function to initialize a view.

The public interf. includes methods to open, close, move, size, drag, and update a view, as well as an initialization method.

Implementation



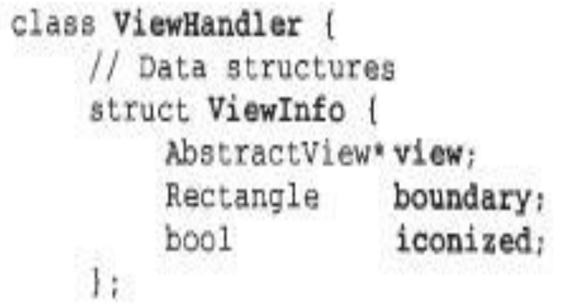
```
class AbstractView {
protected:
     // Draw the view
     virtual void displayData() = 0;
     virtual void displayWindow(Rectangle boundary) = 0;
     virtual void eraseWindow() = 0;

public:
     // Constructor and Destructor
     AbstractView() {};
     ~AbstractView() {};
     // Initialize the view
     void initialize() = 0;
     // View handling with default implementation
     virtual void open(Rectangle boundary) { /* ... */ };
     virtual void close() { /* ... */ };
     virtual void move(Point point) { /* ... */ };
     virtual void size(Rectangle boundary) { /* ... */ };
     virtual void drag(Rectangle boundary) { /* ... */ };
     virtual void update() { /* ... */ };
};
```

Implement the views. *Derive a separate class from the* ***AbtrsactView*** class for each specific type of view identified in step 1. Implement the view-specific parts of the interf., such as the **displayData *()*** method in our example. Override those methods whose default implementation does not meet the requirements of the specific view.

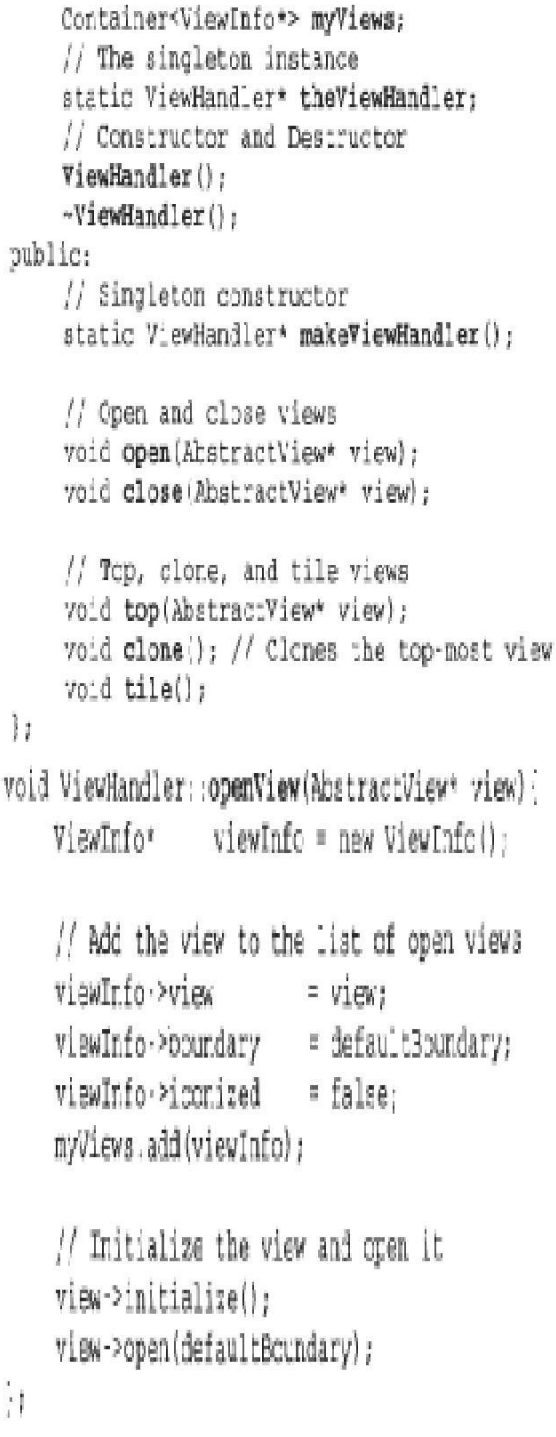
In our example we implement three view classes: *Editview*, *Layoutview,* and *Thumbnailview***,** as specified in the solution section.

Define the **view handler:** *Implement functions for creating views as* Factory Methods.

The view handler in our example document editor provides functions to open and close views, as well as to tile them, bring them to the foreground, and clone them. Internally the view handler maintains references to all open views, including information about their position and size, and whether they are iconize.



```
class ViewHandler {
     // Data structures
     struct ViewInfo {
          AbstractView* view;
          Rectangle     boundary;
          bool          iconized;
     };
```



```
     Container<ViewInfo*> myViews;
     // The singleton instance
     static ViewHandler* theViewHandler;
     // Constructor and Destructor
     ViewHandler();
     ~ViewHandler();
public:
     // Singleton constructor
     static ViewHandler* makeViewHandler();

     // Open and close views
     void open(AbstractView* view);
     void close(AbstractView* view);

     // Top, clone, and tile views
     void top(AbstractView* view);
     void clone(); // Clones the top-most view
     void tile();
};
void ViewHandler::openView(AbstractView* view){
    ViewInfo*     viewInfo = new ViewInfo();

    // Add the view to the list of open views
    viewInfo->view        = view;
    viewInfo->boundary    = defaultBoundary;
    viewInfo->iconized    = false;
    myViews.add(viewInfo);

    // Initialize the view and open it
    view->initialize();
    view->open(defaultBoundary);
};
```

Strengths

- ❑ Uniform handling of views
  - ▪ All views share a common interf.
- ▫ Extensibility and changeability of views
  - ▪ New views or changes in the implementation of one view don’t affect other component
- ▫ Application-specific view coordination

▪ Views are managed by a central instance, so it is easy to implement specific view coordination strategies (e.g. order in updating views)

Weaknesses

▫Efficiency loss (indirection)

▪ Negligible

▫ Restricted applicability : useful only with

▪ Many different views

▪ Views with logical dependencies

▪ Need of specific view coordination strategies

Variant

▫ View Handler with Command objects

▪ Uses command objects to keep the view handler independent of specific view interf.

▪ Instead of calling view functionality directly, the view handler creates an appropriate command and executes it

Known uses

▫ Macintosh Window Manager

▪ Window allocation, display, movement and sizing

▪ Low-level view handler : handles individual window

▫ Microsoft Word

▪ Window cloning, splitting, tiling…

Idioms

Introduction

➢ idioms are low-level patterns specific to a programming language

➢ An idiom describes how to implement particular aspects of components or the relationships between them with the features of the given language.

➢ Here idioms show how they can define a programming style, and show where you can find idioms.

➢ A programming style is characterized by the way language constructs are used to implement a solution, such as the kind of loop statements used, the naming of program elements, and even the formatting of the source code

What Can Idioms Provide?

- A single idiom might help you to solve a recurring problem with the programming language you normally use.

- They provide a vehicle for communication among software developers.(because each idiom has a unique name)

- idioms are less 'portable‘ between programming languages

Idioms and Style

If programmers who use different styles form a team, they should agree on a single coding style for their programs. For example, consider the following sections of C/C++ code, which both implement a string copy function for 'C-style' string

```
void strcopyRR(char *d, const char *s)
 { while (*d++=*s++) ; }

void strcopyPascal (char d [ I , const char s [I )
 { int i ;
   for (i = 0: s[il != ' \ O 1 : i = i + 1)
  { d[i] = s [i]; }
  d[i] = '\0' ; /* always asign 0 character */
}/* END of strcopyPasca1 */
```

Idioms and Style

A program that uses a mixture of both styles might be much harder to understand and maintain than a program that uses one style consistently.

Corporate style guides are one approach to achieving a consistent style throughout programs developed by teams.

Style guides that contain collected idioms work better. They not only give the rules, but also provide insight into the problems solved by a rule. They name the idioms and thus allow them to be communicated.

Idioms from conflicting styles do not mix well if applied carelessly to a program. Different sets of idioms may be appropriate for different domains. example, you can write C++ programs in an object-oriented style with inheritance and dynamic binding.

In real time system dynamic binding is not used which is required.

A single style guide can therefore be unsuitable for large companies that employ many teams to develop applications in different domains.

A coherent set of idioms leads to a consistent style in your programs.

Here is an example of a style guide idiom from Kent Beck's *Smalltalk Best Practice Patterns :*

Name : Indented Control Flow
Problem : How do you indent messages?
Solution : Put zero or one argument messages on the same lines as their receiver.

```
foo isNil
2 + 3
a < b ifTrue: [ . . . ]
```

Put the keyword/argument pairs of messages with two or more keywords each on its own line, indented one tab.

```
a < b
        ifTrue: [ . . . ]
        ifFalse: [ . . . I
```

- ➢ Different sets of idioms may be appropriate for different domains.

- ➢ For example, you can write C++ programs in an object-oriented style with inheritance and dynamic binding.

- ➢ In some domains. such as real-time systems, a more 'efficient' style that does not use dynamic binding is required.

- ➢ A single style guide can therefore be unsuitable for large companies that employ many teams to develop applications in different domains.

- ➢ A style guide cannot and should not cover a variety of styles.

Where Can You Find Idioms?

- ➢ Idioms that form several different coding styles in C++ can be found for example in Coplien's *Advanced C++ Barton and Neck man's Scientific and Engineering C++ and Meyers' Effective C++* .

- ➢ You can find a good collection of Smalltalk programming wisdom in the idioms presented in Kent Beck's columns in the *Smalltalk Report.*

- ➢ His collection of *Smalltalk Best Practice Patterns is about to be* published as a book .

- ➢ Beck defines a programming style with his coding patterns that is consistent with the Smalltalk class library, so you can treat this pattern collection as a Smalltalk style guide.